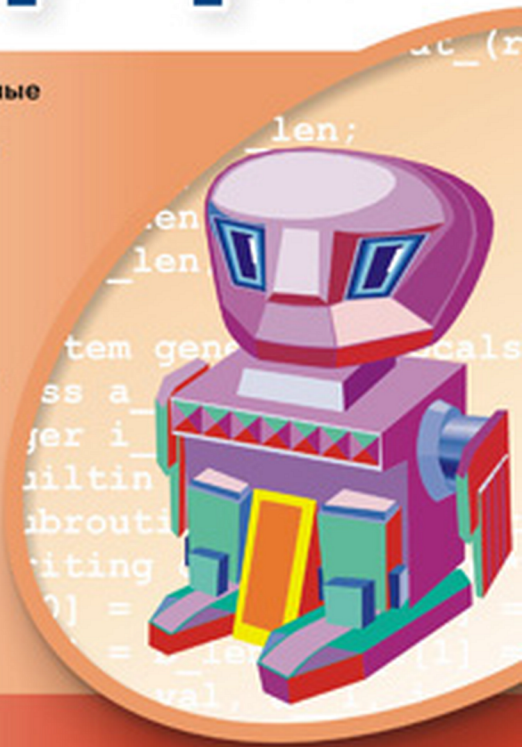
bhv

Сергей Немнюгин
Ольга Стесик

САМОУЧИТЕЛЬ

# Современный Фортран

- Синтаксис и основные алгоритмические конструкции языка
- Встроенные математические подпрограммы
- Операторы ввода/вывода и средства работы с файлами
- Компиляция, отладка и оптимизация программ
- Новый стандарт языка



*Эффективное средство решения вычислительных задач любой сложности*

Сергей Немнюгин
Ольга Стесик

# Современный Фортран

## САМОУЧИТЕЛЬ

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068+800.92фортран
ББК 32.973.26-018.1
Н50

**Немнюгин М. А., Стесик О. Л.**

Н50 Современный Фортран. Самоучитель. — СПб.: БХВ-Петербург, 2004. — 496 с.: ил.

ISBN 5-94157-302-2

Книга является пособием по изучению языка Фортран. Последовательно излагается синтаксис языка, рассматривается реализация основных алгоритмических конструкций. Особое внимание уделено встроенному математическому аппарату, средствам работы с массивами, операциям ввода/вывода. Изложение следует современным стандартам языка — Фортран 90/95. Приводится описание ряда наиболее распространенных компиляторов языка, вспомогательных средств разработки и отладки программ. Рассматриваются вопросы смешанного программирования на Фортране и C, реализация объектно-ориентированного подхода на Фортране. Дается описание одного из основных средств параллельного программирования — Высокопроизводительного Фортрана. Впервые на русском языке описывается новый стандарт языка — Фортран 2003. В приложениях приведена полезная справочная информация, которая не только поможет в повседневной практической работе, но и станет отправной точкой в дальнейшем знакомстве с одним из наиболее мощных языков вычислительного программирования.

*Для программистов*

УДК 681.3.068+800.92фортран
ББК 32.973.26-018.1

**Группа подготовки издания:**

| | |
|---|---|
| Главный редактор | *Екатерина Кондукова* |
| Зам. главного редактора | *Анатолий Адаменко* |
| Зав. редакцией | *Григорий Добин* |
| Редактор | *Леонид Мирошенков* |
| Компьютерная верстка | *Натальи Караваевой* |
| Корректор | *Виктория Пиотровская* |
| Дизайн обложки | *Игоря Цырульникова* |
| Зав. производством | *Николай Тверских* |

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.11.03.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 40.
Тираж 3 000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-302-2

© Немнюгин С. А., Стесик О. Л., 2004
© Оформление, издательство "БХВ-Петербург", 2004

# Содержание

**Предисловие ..... 1**

**Глава 1. Элементы языка Фортран ..... 5**

Принципы языков программирования. Договоримся о терминах ..... 5
Типы языков программирования ..... 7
Алфавит, лексемы, специальные слова языка Фортран ..... 9
Формат записи исходного текста программы ..... 11
Структура программы ..... 16
Типы данных ..... 19
Переменные ..... 20
Именованные и буквальные константы ..... 22
Массивы ..... 24
Производные типы данных ..... 24
Указатели ..... 25
Метки ..... 25
Комментарии ..... 26
Операторы ..... 26
Условный оператор *IF...THEN...END IF* ..... 29
Условный оператор *IF...THEN...ELSE...END IF* ..... 30
Оператор *SELECT* ..... 31
Оператор цикла со счетчиком *DO...END DO* ..... 33
Оператор цикла с предусловием *DO WHILE...END DO* ..... 34
Примеры программ ..... 35
Вопросы и задания для самостоятельной работы ..... 38

**Глава 2. Типы данных ..... 41**

Числовые типы ..... 41
Целый тип ..... 41
Вещественный тип ..... 44
Комплексный тип ..... 46

Нечисловые типы ....47
Логический тип данных ....47
Символьные данные ....48
Производные типы данных ....49
Массивы встроенных типов ....50
Указатели (ссылки) ....54
Динамические структуры данных ....55
Вопросы и задания для самостоятельной работы ....56

**Глава 3. Операторы описания ....57**

Неявное определение типа. Оператор *IMPLICIT* ....57
Структура оператора описания ....60
Определение производных типов ....62
Атрибуты ....64
Атрибут *PARAMETER*. Именованные константы и константные выражения ....64
Атрибуты *DIMENSION* и *ALLOCATABLE*. Описание массивов ....66
Атрибуты *POINTER* и *TARGET* ....67
Атрибуты *PUBLIC* и *PRIVATE* ....68
Атрибут *SAVE*. Сохранение значений локальных переменных ....69
Атрибуты *OPTIONAL* и *INTENT*. Описание формальных параметров ....70
Атрибуты *INTRINSIC* и *EXTERNAL* ....71
Назначение исходных значений переменных ....72
Вопросы и задания для самостоятельной работы ....73

**Глава 4. Операторы присваивания. Выражения ....75**

Определенные и неопределенные переменные ....76
Скалярные числовые выражения и скалярное числовое присваивание ....78
Скалярные операции отношения ....81
Скалярные логические выражения и присваивания ....83
Скалярные символьные выражения и присваивания ....84
Конструкторы структур и определение скалярных операций для переменных производного типа ....85
Выражения с массивами и присваивание массивов ....89
Указатели (ссылки) в выражениях и операторах присваивания ....92
Вопросы и задания для самостоятельной работы ....93

**Глава 5. Арифметика Фортрана ....95**

Арифметические выражения ....95
Задаваемые операции ....97
Порядок выполнения арифметических операций ....98
Арифметическое присваивание ....99
Преобразование типов ....100

Инициализация переменных ..... 107
Встроенные математические функции ..... 110
Точность вычислений ..... 115
Оптимизация вычислений ..... 128
Вопросы и задания для самостоятельной работы ..... 134

**Глава 6. Основные алгоритмические конструкции ..... 137**

Ветвления ..... 137
Оператор останова ..... 143
Оператор перехода ..... 144
Организация циклов ..... 145
Вложения управляющих конструкций и операторы перехода ..... 150
Вопросы и задания для самостоятельной работы ..... 152

**Глава 7. Структура программы. Подпрограммы и модули ..... 155**

Главная программа ..... 155
Внешние подпрограммы ..... 156
Внутренние подпрограммы ..... 158
Интерфейсы ..... 160
Параметры подпрограмм ..... 161
  Ограничения, накладываемые на фактические параметры ..... 162
  Вид связи формального параметра ..... 163
  Ссылки как формальные и фактические параметры ..... 164
  Параметры с атрибутом *TARGET* ..... 164
  Подпрограммы как параметры ..... 164
Именованные интерфейсы и совмещение процедур ..... 167
Модули ..... 169
Рекурсивные процедуры и функции ..... 171
Области видимости имен и меток ..... 172
Оператор *USE* ..... 173
Области общей памяти ..... 176
Порядок операторов ..... 179
Вопросы и задания для самостоятельной работы ..... 180

**Глава 8. Массивы ..... 181**

Основные сведения о массивах ..... 181
Массивы с подразумеваемой формой и автоматические объекты ..... 182
Элементные операции и присваивания ..... 184
Конструкторы массивов ..... 184
Подобъекты массивов ..... 186
Обращения к элементам и подобъектам массивов ..... 187
Массивы ссылок ..... 189

Массивы-маски ... 189
Оператор и конструкция *WHERE* ... 190
Оператор и конструкция *FORALL* ... 191
Динамические массивы ... 192
Операторы *ALLOCATE*, *DEALLOCATE* и *NULLIFY* ... 193
Вопросы и задания для самостоятельной работы ... 194

**Глава 9. Ввод и вывод ... 197**

Файлы ... 197
Операторы передачи данных ... 203
Форматирование ввода-вывода ... 222
Соединение файла с логическим устройством ... 230
Операторы управления файловым указателем ... 234
Оператор *INQUIRE* ... 235
Оператор *NAMELIST* ... 236
Повышение производительности ввода-вывода ... 238
Вопросы и задания для самостоятельной работы ... 240

**Глава 10. Встроенные функции и процедуры ... 243**

Оператор *INTRINSIC* ... 243
Ключевые параметры встроенных функций и процедур ... 244
Справочные функции ... 244
  Справочные функции для любого типа ... 244
  Числовые справочные функции ... 245
  Справочная функция для строк ... 247
  Справочные функции для массивов ... 247
  Функции для поиска в массиве ... 248
Процедуры определения даты и времени ... 249
Элементные функции ... 250
  Числовые функции для действий над действительными числами ... 250
  Математические функции ... 251
Операции над массивами ... 255
  Функции для создания массивов и операций над ними ... 255
Текстовые функции ... 259
Процедуры для работы с битами ... 261
Вопросы и задания для самостоятельной работы ... 263

**Глава 11. Компиляция, выполнение и отладка программ на Фортране ... 265**

Основные этапы создания исполняемого файла программы ... 266
Язык и система программирования F ... 267
Компилятор фирмы Intel ... 273
  Этапы компиляции ... 274
  Ключи компилятора ... 275

Запуск компилятора ... 282
Работа с проектом ... 285
Настройка среды компиляции ... 285
Препроцессор ... 288
Компиляция программ с модулями ... 288
Преобразование из формата Little-endian в формат Big-endian ... 289
Выходные файлы ... 290
Ключи отладки ... 290
Оптимизация ... 291
Управление точностью операций с плавающей точкой ... 292
Другие виды оптимизации ... 292
Распараллеливание и векторизация ... 293
Смешанное программирование на языках C и Фортран ... 293
Ограничения IFC7.0 ... 294
Утилита nmake ... 294
Компилятор GNU Fortran ... 295
Основные ключи G77 ... 296
Особенности диалекта Фортрана GNU ... 302
Отладка ... 303
Система программирования Compaq Visual Fortran ... 304
Проекты CVF ... 304
Обработка ошибок ... 305
Ограничения компилятора ... 306
Компилятор Lahey/Fujitsu Fortran 95 ... 306
Компиляторы Pro Fortran фирмы Absoft ... 307
Компилятор FortranPlus ... 309
Отладчик dbx ... 309
Точки останова ... 311
Отладчик gdb ... 315
Вопросы и задания для самостоятельной работы ... 315

**Глава 12. Фортран и объектно-ориентированное программирование ... 317**

Объектно-ориентированное программирование ... 318
Классы и объекты ... 319
Инкапсуляция ... 321
Наследование ... 322
Полиморфизм ... 324
Объектная структура программы ... 324
Реализация основных принципов ООП в программах на языке Фортран 90 ... 324
Инкапсуляция данных с помощью массивов ... 325
Перегрузка функций ... 330
Производные типы, классы и объекты ... 331
Наследование ... 336

Производные типы с указателями ... 340
Динамическая диспетчеризация ... 342
Реализация классов ... 345
Множественное наследование ... 351
Вопросы и задания для самостоятельной работы ... 352

**Глава 13. Фортран и высокопроизводительные вычисления ... 353**

High Performance Fortran ... 353
Директивы распределения данных ... 354
OpenMP ... 365
Общая форма директив OpenMP и общие принципы их использования ... 366
Вопросы и задания для самостоятельной работы ... 383

**Глава 14. Фортран и другие языки программирования ... 385**

Фортран и Паскаль ... 385
Фортран и C ... 386
Фортран и C++ ... 388
Идентификаторы и специальные слова ... 388
Записи ... 388
Многомерные массивы ... 389
Неинициализированные переменные ... 389
Указатели ... 389
Висячие ссылки ... 389
Логические объекты и операции ... 389
Отношения ... 390
Арифметические операции ... 390
Подпрограммы ... 390
Рекурсия ... 390
Перегрузка процедур и операций ... 390
Исключения и обработка ошибок ... 391
Поддержка параллелизма ... 391
Программы и единицы трансляции ... 391
Некоторые причины более низкой эффективности программ на языке C++ ... 391
Конвертер f2c ... 392
Имена ... 392
Типы ... 392
Возвращаемые значения ... 393
Списки аргументов ... 394
Расширения Фортрана 77, поддерживаемые f2c ... 394
Примеры ... 397

Смешанное программирование на Фортране и C/C++ ..... 402
Вызов подпрограмм Фортрана из программы на C++ ..... 404
Вызов подпрограмм C++ из программы на Фортране ..... 405

**Глава 15. Фортран 2003 ..... 409**

Новое в Фортране 2003 ..... 410
Типы ..... 410
Разное ..... 418
Ввод-вывод ..... 426
Фортран и C ..... 430
Удаленные и устаревшие элементы языка ..... 436

**Приложение 1. Перечень операторов языка ..... 439**

Неисполняемые операторы ..... 439
Исполняемые операторы ..... 443

**Приложение 2. Перечень встроенных подпрограмм ..... 447**

**Приложение 3. Библиотеки численных методов ..... 457**

BLAS ..... 457
ATLAS ..... 457
LAPACK ..... 458
ScaLAPACK ..... 458
IMSL ..... 459
Intel MKL ..... 459
NAG ..... 459

**Приложение 4. Ресурсы Интернета, посвященные Фортрану ..... 463**

Информация о компиляторах ..... 463
Информация о стандартах ..... 464
Информация общего характера ..... 464
Информация о вспомогательных программах и утилитах ..... 466
Информация о библиотеках ..... 467

**Список литературы ..... 469**

**Предметный указатель ..... 471**

*Памяти Станислава Петровича Меркурьева*

# Предисловие

Фортран занимает почетное место среди современных языков программирования. Это один из первых языков программирования высокого уровня и с самого своего рождения он предназначался для решения сложных вычислительных задач. В среде прикладных программистов Фортран сначала был встречен скептически, поскольку считалось, что заплатить за удобство программирования на языке высокого уровня придется значительной потерей скорости вычислений. Если речь идет о моделировании сложных процессов или обработке больших объемов информации, скорость вычислений является решающим фактором, определяющим выбор языка, вычислительной платформы и технологии программирования.

Разработчикам Фортрана удалось найти компромисс между удобством программирования и эффективностью программ, написанных на этом языке. Синтаксис языка строился таким образом, чтобы обеспечить максимальную эффективность автоматической оптимизации исполняемого кода. Это позволило в дальнейшем создавать оптимизирующие компиляторы, поставившие вычислительные возможности программ на Фортране вне всякой конкуренции. Язык был оснащен богатым набором встроенных математических функций и функций ввода-вывода, что существенно упрощает процесс программирования вычислительных задач.

Несмотря на свой "почтенный" возраст, Фортран постоянно обновляется. В среднем один раз в 10 лет выходит новый стандарт языка, учитывающий современное состояние программирования с одной стороны и пожелания программистов-прикладников с другой. Один раз в 5 лет выпускается стандарт, который включает относительно небольшие дополнения и изменения. Строгая стандартизация и постоянное обновление позволяют защитить инвестиции в прикладное программное обеспечение и сделать его универсальным. Разработкой стандартов занимается специальный комитет, который собирает и обобщает предложения, а затем выпускает серию проектов стандарта, доступных для всеобщего обсуждения. Трудно назвать какой-либо

другой язык, который сочетал бы постоянное обновление и достаточно строгое следование стандартам.

Темп обновления Фортрана может показаться медленным, однако это избавляет программиста-прикладника от необходимости постоянно знакомиться с новыми версиями языка и дает возможность сосредоточиться на решении задач из своей предметной области. Фортран впитывает те достижения Computer Science, которые действительно необходимы и полезны при программировании вычислений и не сказываются сколько-нибудь заметным образом на их скорости. В этом суть философии Фортрана.

Оптимизирующие компиляторы для Фортрана были включены в список десяти наиболее выдающихся достижений Computer Science XX века, опубликованный IEEE.

Удачная "биография" языка во многом была определена личностью человека, который руководил его разработкой. Это — Джон Бэкус. Джон Бэкус родился 3 декабря 1924 года в США. В 1942 году он окончил школу, но к учебе, по его собственному признанию, относился несерьезно. По окончании школы Джон по совету отца поступил в университет Вирджинии и приступил к изучению химии, но в 1943 году бросил учебу и ушел в армию. В армии Бэкус был направлен на обучение в медицинский госпиталь со специализацией в области нейрохирургии, но учеба не пошла и там. Бэкус считал, что она сводилась к зубрежке, а места размышлениям в ней не было, и через девять месяцев на карьере медика был поставлен крест.

Бэкус переехал в Нью-Йорк. Размышляя о смысле жизни, Джон пришел к неожиданному выводу, что для полного комфорта ему необходим аппарат для высококачественного воспроизведения музыки. Таких аппаратов в то время не было, Бэкус решил взять судьбу в собственные руки и отправился в школу радиотехников. Здесь он встретил первого в своей жизни учителя, который вызвал у него доверие. Учитель попросил Джона помочь ему с расчетом характеристик радиосхем. Сравнительно простой расчет усилителя, в общем нудный и громоздкий, вызвал у Бэкуса интерес к математике.

Бэкус поступил в Колумбийский университет Нью-Йорка и стал изучать математику. Университет Джон закончил в 1949 году. Незадолго до окончания университета он посетил вычислительный центр фирмы IBM, куда вскоре и был принят на работу программистом. Это произошло в 1950 году. Именно в IBM Бэкус возглавил работы по разработке первого языка высокого уровня Фортран (FORTRAN, от FORmula TRANslator — "переводчик формул" на машинный язык). Первая коммерческая версия языка была выпущена в 1957 году.

Вспоминая работу над проектом, Бэкус писал, что разработчики поначалу не знали точно, чего они хотят добиться. Они бурно обсуждали структуру языка и методы синтаксического анализа выражений. Результатом этой работы стал замечательный язык программирования, вскоре завоевавший популярность.

Бэкус известен в истории программирования не только как разработчик Фортрана. В 1959 году он предложил систему обозначений для описания синтаксиса языков программирования высокого уровня. Она называется формой Бэкуса — Наура (BNF).

Таким образом, первую скрипку в процессе работы над Фортраном, языком, предназначенным для программирования вычислений, играл человек с хорошим математическим образованием и опытом численных расчетов в области физики. Как знать, может быть картина современного программирования оказалась бы иной, если бы безвестный учитель математики не попросил Бэкуса помочь ему провести расчеты радиоусилителя!

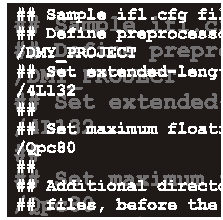
В книге, которую вы, уважаемый читатель, держите в своих руках, содержится описание Фортрана согласно стандартам, которые обычно называют Фортран 90 и Фортран 95. Иногда, и мы постарались это особо оговаривать, речь идет о Фортране 77, который все еще достаточно распространен в среде прикладных программистов и научно-технических работников.

Мы старались избегать ориентации на конкретные компиляторы или среды программирования. Книга содержит примеры-листинги с исходными текстами, иллюстрирующими материал. Мы надеемся, что это облегчит процесс самостоятельного изучения языка. Настоятельно рекомендуем выполнять задания для самостоятельной работы и отвечать на вопросы, которыми завершается большая часть глав книги.

Одна из глав посвящена вопросам взаимодействия между языками Фортран и C/C++. Дается также описание основных особенностей Высокопроизводительного Фортрана и Фортрана 2003. Фортран 2003 — это очередное значительное обновление языка, которое будет официально принято в качестве стандарта в ближайшем будущем.

Мы надеемся, что наш скромный труд принесет пользу программистам, приступающим к изучению Фортрана или использующим его в своей повседневной работе. В последнем случае книгу можно использовать в качестве справочника и сборника полезных примеров. Авторы будут благодарны за замечания и предложения по содержанию книги, которые можно направлять по адресу электронной почты:

**club1982@yandex.ru**




# Глава 1

# Элементы языка Фортран

В этой главе мы познакомимся с основными элементами языка Фортран — его алфавитом, специальными словами и символами; рассмотрим структуру программы. Здесь же мы узнаем о встроенных типах Фортрана, важнейших операторах, управляющих ходом выполнения программы, а также об операторах ввода/вывода. Это позволит приступить к написанию собственных, пока еще достаточно простых программ и поможет читать исходные тексты Фортрана. Завершается урок вопросами и упражнениями. В данной главе и далее в книге мы будем сопровождать описание конструкций языка и приемов программирования примерами программ — как относительно простых, так и более сложных. Многие темы данной главы более подробно рассматриваются в следующих главах.

## Принципы языков программирования. Договоримся о терминах

В различных языках программирования, несмотря на все различия между ними, иногда довольно значительные, можно найти много общих черт. Можно сказать, что в основу различных языков положены общие принципы, но реализация этих принципов может быть разной. Прежде всего, стандарт (спецификация) любого языка ограничивает набор символов, которые можно использовать в программах. Этот набор называется *алфавитом* языка. Символы алфавита являются элементами более сложных образований — *лексем*. Лексемы играют роль слов — минимальных смысловых элементов языка. Так и в обычном языке буквы алфавита используются для построения слов, каждое из которых несет определенную смысловую нагрузку. Лексемами языка программирования 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 построения имен. В правилах присвоения имен переменным (и другим объектам) следует обратить внимание на то, какие символы можно использовать в именах, какова максимальная длина имени и различается ли регистр букв.

*Специальные слова* используются для построения предложений — базовых единиц языка, обладающих смысловой завершенностью. Специальные слова бывают двух видов: зарезервированные и ключевые. *Зарезервированные слова* имеют вполне определенные смысл и назначение, которые определяются спецификацией или стандартом языка. Любая неточность в применении зарезервированных слов является серьезной ошибкой. Назначение *ключевых слов* зависит от контекста, то есть от того, где в программе слово используется.

Любая программа содержит переменные, константы, предложения описания и операторы. *Переменная* связана с областью памяти компьютера, которой присвоено определенное имя и которая наделена определенными свойствами. Иногда говорят, что переменной соответствует одна абстрактная ячейка памяти. Физический размер этой ячейки зависит от типа переменной, то есть от того, какая информация в ней хранится.

Переменная характеризуется набором атрибутов. *Имя переменной* является одним из основных ее атрибутов. Следующий атрибут — *значение переменной*. Содержимое ячейки памяти, связанной с переменной, может изменяться в ходе выполнения программы (собственно, поэтому переменная и называется переменной!). *Тип переменной* определяет вид информации, содержащейся в абстрактной ячейке памяти, а также набор операций и множество ее допустимых значений. Важнейшей характеристикой переменной является и ее *адрес*. В линейной модели памяти это порядковый номер первой физической ячейки памяти, входящей в состав абстрактной ячейки. *Область видимости переменной* — это такая часть программы, в которой операторы могут использовать данную переменную. Обычно говорят о *глобальных* и *локальных* переменных. Глобальные переменные доступны во всей программе, а локальные только в отдельных ее блоках.

*Константа* отличается от переменной прежде всего тем, что ее значение не изменяется в ходе выполнения программы. Принято различать *буквальные* и *именованные* константы. Буквальные константы представляют собой значения, которые воспринимаются в программе в точности так, как они изображены. Значения могут быть числовыми, символьными или другими. Буквальные константы еще называют литералами.

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 (атрибуты) переменных и других объектов, используемых в программе, называются *предложениями описания* или декларативными операторами. Операторы, управляющие ходом выполнения программы, называются *управляющими операторами.* Управляющие операторы, прежде всего, реализуют основные алгоритмические конструкции, такие как бинарные и многовариантные ветвления (условные операторы, операторы выбора) и циклы. Имеются операторы ввода и вывода информации и т. д.

*Программа* представляет собой последовательность операторов и других элементов языка, построенную в соответствии с правилами языка и предназначенную для решения определенной задачи. Правила написания программы, определяющие, какие последовательности символов можно использовать в программе, называются *синтаксисом* языка программирования.

# Типы языков программирования

Языки программирования используются для записи алгоритмов с целью их последующего выполнения на компьютере. Напомним читателю, что *алгоритм* представляет собой конечный набор действий, расположенных в определенном логическом порядке, позволяющий исполнителю решать любую конкретную задачу из некоторого класса однотипных задач. Исполнителем алгоритма может быть компьютер или другое устройство. Алгоритм для компьютера может быть составлен и записан с разной степенью подробности. Эта степень зависит от того, насколько детально учитывается архитектура компьютера, на котором предполагается выполнять программу. Программист пишет программу для некоторой абстрактной машины. *Абстрактная вычислительная машина* представляет собой упрощенную модель реального компьютера, в которой отсутствуют детали, не имеющие значения для программиста. Чем больше деталей устройства реального компьютера содержится в описании этой машины, тем ниже уровень абстракции. На одном из самых высоких уровней абстракции находится представление о компьютере как машине, состоящей из процессора, памяти и устройств ввода/вывода. Такую архитектуру называют фон-неймановской.

*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, достаточно длинной, следовательно, увеличивается вероятность появления ошибок. Для составления программы требуется предварительное изучение архитектуры определенного типа компьютеров, что увеличивает трудоемкость программирования. Программа оказывается привязанной к конкретной архитектуре. 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. Тексты программ при этом оказываются более короткими и универсальными (независимыми от архитектуры), их легче читать, в них проще разобраться, а время их разработки значительно сокращается. Однако объем занимаемой памяти и время выполнения таких программ значительно больше, чем у тех, что написаны на языках низкого уровня.

Языки программирования высокого уровня обычно делят на 4 класса.

1. Императивные (процедурные).
2. Функциональные.
3. Логические.
4. Объектно-ориентированные.

Основными объектами в *императивных* языках являются переменные, операторы присваивания, стандартные алгоритмические конструкции. Императивные языки программирования привязаны к традиционной фоннеймановской архитектуре.

В *функциональных* языках программирования используются функции, значения которых определяются по заданным параметрам. Традиционные переменные, операторы присваивания при этом уже не нужны или, по крайней мере, не обязательны.

В программах, написанных на *логических* языках, нет определенного, фиксированного порядка выполнения правил и шагов алгоритма, а выбор подходящей последовательности возлагается на систему.

*Объектно-ориентированные* языки упрощают программирование за счет использования технологии объектно-ориентированного программирования (ООП).

Фортран является языком программирования высокого уровня. Он относится к категории императивных языков и изначально создавался для программирования вычислений.

# Алфавит, лексемы, специальные слова языка Фортран

Алфавит языка Фортран включает 26 латинских букв:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z,

причем регистр букв в программах различается только для строковых констант. Есть также арабские цифры:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

символ подчеркивания и специальные символы.

Все специальные слова языка Фортран являются ключевыми. Они могут использоваться в составе предложений или операторов языка, но допускается и произвольное их применение. Последнее, впрочем, следует считать нежелательной практикой и плохим стилем программирования. Его следует избегать.

В табл. 1.1 приведены основные ключевые слова Фортрана.

***Таблица 1.1.*** *Основные ключевые слова языка Фортран*

| | | | |
|---|---|---|---|
| ADMIT | ALLOCATABLE | ALLOCATE | ASSIGN |
| ASSIGNMENT | ATEND | BACKSPACE | BLOCKDATA |
| CALL | CASE | CHARACTER | CLOSE |
| COMMON | COMPLEX | CONTAINS | CONTINUE |
| CYCLE | DATA | DEALLOCATE | DEFAULT |
| DIMENSION | DO | DOUBLE | ELSE |
| END | ENDFILE | END IF | END SELECT |
| ENTRY | EQUIVALENCE | EXIT | EXTERNAL |
| FORMAT | FUNCTION | GO TO | IF |
| IMPLICIT | INCLUDE | INQUIRE | INTEGER |
| INTENT | INTERFACE | INTRINSIC | LOGICAL |
| MAP | MODULE | NAMELIST | NONE |
| OPEN | OPTIONAL | PARAMETER | PAUSE |
| POINTER | PRINT | PRECISION | PROCEDURE |
| PROGRAM | READ | REAL | RECORD |
| RECURSIVE | RETURN | REWIND | SAVE |
| STOP | STRUCTURE | SUBROUTINE | TARGET |
| THEN | TYPE | UNION | USE |
| WHILE | WRITE | | |

Лексемами являются обозначения логических операций и логических констант, которые приведены в табл. 1.2. Запись отношений с помощью 4-символьных обозначений характерна для Фортрана 77 и более старых версий языка. Начиная с Фортрана 90, используются более удобные обозначения, приближенные к математической нотации (т. е. правилам записи).

***Таблица 1.2.*** *Логические операции и константы языка Фортран*

| Символ | Описание | Символ | Описание |
|---|---|---|---|
| .GT. | Отношение "больше" | .OR. | Логическая операция "ИЛИ" (логическое сложение) |
| .LT. | Отношение "меньше" | .AND. | Логическая операция "И" (логическое умножение) |
| .GE. | Отношение "больше или равно" | .NOT. | Логическое отрицание |
| .LE. | Отношение "меньше или равно" | .FALSE. | Логическая константа "ложь" |
| .NE. | Отношение "не равно" | .TRUE. | Логическая константа "истина" |
| .EQ. | Отношение "равно" | .EQV. | Отношение эквивалентности |
| .NEQV. | Отношение неэквивалентности | | |

В программах на языке Фортран используются как отдельные специальные символы, так и пары символов, которые имеют специальное значение. Перечень таких символов с описанием некоторых вариантов их применения приведен в табл. 1.3.

***Таблица 1.3.*** *Одиночные и двойные специальные символы языка Фортран*

| Символ | Описание | Символ | Описание |
|---|---|---|---|
| = | Оператор присваивания | . | Десятичная точка в буквальных числовых константах или элемент логической операции/константы |
| + | Арифметическая операция "сложение" | (/ /) | Ограничители в конструкторах массивов |
| - | Арифметическая операция "вычитание" | ' " | Ограничители строковой константы |

***Таблица 1.3*** *(окончание)*

| Символ | Описание | Символ | Описание |
|---|---|---|---|
| * | Арифметическая операция "умножение" | : | Разделитель между именем конструкции и первым ее ключевым словом, разделитель при указании диапазона значений индекса, разделитель для ветви ONLY оператора использования USE |
| / | Арифметическая операция "деление", ограничитель для имени COMMON-блока | ! | Начало комментария |
| (<br>) | Список параметров подпрограммы, индексы массива, циклы | | Пробел |
| , | Разделитель в списках | :: | Разделитель в предложениях описания |
| & | Признак переноса оператора на следующую строку | == | Отношение равенства |
| ; | Разделитель операторов в строке в свободном формате записи программы | => | Присваивание указателя |
| // | Объединение (конкатенация) строк | ** | Возведение в степень |
| > | Отношение "больше" | /= | Отношение неравенства |
| < | Отношение "меньше" | >= | Отношение "больше или равно" |
| % | Селектор компонента структуры | <= | Отношение "меньше или равно" |

# Формат записи исходного текста программы

Для записи исходного текста программы на Фортране могут использоваться *фиксированный* и *свободный форматы*. Первый из них характерен для стандарта Фортран 77 и более старых, являясь "наследством" перфокарточной эры программирования, а второй применяется в Фортране 90. Фортран 90 поддерживает также фиксированный формат, что обеспечивает совместимость со старыми стандартами записи.

При записи исходного текста в фиксированном формате строка имеет длину 72 позиции (см. рис. 1.1). Первые пять позиций отведены для меток, а шестая может быть пустой или содержать любой, отличный от пробела символ. В последнем случае строка считается строкой продолжения и при обработке компилятором присоединяется к предыдущей строке программы. Допускается не более 19 строк продолжения одного оператора. Оператор может занимать позиции с 7 по 72, а его длина не может превышать 1320 символов (с учетом пробелов). Пробел в исходном тексте программы игнорируется компилятором и используется только для улучшения читаемости программы.

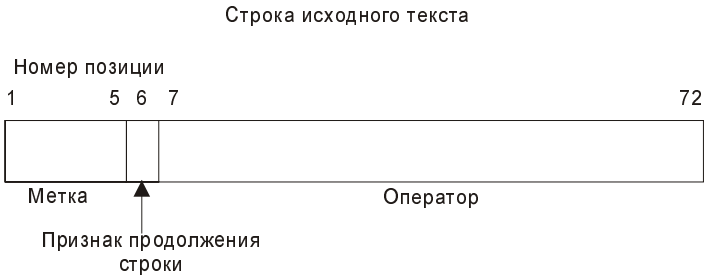



**Рис. 1.1.** Фиксированный формат записи исходного текста программ на Фортране

В свободном формате записи все позиции строки равноправны, ее длина составляет 132 символа, а максимальная длина оператора — 2640 символов, включая пробелы. При этом пробелы могут быть значимыми символами. Остановимся на правилах их использования в свободном формате.

Пробелы не должны появляться в лексемах, за исключением строковых значений. Например, не допускаются пробелы между звездочками в обозначении операции возведения в степень `**`. Пробелы используются в качестве разделителей между именами, константами или метками и соседними ключевыми словами, именами, константами и метками. В операторах:

```
REAL ELMASS
GO TO 100
DO I=1,10
```

пробелы требуются после слов `REAL`, `TO` и `DO`.

Некоторые ключевые слова обязательно должны отделяться друг от друга пробелами, другие этого не требуют. Например, `BLOCK DATA` можно записать и как `BLOCKDATA`. В табл. 1.4 приводятся ключевые слова, которые требуют использования пробелов и те, в которых пробел не обязателен.

***Таблица 1.4.*** *Использование пробелов в различных конструкциях языка Фортран*

| Пробелы не требуются | Пробелы требуются |
|---|---|
| BLOCK DATA | CASE DEFAULT |
| DOUBLE COMPLEX | DO WHILE |
| DOUBLE PRECISION | IMPLICIT спецификация типа |
| ELSE IF | IMPLICIT NONE |
| END BLOCK DATA | INTERFACE ASSIGNMENT |
| END DO | INTERFACE OPERATOR |
| END FILE | MODULE PROCEDURE |
| END FORALL | RECURSIVE FUNCTION |
| END FUNCTION | RECURSIVE SUBROUTINE |
| END IF | RECURSIVE тип FUNCTION |
| END INTERFACE | Тип FUNCTION |
| END MODULE | Тип RECURSIVE FUNCTION |
| END PROGRAM | |
| END SELECT | |
| END SUBROUTINE | |
| END TYPE | |
| END WHERE | |
| GO TO | |
| IN OUT | |
| SELECT CASE | |

Следующая конструкция, которая может появиться в программе при случайном вводе точки вместо запятой, будет восприниматься как ошибочная:

```
DO 3 I = 1. 3
```

В противном случае (при использовании фиксированного формата) результатом такой опечатки будет синтаксически правильный оператор присваивания значения 1.3 переменной `DO1I`, что является грубой логической ошибкой:

```
DO1I = 1.3
```

Приведем одну из легенд программирования. Считается, что подобная ошибка была допущена в управляющей программе, составленной для бортового компьютера первой американской станции, которую предполагалось отправить к Венере. Компилятор не смог обнаружить ошибку, в результате чего "потеря хвостика запятой" обернулась неудачей дорогостоящего проекта.

Примеры исходных текстов программ в фиксированном и свободном форматах приведены в листингах 1.1 и 1.2.

**Листинг 1.1. Пример записи исходного текста программы в фиксированном формате**

```
      SUBROUTINE CONVCASE(CIN, COUT, LEN, UPCASE)
      IMPLICIT NONE
      INTEGER :: LEN
      CHARACTER (LEN = *) :: CIN, COUT
      LOGICAL :: UPCASE
!
!  1) CONVERT A CHARACTER STRING TO ALL UPPER CASE OR ALL
!     LOWERCASE.
!
!  2) ARGUMENTS:
!
!     CIN    = INPUT CHARACTER STRING
!     COUT   = OUTPUT CHARACTER STRING IN CONVERTED CASE
!     LEN    = THE CHARACTER LENGTH OF CIN
!     UPCASE = CASE CONVERSION FLAG
!            = .TRUE.  FOR UPPER CASE
!            = .FALSE. FOR LOWER CASE
!
      INTEGER :: I, ILETTR, ILOWR0, IUPPR0
      IUPPR0 = IACHAR('A') - 1
      ILOWR0 = IACHAR('a') - 1
      COUT = ''
      IF(UPCASE) THEN
        DO I = 1, LEN
          ILETTR = IACHAR(CIN(I:I)) - ILOWR0
          IF(ILETTR .GE. 1 .AND. ILETTR .LE. 26) THEN
!           CONVERT TO UPPER CASE.
            COUT(I:I) = ACHAR(IUPPR0 + ILETTR)
          ELSE
            COUT(I:I) = CIN(I:I)
          END IF
        END DO
      ELSE
```

```
        DO I = 1, LEN
          ILETTR = IACHAR(CIN(I:I)) - IUPPR0
          IF     (ILETTR .GE. 1 .AND. ILETTR .LE. 26) THEN
!           CONVERT TO UPPER CASE.
            COUT(I:I) = ACHAR(ILOWR0 + ILETTR)
          ELSE
            COUT(I:I) = CIN(I:I)
          END IF
        END DO
      END IF
      END SUBROUTINE CONVCASE
```

**Листинг 1.2. Пример записи исходного текста программы в свободном формате**

```
SUBROUTINE CAL_GETENV(var, varval)
! Return as VARVAL the contents of the operating system
! environment variable VAR, if defined.
!
! INPUT:
!    VAR: name of environment variable            [A*]
!
! OUTPUT:
!    VARVAL: contents of enviroment variable
!
 USE MSFLIB
 CHARACTER(LEN = *) VAR, VARVAL
 INTEGER(4) :: lval, lenv
 INTENT(in) :: VAR
 INTENT(out) :: VARVAL
 Lenv = LEN_TRIM(var) !ФОРТРАН 90 intrinsic
 Lval = 0
 IF(lenv .GT. 0) THEN
       lval = GETENVQQ(var(:lenv), varval)
 END IF
 IF(lval .EQ. 0) VarVal = ' '
RETURN
END SUBROUTINE CAL_GETENV
```

Остановимся более подробно на особенностях записи текста в свободном формате. Мы уже знаем, что иногда оператор или другая конструкция

программы не умещается в одной строке. Вообще говоря, стоит избегать слишком длинных конструкций, поскольку они ухудшают читаемость программы. Если все-таки возникла необходимость в переносе строки, то в конце строки, в месте переноса, необходимо поставить знак & (амперсанд), 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 же, напротив, необходимо в одной строке разместить несколько операторов, они отделяются друг от друга символом "точка с запятой" (;), например:

```
A = 2.2; B = 4; C = 7.8
```

Операторы программы следует располагать таким образом, чтобы обозначить и 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. Компьютерные программы содержат большое количество концентрированной информации, и хорошее форматирование значительно упрощает их чтение. Следует иметь в виду, что на отладку сложной программы может уйти значительное время, иногда больше половины времени ее разработки. Должное внимание к хорошему форматированию, тщательный и обоснованный выбор имен переменных, подпрограмм и других объектов программы, полное документирование программы увеличивает эффективность труда разработчика. При наборе исходного текста не следует размещать в одной строке более одного-двух операторов. Пустые строки можно использовать для выделения логически связанных групп операторов.

# Структура программы

Программа на Фортране состоит из *главной программы* и, возможно, некоторого числа *подпрограмм*. Подпрограммы могут быть *функциями* или *процедурами*, внешними, внутренними или модульными. Главную программу и подпрограммы будем называть *программными компонентами*. Различные программные компоненты могут компилироваться раздельно. Это несомненное достоинство языка, которое позволяет вносить изменения в от-

дельные части программы, не выполняя полной перекомпиляции кода. Компиляторы Фортрана обычно тщательно и неторопливо выполняют свою работу — создание исполняемого файла при компиляции большой программы может потребовать значительного времени. В таких случаях большим удобством является то, что после редактирования одной отдельно взятой подпрограммы можно перекомпилировать только ее. Полученный в результате такой "выборочной" компиляции объектный файл используется при окончательной "сборке" программы.

Первым оператором главной программы является ее заголовок `PROGRAM`. За ним следует идентификатор — имя программы:

```
PROGRAM ИМЯ_ПРОГРАММЫ
```

Имя программы обязательно начинается с буквы, затем могут идти буквы, цифры и символы подчеркивания, например:

```
PROGRAM SUMMATION
PROGRAM QUADRATIC_EQUATION_SOLVER
PROGRAM MERCEDES600
```

Максимальная длина имени программы — 31 символ. Это правило распространяется на любые имена в программах на Фортране.

Заголовок программы может быть опущен. Имя программы не должно совпадать с именами переменных или других ее объектов и 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`, который является исполняемым оператором. Заключительный оператор главной программы может иметь также следующий вид:

```
END PROGRAM ИМЯ_ПРОГРАММЫ
```

`ИМЯ_ПРОГРАММЫ` является необязательной частью оператора, как, впрочем, и `PROGRAM`.

После заголовка обычно следуют описания переменных, констант, меток, подпрограмм и других объектов, используемых в программе. Эта ее часть называется *разделом описаний*. В простых программах указанный раздел может быть пустым, т. е. не содержать никаких описаний.

**Внимание**

Простейшая программа на языке Фортран состоит... из единственного оператора `END`! Такую программу можно откомпилировать, компилятор создаст исполняемый файл, только вот единственное действие, выполняемое такой программой, — немедленное завершение работы.

После раздела описаний следует часть, которая выполняет какие-либо действия. Она называется *разделом операторов*. Структура программы изображена на рис. 1.2.

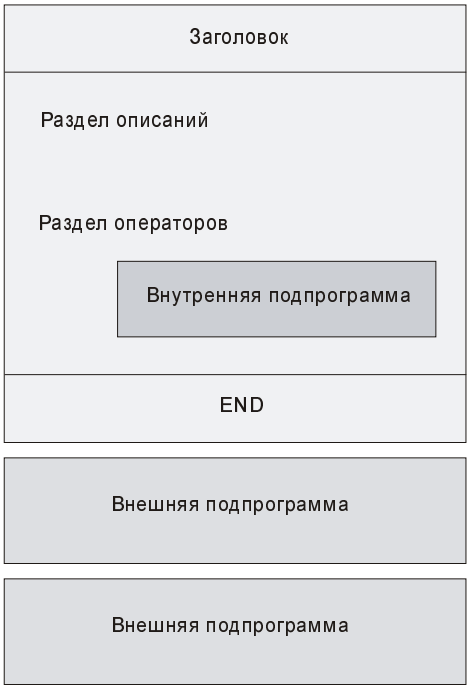



**Рис. 1.2.** Структура программы на языке Фортран

Раздел описаний состоит из предложений описания переменных, именованных констант, `COMMON`-блоков и некоторых других. Предложений описания переменных может быть несколько, размещаются они между заголовком программного компонента и его разделом операторов. Располагаться предложения описания переменных могут вместе (это — одна из составных частей хорошего стиля программирования), но могут и чередоваться с описаниями других объектов. Предложения описаний должны находиться до предложений `DATA` и исполняемых операторов.

В разделе описаний операторы `IMPLICIT` должны располагаться до всех прочих предложений описания кроме предложений `PARAMETER`. Обычно строка `IMPLICIT` является второй строкой.

Любое предложение описания, которое определяет тип имени именованной константы, должно располагаться до предложения `PARAMETER`. Если в предложении `PARAMETER` используются другие именованные константы, соответствующие им предложения `PARAMETER` должны находиться раньше.

# Типы данных

Количество базовых (встроенных или предопределенных) типов Фортрана невелико, и в этом отношении он уступает другим языкам программирования высокого уровня. Есть возможность определять производные типы.

Встроенными являются числовые типы, логический тип и строковый (символьный):

- INTEGER;
- REAL;
- COMPLEX;
- LOGICAL;
- CHARACTER.

Здесь INTEGER — это специальное (ключевое) слово, обозначающее числовой тип, допустимыми значениями которого являются целые числа. Особенностью целого типа является абсолютно точное представление числового значения.

Числовой тип REAL представляет вещественные значения. Переменные типа REAL не могут принимать значения, сколь угодно близкие к нулю и слишком большие по абсолютной величине. Вещественный тип используется для хранения (приближенных) дробных значений. 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, фактически, множество упорядоченных пар вещественных значений, первое из которых является вещественной частью комплексного числа, а второе — его мнимой частью. Комплексное значение занимает в памяти в два раза больше места, чем вещественное. Наличие в Фортране комплексного типа — большой подарок программистам-вычислителям, которым достаточно часто приходится пользоваться комплексной арифметикой!

Диапазон допустимых значений числовых типов в Фортране не стандартизирован, но программист может самостоятельно его определить. Механизм подобных определений основан на концепции разновидности типа. У каждого встроенного типа Фортрана есть несколько разновидностей, которые отличаются друг от друга диапазоном значений и физическим размером абстрактной ячейки памяти, соответствующей данному типу. В дальнейшем мы познакомимся со способами определения множеств допустимых значений различных типов.

Множество допустимых значений логического типа LOGICAL состоит из двух значений: "истина" и "ложь", которым соответствуют логические константы

.TRUE. и .FALSE. Логический тип и логические операции являются реализацией булевой (двузначной) логики.

Значение символьного типа CHARACTER представляет собой строку символов. Длина строкового значения в Фортране может быть произвольной, она задается с помощью параметра LEN в предложении описания строковой переменной:

```
CHARACTER(LEN = 430) :: Shakespeare_sonet
```

Пробел является значимым символом в символьном значении. Длина символьного значения в байтах равна длине строки. Каждому символу в строке соответствует номер. Номера присваиваются последовательно, начиная с 1 для самого левого символа.

# Переменные

При работе с переменными важнейшим является вопрос о том, как сопоставить переменной набор атрибутов и, прежде всего, как определить тип переменной. В Фортране используются два механизма определения типа. Первый из них — *определение типа по имени*. Если первым символом имени переменной или функции являются буквы I, J, K, L, M или N, то по умолчанию предполагается, что переменная имеет тип INTEGER, а если любая другая буква или символ подчеркивания, то подразумевается тип REAL. Этот механизм называется *неявным связыванием имени переменной с ее типом*.

Предложение IMPLICIT позволяет модифицировать (изменить) правила неявного определения типов, а предложение IMPLICIT NONE отменяет действие правила объявления типа по первой букве имени. Отказ от этого, весьма "демократичного" правила гарантирует, что любое имя, отсутствующее в предложениях описания, будет обнаружено компилятором. Таким образом, могут быть исключены ошибки, связанные с появлением "ложных" переменных, образовавшихся в программе в результате опечаток при наборе текста. Авторы настоятельно рекомендуют использовать IMPLICIT NONE!

Второй способ определения типов переменных состоит в использовании предложений описания типов. Этот механизм называют *явным связыванием имен переменных и их типов*.

Предложение описания переменных в Фортране 90 имеет вид:

```
ТИП :: СПИСОК_ПЕРЕМЕННЫХ
```

или

```
ТИП, АТРИБУТЫ :: СПИСОК_ПЕРЕМЕННЫХ
```

В списке переменных имена разделяются запятыми, а ТИП задает общий тип переменных из данного списка, являясь идентификатором типа. Рассмотрим пример описания переменных:

```
INTEGER :: cows, sheeps
REAL, PARAMETER :: salary = 2000
```

Здесь INTEGER, REAL — названия типов, cows и sheeps — имена переменных, а PARAMETER — атрибут объекта salary, определяющий, что он является именованной константой.

В Фортране 90 предложения описания стали более информативными и компактными, чем в Фортране 77, у которого в предложении описания отсутствует разделитель "двойное двоеточие" и нет списка атрибутов, например:

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

Тот же пример, но переписанный на Фортране 90, умещается в две строки исходного текста:

```
REAL, PARAMETER :: PINUMBER = 3.14159
REAL, DIMENSION(1:3) :: BARRAY = (/1.0, 2.0, 3.0/)
```

В Фортране 90 используются следующие атрибуты объектов:

- PARAMETER — объект является именованной константой;
- PUBLIC — объект является доступным за пределами модуля;
- PRIVATE — объект недоступен за пределами модуля;
- POINTER — объект является ссылкой (указателем);
- TARGET — объект можно использовать в качестве адресата в операторах назначения ссылок;
- ALLOCATABLE — объект является динамическим массивом;
- DIMENSION — объект является массивом;
- INTENT — определяет вид связи для параметра процедуры (т. е. является он входным, выходным или и входным и выходным);
- OPTIONAL — необязательный параметр процедуры;
- SAVE — сохранять значение локальной переменной подпрограммы в промежутке между ее вызовами;
- EXTERNAL — для внешней функции;
- INTRINSIC — для внутренней функции.

Атрибут POINTER нельзя использовать совместно с атрибутами TARGET, INTENT, EXTERNAL и INTRINSIC. Атрибут TARGET нельзя использовать вместе с PARAMETER, а POINTER вместе с ALLOCATABLE. Атрибут TARGET несовместим с атрибутами EXTERNAL и INTRINSIC. Есть и другие ограничения на использование атрибутов. С некоторыми из этих ограничений, а также с назначением различных атрибутов мы познакомимся в дальнейшем.

В предложениях описания могут использоваться объекты, описания которых содержатся в других предложениях описания. Следует соблюдать правило: если в описании используется какой-либо объект, определенный пользователем, его описание должно предшествовать данному описанию.

# Именованные и буквальные константы

В разделе описаний программы должны быть объявлены не только переменные, но и именованные константы. Константа в Фортране может быть числовой, логической или символьной. Значение константы не изменяется в процессе выполнения программы. Буквальные константы специального описания не требуют. Предложение описания именованных констант имеет вид:

```
ТИП, PARAMETER :: ИМЯ_1 = ЗНАЧЕНИЕ_1, ИМЯ_2 = ЗНАЧЕНИЕ_2, …
```

или

```
PARAMETER (ИМЯ_1 = ЗНАЧЕНИЕ_1, ИМЯ_2 = ЗНАЧЕНИЕ_2, … )
```

Здесь `ИМЯ_1`, `ИМЯ_2`, … — имена констант, а `ЗНАЧЕНИЕ_i` — значения этих констант. Второй вариант используется в Фортране 77.

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` для разновидности, принятой по умолчанию:

```
2003
0
-12
+377
```

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, записанная в формате *с фиксированной точкой* (иногда говорят — запятой), состоит из необязательного знака, целой части, десятичной точки, дробной части (см. рис. 1.3). Любая из этих частей, кроме точки, может быть опущена.

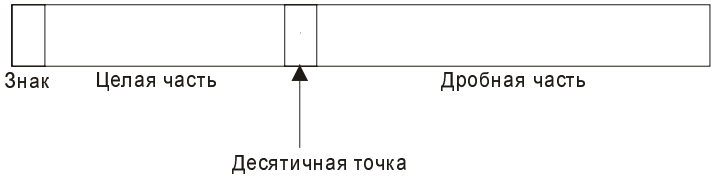



**Рис. 1.3.** Формат вещественной константы с фиксированной точкой

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

В формате *с плавающей точкой* (см. рис. 1.4) дополнительно используются символы `E` или `D`. Любой из этих символов обозначает основание 10, различие состоит в том, что первый символ отвечает простой точности, а второй — двойной точности. После основания следует порядок. Например, запись константы `.31415E1` следует читать так: $.31415 \times 10^1$.

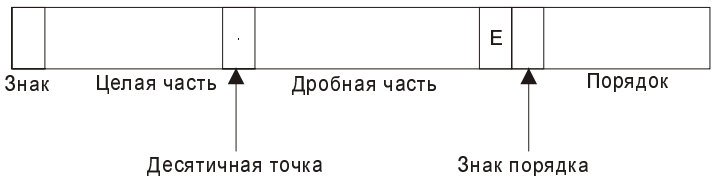



**Рис. 1.4.** Формат вещественной константы с плавающей точкой

Комплексная буквальная константа записывается в круглых скобках:

- `(0., 1.)` — соответствует мнимой единице *i*;
- `(1., 0.)` — соответствует вещественной единице;
- `(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. Вместо апострофов могут быть использованы кавычки.

Примеры буквальных констант типа `CHARACTER`:

```
'STRING'
"HAPPY BIRTHDAY TO JOHN BACKUS!"
```

# 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 *индексами*. Индекс можно считать обобщением понятия номер. Индекс может быть целым значением, как отрицательным, так и положительным.

Надо сказать, что при решении расчетных 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 прикладному программисту-вычислителю приходится решать задачи, сформулированные в терминах уравнений. Это могут быть системы линейных алгебраических уравнений, дифференциальные, интегральные уравнения. В первом случае естественной структурой данных для хранения матрицы коэффициентов, вектора правой части и вектора неизвестных является массив. Во втором и третьем случаях дифференциальную или интегральную формулировку задачи, непрерывную по своей природе, придется заменить дискретной формулировкой, введя, например, сетку или систему базисных функций. В результате, опять придется работать с массивами. Компьютерная математика дискретна по своей природе, ведь множество чисел, которые могут использоваться компьютером, конечно. Фортран — один из немногих языков, обладающих удобными средствами работы с массивами.

Описываются массивы с помощью атрибута `DIMENSION`:

```
REAL, DIMENSION(1:100) :: C
```

В круглых скобках указываются нижняя и верхняя границы диапазона индексов. Массивам посвящена отдельная глава книги.

# Производные типы данных

Кроме предопределенных или встроенных типов данных, заложенных в язык его разработчиками, Фортран допускает создание новых, производных, типов данных. *Производный тип* в Фортране представляет собой структуру, которая состоит из нескольких компонентов. Типы компонентов могут различаться, причем тип компонента, в свою очередь, может быть производным типом. Предложение описания типа начинается с ключевого слова `TYPE`:

```
TYPE NEWTYPE
 REAL X, Y
 INTEGER COLOR
 CHARACTER(LEN =1 5) OBJECT_NAME
END TYPE NEWTYPE
```

В данном примере компонентами нового типа NEWTYPE являются X, Y, COLOR и OBJECT_NAME.

Производные типы в определенном смысле можно считать аналогом записей в языке Паскаль и структур в языке С. Иногда удобно объединить под общим именем разнородные данные, имеющие разный тип. Массивы объединяют однотипные данные и, следовательно, не позволяют решить эту проблему.

Кроме производных типов Фортран позволяет определить и новые операции. Операции, определенные пользователем, расширяют возможности встроенных операций.

# Указатели

Объекты, которые указывают на другие объекты, называются *ссылками* или *указателями*. Более строгое определение заключается в том, что указателем называется переменная, диапазон значений которой состоит из адресов ячеек памяти. Для описания ссылки используется атрибут POINTER. Ссылка может указывать как на существующий объект, так и на специально отведенную область памяти. Указатели можно использовать для программирования динамических структур данных, таких, например, как списки, очереди, стеки и деревья, а также в других случаях.

# Метки

*Метка* — это элемент языка, который утратил свою популярность уже достаточно давно, но, тем не менее, окончательно отказаться от меток в программах сложно. Метки в программах на Фортране используются обычно для реализации алгоритмической структуры "обход" (совместно с оператором безусловного перехода GOTO), для обозначения последнего оператора цикла DO и для ссылок на операторы формата.

В фиксированном формате первые пять позиций любой строки специально отведены для записи меток. Метка представляет собой целое число без знака от 1 до 99999 и может находиться в любой из этих первых пяти позиций. Метки нельзя дублировать, в пределах одного программного компонента (главной программы или подпрограммы), хотя многократные ссылки на одну и ту же метку вполне допустимы. Пример (метка 1):

```
      I = 0
  1   I = I + 1
      CALL MISTERY(I, YOHO)
      PRINT *, YOHO
      GO TO 1
```

В свободном формате метка также является целой буквальной константой без знака, она отделяется от помеченной конструкции одним или несколькими пробелами. В приведенном ниже примере меткой является 4:

```
PRINT *, "X2 =", X2; 4 AAA = '1'; PRINT *, AAA; GO TO 4
```

# Комментарии

Важной частью исходного текста программы являются *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. Программисты считают, что хорошая программа — это программа с подробными описаниями различных ее фрагментов, оформленными в виде комментариев. Грамотное применение комментариев упрощает понимание программы, облегчает жизнь программистам, работающим с готовым текстом. Вряд ли стоит доказывать, что гораздо легче разобрать работу подробно прокомментированной программы, чем программы, состоящей только из операторов. Ну а это — важный фактор, влияющий на эффективность труда программиста.

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.

В фиксированном формате строка комментария обозначается буквой `C` в первой позиции строки. В свободном формате такие комментарии не допускаются. В фиксированном формате для обозначения комментария можно использовать точку с запятой или восклицательный знак, но эти символы должны находиться в позициях с 7 по 72. Восклицательный знак может располагаться и в любой из первых шести позиций.

В свободном формате комментарий начинается символом "`!`", который может находиться в любом месте строки, а заканчивается признаком конца строки.

Строки комментария не продолжаются, поскольку любой символ в строке комментария, в том числе и символ продолжения, считается частью комментария и утрачивает свои специальные служебные функции.

# Операторы

Ранее мы уже познакомились с предложениями описания. Кроме них в программе обычно присутствуют исполняемые операторы. Тело каждой программы или подпрограммы состоит 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
```

Выражение справа от символа присваивания состоит из констант, переменных, обращений к функциям и знаков операций. 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)
```

Следует помнить, что, несмотря на внешнее сходство записи оператора присваивания Фортрана с математическим символом равенства, оператор присваивания описывает некоторую последовательность действий (вычисления, выборка из памяти и запись в память), а математическое равенство является отношением между двумя величинами.

Выражение в правой части оператора присваивания может быть арифметическим, логическим или иным. Арифметические выражения строятся из переменных, констант, вызовов функций и символов арифметических операций. Эти символы приведены в табл. 1.3. 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`. Следует заметить, что замечательной особенностью языка Фортран является наличие операции возведения в степень `**`. В выражении

```
A**B
```

`A` является основанием, а `B` — показателем степени. На допустимые значения основания накладываются понятные и естественные ограничения — нельзя, например, возводить отрицательное вещественное число в вещественную степень. Наличие простого способа вычисления произвольной степени удобно при программировании сложных вычислений.

При программировании арифметических выражений следует помнить о *приоритетах операций*. Приоритет операций определяет порядок их выполнения. Первыми выполняются унарные операции (возведение в степень и т. п.). Затем арифметические операции умножения и деления, они считаются равноприоритетными. Далее выполняются операции сложения и вычитания, тоже равноприоритетные. Если подряд идут несколько операций с одинаковыми приоритетами, они выполняются слева направо, по очереди. Исключение составляет операция возведения в степень. Несколько идущих подряд операций возведения в степень выполняются справа налево.

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`.

Операции в порядке убывания их приоритета (чем больше значение, тем меньше приоритет) приведены в табл. 1.5.

***Таблица 1.5.*** *Приоритеты стандартных арифметических и логических операций*

| Приоритет | Операция |
|---|---|
| 1 | ** |
| 2 | * |
| | / |
| 3 | + |
| | - |
| 4 | .EQ. |
| | .NE. |
| | .LT. |
| | .LE. |
| | .GT. .GE. |
| | == /= |
| | < <= |
| | > >= |
| 5 | .NOT. |
| 6 | .AND. |
| 7 | .OR. |
| 8 | .EQV. |
| | .NEQV. |

В арифметических выражениях можно использовать операнды (*операнд* — это значение, к которому применяется операция) различных типов и различных разновидностей одного типа. Это одно из отличий языка Фортран

от других языков программирования. Допустимой является, таким образом, конструкция вида:

```
INTEGER :: IVAR
REAL :: RVAR1, RVAR2
…
RVAR2 = IVAR + RVAR1
```

При вычислении значения арифметического выражения в правой части оператора присваивания сначала выполняется преобразование всех значений к наиболее "сложному" типу (целый "проще", чем вещественный, а вещественный "проще", чем комплексный). Результат преобразуется в формат, соответствующий типу переменной в левой части оператора присваивания.

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

## Условный оператор *IF...THEN...END IF*

Оператор `IF…THEN…END IF` называется *условным оператором* и имеет вид:

```
IF(ВЫРАЖЕНИЕ) THEN
       ОПЕРАТОР_1
       …
       ОПЕРАТОР_N
END IF
```

где `ВЫРАЖЕНИЕ` является *логическим*. Логическое выражение принимает одно из двух возможных значений — "истина" или "ложь". Часто в роли логического выражения выступает какое-то 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", выполняются операторы `ОПЕРАТОР_1`, `ОПЕРАТОР_2` и т. д. В противном случае выполняться будет оператор, следующий после `ENDIF`.

Перед ключевым словом `IF` может находиться имя:

```
ИМЯ : IF(ВЫРАЖЕНИЕ) THEN
       ОПЕРАТОР_1
       …
       ОПЕРАТОР_N
ENDIF ИМЯ
```

Использование именованных конструкций позволяет улучшить читаемость программы. Пример условного оператора:

```
IF(CENTIGRADE == 0) THEN
       PRINT *, 'ТЕМПЕРАТУРА ЗАМЕРЗАНИЯ ВОДЫ! УХ, КАК ХОЛОДНО!'
ENDIF
```

# Условный оператор *IF...THEN...ELSE...END IF*

Оператор `IF...THEN...ELSE...END IF` является полной версией условного оператора и имеет вид:

```
IF(ВЫРАЖЕНИЕ) THEN
       ОПЕРАТОР_1_1
       …
       ОПЕРАТОР_1_N
ELSE
       ОПЕРАТОР_2_1
       …
       ОПЕРАТОР_2_N
END IF
```

Если `ВЫРАЖЕНИЕ` принимает значение "истина", 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!'
END IF
```

Условные операторы можно вкладывать друг в друга, программируя таким образом сложные (многовариантные) ветвления. Рассмотрим следующий оператор:

```
IF(ВЫРАЖЕНИЕ_1) THEN
       ОПЕРАТОР_1
       …
ELSE
       IF(ВЫРАЖЕНИЕ_2) THEN
              ОПЕРАТОР_2
              …
       ELSE IF(ВЫРАЖЕНИЕ_3) THEN
              ОПЕРАТОР_3
              …
       ELSE IF(ВЫРАЖЕНИЕ_N) THEN
              ОПЕРАТОР_N
              …
       ENDIF
END IF
```

Здесь вначале вычисляется значение логического выражения ВЫРАЖЕНИЕ_1. Если оно истинно, выполняется оператор ОПЕРАТОР_1 и все последующие до ключевого слова ELSE, если же это значение ложно, вычисляется значение выражения ВЫРАЖЕНИЕ_2. В том случае, когда полученное значение истинно, будет выполняться оператор ОПЕРАТОР_2, при значении "ложь" будет вычисляться выражение ВЫРАЖЕНИЕ_3 и т. д.

Если выражения ВЫРАЖЕНИЕ_i независимы, то есть вычисление их значений в любом порядке дает один и тот же результат для каждого из них, имеет смысл располагать их в таком порядке, чтобы выражение, с наибольшей вероятностью принимающее значение "истина", стояло на первом месте, выражение, принимающее значение "истина" с меньшей вероятностью, — на втором и т. д. Это уменьшит время выполнения данного фрагмента программы, особенно если вложенный оператор появляется в цикле, который выполняется многократно.

Пример вложенных условных операторов:

```
IF(TWO == 2) THEN
       IF(ONE == 1) THEN
              PRINT *, 'ЕДИНИЦА РАВНА 1'
       ELSE
              PRINT *, 'ЕДИНИЦА НЕ РАВНА 1'
       END IF
ELSE
       IF(THREE == 3) THEN
              PRINT *, 'ТРИ РАВНО 3'
       ELSE
              PRINT *, 'ТРИ НЕ РАВНО 3'
       END IF
END IF
```

## Оператор *SELECT*

Для ситуаций, где имеется несколько (три и более) альтернатив, больше подходит оператор SELECT CASE. Этот оператор реализует многовариантное ветвление, называется *оператором выбора* и имеет следующий вид:

```
SELECT CASE(ВЫРАЖЕНИЕ)
 CASE(МНОЖЕСТВО_ЗНАЧЕНИЙ_1)
       ОПЕРАТОРЫ_1
 CASE(МНОЖЕСТВО_ЗНАЧЕНИЙ_2)
       ОПЕРАТОРЫ_2
...
```

```
 CASE(МНОЖЕСТВО_ЗНАЧЕНИЙ_N)
       ОПЕРАТОРЫ_N
 CASE DEFAULT
       ОПЕРАТОРЫ
END SELECT
```

Рассмотрим элементы этой конструкции. После слов `SELECT CASE` находится заключенное в круглые скобки выражение, называемое *селектором* оператора `SELECT`. Значение селектора может оказаться в одном из списков значений, находящихся в круглых скобках после слов `CASE`. Ветвь `CASE DEFAULT` отвечает всем не перечисленным в списках значениям выражения. При выполнении оператора `SELECT` вначале вычисляется значение селектора. Затем выбирается тот список значений, которому принадлежит полученное значение, и выполняются соответствующие операторы.

Рассмотрим следующий пример. Пусть необходимо преобразовать целое число N в зависимости от величины остатка от его деления на 17 следующим образом:

- если остаток 0, то $N = 0$;
- если остаток 1 или 6, то $N = -N$;
- если остаток 2, 3 или 5, то $N = 2 \times N$;
- если остаток 4, то $N = 3 \times N$;
- во всех прочих случаях $N = 5 \times N$.

Решение этой задачи на Фортране выглядит следующим образом:

```
SELECT CASE(MOD(N, 17))
 CASE(0)
       N = 0
 CASE(1, 6)
       N = -N
 CASE(2, 3, 5)
       N = 2 * N
 CASE(4)
       N = 3 * N
 CASE DEFAULT
       N = 5 * N
END SELECT
```

В данном примере селектором является выражение `MOD(N, 17)`. Это встроенная функция, которая вычисляет значение остатка от деления первого аргумента на второй. Кроме того, имеются 4 списка значений и ветвь `DEFAULT`.

# Оператор цикла со счетчиком *DO...END DO*

*Оператор цикла* является важнейшим оператором и имеется в большинстве современных языков программирования. Цикл позволяет многократно выполнить некоторое множество действий, задаваемых операторами, составляющими его *тело*. В Фортране имеется две разновидности оператора цикла. Основной оператор цикла — *цикл со счетчиком* (или цикл с параметром). Он имеет вид:

```
DO ПАРАМЕТР_ЦИКЛА = ВЫРАЖЕНИЕ_1, ВЫРАЖЕНИЕ_2
       ОПЕРАТОР_1
       ОПЕРАТОР_2
       …
END DO
```

Здесь переменная ПАРАМЕТР_ЦИКЛА, называемая *управляющей переменной* цикла DO, является произвольным идентификатором, который объявляется как переменная любого скалярного типа (к скалярным относятся целый, символьный, булев и перечисленные типы).

При выполнении оператора DO сначала вычисляется значение выражения ВЫРАЖЕНИЕ_1, затем вычисляется значение выражения ВЫРАЖЕНИЕ_2, далее управляющая переменная цикла последовательно пробегает все значения от ВЫРАЖЕНИЕ_1 до ВЫРАЖЕНИЕ_2. Сами эти значения остаются неизменными в ходе выполнения всего цикла. В том случае, когда значение ВЫРАЖЕНИЕ_1 оказывается больше значения ВЫРАЖЕНИЕ_2, тело цикла не будет выполняться вовсе.

Цикл может быть задан следующим образом:

```
DO ПАРАМЕТР_ЦИКЛА = ВЫРАЖЕНИЕ_1, ВЫРАЖЕНИЕ_2, ВЫРАЖЕНИЕ_3
       ОПЕРАТОР_1
       ОПЕРАТОР_2
       …
END DO
```

В этом случае ВЫРАЖЕНИЕ_3 задает шаг изменения параметра.

Циклу можно присвоить имя, оно должно находиться перед заголовком цикла и отделяется от него двоеточием:

```
ИМЯ_ЦИКЛА:DO ПАРАМЕТР_ЦИКЛА = EXPRESSION_1, EXPRESSION_2
       ОПЕРАТОР_1
       ОПЕРАТОР_2
       …
END DO ИМЯ_ЦИКЛА
```

Использование именованных циклов позволяет более четко обозначить логику работы программы, что особенно удобно при использовании вложенных

циклов. Если вы, уважаемый читатель, уже имеете опыт программирования, то знаете, насколько иногда неудобно бывает разобраться в том, какие операторы какому циклу принадлежат.

В теле цикла DO нельзя использовать операторы, изменяющие значение управляющей переменной. Использование подобных конструкций приводит к ошибкам компиляции. После выполнения цикла DO значение управляющей переменной становится неопределенным.

## Оператор цикла с предусловием *DO WHILE...END DO*

Оператор *цикла с предусловием* имеет вид:

```
DO WHILE(ВЫРАЖЕНИЕ)
 ОПЕРАТОР_1
 …
END DO
```

При выполнении этого оператора вначале вычисляется значение логического выражения в круглых скобках после слова WHILE. Если это значение истинно, выполняются операторы ОПЕРАТОР_1 и следующие после него, до строки ENDDO. Эти операторы составляют тело цикла. Затем значение выражения проверяется вновь и т. д., до тех пор, пока выражение не примет значение "ложь". Если выражение принимает значение "ложь" при первой же проверке, то операторы не выполняются вообще. Отметим частный случай:

```
DO WHILE(.TRUE.)
 ОПЕРАТОР_1
 …
END DO
```

Здесь тело цикла будет выполняться бесконечно. В таких случаях говорят о "зацикливании" программы. Прервать работу зациклившейся программы можно средствами операционной системы. Иногда бывает удобно проводить проверку на возможный выход из цикла где-нибудь в его середине, а не в начале или конце. Такой выход из цикла обеспечивается вызовом процедуры EXIT, прерывающей выполнение самого внутреннего вложенного цикла, будь то DO или DO WHILE. Пример оператора цикла с предусловием:

```
COUNTER = 0
DO WHILE(COUNTER < 10)
       PRINT *, 'ЗНАЧЕНИЕ СЧЕТЧИКА РАВНО ', COUNTER
       COUNTER = COUNTER + 2
END DO
```

Перед циклом DO WHILE, как правило, тем переменным, которые изменяются во время выполнения цикла (каждое выполнение цикла называют итера-

цией цикла), присваивается начальное значение, то есть выполняется их инициализация.

Оператор цикла с постусловием в Фортране отсутствует. Для применения циклов можно сформулировать следующие рекомендации:

- используйте цикл `DO` в том случае, когда точно знаете, сколько раз должно быть выполнено тело цикла. В противном случае обратитесь к циклу `DO WHILE`;
- используйте `DO WHILE`, если хотите, чтобы проверка была произведена прежде, чем будет выполняться тело цикла, а число повторений цикла заранее неизвестно.

# Примеры программ

Познакомившись с основными операторами языка Фортран, рассмотрим примеры простых программ. В первом примере необходимо вычислить сумму натуральных чисел от 1 до 20. Читатель, знакомый с математикой, знает, что сумму первых n натуральных чисел можно найти и по формуле $S_n = n(n + 1)/2$. Используя эту формулу, можно проверить, правильно ли работает программа и не допущены ли при составлении алгоритма и во время набора программы ошибки. При разработке программы ее отладка, проверка правильности работы является обязательным этапом работы программиста. Может оказаться, что даже если все операторы программы написаны правильно с точки зрения формальных правил языка, ошибка допущена в самом алгоритме или в его записи на языке программирования. Такие ошибки иногда труднее обнаружить, чем синтаксические. Программист должен убедиться в том, что программа дает правильный результат. Для тестирования программы используются такие наборы значений входных параметров или такие предельные случаи ее работы, для которых известен точный результат. Результат работы программы в этом случае сравнивается с точными значениями. Исходный текст программы приведен в листинге 1.3.

**Листинг 1.3. Вычисление суммы натуральных чисел**

```
PROGRAM SUMMATION
 IMPLICIT NONE
 INTEGER(2) :: I, SUMMA = 0
 DO I = 1, 20
       SUMMA = SUMMA + I
 END DO
 PRINT *, '1 + 2 + ... + 20 = ', SUMMA
END
```

В программе суммирования отметим некоторые элементы. Это предложение IMPLICIT NONE, отменяющее правила неявного объявления типа. Затем следует предложение описания переменных I и SUMMA, используемых в программе, причем последней также присваивается начальное значение 0 (выполняется инициализация):

```
INTEGER(2) :: I, SUMMA = 0
```

Такой способ инициализации предпочтителен, потому что значение записывается в ячейку памяти при компиляции программы, а во время ее выполнения на эту операцию время уже не тратится.

Переменная SUMMA используется для хранения частичной суммы. До первого оператора присваивания, содержащего имя переменной в левой части, ее значение не определено. В некоторых реализациях языка неопределенные переменные могут автоматически обнуляться. Тем не менее, рекомендуется явным образом инициализировать переменные. Это может избавить программиста от некоторых весьма неприятных ситуаций!

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 в программах на языке Фортран 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, текст которой приведен в листинге 1.5, предназначена для решения квадратного уравнения, если заданы его коэффициенты *a*, *b* и *c*:

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

**Листинг 1.4. Программа решения квадратного уравнения**

```
PROGRAM QUADR_EQUATION
 IMPLICIT NONE
 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
 PRINT *, "X2 =", X2
END PROGRAM QUADR_EQUATION
```

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 стандартное устройство вывода, то есть на экран монитора.

Программа `CELSIUS_TO_FAHRENHEIT` (листинг 1.5) 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 1.5. Вывод таблицы соответствия температур по Цельсию и Фаренгейту**

```
PROGRAM CELSIUS_TO_FAHRENHEIT
 IMPLICIT NONE
 INTEGER(2) :: 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
```

Простая, казалось бы, операция деления `/` имеет в Фортране особенности, которые обязательно следует учитывать при программировании арифметических выражений. Делимое и делитель могут иметь любой числовой тип. 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 для самостоятельной работы

1. Дайте характеристику языка Фортран.
2. Какие символы включает алфавит языка Фортран?
3. Какие специальные символы можно использовать в программах на Фортране?
4. Опишите структуру программы на Фортране.
5. Можно ли в одном арифметическом выражении использовать операнды разного типа?
6. Как выполняется операция деления, когда делимое и делитель имеют разные типы?
7. Перечислите и дайте характеристику встроенным типам Фортрана.
8. Приведите примеры описания переменных комплексного типа.
9. Приведите примеры описания констант комплексного типа.
10. Является ли правильным следующее описание?

```
INTEGER INTEGER
```

11. Является ли правильным следующее описание?

```
LOGIC ALL
```

12. Как описать константу со значением 2.718?
13. Является ли правильным следующее описание?

```
COMMON :: VAR
```

14. Поясните смысл следующей строки:

```
A = 1; B = 2.1 ! Integer and real variables; C = 1.7; D = 30
```

15. Являются ли правильными следующие строки?

```
Y = COS(MAX(X, Y)) * EXP(-COS(Z)**I) - TAN(AT&
AN(X4))
```

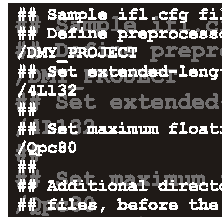
16. Не прибегая к помощи компьютера, определите результат выполнения программы:

```
PROGRAM EX16
INTEGER(4) :: N
REAL :: X
N = 0; X = 1.0;
DO WHILE(X < 1.0E5)
N = N + 1
X = 2.0 * X
```

```
END DO
PRINT *, N, ' ', X
END
```

17. Не прибегая к помощи компьютера, определите результат выполнения программы:

```
PROGRAM EX17
INTEGER(2) :: N
REAL :: S, X
S = 0.0; N = 1
DO WHILE(N <= 100)
X = 3.0 * N + 2.0; X = 1.0 / X
S = S + X
N = N + 1
END DO
PRINT *, N, ' ', S
END
```




# Глава 2

# Типы данных

Тип данных — это характеристика, определяющая множество допустимых значений и поведение объекта. Типом данных в Фортране характеризуются объекты нескольких видов: константы — именованные и буквальные, имена переменных, выражения и процедуры. Различают встроенные и производные типы данных. К встроенным относятся типы, существование которых обеспечивается языком по умолчанию. В Фортране присутствуют пять встроенных типов. Тип объекта может определяться с помощью операторов явного описания, или по правилам неявного определения типа, или по способу написания объекта (для буквальных констант). Начиная с Фортрана 90 в язык введено понятие *разновидности типа*, разграничившее задачи назначения типа объекта и указания точности его представления. Одна из разновидностей определяется как стандартная и назначается типу по умолчанию.

Производные типы данных определяются как комбинации данных встроенных типов (структуры), позволяя описывать и использовать объекты произвольного характера как единое целое. Производные типы — мощное средство языка, благодаря которому достигается высокая степень абстракции данных.

## Числовые типы

Из пяти встроенных типов Фортрана к числовым относятся три: целый, вещественный и комплексный. Числовые типы используются для выполнения арифметических действий: сложения, вычитания, умножения и деления, а также как аргументы и результаты математических функций. Все числовые типы предусматривают наличие нулевого значения, не зависящее от знака: нуль без знака не отличается от нуля с плюсом или минусом.

### Целый тип

*Целый тип* используется для представления ряда целых чисел в некотором диапазоне допустимых значений, конкретная величина которого зависит от

особенностей процессора. Переменные целого типа описываются ключевым словом INTEGER:

```
INTEGER :: I, J, K
```

Целые числа представляются *точными* значениями. Разновидности целого типа определяют *диапазон допустимых значений объекта* и связаны с *числом разрядов внутреннего представления* целого числа. Стандартная разновидность целого типа обычно определяется длиной компьютерного слова и обеспечивает диапазон допустимых значений, включая все целые числа от $-2^{n-1}$ до $+2^{n-1} - 1$, где $n$ — длина компьютерного слова. Для 32-разрядных компьютеров диапазон допустимых значений целого типа стандартной разновидности составляет $[-2^{31}, 2^{31} - 1]$.

Стандарт языка не устанавливает правил задания параметров разновидности, кроме того, что они должны быть неотрицательными. Естественным представлением параметра разновидности целого типа является число байтов, отводящихся под целое, и многие разработчики компиляторов следует по этому пути. Так, параметр стандартной разновидности целого типа для 32-разрядного компьютера равен 4. При этом разновидности 1, 2 и 3 могут не поддерживаться совсем.

Как правило, для объектов целого типа необходимо гарантировать требуемый диапазон значений. Параметры разновидности зависят от процессора, и при переносе программы на компьютеры с процессорами другого типа должны пересматриваться. Но Фортран располагает встроенными средствами, позволяющими определять нужную разновидность целого типа по заданному диапазону значений, не связывая корректность вычислений с конкретной аппаратной платформой.

## Буквальные константы целого типа

Тип буквальных констант определяется способом их изображения в программе. Буквальные константы целого типа представляются цепочками цифровых символов со знаком или без знака, трактуемых, если нет никаких специальных знаков, как десятичные целые числа:

```
20
-1024
32767
+301
```

Разновидность типа буквальной константы целого типа может быть указана параметром разновидности, помещенным сразу после последней цифры константы за знаком подчеркивания:

```
32767_3
20_1
+301_2
```

что может быть понято как числа 32767, 20 и 301 длиной в 3, 1 и 2 байта соответственно. Но для такого способа указания разновидности типа необходимо знать заранее, что разновидности 1, 2 и 3 поддерживаются процессором, а это может быть неизвестно. Отсутствие указания разновидности приведет к назначению для всех буквальных констант стандартной разновидности, которая может меняться при переходе на процессор другого типа. Правильным решением задачи выбора разновидности для буквальных констант является параметрическое определение нужного значения. Встроенная функция Фортрана `SELECTED_INT_KIND(n)` *(см. гл. 10)* возвращает значение параметра разновидности, обеспечивающее диапазон целых от $-(10^n - 1)$ до $+(10^n - 1)$ с наименьшим запасом. Описание и определение именованной константы целого типа, содержащей значение нужного параметра разновидности, гарантирует правильность задания разновидности типа буквальных констант без привязки к конкретным значениям. В операторы описания следует включить описание целого параметра, например, с именем `n6`:

```
INTEGER, PARAMETER :: n6 = SELECTED_INT_KIND(6)
```

Тогда в тексте программы можно вводить буквальные константы целого типа, например,

```
20405_n6
```

с уверенностью в том, что параметр разновидности `n6` обеспечивает диапазон целых от −99999 до 99999.

Встроенная функция `SELECTED_INT_KIND(R)` возвращает наименьший параметр разновидности, обеспечивающий диапазон целых чисел от $-(10^R - 1)$ до $(10^R - 1)$. При недостижимости заданного диапазона возвращается −1.

Встроенные функции `KIND(n)` и `RANGE(n)` *(см. гл. 10)* позволяют определить параметр разновидности для конкретного типа данных и действительный диапазон значений для заданного объекта. Пример переносимой программы, использующей буквальные константы различных разновидностей целого типа, приведен в листинге 2.1.

**Листинг 2.1. Пример использования буквальных констант целого типа (свободный формат записи)**

```
PROGRAM int_types
 INTEGER, PARAMETER :: n6 = SELECTED_INT_KIND(6)
 INTEGER, PARAMETER :: n12 = SELECTED_INT_KIND(12)
 INTEGER i, j, k
 IF (n6 > 0 .AND. n12 > 0) THEN
       k = 200_n6
       j = -10000_n12
 ELSE
```

```
      STOP 'Диапазон значений переменных не обеспечен'
 END IF
 DO I = 1_n6, k, 2_n6
      j = j + 20_n12 * k
      PRINT *, j
 END DO
 PRINT *, k
END PROGRAM int_types
```

Буквальные константы целого типа могут быть представлены не только числами десятичной системы счисления, но также числами двоичной, восьмеричной и шестнадцатеричной систем счисления. Они вводятся оператором DATA как строковые константы (последовательность цифровых или алфавитно-цифровых символов, заключенная в апострофы или двойные кавычки) с предшествующим символом, указывающим на основание системы счисления:

- B'10010111' — буквальная константа в двоичной системе счисления;
- O"075476" — буквальная константа в восьмеричной системе счисления;
- Z'2B5E' — буквальная константа в шестнадцатеричной системе счисления.

## Вещественный тип

К вещественному типу относятся объекты, числовые значения которых подразумевают наличие дробной части и определяются только с некоторой степенью точности. Число точных знаков вещественного значения определяется разновидностью типа. Стандарт языка не определяет ни допустимый диапазон значений вещественного типа, ни число значащих цифр. Типичными для 32-разрядных процессоров являются диапазон от $10^{-38}$ до $10^{38}$ и шесть-семь значащих цифр для стандартной разновидности вещественного типа.

Переменные вещественного типа описываются ключевым словом REAL:

```
REAL :: R, Z, Y
```

Стандарт обязывает поддерживать хотя бы еще одно дополнительное представление величин вещественного типа, обеспечивающего большую, чем стандартная, точность. В принципе, это представление может быть выражено определением типа DOUBLE PRECISION, который все еще поддерживается, но не рекомендуется к использованию как устаревший. Действительно, точность вещественной величины является характеристикой разновидности вещественного типа, а не определением другого типа данных.

### Буквальные константы вещественного типа

Чтобы буквальная числовая константа определялась как вещественная, в ее записи должен присутствовать хотя бы один атрибут формы записи вещест-

венных констант. Вещественная константа состоит из целой части со знаком или без знака, десятичной точки, дробной части и степенной части со знаком или без знака. Любая из составляющих может быть опущена, однако нельзя одновременно опускать целую и дробную части или степенную часть и десятичную точку. Степенная часть состоит из буквы "E" и целого числа со знаком или без знака. Вещественная константа, записанная по полной форме, например,

```
-82.34E-2
```

означает $-82.34 \times 10^{-2}$. Она содержит знак, целую часть, десятичную точку и степенную часть. Эту же константу можно записать как `-.8234` или `-0.8234`. Следующие числа также будут восприняты как вещественные константы стандартной разновидности:

```
-0.2
3E-1
3.141592653
```

Запись `-10` будет истолкована как буквальная константа целого типа. Запись `.E+0`, претендующая на представление вещественного нуля стандартной разновидности, считается ошибкой.

Количество знаков, заданных в буквальной константе, не влияет на ее разновидность.

## Разновидности вещественного типа

Как и с параметрами разновидности целого типа, показатель разновидности вещественного типа может быть естественно сведен к числу байтов, занимаемых вещественным числом, однако стандарт языка не требует этого. Если параметр разновидности известен заранее, его можно указать непосредственно при задании буквальной константы:

```
-1.7_4,
```

однако такой способ определения разновидности может отрицательно сказаться на переносимости программы. Оптимальным решением является введение в блоке описания именованной константы, содержащей требуемый параметр разновидности:

```
INTEGER, PARAMETER :: L9 = SELECTED_REAL_KIND(9, 99)
```

Затем, уже в тексте программы, можно использовать `L9` для задания параметра разновидности в буквальных константах вещественного типа.

Встроенная функция `SELECTED_REAL_KIND(P,R)` *(см. гл. 10)* определяет параметр разновидности, гарантирующий `P` точных десятичных знаков вещественного числа с абсолютным значением в диапазоне от $10^{-R}$ до $10^{R}$. Если требуемые диапазон и точность достижимы, то функция возвращает значение наименьшего параметра разновидности, удовлетворяющего аргументам `P` и `R`. Если подходящая разновидность отсутствует, то возвращается значение $-1$.

При недостижимости требуемой точности возвращается −2, а при недоступности степенного диапазона — −3. Пример программы, обеспечивающей переносимость для буквальных констант вещественного типа, приведен в листинге 2.2.

**Листинг 2.2. Пример использования буквальных констант вещественного типа (свободный формат записи)**

```
PROGRAM p_types
 INTEGER, PARAMETER :: r6 = SELECTED_INT_KIND(6)
 INTEGER, PARAMETER :: r12 = SELECTED_REAL_KIND(9)
 REAL g
 IF (r6 > 0 .AND. r12 > 0) THEN
   DO
      WRITE(*, *) 'Введите число от 0 до 100:'
      READ(*, *) g
      WRITE(*, *) g, ' световых лет = ', &
      &g * 365_r6 * 24_r6 * 3600_r6 * 2.99792458e08_r12',' метров'
   END DO
 ELSE
   STOP 'Диапазон значений переменных не обеспечен'
 END IF
END PROGRAM p_types
```

В более ранних версиях Фортрана (до 90) для представления вещественных величин с удвоенной точностью использовался тип `DOUBLE PRECISION`, а для определения буквальных констант этого типа использовалось степенное представление с буквой "D" в ограничителе степенной части:

```
-1.7D01
```

Такое представление буквальных констант допустимо. Разновидность типа для буквальной константы, заданной таким образом, указывать уже не нужно (и нельзя), так как ее разновидность уже фактически определена. Начиная с Фортрана 90 тип `DOUBLE PRECISION` эквивалентен типу `REAL(KIND = DP)`, где `DP` — целая именованная константа, имеющая значение, возвращаемое функцией `KIND(0.D0)`.

## Комплексный тип

Одним из достоинств Фортрана как языка научных и инженерных расчетов является наличие встроенного комплексного типа данных. Переменные комплексного типа описываются ключевым словом `COMPLEX`:

```
COMPLEX :: Z
```

Объект комплексного типа определяется парой объектов вещественного типа, и разновидности комплексного типа совпадают (и определяются) разновидностью вещественных компонентов комплексного объекта.

Однако если параметр разновидности определяется как число байтов, отводимых для хранения объекта, то параметр разновидности комплексного типа совпадает не с числом байтов, занятых объектом, а с половиной этого числа.

Буквальные константы комплексного типа обозначаются разделенной запятой парой вещественных или целых буквальных констант, заключенных в круглые скобки. Например:

```
(-1., 4.5)
(0.23, 99E-2)
(1, 4.6_4)
(12, 34)
```

Первый элемент из пары всегда обозначает вещественную часть комплексной константы, а второй (после запятой) соответствует ее мнимой части. Разновидность типа буквальной комплексной константы определяется разновидностью типа компоненты с более высокой точностью. Если обе компоненты целого типа, то комплексная константа получит разновидность, соответствующую стандартной разновидности вещественного типа.

Функции `KIND`, `PRECISION` и `RANGE` применимы и к комплексным объектам.

# Нечисловые типы

К нечисловым встроенным типам данных относятся логический и символьный типы. Объекты логического типа используются для представления результатов логических операций в операторах управления. Символьные типы используются для представления текстовых объектов. Так же, как и числовые, символьный и логический типы могут поддерживать более одной разновидности типа.

## Логический тип данных

Объекты логического типа способны принимать только два значения — "истина" и "ложь" (или находиться в состоянии неопределенности). Для их представления достаточно одного бита, но, как правило, отводится не меньше байта. Поддержка коротких разновидностей логического типа очень полезна для упаковки больших логических массивов.

Буквальные константы логического типа могут принимать только два значения: `.TRUE.` и `.FALSE.`. Узнать значение параметра стандартной разновидности логического типа можно с помощью выражения `KIND(.TRUE.)`.

## Символьные данные

Объекты символьного типа представляют собой наборы текстовых символов, не ограниченных набором символов Фортрана. Для описания переменных символьного типа используется ключевое слово `CHARACTER`. Оператор описания:

```
CHARACTER :: LETTER
```

описывает символьную переменную длиной в один символ — букву. Длину символьной переменной можно указать в параметрах ключевого слова:

```
CHARACTER(LEN = 16) :: WORD
```

Здесь `WORD` — переменная из 16 символов.

Фортран обязан поддерживать набор символов ASCII (American Standard Code for Information Interchange, — Американский Стандарт Кода для Обмена Информацией), но объекты символьного типа могут состоять из любых символов, поддерживаемых процессором.

Буквальные константы стандартной разновидности символьного типа представляют собой цепочки символов, заключенных в апострофы или в двойные кавычки:

```
'Кто-то пришел'
"A String"
```

Стандартную разновидность символьных объектов можно определить с помощью функции `KIND` от любой буквальной константы символьного типа, например, `KIND('ASCII')`.

Другие разновидности символьного типа могут использоваться для представления символов, занимающих более одного байта, таких как символы UNICODE 2 или иероглифы, если процессор их поддерживает.

В отличие от всех остальных встроенных типов, параметр разновидности символьного типа *предшествует* буквальной константе:

```
1_'Long letter'
```

Все разновидности символьного типа должны поддерживать символ пробела.

Апострофы и кавычки являются ограничителями и в состав константы не входят. Для представления их в составе текстовой константы можно использовать другие ограничители:

```
'Никогда не говори "Никогда!" '
```

Или использовать удвоение ограничителя без пробелов:

```
'Признак Д''Аламбера'
```

так как удвоенный ограничитель без пробелов воспринимается как одиночный буквальный символ.

Пробел, включенный в буквальную символьную константу, имеет значение, как и регистр символов: константы `"A String"` и `"aStringG"` не совпадают.

Управляющие символы, например, перевод строки, в буквальных константах не допускаются. Символьная константа, состоящая из нескольких строк, записывается как цепь строк продолжения:

```
LONG_STRING = 'Ее глаза на звезды не похожи,&
& Нельзя уста кораллами назвать,&
& Не белоснежна плеч открытых кожа,&
& И черной проволокой вьется прядь.'
```

Символы "&" и "!" могут входить в состав такой многострочной константы, но комментарии в ней недопустимы. Кроме того, каждая строка продолжения должна начинаться с амперсанда.

Предусматривается возможность сортировки символов по определенным правилам. Фортран требует, чтобы сортирующая последовательность удовлетворяла следующим требованиям:

- A меньше B меньше C меньше D меньше E ... меньше Y меньше Z;
- 0 меньше 1 меньше 2 меньше 3 ... меньше 9;
- пробел меньше A и Z, меньше 0 или;
- пробел меньше 0 и 9 меньше A.

Аналогичные требования накладываются и на строчные буквы, если процессор их поддерживает.

# Производные типы данных

Современный Фортран предоставляет программисту возможность описывать собственные типы данных, комбинируя их из элементов встроенных типов. Новый тип вводится оператором описания `TYPE`. Так, можно определить производный тип, соответствующий дате, в форме "день, месяц, год", как это показано в листинге 2.3.

**Листинг 2.3. Описание производного типа *DATE***

```
PROGRAM derived_types
 TYPE DATE
       INTEGER DAY
       INTEGER MONTH
       INTEGER YEAR
END TYPE DATE
...
END PROGRAM derived_types
```

Скалярные переменные производных типов называются структурами. Чтобы описать переменную типа DATE, используем оператор описания:

```
TYPE(DATE) BIRTHDAY
```

Для назначения величин компонентам структуры можно использовать конструктор структур:

```
BIRTHDAY = DATE(21, 10, 1953)
```

или присвоить значения каждой компоненте, извлекая их по очереди из структуры *селектором* компонента %:

```
BIRTHDAY%DAY = 21
BIRTHDAY%MONTH = 10
BIRTHDAY%YEAR = 1953
```

Все обращения к конструкторам типов и элементам структур должны появляться в тексте программы строго после описания соответствующего производного типа. Для объектов производных типов могут быть определены присваивание и арифметические операции. Как это делается, будет показано в дальнейшем *(см. разд. "Конструкторы структур и определение скалярных операций для переменных производного типа" гл. 4 и разд. "Именованные интерфейсы и перегрузка процедур" гл. 7).*

Производные типы могут включаться в состав других производных типов. Используем тип DATE в описании типа PERSON, представляющего персональную карточку служащего фирмы:

```
TYPE PERSON
       CHARACTER(LEN = 20)
       FULLNAME TYPE(DATE) DATE_OF_BIRTH
       INTEGER ID
END TYPE PERSON
```

Теперь, чтобы "достать" из структуры типа PERSON год рождения объекта, операцию селекции компонента структуры нужно выполнить дважды:

```
TYPE(PERSON) JOHN
JOHN%DATE_OF_BIRTH%YEAR = 1976
```

Понятно, что с помощью производных типов можно получать программные эквиваленты объектов весьма сложной природы, и относительно просто оперировать ими.

## Массивы встроенных типов

Массивы встроенных типов представляют собой упорядоченные наборы элементов одного типа с одинаковым параметром разновидности. Массивы могут иметь фиксированный размер, когда число элементов массива задается при его описании.

Другим случаем являются динамические массивы, число элементов которых определяется или меняется при выполнении программы. Сначала рассмотрим массивы фиксированного размера. Переменная A определяет массив из 20-ти вещественных элементов, если в ее описании присутствует атрибут DIMENSION с одним параметром, равным 20:

```
REAL, DIMENSION(20) :: A
```

Элементы массива располагаются в порядке возрастания индекса, начиная с 1:

```
A(1), A(2), A(3),…, A(19), A(20)
```

В Фортране 90 появилась возможность задавать не только верхнюю, но и нижнюю границу массива. Индекс первого элемента — нижняя граница массива — может быть любым целым числом, в том числе и отрицательным:

```
REAL, DIMENSION(-5:15) :: B
```

Массив в Фортране может иметь до семи измерений:

```
REAL, DIMENSION(2,-10:5,-5:5,4,-3:2,5,2) :: S
```

Число элементов массива называется его *размером*. Размер массива S равен $2 \times 16 \times 11 \times 4 \times 6 \times 5 \times 2 = 84480$; многомерные массивы иногда требуют очень много памяти.

Число измерений называется *рангом* массива. (Скалярная величина имеет нулевой ранг.)

Число элементов массива в одном измерении называется *экстентом* (протяженностью) массива в данном измерении. Размер массива равен произведению его экстентов по всем измерениям. Совокупность экстентов — целочисленный массив, размер которого равен рангу исходного массива — называется *формой* массива. Итак, массив S, описанный выше, имеет размер 84480, ранг 7 и форму (2, 16, 11, 4, 6, 5, 2).

При передаче массива в подпрограмму формальный параметр может перенимать форму фактического параметра, предопределяя только тип и ранг формального параметра *(см. гл. 8)*. Пример подпрограммы, получающей массив неизвестной формы в качестве формального параметра, приведен в листинге 2.4.

**Листинг 2.4. Определение параметров массива**

```
SUBROUTINE get_array(A)
 REAL, DIMENSION(:, :) :: A
 INTEGER S, I
 INTEGER, DIMENSION(2) :: F
 S = SIZE(A)
 WRITE(*, *) 'Размер полученного массива = ', S
```

```
 F = SHAPE(A)
 WRITE(*, *) 'Экстенты полученного массива:'
 DO I = 1, SIZE(F)
       WRITE(*, *) I, ': ', F(I)
 END DO
 ...
END SUBROUTINE get_array
```

Производные типы могут включать массивы в качестве своих компонентов:

```
TYPE COLOR_POINT
       REAL, DIMENSION(3) :: COORDS
       INTEGER, DIMENSION(3) :: COLOR_RGB
END TYPE COLOR_POINT
```

а массивы могут состоять из элементов производных типов:

```
TYPE(COLOR_POINT), DIMENSION(10) :: POINTSET
```

Некоторые операторы в применении к массивам обрабатывают его элементы один за другим в последовательности, называемой *порядком следования элементов массива*. Этот порядок сводится к тому, что индексы перебираются в порядке слева направо. Сначала пробегает все свои значения самый левый индекс, затем — второй слева и так далее (см. рис. 2.1).

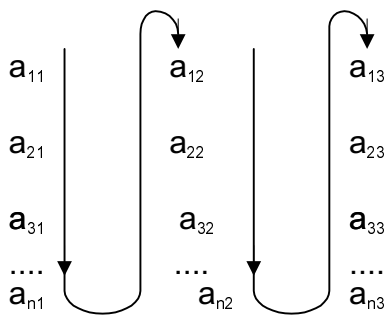



**Рис. 2.1.** Порядок следования элементов двумерного массива: элементы массива расположены "по столбцу"

Во многих реализациях Фортрана элементы массивов располагаются в памяти именно в таком порядке, но это не является обязательным требованием.

Для обращения к отдельному элементу массива нужно задать значение индекса элемента. Индекс можно указать буквальной константой целого типа стандартной разновидности, например, `A(3)`, или набором буквальных констант: `S(1, 0, -2, 3, 0, 1, 1)`. Но, вообще говоря, для задания индекса можно использовать любое *целое скалярное выражение* — арифметическое

выражение, результатом которого является скалярная переменная целого типа, например:

```
A(I * J)
```

где `I` и `J` — переменные целого типа, или

```
A(INT(X + 0.5))
```

где `X` — переменная вещественного типа, а `INT` — функция преобразования к целому типу.

Каким бы образом ни задавались индексы, всегда необходимо, чтобы их значения не выходили за границы массивов, а число индексов совпадало с рангом массива.

## Массивоподобные объекты

Фортран допускает обращение к частям массива и операции с ними. Если в качестве индекса массива указать диапазон значений в рамках границ массива, то полученный объект, называемый *сечением массива*, можно использовать почти так же, как и исходный массив. Например, выражение `A(0:4)` выделяет из массива `A` часть идущих подряд элементов с нулевого по четвертый включительно. При обращении к элементам сечения массива нельзя вводить новую индексацию (например, нельзя записать `A(0:4)(3)`) — нужно пользоваться индексами исходного массива.

Нерегулярное сечение массива можно задать с помощью векторного индекса — выборки элементов массива, индексы которых заданы в виде целочисленного массива:

```
A(INDEX)
```

где `INDEX` — массив типа `INTEGER`.

Более детальное изложение сведений о массивах и массивоподобных объектах можно найти в *гл. 8*.

## Строки и символьные массивы

Массивы могут состоять их элементов любого типа, в том числе, и символьного. Например, описание :

```
CHARACTER, DIMENSION(80) :: STRING
```

определяет символьный массив из 80 элементов и вполне может представлять строку. Каждый элемент массива (отдельный символ) доступен по указанию индекса, а часть строки может быть представлена сечением массива, например `STRING(10:45)`. Для представления массива строк понадобится двумерный массив. Однако в Фортране есть другой, более удобный способ для преставления строк. Длина символьной константы может быть произвольной, ее можно задать ключевым параметром `LEN`:

```
CHARACTER(LEN = 80) :: STRING
```

Подстроки символьной константы также можно выделять как сечения массива:

- `STRING(I:J)` — `I` и `J` целые, `I < J`, выбираются элементы с `I`-ого по `J`-ый; если один из индексов диапазона опущен, то он заменяется крайним индексом строки с соответствующей стороны;
- `STRING(:J)` соответствует началу строки до элемента `J` — `STRING(1:J)`;
- `STRING(I:)` соответствует концу строки от элемента `I` — `STRING(I:80)`, если при выборе подстроки `I > J`, то выбирается строка нулевой длины.

Массив строк, например, страницу, можно определить как массив символьных констант:

```
CHARACTER(LEN = 80), DIMENSION(60) :: PAGE
```

На подстроку `I:J` `K`-ой строки этого массива строк можно сослаться как на `PAGE(K)(I:J)`.

Подстроку можно выделить и в буквальной константе символьного типа:

```
'asbdgetrwfcsdhjhgkjwuqgsfgx'(3:10)
```

и в символьном элементе структуры:

```
PERSON%FULLNAME(1:5)
```

Говорить о переменных символьного типа как о константах приходится потому, что вышесказанное относится к символьным наборам фиксированной длины. Оперировать строками произвольной длины можно, например, определив подходящий производный тип.

## Указатели (ссылки)

Указатели в Фортране используются как временные универсальные псевдонимы, с помощью которых можно ссылаться на различные реальные объекты определенного типа. Связь указателя с действительным объектом может быть установлена и разорвана в ходе выполнения программы. С помощью указателей можно создавать связные списки, назначать имена сечениям массивов и векторным индексам. Под указатель можно выделить область динамической памяти.

Переменная является указателем, если в ее описании присутствует атрибут `POINTER`:

```
REAL, POINTER :: A
INTEGER, POINTER, DIMENSION(:) :: INDEX
COMPLEX, POINTER, DIMENSION(:,:) :: MATRIX
```

Для объявления указателя достаточно определить тип объекта и, если предполагается создать указатель на массив, то и ранг массива. Границы массива указывать не обязательно. Так, например, с помощью ссылки мы можем получить динамический массив вещественного типа (листинг 2.5).

**Листинг 2.5. Пример использования ссылки для создания динамического массива**

```
PROGRAM pointer_demo
 REAL, DIMENSION(:, :), POINTER ::A
 INTEGER N, S

       WRITE(*, *), ' Задайте размер матрицы '
       READ(*, *), N
              ALLOCATE(A(N, N), STAT = S)
                     CALL RANDOM_NUMBER(A)
                     WRITE(*, *) A
                     DEALLOCATE(A, STAT = S)

END PROGRAM pointer_demo
```

Указатель может входить в состав производного типа, в том числе, это может быть указатель на объект описываемого типа. В листинге 2.6 приведен пример цепочки (связного списка) цветных точек (`COLOR_POINT`) — переменных производного типа, упомянутых в разделе "Массивы встроенного типа" этой главы.

**Листинг 2.6. Пример ссылки в составе производного типа**

```
TYPE BEADS
       REAL, DIMENSION(3)   :: COORDS
       INTEGER, DIMENSION(3) :: COLOR_RGB
       TYPE(BEADS), POINTER :: NEXT
END TYPE BEADS
```

Явное задание массивов указателей не допускается, но аналогичный эффект может быть достигнут путем описания массива производного типа с компонентом-указателем.

## Динамические структуры данных

Одним из самых серьезных недостатков ранних версий Фортрана, включая Фортран 77, было отсутствие возможности использования динамической памяти. Память под массивы должна была выделяться на стадии компиляции, фактически при описании массивов. В Фортране 90 этот недостаток был исправлен, теперь в языке допускаются динамические массивы, размер

которых, и соответственно область памяти, определяются в процессе выполнения программы. Динамическая память также может быть выделена под указатель. Оператор ALLOCATE, выделяющий область памяти для объекта, применим только к массивам, в описании которых присутствует атрибут ALLOCATABLE, и к указателям.

## Вопросы и задания для самостоятельной работы

1. Перечислите все встроенные типы Фортрана.
2. Какие типы относятся к числовым?
3. Дайте определение разновидности типа.
4. Существуют ли разновидности логического типа?
5. Определите параметр разновидности вещественного типа, соответствующий типу DOUBLE PRECISION.
6. Определите, к какому типу относятся следующие выражения, если они являются буквальными константами:

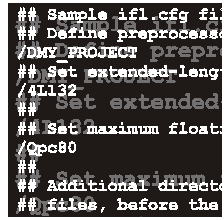
```
'TREE' -3.8   476    3.7 - 4   1_2    '(5, -6)'
0.0001E-10    Z10    B'04'   Z'AF'  (0.4, 7)      .TRUE._1
'I CAN''T'    5 1    'ОБ'    'ЕКТ'    E5         1.D-10 0_4
```

7. Опишите производный тип, представляющий разноцветные (RGB, Red, Green, Blue, — красный, зеленый, синий) точки на плоскости.
8. Напишите программу, определяющую стандартную разновидность типа целого, вещественного, символьного и логического типов.
9. Опишите производный тип, указывающий номер следующего элемента в символьном массиве.
10. Опишите производный тип, представляющий строку переменной длины.
11. Определите, какие выражения, использующие двумерный массив A(2:21, 10), правильны, а какие — нет:

```
A(:, 1)   A(1, :)   A(/2, 5, 7, 11/, 3)   A(3,:)(4)   A(:,4:11)
```

12. В следующих массивах выберите пятый, десятый и последний по счету элементы:

```
INTEGER, DIMENSION(11) :: ID
REAL, DIMENSION(-5:27) :: R
COMPLEX, DIMENSION(10, 10) :: Z
REAL, DIMENSION(5, -3:2, 4) :: V
```




# Глава 3

# Операторы описания

Операторы описания, присутствующие в явном или неявном виде, содержат информацию обо всех объектах, использующихся в программе. По правилам Фортрана блоки описания всегда помещаются в начале программного компонента — основной программы, подпрограммы или модуля. В предыдущей главе уже встречались фрагменты блоков описания, по которым можно было составить некоторое представление о структуре операторов описания, определениях типа и атрибутах. Теперь мы рассмотрим операторы описания подробно.

## Неявное определение типа. Оператор *IMPLICIT*

Фортран не относится к языкам со *строгим контролем типа*. То есть явное определение типа каждой переменной по правилам Фортрана не требуется. Объект, тип которого не определен явно ни в одном из операторов описания, типизируется по правилам *неявного определения типа*. Правила неявного определения типа либо подразумеваются по умолчанию, либо вводятся оператором `IMPLICIT`. Предшествовать оператору `IMPLICIT` в программном компоненте могут только операторы `USE`, `PARAMETER` и `FORMAT`.

Правила неявного определения типов устанавливают тип объекта по его имени. Предполагается, что все объекты программного компонента, чьи имена начинаются с букв `I`, `J`, `K`, `L`, `M` и `N`, относятся к целому типу стандартной разновидности. Объекты, имена которых начинаются со всех остальных букв — `A`, `B`, ..., `H`, `O`, `P`, ..., `Z`, — относятся к стандартной разновидности вещественного типа. В программе, приведенной в листинге 3.1, все переменные типизируются неявным образом — в ней нет ни одного оператора описания.

**Листинг 3.1. Пример использования неявного описания**

```
PROGRAM imdesc
       j = 3
       k = 6
```

```
       i = j + k
       f = 0.5 + 1.3**i
 WRITE (*, *) ' I =', i, ' j =', j, ' k =', k,&
&' Разновидность типа I =', KIND(i)
 WRITE (*, *) ' f =', f, ' Разновидность типа f = ', KIND(f)
END PROGRAM imdesc
```

Изменить правила неявного определения типа можно с помощью оператора IMPLICIT, который устанавливает соответствие типа буквенному диапазону. Его синтаксис:

```
IMPLICIT название_типа (диапазон_букв)
```

Например:

```
IMPLICIT REAL(SELECTED_REAL_KIND(10)) (P - S)
IMPLICIT COMPLEX(KIND(0.0D0)) (C)
IMPLICIT TYPE(POINT) (U - Z)
```

В результате действия этих операторов все переменные и процедуры-функции, чьи имена начинаются с букв P, Q, R и S, будут отнесены к вещественному типу с параметром разновидности, определяемым функцией SELECTED_REAL_KIND(10); все объекты, имена которых начинаются с буквы C, будут причислены к комплексному типу "двойной" точности; все объекты, имена которых начинаются с букв диапазона U - Z, будут отнесены к производному типу POINT (если, конечно, этот тип доступен). В листинге 3.2 приведен пример объявления всех имен в диапазонах (A - H, O - Z) вещественными двойной точности.

**Листинг 3.2. Пример изменения правил неявного описания**

```
PROGRAM imdesc1
IMPLICIT REAL(KIND(0.0D0)) (A - H, O - Z)
       J = 3
       K = 6
       I = J + K
       f = 0.5 + 1.3**I
 WRITE (*, *) ' I =', I,' J =', J, ' K =', K,&
&' Разновидность типа I =', KIND(I)
 WRITE (*, *) ' f =', f, ' Разновидность типа f =', KIND(f)
END PROGRAM imdesc1
```

При неявном определении типа переменная, не описанная явным образом, без "лишнего шума" получает тип, соответствующий первой букве своего имени. Таким образом, опечатка в имени переменной может пройти совер-

шенно незамеченной. Небольшие замены в программе из листингов 3.1 или 3.2 не вызовут ошибок компиляции, но приведут к совершенно другому результату (листинг 3.3).

**Листинг 3.3. Пример ошибки при неявном описании**

```
PROGRAM imdesc_error
       j = 3
       k = 6
 !Опечатка ↓
        i = l + k
       f = 0.5 + 1.3**i
 WRITE (*, *) ' I =', i, ' j =', j, ' k =', k
 WRITE (*, *) ' f =', f
END PROGRAM imdesc_error
```

Весьма сложная картина может получиться при наложении правил неявного описания внутренних подпрограмм и компонентов-носителей, а также при использовании модулей.

Поэтому правила неявного определения типа лучше не использовать вовсе. Отключить их можно оператором:

```
IMPLICIT NONE
```

Тогда все неописанные переменные будут отмечены компилятором, и трансляция не завершится, пока все переменные не будут описаны явно (листинг 3.4).

**Листинг 3.4. Пример использования оператора *IMPLICIT NONE***

```
PROGRAM force_desc
 IMPLICIT NONE
 ! Теперь описание обязательно
 INTEGER i, j, k
       j = 3
       k = 6
 !Ошибочное ↓ выражение не пройдет незамеченным
       i = l + k
       f = 0.5 + 1.3**i
 WRITE (*, *) ' I =', i, ' j =', j, ' k =', k
 WRITE (*, *) ' f =', f
END PROGRAM force_desc
```

Перед оператором IMPLICIT NONE могут помещаться только операторы USE и FORMAT. Другие операторы IMPLICIT в том блоке видимости, где есть IMPLICIT NONE, не допускаются.

# Структура оператора описания

Оператор описания должен определять тип объекта, а также может определять его атрибуты и начальные значения. В общем виде оператор описания выглядит как

```
определитель_типа [, список_атрибутов :: ] список_объектов
```

Здесь допустимы следующие *определители типа.*

- Одно из названий встроенных типов с необязательными назначениями параметров.
  - CHARACTER[(список_параметров_типа)].
  - COMPLEX [(KIND = значение_параметра_разновидности)].
  - INTEGER [(KIND = значение_параметра_разновидности)].
  - LOGICAL [(KIND = значение_параметра_разновидности)].
  - REAL [(KIND = значение_параметра_разновидности)].
- Указание ранее описанного производного типа:

  TYPE(название_производного_типа).
- Сохраняемое для обратной совместимости указание, соответствующее вещественному типу с параметром разновидности, обеспечивающим повышенную точность:

  DOUBLE PRECISION.

Значение_параметра_разновидности должно быть одним из *инициализирующих выражений (см. разд. "Назначение исходных значений переменных" данной главы)*, имеющих допустимое для данного процессора значение.

В список_параметров_типа для типа CHARACTER могут входить:

- указание длины:

  [LEN =] значение_длины
- и/или назначение параметра разновидности:

  [KIND =] значение_параметра_разновидности

Ключевые слова LEN = и KIND = могут быть опущены, если присутствуют оба значения параметров и значение_длины стоит на первом месте.

*Атрибуты* в перечне список_атрибутов могут принимать следующие значения:

- ALLOCATABLE;
- DIMENSION(список_экстентов);

- EXTERNAL;
- INTENT(спецификация_входа/выхода);
- INTRINSIC;
- OPTIONAL;
- PARAMETER;
- POINTER;
- PRIVATE;
- PUBLIC;
- SAVE;
- TARGET.

В одном операторе любой атрибут не может появляться более одного раза. Смысл атрибутов различен, и некоторые атрибуты взаимно исключают друг друга (например, INTRINSIC и EXTERNAL, POINTER и TARGET, POINTER и ALLOCATABLE для массивов). Объект с атрибутом POINTER, например, не может иметь атрибутов INTRINSIC, EXTERNAL и PARAMETER, так как их свойства несовместимы.

Список_объектов 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 объекта 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 PARAMETER присутствует в списке атрибутов оператора присваивания, то и назначение начальных значений должно производиться тем же оператором.

Примеры:

```
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, оператор описания принимает вид:

```
определитель_типа список_объектов_без_инициализации
```

Например,

```
INTEGER INDEX(3), I, J, K
```

Инициализация объектов (кроме именованных констант) в таком случае 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
```

Однако при таком "сокращенном" операторе описания назначение атрибутов объектам выполняется 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. Пример неудачного назначения атрибутов приведен в листинге 3.5.

**Листинг 3.5. Пример ошибки при разделении описания и назначения атрибутов**

```
PROGRAM err_desc1
 IMPLICIT INTEGER(C - F)
 PARAMETER (I = 3, PI = 3.14159265, E = 2.718282)
       J = 2 * I + 4
       R = 6 * E + PI
       C = I * E
 WRITE (*, *) ' C =', C,' R =', R
 WRITE (*, *) ' E =', E
END PROGRAM err_desc1
```

# Определение производных типов

Определение производного типа содержит комплекс операторов описания составляющих его компонент и должно размещаться в блоке описания перед описаниями этих элементов. Самым подходящим местом для размещения определений производных типов может быть модуль, включаемый в остальные программные компоненты оператором USE. Общий вид оператора определения производного типа:

```
TYPE [ [, атрибут_доступа ] :: ] имя_типа
       [ PRIVATE ]
       [SEQUENCE]
```

```
        оператор_описания_компонента
        [оператор_описания_компонента]…
        …
        [оператор_описания_компонента]
END TYPE [имя_типа]
```

где имя_типа не должно совпадать с именами встроенных типов.

Например:

```
TYPE, PRIVATE :: VARIATION
        INTEGER LENGTH
        CHARACTER*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 (листинг 3.4).

**Листинг 3.4. Описание производного типа**

```
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 компонент, ни генерация структур данного типа.

Атрибут SEQUENCE используется для "совмещения" объектов производных типов посредством ассоциированной памяти — прием, относящийся к нере-

комендуемым свойствам языка. Более подробно вопрос об ограничении доступа будет рассмотрен в *гл. 7*.

# Атрибуты

Свойства объектов, не связанные с их типом, назначаются атрибутами. Так, атрибут `PARAMETER` определяет именованную константу, атрибут `DIMENSION` с набором экстентов описывает массив, указатель имеет атрибут `POINTER`, указатель массива описывается парой атрибутов `POINTER`, `DIMENSION`. Один и тот же объект может иметь несколько атрибутов, но далеко не все атрибуты совместимы.

Все атрибуты могут назначаться операторами назначения атрибутов, имена которых совпадают с именами назначаемых атрибутов. Операторы назначения атрибутов размещаются в блоках описания данных наравне с операторами описания. В некоторых случаях использование оператора назначения атрибута возможно, но нежелательно. Например, назначение атрибутов `POINTER`, `ALLOCATABLE` и `TARGET` лучше не отрывать от описания объектов. Атрибуты `PRIVATE`, `PUBLIC` и `SAVE` могут назначаться одноименными операторами без списка объектов — тогда они назначаются всем объектам области видимости. Атрибуты `EXTERNAL` и `INTRINSIC` разумнее назначать только операторами.

## Атрибут *PARAMETER*. Именованные константы и константные выражения

Именованные константы используются тогда, когда необходимо многократно в разных местах программы обращаться к одной и той же постоянной величине, к ряду таких величин или значениям, производным от них. Хорошим примером такого обращения являются параметры разновидности буквальных констант числовых типов, физические или математические константы (например, числа e и $\pi$, скорость света, ускорение свободного падения и т. п.). Чтобы значение такого объекта сохранялось неизменным, нужно защитить его от случайного переназначения. Использование буквальной константы в качестве такой величины приведет к пересмотру множества операторов и выражений, если понадобится изменить значение постоянной. Для поддержания объектов такого рода применяются именованные константы. Именованная константа должна быть обязательно инициализирована при описании. Появление именованной константы в левой части оператора присваивания квалифицируется как ошибка компиляции. Описанное здесь поведение именованных констант обеспечивается заменой на этапе компиляции всех вхождений именованной константы ее значением, инициализированным в блоке описания.

Именованная константа определяется атрибутом PARAMETER, который может появляться:

- в списке атрибутов оператора описания:

```
INTEGER, PARAMETER :: NUM = 10;
```

- в операторе назначения атрибута (не рекомендуется):

```
INTEGER NUM, I, J, K
PARAMETER (NUM = 100)
```

Именованная константа может быть массивом:

```
INTEGER, PARAMETER, DIMENSION(3) :: DIMS = (/0, 1, 2/),
```

или структурой, т. е. объектом производного типа:

```
TYPE POINT
       REAL X, Y
       INTEGER I
END TYPE POINT
TYPE(POINT), PARAMETER :: ORIGIN = POINT(0., 0., 1)
```

Обязательным элементом определения именованной константы, как в операторе описания, так и в операторе назначения атрибута, является ее инициализация в *списке_объектов* после двойного двоеточия:

```
REAL, PARAMETER :: PI = 3.14159265,
```

или в скобках после оператора PARAMETER:

```
REAL PI
PARAMETER (PI = 3.14159265 )
```

Ввиду обязательности инициализации, в описании именованных констант символьного типа можно использовать подразумеваемую длину:

```
CHARACTER(LEN = *), PARAMETER :: TEXT = 'Длинная строка'
```

Так как значение именованной константы подставляется в выражения на этапе компиляции, их можно использовать в *константных выражениях* последующих операторов описания, например, в качестве экстентов массивов:

```
INTEGER, PARAMETER :: NUM = 100
REAL X(NUM), Z(NUM, NUM), Y(2 * NUM)
CHARACTER(LEN = NUM) TEXT(NUM**2)
```

Константным называется такое выражение, в котором все операции встроенные, а операндами могут быть:

- константа или подобъект константы, в котором все индексные выражения представлены константными выражениями;
- конструкторы структур или массивов, в выражениях которых используются только константные выражения или переменные неявных циклов;

- обращение к встроенной элементной или преобразующей функции, аргументы которой являются константными выражениями;
- обращение к справочной функции (кроме функций PRESENT, ASSOCIATED или ALLOCATED), в котором каждый аргумент — константное выражение или явно назначенная переменная.

В инициализирующем константном выражении допускаются:

- возведение в степень с целым показателем;
- встроенные элементные функции с результатами и аргументами целого или символьного типа;
- преобразующие функции REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER и TRIM.

Например:

```
INTEGER, PARAMETER :: PAM = 12, DOUBLE_REAL = KIND(0.D0)
INTEGER, PARAMETER :: LONG = SELECTED_REAL_KIND(PAM)
```

Объект, отмеченный атрибутом PARAMETER, не может быть формальным параметром, не может быть адресатом ссылки или указателем, не может быть выделяемым объектом; и поэтому с ним несовместимы атрибуты POINTER, TARGET, ALLOCATABLE. Трудно представить, зачем может понадобиться совмещение атрибута PARAMETER с атрибутами INTRINSIC или EXTERNAL. Именованная константа модуля может иметь атрибуты PUBLIC или PRIVATE. Единственный атрибут, уверенно совмещаемый с атрибутом PARAMETER, — это DIMENSION.

## Атрибуты *DIMENSION* и *ALLOCATABLE*. Описание массивов

Атрибут DIMENSION описывает массивы. Параметром этого атрибута должен быть список экстентов описываемых массивов в виде

```
(экстент_1, экстент_2, …, экстент_n)
```

Число экстентов определяет ранг массива. Каждый экстент выражается как

```
[ нижняя_граница : ] верхняя_граница
```

Например:

```
REAL, DIMENSION(0:10, 2, -3:3,11) :: FGRID
```

Если значение нижней границы опущено, ее значение полагается равным 1. Двоеточие, разделяющее нижнюю и верхнюю границы измерения, при отсутствии указания нижней границы тоже может быть опущено.

В некоторых случаях список экстентов сводится к списку разделенных запятыми двоеточий, число которых равно рангу массива. Только число экстентов задается при описании указателя массива:

```
REAL, POINTER, DIMENSION( :, :, :, : ) :: TO_GRID
```

и при объявлении динамического массива:

```
REAL, ALLOCATABLE, DIMENSION(:, :) :: BE_LATER
```

Динамический массив — это массив, размер которого определяется при выполнении программы, тогда же происходит и выделение памяти под него. Экстенты динамического массива в блоке описания еще неизвестны, и атрибут DIMENSION определяет только ранг "будущего" массива (листинг 3.5).

**Листинг 3.5. Описание динамического массива**

```
PROGRAM dyn_array
IMPLICIT NONE
 INTEGER SIZE
 REAL, ALLOCATABLE, DIMENSION(:) ::array
 WRITE(*, *) 'SIZE?'
 READ(*, *) SIZE
 IF(SIZE > 0) ALLOCATE(array(SIZE))
...
 IF(ALLOCATED(array)) DEALLOCATE(array)
END PROGRAM dyn_array
```

Оператор описания массива может не содержать атрибута DIMENSION, форма массива может быть указана непосредственно после его идентификатора:

```
REAL X(10, 20, 30), Y(100), Z(2, 300, 2, 4)
```

Как уже упоминалось, такая форма удобна для одновременного описания массивов разной формы, но нежелательна, если описываемые массивы имеют какие-либо атрибуты.

Атрибут ALLOCATABLE несовместим с атрибутом POINTER; его нельзя назначать формальным параметрам и совмещать с атрибутом PARAMETER.

## Атрибуты *POINTER* и *TARGET*

Атрибут POINTER присваивается объектам, используемым как ссылки (указатели) на другие объекты. Для объектов-ссылок требуется определение типа адресата и его ранга, если предполагаемый адресат является массивом. Примеры назначения атрибута POINTER приводились неоднократно:

```
INTEGER, DIMENSION(:, :), POINTER :: PNTR
```

Однако не всякий двумерный массив целого типа может стать адресатом указателя PNTR. Для оптимизации работы компиляторов (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 DOUBLE, описанный как

```
INTEGER, DIMENSION(10, 10), TARGET :: DOUBLE
```

в дальнейшем позволяет прикрепить указатель PNTR, например, к сечению DOUBLE(1:5, 2):

```
PNTR => DOUBLE(1:5, 2)
```

## Атрибуты *PUBLIC* и *PRIVATE*

Атрибуты PUBLIC и PRIVATE используются для управления доступом к объектам, принадлежащим модулям *(см. гл. 5)*. Посредством оператора USE модуль может стать частью любой другой программы, компоненты которой получают 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 — модульные переменные могут использоваться, например, для передачи информации из одной модульной процедуры в другую. В этом случае, внешнее изменение этих переменных может привести к непредусмотренным искажениям результатов работы процедур. Атрибут PRIVATE, назначаемый объектам модуля, ограничивает область их доступности рамками модуля, которому они принадлежат.

Атрибуты доступа могут быть назначены в списке атрибутов при описании объектов:

```
INTEGER, PRIVATE :: A, B, C
INTEGER, PUBLIC :: F, G, H,
```

или установлены отдельными операторами назначения атрибута:

```
INTEGER A, B, C, F, G, H
PRIVATE A, B, C
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 3.6 иллюстрирует применение атрибутов PRIVATE и PUBLIC на примере модульных переменных.

**Листинг 3.6. 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
```

## Атрибут *SAVE*. Сохранение значений локальных переменных

Значения локальных переменных после выхода из процедуры могут не сохраняться. После выхода из процедуры локальные переменные оказываются в состоянии неопределенности; ссылки, прикрепленные к ним, также приходят в состояние неопределенности. Если в подпрограмме используется переменная из модуля *(см. гл. 5)*, она также приходит в состояние неопределенности. Атрибут `SAVE` гарантирует сохранение значения локальной или модульной переменной между вызовами процедуры. Атрибут `SAVE` в применении к указателю означает сохранение состояния привязанности после выхода из процедуры, а в применении к выделяемому массиву — сохранение его значений и состояния выделенности. Так, назначение атрибута `SAVE` позволяет сохранять внутри процедуры сумму значений формального параметра по всем обращениям к процедуре (листинг 3.7).

**Листинг 3.7. Сохранение значения локальной переменной**

```
REAL FUNCTION COUNT(X)
 REAL X
 REAL, SAVE :: A
 INTEGER :: COUNTER = 1
IF(COUNTER .EQ. 1) THEN
       A = X
ELSE
       A = A + X
END IF
 COUNTER = COUNTER + 1
 COUNT = A
END FUNCTION COUNT
```

В переменной COUNTER сохраняется число обращений к подпрограмме, несмотря на отсутствие в его описании атрибута SAVE: всем локальным переменным, инициализирующимся в подпрограмме, атрибут SAVE назначается автоматически. В рекурсивной подпрограмме значение сохраняемой переменной остается общим для всех уровней вызова.

Атрибут SAVE не может назначаться формальным параметрам, результатам функций и автоматическим объектам.

Атрибут SAVE может назначаться оператором SAVE:

```
SAVE [ [ :: ] список_имен_переменных ]
```

Если список_имен_переменных отсутствует, все доступные переменные получают атрибут SAVE. Появление атрибута или оператора SAVE в главной программе не запрещено, но не приводит ни к какому действию.

## Атрибуты *OPTIONAL* и *INTENT*. Описание формальных параметров

Атрибуты OPTIONAL и INTENT имеют смысл и применяются только к формальным параметрам процедур. Атрибут OPTIONAL отмечает *необязательные* формальные параметры, т. е. такие, которые могут не присутствовать среди фактических параметров при вызове процедуры. Атрибут OPTIONAL можно назначать в операторе описания или оператором назначения атрибута:

```
SUBROUTINE SUBTRACE( A, B, C, D, E, F)
REAL, OPTIONAL, DIMENSION(:) :: E, F
REAL :: A, B, C
INTEGER :: D
OPTIONAL :: C, D
```

Формальный параметр, имеющий атрибут OPTIONAL, не должен иметь никаких других атрибутов.

Атрибут INTENT указывает назначение формального параметра: формальный параметр может быть входным, выходным и универсальным. Этим назначениям соответствуют параметры атрибута IN, OUT и INOUT. Объявление назначения параметра способствует улучшению контроля за их использованием и повышает надежность процедуры и всей программы в целом. Атрибут INTENT назначается или в атрибутах оператора описания, или непосредственно оператором назначения атрибута:

```
SUBROUTINE SUBBY(A, B, C, D, E, F)
REAL, INTENT(IN) :: A,  B, C
INTEGER, DIMENSION( :, : ) :: D
COMPLEX :: E, F
```

```
INTENT(INOUT) :: D, E
INTENT(OUT) :: F
```

Атрибут INTENT нельзя применять к формальным процедурам и ссылкам.

## Атрибуты *INTRINSIC* и *EXTERNAL*

Атрибут EXTERNAL указывает на принадлежность объекта к внешним или формальным процедурам. Атрибут может назначаться как в операторе описания, так и отдельным оператором:

```
EXTERNAL список_внешних_имен
```

Указание атрибута EXTERNAL заставляет компилятор воспринимать имена функций как внешних процедур, встроенные функции с такими же именами становятся недоступны для компилятора.

Атрибут INTRINSIC назначается объекту-имени функции для того, чтобы компилятор воспринимал функцию с этим именем как встроенную, исключая поиск внешних функций с таким же именем. Атрибут может назначаться в операторе описания и отдельным оператором:

```
INTRINSIC список_встроенных_функций
```

Пример назначения атрибутов INTRINSIC и EXTERNAL приведен в листинге 3.8.

**Листинг 3.8. Атрибуты *INTRINSIC* и *EXTERNAL***

```
PROGRAM fcall
IMPLICIT NONE
 REAL X, S
 REAL, INTRINSIC :: SIN
 REAL, EXTERNAL :: COUNT
 X = 0.0
 DO
 IF(X < 1.0) THEN
       S = COUNT(SIN(X)) !SIN — встроенная, COUNT — из листинга 3.7
       X = X + 0.1
 ELSE
       EXIT
 END IF
 END DO
END PROGRAM fcall
```

Назначение атрибута EXTERNAL заставит компилятор искать внешнюю функцию с именем COUNT (она приведена в листинге 3.7) и игнорировать встроенную функцию с тем же именем *(см. гл. 10)*.

# Назначение исходных значений переменных

Операторы описания можно использовать для задания исходных значений переменных. Инициализация именованных констант в операторах описания обязательна. Нельзя инициализировать формальные параметры, ссылки, выделяемые массивы, автоматические объекты, результаты функций и ассоциированные переменные. Мы уже рассматривали назначение переменных и именованных констант в операторах описания: оно выполняется посредством встроенного присваивания, вызова конструкторов массивов или структур и привлечения неявных циклов.

```
REAL :: A = 0.0, B = 1.0
REAL, DIMENSION(3) :: F=(/ 0.6, 0.54, 0.01/)
INTEGER, DIMENSION(300) :: (C(I) = I, I = 1, 300)
```

Все, даже частично, инициализированные переменные автоматически получают атрибут SAVE. При инициализации в операторе описания, объекты должны инициализироваться полностью. Кроме операторов описания, исходные значения могут назначаться оператором DATA. Оператор DATA позволяет выполнять частичную инициализацию объектов. Оператор DATA имеет вид:

```
DATA список_объектов /список_значений/
[[,]список_объектов /список_значений/]
```

где список_объектов — это список переменных и неявных циклов, а список_значений — список скалярных констант и конструкторов структур. Например:

```
REAL :: C, G, F
REAL, DIMENSION(5) :: A
INTEGER :: I, J, K, L
DATA A, C, G, F /5.6, 7.8, 2.1, 4.7, 5.9, 1., 2., 5. /,&
&I, J, K, L /2, 5, 7, 3/
```

В каждом списке значений должно быть столько констант, сколько скалярных значений содержится в списке объектов. Массив A состоит из 5 вещественных чисел, следующие 3 числа являются значениями вещественных скалярных переменных C, G и F. Идущие подряд одинаковые значения в списке значений можно записывать с коэффициентом повторения:

```
DATA I, J, K, L /4 * 1/
```

Здесь все переменные I, J, K и L целого типа получают значение 1.

Массивы можно инициализировать несколькими способами:

- полностью, задавая все значения одинаковыми:
  - `REAL, DIMENSION(5, 5) :: R`
  - `DATA R /25 * 2.5/`

- частично, назначая значения отдельным элементам и сечениям:
  - `DATA R(1, 1), R(2, 2), R(3, 3), R(4, 4), R(5, 5) / 5 * 1.0/`
  - `DATA R(3:5, 1:2) /6*0.0/`
- с использованием неявного цикла, если с его помощью можно описать порядок расположения инициализируемых элементов:

```
DATA ((R(I, J), I = 1, 3, 5), J = 1, 3, 5 ) /9 * 3.0/
```

Общий вид неявного цикла интерпретируется по аналогии с конструкцией `DO`, но область видимости переменной неявного цикла ограничена самим циклом:

```
(список, переменная = выражение, выражение [, выражение])
```

В качестве переменных в выражениях могут использоваться только переменные охватывающих неявных циклов, в качестве индексов компонентов списка — переменные того же или охватывающих неявных циклов. Все используемые в неявном цикле операции должны быть встроенными.

Чтобы инициализировать в операторе `DATA` объект производного типа, нужно использовать конструктор соответствующей структуры с инициализирующим выражением для каждого компонента или воспользоваться оператором извлечения компонентов структуры с инициализирующим выражением:

```
TYPE POINT
   REAL X, Y
   INTEGER I
END TYPE POINT
.....
TYPE(POINT) :: P1, P2
DATA P1 / POINT(1.0, 1.0, 1)/, P2 %X /2/
```

Тип каждой переменной к моменту ее инициализации в операторе `DATA` либо описан заранее в операторах описания, либо определяется в операторе DATA по правилам неявного описания; в последнем случае дальнейшее описание инициализированной переменной должно подтвердить ее тип, установленный правилами неявного описания. Переменная или любая ее часть не может быть инициализирована в одном блоке видимости более одного раза.

# Вопросы и задания для самостоятельной работы

1. Напишите операторы описания для представления следующих объектов.
   - Набора чисел, представляющих продолжительность всех месяцев года в днях.

- Зарплату сотрудников небольшого (20 человек персонала) отдела маркетинга процветающей коммерческой фирмы.
- Матрицу размером $50 \times 50$, заполненную случайными числами в диапазоне от 0 до 1.
- Карточку персонального учета, хранящуюся в отделе кадров.
- Набор из 40 двухпозиционных переключателей.
- Пакет карточек персонального учета в отделе кадров для сотрудников упоминавшегося выше отдела маркетинга.

2. Там, где это возможно, дополните описания объектов из предыдущего упражнения инициализирующими выражениями и, если это необходимо, атрибутами, гарантирующими постоянство значения в области видимости.
3. Совместимы ли следующие пары атрибутов. Если нет, то почему?
   - `ALLOCATABLE`, `TARGET`.
   - `TARGET`, `POINTER`.
   - `ALLOCATABLE`, `SAVE`.
   - `POINTER`, `DIMENSION`.
   - `PRIVATE`, `DIMENSION`.
   - `PARAMETER`, `ALLOCATABLE`.
   - `ALLOCATABLE`, `INTRINSIC`.
   - `POINTER`, `EXTERNAL`.
4. Чем отличаются объекты следующих двух операторов описания?

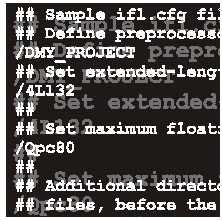
```
REAL :: PI = 3.141592
```

и

```
IMPLICIT REAL(P)
PARAMETER (PI = 3.141592)
```

5. При каком из описаний переменной `PI`, приведенном в предыдущем упражнении, оператор `PI = 2.718282` будет воспринят компилятором как ошибочный?

# Глава 4




# Операторы присваивания. Выражения

Основу любой вычислительной программы составляют выражения — последовательность операций, которые компьютер должен произвести, и присваивания — сохранение результатов выполнения выражений в переменных для использования в последующих действиях. Последовательное вычисление выражений и выполнение присваиваний в блоках управляющих конструкций, по сути, и является программой. В этой главе мы рассмотрим правила для составления выражений и присваиваний.

Выражение образуется из операндов и знаков операций, объединенных по правилам синтаксиса Фортрана. Все выражения можно представить как композицию бинарных (с двумя операндами) и унарных (с одним операндом) простых выражений. Простое бинарное выражение имеет вид:

```
операнд операция операнд
```

например,

```
A * B
```

или

```
Z — S.
```

Простое унарное выражение имеет вид:

```
операция операнд
```

например,

```
-X
```

или

```
+5.
```

В качестве операндов могут использоваться буквальные и именованные константы, переменные или функции, и сами выражения. Операции могут быть *встроенными* (всегда доступными в языке) или *задаваемыми* (переопределенными или перегруженными) *(см. разд. "Конструкторы структур и определение скалярных операций для переменных производного типа" настоящей гл.).*

Таким образом, выражение представляет собой последовательность чередующихся операндов и операций:

```
операнд операция операнд операция операнд...
```

В выражении, в котором нет скобок, операции выполняются друг за другом слева направо в соответствии со своим приоритетом. Так, в выражении:

```
A * B + C / D
```

сначала выполняются умножение и деление, а затем — сложение.

Исключением из этого правила является возведение в степень, выполняющееся справа налево.

Части выражения, заключенные в скобки, получают повышенный приоритет и выполняются раньше остальных операций. В выражении:

```
A * (B + C) / D
```

сначала будет выполнено сложение, затем — умножение и деление.

Если в выражении скобок нет, допускается неявная расстановка скобок процессором в тех случаях, когда изменение очередности исполнения повысит эффективность вычислений, не изменяя общего результата выражения. Если, например, деление выполняется намного медленнее умножения, то выражение:

```
X / Y / Z
```

эффективнее исполнять как

```
X / (Y * Z)
```

Изменения в точности вычислений при этом не учитываются. Поэтому там, где порядок вычислений важен для точности результата, следует расставлять скобки, не полагаясь на общие правила очередности выполнения операций.

# Определенные и неопределенные переменные

В *гл. 3*, в описании атрибута сохранения значения переменной, уже упоминалось о состоянии неопределенности, в котором оказывается локальная переменная, не имеющая атрибута `SAVE`, после выхода из той процедуры, где она описана. Что такое состояние неопределенности? Переменная, тип и атрибуты которой заданы операторами описания и операторами назначения атрибутов, существует, но не имеет никакого конкретного значения, т. е. *не определена*. Результат обращения к такой переменной непредсказуем, так как использовать несуществующее значение нельзя. Рассмотрим простой пример (листинг 4.1).

**Листинг 4.1. Пример обращения к переменной, находящейся в состоянии неопределенности**

```
PROGRAM var_undef
 INTEGER, PARAMETER :: A = 1
 REAL C
 PRINT *, ' A =', A, ' C =', C
END PROGRAM var_undef
```

Переменная C не получила никакого значения, но программа пытается использовать ее. Скорее всего, эта программа оттранслируется и успешно проработает, напечатав строку:

```
A = 1 C = 0,
```

так как большинство компиляторов инициализируют неинициализированные числовые переменные нулями нужного типа. Однако полагаться на компилятор в таком случае было бы большой ошибкой.

Обычный способ сделать переменную определенной — назначить ей конкретное значение с помощью оператора присваивания:

```
REAL C
C = 0.25
```

Теперь переменная C имеет значение, и ее можно использовать, не опасаясь непредсказуемого результата.

Составной объект находится в состоянии определенности тогда, когда определены все его подобъекты в отдельности. Массив, например, только тогда считается определенным, когда определены все его элементы; переменная производного типа определена только тогда, когда определены все ее компоненты, а текстовая переменная — когда определены все составляющие ее символы.

Состояние определенности ссылки связано с ее прикрепленностью к адресату. Ссылка, никогда не прикреплявшаяся ни к какому адресату, находится в состоянии *неопределенности* (это исходное состояние указателя). Ссылка, прикрепленная к адресату, находится в состоянии *связанности* (прикрепленности). Ссылка, освобожденная от связи с адресатом, *откреплена*. В двух последних случаях состояние ссылки может быть проверено встроенной функцией ASSOCIATED *(см. гл. 10)*. Но даже если ссылка прикреплена к адресату, в состоянии неопределенности может находиться адресат ссылки.

Состояние переменных в процессе исполнения программы может меняться, — основной причиной этого является освобождение памяти при возврате управления, что происходит при выходе из подпрограмм. Если хотя бы один элемент массива становится неопределенным, в состоянии неопределенности оказывается весь массив.

# Скалярные числовые выражения и скалярное числовое присваивание

Числовым выражением называется такое выражение, все операнды которого относятся к числовым типам — целому, вещественному или комплексному, а все операции являются числовыми встроенными операциями. К встроенным числовым операциям Фортрана относятся (в порядке убывания приоритета):

- ** — возведение в степень;
- *, / — умножение, деление;
- −, + — вычитание, сложение.

Минус (−) и плюс (+) используются также как знаки унарных операций:

`-X` или `+Y`.

Приоритет унарных операций выше приоритета бинарных операций, обозначаемых тем же знаком. Знак унарной операции не должен следовать непосредственно за знаком бинарной операции, при необходимости, простое унарное выражение следует заключать в скобки. Например, если нужно произвести умножение на результат унарного отрицания $x(-y)$ или возвести в степень со знаком минус в показателе $x^{-y}$, простое унарное отрицание необходимо заключить в скобки, т. е. записать:

```
x * (-y) и x**(-y)
```

В выражении без скобок в первую очередь выполняются операции возведения в степень. Как уже упоминалось, порядок выполнения 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`.

Тип и разновидность типа унарных выражений совпадает с типом и разновидностью операнда унарного выражения. Числовые типы и разновидности сложных выражений определяются типом и разновидностью типа того операнда, который имеет наиболее мощный тип. Фортран допускает выражения со смешанным типом операндов и не требует от программиста явного приведения типов. Все операнды автоматически приводятся к типу и разновидности операнда, имеющего самые мощные тип и разновидность (исключение составляет возведение вещественных и комплексных величин в целую степень). Например, при вычислении выражения `X * I`, где `X` имеет тип `REAL` с разновидностью `k`, а `I` — переменная целого типа, будет выполнено приведение типа переменной `I` к типу и разновидности переменной `X`. Фактически, будет выполняться выражение:

```
X * REAL(I, KIND(X)),
```

где `REAL(I, KIND(X))` — встроенная функция приведения целого к вещественному типу с заданной разновидностью типа *(см. гл. 10)*. В табл. 4.1 и 4.2 приведены типы результатов выполнения операций в зависимости от типов операндов. Встроенные функции `REAL`, `CMPLX` и `KIND` описаны в *гл. 10*, а выражение:

```
max(KIND(a), KIND(b))
```

означает выбор параметра разновидности, обеспечивающего более высокую точность.

***Таблица 4.1.*** *Типы и разновидности типов результатов выражений x + y, x – y, x * y, x / y в зависимости от типов операндов*

| Тип x | Тип y | Фактически используемое значение x | Фактически используемое значение y | Тип результата | Разновидность типа результата |
|---|---|---|---|---|---|
| INTEGER | INTEGER | x | y | INTEGER | max(KIND(x), KIND(y)) |
| INTEGER | REAL | REAL (x, KIND(y)) | y | REAL | KIND(y) |
| INTEGER | COMPLEX | CMPLX (x, 0, KIND(y)) | y | COMPLEX | KIND(y) |
| REAL | INTEGER | x | REAL(y, KIND(x)) | REAL | KIND(x) |
| REAL | REAL | x | y | REAL | max(KIND(x), KIND(y)) |
| REAL | COMPLEX | CMPLX (x, 0, KIND(y)) | y | COMPLEX | max(KIND(x), KIND(y)) |
| COMPLEX | REAL | x | CMPLX (y, 0, KIND(x)) | COMPLEX | max(KIND(x), KIND(y)) |
| COMPLEX | INTEGER | x | CMPLX (y, 0, KIND(x)) | COMPLEX | KIND(x) |
| COMPLEX | COMPLEX | x | y | COMPLEX | max(KIND(x), KIND(y)) |

***Таблица 4.2.*** *Типы и разновидности типов результатов выражения x**y в зависимости от типов операндов*

| Тип x | Тип y | Фактически используемое значение x | Фактически используемое значение y | Тип результата | Разновидность типа результата |
|---|---|---|---|---|---|
| INTEGER | INTEGER | x | y | INTEGER | max(KIND(x), KIND(y)) |
| INTEGER | REAL | REAL (x, KIND(y)) | y | REAL | KIND(y) |

***Таблица 4.2*** *(окончание)*

| Тип x | Тип y | Фактически используемое значение x | Фактически используемое значение y | Тип результата | Разновидность типа результата |
|---|---|---|---|---|---|
| INTEGER | COMPLEX | CMPLX (x, 0, KIND(y)) | y | COMPLEX | KIND(y) |
| REAL | INTEGER | x | y | REAL | KIND(x) |
| REAL | REAL | x | y | REAL | max(KIND(x), KIND(y)) |
| REAL | COMPLEX | CMPLX (x, 0, KIND(y)) | y | COMPLEX | max(KIND(x), KIND(y)) |
| COMPLEX | INTEGER | x | y | COMPLEX | (KIND(x)) |
| COMPLEX | REAL | x | CMPLX (y, 0, KIND(x)) | COMPLEX | max(KIND(x), KIND(y) |
| COMPLEX | COMPLEX | x | y | COMPLEX | max(KIND(x), KIND(y)) |

При возведении комплексной величины в комплексную степень используется главное значение выражения:

$x^y = \exp\,(y\,(\ln|x| + i \arg x))$, где $-\pi < \arg x \le \pi$

Для данных целого типа результат деления округляется до ближайшего целого числа, абсолютная величина которого не превышает абсолютную величину истинного результата деления. Например, результатом выражения `16 / 5` является `3`, результатом деления `-6 / 4` будет `-1`, а результат выражения `2**(-3)` равен `0`, так как это выражение эквивалентно `1 / 2**3`.

В общем виде скалярное числовое присваивание выглядит так:

```
переменная = скалярное_числовое_выражение
```

Если тип *скалярного_числового_выражения* не совпадает с типом переменной или относится к другой разновидности типа, то перед выполнением присваивания производится приведение типа выражения к типу переменной по правилам, приведенным в табл. 4.3.

***Таблица 4.3.*** *Преобразование типа при выполнении присваивания* `переменная = выражение`

| Тип переменной | Присваиваемое значение |
|---|---|
| INTEGER | INT(выражение, KIND = KIND(переменная)) |
| REAL | REAL(выражение, KIND = KIND(переменная)) |
| COMPLEX | CMPLX(выражение, KIND = KIND(переменная)) |

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. Примеры вычисления выражений и выполнения присваиваний приведены в программе (листинг 4.2).

**Листинг 4.2. Примеры числовых выражений и присваиваний**

```
PROGRAM express
IMPLICIT REAL(KIND(0.0D0)) (A - H)
I = 2
F = 0.004
T = 2.7
WRITE(*, *) 'Числовое выражение I * T**I - 5.2D0**I * F имеет значение ',&
&I * T**I - 5.2D0**I * F
J = I * T**I - 5.2D0**I * F
WRITE(*, *) 'J - целого типа, J = I * T**I - 5.2D0**I * F&
& имеет значение ', J
R = I * T**I — 5.2D0**I * F
WRITE(*, *) 'R - типа REAL, R = I * T**I - 5.2D0**I * F&
& имеет значение ', R
H = I * T**I - 5.2D0**I * F
WRITE(*, *) 'H - типа DOUBLE PRECISION,',&
&' H = I * T**I - 5.2D0**I*F имеет значение ', H
WRITE(*, *) 'Определите тип числового выражения I * T**I - 5.2D0**I * F'
END PROGRAM express
```

# Скалярные операции отношения

Скалярные операции отношения используются для сравнения значений числовых и текстовых выражений с другими значениями того же типа. В Фортране поддерживаются следующие операции отношения, описанные в табл. 4.4.

***Таблица 4.4.*** *Скалярные операции отношения*

| Знак операции | Альтернативное обозначение | Название операции сравнения |
|---|---|---|
| .LT. | < | меньше |
| .LE. | <= | меньше или равно |

***Таблица 4.4*** *(окончание)*

| Знак операции | Альтернативное обозначение | Название операции сравнения |
|---|---|---|
| .GT. | > | Больше |
| .GE. | >= | больше или равно |
| .EQ. | == | равно |
| .NE. | /= | не равно |

Для значений комплексного типа действительны только операции "равно" (.EQ.) и "не равно" (.NE.). В результате выполнения выражения отношения вычисляется величина логического типа, принимающая одно из двух возможных значений — .TRUE. или .FALSE. Анализ результатов выражений отношения позволяет управлять ходом программы посредством операторов управления, которые будут рассмотрены в следующей главе.

Выражения отношения имеют вид:

```
операнд знак_операции операнд,
```

где *операндами* могут быть числовые или текстовые выражения (оба либо числовые, либо текстовые), а *знак операции* — один из знаков, перечисленных в таблице 4.4. Знак операции в выражении отношения имеет самый низкий приоритет, и, 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 и их разновидностей до выполнения сравнения производится приведение операндов к одному типу — к тому, которому принадлежит сумма операндов (правила определения типов результатов числовых выражений см. в табл. 4.1).

Сравнение величин текстового типа возможно только для величин одной разновидности, но величины разной длины сравнивать можно: более короткий из операндов дополняется пробелами (или другими символами, в зависимости от процессора) 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, они считаются равными.

# Скалярные логические выражения и присваивания

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. Значение логических операций приведены в табл. 4.5.

***Таблица 4.5.*** *Логические операции Фортрана в порядке убывания приоритета*

| Операция | Значение операции |
|---|---|
| .NOT. | Логическое отрицание |
| .AND. | Логическое пересечение (умножение, логическое "И") |
| .OR. | Логическое объединение (сложение, логическое "ИЛИ") |
| .EQV. и .NEQV. | Логические эквивалентность и неэквивалентность (логические равенство и неравенство) |

Операция .OR. является включающим логическим "ИЛИ", операция исключающего логического "ИЛИ" (XOR) в Фортране отсутствует и, при необходимости, должна конструироваться из других логических операций.

В логических выражениях и присваиваниях могут использоваться только данные логического типа, никакие другие типы в них не допускаются. 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, N, K, I, J, COND
REAL U, W, X, Y
LOGICAL, PARAMETER :: T = .TRUE., F = .FALSE.
.....
L = .NOT.J
K = L .AND. M .OR. I .AND. J
COND = W + U > X .AND. U * Y < U .OR. K
```

Параметр разновидности результата бинарных логических выражений равен параметру разновидностей операндов, если все операнды имеют одну и ту же разновидность. Для операндов различных разновидностей разновидность результата не стандартизована и зависит от процессора.

# Скалярные символьные выражения и присваивания

Для переменных символьного типа определена только одна операция — конкатенация "//", в результате которой происходит объединение двух символьных операндов в один символьный результат. С помощью конкатенации две символьные константы 'TI' и 'ME' можно объединить в одну строку:

```
'TI'//'ME'
```

Результатом объединения будет строка 'TIME'. Операнды символьных выражений обязаны принадлежать одной разновидности типа, результаты конкатенаций можно присваивать символьным переменным. Можно конкатенировать подстроки:

```
CHARACTER(LEN = 4) :: WORD1 = 'HOLE', WORD2 = 'LOOP'
CHARACTER(LEN = 4) :: WORD3
.....
WORD3 = WORD2(4:4)//WORD1(2:4)
```

Переменная WORD3 получит значение POLE.

Если длина символьного выражения меньше длины той переменной, которой оно присваивается, то выражение дополняется справа пробелами до длины переменной, стоящей в левой части присваивания:

```
CHARACTER(LEN = 10) :: RESULT
RESULT = WORD2//WORD1
```

Переменная RESULT после присваивания получит значение 'LOOPHOLE  ' и окажется в состоянии неопределенности, так как ее подстрока RESULT(9:10) не определена и не может использоваться в выражениях.

Если длина выражения превышает длину переменной, выражение будет урезано справа до нужной длины:

```
CHARACTER(LEN = 6) :: RESULT
RESULT = WORD2//WORD1
```

Значение переменной RESULT окажется равным 'LOOPHO'.

Если переменная в левой части присваивания имеет нулевую длину, то присваивание не происходит. Далее приведены еще несколько примеров логических и символьных выражений (листинг 4.3).

**Листинг 4.3. Примеры логических и символьных выражений и присваиваний**

```
PROGRAM sympress
IMPLICIT NONE
INTEGER count
```

```
LOGICAL :: c = .FALSE., f = .FALSE., g = .FALSE.
CHARACTER(LEN = 6), DIMENSION(7) :: prefix
CHARACTER(LEN = 3), PARAMETER :: day = 'day'
CHARACTER(LEN = 9) result
prefix = (/'   Sun', '   Mon', '  Tues', 'Wednes', ' Thurs',&
&'   Fri', ' Satur'/)
WRITE(*, *) 'Введите число'
READ(*, *) count
c = (count <= 7)
IF(c) THEN
       result = prefix(count)//day
       f = (result .EQ. '   Sunday')
       g = (result .EQ. ' Saturday')
ELSE
       WRITE(*, *) 'Число не должно превышать 7. Приходите в другой раз.'
       GO TO 1
END IF
IF(f .OR. g) WRITE(*, *) 'Сегодня выходной. Не работаем.'
IF(.NOT. f .AND. .NOT. g) 'Кончай курить! Пора работать.'
1  CONTINUE
END PROGRAM sympress
```

# Конструкторы структур и определение скалярных операций для переменных производного типа

Объекты производного типа — структуры — создаются из их компонентов конструкторами структур. Конструкторы структур имеют вид:

```
имя_производного_типа(список_выражений_для_всех_компонентов),
```

где список, заданный в скобках, должен определять значения всех компонентов производного типа (листинг 4.4).

**Листинг 4.4. Объявление производного типа и вызов конструктора структуры**

```
TYPE DERIVED
       INTEGER LENGTH
       CHARACTER(LEN = *) VALUE
END TYPE DERIVED
TYPE POINT
```

```
        INTEGER i
        INTEGER j
        INTEGER k
END TYPE POINT
CHARACTER(LEN = 4) : : WORD1 = 'TIME', WORD2 = 'WAIT'
TYPE(POINT) P
TYPE(DERIVED) D
P = POINT(1, 2, 3)
D = DERIVED(LEN(WORD1//WORD2), WORD1//WORD2)
```

Созданный объект производного типа может быть присвоен переменной этого типа. Такое присваивание определено автоматически, как и присваивание вида: `C = D`, где `C` — переменная того же производного типа, что и `D`. Других операций для объектов производного типа в языке не существует, однако бинарные операции и некоторые виды присваиваний для них можно определить. Для обозначения определяемой операции можно использовать знак встроенной операции или произвольное имя до 31 буквы длиной, окружаемое точками и отличное от `.FALSE.` и `.TRUE.`

Для определения бинарной операции нужно описать функцию, выполняющую ряд действий с парой объектов производного типа и возвращающую объект того же типа, а затем назначить знак бинарной операции, по которому будет производиться вызов заданной функции для операндов-переменных данного производного типа, окружающих знак операции. Так же можно определить присваивания с "приведением типов" — присваивания переменных производного типа переменным других типов и наоборот. Назначение знака операции или присваивания определяется в *интерфейсном блоке (см. гл. 7)*, который размещается в области, доступной для всех программных компонентов, использующих данный производный тип.

**Примечание**

Интерфейс — это программный компонент, содержащий заголовок одной или заголовки нескольких внешних подпрограмм. Интерфейс позволяет явно сравнивать формальные и фактические параметры процедур на этапе компиляции, что снижает вероятность появления ошибок времени исполнения. Интерфейсные блоки, как правило, размещаются в одном или нескольких модулях, включаемых в программные компоненты операторами `USE`.

Рассмотрим назначение операций и присваиваний на примере производного типа, описывающего точки на плоскости в декартовой системе координат:

```
TYPE VECTOR
       REAL X, Y
END TYPE VECTOR
```

Описание функции сложения векторов:

```
FUNCTION SUMMA(A, B)
TYPE(VECTOR) SUMMA
TYPE(VECTOR), INTENT(IN):: A, B
       SUMMA%X = A%X + B%X
       SUMMA%Y = A%Y + B%Y
END FUNCTION SUMMA
```

Атрибут `INTENT(IN)` гарантирует неизменность значений входных параметров функции после обращения к ней.

Интерфейсный блок содержит определение операции "+" для объектов типа `VECTOR`:

```
INTERFACE OPERATOR(+)
       MODULE PROCEDURE SUMMA
END INTERFACE
```

И описание функции `SUMMA`, и интерфейс `OPERATOR(+)` размещаются в модуле, который оператором `USE` включается во все программные компоненты, где планируется работа с объектами типа `VECTOR`. Тогда во всех выражениях вида:

```
T = U + W,
```

где `T`, `U` и `W` — переменные типа `VECTOR`, знак сложения будет заменен вызовом функции `SUMMA`. Фактически, вместо сложения будет производиться вызов функции:

```
T = SUMMA(U, W)
```

Предположим, что в ходе программы будет необходимо часто использовать векторы, лежащие на оси абсцисс. Тогда, чтобы избежать явного многократного конструирования новых переменных типа `VECTOR` с нулевым значением компоненты `Y`, можно описать новую функцию с одним входным параметром вещественного типа. Она будет возвращать переменную типа `VECTOR` с компонентом `Y` равным `0`:

```
FUNCTION REAL2VECTOR(X)
TYPE(VECTOR) REAL2VECTOR
REAL, INTENT(IN) :: X
       REAL2VECTOR = VECTOR(X, 0.0)
END FUNCTION REAL2VECTOR
```

Кроме того, понадобится интерфейсный блок с оператором `ASSIGNMENT`:

```
INTERFACE ASSIGNMENT(=)
       MODULE PROCEDURE REAL2VECTOR
END INTERFACE
```

После определения функции REAL2VECTOR и интерфейса, связывающего операцию присваивания с функцией, в блоках видимости этого определения можно использовать назначение:

```
TYPE(VECTOR) A
A = 10.0
```

Полный текст программных элементов, определяющих и использующих производный тип VECTOR, приведен в листингах 4.5 и 4.6.

**Листинг 4.5. Определение операций для производного типа**

```
MODULE vector_def
IMPLICIT NONE
TYPE VECTOR
       REAL X, Y
END TYPE VECTOR
FUNCTION SUMMA(A, B)
TYPE(VECTOR) SUMMA
TYPE(VECTOR), INTENT(IN):: A, B
       SUMMA%X = A%X + B%X
       SUMMA%Y = A%Y + B%Y
END FUNCTION SUMMA
FUNCTION REAL2VECTOR(X)
TYPE(VECTOR) REAL2VECTOR
REAL, INTENT(IN) :: X
       REAL2VECTOR = VECTOR(X, 0.0)
END FUNCTION REAL2VECTOR
INTERFACE OPERATOR(+)
       MODULE PROCEDURE SUMMA
END INTERFACE
INTERFACE ASSIGNMENT(=)
       MODULE PROCEDURE REAL2VECTOR
END INTERFACE
END MODULE vector_def
```

**Листинг 4.6. Использование производного типа и задаваемых операций**

```
PROGRAM vector_use
USE vector_def
INTEGER, PARAMETER :: I = 5
```

```
INTEGER j, k
TYPE(VECTOR), PARAMETER :: one = VECTOR(1, 1)
TYPE(VECTOR), DIMENSION(I) :: X, Y
TYPE(VECTOR) u, w
DO j = 1, I
       X(j) = REAL(j)
       Y(j) = VECTOR(0,REAL(j))
END DO
u = X(1) + Y(I)
w = Y(1) + X(I)
END PROGRAM vector_use
```

Переопределение встроенных операций для встроенных типов не допускается; поэтому в задаваемом присваивании смысл присваивания для встроенных типов должен оставаться неизменным. Изменение же смысла присваивания для двух объектов одного и того же производного типа возможно.

## Выражения с массивами и присваивание массивов

Операндами выражений числового типа и операций присваивания могут быть не только скалярные объекты, но массивы и массивоподобные объекты. Выражение `-A` для массива `A` означает применение операции унарного отрицания ко всем его элементам. Выражение:

`S = A + B`,

где все операнды являются массивами одной и той же формы, правомочно и означает присваивание элементам массива `S` суммы элементов массивов `A` и `B` с теми же индексами. Так же определено скалярное присваивание для массивов:

`A = x`

Это выражение, в котором `A` — массив, а `x` — скаляр числового типа, означает присваивание всем элементам массива `A` значения `x`. В выражении `A + x` скаляр `x` будет расширен до массива той же формы, что и массив `A`, все элементы которого равны `x`.

Массивы одинаковой формы называются совместимыми. Для совместимых массивов определены все арифметические операции и логические операции отношения. Результатом арифметических операций с массивами является массив, конформный исходным, элементы которого получены применением соответствующей операции к элементам массивов-операндов. Результатом логических операций отношения является массив логического типа, элементы

которого принимают значения .TRUE. или .FALSE. в зависимости от результата применения данной операции к паре элементов массивов-операндов с одинаковыми индексами. Так, например, если A и B — двумерные массивы формы (20, 20), т. е. определены строкой:

```
INTEGER A(20, 20), B(20, 20),
```

то результатом суммирования A + B будет массив формы (20, 20), элемент которого с индексом (i, j) равен сумме A(i, j) + B(i, j).

Результатом сравнения A .EQ. B будет логический массив формы (20, 20), элемент которого с индексом (i, j) имеет значения .TRUE., если выражение A(i, j) .EQ. B(i, j) — истинно, и .FALSE., если оно ложно.

Отметим, что произведением массивов A * B является массив покомпонентных произведений, а не произведение векторов или матриц в алгебраическом смысле.

При наличии сдвига в индексах массивов-операндов учитывается не буквальное значение индекса, а положение индекса в экстенте, т. е. массивы A и B при одинаковой форме могут иметь несовпадающие границы:

```
INTEGER A(-5:14, 11:30), B(2:21, -20:-1)
```

В этом случае при вычислении бинарного выражения в качестве операндов участвуют элементы A(-5, 11) и B(2, -20), A(-4, 11) и B(3, -20) и т. д. Операндами выражений с массивами могут быть сечения массивов и векторные индексы:

```
REAL A(20, 20), V(5)
INTEGER, DIMENSION(5) :: IND=(/2, 4, 7, 9, 19/)
```

Выражение A(I, 12:16) + V определяет одномерный массив из 5-ти элементов, равных:

```
A(I, 11+J) + V(J)
```

Выражение V/A(IND) даст в результате массив той же формы, что и массив V с элементами V(1) / A(2), V(2) / A(4),V(3) / A(7), V(4) / A(9) и V(5) / A(19).

Результат выражения-массива может быть присвоен массиву соответствующей формы:

```
V = V / A(IND)
```

или

```
A(I, 12:16) = V + A(I, 11:15)
```

Примеры выражений с массивами и массивных присваиваний приведены в листинге 4.7.

**Листинг 4.7. Примеры выражений с массивами и массивных присваиваний**

```
PROGRAM arpress
IMPLICIT NONE
INTEGER, PARAMETER :: size_x = 5, size_y = 10
REAL, DIMENSION(size_x, size_y) :: X, Y, Z
INTEGER, DIMENSION(size_x, size_y) :: STAT
INTEGER, DIMENSION(size_x * size_y) :: ISTAT
INTEGER, DIMENSION(SQRT(size_x * size_y)) :: IND
LOGICAL, DIMENSION(size_x, size_y) :: LOT
INTEGER I, J
REAL s, t, r
DO I = 1, SIZE(IND)
       IND(I) = 1 + 2 * I
END DO
s = 1.1; t = 3.2; r = 2.7
X = s; Y = t;
CALL RANDOM_NUMBER(Z)
Z = (Z + X) * (Z - Y)
LOT = (Z > r)
DO I = 1, size_x
  DO J = 1, size_y
       IF(LOT(I, J)) THEN
              STAT(I, J) = 1
       ELSE
              STAT(I, J) = -1
       END IF
  END DO
END DO
ISTAT = RESHAPE(STAT, size_x * size_y))
WRITE(*, *) ISTAT(IND)
END PROGRAM arpress
```

Программа, текст которой приведен в листинге 4.7, искусственно усложнена — встроенные операции с массивами позволяют добиться того же результата гораздо изящнее, но речь о них пойдет позже *(см. гл. 8)*.

Порядок вычисления элементов в массивных выражениях и присваиваниях стандартом Фортрана не оговаривается.

# Указатели (ссылки) в выражениях и операторах присваивания

Указатели (ссылки) в Фортране не имеют того значения, которое придается указателям языка С, хотя так же имеют смысл адресов переменных, уже размещенных в памяти. Указатели языка С допускают действия и с адресами, и с адресатами; указатели Фортрана допускают *действия только с адресатами* и по смыслу аналогичны ссылкам языка С++.

**Примечание**

Термин "указатель" применяется здесь скорее как прямой перевод ключевого слова "pointer", описывающего объекты этого рода, и синоним термина "ссылка".

Операции разыменования и косвенного доступа в Фортране отсутствуют, и использование ссылок в выражениях означает их замену значениями их адресатов. Имеет смысл использовать в выражениях только *связанные* указатели — указатели, находящиеся в состоянии прикрепленности. Указатели, находящиеся в состоянии неопределенности, копируют только свое состояние, приводя в состояние неопределенности результат того выражения, в котором они участвуют. Операция прикрепления указателя к адресату обозначается комбинацией знаков "=>". Адресатом указателя может быть либо другой указатель или любой его подобъект. Если потенциальный адресат указателя не является ссылкой или подобъектом ссылки, то он должен иметь атрибут `TARGET`:

```
REAL , TARGET :: A, B
REAL, POINTER :: X, Y
……..……
A = 10.32
B = 6.8
……..……
X => A
Y => B
```

В этом примере адресатом указателя `X` является переменная `A`, адресатом `Y` — переменная `B`. Прикрепление указателя к другому указателю:

```
X => Y
```

равносильно его переадресации:

```
X=> B
```

В результате присваивания одного указателя другому:

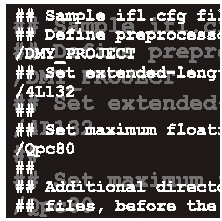
```
X = Y
```

производится присваивание адресатов; в последнем примере переменная `A` получает значение `B`.

Если адресатом указателя является массив, то указатель должен быть описан как массив того же ранга, что и массив-адресат. При прикреплении указателя к адресату границы массива-адресата воспринимаются как результат применения встроенных функций `LBOUND` и `UBOUND` *(см. гл. 10)* к результатам выражений-массивов — нижняя граница всегда равна 1, а верхняя — величине экстента.

# Вопросы и задания для самостоятельной работы

1. Определите и запишите в виде "*тип*(KIND = *параметр_разновидности_типа*)" тип и разновидность типа выражения `X + Y` для следующих случаев.
   - `X` и `Y` целого типа стандартной разновидности.
   - `X` — типа `REAL(KIND = 2)`, `Y` — целого типа.
   - `X` — целого типа, `Y` — комплексного типа стандартной разновидности.
2. Считая все операнды числовыми скалярами, тип которых определяется по правилам неявного определения типа, расставьте скобки в выражениях так, чтобы порядок вычисления соответствовал приоритету встроенных операций Фортрана.
   - `A + B + C.`
   - `F + 4 * D**5.`
   - `G / R / I**K**L.`
3. Расставьте скобки в соответствии с приоритетом операций, считая все операнды скалярами логического типа.
   - `L .OR. M .AND. .NOT. J.`
   - `.NOT. K .OR. .NOT.I .AND. .NOT. M.`
4. Считая все операнды числовыми скалярами, тип которых определяется по правилам неявного определения типа, определите тип следующих выражений.
   - `(j + k) * f + n1.`
   - `f**i.`
   - `i**f.`
   - `j**k / (a + b)**i.`
   - `j**l.`
5. Дополните модуль из листинга 4.5 операциями сравнения векторов по модулю и по расположению относительно заданного вектора, считая положительным направление вращения против часовой стрелки, а углы, образуемые векторами, находящимися в диапазоне $[-\pi, \pi]$. Программу из листинга 4.6 дополните сравнением полученных векторов `u` и `w` с вектором-параметром `one`.




# Глава 5

# Арифметика Фортрана

В этой главе мы познакомимся с особенностями программирования арифметических выражений на Фортране. Арифметические выражения составляют значительную часть вычислительной программы. На вычисление арифметических выражений тратится существенная доля процессорного времени, поэтому при программировании на Фортране правильный учет специфики машинной арифметики приобретает особое значение. Важную роль играет оптимизация вычислений. Оптимизация и управление точностью вычислений также являются темами данной главы.

## Арифметические выражения

*Арифметическое выражение* состоит из операндов, обозначений арифметических операций и круглых скобок. Его результатом является численное значение. *Операнд* — это значение, к которому применяется операция. Операндом может быть:

- буквальная числовая константа;
- скалярная (простая) переменная любого числового типа;
- числовой массив, его элемент и/или сечение;
- элемент структуры;
- вызов функции числового типа.

Значение каждого операнда перед вычислением выражения должно быть определено. В некоторых реализациях языка неопределенной переменной присваивается специальное значение `NaN` (Not-a-Number, — "нет числа"), а в некоторых — нулевое (если это операнд числового типа).

Результат арифметического выражения должен быть математически определен. Это значит, что в нем, например, не допускается деление на ноль. В выражении `A**B` с вещественным показателем результат не определен, если основание отрицательно или равно нулю.

Операции, в зависимости от количества операндов, принято делить на две группы:

- *унарные* (*одноместные*), которые применяются к одному-единственному операнду и имеют вид:

  `знак_операции операнд;`

- *бинарные* (*двухместные*), которые применяются к двум операндам:

  `операнд_1 операция операнд_2`

Например, унарной является операция изменения знака (минус) и операция "плюс":

```
A = -X,
```

а бинарной — операция вычитания:

```
Z = Y - X
```

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

```
A**(-1.5)
```

Здесь подряд идут знак бинарной операции возведения в степень, а затем знак унарной операции "минус".

Операции также бывают *встроенными* и *задаваемыми*. Встроенные операции определены разработчиками языка программирования и имеются в нем изначально. В Фортране такими операциями являются сложение (`+`), вычитание (`-`), умножение (`*`), деление (`/`) и возведение в степень (`**`). Основные арифметические операции имеют собственные обозначения, некоторые арифметические операции реализованы в виде встроенных функций (пример — `MOD`, функция вычисления остатка от деления).

Одним символом могут обозначаться разные операции. Так, например, правила выполнения деления (`/`) различаются для целых, вещественных и комплексных операндов. Это явление называется *перегрузкой операции* или *статическим полиморфизмом*. Операция возведения в степень также полиморфна: степень в выражении `I = J**2` вычисляется посредством умножения, а в выражении `I = J**2.` вычисляется с помощью логарифмов (благодаря точке "." после двойки в показателе степени).

Выражения бывают *скалярными* и *выражениями-массивами*. В первом случае операндами являются скалярные значения — простые переменные, элементы массивов и т. д., результат тоже скалярный. Во втором случае результатом выражения является массив или его часть (сечение).

# Задаваемые операции

Задаваемые операции (их также называют перегружаемыми) определяются для производных типов. Перегрузка операций — это использование одного имени для обозначения разных операций в общей области их действия. Область действия операции совпадает с программной единицей, в которой содержится определение этой операции.

Встроенные операции определены для встроенных типов. Среди них арифметические операции, семантика которых определена для числовых типов (*семантикой* называют описание смысла выражений, операторов, программных единиц). В Фортране есть возможность задавать операции и для производных типов. Конечно, необходимую операцию для производного типа можно реализовать и с помощью обычной подпрограммы, но это не всегда удобно. Легче вместо вызова подпрограммы использовать специальный знак, возможно, наделив его новой семантикой. Это достигается переопределением имеющихся в языке операций или заданием новых.

Рассмотрим пример переопределения операции. Пусть имеется производный тип:

```
TYPE COORDS
 REAL X, Y
END TYPE COORDS
```

Для того чтобы определить бинарную операцию для этого типа, необходимо определить функцию, например, вида:

```
FUNCTION ADD_COORDS(A, B)
TYPE(COORDS) ADD_COORDS
TYPE(COORDS), INTENT(IN) :: A, B
ADD_COORDS%LOWER = A%LOWER + B%LOWER
ADD_COORDS%UPPER = A%UPPER + B%UPPER
END FUNCTION ADD_COORDS
```

Необходимо также задать интерфейсный блок:

```
INTERFACE OPERATOR(+)
MODULE PROCEDURE ADD_COORDS
END INTERFACE
```

При вычислении выражений, содержащих переопределенную операцию сложения, с операндами типа `COORDS`:

```
COORDS :: X1, Y1, Z1
X1 = Y1 + Z1
```

будет вызываться указанная функция. Аналогично переопределяются унарные операции, но в этом случае используется только один входной аргумент.

В качестве имени операции можно использовать обозначение уже существующей операции, а можно использовать последовательность букв (длиной не более 31), которой предшествует точка и которая завершается точкой. Если, например, заголовок интерфейсного блока имеет вид:

```
INTERFACE OPERATOR(DOTPRODUCT),
```

то допустимы выражения вида:

```
Z = X.DOTPRODUCT.Y
```

Если для обозначения задаваемой операции используется имя существующей операции, количество операндов должно остаться таким же. При переопределении сохраняется приоритет исходной операции. При переопределении не наследуются свойства коммутативности и ассоциативности, т. е. результат выражения `x + y - z` может отличаться от результата выражения `y - z + x` и зависит от того, как определены соответствующие операции. Приоритет у одноместных задаваемых операций самый высокий, а у двухместных — самый низкий. Если для обозначения операции используются два имени (например, < и .LT.), при переопределении одной из них, автоматически переопределяется и другая. Имя задаваемой одноместной операции, если для нее не используется встроенная лексема, может следовать за другим знаком операции. Встроенную операцию, как в примере:

```
INTEGER :: X, Y, Z
...
X = Y + Z
```

переопределить нельзя.

# Порядок выполнения арифметических операций

Последовательность выполнения арифметических операций в выражении определяют три фактора:

- последовательность операндов в выражении;
- приоритеты операций;
- расстановка круглых скобок, выделяющих части выражения (подвыражения).

Если в выражении используются операции с равным приоритетом, они выполняются по порядку, слева направо. Если приоритет операций различается, сначала выполняются операции с наибольшим приоритетом, затем менее приоритетные операции и в последнюю очередь — операции с наименьшим приоритетом. Приоритеты операций приведены в *гл. 1.*

При вычислении выражения приоритет очередной операции сравнивается с приоритетом следующей операции. Если приоритет следующей операции

меньше, вычисляется результат данной операции (в выражении `1 * 2 - 3` сначала выполняется умножение, затем вычитание). Если приоритеты равны, то применяются правила ассоциативности (в выражении `1 * 2 * 3` операции выполняются так: `(1 * 2) * 3`, а в выражении `2**3**4` — так: `2**(3**4)`). Наконец, если приоритет следующей операции выше, "сканирование" выражения продолжается. Унарные операции имеют наивысший приоритет.

Часть выражения (подвыражение) может выделяться скобками. Подвыражения вычисляются в первую очередь. Если в выражении используются вложенные скобки, сначала выполняются подвыражения в скобках самого глубокого уровня вложенности. Затем, по порядку, вычисляются подвыражения более высокого уровня.

# Арифметическое присваивание

Напомним, что оператор присваивания в Фортране имеет вид:

```
переменная = выражение
```

Присваивание называется *арифметическим*, если типы переменной и выражения являются числовыми (то есть целыми, вещественными или комплексными).

Выражение справа должно иметь тип, диапазон которого соответствует диапазону типа переменной в левой части. Например, вещественное выражение в правой части оператора присваивания, результат которого равен $3 \times 109$, недопустимо, если слева находится переменная стандартного целого типа (`INTEGER(4)`).

При преобразовании стандартного целого типа, содержащего 10 десятичных разрядов, в стандартный вещественный тип (`REAL(4)`), который содержит 7 значащих цифр, может произойти потеря точности.

Если типы выражения и переменной различаются, выполняется преобразование типа результата выражения к типу переменной.

В левой части оператора присваивания не может находиться константа. Например: `2 = X + Y` — это ошибочная конструкция, поскольку `2` является константой.

Типы объектов в левой и в правой частях оператора присваивания должны соответствовать друг другу. Конструкция `I = A//B` является ошибочной, потому что справа находится строковое выражение, а слева — переменная целого типа.

Формы объектов в левой и правой частях оператора присваивания должны совпадать. Очевидно, что `X = (:)` — ошибочная конструкция.

В левой части оператора присваивания не может находиться выражение, т. е. `X + Y = Z` — ошибочная конструкция.

## Преобразование типов

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 (в скобках приведен результат):

```
1 + 2 (3)
3.14 - 2.71 (0.43)
5 * 6 (30)
12 / 5 (2)
1 / 2 (0)
```

Последний результат объясняется тем, что при выполнении целочисленного деления отбрасывается дробная часть (но не округляется!).

Сложнее обстоит дело с вычислением неоднородных (смешанных) арифметических выражений, то есть выражений, в которых содержатся переменные и константы разных типов. При выполнении бинарных арифметических операций операнды всегда сначала преобразуются к общему типу, и лишь затем выполняется операция, причем делается это всегда автоматически. Результат имеет общий тип. Эти правила действуют и в случае, если один из операндов является константой.

Исключением является операция возведения в степень `X**N`. Для нее показатель степени, если он целого типа, останется без изменения, а результат будет вычислен с помощью многократного перемножения основания степени, т. е. `2**4` равносильно `2 * 2 * 2 * 2`.

Примеры смешанных выражений:

```
3 + 0.5 (3.5)
1 / 2.0 (0.5)
5.0**(1 / 2) (1)
```

Логические значения в составе арифметического выражения трактуются как целые значения.

Преобразование типов операндов определяется их *рангом*. Ранг является характеристикой типа. Перечень встроенных типов в порядке возрастания их ранга дан ниже.

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, результат будет иметь тип операнда с наибольшим рангом. Это общее правило. Если операнды целые, результат имеет тот же параметр разновидности, что и у операнда с более широким диапазоном. Если операнды вещественные или комплексные, результат имеет тот же параметр разновидности, что и операнд с большей точностью.

Если в операции с комплексным операндом присутствует целый операнд, он вначале преобразуется в вещественный тип. Результат преобразования считается вещественной частью комплексного числа, а мнимая часть полагается равной нулю.

Преобразование типов может приводить к различным 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 5.1).

**Листинг 5.1. 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 (Compaq Visual Fortran):

```
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.

Проведем несколько несложных опытов с программой `PRECISION_LOSS`. Если в качестве значения переменной `X` взять число $1.25_{10}$, 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:

$1.01000010100011110101110\ 000101000111101011100001_2$

Для значений стандартного вещественного типа в формате стандарта IEEE 754 (The Institute of Electrical and Electronics Engineers, Inc. — Институт Инженеров Электротехники и Электроники) сохраняются только первые 23 разряда этого представления. 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 5.2.

**Листинг 5.2. Пример нарушения коммутативности сложения**

```
PROGRAM ASSOCIATIVITY_LOSS
REAL(4) :: X = 1.0E16, Y = -1.0E16, Z = 1.0
REAL(4) :: A, B
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 свойство коммутативности арифметического сложения, то есть, результат зависит от того, в каком порядке выполняются операции:

```
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 5.3). И здесь положение не спасает переход к двойной точности.

**Листинг 5.3. Программа, которая демонстрирует потерю малого значения**

```
PROGRAM BIG_TO_SMALL
IMPLICIT NONE
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
```

Программа `SUMS` (листинг 5.4) демонстрирует зависимость результата вычисления суммы от порядка слагаемых:

$$A = \sum_{k=1}^{I} \frac{1}{k},$$

$$B = \sum_{k=I}^{1} \frac{1}{k}.$$

Здесь $A$ — сумма в прямом порядке, а $B$ — та же сумма, но в обратном порядке.

**Листинг 5.4. Суммирование в прямом и обратном порядке**

```
PROGRAM SUMS
 IMPLICIT NONE
 INTEGER :: I, K
 REAL :: A, B
 DATA A /0./, B /0./, I /1000000/
 DO K = 1, I
       A = A + 1. / FLOAT(K)
       B = B + 1. / FLOAT(I - K + 1)
 ENDDO
 PRINT *, "A =", A
 PRINT *, "B =", B
END PROGRAM SUMS
```

Для Compaq Visual Fortran результат выглядит следующим образом:

```
A = 14.35736
B = 14.39265
```

При вычислении суммы в прямом порядке к частной сумме при каждом проходе цикла добавляется значение очередного слагаемого. При этом величина слагаемого уменьшается, а величина частичной суммы возрастает. Наступает момент, когда при выполнении суммирования слагаемые значительно различаются по величине, следствием чего является увеличение погрешности вычисления суммы. При суммировании в обратном порядке суммируемые значения примерно одного порядка величины. Погрешность в этом случае меньше. Таким образом, второй результат (значение `B`) следует считать более точным, чем первый (значение `A`). При переходе к двойной точности различие наблюдается только в 13-м знаке после запятой.

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. 5.1 приведены встроенные функции преобразования типов. Некоторые имена этих функций являются родовыми, то есть тип аргумента может быть разным. У некоторых из этих функций есть необязательный скалярный целый входной аргумент `KIND`, который задает разновидность результата.

***Таблица 5.1.*** *Встроенные элементные функции преобразования типов*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| INT(X [, KIND]) | INTEGER<br>REAL<br>COMPLEX | INTEGER | Преобразование к целому типу. Если второй аргумент отсутствует, результат имеет стандартный целый тип. Если указанная разновидность не поддерживается процессором, результат считается неопределенным.<br>Если аргумент X целый, результат совпадает со значением аргумента.<br>Если аргумент X имеет вещественный тип, а его значение по абсолютной величине меньше единицы, результатом будет нулевое значение.<br>Если аргумент X вещественный и по абсолютной величине больше единицы, результатом будет целое значение, абсолютная величина которого не превосходит значение \|X\|, а знак тот же, что и у аргумента.<br>Если аргумент комплексный, вышеперечисленные правила применяются к вещественной части аргумента |
| REAL(X [, KIND]) | INTEGER<br>REAL<br>COMPLEX | REAL | Преобразование к вещественному типу. Если второй аргумент отсутствует, результат имеет стандартный вещественный тип. Если аргумент комплексный, а параметр KIND отсутствует, разновидность результата та же, что и у первого параметра |
| AIMAG(X) | COMPLEX | REAL | Вычисление мнимой части комплексного числа. Результат имеет тот же параметр разновидности, что и X. Для вычисления вещественной части комплексного числа используется функция REAL |

***Таблица 5.1*** *(продолжение)*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| FLOAT(X) | INTEGER | REAL(4) | Преобразование целого типа к вещественному типу. Действует аналогично функции REAL |
| DBLE(X) | INTEGER<br>REAL<br>COMPLEX | REAL(8) | Преобразование аргумента к вещественному типу с двойной точностью. Результат имеет максимально возможную числовую точность |
| CMPLX(X [, Y] [, KIND])) | INTEGER<br>REAL<br>COMPLEX | COMPLEX(4) | Преобразование к комплексному типу. Эту функцию нельзя передавать в качестве фактического аргумента. Если оба аргумента не комплексные, первый аргумент преобразуется в вещественную часть результата, а второй в мнимую его часть.<br><br>Необязательный входной аргумент Y вещественный или целый. Он задает мнимую часть комплексного значения и не требуется при комплексном X.<br><br>Если отсутствует аргумент KIND, параметр разновидности соответствует стандартному вещественному типу.<br><br>Если имеется только один не комплексный аргумент, он преобразуется в вещественную часть комплексного значения, а мнимая часть равна нулю. Если Y отсутствует, а X комплексное, результатом является CMPLX(REAL(X), AIMAG(X)) |
| TRANSFER(X, Y [, SIZE])) | INTEGER<br>REAL<br>COMPLEX | REAL | Функция переноса, которая переводит данные одного типа в данные другого типа без изменения их физического представления (значения двоичных разрядов значений обоих типов совпадают). Возвращает результат того же типа и с такими же параметрами типа, что и у Y. |

***Таблица 5.1*** *(окончание)*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| TRANSFER(X, Y [, SIZE])) | INTEGER<br>REAL<br>COMPLEX | REAL | Если разрядность типов X и Y совпадает, двоичные разряды в представлении обоих аргументов полностью совпадают. Если разрядность X меньше разрядности Y, дополнительные (младшие) разряды Y заполняются нулями. Если разрядность X больше разрядности Y, "лишние" разряды слева обрезаются |

# Инициализация переменных

Присвоение переменной начального значения называется ее *инициализацией*. В Фортране имеется несколько способов инициализации переменных:

- в предложениях описания;
- 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. Прежде всего, напомним, что при выполнении присваивания происходит выполнение целой последовательности действий, таких как обращение к памяти, запись в регистры и т. д. Все это приводит к определенным 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.

Часто переменные инициализируются нулевыми значениями. Некоторые компиляторы сами обнуляют те переменные, которые не инициализированы

явно. Тем не менее, лучше выполнять явную инициализацию. Это улучшает переносимость программы. Следует также помнить, что составной объект считается определенным, если определены все его компоненты (элементы массива, поля структуры).

Примеры инициализации переменных в предложениях описания:

```
INTEGER :: YEAR = 1959
REAL(4), DIMENSION(3) :: COORDS = (/0.0, 1.21, 2.17/)
CHARACTER(LEN = 8) :: NAME = 'SVETLANA'
```

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 неявных циклов, список_значений — список буквальных скалярных констант и конструкторов структур. Количество элементов в списке объектов должно совпадать с количеством элементов в списке значений. В операторе DATA не требуется строгое соответствие типов объектов и значений, достаточно, чтобы выполнялось условие совместимости типов по присваиванию. Переменную в любом блоке видимости можно инициализировать только один раз.

Примеры операторов DATA:

```
REAL(8) :: X, Y
INTEGER, DIMENSION(4) :: A12
DATA X, Y /1.2978D+01, -3.1222D0/
DATA A12 /1, 5, 5, 0/
```

Здесь переменной X присваивается значение 1.29789D1, переменной Y — отрицательное значение -3.1222D0, а элементам массива A12 величины:

```
A12(1) = 1
A12(2) = 5
A12(3) = 5
A12(4) = 0
```

В этом примере два одинаковых значения 5 идут подряд. Если в списке значений несколько одинаковых значений идут подряд, их можно записать в виде одной константы со скалярным целым коэффициентом повторения:

```
INTEGER, DIMENSION(4) :: A12
DATA A12 /1, 2 * 5, 0/
```

Коэффициент повторения может быт именованной или буквальной константой, которая отделяется от значения звездочкой. Значение коэффициента повторения может быть положительным или нулевым.

Оператор DATA можно использовать для инициализации сечения массива:

```
DATA RDATA(3:6, 2) /4*0.0/
```

В этом примере нулевые значения присваиваются четырем элементам массива RDATA (RDATA(3, 2), RDATA(4, 2), RDATA(5, 2) и RDATA(6, 2)).

В операторе DATA можно использовать и неявные циклы с параметром:

```
DATA ((RDATA(I, J), I = 1, 21, 2), J = 1, 10) /110*0.0/
```

Неявный цикл с параметром имеет вид:

```
(список, параметр_цикла = начальное_значение, конечное_значение, [, шаг])
```

Здесь список может содержать элементы массивов, скалярные компоненты структур, вложенные неявные циклы с параметром. Начальное, конечное значения и шаг должны быть скалярными целыми выражениями, в которые могут входить и параметры циклов, но только охватывающих данный, а параметр_цикла — скалярная переменная целого типа. Если шаг не указан, он считается равным единице. В вышеприведенном примере список представляет собой вложенный неявный цикл, а начальное и конечное значения заданы буквальными константами 1 и 10. Список во вложенном цикле состоит из элементов массива RDATA.

Оператор DATA можно использовать для инициализации объектов производного типа. Значением в этом случае является конструктор структуры, который инициализирует все компоненты структуры. Пример:

```
TYPE COORDINATES
 REAL X, Y, Z
 CHARACTER A
END TYPE COORDINATES
TYPE(COORDINATES) :: POINT1, POINT2
DATA POINT1 /COORDINATES(1., 1.1, 2., 'P1')/, POINT2 /COORDINATES(-1.,
-1.1, -2., 'P2')/
```

При инициализации переменная получает атрибут SAVE, даже если инициализирована только часть переменной. Допускается инициализация переменных всех видов. Не допускается инициализация формальных параметров, ссылок, автоматических массивов, автоматических объектов, результатов функций и переменных, которые доступны через USE-ассоциирование или ассоциирование через носитель *(см. гл. 8)*.

Оператор BLOCK DATA является заголовком программной единицы, которая называется блоком данных и используется для задания начальных значений из именованного COMMON-блока при помощи операторов DATA. Вообще же говоря, инициализировать значения из общего блока с помощью оператора DATA нельзя.

Блок данных имеет следующую структуру:

```
BLOCK DATA [ИМЯ]
Предложения описания
COMMON-блоки
END [BLOCK DATA [ИМЯ]]
```

После заголовка BLOCK DATA может идти имя блока, оно не является обязательным, но в программе допускается только один такой неименованный блок. Имя, идущее после слова END, должно совпадать с именем блока. Предложения описания являются необязательной частью блока и представляют собой предложения описания переменных (но не ссылок). В предложениях описания нельзя использовать атрибуты ALLOCATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE и PUBLIC. В блоке данных нельзя использовать интерфейсный блок и исполняемые операторы.

Допускается использование предложений USE, IMPLICIT, PARAMETER, INTRINSIC, POINTER, TARGET, COMMON, DIMENSION, EQUIVALENCE, DATA, SAVE и, конечно, предложений описания переменных, а также описаний производных типов. В предложениях описания типа в блоке данных можно использовать только атрибуты DIMENSION, INTRINSIC, PARAMETER, POINTER, SAVE и TARGET. Блок данных может определить начальные значения нескольких COMMON-блоков, но любой COMMON-блок может появиться только в одном блоке данных программы. Имя блока данных может появиться в предложении EXTERNAL другого блока программы. Это делается в случае, когда необходимо при загрузке программных компонентов из библиотеки загрузить нужный блок данных.

Если оператор DATA инициализирует любую переменную в именованном общем COMMON-блоке, блок данных должен иметь полный набор предложений описания, определяющих этот общий блок. Однако все переменные в блоке данных инициализировать не обязательно.

Переменные, значения которых заданы в BLOCK DATA, могут использоваться в любой программной единице, содержащей общий блок с тем же именем. Лучше в каждом блоке данных собирать общие блоки, связанные между собой логически.

Конструкцию BLOCK DATA в настоящее время относят к числу "избыточных" свойств языка потому, что те же функции могут выполнять модули.

# Встроенные математические функции

В состав арифметических выражений часто входят математические функции. В табл. 5.2 приведены основные математические функции Фортрана. У некоторых из этих функций есть необязательный скалярный целый входной аргумент KIND, который задает разновидность результата.

***Таблица 5.2.*** *Встроенные элементные математические функции Фортрана*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| SQRT(X) | REAL(4)<br>COMPLEX(4) | REAL(4)<br>COMPLEX(4) | Квадратный корень из X. Имя SQRT является родовым, тип результата зависит от типа аргумента. Если тип аргумента вещественный, его значение не должно быть отрицательным. Тип результата совпадает с типом аргумента. Для комплексного аргумента (A, B) результатом является комплексное значение<br>SQRT(A**2 + B**2) * EXP(-0.5 * (0., 1.) * ATAN(A / B)) |
| ABS(X) | INTEGER(4)<br>REAL(4)<br>COMPLEX(4) | INTEGER(4)<br>REAL(4)<br>COMPLEX(4) | Абсолютная величина X. Имя ABS является родовым, тип результата зависит от типа аргумента. Если аргумент X целый или вещественный, результатом является \|X\|, в случае комплексного значения (A, B), результат — вещественное значение SQRT(A**2 + B**2) |
| MOD(X, Y) | INTEGER(4)<br>REAL(4) | INTEGER(4)<br>REAL(4) | Остаток от деления X на Y. Имя MOD является родовым, тип результата совпадает с типом аргумента. Оба аргумента должны быть одного типа и иметь одну разновидность. Второй аргумент должен отличаться от нуля. Результат:<br>X - INT(X / Y) * Y |
| MODULO(X, Y) | INTEGER<br>REAL | INTEGER<br>REAL | X по модулю Y. Результат R:<br>R = N * Y + X,<br>где N — целое число, причем \|R\| < Y. Если второй аргумент равен нулю, результат не определен. Если оба аргумента вещественные, результат:<br>R = X - FLOOR(X / Y) * Y |
| CONJG(Z) | COMPLEX(4) | COMPLEX(4) | Комплексное сопряжение: (X, Y) преобразуется в (X, -Y) |

***Таблица 5.2*** *(окончание)*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| DPROD(X, Y) | REAL(4) | REAL(8) | Произведение вещественных аргументов с двойной точностью |
| FLOOR (X [, KIND]) | REAL | INTEGER | Наибольшее стандартное целое, меньшее или равное значению вещественного аргумента X.<br>Необязательный второй аргумент KIND у этой функции появился в Фортране 95. Если его нет, разновидность результата соответствует стандартному целому типу. Если указанная разновидность не поддерживается процессором, результат считается неопределенным |
| AINT (X [, KIND]) | REAL | REAL | Наибольшее целое, абсолютная величина которого не превосходит \|X\|, а знак совпадает со знаком X. Если \|X\| меньше единицы, результат — нулевое значение.<br>Если этот аргумент отсутствует, разновидность результата совпадает с разновидностью X.<br>Для округления следует использовать функцию ANINT |
| SIGN(X, Y) | INTEGER<br>REAL | INTEGER<br>REAL | X со знаком Y |

Кроме математических функций, в Фортране есть набор встроенных элементных функций, предназначенных для работы с двоичными разрядами (битами) целых значений. Некоторые из этих функций приведены в табл. 5.3.

***Таблица 5.3.*** *Элементные встроенные битовые функции*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| BIT_SIZE(I) | INTEGER | INTEGER | Число двоичных разрядов, необходимых для представления целого числа с таким же параметром разновидности, что и аргумент. Для аргумента типа INTEGER(1) результат 8, для типов INTEGER(2) и INTEGER(4) значения результата соответственно 16 и 32 |

***Таблица 5.3*** *(окончание)*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| `ISHFT(I, SHIFT)` | `INTEGER` | `INTEGER` | Логический сдвиг. Тип результата совпадает с типом аргумента `I`. Результат получается сдвигом битов `I` на `SHIFT` позиций. При положительном значении аргумента `SHIFT` сдвиг выполняется влево, при отрицательном — вправо. При нулевом значении `SHIFT` сдвиг не выполняется. При сдвиге влево лишние биты теряются, а при сдвиге вправо новые разряды заполняются нулями. Абсолютное значение параметра `SHIFT` должно быть не больше значения `BIT_SIZE(I)` |
| `ISHFTC(I, SHIFT [, SIZE])` | `INTEGER` | `INTEGER` | Циклический логический сдвиг младших (правых битов). Абсолютное значение `SHIFT` должно быть меньше или равно значению `SIZE`. Значение `SIZE` должно быть положительным и не должно превосходить `BIT_SIZE(I)`. Если третий аргумент отсутствует, считается, что его значение равно `BIT_SIZE(I)`.<br>Тип результата совпадает с типом `I`. Результат получается циклическим сдвигом `SIZE` самых правых битов `I` на `SHIFT` позиций. При положительном значении `SHIFT` сдвиг выполняется влево, а при отрицательном — вправо. При нулевом значении `SHIFT` сдвиг не выполняется вообще.<br>Биты не теряются. Биты в `I`, не попавшие в позиции, заданные `SIZE`, остаются неизменными |

В различных реализациях Фортрана могут быть также дополнительные функции сдвига.

Приведем примеры, поясняющие действие функций сдвига. Функция

```
ISHFT(3, 2)
```

возвращает значение 6, а функция

```
ISHFT(2, -1)
```

— значение 1.

Функция

```
ISHFT((/5, 5, 5/), (/2, -1, 0/)
```

возвращает значение массива

```
(/20, 2, 5/)
```

А вот что происходит при выполнении следующего фрагмента программы:

```
INTEGER(1) :: I = 10, A, B
A = ISHFTC(I, 2, 3)
B = ISHFTC(I, -2, 3)
```

Исходное значение переменной `I` в двоичном представлении 0000 1010. При выполнении оператора присваивания для переменной `A` выполняется циклический левый сдвиг на две позиции трех самых младших двоичных разрядов. В результате получим 0000 1001 или в десятичном представлении 9. При выполнении второго оператора присваивания 3 младших бита циклически сдвигаются на 2 разряда вправо, и переменной `B` будет присвоено значение 0000 1100 или 12.

В табл. 5.4 приведены другие математические функции. Имена некоторых математических функций являются родовыми, они автоматически используют правильный тип данных и разновидность, а аргумент определяет разновидность результата, которая обычно та же, что и у аргумента. Эти функции являются элементными, то есть аргументом может быть массив или его сечение. В этом случае результатом является также массив (математик сказал бы, что эти функции являются вектор-функциями).

***Таблица 5.4.*** *Встроенные элементные математические функции Фортрана*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| `EXP(X)` | `REAL(4)`<br>`COMPLEX(4)` | `REAL(4)`<br>`COMPLEX(4)` | Экспонента. Результат имеет тот же тип, что и аргумент. Если `X` комплексного типа, считается, что его мнимая часть задана в радианах |
| `LOG(X)` | `REAL(4)`<br>`COMPLEX(4)` | `REAL(4)`<br>`COMPLEX(4)` | Натуральный логарифм. Значение вещественного аргумента должно быть положительным. Значение комплексного аргумента должно отличаться от нуля. Результат имеет тот же тип, что и аргумент. Если аргумент комплексный `(A, B)`, результатом является комплексное число:<br>`(LOG(SQRT(A**2 + B**2))), ATAN2(A, B))`<br>Значение мнимой части результата находится в диапазоне от $-\pi$ до $\pi$. Она равна $\pi$, если `A` < 0, а `B` = 0 |

***Таблица 5.4*** *(окончание)*

| Функция | Тип аргумента | Тип результата | Описание |
|---|---|---|---|
| LOG10(X) | REAL(4) | REAL(4) | Десятичный логарифм. Аргумент должен быть положительным |
| SIN(X) | REAL(4)<br>COMPLEX(4) | REAL(4)<br>COMPLEX(4) | Синус. Аргумент задается в радианах и берется по модулю 2 * π. Вещественная часть комплексного аргумента считается заданной в радианах. Тип результата совпадает с типом аргумента |
| ASIN(X) | REAL(4) | REAL(4) | Арксинус. Аргумент должен удовлетворять условию \|X\| < 1. Результат выражен в радианах, а его значение находится в диапазоне от –π / 2 до π / 2 |
| COS(X) | REAL(4)<br>COMPLEX(4) | REAL(4)<br>COMPLEX(4) | Косинус. Аргумент задается в радианах и рассматривается по модулю 2 * π. Вещественная часть комплексного аргумента считается заданной в радианах. Тип результата совпадает с типом аргумента |
| ACOS(X) | REAL(4) | REAL(4) | Арккосинус. Аргумент должен удовлетворять условию \|X\| < 1. Результат выражен в радианах, а его значение лежит в промежутке от 0 до π |
| ATAN(X) | REAL(4) | REAL(4) | Арктангенс. Результат выражен в радианах, а его значение лежит в промежутке от –π / 2 до π / 2 |
| SINH(X) | REAL(4) | REAL(4) | Гиперболический синус |
| COSH(X) | REAL(4) | REAL(4) | Гиперболический косинус |
| TANH(X) | REAL(4) | REAL(4) | Гиперболический тангенс |
| TAN(X) | REAL(4) | REAL(4) | Тангенс. Аргумент задается в радианах и берется по модулю 2 * π |

# Точность вычислений

Замечательным свойством современного Фортрана является возможность управлять, в определенных пределах, точностью вычислений. Эта возможность основана на концепции *моделей данных* числового типа. Используются две модели данных: целого и вещественного типа. Разновидности целого типа можно описать следующей моделью:

$$I = s\sum_{k=1}^{q} w_k r^{k-1},$$

где $I$ — целое число, $s$ — знак (±1), $q$ — количество разрядов (положительное целое число), $r$ — основание системы счисления в модели ($r = 2$), $w_k$ — неотрицательное целое число (значение разряда, $0 \le w_k \le r-1$). Разным значениям параметра разновидности соответствуют разные значения $q$. Все 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. Никаких проблем не возникает и при вычислении отношений.

Вещественный тип данных моделирует вещественные числа, но только приближенно. Так, например, число 0.12 в десятичной системе счисления является бесконечной дробью в двоичной системе, следовательно, для хранения этого числа с абсолютной точностью потребовался бы компьютер с бесконечно большой разрядностью. Дробь приходится обрезать, отбрасывая "лишние" двоичные разряды. Это и является источником погрешности. Таким образом, множество вещественных чисел, допускающих машинное представление, — это множество дробных двоичных чисел с конечной разрядностью. Такое множество содержит конечное число элементов (значений), а значения любых двух элементов из этого множества отличаются на малую, но конечную величину. В математике такое множество называется конечным и дискретным. Учет особенностей модели вещественных значений важен при программировании вычислений, ведь любой численный алгоритм, корректность которого доказана теоретически (для множества обычных вещественных значений с произвольной разрядностью), после его "отображения" на дискретное множество значений может потерять некоторые из своих свойств. Это часто приводит к появлению погрешности счета, а иногда и к неправильной работе самого алгоритма.

Модель вещественного значения $X$ описывается формулой:

$$X = sb^e \sum_{k=1}^{p} f_k b^{-k},$$

где $s$ — знак (±1), $p$ — положительное целое число, $b$ — основание системы счисления ($b = 2$), $e$ — целое число из диапазона $e_{\min} \le e \le e_{\max}$, $f_k$ — целое число ($0 \le f_k < b$, $f_1 \ne 0$). Основными характеристиками любой разновидности вещественного типа являются значения $b$ и $p$, а также диапазон значений $e$.

Вычисления с вещественными типами выполняются приближенно, однако это не всегда важно, потому что если речь идет, например, об обработке экспериментальных данных, сами эти данные обычно приближенные и получаются с определенной погрешностью.

В старых версиях Фортрана простая точность на одном компьютере могла соответствовать двойной (повышенной) точности на другом. Динамический доступ к численным свойствам реализации позволяет создавать переносимое и численно устойчивое программное обеспечение. В Фортране 90 такой доступ обеспечивается с помощью концепции моделей числовых значений и набора соответствующих встроенных функций для получения характеристик моделей.

Прежде чем мы перейдем к обзору средств управления точностью вычислений, обратимся к форматам внутреннего представления числовых значений. К числу основных форматов относятся форматы IEEE. Институт Инженеров Электротехники и Электроники разработал стандарты представления чисел с плавающей точкой (IEEE Std 754-1985). Вообще говоря, имеется несколько международных стандартов на формат хранения вещественных чисел, но стандарт IEEE является основным. Формат хранения основан на представлении числа в виде:

$(-1)^s * мантисса * 2^{порядок}$

где $s$ — знак числа, *мантисса* записывается в нормализованном виде, то есть после десятичной точки идет значащая цифра, а не ноль.

В старшем разряде кодируется знак числа, затем записывается порядок со смещением и абсолютная величина мантиссы. С точки зрения технической реализации удобно, если порядок положительный. В этом случае операции с порядками могут быть сведены к операциям над целыми числами без знака. Для того чтобы обеспечить неотрицательность порядка, к его значению в дополнительном коде суммируется смещение $(2^{m-1} - 1)$, где $m$ — разрядность порядка. Порядку в диапазоне от −127 до 128 в этом случае соответствует диапазон значений от 0 до 255 ($m = 8$, смещение 127).

Точность представления вещественного числа зависит от количества разрядов, отводимых под запись мантиссы, — чем больше разрядность, тем выше точность. Чем больше разрядность порядка, 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 разряда отводятся для мантиссы, а 8 для порядка (смещение равно 127). 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 (см. табл. 5.5);
- формат с двойной точностью — 64-разрядное число, в котором 53 разряда отводятся мантиссе и знаку, а 11 — порядку (смещение равно 1023).

Таким образом, арифметика IEEE основана на двоичном ($b = 2$) представлении с $p = 24$ (простая точность), $p = 56$ (двойная точность) и $-127 < e < 127$ (простая точность) или $-1023 < e < 1023$ (двойная точность). В большинстве реализаций отличие заключается в значении $p$, хотя возможно и различие диапазонов изменения $e$.

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.

Полезно представлять в общих чертах, как выполняются арифметические операции с вещественными числами. Перед выполнением операции числа загружаются в специальные регистры арифметико-логического устройства. 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.

При умножении двух чисел их порядки складываются, а мантиссы перемножаются. При делении двух чисел из порядка делимого вычитается порядок делителя, а мантисса делимого делится на мантиссу делителя. При необходимости результат приводится к нормальному виду. Деление для компьютера является самой сложной арифметической операцией.

***Таблица 5.5.*** *Формат IEEE для вещественного типа с простой точностью*

| **Порядковый номер бита** | **31** | **30:23** | **23:0** |
|---|---|---|---|
| Назначение | Бит знака (s) | Порядок (e) | Мантисса |

Биты целой и дробной частей называются *мантиссой* (в англоязычной литературе используется термин significand — значащая часть числа). У нормализованных чисел бит целой части равен 1, поэтому единицу хранить не имеет смысла (и так известно, что она есть), а "экономия" этого бита освобождает дополнительный двоичный разряд для использования с целью увеличения точности. Этот бит называют "скрытым". Числа хранятся в формате знак-абсолютная величина, поэтому отрицательное число имеет такую же мантиссу, что и положительное, но с дополнительным битом знака. Константа-"смещение" прибавляется к показателю степени, что позволяет сделать все показатели положительными.

Если все разряды поля порядка содержат все единицы — это особый результат. Нулевая мантисса в этом случае представляет бесконечность (положи-

тельную или отрицательную), а ненулевая мантисса представляет особое значение NaN. Значение NaN является следствием некорректной численной операции.

Обычно возможности численных расчетов обеспечиваются простой и двойной точностью вещественных типов данных, комплексным типом и библиотеками подпрограмм численных методов. Иногда в реализации языка включаются дополнительные возможности (такие как, например, четверная точность).

В 32-битовом формате вещественного числа с простой точностью 24 двоичных разряда мантиссы, включая "скрытый" бит, представляют 7 значащих десятичных цифр. Вы уже знаете, что "истинное" множество чисел непрерывно. А вот множество чисел, которые могут быть представлены с помощью конечного числа двоичных разрядов, — конечно, и между любыми двумя числами имеется промежуток. Если число нельзя представить точно, оно заменяется ближайшим, допускающим точное машинное представление.

Следствием того, что для представления всех нормализованных чисел используется одинаковое количество разрядов, является то, что чем меньше порядок, тем больше плотность представимых чисел. Между 1 и 2 имеется примерно $8 \times 10^6$ вещественных значений с простой точностью и только около 8000 между 1023 и 1024.

Программа STRANGE_RESULT, исходный текст которой приведен в листинге 5.5, демонстрирует неожиданное, на первый взгляд, поведение. После умножения числа 0.01 на 100 получаем результат, отличный от 1.

**Листинг 5.5. Потеря точности при умножении**

```
PROGRAM STRANGE_RESULT
REAL :: 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'
ENDIF
END PROGRAM STRANGE_RESULT
```

Объясняется такое поведение тем, что в формате IEEE с простой точностью десятичное значение 0.01 нельзя представить точно, поэтому, скорее всего, будет напечатано первое сообщение.

В следующем примере (листинг 5.6) цифры, которые на первый взгляд являются значимыми, на самом деле не имеют смысла.

**Листинг 5.6. Потеря точности при вычитании**

```
PROGRAM ADDITIONAL_DIGITS
REAL :: A, Y = 1000.2
A = Y - 1.E3
PRINT *, A
END PROGRAM ADDITIONAL_DIGITS
```

Вещественное значение с простой точностью (REAL) имеет не более 7 значимых десятичных цифр, поэтому результат может представлять только около 3 десятичных цифр. Программа выведет число 0.2000122. Это связано с тем, что число 1000.2 не допускает точное представление в двоичном формате с плавающей точкой, а 1000 допускает, поэтому результат будет немного больше 0.2. Четвертая и все следующие цифры не имеют смысла.

Программист должен сам следить за точностью вычислений и количеством значащих цифр. Особое внимание при этом следует обратить:

- на вычитание чисел, которые почти равны друг другу;
- на сложение чисел, почти равных по модулю, но различающихся знаком;
- на сложение и вычитание чисел, которые значительно различаются по величине.

На разных компьютерах для представления чисел с плавающей точкой может использоваться разное количество разрядов. Даже при использовании аналогичных форматов IEEE разный размер промежуточных регистров может приводить к разным результатам. Для увеличения переносимости следует избегать сравнений на равенство для вещественных чисел, заменяя их сравнением абсолютной величины разности двух чисел с маленьким положительным значением (EPSILON в примере листинг 5.7).

**Листинг 5.7. "Правильное" сравнение вещественных значений**

```
PROGRAM REAL_COMPAR
REAL, PARAMETER :: EPSILON = 1.E-6
REAL :: X, A = 13.9, B = 5.E-6
X = (A * B) / B
IF(ABS(X - A).LE.(ABS(X) * EPSILON)) THEN
 PRINT *, 'X ПРИБЛИЗИТЕЛЬНО РАВНО A'
END IF
IF((X - A) .GT. (ABS(X) * EPSILON)) THEN
 PRINT *, 'X БОЛЬШЕ A'
END IF
```

```
IF((A - X) .GT. (ABS(X) * EPSILON)) THEN
 PRINT *, 'X МЕНЬШЕ A'
END IF
END PROGRAM REAL_COMPAR
```

Даже сравнения "больше чем", "меньше или равно" и другие могут приводить к неожиданным, на первый взгляд, результатам. В следующем примере (листинг 5.8) 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 5.8. Потеря точности при сложении**

```
PROGRAM EXAMPLE
REAL :: X, DELTA = .001
DO J = 1, 100000
 X = J + DELTA
 IF(X / J .LE. 1.0) THEN
 PRINT *, 'X МЕНЬШЕ J!'
 STOP
 END IF
ENDDO
END PROGRAM EXAMPLE
```

Итак, мы выяснили, что для того, чтобы эффективно использовать возможности машинной арифметики с плавающей точкой, надо знать ее ограничения:

- в IEEE-формате с простой точностью могут быть представлены только 7 десятичных цифр и 16 — в формате с двойной точностью;
- точность может теряться при каждом преобразовании внешнего десятичного представления в двоичное и наоборот.

Кроме того, всегда используйте "безопасную" форму сравнений!

Иногда полезно знать, как устроено арифметико-логическое устройство. Так например, регистры с плавающей точкой процессоров Intel Pentium имеют 64 разряда для мантиссы, 15 битов порядка и разряд знака (всего 80). Это позволяет производить промежуточные вычисления с гораздо меньшими потерями точности, чем многие другие реализации. В зависимости от способа сохранения промежуточных значений в регистрах, внешне одинаковые вычисления могут приводить к разным результатам.

Встроенная функция `SELECTED_REAL_KIND([X][, Y])` позволяет программисту задать минимальную десятичную точность и/или диапазон порядка.

У этой функции есть два необязательных целых аргумента, но, по крайней мере, один из них при вызове должен быть задан. Функция возвращает значение параметра разновидности вещественного типа с десятичной точностью не меньше X, с десятичным степенным диапазоном не хуже Y. Этот тип содержит все вещественные числа из диапазона $[-10^Y, 10^Y]$. Если в данной реализации есть несколько подходящих разновидностей, выбирается та, которая имеет наименьшую десятичную точность. Если такого типа нет, возвращается код ошибки. Если искомой разновидности нет, возвращается значение −1. Если недоступна искомая точность, возвращается значение −2, а если недоступны и разновидность и точность, возвращается −3. Если заданному критерию удовлетворяет более одного параметра разновидности, возвращается значение с наименьшей десятичной точностью.

В примере:

```
INTEGER, PARAMETER :: PREC10 = SELECTED_REAL_KIND(10)
REAL(KIND = PREC10) :: X, Y, EF12
```

описываются переменные X, Y и EF12, которые должны иметь вещественный тип, по крайней мере, с 10 десятичными цифрами точности. На компьютере с процессором Intel это двойная точность (параметр разновидности 8).

Вещественные буквальные константы любой разновидности можно задавать, приписывая значение параметра разновидности к значению константы после символа подчеркивания. Таким образом, равносильны следующие формы записи констант:

`3.14_4` и `3.14E0`

`1.41_8` и `1.41D0`

Встроенная функция преобразования `SELECTED_INT_KIND(X)` возвращает параметр разновидности целого типа, представляющего все значения в диапазоне $[-10^X, 10^X]$. Если ни одного такого параметра разновидности нет на процессоре, результат равен −1. Если критерию удовлетворяет более одного параметра разновидности, возвращается значение с наименьшим десятичным степенным диапазоном. Например, после вызова:

```
I = SELECTED_INT_KIND(8)
```

переменной I будет присвоено значение 4.

С помощью справочных функций программист может получить доступ к параметрам модели целых или вещественных чисел. Аргументом справочной функции может быть константа, скалярная переменная или массив. Если аргументом является переменная, ее значение не играет роли, переменная вообще может быть не определена.

Встроенная функция `KIND(X)` возвращает стандартное целое значение — параметр разновидности аргумента. Аргументом может быть переменная или константа.

Функция DIGITS(X) возвращает число двоичных значащих цифр в модели аргумента X. Перечень возвращаемых значений представлен в табл. 5.6.

***Таблица 5.6.*** *Результат вычисления функции DIGITS*

| Тип аргумента | Результат |
|---|---|
| INTEGER(1) | 7 |
| INTEGER(2) | 15 |
| INTEGER(4) | 31 |
| REAL(4) | 24 |
| REAL(8) | 53 |

Встроенная справочная функция EPSILON(X) возвращает наименьшее число, которое компьютер может "отличить" от нуля. Аргумент должен быть вещественного типа, скалярный или массив. Результат скалярный и имеет тот же тип и разновидность, что и аргумент. Добавление этого числа к единице дает результат, больший единицы. Суммирование меньшего значения даст единицу. Данная функция может использоваться для определения минимально допустимого шага в алгоритмах поиска корней методом перебора. Если фактическое значение шага будет меньше минимально допустимого, алгоритм "зациклится". Возможные значения приведены в табл. 5.7.

***Таблица 5.7.*** *Результат вычисления функции EPSILON*

| Тип аргумента | Результат |
|---|---|
| REAL(4) | 1.192093E-07 |
| REAL(8) | 2.22044049250313E-016 |

Встроенная справочная функция HUGE(X) возвращает наибольшее для модели аргумента значение (табл. 5.8).

***Таблица 5.8.*** *Результат вычисления функции HUGE*

| Тип аргумента | Результат |
|---|---|
| INTEGER(1) | 127 |
| INTEGER(2) | 32767 |
| INTEGER(4) | 2147483647 |
| REAL(4) | 3.402823E+38 |
| REAL(8) | 1.797693134862316E+308 |

Результат является скаляром того же типа и разновидности, что и аргумент.

Встроенная справочная функция `MAXEXPONENT(X)` возвращает максимальное значение показателя степени в модели, определяемой типом и разновидностью аргумента. Аргумент `X` — вещественный (скаляр или массив). Результат имеет стандартный целый тип и равен $e_{max}$.

Встроенная справочная функция `MINEXPONENT(X)` возвращает минимальное значение показателя степени в модели, определяемой типом и разновидностью аргумента. Аргумент `X` — вещественный (скаляр или массив). Результат имеет стандартный целый тип и равен $e_{min}$.

Встроенная справочная функция `PRECISION(X)` возвращает число значащих цифр после десятичной точки, используемых для представления чисел в той же модели, что и аргумент. Аргумент может быть вещественным или комплексным, скалярным или массивом. Результат является скаляром стандартного целого типа. Его значение `INT((DIGITS(X) - 1) * LOG10(RADIX(X)))`. Если `RADIX(X)` является целой степенью 10, к результату прибавляется 1. Возможные значения результата приведены в табл. 5.9.

***Таблица 5.9.*** *Результат вычисления функции PRECISION*

| Тип аргумента | Результат |
|---|---|
| `REAL(4)` | 6 |
| `REAL(8)` | 15 |
| `COMPLEX(4)` | 6 |
| `COMPLEX(8)` | 15 |

Встроенная справочная функция `RADIX(X)` возвращает основание системы счисления, используемое в модели, в которой представлен аргумент. Аргумент может быть целого или вещественного типа (скаляр или массив). Результат имеет стандартный целый тип и равен *r*.

Встроенная справочная функция `RANGE(X)` возвращает десятичный степенной диапазон значений экспоненты в модели аргумента: `INT(LOG10(HUGE))` для целых и `INT(MIN(LOG10(HUGE), -LOG10(TINY)))` для вещественных и комплексных. Результат приводится в табл. 5.10. Аргумент может быть скаляром или массивом, а результат имеет стандартный целый тип.

***Таблица 5.10.*** *Результат вычисления функции RANGE*

| Тип аргумента | Результат |
|---|---|
| `INTEGER(1)` | 2 |
| `INTEGER(2)` | 4 |

***Таблица 5.10*** *(окончание)*

| Тип аргумента | Результат |
|---|---|
| INTEGER(4) | 9 |
| REAL(4) | 37 |
| REAL(8) | 307 |
| COMPLEX(4) | 37 |
| COMPLEX(8) | 307 |

Встроенная справочная функция TINY(X) возвращает наименьшее положительное значение в модели аргумента (табл. 5.11). Аргументом может быть скаляр или массив. Результат является скаляром того же типа и разновидности, что и аргумент.

***Таблица 5.11.*** *Результат вычисления функции TINY*

| Тип аргумента | Результат |
|---|---|
| REAL(4) | 1.175494E-38 |
| REAL(8) | 2.225073858507201E-308 |

Встроенная элементная функция EXPONENT(X) возвращает экспоненциальную часть аргумента в используемой модели вещественной арифметики. Аргумент должен быть вещественного типа. Результат имеет целый тип. Если аргумент ненулевой, значение результата является экспоненциальной частью аргумента. Порядок должен находиться в пределах диапазона для стандартного целого типа, в противном случае результат неопределенный. Результатом для нулевого аргумента является нулевое значение.

Встроенная элементная функция FRACTION(X) возвращает дробную часть в модели представления вещественного значения, заданного в качестве аргумента. Аргумент должен быть вещественного типа. Тип результата совпадает с типом аргумента. Если аргумент равен нулю, результат также будет нулевым.

Встроенная элементная функция NEAREST(X, S) возвращает вещественное значение, равное ближайшему к X числу, допускающему машинное представление, большему X (при S > 0) или меньшему X (при S < 0). Аргументы вещественные, причем второй аргумент ненулевой. Тип результата совпадает с типом аргумента. Пример приведен в листинге 5.9.

**Листинг 5.9. Пример использования функции *NEAREST***

```
PROGRAM NEAREST_TEST
REAL(4) :: A = 4.0
REAL(8) :: B = 111502.07D0, RES
 RES = NEAREST(A, -2.0)
 WRITE(*, *) RES
 RES = NEAREST(B, 2.0)
 WRITE(*, '(1X, Z16)') RES
 RES = NEAREST(B, -2.0)
 WRITE(*, '(1X, Z16)') RES
 END PROGRAM NEAREST_TEST
```

Результат:

```
3.99999976158142
40FB38E11EB851ED
40FB38E11EB851EB
```

При вычислении в этом примере значения, ближайшего к значению типа `REAL(8)`, различие можно заметить только в шестнадцатеричном представлении чисел.

`RRSPACING(X)` — обратная величина относительного расстояния между числами в модели в области, близкой к `X`.

Встроенная элементная функция масштабирования `SCALE(X, I)` возвращает $X * B^I$, где `B` — это основание в модели для `X`. Первый параметр (`X`) должен быть вещественным, а второй (`I`) — целым. Тип результата совпадает с типом параметра `X`.

Встроенная элементная функция `SET_EXPONENT(X, I)` возвращает значение $X * B^{I-EXPONENT(X)}$. У этого числа разновидность типа такая же, как и у первого аргумента, дробная часть равна дробной части модельного представления `X`, а степенная часть равна `I`.

Встроенная элементная функция `SPACING(X)` возвращает абсолютное расстояние между числами в области, близкой к `X`. Аргумент должен быть вещественного типа. Если результат находится за пределами диапазона для модели вещественных чисел, результат `TINY(X)`.

В качестве примера использования функции `SPACING`, рассмотрим программу, текст которой приведен в листинге 5.10 и которая предназначена для вычисления корня нелинейного уравнения. Корнем уравнения называют значение аргумента $x^*$, при котором функция $f(x)$ обращается в ноль:

$$f(x^*) = 0.$$

Для вычисления корня используется метод Ньютона (метод касательных). Это метод последовательных приближений, в котором выбирается начальное приближение $x_0$, а затем многократно применяются следующие формулы:

$$x_{n+1} = x_n - \Delta x_n,$$

$$\Delta x_n = \frac{f(x_n)}{f'(x_{n+1})}.$$

В нашем примере корень вычисляется с точностью, максимальной для данной разновидности вещественного типа при минимуме итераций. Очевидно, не имеет смысла выполнение ньютоновских итераций после того, как величина шага оказывается меньше разности между двумя последовательными вещественными значениями, допускающими машинное представление. В качестве функции выбран тригонометрический синус, функция $f(x)$ запрограммирована в подпрограмме-функции `F`, а ее производная — в подпрограмме `DF`.

**Листинг 5.10. Решение нелинейного уравнения методом Ньютона**

```
PROGRAM NEWTON
 IMPLICIT NONE
 REAL(8) :: X, DX, F, DF
 X = 3.3 ! НАЧАЛЬНОЕ ПРИБЛИЖЕНИЕ
 DO ! НЬЮТОНОВСКИЕ ИТЕРАЦИИ
       DX = F(X) / DF(X) ! ВЫЧИСЛЕНИЕ ШАГА
      X = X - DX ! ВЫЧИСЛЕНИЕ ОЧЕРЕДНОГО ПРИБЛИЖЕНИЯ
       IF(DX <= SPACING(X)) EXIT ! ЦИКЛ ЗАВЕРШАЕТСЯ, КОГДА
                          ! ШАГ МЕНЬШЕ РАССТОЯНИЯ МЕЖДУ
                          ! ДВУМЯ ПОСЛЕДОВАТЕЛЬНЫМИ
                          ! ВЕЩЕСТВЕННЫМИ ЗНАЧЕНИЯМИ
 END DO
 PRINT *, X ! ВЫВОД ЗНАЧЕНИЯ КОРНЯ
 PRINT *, F(X) ! ВЫВОД ЗНАЧЕНИЯ ФУНКЦИИ В ТОЧКЕ X
 PRINT *, DF(X) ! ВЫВОД ЗНАЧЕНИЯ ПРОИЗВОДНОЙ
               ! ФУНКЦИИ В ТОЧКЕ X
END PROGRAM NEWTON

REAL(8) FUNCTION F(X)
 IMPLICIT NONE
 REAL(8) :: X
```

```
 F = SIN(X)
RETURN
END

REAL(8) FUNCTION DF(X)
 IMPLICIT NONE
 REAL(8) :: X
 DF = COS(X)
RETURN
END
```

# Оптимизация вычислений

Основная часть программ, написанных на Фортране, — вычислительные программы. Время их работы варьируется в широких пределах. Это могут быть небольшие программы, выполнение которых требует нескольких минут процессорного времени. При разработке таких программ проблема повышения эффективности, как правило, не возникает. Действительно, можно ли считать серьезной проблему сокращения времени счета, например, с двух минут процессорного времени до одной? Совершенно иной является ситуация, когда программа предназначена для решения достаточно трудоемкой задачи и время счета измеряется уже часами, а порой сутками и неделями. Сокращение времени счета в этом случае позволяет увеличить оборачиваемость программы, то есть количество ее прогонов с разными входными данными за единицу времени. Оптимизация программы в этом случае позволяет повысить эффективность труда пользователя. В этом разделе мы кратко перечислим основные методы повышения эффективности вычислений и дадим практические рекомендации по оптимизации некоторых типичных программных конструкций.

Идея *оптимизации* проста и заключается в том, что сократить время выполнения программы в целом или какого-либо ее фрагмента можно, уменьшив *количество операций*, сохраняя при этом, разумеется, логическую структуру алгоритма. Сократить же время выполнения отдельных операций можно, лишь сменив компьютер.

Принято говорить о двух видах оптимизации.

1. Автоматическая оптимизация, выполняемая компилятором.
2. Оптимизация, выполняемая "вручную".

Роль программиста при автоматической оптимизации сводится к указанию необходимых ключей при запуске компилятора (например, `-O2`). Очевидно, этот способ наименее трудоемкий, но он не позволяет добиться большого выигрыша в производительности программы потому, что компилятор "ста-

рается" избежать такой перестройки кода, которая могла бы нарушить логику его работы. Автоматическая оптимизация выполняется очень осторожно и почти "безопасно". Тем не менее, результат автоматической оптимизации следует проверять. В частности, при автоматической оптимизации может изменяться порядок вычислений, а это, в свою очередь, может привести к потере точности.

Более значительный выигрыш в быстродействии позволяет получить оптимизация, выполняемая программистом вручную. Лучше всего, конечно, сразу использовать оптимальные формы записи арифметических и логических выражений, а также уделять должное внимание аккуратной записи других конструкций программы.

Следует учитывать также и то обстоятельство, что один и тот же фрагмент программы может быть записан по-разному. Может оказаться, что один вариант относительно легко оптимизируется компилятором, а в случае неудачной записи автоматическая оптимизация может оказаться невозможной. Это еще один аргумент в пользу необходимости выработать привычку писать программу сразу оптимально.

Обратимся к приемам оптимизации, которые могут использоваться программистом и не зависят от компилятора.

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. Ниже приведены примеры арифметических операций, расположенные в порядке возрастания их трудоемкости и времени выполнения. Это распределение приблизительное, оно может немного меняться в различных ситуациях.

1. Сложение и вычитание с целыми операндами.
2. Сложение и вычитание с вещественными операндами.
3. Умножение с целыми операндами.
4. Умножение с вещественными операндами.
5. Деление с целыми операндами.
6. Деление с вещественными операндами.
7. Возведение в положительную целую степень с показателем — константой.
8. Возведение в степень с показателем — целой переменной.
9. 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, степень вычисляется простым перемножением основания. Если же показатель вещественный, ситуация становится сложнее и 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 + A(3) * 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 знакопеременного ряда:

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

Эту сумму можно вычислить следующим образом:

```
SUM = 0
SERIES: DO I = 0, N
 SUM = SUM + (-1)**I * A(I)
END DO SERIES
```

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

```
SUM = 0
SERIES: DO I = 0, N - 1
 SUM = SUM - A(I) + A(I+1)
END DO SERIES
```

В арифметических выражениях следует 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 против трех в первом случае.

При использовании временных переменных появляются дополнительные накладные расходы на обращения к памяти, поэтому там, где это возможно и удобно (при вычислении сравнительно простых выражений), следует избегать промежуточных вычислений и вспомогательных присваиваний. Таким образом, вместо:

```
TEMP1 = X + RO
TEMP2 = Z * MMI * SV
ETA = TEMP1 + TEMP2
```

следует использовать:

```
ETA = X + RO + Z * MMI * SV
```

Следует тщательно выбирать типы переменных, избегая там, где это возможно, 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 простой пример:

```
A = X + Y + Z + I + J + K
```

Предположим, что первые три переменных в выражении имеют вещественный тип, а остальные — целый. В этом выражении выполняются 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.

Значительное увеличение производительности программы может дать оптимизация циклов. Используя цикл, следует иметь в виду, что часть времени выполнения цикла составляет его инициализация. Инициализация включает вычисление числа повторений цикла и некоторые другие действия. Учитывая это, можно дать несколько советов по "правильному" использованию циклов. Если цикл короткий, от него можно отказаться, "развернув" последовательные итерации в линейную последовательность операторов. Исходный текст программы станет немного длиннее, но скорость ее выполнения увеличится.

При использовании вложенных циклов следует, если возможно, размещать их таким образом, чтобы самым глубоко вложенным был цикл с наибольшим числом повторений, а самым внешним — самый короткий. Сравним две конструкции:

```
CYCLE1: DO I = 1, 1000
CYCLE2: DO J = 1, 50
```

```
CYCLE3: DO K = 1, 10
              SVROM(I, J, K) = A(I) + B(J) + F(K, 12)
       END DO CYCLE3
END DO CYCLE2
END DO CYCLE1
```

и

```
CYCLE3: DO I = 1, 10
CYCLE2: DO J = 1, 500
CYCLE1: DO K = 1, 10000
              SVROM(I, J, K) = A(I) + B(J) + F(K, 12)
       ENDDO CYCLE1
END DO CYCLE2
END DO CYCLE3
```

В первом случае число инициализаций — 51 001, а во втором — 5011, т. е. приблизительно в 10 раз меньше. Разумеется, не всегда можно расположить вложенные циклы в желательном порядке.

В цикле следует избегать явных индексных вычислений:

```
CYCLE1: DO K = 1, 10000
       SVROM = A(2 * K)
END DO CYCLE1
```

Вместо них можно использовать шаг цикла:

```
CYCLE1: DO K = 2, 10000, 2
       SVROM = A(K)
END DO CYCLE1
```

Иногда применяют прием, который называется "разверткой цикла". Он заключается в уменьшении числа проверок при выполнении цикла, например, исходный код:

```
CYCLE1: DO I = 1, 200
       SVROM = A(I) / B(I)
END DO CYCLE1
```

разворачивается в цикл с меньшим числом повторений:

```
CYCLE1: DO I = 2, 200, 2
       SVROM = A(I - 1) / B(I - 1)
       SVROM = A(I) / B(I)
END DO CYCLE1
```

Как видно, в одном проходе развернутого цикла содержатся несколько проходов исходного цикла (это число называют степенью развертки). Число итераций цикла должно быть кратно степени развертки.

Другой прием — объединение циклов. Циклы, идущие друг за другом:

```
CYCLE1: DO I = 1, N
       VR = B(I) + SIN(DELTA * FLOAT(I))
ENDDO CYCLE1
CYCLE2: DO I = 1, N
       SE = X(I) - Y(I)
END DO CYCLE2
```

иногда можно объединить, сократив количество инициализаций:

```
CYCLE1: DO I = 1, N
       VR = B(I) + SIN(DELTA * FLOAT(I))
       SE = X(I) - Y(I)
ENDDO CYCLE1
```

Обратным приемом является расщепление циклов. Этот прием применяется к тем циклам, которые содержат инвариантные логические проверки. Пример цикла до расщепления:

```
CYCLE1: DO I = 1, N
       IF(X > 0) THEN
       SE = CX(I) * Y(I)
       ELSE
       SE = CX(I) / Y(I)
       END IF
END DO CYCLE1
```

Тот же цикл после расщепления:

```
IF(X > 0) THEN
CYCLE1: DO I = 1, N
       SE = CX(I) * Y(I)
       END DO CYCLE1
ELSE
       CYCLE2: DO I = 1, N
       SE = CX(I) / Y(I)
       END DO CYCLE2
END IF
```

В этом случае нет многократных проверок — вычислений логического выражения в условном операторе.

В следующем примере:

```
CYCLE1: DO I = 1, N
       SE = CX(I) * Y(I) / EPS
END DO CYCLE1
```

операция деления на переменную EPS одинакова для всех итераций цикла. Ее можно вынести за пределы цикла:

```
CYCLE1: DO I = 1, N
        SE = CX(I) * Y(I)
END DO CYCLE1
SE = SE / EPS
```

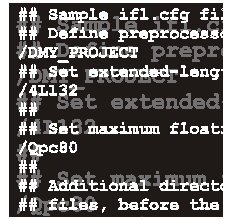
В этом случае уменьшается количество операций, выполняемых в цикле. Деление на переменную EPS является примером *инварианта цикла. Инвариантные коды* — это выражения или части выражений, которые не зависят от управляющей переменной цикла. Иногда компилятор во время компиляции программы сам выносит за пределы цикла инвариантные коды, сокращая тем самым количество лишних операций. Лучше, если об удалении инвариантных кодов позаботится сам программист.

# Вопросы и задания для самостоятельной работы

1. Перечислите основные и дополнительные арифметические операции языка Фортран.
2. Что такое операнд?
3. Какие бывают операции?
4. Что называют перегрузкой операций?
5. Какие выражения называют однородными?
6. Какие выражения называются смешанными? Как происходит их вычисление?
7. Чем определяется порядок выполнения арифметических операций в выражении?
8. Расскажите об операторе присваивания.
9. Что такое параметр разновидности типа?
10. Перечислите встроенные математические функции Фортрана.
11. Перечислите встроенные функции преобразования типа.
12. Опишите модель целых чисел Фортрана.
13. Опишите модель вещественных чисел Фортрана.
14. Расскажите о встроенных функциях Фортрана, позволяющих получить параметры моделей целых и вещественных чисел.
15. Дайте сравнительный обзор различных методов инициализации переменных.

16. Расскажите о программной единице BLOCK DATA.
17. Сформулируйте список полезных советов по оптимизации вычислений.
18. Что называется инвариантным кодом?
19. Напишите программу и проведите исследование характеристик множеств целых и вещественных чисел с разным параметром разновидности. Постройте графическое представление этих множеств.
20. Сделайте доклад "Проблема потери точности при программировании вычислений".
21. Что такое ранг операнда?
22. Напишите программу поиска экстремума заданной функции, ближайшего к заданной точке, методом перебора значений аргумента. Для шага используйте минимально возможное в модели используемого вещественного типа значение.
23. Напишите программу решения квадратного уравнения $x^2 - 2\,a\,x + e = 0$ с максимально возможной точностью. Здесь $e$ — малое число.
24. Проведите исследование погрешности вычисления производных разных порядков с помощью конечно-разностных аппроксимаций. По результатам исследования напишите модуль, содержащий подпрограммы численного дифференцирования с максимально возможной точностью.
25. 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. Поставьте автору одну из следующих оценок: "Эксперт Фортрана", "Владеет Фортраном со словарем", "Немедленно прочитать книгу Немнюгина и Стесик!"

# Глава 6




# Основные алгоритмические конструкции

Как было отмечено в предыдущей главе, последовательное вычисление выражений и исполнение присваиваний составляет основу любой вычислительной программы, но ни одна реальная программа не состоит только из них. Как правило, в зависимости от результатов промежуточных вычислений требуется совершать различные действия; может появиться необходимость передать управление в другую часть программы или остановить ее работу. Одна и та же часть программы может выполняться несколько раз подряд с различными значениями переменных; в языке должно быть средство, позволяющее совершать это без многократного повторения одних и тех же операторов. Все эти действия обеспечиваются управляющими конструкциями Фортрана.

Большая часть управляющих конструкций представляет собой блок, ограниченный оператором-заголовком и оператором завершения. В некоторых управляющих конструкциях между операторами заголовка и завершения могут присутствовать промежуточные операторы. Операторы, размещенные внутри управляющего блока, доступны только при входе через заголовок блока. Управляющие конструкции могут вкладываться одна в другую и должны помещаться внутри заключающего блока полностью.

## Ветвления

Для логического разделения программы или ее частей на две или более ветвей применяются оператор `IF`, конструкция `IF… END IF` и конструкция `SELECT(CASE)`.

Оператор `IF` представляет собой случай простейшего ветвления — выполнение одного действия по условию. Этот оператор имеет вид:

```
IF (скалярное_логическое_выражение) действие,
```

где `действие` — любой исполняемый оператор, который будет выполнен только при условии истинности скобочного выражения. Например:

```
IF (NUMBER .LT. 0) NUMBER = -NUMBER
```

```
IF (FLAG) GO TO 1000
IF (COND .OR. R .GT. Q .AND. A .LE. B) REST(R, Q) = -D(Q, R) / A * B.
```

Оператор IF наиболее продуктивен при операторе перехода (GO TO Метка). Для переходов по условию и одиночных присваиваний он, главным образом, и используется. Простой пример использования условного оператора приведен в листинге 6.1.

**Листинг 6.1. Пример использования условного оператора**

```
PROGRAM cond
PARAMETER (I = 5)
WRITE(*, *) 'Угадайте число: I ='
       READ(*, *) J
       IF(I .EQ. J) WRITE(*, *) 'Угадали!'
       IF(I .LT. J) WRITE(*, *) 'Не угадали! I меньше.'
       IF(I .GT. J) WRITE(*, *) 'Не угадали! I больше.'
END PROGRAM cond
```

Если по условию нужно выполнить несколько действий, то должна применяться конструкция IF…THEN, которая имеет вид:

```
[имя:]   IF (скалярное_логическое_выражение) THEN
              блок_действий
         END IF [имя]
```

Например:

```
replace: IF (x < y) THEN
              t = x
              x = y
              y = t
         END IF replace
```

Блок_действий может содержать произвольное число любых исполняемых операторов, кроме оператора END и неполных управляющих конструкций. Этот блок будет выполняться, если скалярное_логическое_выражение истинно, и не будет выполняться, если оно ложно (т. е. во втором случае только проверяется скобочное выражение). Необязательное имя (необязательные элементы синтаксического описания заключаются в квадратные скобки) может быть назначено любой управляющей конструкции и должно быть правильным уникальным именем Фортрана. Назначение имени влечет за собой обязательное указание его в операторе завершения конструкции. Именование может принести немалую пользу при наличии даже не очень большого числа вложенных конструкций.

При необходимости выполнения блока альтернативных действий, конструкция IF…THEN может быть расширена блоком ELSE:

```
[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:

```
replace: IF (x < y) THEN
              t = x
              x = y
              y = t
         ELSE
              t = -x
              x = -y
              y = t
         END IF replace
```

Конструкция IF…THEN…ELSE расширяет возможности программы по угадыванию задуманного числа из листинга 6.1 — смотрите листинг 6.2.

**Листинг 6.2. Пример использования конструкции *IF…THEN…ELSE***

```
PROGRAM condi
PARAMETER (I = 5)
       WRITE(*, *) 'Угадайте число: I ='
       READ(*, *) J
IF(I .EQ. J) WRITE(*, *) 'Угадали!'
IF(I .LT. J) THEN
       WRITE(*, *) 'I меньше!'
ELSE
       WRITE(*, *) 'I ,больше!'
END IF
END PROGRAM condi
```

И, наконец, в наиболее полной форме, конструкция IF…THEN…ELSE может содержать несколько блоков альтернативных действий, вводимых оператором ELSE IF…THEN:

```
[имя:]  IF (скалярное_логическое_выражение) THEN
              блок_действий
```

```
ELSE IF(скалярное_логическое_выражение_1) THEN [имя]
            блок_альтернативных_действий_1
ELSE IF(скалярное_логическое_выражение_2) THEN [имя]
            блок_альтернативных_действий _2
.....
ELSE [имя]
      блок_альтернативных_действий
END IF [имя]
```

В этом варианте конструкции при истинности содержимого первых скобок (скалярное_логическое_выражение) будет выполнен блок_действий; в противном случае поочередно будут проверяться последующие скобочные выражения (скалярное_логическое_выражение_i) до тех пор, пока не будет найдено истинное. Тогда проверка прекратится, и будет выполнен соответствующий блок_альтернативных_действий_i, после чего управление будет передано оператору завершения конструкции END IF. Если среди условий во всех операторах ELSE IF не будет найдено ни одного истинного, выполнится блок_альтернативных_действий, следующий за оператором ELSE. Если этот оператор опущен (что допускается), то не будет совершено никаких действий. Операторов ELSE IF…THEN в конструкции может быть несколько (сколько угодно, в том числе, и ни одного), оператор ELSE может появляться не более одного раза (может и отсутствовать). Имя в операторах ELSE IF…THEN может появляться только в случае наличия имени у всей конструкции и обязано совпадать с ним. Пример применения расширенной конструкции IF…THEN…ELSE IF… приведен ниже (листинг 6.3).

**Листинг 6.3. Пример использования конструкции *IF…THEN…ELSE IF***

```
PROGRAM condic
PARAMETER (I = 5)
       WRITE(*, *) 'Угадайте число: I ='
       READ(*, *) J
       IF(I .LT. J) THEN
                     ELSE IF(I .GT. J) THEN
              WRITE(*, *) 'I ,больше!'
       ELSE
              WRITE(*, *) 'Угадали!'
       END IF
END PROGRAM condic
```

Блоки действий могут содержать метки и операторы перехода к этим меткам, но переход к меткам внутренних блоков конструкции IF…THEN…ELSE

извне самих блоков не допускается. Переход к метке в блоках ELSE IF и ELSE недопустим ниоткуда. Следующая конструкция ошибочна:

```
IF( t < 0 ) THEN
       A = 1
       GO TO 1
ELSE
       A = 2
       B = 3
END IF
```

Из конструкции можно передать управление только оператору завершения данной конструкции:

```
IF(t < 0) THEN
       A = 1
       GO TO 1
ELSE
       A = 2
       B = 3
1  END IF
```

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
```

Структурированное расположение операторов в записи управляющих конструкций облегчает восприятие этих конструкций и потому весьма желательно, однако при большей глубине вложения рекомендуется использовать имена.

Другой управляющей конструкцией, позволяющей создавать разветвления кода выбором одного варианта из нескольких возможных, является конструкция SELECT. В этой конструкции, в отличие от конструкции IF, 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]
```

Элемент `Имя` конструкции `SELECT`, как, впрочем, имена всех именуемых конструкций, является необязательным, но если он указывается в заголовке конструкции, то становится обязательным для завершающего оператора. В промежуточных операторах имя не обязательно, но если имя появляется в них, то оно должно совпадать с именем заголовка конструкции. `Выражение`, помещаемое в заголовке конструкции `SELECT`, должно быть скалярным выражением целого, логического или текстового типа. `Селектор_i` — это выражение, которое должно относиться к тому же типу, что и `выражение` в заголовке; разновидность типа может отличаться для *выражений* и *селекторов* числового и логического типов и должна совпадать в случае текстового типа. *Селектор* имеет вид заключенного в круглые скобки перечня неперекрывающихся значений и интервалов, например:

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

Простейший селектор — скалярное инициализирующее выражение, например, буквальная константа:

```
CASE (1)
```

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

```
CASE(1:9)
CASE(A:Z)
```

Одна из границ в диапазоне может быть опущена, тогда диапазон принимает смысл неравенства: диапазон (:-1) означает "все значения меньше 0" (для выражений целого типа). В следующем примере конструкция `SELECT` моделирует функцию определения знака целого числа (листинг 6.4):

**Листинг 6.4. 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
```

Так как диапазоны селекторов обязаны не пересекаться, невозможно выполнение более чем одного блока CASE. Последний оператор в выполняющемся блоке передает управление оператору END SELECT. Если выражение не удовлетворяет ни одному из селекторов, в конструкции не выполняется никаких действий и управление передается оператору, следующему за END SELECT. Тогда, когда значение *выражения* заголовка может оказаться вне всех диапазонов селекторов, можно использовать селектор DEFAULT — он равносилен совокупности всех значений, не охваченных объединениями диапазонов всех остальных селекторов. В конструкции не может быть больше одного оператора CASE DEFAULT, порядок размещения CASE DEFAULT среди других операторов CASE не имеет значения. В следующем примере конструкция SELECT с оператором 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
```

Конструкцию SELECT можно вкладывать одну в другую.

Переходы по меткам разрешены только внутри одного блока; не допускаются переходы на оператор CASE; переход на оператор END SELECT допустим только из той конструкции SELECT, которую он завершает.

# Оператор останова

Работа программы может быть остановлена в любом месте. Для этого нужно использовать оператор STOP. Оператор STOP можно разместить в любом месте программы, он может иметь метку и быть частью условного оператора.

```
IF(IEND .GT. 100) STOP
```

Если оператор STOP снабжен *сообщением*, которым может быть любая текстовая константа или цепочка цифр (от одной до пяти, начальные нули не учитываются), то место останова может быть идентифицировано этим сообщением, например:

```
IF( INPUT .LT. DATA_LENGTH) STOP 'Unexpected end of data'
STOP 'Exit from subroutine ROUTE'
STOP 341
```

Например, нашу программу с угадыванием числа можно оформить так (листинг 6.5).

**Листинг 6.5. Пример**

```
PROGRAM conds
PARAMETER (I = 5)
WRITE(*, *) 'Угадайте число: I ='
READ(*, *) J
IF(I .EQ. J) STOP 'Угадали!'
WRITE(*, *) 'Не угадали!'
END PROGRAM conds
```

# Оператор перехода

Оператор перехода во всех языках программирования является предметом споров, а его использование может привести к получению административных взысканий. Действительно, с одной стороны, изобилие переходов делает даже очень простую программу неудобочитаемой, с другой стороны, во многих ситуациях без операторов перехода невозможно обойтись.

Оператор перехода имеет вид:

```
GO TO метка
```

Метка должна отмечать один из исполняемых операторов программы, этот оператор называется адресатом перехода:

```
X = Y + Z
GO TO 1
C = A + B
2 D = E * F
1 WRITE (*, *) X, C, D
```

Здесь приведен пример безусловного перехода: после оператора `X = Y + Z` управление передается оператору `WRITE (*, *) X, C, D`. При этом у оператора `D = E * F` еще сохраняется шанс быть выполненным, так как у него есть метка и он может стать адресатом перехода. У оператора `C = A + B` нет никаких шансов на выполнение, он представляет собой так называемый *потерянный код* — явный признак плохо составленной программы. Посмотрите в листинге 6.6, как можно использовать оператор перехода в нашей программе по угадыванию числа.

**Листинг 6.6. Пример использования оператора перехода**

```
PROGRAM condg
PARAMETER (I = 5)
       WRITE(*, *) 'Угадайте число: I ='
       READ(*, *) J
       IF(I .EQ. J) GO TO 1
       WRITE(*, *) 'Не угадали!'
   1   WRITE(*, *) 'Угадали!'
END PROGRAM condg
```

# Организация циклов

В программировании часто возникает необходимость многократного повторения одних и тех же действий. Например, для вычисления массива `R` из 9 чисел вида 1/n придется записать девять операторов:

```
R(1) = 1.; R(2) = 1./2.; ….. R(9) = 1./9.
```

При необходимости выполнения итераций — вычислений рекуррентного типа — так же придется несколько раз повторять почти идентичный набор операторов несколько раз. Так, для вычисления суммы всех элементов полученного массива `R` необходимо 8 раз повторить сложение:

```
SUM = R(1); SUM = SUM + R(2); SUM = SUM + R(3);….SUM = SUM + R(9)
```

Безрадостность такого повторения легко устраняется организацией *цикла операторов* с помощью подручных средств:

```
I = 0; SUM = 0.
1      I = I + 1
       R(I) = 1. / I
       SUM = SUM + R(I)
IF(I .LT. 9) GO TO 1
```

Однако при самодеятельной организации цикла легко сделать ошибку, например, неправильно поставить метку, забыть оператор приращения шага, неправильно записать оператор перехода и т. д. В Фортране есть встроенное средство для организации циклов — конструкция `DO`. Решение двух предыдущих задач с этой конструкцией выглядит предельно естественно:

```
SUM = 0.
DO I = 1, 10
       R(I) = 1. / I
       SUM = SUM + R(I)
END DO
```

В общем виде конструкция DO имеет вид:

```
[имя:]  DO  [ список_цикла ]
            блок_действий
        END DO [имя]
```

Имя является необязательным элементом, но если оно появилось в заголовке конструкции, оно обязательно должно быть указано и в операторе завершения. Кроме имени и ключевого слова DO, в заголовке конструкции может присутствовать необязательный список_цикла, определяющий число шагов (или итераций) цикла.

**Внимание**

Если в заголовке конструкции DO список цикла отсутствует, цикл становится бесконечным.

Для выхода из цикла используется оператор EXIT; это единственно возможный способ выхода из бесконечного цикла:

```
I = 0
DO
       I = I + 1
       ……
       IF (I .GT. 1001) EXIT
END DO
```

Посмотрите в листинге 6.7, как зациклить игру по угадыванию числа до получения положительного результата (или до останова программы системными средствами).

**Листинг 6.7. Пример бесконечного цикла с условным выходом**

```
PROGRAM conddo
PARAMETER (I = 5)
DO
       WRITE(*, *) 'Угадайте число: I ='
       READ(*, *) J
       IF(I .LT. J) THEN
              WRITE(*, *) 'Не угадали, I меньше!'
       ELSE IF(I .GT. J) THEN
              WRITE(*, *) 'Не угадали, I больше!'
       ELSE
              WRITE(*, *) 'Угадали!'
              EXIT
```

```
      END IF
END DO
END PROGRAM conddo
```

Оператор EXIT в общем виде имеет атрибут

```
EXIT [имя]
```

Присутствующее имя указывает, из какой именно конструкции в комплексе вложенных конструкций DO нужно выйти. Если имени нет, управление передается ближайшему по уровню вложенности оператору END DO.

Другой оператор, употребляемый в конструкции DO, передает управление либо именованному, либо ближайшему оператору END DO:

```
CYCLE [имя]
```

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

В общем случае, список_цикла имеет вид:

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

Здесь переменная — это описанная ранее 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, его величина считается равной единице. Таким образом, число повторений *блока действий* равно числу значений, принимаемых переменной цикла. Если конечное_значение меньше, чем начальное_значение, значение *переменной* устанавливается равным 1, но блок_действий не выполняется ни разу, и управление передается оператору, следующему за оператором END DO.

Мы можем ограничить количество попыток и гарантировать завершение программы угадывания числа, если дополним цикл *списком цикла* (листинг 6.8).

**Листинг 6.8. Пример списочного цикла**

```
PROGRAM listdo
IMPLICIT NONE
INTEGER, PARAMETER :: I = 5
INTEGER J, K
DO K = 1, 10
       WRITE(*, *) 'Угадайте число: I ='
       READ(*, *) J
       IF(I .LT. J) THEN
```

```
            WRITE(*, *) 'Не угадали, I меньше!'
      ELSE IF(I .GT. J) THEN
            WRITE(*, *) 'Не угадали, I больше!'
      ELSE
            WRITE(*, *) 'Угадали!'
            EXIT
      END IF
END DO
END PROGRAM listdo
```

Все параметры заголовка цикла являются переменными целого типа, и любая из них, если только она не задана буквальной константой, доступна как внутри конструкции `DO`, так и вне ее. Если во время работы программы `конечное_значение` становится отрицательным или нулевым, переменная цикла принимает значение 1, а выполнение цикла прекращается. Непреднамеренная замена значения шага приращения переменной цикла нулем приведет к зацикливанию программы: цикл будет исполняться вечно. В этом смысле работа с конструкцией `DO` требует от программиста внимания и аккуратности. Самые простые приемы, гарантирующие предсказуемое выполнение цикла, — *никогда* не использовать переменные-параметры цикла вне цикла, задавать границы изменения и шаг переменной цикла буквальными или именованными константами, не всегда применимы.

Конструкция `DO` может быть записана в иной форме:

```
DO метка список_цикла
блок_действий
 метка END DO
```

или как

```
DO метка список_цикла
блок_действий
 метка CONTINUE
```

Например,

```
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 Фортрана — просто передает управление следующему оператору и используется не только в конструкции DO, но и во многих других, например, как оператор-адресат оператора перехода.

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

```
DO I = 1, N
       A(I) = 0.
       DO J = 1, M
              A(I) = A(I) + B(J)
       END DO
END DO
```

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

```
DO 20 I = 1, L
   DO 10 J = I, K
……….
   10 CONTINUE
20 CONTINUE
```

Альтернативная (и избыточная) форма записи является аналогом частного случая конструкции DO — бесконечного цикла с выходом по условию:

```
DO
……
IF(скалярное_логическое_выражение) EXIT
END DO
```

Она имеет вид:

```
DO метка WHILE (.NOT. скалярное_логическое_выражение)
……
 метка CONTINUE
```

Цикл из программы угадывания числа в версии DO…WHILE выглядел бы так (листинг 6.9).

**Листинг 6.9. Пример цикла в форме *DO…WHILE***

```
PROGRAM dowhile
IMPLICIT NONE
INTEGER, PARAMETER :: I = 5
INTEGER J, K
K = 1
```

```
DO 1 WHILE(K < 11)
       WRITE(*, *) 'Угадайте число: I ='
       READ(*, *) J
       IF(I .LT. J) THEN
              WRITE(*, *) 'Не угадали, I меньше!'
       ELSE IF(I .GT. J) THEN
              WRITE(*, *) 'Не угадали, I больше!'
       ELSE
              STOP 'Угадали!'
       END IF
       K = K + 1
1 CONTINUE
WRITE(*, *) 'Вам повезет в другой раз.'
END PROGRAM dowhile
```

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. В качестве переменных в неявных DO-циклах могут использоваться только переменные охватывающих вложенных циклов:

```
DATA((B(J, K), J = 1, 8), K = J, 8)
```

# Вложения управляющих конструкций и операторы перехода

Различные управляющие конструкции можно вкладывать друг в друга на неограниченную глубину. 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 * I
       IF(A(I) .GT. 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 .EQ. 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
```

Но недопустима следующая конструкция:

```
IF (T .NE. 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 подчиняется трем основным правилам:

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

Пример:

```
      GO TO 10       !ошибка: переход внутрь управляющей конструкции
 2    C = A + B
……….
      IF (G .GE. -2.) THEN
……….
```

```
      GO TO 2         !переход на внешний оператор
 10   F = V * D
……..
      GO TO 20        !ошибка: переход на оператор ELSE недопустим
……..
      GO TO 10        !переход внутри одного блока управляющей конструкции
 20   ELSE
……..
      GO TO 10        !ошибка: переход в другой блок конструкции
……..
      GO TO 3         !переход на завершающий оператор конструкции
……..
      GO TO 4         !переход на внешний оператор
……..
 3    END IF
……..
 4    G = B + C
```

# Вопросы и задания для самостоятельной работы

1. Перепишите программу угадывания числа, используя конструкцию `SELECT`.
2. Определите, какие из следующих выражений верны, а какие — нет:

```
а)      IF(A > B) C = A - B
        END IF
б)      IF(J < C) STOP 'J < C'
в)      IF(K < 100)THEN
            IF(L > K) THEN
                IF(J == I) THEN
                    C = A + B
                    F = G * H
                END IF
            ELSE
                C = A + F
            END IF
        ELSE STOP 2
г)      WHILE(X < Y) DO
            X = X**2
        END DO
```

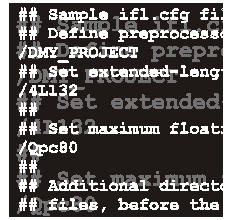
```
д)        DO I = 1, 20
                C = 21 / I
     2          GO TO 1
                IF(C > B) THEN
                       B = D + E
     1                 C = SIN(B)
                ELSE
                       GO TO 2
                END IF
          END DO
```

3. Используя управляющие конструкции, напишите программы, считывающие целую величину N и вычисляющие первые N
   - простых чисел;
   - чисел Фибоначчи (первые два числа равны единице, каждое следующее равно сумме двух предыдущих);
   - строк таблицы биномиальных коэффициентов в виде треугольника Паскаля:

```
                    1
               1         1
          1         2         1
     1         3         3         1
1         4         6         4         1
```

..... и т. д.

4. Напишите программу, которая вычисляет N первых простых чисел; модифицируйте программу так, чтобы N считывалось в процессе исполнения.
5. Напишите программу, меняющую порядок элементов массива в заданном диапазоне на обратный.




# Глава 7

# Структура программы. Подпрограммы и модули

Программа, написанная на Фортране, может полностью помещаться в одном блоке, такие программы уже встречались нам в предыдущих главах. Но, как правило, реальные программы намного объемнее приводимых здесь примеров, и размещение их в одном блоке затрудняет работу программиста. С другой стороны, любая задача имеет свою логическую структуру и может быть разделена на относительно самостоятельные подзадачи. Для каждой подзадачи можно выделить отдельный блок, — такое размещение способствует разделению работы над программой на независимые этапы, локализации проблем и оперативному внесению изменений. Блоки, выделенные для подзадач, называются программными компонентами; их структура зависит от поручаемых им действий. В Фортране выделены три вида независимых программных компонент: *главная программа*, *внешняя подпрограмма* и *модуль*. Все элементы программы в Фортране сосредоточены внутри программных компонент, любая информация, размещенная вне главной программы, подпрограмм и модулей, недоступна. Частью независимых программных компонентов могут быть *внутренние подпрограммы* — подпрограммы, описание которых заключено внутри компонента-носителя.

## Главная программа

Главная программа является обязательным компонентом любой программы. Это тот набор операторов, которому операционная система передает управление при запуске программы на исполнение. Главная программа должна быть единственной, иначе у операционной системы не будет возможности определить точку входа — место, с которого начинается исполнение всей программы. Структура главной программы такова:

```
[PROGRAM имя_программы]
[операторы_описания]
[исполняемые_операторы]
[CONTAINS
```

```
        внутренние_подпрограммы]
END [PROGRAM [имя_программы]]
```

Как мы видим, единственным обязательным элементом главной программы является оператор `END`, все остальные элементы могут отсутствовать. Однако, отличительным признаком главной программы является *отсутствие* оператора "`SUBROUTINE`" в ее заголовке: именно это характеризует данный компонент как главную программу. Оператор `END` главной программы не только отмечает конец программного компонента, но и является сигналом к остановке всей программы. Одним из признаков хорошей организации программы является возврат управления в главную программу на завершающей стадии работы, но если необходимо остановить программу в другом месте, следует использовать оператор `STOP` *(см. гл. 6)*.

Если оператор `END` содержит `имя_программы`, то оно должно совпадать с именем, указанным оператором `PROGRAM` в заголовке.

Блок `Операторы_описания`, если он есть, должен предшествовать блоку `исполняемые_операторы`. Последние также могут отсутствовать (в главной программе есть только один обязательный элемент), но смысла в такой программе будет не больше, чем в программе, состоящей из единственного оператора `END`.

Оператор `CONTAINS` отделяет исполняемые операторы от описания внутренних подпрограмм. `Внутренние подпрограммы` являются еще одной разновидностью программных компонентов Фортрана, однако их нельзя считать самостоятельными программными единицами. Внутренние подпрограммы могут использоваться только тем программным компонентом, в котором они описаны, извне компонента-носителя они недоступны.

# Внешние подпрограммы

Внешние подпрограммы обычно служат для выполнения отдельных задач в рамках общей задачи программы в целом. Внешние подпрограммы вызываются из главной программы или из других подпрограмм и при завершении возвращают управление вызывающему компоненту. Структура внешней подпрограммы похожа на структуру главной программы, за исключением заголовка:

```
Заголовок_подпрограммы
[операторы_описания]
[исполняемые_операторы]
[CONTAINS
        внутренние_подпрограммы]
END [спецификатор_подпрограммы [ имя_подпрограммы]]
```

Здесь Заголовок_подпрограммы имеет вид:

```
[префикс_спецификатора] спецификатор_подпрограммы имя_подпрограммы
[(список_формальных_параметров)]
```

где спецификатор_подпрограммы — один из операторов: SUBROUTINE или FUNCTION; имя_подпрограммы — уникальное имя, удовлетворяющее правилам Фортрана, за которым может следовать необязательный список_формальных_параметров, заключенный в круглые скобки. Спецификатор подпрограммы указывает, является ли подпрограмма *процедурой* или *функцией*. Процедура не имеет возвращаемого значения: результаты ее работы передаются либо через формальные параметры, либо через глобальные переменные. Функция же возвращает значение заданного типа, которое может быть присвоено переменной или подставлено в выражение. Поэтому префикс_спецификатора для функций содержит тип возвращаемого значения.

Обращение к процедуре осуществляется оператором CALL, имеющим вид:

```
CALL имя_подпрограммы [(список_фактических_параметров)],
```

например,

```
CALL BIGSUB
```

Вызов функции не требует никакого специального оператора:

```
REAL A, X
REAL, EXTERNAL :: MYFUNC
.....
      A = MYFUNC(X)
```

Оператор END подпрограммы возвращает управление тому программному компоненту, из которого подпрограмма была вызвана. Есть способ сделать это, не дожидаясь конца подпрограммы — оператор RETURN прекратит выполнение подпрограммы в любом ее месте и передаст управление в место вызова. Как и оператор STOP, он может иметь метку и быть частью условного оператора. Оператор RETURN не может появляться в главной программе и в подпрограммах-функциях.

Если в операторе END указано имя подпрограммы или функции, оно должно совпадать с именем, указанным в заголовке. Пример главной программы с внешней подпрограммой приведен в листинге 7.1.

**Листинг 7.1. Пример главной программы с внешней подпрограммой**

```
PROGRAM main
IMPLICIT NONE
REAL X, Y, S
DATA X, Y /0.755545, 6.873452/
CALL outer_sub(X, Y, S)
```

```
WRITE(*, *) 'Result of outer_sub S =', S
END PROGRAM main
SUBROUTINE outer_sub(A, B, C)
REAL A, B, C
C = SQRT(A**2 + B**2)
END SUBROUTINE outer_sub
```

# Внутренние подпрограммы

Внутренние подпрограммы, в отличие от внешних, не могут содержать собственных внутренних подпрограмм и имеют вид:

```
Заголовок_подпрограммы
[операторы_описания]
[исполняемые_операторы]
END [спецификатор_подпрограммы [имя_подпрограммы]]
```

Внутренняя подпрограмма автоматически получает доступ ко всем объектам своего носителя, в том числе, к другим внутренним подпрограммам. Все обращения к внутренним подпрограммам могут производиться только из компонента-носителя. Пример внутренней подпрограммы в главной программе приведен в листинге 7.2.

**Листинг 7.2. Пример главной программы с внутренней подпрограммой**

```
PROGRAM main
IMPLICIT NONE
REAL X, Y, S
DATA X, Y /0.755545, 6.873452/
CALL inner_sub(X, Y, S)
WRITE(*, *) 'Result of inner_sub S =', S
CONTAINS
       SUBROUTINE inner_sub(A, B, C)
       REAL A, B, C
       C = SQRT(A**2 + B**2)
       END SUBROUTINE inner_sub
END PROGRAM main
```

Приведенные здесь примеры слишком просты и не дают ответа на вопрос, зачем вообще нужны внутренние подпрограммы. Однако они имеют ряд очевидных достоинств, весьма удобных для программиста. В листинге 7.3 приведен пример программы, состоящей из двух компонент с внутренними

подпрограммами, имена которых совпадают. Тем не менее, каждый компонент уверенно вызывает только свою подпрограмму.

**Листинг 7.3. Внутренние подпрограммы**

```
PROGRAM main
IMPLICIT NONE
REAL X, Y, S
DATA X, Y /0.755545, 6.873452/
CALL inner_sub(X, Y, S)
WRITE(*, *) 'Result of main''s inner_sub S =', S
CALL outer_sub(X, Y, S)
WRITE(*, *) 'Result of outer_sub S =', S
CONTAINS
       SUBROUTINE inner_sub(A, B, C)
       REAL, PARAMETER :: R = 5.,RR = SQRT(5. / 2.0)
       REAL A, B, C
       C = SQRT(A**2 + B**2)
       IF (C > R) THEN
              IF(A > RR) A = RR
              IF(B > RR) B = RR
       END IF
       END SUBROUTINE inner_sub
END PROGRAM main
SUBROUTINE outer_sub(A, B, C)
REAL A, B, C
REAL S
CALL inner_sub(A, B, S)
WRITE(*, *) 'Result of outer''s inner_sub S =', S
C = SQRT(A**2 + B**2)
CONTAINS
       SUBROUTINE inner_sub(A, B, C)
       REAL A, B, C
       C = SQRT(ABS(A**2 - B**2))
       END SUBROUTINE inner_sub
END SUBROUTINE outer_sub
```

# Интерфейсы

Для организации вызова любой подпрограммы необходимо и достаточно знать ее *интерфейс*: заголовок, список формальных параметров и их описания. Интерфейс внутренней подпрограммы всегда известен в области ее использования: компонент-носитель знает все о параметрах внутренней подпрограммы; другим же программным компонентам внутренние подпрограммы недоступны. Интерфейс внутренней подпрограммы представляет образец *явного* интерфейса: проверить корректность обращения к ней можно еще на стадии компиляции. Другим примером явного интерфейса являются модульные подпрограммы и встроенные функции.

Внешняя подпрограмма и формальная процедура *(см. разд. "Подпрограммы как параметры" настоящей гл.)*, как правило, не обладают явным интерфейсом. У компилятора нет сведений о параметрах внешних и формальных процедур, т. е. их интерфейс является *неявным*. Поэтому на этапе компиляции невозможно, например, выполнить проверку на соответствие между фактическими и формальными параметрами.

Сделать неявный интерфейс явным можно, описав его в интерфейсном блоке:

```
INTERFACE
       заголовок_подпрограммы
       описание_формальных_параметров
       [описание_локальных_переменных]
       оператор_END_подпрограммы
END INTERFACE
```

В интерфейсный блок нельзя включать операторы `DATA` и `FORMAT`. Имена формальных параметров и порядок операторов описания можно, а в некоторых случаях, и нужно, менять. В интерфейсном блоке можно поместить описание интерфейса программы, написанной на другом языке программирования, например, на языке ассемблера или на языке C.

Интерфейсный блок размещается среди операторов описания; после этого интерфейс подпрограммы, содержащейся в нем, становится явным.

В листинге 7.4 интерфейс внешней подпрограммы сделан явным посредством включения интерфейсного блока в главную программу.

**Листинг 7.4. Интерфейсный блок**

```
PROGRAM main
IMPLICIT NONE
REAL X, Y, S
INTERFACE
       SUBROUTINE outer_sub(A, B, C)
```

```
       REAL A, B, C
       END SUBROUTINE outer_sub
END INTERFACE
DATA X, Y /0.755545, 6.873452/
CALL inner_sub(X, Y, S)
WRITE(*, *) 'Result of main''s inner_sub S =', S
CALL outer_sub(X, Y, S)
WRITE(*, *) 'Result of outer_sub S =', S
CONTAINS
       SUBROUTINE inner_sub(A, B, C)
       REAL, PARAMETER :: R = 5., RR = SQRT(5. / 2.0)
       REAL A, B, C
       REAL, INTENT(OUT) :: C
       C = SQRT(A**2 + B**2)
       IF (C > R) THEN
              IF(A > RR) A = RR
              IF(B > RR) B = RR
       END IF
       END SUBROUTINE inner_sub
END PROGRAM main
```

В следующих разделах этой главы мы рассмотрим использование интерфейсов для создания родовых имен и переопределения встроенных операций.

# Параметры подпрограмм

Все подпрограммы — и процедуры, и функции, — должны иметь возможность обмениваться данными с теми компонентами программы, из которых они были вызваны. Подпрограммы получают входные данные от вызывающего компонента и передают ему выходные данные — результаты своей работы. Обмен информацией с подпрограммами может происходить через параметры подпрограмм и/или через глобальные данные. Глобальные данные — это переменные, доступные в равной степени и подпрограмме, и вызывающему компоненту. Организация глобальных данных будет рассмотрена в разделе "Модули" настоящей главы. Другой способ обмена информацией — передача данных через параметры подпрограмм. В этом случае в описании подпрограммы указывается *список_формальных_параметров* — перечень переменных (среди них могут быть и имена функций), которые так или иначе будут использоваться в ходе выполнения подпрограммы. Формальные параметры должны быть (явно или неявно) описаны в операторах описания подпрограммы. При вызове подпрограммы происходит подстановка фактических параметров на место формальных (листинг 7.5).

**Листинг 7.5. Формальные и фактические параметры**

```
SUBROUTINE sub(A, B, C) !A, B, C - формальные параметры
REAL A, B, C
       C = SQRT(A**2 + B**2)
END SUBROUTINE sub
PROGRAM main
IMPLICIT NONE
REAL X, Y, S
DATA X, Y /0.755545, 6.873452/
       CALL sub(X, Y, S) ! X, Y, S - фактические параметры
       WRITE(*, *) 'Result of sub S =', S
END PROGRAM main
```

## Ограничения, накладываемые на фактические параметры

Фактические параметры обязаны согласовываться с соответствующими формальными параметрами по типу, параметрам типа и по форме. Совпадения имен параметров не требуется. Более того, на место формального параметра может быть подставлено выражение, тип и разновидность типа которого совпадают с типом и разновидностью формального параметра.

Предполагается, что формальные параметры являются отдельными объектами, не совпадающими ни с какими другими объектами подпрограммы. Компилятор может скопировать входной параметр во внутреннюю переменную при входе в подпрограмму с тем, чтобы на выходе из нее вернуть сохраненное значение фактическому параметру. Поэтому на фактические параметры накладываются следующие ограничения:

- любые действия, влияющие на фактический параметр в обход формального параметра, запрещены;
- если хотя бы часть фактического параметра получает значение через формальный параметр, то ссылаться на этот фактический параметр можно только через формальный параметр.

То есть если в подпрограмме введен формальный параметр `A`, то в рамках подпрограммы он будет доступен только как `A`, независимо от того, какой фактический параметр подставляется на его место. Возможность доступа к фактическому параметру в обход формального появляется, например, тогда, когда фактическим параметром становится глобальная переменная из присоединенного модуля или переменная, доступная через `COMMON`-блок.

## Вид связи формального параметра

В Фортране можно указать назначение формального параметра. Является ли формальный параметр входным, выходным или универсальным, определяет атрибут `INTENT(параметр)`, где `параметр` принимает значения `IN`, `OUT` и `INOUT`:

```
SUBROUTINE STUFF(N, PIN, RESULT)
INTEGER, INTENT(IN) :: N
REAL, INTENT(OUT) :: PIN
REAL, DIMENSION(:), INTENT(INOUT) :: RESULT
```

Значение `IN` определяет параметр как входной: такой параметр не должен измениться во время работы подпрограммы. То есть он не может появиться в левой части оператора присваивания внутри подпрограммы, его нельзя передать фактическим параметром в подпрограмму, которая его переопределит, и т. д.

Значение `OUT` определяет параметр как выходной, т. е. параметр *должен* быть переопределен в подпрограмме, поэтому при входе в подпрограмму он переходит в состояние неопределенности. Соответствующий фактический параметр обязан быть переменной.

Универсальным параметрам соответствует значение `INOUT`. Предполагается, что значение универсального параметра будет переопределено, поэтому соответствующий фактический параметр должен быть переменной.

Если вид связи формального параметра не указан, то фактическим параметром может быть и переменная, и выражение. Но если значение формального параметра в подпрограмме переопределяется, то фактический параметр обязан быть переменной.

Рекомендуется указывать вид связи параметра везде, где это возможно. Однако если формальный параметр имеет атрибут `POINTER`, указать для него вид связи не удастся, так как невозможно определить, к чему именно относится вид связи: к адресату ссылки или к состоянию привязанности ссылки. В листинге 7.6 параметры подпрограммы `sub` помечены как входные и выходные.

**Листинг 7.6. Вид связи формальных параметров**

```
SUBROUTINE sub(A, B, C) !A, B, C - формальные параметры
REAL, INTENT(IN) :: A, B
REAL ,INTENT(OUT) :: C
       C = SQRT(A**2 + B**2)
END SUBROUTINE sub
```

## Ссылки как формальные и фактические параметры

Если формальный параметр имеет атрибут POINTER, фактический параметр обязан быть ссылкой. Ранг фактического параметра обязан совпадать с рангом формального параметра. При обращении к подпрограмме состояние привязки фактического параметра передается формальному параметру; при выходе фактический параметр перенимает состояние привязки формального параметра. Если последний имеет адресатом локальную переменную без атрибута SAVE, фактический параметр окажется привязанным к адресату, находящемуся в состоянии неопределенности.

Фактический параметр может быть ссылкой и тогда, когда формальный параметр ссылкой не является. В этом случае с формальным параметром будет ассоциирован адресат ссылки-фактического параметра.

## Параметры с атрибутом *TARGET*

Если фактический параметр является адресатом ссылки, ее состояние при входе в подпрограмму не меняется, т. е. ссылка к соответствующему формальному параметру не привязывается. Если же формальный параметр имеет атрибут TARGET, любая ссылка, прикрепленная к нему, по выходе из подпрограммы становится неопределенной.

## Подпрограммы как параметры

Параметрами подпрограмм могут быть не только переменные и выражения, но и процедуры. Представим себе подпрограмму, задачей которой является вычисление интеграла произвольной функции на заданном отрезке. Имя подпрограммы, вычисляющей значения функции, не может быть определено заранее и включено в текст подпрограммы вычисления интеграла, его нужно передать через параметры подпрограммы. В списке формальных параметров указывается формальное имя подпрограммы-функции, в операторах описания этому параметру назначается атрибут EXTERNAL. Интерфейс формальной подпрограммы описывается в интерфейсном блоке (листинг 7.7).

**Листинг 7.7. Описание интерфейса внешней подпрограммы как формального параметра**

```
REAL FUNCTION INTEGRAL(A, B, FUNC)
INTERFACE
       REAL FUNCTION FUNC(X)
              REAL, INTENT(IN) :: X
       END FUNCTION FUNC
END INTERFACE
REAL, INTENT(IN) :: A, B
```

```
REAL X, F
……..……
F = FUNC(X)
……..……
END FUNCTION INTEGRAL
```

При вызове функции `INTEGRAL` на место `FUNC` в формальных параметрах подставляется реальная функция, интерфейс которой тоже желательно описать (листинг 7.8).

**Листинг 7.8. Описание интерфейса внешней подпрограммы как фактического параметра**

```
PROGRAM INTEGR
INTERFACE
       REAL FUNCTION MYFUN(X)
       REAL, INTENT(IN) :: X
       END FUNCTION MYFUN
END INTERFACE
REAL F
……
F = INTEGRAL(0.5, 2.5, MYFUN)
……
END PROGRAM INTEGR
REAL FUNCTION MYFUN(X)
REAL, INTENT(IN) :: X
……
END FUNCTION MYFUN
```

Передаваемая функция должна быть внешней или модульной. Не допускается передача внутренних процедур в качестве параметров, так как они могут реализовываться встроенным кодом, что в принципе не годится для динамической передачи управления. Нельзя в параметрах указывать родовое имя процедуры *(см. разд. "Именованные интерфейсы и совмещение процедур" настоящей гл.)*.

**Примечание**

Фортран разрешает неявное определение формальной подпрограммы: компилятор определит формальный параметр как имя подпрограммы по оператору вызова. Пользоваться определением формальной подпрограммы по умолчанию решительно не рекомендуется, так как контроль корректности вызова при этом полностью утрачивается.

## Ключевые и необязательные параметры

Обычно при вызове подпрограмм фактические параметры подставляются на место соответствующих формальных параметров; назначение параметра определяется его позицией в списке аргументов оператора вызова. Такой метод задания фактических параметров называется *позиционным*. Но есть другой способ установить соответствие фактического параметра формальному: передать фактические параметры в виде последовательности пар:

```
имя_формального_параметра = фактический_параметр
```

При таком способе передачи порядок пар не имеет значения; этот метод называется *ключевым*.

Например, подпрограмму:

```
SUBROUTINE KEY(NUM, RECORD, FILE)
INTEGER, INTENT(IN) :: NUM, FILE
CHARACTER(LEN = *),INTENT(IN) :: RECORD,
```

для параметров:

```
NUMBER = I
TALE = 'Запись в файл'
WR = 11,
```

можно вызвать так:

```
CALL KEY(NUMBER, TALE, WR)
```

Это — позиционный вариант вызова. В ключевом варианте вызова порядок параметров может быть любым:

```
CALL KEY(RECORD = TALE, FILE = WR, NUM = NUMBER)
```

Ключевой и позиционный методы могут смешиваться. Так, например, при вызове подпрограммы часть параметров может быть опущена, если, например, подпрограмма предусматривает замену недостающих параметров некоторыми значениями по умолчанию. Такие параметры называются необязательными; соответствующие им формальные параметры отмечаются атрибутом `OPTIONAL`. Так, например, в описании подпрограммы `SUBJ` последний формальный параметр является необязательным:

```
SUBROUTINE SUBJ(STRING, LENGTH, IBEGIN)
CHARACTER, DIMENSION(:) :: STRING
INTEGER LENGTH
INTEGER, OPTIONAL :: IBEGIN
```

Предположим, что подпрограмма `SUBJ` выделяет из символьного массива `STRING` подстроку длиной `LENGTH` символов, начиная с символа с индексом `IBEGIN`. Если в списке фактических параметров при вызове подпрограммы `SUBJ` этот параметр отсутствует, ему будет присвоено значение 1:

```
IF(.NOT. PRESENT(IBEGIN)) IBEGIN = 1
```

Встроенная функция *(см. гл. 10)* `PRESENT` возвращает `.TRUE.`, если указанный формальный параметр присутствует в фактических параметрах. Таким образом, подпрограмма имеет возможность проанализировать наличие необязательного параметра.

Если часть фактических параметров отсутствует, то определить по положению в списке, каким формальным параметрам соответствуют те фактические параметры, которые были переданы в подпрограмму, невозможно. Для "опознавания" фактических параметров, позиция которых не совпадает с позицией соответствующего формального параметра, применяются *ключи*. Так, если бы в подпрограмме `SUBJ` было два необязательных параметра:

```
SUBROUTINE SUBJ(STRING, LENGTH, IBEGIN, IEND)
CHARACTER, DIMENSION(:) :: STRING
INTEGER LENGTH
INTEGER, OPTIONAL :: IBEGIN, IEND,
```

то в вызове этой подпрограммы с набором фактических параметров, соответствующих формальным параметрам `STRING`, `LENGTH`, `IEND`, последний параметр было бы нужно передавать с ключом, например, так:

```
CALL SUBJ(STR, LEN, IEND = 20)
```

Параметры, соответствие которых может быть установлено по их положению, называются *позиционными*. Необязательные параметры, идентифицирующиеся ключами, называются *ключевыми*. Ни ключевые, ни позиционные параметры не могут появляться в списке фактических параметров более одного раза. Если параметров много, рекомендуется указывать ключи даже для позиционных параметров.

Наличие в подпрограмме хотя бы одного необязательного параметра обязывает ее иметь явный *интерфейс*.

# Именованные интерфейсы и совмещение процедур

Группу подпрограмм можно объединить под одним именем так, чтобы впоследствии любую из них можно было бы вызывать по их общему имени. Так, например, можно объединить функции, вычисляющие одну и ту же величину для параметров целого, вещественного и комплексного типа: обойтись одной подпрограммой для параметров разного типа невозможно, но обращения к ним можно унифицировать. Объединение процедур под одним именем традиционно называют *перегрузкой*, используя буквальный перевод английского слова *overloading*, хотя то, что при этом происходит, можно довольно точно определить как *совмещение*. Общее имя совмещенных подпрограмм называют *родовым* именем.

Для объединения подпрограмм необходимо описать именованный интерфейс, содержащий заголовки всех процедур группы, например:

```
INTERFACE GAMMA
       REAL FUNCTION RGAMMA(X)
              REAL, INTENT(IN) :: X
       END FUNCTION RGAMMA
       COMPLEX FUNCTION CGAMMA(X)
              COMPLEX, INTENT(IN) :: X
       END FUNCTION CGAMMA
       INTEGER FUNCTION IGAMMA(X)
              INTEGER, INTENT(IN) :: X
       END FUNCTION IGAMMA
END INTERFACE GAMMA
```

Такой объединяющий интерфейс называется *родовым*, а его имя — *родовым именем*. В данном случае, родовое имя не совпадает ни с одним из имен совмещаемых подпрограмм, но это не обязательно: родовое имя вполне может совпадать с одним из имен внутренних подпрограмм интерфейса. Если в родовой интерфейс включаются модульные подпрограммы *(см. разд. "Модули" настоящей гл.)*, интерфейс которых всегда является явным и потому не требует повторения, достаточно внести в описание родового интерфейса оператор `MODULE PROCEDURE` с перечнем имен включаемых модульных подпрограмм:

```
INTERFACE GAMMA
.....
MODULE PROCEDURE MGAMMA, PGAMMA, DGAMMA
.....
END INTERFACE
```

Так как при обращении по родовому имени выбор конкретной подпрограммы будет осуществляться по набору фактических параметров, совмещаемые подпрограммы должны однозначно определяться своими формальными параметрами. Их параметры обязаны разниться или числом, или типом, или разновидностью типа. Все объединяемые подпрограммы обязаны быть либо процедурами, либо функциями. Имена функций или процедур и порядок следования их формальных параметров не имеют значения. Функции:

```
REAL FUNCTION FUN(X, I)
REAL X
INTEGER I
```

и

```
REAL FUNCTION UNF(I, X)
INTEGER I
REAL X
```

с точки зрения родового интерфейса неразличимы: специфическое имя функции при вызове по родовому имени недоступно, а параметры определяют функцию только в позиционном варианте вызова — `(Y, K)`. В ключевом же варианте — `(X = Y, I = K)`, — вызов одинаков для обеих функций.

Если несколько родовых интерфейсов имеют одно родовое имя, то под этим именем становятся доступными все подпрограммы, заключенные в них. Это очень удобно, так как позволяет по мере надобности расширять уже имеющуюся родовую группу.

Другой формой совмещения является расширение встроенных операций и присваивания при определении их для производных типов. Для переопределения (перегрузки) встроенных операций так же описывается именованный интерфейсный блок с родовым именем `OPERATOR(знак_операции)`, например:

```
INTERFACE OPERATOR(+)
тело_интерфейса
END INTERFACE,
```

где `тело_интерфейса` содержит заголовки функций, замещающих обозначенный в заголовке интерфейса знак встроенной операции, или оператор-ссылку `MODULE PROCEDURE` к соответствующим модульным функциям. Функции, расширяющие встроенную бинарную операцию, должны иметь два формальных входных (атрибут `INTENT(IN)`) параметра; функции, расширяющие унарную операцию, должны иметь один такой параметр. Число аргументов обязано совпадать с числом операндов встроенной операции, например, унарную операцию `*` определить нельзя.

Для расширения присваивания задается интерфейс с родовым именем `ASSIGNMENT(=)`:

```
INTERFACE ASSIGNMENT(=)
тело_интерфейса
END INTERFACE
```

Подпрограммы, заголовки которых включаются в `тело_интерфейса`, должны быть процедурами с двумя обязательными параметрами, первый из которых имеет вид связи `OUT` или `INOUT`, а второй — вид связи `IN`. При перегрузке встроенных функций или процедур правила, обеспечивающие однозначность вызовов, должны применяться к набору специфических подпрограмм, параметры которых обеспечивают все допустимые комбинации типа, разновидности типа и ранга аргументов переопределяемой функции.

# Модули

Если целью внутренних и внешних подпрограмм является выполнение действий, определяемых подзадачей в рамках всей программы (внешние подпрограммы) или данного программного компонента (внутренние подпрограммы),

то задачей модуля является аккумуляция данных. В ранних версиях Фортрана возможности доступа к единому блоку данных из нескольких программных компонентов не было; области общей памяти, где хранятся глобальные переменные, должны были описываться в каждой компоненте заново. Это очень затрудняло использование глобальных данных в программах и служило неиссякаемым источником ошибок. Директива компилятора `INCLUDE`, появившаяся в Фортране 77, позволила свести описание глобальных данных в один файл и устранить источник противоречий между глобальным содержанием и его локальными описаниями. Однако директивы `INCLUDE` нельзя рассматривать как оператор Фортрана: она исполняется компилятором. Полностью проблема организации доступа к глобальным данным была решена только в Фортране 90, когда в языке появились модули. Структура модуля такова:

```
MODULE имя_модуля
       [операторы_описания]
[CONTAINS
модульные_подпрограммы]
END [MODULE [имя_модуля]]
```

Модуль включается в другие программные компоненты оператором `USE`. Все объекты модуля, не имеющие атрибута `PRIVATE`, могут использоваться во включающих компонентах.

Оператор `USE` имеет возможность ограничить доступ к частям модуля или переименовать объекты, доступ к которым разрешен. (Подробнее об операторе `USE` будет рассказано в одном из следующих разделов.) Если ограничений не установлено, все объекты модуля доступны. `Модульные_подпрограммы` имеют структуру внешних подпрограмм, за исключением того, что оператор `END`, завершающий их описание, *обязан* содержать слово `SUBROUTINE` или `FUNCTION` в соответствии с видом подпрограммы. Все объекты модуля автоматически доступны для модульных подпрограмм.

Модули можно вкладывать друг в друга, используя все тот же оператор `USE`, при этом модуль не должен ссылаться сам на себя ни прямо, ни косвенно — через цепочку модулей.

Самый простой пример использования модуля — это помещение в него описаний данных. Описываем модуль `ARRAY` и помещаем в него описание некоторого массива:

```
MODULE ARRAY
       REAL, DIMENSION(20) :: A
END MODULE ARRAY,
```

и затем включаем этот модуль в набор подпрограмм `SUB1`, `SUB2` и `SUB3`:

```
SUBROUTINE SUB1
USE ARRAY
```

```
.....
END SUBROUTINE SUB1
```

и так далее. Массив A будет доступен для всех трех подпрограмм:

```
PROGRAM SUBS
       CALL SUB1
       CALL SUB2
       CALL SUB3
END PROGRAM SUBS
```

Модуль является идеальным местом для размещения определений производных типов и операций с ними, интерфейсных блоков и списков NAMELIST.

# Рекурсивные процедуры и функции

Предполагается, что подпрограммы не содержат обращений к себе: по умолчанию рекурсия подпрограмм запрещена. Подпрограмму можно объявить рекурсивной, для этого нужно внести в ее заголовок декларацию RECURSIVE:

```
RECURSIVE SUBROUTINE FACTORIAL(N, NF)
```

тогда обращения из подпрограммы к самой себе будут разрешены. При создании рекурсивной подпрограммы необходимо построить ее алгоритм так, чтобы выход из нее гарантировался. Рассмотрим простейшую рекурсивную подпрограмму вычисления факториала целого числа N (листинг 7.9):

**Листинг 7.9. Пример рекурсивной процедуры**

```
RECURSIVE SUBROUTINE FACTORIAL(N, NF)
INTEGER, INTENT(IN) :: N
INTEGER, INTENT(INOUT) :: NF
DATA NF /1/
IF(N .GT. 1) THEN
       NF = NF * N
       CALL FACTORIAL(N - 1, NF)
END IF
END
```

При каждом вызове рекурсивной подпрограммы создается новый набор локальных объектов подпрограммы. Исключением являются объекты, имеющие атрибут SAVE, и объекты, начальное значение которых задается оператором DATA. Объекты, инициализируемые в подпрограммах, получают атрибут SAVE автоматически, а оператор DATA при рекурсивном вызове выполняется только один раз.

Рекурсивные функции несколько отличаются от рекурсивных процедур: если рекурсивная функция вызывает сама себя, результат ее должен иметь другое имя. Описанная здесь подпрограмма FACTORIAL является процедурой; сделаем из нее рекурсивную функцию (листинг 7.10).

**Листинг 7.10. Пример рекурсивной функции**

```
INTEGER RECURSIVE FUNCTION FACTORIAL(N) RESULT(NF)
INTEGER N, NF
DATA NF /1/
       IF(N .GT. 1) NF = N * FACTORIAL(N - 1)
END
```

Директива RESULT помещает результат рекурсивной функции в указанную переменную, решая проблему сохранения результата предыдущей итерации.

# Области видимости имен и меток

Для именованных объектов и меток существует пространство, в рамках которого они доступны по своему идентификатору. Это пространство называется областью видимости. Для каждой подпрограммы определен свой набор меток; даже внутренние подпрограммы имеют свой, независимый от компонента-носителя набор. То есть метки компонента-носителя и внутренней подпрограммы могут быть одинаковыми, они не будут смешиваться.

К именованным объектам относятся переменные, именованные константы, подпрограммы (внутренние и внешние), производные типы и списки NAMELIST.

Каждая подпрограмма определяет свое пространство имен, состоящее из всех объектов, описанных в ней. Это *локальные* объекты. Кроме локальных объектов, внутри подпрограммы могут использоваться *ассоциированные* (присоединенные) и *глобальные* объекты. Ассоциирование может происходить через носитель (подпрограмма является внутренней подпрограммой) или через включение в компонент модуля (Use-*ассоциирование*). Глобальные объекты — к ним относятся имена программных компонентов и внешние подпрограммы — не требуют объявления и автоматически доступны в любом месте программы.

Локальные именованные объекты доступны для всех операторов того программного компонента, в котором они определены, за вычетом интерфейсов. Блоки определения производных типов и внутренних подпрограмм имеют доступ к ассоциированным объектам через носитель, если только в них не объявлен локальный объект с тем же именем — локальные объекты

имеют приоритет перед объектами, ассоциированными через носитель. Для носителя локальные объекты блоков определения производных типов, внутренних подпрограмм и интерфейсов недоступны. Например, в программе INSIDE, переменная A доступна ассоциированием через подпрограмму-носитель OUTSIDE, а переменная B носителя недоступна, так как в INSIDE объявлена локальная переменная с тем же именем:

```
SUBROUTINE OUTSIDE
REAL A, B
.....
CONTAINS
       SUBROUTINE INSIDE
              REAL B, Y
              Y = A + B
       END SUBROUTINE INSIDE
END SUBROUTINE OUTSIDE
```

Имена Use-ассоциированных объектов использовать в описаниях локальных объектов, вообще говоря, нельзя. Оператор:

```
USE имя_модуля
```

рассматривается как повторное описание всех объектов модуля внутри программного компонента с сохранением своих имен и свойств. Другое дело, что возможности оператора USE позволяют обойти нежелательное совпадение имен объектов модуля и программы, использующей его.

Имя каждого именованного объекта внутри блока видимости должно быть уникальным. Например, определение внутренней подпрограммы с именем SIN сделает недоступной одноименную встроенную процедуру.

# Оператор *USE*

Оператор USE открывает программному компоненту доступ к переменным, подпрограммам, интерфейсам и другим объектам, заключенным в указанном модуле. Этот оператор относится к операторам описания *(см. гл. 3)*; все остальные операторы описания должны размещаться после него. Простейшая форма оператора USE:

```
USE имя_модуля
```

уже неоднократно упоминалась; теперь мы рассмотрим этот оператор более подробно.

При включении сторонних объектов в программу неизбежно встает вопрос о разрешении конфликтов имен, модульных и локальных, так как при совпадении имен использование модулей будет проблематичным. Оператор

в полной форме USE позволяет переименовывать объекты подключаемого модуля:

```
USE имя_модуля, список_переименований,
```

где список_переименований состоит из разделенных запятыми пар:

```
локальное_имя => модульное_имя
```

или из директивы:

```
ONLY: [only_список]
```

Список_переименований в простой форме позволяет назначать именам модульных объектов локальные имена, а все не переименованные объекты остаются видимыми под своими модульными именами. ONLY-форма списка переименований дает возможность выбирать из множества модульных объектов те, которые предполагается использовать в данной компоненте, все остальные объекты становятся недоступными и потому не могут конфликтовать с локальными объектами или объектами из других модулей.

Only_список содержит разделенные запятыми пары:

```
локальное_имя => модульное_имя
```

и/или модульные_имена объектов, доступных под своими модульными именами.

Рассмотрим варианты переименований на следующем примере. В листинге 7.11 помещено описание модулей A и B, содержащих описание разных функций с именем S.

**Листинг 7.11. Одноименная функция в модулях**

```
MODULE A
CONTAINS
       REAL FUNCTION S(X, Y)
       REAL, INTENT(IN) :: X,Y
              S = SIN(X) * COS(Y) + COS(X) * SIN(Y)
       END FUNCTION S
END MODULE A
MODULE B
CONTAINS
       REAL FUNCTION S(X, Y)
       REAL, INTENT(IN) :: X, Y
              S = COS(X) * COS(Y) - SIN(X) * SIN(Y)
       END FUNCTION S
END MODULE B
```

Программа из листинга 7.12 включает оба модуля и содержит собственную функцию с именем `S`.

**Листинг 7.12. Пример оператора *USE* со списком переименований**

```
PROGRAM prog
USE A, AS => S
USE B, BS => S
REAL X, Y
DATA X, Y /1.7, 1.5/
WRITE(*, *) 'Синус X + Y = ',AS(X, Y)
WRITE(*, *),'Косинус X + Y = ',BS(X, Y)
WRITE(*, *), 'Синус X - Y = ',S(X, Y)
CONTAINS
       REAL FUNCTION S(X, Y)
       REAL, INTENT(IN) :: X, Y
              S = SIN(X) * COS(Y) - COS(X) * SIN(Y)
       END FUNCTION S
END PROGRAM prog
```

Внутренняя функция `S` используется со своим именем, одноименные функции из модулей `A` и `B` используются под именами `AS` и `BS` соответственно. Если в одном из модулей определены несколько функций, а планируется использовать только их часть, можно применить `ONLY`-версию списка переименований (листинги 7.13 и 7.14).

**Листинг 7.13. Модуль с несколькими функциями**

```
MODULE A
CONTAINS
       REAL FUNCTION S(X, Y)
       REAL, INTENT(IN) :: X, Y
              S = SIN(X) * COS(Y) + COS(X) * SIN(Y)
       END FUNCTION S
       REAL FUNCTION R(X, Y)
       REAL, INTENT(IN) :: X, Y
              R = SIN(X / 2) * COS(Y) + COS(X) * SIN(Y / 2)
       END FUNCTION R
       REAL FUNCTION T(X, Y)
       REAL, INTENT(IN) :: X, Y
```

```
            T = SIN(X)**2 * COS(Y) + COS(X) * SIN(Y)**2
      END FUNCTION T
END MODULE A
```

**Листинг 7.14. *ONLY*-версия списка переименований оператора *USE***

```
PROGRAM prog
USE A, ONLY: AS => S, R !Из модуля A будут доступны 2 функции из 3-х.
USE B, BS => S
REAL X, Y
DATA X, Y /1.7, 1.5/
……….……
CONTAINS
      REAL FUNCTION S(X, Y)
      REAL, INTENT(IN) :: X, Y
            S = SIN(X) * COS(Y) - COS(X) * SIN(Y)
      END FUNCTION S
END PROGRAM prog
```

Теперь функция `S` модуля `A` будет доступна под именем `AS`, функция `R` — под своим собственным именем, а функция `T` не будет доступна.

# Области общей памяти

Необходимо отметить, что в Фортране существует еще одно средство глобализации данных — это области общей памяти, называемые `COMMON`-блоками. `COMMON`-блоки могут быть именованными и неименованными. Можно описать только один неименованный `COMMON`-блок для всей программы, именованных блоков можно описать множество. Область общей памяти описывается оператором `COMMON`, полный синтаксис которого имеет вид:

```
COMMON [/имя_общего_блока/] список [, /имя_общего_блока/] список…,
```

где `имя_общего_блока` — необязательное имя `COMMON`-блока, а `список` — список переменных, размещаемых в указанной области общей памяти. Приведем примеры именованного и безымянного блоков:

```
COMMON /BASE/ X, Y, Z, INDEX(10)
COMMON / / RESULT(200)
```

Переменные, входящие в `COMMON`-блок, должны быть описаны в программном компоненте, использующем этот блок. Если переменная включена в один `COMMON`-блок, она не может входить ни в какой другой `COMMON`-блок данного программного компонента. Объектами общих блоков не могут быть

именованные константы и формальные параметры, автоматические объекты, выделяемые массивы и функции. Дело в том, что COMMON-блок представляет собой отрезок памяти, куда последовательно записываются включенные в него объекты. Общая область доступна для всех программных компонентов, объявивших ее. При этом имена переменных именованного COMMON-блока в различных программных компонентах вовсе не должны совпадать — должно совпадать полное число единиц памяти, отведенных под него. Например, COMMON-блок /BASE/ в другой подпрограмме может выглядеть так:

```
COMMON /BASE/ A(3), I, J, K(8)
```

Длина неименованного COMMON-блока в различных программных компонентах может варьироваться. Компонент может дописывать в COMMON-блок свои объекты — программа будет использовать самый длинный вариант данного блока. Новые объекты можно только добавлять к общей области — удалить объекты из общей памяти невозможно.

Данные COMMON-блока становятся неопределенными после выхода из подпрограммы, описывающей его, если только другие подпрограммы или главная программа не используют его. Атрибут SAVE не может быть назначен объектам COMMON-блока по отдельности, но может быть назначен всему блоку в целом:

```
SAVE /BASE/
```

Следует заметить, что модули, предоставляя в распоряжение программиста все возможности общих областей памяти, свободны от их недостатков и нежелательных побочных эффектов; использование модулей полностью устраняет необходимость в COMMON-блоках.

Кроме глобального совмещения в памяти, в Фортране возможно локальное совмещение объектов в памяти, определяемое оператором EQUIVALENCE. Запись:

```
REAL A, B, C, D, E
EQUIVALENCE (A, B), (C, D, E)
```

приведет к тому, что под переменные A и B будет выделена одна область памяти. Переменные C, D и E будут помещены в другую, но общую для всех троих, область.

Операндами оператора EQUIVALENCE могут быть как отдельные элементы массивов, так и массивы без указания индексов. Вот пример совмещения в памяти элементов X(1) и Y(1):

```
REAL X(100), Y(10)
EQUIVALENCE (X, Y)
```

Нельзя совмещать объекты, использующие память разных типов:

```
REAL A
CHARACTER * 4 WORD
EQUIVALENCE (A, WORD) ! - ошибка! Недопустимое совмещение.
```

Но можно совмещать объекты разных типов, использующих однотипную (например, числовую) память:

```
INTEGER L(100)
REAL X(100)
EQUIVALENCE (L, X)
```

Такое совмещение уместно тогда, когда массивы `L` и `X` не используются одновременно. Размещая их в одной области, можно сэкономить общую память программы, но о проделанном трюке придется постоянно помнить.

Можно совмещать объекты разной длины:

```
CHARACTER(LEN = 4) V
CHARACTER(LEN = 6) W(2)
EQUIVALENCE (V, W(1)(5:))
```

Но нельзя совмещать подобъекты одного объекта так, чтобы они занимали одну и ту же область памяти:

```
EQUIVALENCE (W(1), V), (W(2), V) !Ошибка.
```

Совмещаемые объекты могут быть только переменными. Не допускается совмещение формальных параметров, ссылок, выделяемых массивов, функций, компонентов структур.

Переменные одного и того же производного типа могут совмещаться, если производный тип обладает атрибутом `SEQUENCE`, т. е. его компоненты размещены в памяти последовательно:

```
TYPE TRINITY
SEQUENCE
       REAL PARENT
       INTEGER INDEX
       COMPLEX SON
END TYPE TRINITY
```

Ассоциирование памяти для двух или более переменных приводят к тому, что любое изменение одной из них автоматически приведет к такому же изменению другой. Точно такой же результат можно получить, назначив `B` ссылкой на переменную `A`:

```
REAL, TARGET :: A
REAL, POINTER :: B

B => A
```

Экономической эффект от применения оператора `EQUIVALENCE` можно легко превзойти привлечением динамической памяти и автоматических массивов.

# Порядок операторов

Теперь, когда мы познакомились почти со всеми операторами Фортрана и со структурой всех программных компонентов, рассмотрим правила следования операторов внутри программной компоненты — главной программы или подпрограммы.

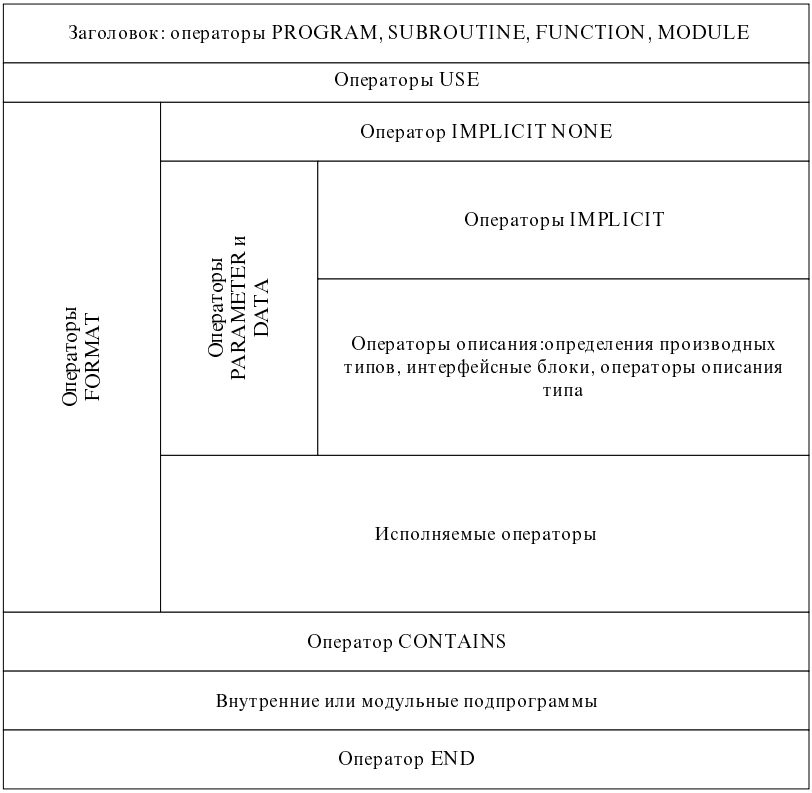


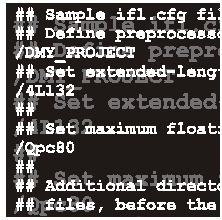

**Рис. 7.1.** Порядок операторов в программной компоненте

Операторы Фортрана разделяются на две группы: исполняемые операторы и операторы описания. К операторам описания относятся все операторы определения типа и назначения атрибутов, операторы описания производных типов, интерфейсы, а также операторы `USE`, `IMPLICIT`, `COMMON`, `EQUIVALENCE`, `FORMAT` и `DATA`. Операторы описания размещаются сразу после оператора-заголовка компоненты и всегда предшествуют исполняемым операторам (см. рис. 7.1); исключением являются: оператор `FORMAT` *(см. гл. 9)*, который не может предшествовать оператору `USE`, и оператор `DATA`, который может размещаться и среди исполняемых операторов, но по традиции заключает

группу операторов описания. Порядок следования операторов описания не произволен. Сначала размещаются операторы `USE`, затем — операторы `IMPLICIT` и `PARAMETER`. Оператор `IMPLICIT NONE` помещается впереди всех других операторов `IMPLICIT`. Операторы описания производных типов, интерфейсные блоки и другие операторы описания следуют после всех операторов `IMPLICIT`. Оператор `PARAMETER` может размещаться среди них. Затем следуют исполняемые операторы: вызовы процедур, управляющие конструкции, операторы присваивания. Оператор `CONTAINS` отделяет исполняемые операторы от описания внутренних подпрограмм. Оператор `END` завершает этот блок описания и всю программную компоненту.

## Вопросы и задания для самостоятельной работы

1. Опишите производный тип `PAIR` так, чтобы переменные этого типа вели бы себя так же, как комплексные числа.
2. Дополните производный тип `PAIR` арифметическими действиями с переменными встроенных типов.
3. Что нужно сделать для того, чтобы описанный производный тип был доступен другим программам?
4. Напишите программу, конструирующую несколько переменных типа `PAIR` и распечатывающую их.
5. Напишите подпрограмму, возвращающую значение выражения $x^2 + 3xy + y^2 + 8$, в котором $x$ и $y$ — переменные типа `PAIR`. Оформите подпрограмму как:
   - внутреннюю;
   - внешнюю.
6. Сформулируйте различия в описании внешних и внутренних подпрограмм.




# Глава 8

# Массивы

Фортран — язык, ориентированный на вычисления, а вычисления всегда связаны с массивами. Современные вычислительные средства, реализуя векторную модель выполнения операций на аппаратном уровне, позволяют рассматривать массив как единое целое. Средства Фортрана для работы с массивами соответствуют векторным представлениям вычислений; они не только удобны для программиста, но и дают дополнительные — по сравнению с циклами — возможности для оптимизации кода применительно к разным аппаратным платформам.

## Основные сведения о массивах

Массивы представляют собой упорядоченные наборы однотипных элементов. Массив характеризуется типом элементов, числом элементов и способом их нумерации. Число элементов называется размером, или длиной, массива; способ нумерации описывается одномерным целочисленным массивом, называемым формой массива.

Предполагается, что существуют массивы, не содержащие ни одного элемента — массивы нулевой длины, необходимые для описания многих естественных ситуаций, например, такой, которая возникает при использовании сечений в решении системы линейных уравнений:

$$Ax = B$$

с треугольной матрицей $A$ (листинг 8.1).

**Листинг 8.1. Пример сечения — массива нулевой длины**

```
DO I = N, 1, -1
       X(I) = B(I) / A(I, I)
       B(1:I - 1) = B(1:I - 1) - A(1:I - 1, I) * X(I)
END DO
```

На последнем шаге цикла пределы сечения равны (1:0); сечения принимают нулевой размер, и никаких действий с ними не производится. В данном случае именно это и нужно; так массив нулевой длины помогает сохранять общность действия в предельном случае.

Всякий раз, когда оказывается, что нижняя граница массива превосходит верхнюю, длина массива становится нулевой. Форма массива нулевой длины сохраняется; два массива нулевой длины разных форм несовместимы и не могут быть операндами бинарной операции. То есть выражение:

```
A(6:7, 7:6) + B(7:6, 6:7)
```

ошибочно, так как сечение `A(6:7, 7:6)` имеет форму `(2, 0)`, а сечение `B(7:6, 6:7)` — форму `(0, 2)`.

Массив нулевой длины всегда находится в состоянии определенности.

Массив в Фортране может иметь до семи измерений, то есть каждый элемент массива может иметь до семи индексов, характеризующих его положение в массиве, например:

```
A(3, 4, I, J), B(1, 2, 3 ,4, 5, 6, 7), C(K)
```

Число измерений называется рангом массива; число элементов в измерении называется экстентом массива в данном измерении. Целочисленный массив, длина которого равна рангу заданного массива, а каждый его элемент равен экстенту массива в данном измерении, называется формой массива. Например, массивы:

```
REAL, DIMENSION(10, 23) :: A
INTEGER I(-1:8, -25:-2), K(-1:1, 2:5, 1, 5, 0:3)
```

Здесь `A` и `I` имеют одинаковую форму `(10, 23)`, их длины составляют $10 \times 23 = 230$, ранг равен двум; форма массива `K`: `(3, 4, 1, 5, 4)`, ранг массива `K` равен пяти, а длина равна $3 \times 4 \times 1 \times 5 \times 4 = 240$. Границы массива в каждом измерении указываются в виде пары:

```
(нижняя_граница : верхняя_граница )
```

Если `нижняя_граница` опущена, то она принимается равной 1.

Массивы, имеющие одинаковую форму, называются *совместимыми*.

Размер массива и его форма могут быть определены встроенными функциями `SIZE` и `SHAPE` соответственно *(см. гл. 10)*. Эти функции могут использоваться для описания массивов, размеры которых задаются неконстантными выражениями.

## Массивы с подразумеваемой формой и автоматические объекты

Размеры массивов не всегда могут быть определены заранее, например, может потребоваться, чтобы размеры массива — формального параметра,

перенимались у соответствующего фактического параметра. Тогда при описании массива — формального параметра — можно указать только его форму или просто опустить верхнюю границу. Рассмотрим пример подпрограммы, меняющей местами содержимое конформных массивов-аргументов (листинг 8.2).

**Листинг 8.2. Описание массива с подразумеваемой формой и автоматического массива**

```
SUBROUTINE replace(A, B)
REAL, DIMENSION(:) :: A, B
REAL, DIMENSION(SIZE(B)) :: R
       R = A
       A = B
       B = R
END SUBROUTINE replace
```

Встроенная функция `SIZE (B)`, возвращающая целое число, равное длине массива, допускается в операторах описания как инициализирующее выражение.

Теперь, после вызова из другой программной единицы с фактическими параметрами:

```
REAL, DIMENSION(1:200) :: X, Y
.....
X = 2; Y = 4
CALL subr(X, Y)
```

границы массивов — формальных параметров `A` и `B` — примут значения границ массива — фактического параметра `X`. Массивы `A` и `B` называются массивами с подразумеваемой формой. Массивы с подразумеваемой формой перенимают экстенты фактических параметров, а не конкретные значения границ. Массивы `X` и `Y` могут иметь другие границы:

```
REAL, DIMENSION(201:400) :: X
REAL, DIMENSION(-99:100) :: Y
```

Индексы массива с подразумеваемой формой будут находиться в пределах экстентов, если только нижняя граница не задана явно в его описании, например:

```
SUBROUTINE replace(A, B)
REAL, DIMENSION(-19:) :: A, B
```

Массив `R` — локальный массив переменного размера, не являющийся формальным параметром, — представляет собой пример автоматического массива. Описания автоматических объектов зависят от неконстантных выражений, не выходящих за пределы выражений описания *(см. гл. 3)*. Такие

объекты создаются программой в момент вызова процедуры и уничтожаются по выходу из нее.

Другим примером автоматических объектов могут служить текстовые переменные изменяемой длины:

```
SUBROUTINE LETTER(WORD)
CHARACTER(LEN = *) WORD
CHARACTER(LEN = LEN(WORD)) NAME
```

На время выполнения подпрограммы, длина автоматических объектов фиксируется и не изменяется даже тогда, когда значение описывающего ее выражения становится неопределенным.

Если результат функции является объектом изменяемой длины, то интерфейс этой функции в точке вызова должен быть явным.

Автоматические объекты не могут иметь атрибута SAVE, не должны появляться в списках NAMELIST, COMMON-блоках и в операторе EQUIVALENCE.

# Элементные операции и присваивания

В *разд. "Выражения с массивами и присваивание массивов" гл. 4* уже рассказывалось об использовании встроенных операций в выражениях с массивами. Элементные операции, примененные к совместимым операндам-массивам, создают массив, все элементы которого являются результатом применения операции к соответствующим элементам массивов-операндов. Присваивание массива массиву эквивалентно присваиванию значений элементов одного массива соответствующим элементам другого массива. Присваивание массиву скаляра сводится к присваиванию скаляра каждому элементу массива.

Элементными могут быть не только встроенные операции; почти все встроенные функции, имеющие формальные аргументы-скаляры, являются элементными и пригодны для действий над массивами, если фактические аргументы-массивы имеют одинаковую форму. Если A — вещественный массив, массив B синусов его элементов можно получить одним оператором, например,

```
B = SIN(A)
```

В элементных процедурах с формальным параметром-массивом все формальные параметры с видом связи OUT или INOUT тоже должны быть массивами.

# Конструкторы массивов

Мы уже упоминали о том, как можно задать значения всех элементов массива, но нигде не рассказывали об этом подробно. В *разд. "Назначение исходных значений переменных" гл. 3* упоминались конструкторы массивов и оператор DATA; *в разд. "Элементные операции и присваивания" настоящей гл.*

и в *разд. "Выражения с массивами и присваивание массивов" гл. 4* говорилось о присваивании скаляра массиву. Если все элементы массива должны получить одно и то же значение, нужно использовать присваивание скаляра массиву, например, при его описании:

```
REAL, DIMENSION(2, 2) :: (A = 2.0)
```

Любые различные значения можно назначить элементам массива с помощью конструктора массива:

```
(/список_значений_конструктора/)
```

Каждое *значение конструктора* из списка может быть либо выражением, либо неявным циклом. Например:

```
(/0.2, 3.1/2.03, 5.5 * e, a**b/)
```

или

```
(/REAL(I), I = 1, 4/)
```

задают вещественный массив из четырех элементов; в первом случае элементы определяются выражениями, во втором — неявным циклом. Неявный цикл конструктора массива имеет вид:

```
(индексированное_значение_конструктора, индекс = индекс1,
индекс2 [, индекс3])
```

`Индекс1`, `индекс2` и необязательный `индекс3` — это скалярные выражения целого типа, которые задают нижнюю и верхнюю границы изменения индекса и необязательный шаг изменения (по умолчанию 1). `Индексированное_значение_конструктора` вычисляется `MAX((индекс2 - индекс1) / индекс3, 0)` раз от значений `индекс1`, `индекс1 + индекс3` и т. д. Область видимости переменной `индекс` ограничена самим неявным циклом; переменная с таким же именем может использоваться в другой части того же конструктора массива.

`Индексированное_значение_конструктора` само может быть неявным циклом — неявные циклы можно вкладывать друг в друга. Массив `I` целого типа, созданный конструктором:

```
(/((I - J, I = 1, 4), J = 1, 4)/),
```

эквивалентен:

```
(/0, 1, 2, 3, -1, 0, 1, 2, -2, -1, 0, 1, -3,-2, -1, 0/)
```

Конструктор всегда задает одномерный (ранга 1) массив, но с помощью встроенной функции `RESHAPE` *(см. гл. 10)* его можно преобразовать в массив любой формы.

```
RESHAPE(SOURCE = (/((I - J, I = 1, 4), J = 1, 4)/), SHAPE = (/4, 4/))
```

Приведем уже знакомый нам одномерный массив из 16-ти элементов к виду:

```
 0  1  2  3
-1  0  1  2
-2 -1  0  1
-3 -2 -1  0
```

# Подобъекты массивов

В *гл. 2 в разд. "Массивоподобные объекты"* упоминалось о регулярных сечениях и сечениях с векторным индексом — подмножествах элементов массивов, которыми можно оперировать почти так же, как и массивами. Здесь мы рассмотрим этот аспект обращения с массивами подробнее.

Сечения массивов дают удобный способ доступа к регулярным подмножествам элементов массива:

- `A(I:K)` — все элементы одномерного массива с `I`-ого по `K`-ый включительно;
- `B(2, 1:N - 1)` — элементы с первого по `N - 1` включительно второй строки двумерного массива `B`.

Для выбора подмножества элементов в экстенте используется триплет вида:

```
[индекс1]:[индекс2[:индекс3]],
```

где все *индексы* — необязательные выражения целого типа, имеющие смысл нижней границы сечения, верхней границы и шага, с которым выбираются элементы. Если опущено выражение для нижней или верхней границы, предполагается, что эта граница сечения совпадает с соответствующей границей массива в данном измерении; опущенный шаг предполагается равным единице (листинг 8.1). Нижняя граница может превосходить верхнюю, а шаг выбора может быть отрицательным — это позволяет выбирать элементы массива в обратном порядке, например,

```
B(I, 100:0:-10)
```

Фактические значения индексов, задаваемых выражением выбора, не должны выходить за границы массива. Выражение:

```
A(2:11:2)
```

для массива `A(1:10)` не является неправильным, так как фактического выхода за границы массива при выборе сечения не произойдет.

Если соотношения границ и шага в триплете таковы, что ни один элемент массива в заданном экстенте не будет выбран, экстент становится нулевым. Нулевое значение шага в триплете недопустимо.

Для обращения к подмножеству элементов массива, выбираемых нерегулярным способом, можно использовать векторный индекс — массив целого типа ранга 1:

```
INTEGER, DIMENSION(1:4) :: INDEX = (/2, 7, 3, 19 /)
REAL, DIMENSION(1:50) :: A
```

Выражение `A(INDEX)` создает сечение из четырех элементов массива `A` с индексами 2, 7, 3 и 19, следующими именно в таком порядке. Конструктор массива удобно применить непосредственно в месте вызова сечения:

```
A(/2, 7, 3, 19/)
```

С помощью векторного индекса можно создавать неоднозначные сечения, например:

```
A(/2, 7, 3, 7/)
```

включает один и тот же элемент исходного массива дважды. Неоднозначное сечение не должно появляться в левой части оператора присваивания, так как оно может создать неопределенность выбора значения для элемента массива (в нашем случае, с номером 7):

```
A(/2, 7, 3, 7/) = (/4, 5, 6, 9/) ! недопустимое выражение
```

Сечения массивов очень удобны, так как сводят к минимуму затраты на извлечение групп элементов из массивов; однако их реализация сложна для компиляторов, и потому на их использование в некоторых ситуациях наложены ограничения. Так, сечения массивов, используемые в качестве фактических параметров, рассматриваются как выражения и не могут подставляться в формальные параметры с видами связи `OUT` или `INOUT`. Сечения с векторным индексом нельзя использовать в качестве адресата ссылки — разрешение такой конструкции очень сильно бы усложнило механизм, применяемый компиляторами для реализации ссылок. Нельзя также использовать сечения массива в качестве внутреннего файла *(см. гл. 9)*.

# Обращения к элементам и подобъектам массивов

Мы уже многократно записывали элементы и подобъекты массивов в виде:

```
имя_массива(список_индексов),
```

но эта запись является только простейшим случаем обращения к элементу массива. Дело в том, что массив может быть частью структуры, а структура — элементом массива структур и так далее. Поэтому в общем случае элемент массива представляется скаляром вида:

```
частный_указатель[%частный_указатель]…,
```

где каждый `частный_указатель` представляет собой:

```
частное_имя[(список_индексов)]
```

Последний `частный_указатель` в цепочке должен иметь `список_индексов` обязательно. Для примера рассмотрим производный тип `LEVEL`, в котором содержатся два различных массива:

```
TYPE LEVEL
CHARACTER * 10 :: LABEL
INTEGER, DIMENSION(3) :: ORTS
REAL, DIMENSION(3, 3) :: PHASE
END TYPE LEVEL,
```

и массив структур типа LEVEL:

```
TYPE(LEVEL), DIMENSION(20, 30, 60) : : LEV
```

Элемент этого массива — скаляр типа LEVEL, например,

```
LEV(I - 1, J * 2, K + L) !(Все переменные: I, J, K и L — целого типа.)
```

Массивы ORTS и PHASE, принадлежащие элементу с индексом:

```
(I - 1, J * 2, K + L),
```

доступны как `LEV(I - 1, J * 2, K + L)%ORTS`

и `LEV(I - 1, J * 2, K + L)%PHASE`.

К отдельным элементам этих массивов можно обратиться, указав нужные индексы:

```
LEV(I - 1, J * 2, K + L)%ORTS(2)
```

или

```
LEV(I - 1, J * 2, K + L)%PHASE(2, 3)
```

Но сформировать обращениями массивы `LEV%ORTS(2)` и `LEV%PHASE(2, 3)` невозможно.

Список_индексов состоит из списка выражений целого типа, разделенных запятыми; число индексов должно равняться рангу массива, значения индексов не должны выходить за границы массива в данном измерении. Список_индексов может следовать только за именем массива: никакие подобъекты массива и выражения с массивами — сечения, векторные индексы, обращения к элементной функции, выражения с массивами, заключенными в скобки, — не могут специфицироваться списком индексов.

Только элементы массивов текстового типа, рассматриваемые как строки, могут сопровождаться указателем диапазона подстроки:

```
CHARACTER * 80, DIMENSION(60) :: PAGE
.....
PAGE(J) (I + 10:L -5) ! (J, I и L - целого типа.)
```

Элемент массива можно рассматривать как частный случай сечения массива, а именно как частный_указатель нулевого ранга. По аналогии с общим видом элемента массива, подобъект массива в общем виде может быть записан как цепочка:

```
частный_указатель[%частный_указатель]…..,
```

в которой частный_указатель выглядит как

```
частное_имя[(список_индексов_сечения)]
```

и в которой содержится не более одного *частного указателя* ненулевого ранга.

Ранг *частного указателя* со списком список_индексов_сечения равен числу содержащихся в нем векторных индексов и индексных триплетов. Так, например,

сечение массива F(:, I, 2:20:2) является частным указателем второго ранга, а ранг частного указателя, соответствующий элементу массива F(2, I, 3), равен нулю.

Подобъекты массивов типа CHARACTER могут дополняться указанием диапазона подстроки. Так, для символьного массива PAGE выражение PAGE(I:J) означает множество строк массива от I-ой по J-ую; если в этих строках нужно выделить подстроки с K-ой позиции по L-ую, следует записать:

```
PAGE(I:J) (K:L)
```

Для выделения подстроки во всем массиве можно выписать сечение:

```
PAGE(:) (K:L)
```

# Массивы ссылок

Прямое объявление массива ссылок в Фортране невозможно. Описание вида:

```
POINTER, DIMENSION(:) : : A
```

является описанием ссылки на массив вещественного типа, а не описанием массива ссылок. Однако это ограничение легко обойти введением производного типа с одним ссылочным компонентом:

```
TYPE PASS
REAL, POINTER :: A
END TYPE
```

с последующим объявлением массива типа PASS:

```
TYPE(PASS), DIMENSION(:) :: B
```

После этого можно выделить память под этот массив, например, так:

```
DO I = 1, N
       ALLOCATE(B(I)%A)
END DO
```

или прикрепить все его элементы к элементам массива вещественного типа с атрибутом TARGET:

```
DO I = 1, SIZE(C)
       B(I)%A => C(I)
END DO
```

# Массивы-маски

В операциях с массивами очень часто используются логические массивы-маски, возникающие, например, при применении операции сравнения к массивам. Так, для массивов:

```
REAL, DIMENSION(100, 100) :: A, B
```

выражение (A > B) порождает конформный исходным массивам массив типа LOGICAL, заполненный логическими константами .TRUE. и .FALSE. В конструкциях, которые будут рассмотрены в следующих разделах, логические массивы незаменимы. Возникающие логические массивы могут занимать много памяти, которую целесообразно уменьшить, описав заранее логические массивы той разновидности логического типа, которая занимает 1 бит (если, конечно, эта разновидность поддерживается процессором):

```
LOGICAL(KIND = 0), DIMENSION(100, 100) :: C
```

и помещая в него результат логической операции с массивами:

```
C = (A > B)
```

# Оператор и конструкция *WHERE*

Оператор WHERE можно рассматривать как аналог оператора IF для массивов. Если аргументом оператора IF является скалярное логическое выражение, то аргументом оператора WHERE является логический массив *(см. предыдущий разд. "Массивы-маски")*. Оператор WHERE применяется, когда необходимо выполнить какие-то действия только над элементами массива, удовлетворяющими некоторому условию. Например, чтобы все положительные элементы вещественного массива A заменить значениями их логарифмов, нужен оператор:

```
WHERE (A > 0.0) A = LOG(A),
```

который, вычислив массив-маску (A > 0), "наложит" ее на исходный массив. Следующий этап — вычисление логарифмов — будет выполнен только для элементов с маской .TRUE., без попыток вычислить логарифмы неположительных элементов массива A. Далее, элементам с маской .TRUE. будут присвоены полученные значения, а элементы с маской .FALSE. останутся неизмененными. Общий вид оператора:

```
WHERE (логическое_выражение_массив) переменная_массив = выражение_массив
```

Форма логического массива-маски, являющегося результатом проверки условия логическое_выражение_массив, должна совпадать с формой переменная_массив. Выражение_массив в правой части оператора присваивания вычисляется только для элементов, соответствующих истинным значениям логического выражения. Присваивание в правой части оператора WHERE может быть только встроенным.

Для выполнения группы присваиваний массивов применяется конструкция WHERE…END WHERE:

```
WHERE (логическое_выражение_массив)
       присваивание_массивов
END WHERE
```

В ходе выполнения действий присваивание_массивов условие логическое_выражение_массив может изменить свое значение, но это не скажется на выполнении присваиваний, так как условие вычисляется только один раз, при исполнении оператора WHERE. Конструкция позволяет организовать группу альтернативных назначений:

```
WHERE (логическое_выражение_массив)
       присваивание_массивов
ELSE WHERE
присваивание_массивов
END WHERE
```

Вторая группа присваиваний выполняется в том же порядке и по тем же правилам, но под управлением маски (.NOT. логическое_выражение_массив), которая также вычисляется один раз в момент выполнения оператора WHERE и не изменяется во время выполнения присваивание_массивов.

Используя оператор и конструкцию WHERE, необходимо помнить, что маскирование не распространяется на параметры неэлементных массивных функций. Например,

```
WHERE (A > 0.0) A = A / SUM(A)
```

поделит положительные элементы массива A на сумму *всех* элементов массива, а не только положительных.

# Оператор и конструкция *FORALL*

Если оператор WHERE организует работу, скажем так, с условной выборкой элементов массива, сечения массивов — с выборкой по множеству индексов, то оператор FORALL комбинирует оба способа выбора элементов. Общий вид оператора:

```
FORALL (индексное_выражение[, скалярная_логическая_маска])
оператор_назначения
```

Если, например, все положительные элементы массива в диапазоне индексов (2:n, 4:m) нужно заменить их обратными значениями, то следует записать оператор:

```
FORALL (i = 2:n, j = 4:m, A(i, j) > 0) A(i, j) = 1 / A(i, j)
```

Если выполняется несколько *операторов назначения*, то они завершаются оператором END FORALL:

```
FORALL (i = 1:n:2, j = n:1:2, A(i, j) /= 0.0 )
       A(i, j) =1.0 /A(i, j)
       A(i, i) = B(i)
END FORALL
```

На первый взгляд, конструкция FORALL кажется аналогом оператора цикла в более компактной записи. Но это не так, хотя в некоторых случаях и цикл, и FORALL дают одинаковые результаты. Схема выполнения оператора FORALL отличается от цикла принципиально. Сначала вычисляется индексное_выражение, за ним — маска для всех индексов. Для всех элементов их *индексного выражения* с маской .TRUE. вычисляются правые части выражения присваивания, и только затем выполняется присваивание. Поэтому одни и те же *операторы назначения*, имеющие самые незначительные элементы рекурсивности, будучи выполненными в цикле и операторе FORALL, приведут к различным результатам, так как в FORALL нет чередования вычисления выражений и выполнения присваиваний. Рассмотрим простой пример: над элементами с номерами 2 и 3 массива A, состоящего из четырех элементов со значениями 1, 2, 3 и 4, выполним действия вида:

```
A(i) = A(i - 1) + A(i) + A(i + 1)
```

в операторе цикла

```
DO i = 2, 3
       A(i) = A(i - 1) + A(i) + A(i + 1)
END DO
```

и в операторе FORALL:

```
FORALL(i = 2:3) A(i) = A(i - 1) + A(i) + A(i + 1)
```

По выполнении цикла получим:

```
A(2) = 1 + 2 + 3 = 6
A(3) = 6 + 3 + 4 = 13
```

FORALL же в результате даст:

```
A(2) = 1 + 2 + 3 = 6
A(3) = 2 + 3 + 4 = 9
```

# Динамические массивы

Размер массива не всегда известен заранее; для того, чтобы определить его, нужно, например, считать некоторые данные или произвести вычисления. В таком случае можно использовать динамические массивы или массивы-ссылки, память под которые выделяется в ходе исполнения программы.

Динамический массив должен описываться с атрибутом ALLOCATABLE, например:

```
REAL, ALLOCATABLE, DIMENSION(:,:) :: X
```

Экстенты динамического массива при описании не указываются, так как они к этому времени неизвестны; указывается только ранг массива.

Выделение памяти и определение фактических границ массива производится оператором ALLOCATE:

```
ALLOCATE(X (N, N))
```

где N — переменная целого типа с определенным положительным значением. Выделенная память автоматически не освобождается, ее необходимо освобождать оператором DEALLOCATE:

```
DEALLOCATE(X)
```

Того же результата можно добиться, используя ссылку на массив — память под ссылки также может быть выделена динамически:

```
REAL, POINTER, DIMENSION(:, :, :) :: Y
.....
ALLOCATE(Y(2:10, 3:K, -2:K - 1), STAT = IER)
```

# Операторы *ALLOCATE, DEALLOCATE* и *NULLIFY*

Общий вид оператора ALLOCATE:

```
ALLOCATE(список_выделяемых_объектов [,STAT = состояние]),
```

где список_выделяемых_объектов — это разделенный запятыми перечень имен массивов или ссылок с указанием границ в виде:

```
имя_объекта[([нижняя_граница:]верхняя_граница)]
```

В качестве нижней и верхней границ может использоваться скалярное выражение целого типа. По умолчанию значение нижняя_граница равно 1. В момент выделения памяти границы выделяемых объектов неопределенны и не могут использоваться в составе выражений. Нельзя, например, в рамках одного оператора использовать результат встроенной функции SIZE:

```
ALLOCATE(Y(N, N, N), X(SIZE(Y))
!неверно: границы массива Y еще не определены!
```

Но совершенно правильно использование ссылки на границы массива, выделенного предыдущим оператором:

```
ALLOCATE(Y(N, N, N))
ALLOCATE(X(SIZE(Y))
```

Необязательный ключевой параметр STAT содержит информацию о результате выполнения операции выделения памяти. Параметр STAT должен быть переменной целого типа, не входящей в состав выделяемых объектов. Если выделение прошло успешно, переменная состояние принимает значение 0; если выделить память не удалось, то эта переменная принимает положительное значение.

**Внимание**

При отсутствии параметра STAT неудача при выделении (или освобождении) области памяти приводит к останову программы.

В принципе, при наличии ссылок, выделяемые массивы и не были бы нужны; но ссылки, кроме способности занимать память динамически, обладают рядом свойств, которые нельзя не учитывать. Оператор ALLOCATE выделяет ссылке нового адресата даже в том случае, когда она прикреплена к другому адресату. Если предыдущий адресат был создан также оператором ALLOCATE и не имеет других прикрепленных к нему ссылок, он становится недоступен — так может возникнуть утечка памяти.

Оператор DEALLOCATE в общем виде:

```
DEALLOCATE(список_выделенных_объектов[,STAT = состояние])
```

подобен оператору ALLOCATE, кроме того, что в *списке выделенных объектов* не нужно указывать границ. Необязательный ключевой параметр STAT содержит информацию о результате выполнения операции освобождения памяти. Параметр STAT должен быть переменной целого типа, не входящей в состав список_выделенных_объектов. Если освобождение памяти прошло успешно, переменная состояние принимает значение 0; если освободить память не удалось, то эта переменная принимает положительное значение. При отсутствии параметра STAT неудача при освобождении памяти приводит к прекращению исполнения программы.

Оператор DEALLOCATE может освободить область памяти при наличии ссылок к адресатам, связанным с этой областью. Такие ссылки приходят в состояние неопределенности, в котором они будут находиться до прикрепления их к существующим адресатам. Открепить ссылку от адресата можно оператором NULLIFY — он применяется также и для перевода неопределенной ссылки в состояние открепленности. Оператор имеет вид:

```
NULLIFY(список_ссылочных_объектов)
```

Оператор NULLIFY не освобождает, однако, память, занятую адресатом ссылки, так как последний может использоваться и через другие ссылки. Память освобождается только оператором DEALLOCATE.

Работа с динамической памятью требует соблюдения строгого порядка в выделении и освобождении областей памяти, нарушение которого может привести к накоплению недоступной и неиспользуемой памяти.

# Вопросы и задания для самостоятельной работы

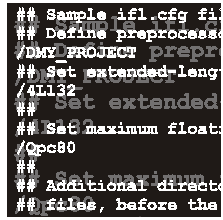
1. Для массива, описанного как REAL, DIMENSION(20, 20, 20) опишите сечения, задающие:
   - первую строку нижней плоскости;
   - последний диагональный вертикальный столбец;

- диагональ, проходящую от элемента с индексом (1, 1, 1) к элементу с индексом (20, 20, 20).

2. Напишите программу, задающую треугольную матрицу произвольного размера. (Число, определяющее размер матрицы, вводится с клавиатуры во время исполнения программы.) Представьте ее в виде массива ссылок. Заполните матрицу целыми числами так, чтобы величина всех элементов одной строки равнялась номеру строки. Распечатайте, не выводя нулевые элементы.
3. Используя оператор WHERE, выберите все простые числа из N первых чисел натурального ряда и напечатайте их.
4. Постройте массив из первых N × N простых чисел. Измените форму полученного массива и распечатайте его в виде матрицы N × N.
5. Сформируйте из первых 1000 целых чисел массив ранга 3. Используя оператор FORALL, найдите все простые числа в диагональном сечении полученного массива.
6. Найдите в следующем фрагменте автоматические объекты и массивы с подразумеваемой формой:

```
SUBROUTINE  REPLY(N, X, Y)
REAL, DIMENSION(N, N) :: F
REAL, DIMENSION(:,:) :: X
REAL, DIMENSION(:) :: Y
REAL, POINTER, DIMENSION(:,:) : : D
INTEGER, DIMENSION(SIZE(Y)) :: INDEX
```

# Глава 9




# Ввод и вывод

Эта глава посвящена операциям ввода-вывода. Вначале даются основные сведения о структуре файлов и методах доступа к ним. Затем мы познакомимся с особенностями операций ввода и вывода, рассмотрим операторы языка Фортран для передачи данных, форматирования, управления положением файлового указателя и некоторые другие. В заключение приводятся рекомендации по оптимизации операций ввода-вывода и, как обычно, вопросы и упражнения.

## Файлы

*Файл* представляет собой совокупность однотипных данных, которой присвоено имя и которая находится в памяти компьютера (оперативной или внешней). Файл может содержать обычный текст, данные, исполняемый код или что-нибудь еще. Фортран наделен разнообразными средствами работы с файлами, ведь большинство вычислительных и моделирующих программ, как правило, выполняет 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 очень большим, а их формат — разнообразным. Операции ввода-вывода данных являются относительно медленными операциями, поэтому особое значение имеет правильный выбор методов передачи данных.

Данные хранятся в файлах и передаются между файлами. Стандартным устройствам ввода-вывода, таким, как экран, клавиатура и т. д. сопоставляются специальные внешние файлы. И наоборот, каждому файлу в Фортране сопоставляется свое собственное логическое устройство. Разным логическим устройствам может соответствовать одно физическое устройство хранения информации. Чаще всего роль такого устройства играет жесткий диск компьютера, реже — магнитная лента.

Каждый файл характеризуется набором своих атрибутов. Этот набор зависит от операционной системы и может включать имя файла, права доступа,

"временные отметки" (время создания, модификации, последнего доступа к файлу) и т. д. При разработке программы следует в первую очередь выяснить правила присвоения имен файлам в той операционной системе, в которой будет выполняться программа.

Текущее положение в файле обозначается *файловым указателем*. Операция считывания записи, ее модификация, добавление или удаления из файла выполняются там, где расположен указатель. Указатель перемещается по файлу при выполнении операций ввода-вывода, а также с помощью специальных команд.

Файл состоит из *записей*, расположенных последовательно. Записью называют единицу обмена данными между программой и внешней памятью. Запись представляет собой последовательность значений или символов и является входом в файл. Это может быть строка с терминала или логическая запись в файле на диске или на магнитной ленте. Тип записи определяется соглашением о хранении данных. Все записи в файле имеют один тип. Использование типа записи, отличного от того, с которым файл создавался, может привести к непредсказуемым результатам. Именно записи определяют организацию файла, его внутреннее устройство.

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: наклонная черта вправо ("слэш" — /).

В Фортране различаются три основных типа записей.

1. *Форматные записи*, содержащие форматированные данные, которые требуют преобразования из внутреннего представления во внешнее. Форматные операторы ввода-вывода содержат явные спецификаторы формата (которые могут задать и форматирование, управляемое списком) или `NAMELIST`-спецификаторы (для ввода-вывода с использованием именованных списков). 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. Неформатная запись может вообще не содержать данных. Внутреннее представление неформатированных данных зависит от процессора. Считывать неформатированные данные могут только операторы неформатного ввода.
3. Запись *"конец файла"* — последняя запись файла. Она автоматически создается при завершении вывода данных в файл, а также может быть записана в последовательный файл оператором `ENDFILE`.

Доступ к записям может быть организован по-разному. По методу доступа в Фортране различаются два типа файлов. Во-первых, это *файлы последова-*

*тельного доступа* или просто — последовательные файлы. Файл последовательного доступа состоит из записей, расположенных в той же последовательности, в которой они были записаны в файл. Доступ к данным в файлах с последовательным доступом 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. Последовательные файлы могут храниться как на диске, так и на магнитной ленте. 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, ввод-вывод, управляемый списком, а также `NAMELIST`-ввод-вывод. Внутренние файлы (см. ниже) также являются последовательными файлами.

Файлы с последовательным доступом могут состоять как из записей с фиксированной длиной, так и из записей с переменной длиной, причем последние могут использоваться только в последовательных файлах. В начале и в конце записи с переменной длиной располагается служебная информация — поле счетчика, 4-байтовое целое число, значение которого равно количеству байтов данных в этой записи.

Второй тип файлов — *файлы прямого доступа*, которые состоят из ячеек. Ячейка представляет собой единый участок памяти, в котором хранится часть файла. Ячейки имеют фиксированную постоянную 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 магнитном диске. Файлы такого типа используются в том случае, когда необходим доступ к данным в произвольном порядке. В качестве примера можно привести базы данных. В файлах с прямым доступом используются только записи с фиксированной длиной, которая задается с помощью спецификатора `RECL` в операторе `OPEN` (см. дальше).

В различных реализациях Фортрана могут быть и другие виды доступа. Так например, в ОС (операционная система) VMS (Virtual Memory System) возможен доступ по ключу, при котором для обращения к записи файла необходимо указать ее ключ. Могут использоваться и другие типы записей. В качестве примера можно привести сегментированные и потоковые записи в Compaq Visual Fortran.

В Фортране поддерживаются также три типа файловой структуры: *форматный*, *неформатный* и *двоичный*. У каждой разновидности файлов есть свои особенности, и выбор любой из них определяется особенностями конкретной задачи. 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 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. ниже), или если при создании последовательного файла пропущен параметр `FORM`.

Форматный файл можно редактировать с помощью обычного текстового редактора, в то время как просмотр и прямое редактирование, например, двоичного 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 невозможны. Неформатные файлы создаются, если в операторе `OPEN` используется ключ `FORM = 'UNFORMATTED'` или если при создании файла прямого доступа опущен спецификатор `FORM` (см. ниже).

Данные в двоичных (бинарных) файлах хранятся в двоичном представлении, без специальной служебной информации. Длина записи фиксирована и равна одному байту. Доступ к записям возможен в произвольном порядке. Двоичные файлы создаются при использовании ключа `FORM = 'BINARY'` (см. ниже). Они компактны и подходят для хранения больших объемов информации, но поддерживаются не всеми реализациями Фортрана (пример среды, поддерживающей бинарные файлы, Compaq Visual Fortran).

Если иметь в виду представление данных и способ доступа, в Фортране поддерживаются 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, причем нулевая запись является заголовком. Данные хранятся в символьном виде. Доступ к записям возможен в произвольном порядке. Непосредственное редактирование и просмотр форматных файлов с прямым доступом затруднен. Такие файлы могут храниться только на диске. 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, разделенных маркером записи. Доступ к записям выполняется последовательно. Данные хранятся в двоичном представлении. Неформатные файлы с последовательным доступом могут располагаться только на дисках. Доступ к ним быстрее, чем к форматным файлам любого типа. Непосредственные редактирование и просмотр невозможны.
4. Неформатные файлы с прямым доступом — состоят из записей с постоянной длиной. Нулевая запись является заголовком. Доступ к записям выполняется в произвольном порядке. Данные хранятся в двоичном представлении. Неформатные файлы с прямым доступом могут располагаться только на дисках. Время доступа самое маленькое. Непосредственные редактирование и просмотр невозможны.

С точки зрения среды хранения в Фортране различают два типа файлов: внешние и внутренние. *Внешние файлы* находятся в среде, внешней по отношению к компьютерной программе. Такой средой могут быть, например, жесткий диск или магнитная лента. Записи во внешнем файле могут быть как форматными, так и неформатными.

Переменные в памяти могут вести себя как файлы на диске. Когда переменные используются таким образом, они называются *внутренними файлами*. Есть два типа внутренних файлов.

1. Внутренний файл, который является 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.
2. Внутренний файл, который является символьным массивом, символьным производным типом или массивом другого типа. Внутренний файл такого типа представляет собой последовательность элементов, каждый из которых является записью. Последовательность записей совпадает с порядком элементов массива или производного типа, а 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. Использование именованных списков `NAMELIST` не допускается. Перед выполнением оператора ввода-вывода указатель устанавливается в начало внутреннего файла, перед его первой записью. Если во внутренний файл записано меньше одной полной записи, она дополняется пробелами.

Если символьная переменная является динамическим массивом или его частью, массив перед использованием внутреннего файла должен быть размещен в памяти. Если символьная переменная является указателем, он должен быть связан с адресатом.

С внутренними файлами используются только операторы READ и WRITE. Использование операторов OPEN, CLOSE, REWIND, BACKSPACE и INQUIRE с внутренними файлами не допускается.

Внутренние файлы обычно используют, когда 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 9.1, символьные переменные STR и FNAME определяют внутренние файлы.

**Листинг 9.1. Пример использования внутренних файлов**

```
PROGRAM INTERNAL_FILE
 CHARACTER(10) :: STR, FNAME
 INTEGER :: I, X, Y, Z
 STR = "1959 20 06"
 READ(STR, *) X, Y, Z
 PRINT *, 'X = ', X
 PRINT *, 'Y = ', Y
 PRINT *, 'Z = ', Z
 I = 1
 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. ниже), или символ "звездочка" (*). Значение целого выражения должно быть в пределах от 0 до 2 147 483 640. Вообще говоря, максимальное значение номера логического устройства зависит от реализации.

Перед выполнением каких-либо операций файл необходимо сначала соединить (связать) с логическим устройством. Файл, связанный с логическим устройством, называют *открытым* файлом. Соединение выполняется опера-

тором OPEN (см. ниже). 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, но только для того, чтобы изменить некоторые 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 и не требуют отдельной спецификации. Логическое устройство с номером 5 соответствует стандартному устройству ввода (обычно это клавиатура). Логические устройства 0 и 6 связаны со стандартным устройством вывода (обычно монитор). Логическое устройство * всегда связано с устройствами стандартного ввода-вывода. Файлы, которые по умолчанию соединены с определенными устройствами, не надо присоединять специально, а отсоединятся они при завершении выполнения программы или оператором CLOSE.

Попытка закрыть устройство "звездочка" приводит к ошибке компиляции. Устройства 0, 5 и 6 можно соединить с любым файлом. Если закрыть устройства 0, 5 или 6, они автоматически соединяются заново с соответствующим логическим устройством.

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 передачи данных (см. ниже). 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.

Операции ввода-вывода могут быть *продвигающими* и *непродвигающими*. При выполнении продвигающей операции запись считывается или записывается целиком. Файловый указатель устанавливается за последней обработанной записью. Непродвигающий ввод-вывод выполняется без перемещения файлового указателя к началу следующей записи. Файловый указатель может оказаться внутри текущей записи. Этот вариант ввода-вывода может быть полезен, например, при работе со стандартными устройствами. Если непродвигающий оператор ввода пытается выполнить считывание за пределами текущей записи, вырабатывается состояние "конец записи".

# Операторы передачи данных

*Операторы передачи данных* используются для ввода и вывода данных, соединения файла с логическим устройством, определения характеристик

файла и установки файлового указателя в определенное положение. Операторы передачи данных имеют следующий вид:

```
ОПЕРАТОР(СПИСОК_УПРАВЛЕНИЯ_ВВОДОМ_ВЫВОДОМ) [СПИСОК_ВВОДА_ВЫВОДА]
ОПЕРАТОР ФОРМАТ [, СПИСОК_ВВОДА_ВЫВОДА]
ОПЕРАТОР
```

`Список управления вводом-выводом` состоит из спецификаторов. *Управляющие спецификаторы* могут появиться в списке управления вводом-выводом только один раз и только в одной из следующих форм:

- ключевая форма (например, `UNIT = 15`) — значению предшествует ключевое слово, обозначающее спецификатор, и знак равенства. В этом случае спецификаторы в списке могут находиться в любом порядке;
- неключевая форма — значение спецификатора без указания ключевого слова. В этом случае значение должно находиться на определенном месте в списке управления;
- смешанная форма — в списке присутствуют спецификаторы как в ключевой, так и в неключевой форме. Неключевые параметры в этом случае должны находиться в начале списка, затем идут все ключевые.

Основными операторами ввода-вывода являются `READ` и `WRITE`. Оператор `READ` передает данные из внешнего или внутреннего файла в программу. Оператор `WRITE` передает данные во внешний или внутренний файл. Форма вызова этих операторов зависит от вида форматирования и типа файла, из которого производится считывание данных. Различается также набор допустимых спецификаторов. Рассмотрим возможные варианты вызова этих операторов. Поскольку в приведенных ниже примерах часто используется символ "&", напомним, что он не является частью оператора ввода или вывода, а обозначает продолжение строки программы.

Форматный ввод-вывод с последовательным доступом:

```
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]
```

Форматный ввод-вывод с использованием конструкции NAMELIST:

```
READ([UNIT =]НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА, NML [, IOSTAT] &
& [, ERR] [, END])
READ NM
WRITE([UNIT =]НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА, NML [, IOSTAT] [, ERR])
```

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:

```
READ([UNIT =]НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА, FMT [, IOSTAT] &
& [, ERR] [, END]) [СПИСОК_ВВОДА_ВЫВОДА]
WRITE([UNIT =]НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА, FMT [, IOSTAT] &
& [, ERR]) [СПИСОК_ВВОДА_ВЫВОДА]
```

Значением спецификатора UNIT является номер логического устройства. Это единственный обязательный спецификатор в списке управления вводом-выводом. В случае использования неключевого варианта, номер логического устройства должен быть первым в списке. Спецификатор устройства не требуется в следующих операторах:

- оператор INQUIRE для файла;
- операторы ACCEPT, PRINT и TYPE;
- оператор READ, который содержит только список ввода-вывода и спецификатор формата и который выполняет ввод из стандартного ввода;
- оператор WRITE, который содержит только список ввода-вывода и спецификатор формата и который выполняет вывод на стандартное устройство.

Спецификатор FMT задает формат преобразования данных. Правила описания формата рассматриваются в следующем разделе данной главы. В случае использования неключевого варианта, формат должен находиться на втором месте в списке. Значениями этого спецификатора могут быть:

- метка оператора FORMAT. Оператор FORMAT должен находиться в том же модуле программы, что и оператор передачи данных;

- символ "звездочка" (*), задающий форматирование, управляемое списком. Для файлов с прямым доступом значение "звездочка" не допускается;
- скалярная целая переменная, которой присваивается значение метки оператора FORMAT (с помощью оператора ASSIGN);
- символьное выражение (массив или символьная константа), содержащее динамический формат. Его значением должна быть синтаксически правильная спецификация формата. Если выражение является массивом, оно трактуется так, как если бы все элементы массива были объединены согласно естественному порядку хранения элементов массива;
- имя числового массива или элемента массива, содержащего формат.

Если в списке управления имеется спецификатор формата, использование спецификатора NAMELIST-группы не допускается.

Если значение спецификатора продвигающего ввода-вывода ADVANCE равно 'YES', оператор использует продвигающий ввод-вывод. Значение 'NO' соответствует непродвигающему вводу-выводу. По умолчанию используется значение 'YES'. Спецификатор ADVANCE может находиться только в операторе форматного последовательного ввода-вывода. Его нельзя использовать для передачи данных, управляемой списком или NAMELIST-ввода-вывода.

Спецификатор "счетчик символов" SIZE используется только для форматных последовательных операторов непродвигающего ввода READ. Он определяет скалярную переменную целого типа, значение которой равно количеству символов, считанных к моменту прекращения выполнения оператора READ. Если при открытии файла указано PAD = 'YES', пробелы, дополняющие запись, не учитываются. Спецификатор SIZE нельзя задавать для управляемой списком или NAMELIST-передачи данных.

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. 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, а также получить информацию об операции ввода-вывода. Он может использоваться совместно со спецификаторами ветвления END, EOR и ERR. При выполнении оператора ввода-вывода, содержащего спецификатор IOSTAT, подавляется вывод сообщений операционной системы об ошибках.

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. Оператор-адресат ветвления должен находиться в том же модуле программы, что и оператор передачи данных. 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", файловый указатель устанавливается после текущей записи и выполнение оператора прекращается. Если указан спецификатор SIZE, переменной SIZE присваивается целое значение.

Если возникает одно из вышеперечисленных условий, 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, выполнение программы прекращается. Спецификаторы ветвления используются для того, чтобы изменить "стандартную" реакцию программы на события "конец файла", "конец записи" или ошибку ввода-вывода.

Неключевая форма спецификатора формата в описаниях интерфейса операторов ввода-вывода обозначена словом FORM, а * — спецификатор формата, задающий форматирование, управляемое списком. В операторах WRITE прямого доступа звездочка не используется.

Спецификатором NAMELIST-группы для NAMELIST-ввода-вывода является NML. При использовании неключевой формы имя NAMELIST-группы должно находиться на втором месте в списке управления вводом-выводом после спецификатора логического устройства. Этот спецификатор не совместим со спецификатором формата. Его неключевую форму мы обозначили NM.

Номер ячейки записи для прямого доступа задается спецификатором REC. Список спецификаторов, используемых в операторах передачи данных, приведен в табл. 9.1.

***Таблица 9.1.*** *Список спецификаторов, используемых в операторах передачи данных*

| Спецификатор | Значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| ACCESS | 'SEQUENTIAL'<br>'DIRECT'<br>'APPEND' | Задает метод доступа к файлу | INQUIRE<br>OPEN |

***Таблица 9.1*** *(продолжение)*

| Спецификатор | Значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| ACTION | 'READ'<br>'WRITE'<br>'READWRITE'<br>(по умолчанию) | Задает режим ввода-вывода | INQUIRE<br>OPEN |
| ADVANCE | 'NO'<br>'YES'<br>(по умолчанию) | Определяет, является форматный последовательный ввод продвигающим или непродвигающим | READ |
| BINARY | 'NO'<br>'YES' | Возвращает информацию о том, является ли файл двоичным | INQUIRE |
| BLANK | 'NULL'<br>'ZERO'<br>(по умолчанию) | Указывает, следует ли игнорировать пробелы в числовых полях или их следует трактовать как нули | INQUIRE<br>OPEN |
| DELIM | 'APOSTROPHE'<br>'QUOTE'<br>'NONE'<br>(по умолчанию) | Задает символы-ограничители для данных при вводе-выводе, управляемом списком или NAMELIST-данных | INQUIRE<br>OPEN |
| DIRECT | 'NO'<br>'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 |
| ERR | Целое значение от 1 до 99999 | Задает метку исполняемого оператора, которому передается управление в случае ошибки ввода-вывода | Все, кроме PRINT |
| EXIST | .TRUE.<br>.FALSE. | Возвращает информацию о том, существует ли данный файл и может ли он быть открыт | INQUIRE |

***Таблица 9.1*** *(продолжение)*

| Спецификатор | Значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| FILE | Символьная переменная или выражение. Длина и формат имени определяются операционной системой | Задает имя файла | INQUIRE<br>OPEN |
| FMT | Символьная переменная или выражение | Задает список дескрипторов форматирования | PRINT<br>READ<br>WRITE |
| FORM | 'FORMATTED'<br>'UNFORMATTED'<br>'BINARY' | Задает формат файла | INQUIRE<br>OPEN |
| FORMATTED | 'NO'<br>'YES' | Возвращает информацию о том, присоединен ли файл для форматной передачи данных | INQUIRE |
| IOSTAT | Целая переменная | Задает переменную, значение которой показывает, была ли ошибка ввода-вывода | Все, кроме PRINT |
| NAMED | .TRUE.<br>.FALSE. | Возвращает информацию о том, является ли файл именованным | INQUIRE |
| NEXTREC | Целая переменная | Возвращает информацию о том, может ли быть обработана (считана или выведена) следующая запись файла | INQUIRE |
| NML | NAMELIST-имя | Задает NAMELIST-группу для ввода или вывода | PRINT<br>READ<br>WRITE |
| NUMBER | Целая переменная | Возвращает номер логического устройства, соединенного с файлом | INQUIRE |
| OPENED | .TRUE.<br>.FALSE. | Возвращает информацию о том, присоединен ли файл | INQUIRE |

***Таблица 9.1*** *(продолжение)*

| Спецификатор | Значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| PAD | 'YES' (по умолчанию)<br>'NO' | Указывает, будет ли вводимая запись дополняться пробелами, когда список ввода или формат требует больше данных, чем содержится в записи | INQUIRE<br>OPEN |
| POSITION | 'ASIS' (по умолчанию)<br>'REWIND'<br>'APPEND' | Задает положение файлового указателя | INQUIRE<br>OPEN |
| READ | 'NO'<br>'YES' | Возвращает информацию о том, можно ли считывать файл | INQUIRE |
| READWRITE | 'NO'<br>'YES' | Возвращает информацию о том, можно ли считывать файл и выполнять в него запись | INQUIRE |
| REC | Положительная целая переменная или выражение | Задает первую (или единственную) запись файла, которую необходимо считать или записать | READ<br>WRITE |
| RECL | Положительная целая переменная или выражение | Задает длину записи для файлов прямого доступа или максимальную длину записи в файлах последовательного доступа | INQUIRE<br>OPEN |
| SEQUENTIAL | 'NO'<br>'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'<br>'NEW'<br>'UNKNOWN' (по умолчанию)<br>'SCRATCH' | Задает статус файла после открытия и/или закрытия | CLOSE<br>OPEN |

***Таблица 9.1*** *(окончание)*

| Спецификатор | Значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| UNFORMATTED | 'NO'<br>'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'<br>'YES' | Возвращает информацию о том, открыт ли файл для записи | INQUIRE |

В различных реализациях языка могут быть дополнительные спецификаторы (дополнительные спецификаторы Compaq Visual Fortran приведены в табл. 9.2).

***Таблица 9.2.*** *Список спецификаторов Compaq Visual Fortran*

| Спецификатор | Допустимые значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| ASSOCIATEVARIABLE | Целая переменная | Задает переменную, которой будет присвоено значение номера следующей записи последовательного файла | OPEN |
| BLOCKSIZE | Положительная целая переменная или выражение | Задает или возвращает размер внутреннего буфера, используемого в операциях ввода-вывода | INQUIRE<br>OPEN |
| BUFFERCOUNT | Численное выражение | Задает количество буферов, связанных с логическим устройством при мультибуферном вводе-выводе | OPEN |
| BUFFERED | 'YES'<br>'NO'<br>(по умолчанию) | Задает поведение исполняющей системы при выполнении операций вывода | INQUIRE<br>OPEN |

***Таблица 9.2*** *(продолжение)*

| Спецификатор | Допустимые значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| CARRIAGECONTROL | 'FORTRAN'<br>'LIST'<br>'NONE' | Задает обработку символа возврата каретки | INQUIRE<br>OPEN |
| CONVERT | 'LITTLE_ENDIAN'<br>'BIG_ENDIAN'<br>'CRAY'<br>'FDX'<br>'FGX'<br>'IBM'<br>'VAXD'<br>'VAXG'<br>'NATIVE'<br>(по умолчанию) | Задает численный формат для неформатных данных | INQUIRE<br>OPEN |
| DEFAULTFILE | Символьное выражение | Задает маршрутное имя файла, используемое по умолчанию | INQUIRE<br>OPEN |
| DISPOSE (DISP) | 'KEEP'<br>'SAVE'<br>'DELETE'<br>'PRINT'<br>'PRINT/DELETE'<br>'SUBMIT'<br>'SUBMIT/DELETE'<br>(по умолчанию 'DELETE' для временных файлов и 'KEEP' для прочих) | Задает статус файла после того, как устройство закрыто | OPEN<br>CLOSE |
| NAME | Символьная переменная или выражение. Длина и формат имени определяются операционной системой | Задает имя файла | INQUIRE<br>OPEN |

***Таблица 9.2*** *(продолжение)*

| Спецификатор | Допустимые значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| MAXREC | Численное выражение | Задает максимальное количество записей, которые могут быть переданы в файл прямого доступа | OPEN |
| MODE | 'READ'<br>'WRITE'<br>'READWRITE' (по умолчанию) | То же, что ACTION | INQUIRE<br>OPEN |
| ORGANIZATION | 'SEQUENTIAL' (по умолчанию)<br>'RELATIVE' | Задает внутреннюю организацию файла | INQUIRE<br>OPEN |
| READONLY | | Указывает, что для данного соединения можно использовать только оператор READ | OPEN |
| RECORDSIZE | Положительная целая переменная или выражение | Задает длину записи для файлов прямого доступа или максимальную длину записи в файлах последовательного доступа | INQUIRE<br>OPEN |
| RECORDTYPE | 'FIXED'<br>'VARIABLE'<br>'SEGMENTED'<br>'STREAM'<br>'STREAM_LF'<br>'STREAM_CR' | Задает тип записей в файле | INQUIRE<br>OPEN |
| SHARE | 'COMPAT'<br>'DENYNONE' (по умолчанию)<br>'DENYWR'<br>'DENYRD'<br>'DENYRW' | Управляет одновременным доступом к файлу других процессов в сетевых системах | INQUIRE<br>OPEN |

***Таблица 9.2*** *(окончание)*

| Спецификатор | Допустимые значения | Описание | Операторы, в которых используется данный спецификатор |
|---|---|---|---|
| SHARED | | Указывает, что файл присоединен для совместного доступа из нескольких программ одновременно | OPEN |
| USEROPEN | Имя пользовательской функции | Задает внешнюю функцию, которая управляет открытием файла | OPEN |

Если параметр оператора READ или WRITE является выражением, которое содержит вызов функции, эта функция не должна выполнять операции ввода-вывода, поскольку в этом случае результат непредсказуем.

Список ввода-вывода задает элементы данных, значения которых будут передаваться оператором передачи данных. Это может быть простой список, циклический список или комбинация простого и циклического списков.

Простой список ввода-вывода содержит перечисление элементов, имена которых разделяются запятыми. Элементами списка могут быть переменные и массивы. Переменная не может быть массивом с подразумеваемым размером, за исключением случая, когда для последнего измерения используется индекс, векторный индекс или индекс сечения, задающий его верхнюю границу. В операторах вывода в списке могут находиться также выражения и константы. Оператор передачи данных присваивает значения элементам списка или выводит их значения в том порядке, в котором они располагаются в списке, слева направо.

В следующем примере внешний файл RESULT.DAT соединяется с логическим устройством 12, а затем в этот файл записывается числовое значение:

```
OPEN(UNIT = 12, FILE = 'RESULT.DAT')
WRITE(12, '(F9.3)') VELOCITY
```

Пример вывода на экран символьной константы:

```
WRITE(*, '(1X, A5)') 'Hello'
```

В программе PRECONNECTED_UNIT (листинг 9.2) вывод выполняется на логическое устройство 6, связанное по умолчанию с экраном. Затем это устройство соединяется с внешним файлом, и вывод выполняется в этот файл. Затем логическое устройство 6 вновь соединяется со стандартным устройством вывода (экраном).

**Листинг 9.2. Пример вывода на стандартное устройство 6**

```
PROGRAM PRECONNECTED_UNIT
 REAL :: X, Y
 SCREEN_DO : DO X = 0.1, 6.28, 0.1
       Y = SIN(X)
       WRITE(6, '(F3.1, F5.2)') X, Y
 END DO SCREEN_DO
 OPEN(UNIT = 6, FILE = 'SIN_TAB', STATUS = 'NEW')
 FILE_DO : DO X = 0.1, 6.28, 0.1
       Y = SIN(X)
       WRITE(6, '(F3.1, F5.2)') X, Y
 END DO FILE_DO
 CLOSE(6)
 WRITE(6,' (''  SIN TABLE COMPLETED'')')
END PROGRAM PRECONNECTED_UNIT
```

Если в списке ввода-вывода неформатного оператора содержится несколько имен массивов, считывается или выводится только одна запись, независимо от того, сколько ссылок на имя массива находится в списке. Если в списке ввода используется имя массива, 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, для него должна быть выделена память.

При наличии в списке массива передача данных начинается с его первого элемента и выполняется в порядке возрастания индекса, причем крайний левый индекс изменяется наиболее быстро. Пусть имеется двумерный массив:

```
REAL, DIMENSION(2, 2) :: TAB
```

Если его имя будет использоваться без индексов в операторе `READ`, 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))
```

При выполнении форматной передачи данных список ввода-вывода не может содержать переменные производных типов, но может включать их компоненты. Это ограничение не распространяется на неформатный и двоичный ввод-вывод. Пример ввода значений переменной производного типа приведен в листинге 9.3.

**Листинг 9.3. Пример ввода значений переменной производного типа**

```
PROGRAM TYPE_INDEMO
 TYPE EXPERIMENT
       INTEGER NO
       REAL WEIGHT
 END TYPE EXPERIMENT
 TYPE(EXPERIMENT) :: DAT1
       READ *, DAT1
       PRINT *, DAT1
       READ *, DAT1%NO, DAT1%WEIGHT
 PRINT *, DAT1
END PROGRAM TYPE_INDEMO
```

Операторы ввода в этом примере эквивалентны. Еще один пример форматного вывода для переменной производного типа приведен в листинге 9.4.

**Листинг 9.4. Пример форматного вывода для переменной производного типа**

```
PROGRAM TYPE_INDEMO
 TYPE EXPERIMENT
       INTEGER NO
       REAL ENERGY
       COMPLEX OMEGA
 END TYPE EXPERIMENT
 TYPE(EXPERIMENT) :: DAT2
 DAT2%NO = 1
```

```
 DAT2%ENERGY = 6500.23
 DAT2%OMEGA = (1120.11, 0.002)
 WRITE(*, 100) DAT2
  100 FORMAT(1X, I5, ',', 3(F9.3, 2X))
END PROGRAM TYPE_INDEMO
```

Вывод этой программы выглядит следующим образом:

```
    1, 6500.230   1120.110      0.002
```

Выполняя передачу данных для производных типов, следует иметь в виду следующее. Любой компонент производного типа должен находиться в модуле программы, содержащем оператор ввода-вывода. Производный тип не должен содержать компонент-указатель. В операторе форматного ввода-вывода производный тип трактуется так, как если бы все компоненты структуры были заданы в том же порядке, что и в определении производного типа. В неформатном операторе ввода-вывода производный тип трактуется как один объект.

Если вводимым элементом данных является указатель, он должен быть в этот момент связан с определенным адресатом и данные передаются из файла соответствующему адресату. Если указатель содержится в списке вывода, он также должен быть связан с адресатом.

Список ввода-вывода может содержать внутренние циклические списки, которые имеют следующий вид:

```
(СПИСОК_ПЕРЕМЕННЫХ_ВЫРАЖЕНИЙ_КОНСТАНТ, DOVAR = EX1, EX2 [,EX3]),
```

где `DOVAR` — имя скалярной целой или вещественной управляющей переменной цикла. Это имя не должно встречаться в списке имен вводимых элементов данных в списке. Выражение `EX1` задает начальное значение управляющей переменной цикла, `EX2` — конечное значение, а `EX3` — это шаг изменения переменной. Если выражение `EX3` пропущено, шаг принимается равным 1. Выражения могут быть скалярными целыми или вещественными. Тип выражений не обязательно совпадает с типом управляющей переменной цикла. 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, 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 9.5).

**Листинг 9.5. Использование циклических списков для вывода массива**

```
PROGRAM INPUT_ARRAY
 INTEGER, DIMENSION(5, 5) :: A
 READ (5, 100) IB, JB, ((A(I, J), J = 1, JB), I = 1, IB)
 100 FORMAT (2I4, /, (5I4))
 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 9.6.

**Листинг 9.6. Использование циклических списков**

```
PROGRAM OUTPUT_TABLE
 OPEN(1, FILE ='TEST.DAT')
 WRITE(1, 10) (N, SQRT(FLOAT(N)), N = 1, 100)
 10 FORMAT (I5, F8.4)
END PROGRAM OUTPUT_TABLE
```

Оператор ввода данных из внутреннего файла может быть только форматным. Он должен включать спецификаторы формата, допускается также форматирование, управляемое списком, а NAMELIST-форматирование не допускается.

Форматные операторы ввода READ из внутреннего файла преобразуют данные из символьного в двоичное представление, используя для преобразования спецификаторы формата (если они есть). Преобразованные значения присваиваются элементам списка ввода-вывода в том порядке, в котором они указаны, слева направо. Этот вариант оператора READ действует так, как если бы формат начинался с дескриптора BN (см. ниже).

Перед передачей данных указатель устанавливается в начало первой записи, и она становится текущей. Если количество элементов списка ввода меньше,

чем число полей во входной записи, лишние поля игнорируются. Если число элементов списка ввода-вывода больше, чем число полей во входной записи, она дополняется пробелами. Если при открытии файла был указан спецификатор PAD = 'NO', количество элементов ввода не должно превышать количество вводимых значений. При форматировании, управляемом списком, строки символов не имеют ограничителей.

В программе CONVERSION (листинг 9.7) считывается запись. Затем проверяется ее первый символ. В зависимости от символа следующее значение интерпретируется как десятичное (I), восьмеричное (O) или шестнадцатеричное (Z). Для того чтобы выполнить преобразование из символьного представления в двоичное, используется оператор READ для ввода из внутреннего файла.

**Листинг 9.7. Преобразование между системами счисления**

```
PROGRAM CONVERSION
 INTEGER VALUE
 CHARACTER :: TYPE, STRING * 80
 CHARACTER*(*), PARAMETER :: ACHARFMT = '(Q, A)', &
 INTFMT = '(I10)', OCTALFMT = '(O11)', HEXFMT = '(Z8)'
 WRITE(*, *) 'ВВЕДИТЕ ЦЕЛОЕ, ДЕСЯТИЧНОЕ ИЛИ ШЕСТНАДЦАТЕРИЧНОЕ ', &
& ' ЗНАЧЕНИЕ С ПРЕФИКСОМ I, O, Z'
 READ(*, ACHARFMT) ILEN, STRING
 TYPE = STRING(1:1)
 IF(TYPE .EQ. 'I') THEN
       READ(STRING(2:MIN(ILEN, 11)), INTFMT) VALUE
 ELSE IF(TYPE .EQ. 'O') THEN
             READ(STRING(2:MIN(ILEN, 12)), OCTALFMT) VALUE
      ELSE IF(TYPE .EQ. 'Z') THEN
                   READ(STRING(2:MIN(ILEN, 9)), HEXFMT) VALUE
           ELSE
                   PRINT *, 'ERROR IN PREFIX'
 END IF
 PRINT *, 'DECIMAL VALUE = ', VALUE
END PROGRAM CONVERSION
```

Операторы WRITE могут передавать данные во внутренний файл. Такой вывод может быть только форматным, возможно также форматирование, управляемое списком, а NAMELIST-форматирование не допускается. Внутренние операторы WRITE преобразуют данные из двоичного представления в символьный вид, используя спецификации формата, если они имеются. При работе с внутренними файлами объекты ввода-вывода могут быть как

встроенного, так и производного типа. Для производных типов значения передаются из компонентов встроенных типов.

Если количество символов в записи меньше ее длины, остаток записи дополняется пробелами. Количество выводимых символов не должно превышать длину записи. Символьные константы не ограничены апострофами или кавычками, и каждый внутренний апостроф или кавычка во внешнем представлении соответствует одному апострофу или кавычке.

Операторы неформатного ввода-вывода для файлов с прямым доступом передают двоичные данные без преобразования. Это могут быть объекты как встроенных, так и производных типов. 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. Одно комплексное значение соответствует двум вещественным элементам. Соответствующие значения и элементы должны иметь один параметр разновидности.

Если количество элементов списка ввода-вывода меньше числа полей во входной записи, оператор игнорирует лишние поля. Если количество элементов списка ввода-вывода больше числа полей во входной записи, возникает ошибка. Если значения в списке вывода не заполняют запись, она дополняется пробелами. Если файл открыт для форматного управляемого списком или `NAMELIST`-ввода, 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, при этом могут передаваться объекты встроенных и производных типов. Если список вывода пуст, выводится одна пустая запись. Каждое значение в записи должно быть того же типа, что и соответствующий элемент списка ввода-вывода. Одно комплексное значение соответствует двум вещественным элементам списка, а два вещественных значения могут отвечать одному комплексному элементу списка. Соответствующие друг другу значения и элементы списка ввода должны иметь одну разновидность. Если файл открыт для форматного вывода, форматирования управляемого списком или `NAMELIST`-вывода, передача неформатных данных не допускается.

Если количество элементов списка ввода-вывода меньше числа полей в записи ввода, избыточные поля игнорируются. Если число элементов списка ввода-вывода превышает число полей во вводимой записи, возникает ошибка. Если список ввода отсутствует, оператор пропускает одну полную запись, устанавливает указатель для считывания следующей записи при следующем выполнении оператора `READ`. Последовательные операторы `WRITE`

с NAMELIST-выводом переводят данные из внутреннего представления во внешнее, используя типы данных объектов в соответствующем операторе NAMELIST.

По умолчанию, символьные константы не ограничиваются апострофами или кавычками, и каждый внутренний апостроф или кавычка во внешнем представлении соответствует одному апострофу или кавычке. Это правило можно изменить с помощью спецификатора DELIM в операторе OPEN. Если файл открыт со спецификатором DELIM = 'QUOTE', символьные константы ограничиваются кавычками и каждая внутренняя кавычка во внешнем представлении — две последовательных кавычки. Если файл открыт со спецификатором DELIM = 'APOSTROPHE', ограничителем является апостроф, а каждый внутренний апостроф во внешнем представлении — это два последовательных апострофа.

Буквальная символьная или комплексная константа может оказаться длиннее записи. Если мнимая часть и закрывающие скобки не "укладываются" в текущую запись, конец записи может находиться между запятой и мнимой частью.

Непродвигающий ввод-вывод допускает выполнение операторов READ и WRITE без перемещения указателя на новую запись. Он задается спецификатором ADVANCE ='NO':

```
WRITE(*, 100, ADVANCE = 'NO') X
```

Непродвигающий ввод-вывод не может управляться списком. По умолчанию в READ и WRITE используется продвигающий ввод-вывод.

Оператор PRINT аналогичен форматному, последовательному оператору WRITE и предназначен для вывода на экран. Он не может передавать данные на логическое устройство, заданное пользователем. Оператор TYPE является синонимом оператора PRINT. Оператор PRINT может использоваться следующим образом:

```
PRINT FORM [, СПИСОК_ВВОДА_ВЫВОДА]
PRINT * [, СПИСОК_ВВОДА_ВЫВОДА]
PRINT NM
```

Здесь FORM — неключевая форма спецификатора формата, символ "звездочка" обозначает форматирование, управляемое списком, а NM — неключевая форма спецификатора, задающего NAMELIST-форматирование. Следующие операторы эквивалентны:

```
PRINT '(A18)', 'Hello, Programmer!'
WRITE(*, '(A18)') ' Hello, Programmer!'
TYPE '(A18)', ' Hello, Programmer!'
```

# Форматирование ввода-вывода

При вводе и выводе выполняется преобразование данных между внешним и внутренним представлениями. Преобразованию может сопутствовать форматирование, необходимое для того, чтобы должным образом разместить выводимые значения. Фортран предоставляет обширные возможности для форматирования и преобразования (конвертации) данных. Описание формата, используемое в операторах ввода-вывода, определяет форму передачи данных и преобразование данных, необходимое для соблюдения этой формы. Формат может быть явным или неявным.

Явный формат задается оператором FORMAT или символьным выражением. Оно 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 правильная спецификация формата. Неявный формат определяется посредством управляемого списком или NAMELIST-форматирования. Форматирование, управляемое списком, задается с помощью звездочки, а NAMELIST-форматирование задается групповым NAMELIST-именем.

Оператор FORMAT имеет следующий вид:

```
МЕТКА FORMAT(СПИСОК_ДЕСКРИПТОРОВ)
```

Оператор FORMAT должен быть помечен. 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. Эта переменная размещается непосредственно в операторе ввода-вывода.

Дескрипторы преобразования данных (I, B, O, Z, F, E, EN, ES, D, G, L и A) управляют преобразованием данных между внутренним и внешним представлениями. Есть также управляющие дескрипторы (T, TL, TR, X, S, SP, SS, BN, BZ, P, :) и дескрипторы преобразования строк (H, 'c'). При перечислении, запятая не ставится после дескриптора P, перед дескриптором "слэш", когда отсутствует спецификатор повторения, после дескриптора "слэш", а также с дескриптором "двоеточие" (:). Дескрипторы преобразования могут находиться в круглых скобках. Перед открывающей скобкой может быть задан спецификатор повторения. В операторе ввода-вывода, содержащем список ввода-вывода, спецификация формата должна содержать, по крайней мере, один дескриптор преобразования данных. Спецификация формата не может задавать больше выводимых символов, чем может содержать внешняя запись.

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`, `EN`, `ES`, `D` или `G`. Управляющие дескрипторы и дескрипторы для символьной строки не соответствуют элементам списка ввода-вывода.

Перечень дескрипторов формата приведен в табл. 9.3.

***Таблица 9.3.*** *Дескрипторы формата*

| Спецификатор | Описание |
|---|---|
| `A[w]` | Дескриптор символьных значений. Пример: `A5` можно использовать для вывода символьной строки `'Роман'` |
| `Bw[.m]` | Дескриптор двоичных значений (Фортран 90) |
| `BN` | Игнорировать пробелы внутри строки или в правых позициях поля числового ввода |
| `BZ` | Интерпретировать пробелы внутри строки и в правых позициях поля числового ввода как нули |
| `Dw.d` | Дескриптор вещественных значений с двойной точностью. Пример: `D18.10` |
| `Ew.d[Ee]` | Дескриптор вещественных значений в экспоненциальной форме. Пример: значение в формате `E12.5` — занимает 12 позиций. Из них 5 позиций используются для цифр, 3 для порядка, 1 для знака, 1 для нуля в старшей позиции, 1 для десятичной точки и 1 для пробела |
| `ENw.d[Ee]` | Дескриптор вещественных значений в инженерной нотации |
| `ESw.d[Ee]` | Дескриптор вещественных значений в математической (научной) нотации |
| `Fw.d` | Дескриптор вещественных значений с фиксированной запятой. Пример: значение в формате `F9.3` занимает 9 позиций, из которых 3 используются для дробной части |
| `Gw.d[Ee]` | Универсальный дескриптор значений всех встроенных типов |
| `nHстрока` | Дескриптор символьных значений. Пример: `5Hvalue` является холлеритовской константой |
| `Iw[.m]` | Дескриптор целых значений. Пример: `I6` резервирует 6 позиций |
| `Lw` | Дескриптор логических значений |
| `Ow[.m]` | Дескриптор восьмеричных значений (Фортран 90) |
| `KP` | Интерпретировать вещественные значения с заданным масштабным множителем |

***Таблица 9.3*** *(окончание)*

| Спецификатор | Описание |
|---|---|
| S | Прекратить действие дескрипторов SP и SS и возобновить действие правила вывода необязательного плюса по умолчанию |
| SP | Записывать необязательный плюс (+) в поле числового вывода |
| SS | Подавлять вывод необязательного плюса (+) в поле числового вывода |
| Tn | Табуляция до указанной позиции |
| TLn | Табуляция влево на указанное количество позиций |
| TRn | Табуляция вправо на указанное количество позиций |
| NX | Пропуск n позиций вправо от текущей позиции |
| Zw[.m] | Дескриптор шестнадцатеричных значений (Фортран 90) |
| : | Прекратить действие управления форматом, если в списке ввода-вывода больше нет элементов |
| 'c'<br>"c" | Передать символьную буквальную константу, заключенную между ограничителями (апостроф или кавычки) в выходную запись |

Дескрипторы S, SP, SS, BN, BZ и kP действуют до конца списка форматирования оператора FORMAT или до тех пор, пока не встретится следующий аналогичный дескриптор.

Масштабирование численных значений с помощью дескриптора kP рекомендуется применять при выводе с дескрипторами E и D. В этом случае оно позволяет выбрать удобную форму представления вещественного значения. С помощью масштабного множителя можно избежать появления нуля в качестве первой цифры значения и получить, таким образом, при том же числе позиций больше цифр значащей части числа. Масштабирование с дескриптором F применяется редко.

Дескриптор kP задает значение масштабного множителя k, который действует с дескрипторами числовых значений F, E, EN, ES, D и G следующим образом:

- при вводе (если в поле ввода нет порядка) масштабный множитель переводит число во внешнее представление числа, равное внутреннему представлению, умноженному на $10^k$. Если в поле есть порядок, масштабный множитель не применяется;
- при выводе с дескрипторами E и D мантисса умножается на $10^k$, а порядок уменьшается на k;
- при выводе с дескриптором G действие масштабного множителя подавляется, если только значение, для которого необходимо выполнить преоб-

разование, не находится за пределами диапазона, допускаемого форматом F. В противном случае масштабный множитель действует так же, как с дескриптором E;

- при выводе с дескрипторами EN и ES масштабный множитель не действует;
- при выводе со спецификацией F, действие масштабного множителя заключается в том, что внешнее представление числа равно внутреннему его представлению, умноженному на $10^k$.

Дескрипторы преобразования численных значений I, B, O, Z, F, E, EN, ES, D и G можно использовать для передачи данных любого числового типа. При вводе численных значений старшие пробелы игнорируются, а при выводе значение выравнивается вправо. Если при выводе количество символов превышает ширину поля вывода, все поле заполняется звездочками.

Дескрипторы Iw, Iw.m, Bw, Bw.m, Ow, Ow.m, Zw и Zw.m описывают передачу целых значений. Целая константа w задает ширину поля при вводе, включая знак (если он есть). Целая константа m задает минимальное количество цифр при выводе, m не должно превосходить w. Если количество цифр в числе меньше, чем ширина поля вывода, оно дополняется пробелами. Если w равно нулю, 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.d, Ew.d, Dw.d, Ew.dEe, EN и ES. Действие дескрипторов F, D, E, EN и ES при вводе совпадает. Значение w задает ширину поля, значение d задает количество цифр дробной части. Поле состоит из необязательного знака, одной или нескольких цифр, которые могут содержать десятичную точку. Если десятичной точки нет, правые d цифр интерпретируются как дробная часть числа. Порядок может задаваться целой константой со знаком или с использованием символов E и D (в некоторых реализациях языка допускается также Q, дескриптор "четырехкратной точности"), 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, модифицируется заданным масштабным множителем и округляется до d десятичных цифр. Если w равно нулю, для вывода используется необходимое количество позиций.

При использовании дескрипторов E и D поле вывода может содержать пробелы в старших позициях, необязательный знак, ноль (его наличие зависит от масштабного множителя), десятичную точку, d значащих цифр (преобразование выполняется с округлением), символ E или D, знак порядка, сам порядок с e цифрами, если используется расширенная форма дескриптора Ew.dEe, или двумя цифрами в любом другом случае.

Дескриптор EN формирует поле вывода в инженерной нотации так, что значение порядка кратно трем и абсолютное значение мантиссы больше или равно 1, но меньше 1000, за исключением случая, когда выводимое значение равно нулю. Масштабный множитель на выводимое значение не влияет. Дескрипторы ENw.d и ENw.dEe показывают, что внешнее поле занимает w позиций, дробная часть состоит из d цифр, а порядок содержит e цифр.

Дескриптор ES формирует поле вывода в виде вещественного числа в научной (математической) нотации, так что абсолютное значение значащей части больше или равно 1 и меньше 10, кроме случая, когда выводимое значение равно нулю. Масштабный множитель на выводимое значение не влияет. Дескрипторы ESw.d и ESw.dEe показывают, что внешнее поле занимает w позиций, дробная часть содержит d цифр, а порядок содержит e цифр. При вводе преобразования с ES и EN выполняются аналогично преобразованиям для дескриптора F.

Передача одного комплексного элемента выполняется с помощью двух вещественных дескрипторов. Первый из них определяет преобразование вещественной части, а второй — мнимой. Эти дескрипторы могут быть разными. Между ними могут находиться управляющие спецификации формата. Дескриптор символьных данных может находиться между обоими дескрипторами только при выводе.

Дескриптор данных логического типа Lw 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 Aw задает ширину поля ввода-вывода символьной строки в w позиций. Это значение может быть пропущено, тогда ширина поля равна количеству символов в символьной 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, 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 учитываются. Внутри поля два соседних символа-ограничителя считаются одним символом, и он утрачивает роль ограничителя.

Дескрипторы обобщенного преобразования Gw.d и Gw.dEe можно использовать с элементом списка ввода-вывода любого встроенного типа. Эти дескрипторы указывают, что внешнее поле содержит w позиций, дробная часть которого состоит самое большее из d цифр, а порядок содержит e цифр. Значения d и e игнорируются при использовании с данными целого, логического или символьного типа.

С данными целого типа дескрипторы Gw.d и Gw.dEe действуют аналогично дескриптору Iw. С данными вещественного или комплексного типа форма и интерпретация поля ввода такие же, как в случае дескриптора F. Метод представления поля вывода зависит от передаваемого значения. Если десятичная точка находится непосредственно перед или сразу же после значащих цифр, которые необходимо вывести, универсальный дескриптор действует аналогично дескриптору F. В противном случае его действие аналогично действию дескриптора E. Масштабный множитель k не действует, если значение объекта данных лежит вне диапазона, допускающего эффективное использование дескриптора F. Для данных логического типа дескрипторы Gw.d и Gw.dEe действуют аналогично дескриптору Lw, а для символьных данных — аналогично дескриптору Aw.

Следующая группа дескрипторов (управляющие дескрипторы) управляет форматом и действием последующих дескрипторов преобразования данных.

Дескрипторы позиционирования Tn, TLn, TRn и nX управляют положением символа в текущей записи. Они позволяют выполнять ввод из одной записи дважды. Возможно применение разных преобразований или, например, пропуск некоторых символов. Дескриптор Tn выполняет табуляцию в позицию n от начала записи. Дескрипторы TLn и TRn выполняют переход на n символов влево и вправо, соответственно, от текущей позиции.

Дескриптор "слэш" завершает передачу данных из текущей записи или в нее. Файловый указатель при этом перемещается в начало следующей записи. При выводе в файл, открытый для последовательного доступа, в конец файла добавляется новая запись.

Форматирование, управляемое списком, задается с помощью символа "звездочка" в операторах ввода-вывода:

```
READ*, A
```

или

```
READ(UNIT = 14, FMT = *) I, J
```

Пример форматного вывода приведен в листинге 9.8.

**Листинг 9.8. Пример форматного вывода**

```
PROGRAM FORMAT_DEMO
 IMPLICIT NONE
 REAL :: X
 CHARACTER(LEN = 10) :: FORMAT1
 CHARACTER(LEN = *), PARAMETER :: FORMAT2 = "(F12.3, A)"
 FORMAT1 = "(F12.3, A)"
 X = 3.0
 PRINT FORMAT1, X, ' FORTRAN FOREVER!'
```

```
 WRITE(*, FORMAT2) 2 * X, ' BHV '
 WRITE(*, "(F12.3, A)") 3 + SQRT(X), ' SPB 300'
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.

Первый символ записи, передаваемый на принтер или терминал, обычно используется для управления кареткой и не выводится. Он интерпретируется следующим образом:

- 0 — перед выводом пропускаются две строки;
- 1 — прогон до первой строки следующей страницы;
- \+ — вывод без прогона строк;
- пробел или любой другой символ — прогон одной строки.

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, спецификация формата должна полностью содержаться в этом элементе. Если выражение является символьной константой, ограниченной апострофами (кавычками), для представления апострофа в спецификации формата необходимо использовать два последовательных апострофа (кавычки):

```
PRINT '("NUM can''t be a real number")'
```

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 9.9) спецификация формата изменяется при каждой итерации цикла DO.

**Листинг 9.9. Пример изменения формата в процессе работы программы**

```
PROGRAM FORMAT_DEMO
 REAL TABLE(2, 5)
 CHARACTER * 5 FORCHR(0:5), RPAR * 1, FBIG, FMED, FSML
```

```
 DATA FORCHR(0), RPAR /'(', ')'/
 DATA FBIG, FMED, FSML /'F8.2,', 'F9.4,', 'F9.6,'/
 DATA TABLE /0.001, 0.1, 1., 99., 1000., &
2000.001, 0.0153, 1.0345, 199., 1000. /
 DO I = 1, 2
  DO J = 1, 5
    IF(TABLE(I, J) .GE. 100.) THEN
      FORCHR(J) = FBIG
    ELSE IF(TABLE(I, J) .GT. 0.1) THEN
      FORCHR(J) = FMED
    ELSE
      FORCHR(J) = FSML
    END IF
  END DO
  FORCHR(5)(2:5) = RPAR
  WRITE(6, FORCHR) (TABLE(I, J), J = 1, 5)
 END DO
END PROGRAM FORMAT_DEMO
```

Разберите работу этой программы самостоятельно. Заметим лишь, что форматы здесь размещаются в символьных переменных. Спецификации формата, хранящиеся в массивах, обрабатываются при каждом использовании во время выполнения программы. Вывод выглядит следующим образом:

```
0.001000   1.0000 1000.00 0.015300    199.0000000
0.100000  99.0000 2000.00   1.0345   1000.0000000
```

Пример сравнения дескрипторов `E`, `EN`, `ES` и `G` дан в программе `DESCRIPTORS` (листинг 9.10).

**Листинг 9.10. Сравнение дескрипторов**

```
PROGRAM DESCRIPTORS
 IMPLICIT NONE
 REAL, DIMENSION(4) :: A = (/3.14159, -0.5, 0.00123, 1965.04/)
 PRINT '(4E14.3/ 4EN14.3/ 4ES14.3/ 4G14.3)', A, A, A, A
END PROGRAM DESCRIPTORS
```

Результат выполнения этой программы:

```
0.314E+01    -0.500E+00     0.123E-02     0.197E+04
3.142E+00  -500.000E-03     1.230E-03     1.965E+03
3.142E+00    -5.000E-01     1.230E-03     1.965E+03
3.14         -0.500         0.123E-02     0.197E+04
```

В листинге 9.11 приведен текст программы, выполняющей подсчет символов в текстовом файле, имя которого вводится пользователем.

**Листинг 9.11. 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, IOS
 WRITE(*, *) 'TYPE IN FILE NAME:'
 READ(*, *) FILE_NAME
 OPEN(UNIT = 11, FILE = FILE_NAME, &
 STATUS = "OLD", ACTION = "READ", POSITION = "REWIND")
 CHAR_COUNT = 0
 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
```

Перед дескриптором может быть указан коэффициент повторения:

```
WRITE(*, '(I7, 3F9.4, A10)') IJ, CER(3), SYMB
```

# Соединение файла с логическим устройством

Оператор `OPEN` выполняет одно из следующих действий: соединяет внешний файл с логическим устройством, создает новый файл и соединяет его с устройством или изменяет атрибуты чтения и записи:

```
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, номер логического устройства должен быть первым. Спецификатор FILE позволяет определить имя файла. Имя может быть любым именем, допускаемым операционной системой. Младшие пробелы в имени игнорируются. Если ключевое слово FILE отсутствует, а устройство не соединено с файлом, оператор OPEN должен содержать спецификатор STATUS = 'SCRATCH'. Значением спецификатора ERR является метка оператора, получающего управление в случае ошибки. Спецификатор IOSTAT задает скалярную целую переменную, которой присваивается положительное целое значение (номер сообщения об ошибке), отрицательное, если встретилась запись "конец файла" и ноль при нормальном завершении операции. Основные спецификаторы, которые можно использовать в операторе OPEN, перечислены в табл. 9.1 и 9.2. Значения спецификаторов, которые являются скалярными числовыми выражениями, могут быть любым целым, а в некоторых реализациях (Compaq Visual Fortran) и вещественным выражением. Значение выражения вначале преобразуется в целый формат и только после этого используется в операторе OPEN.

С файлом может быть одновременно связано только одно устройство, но одно устройство можно указать в нескольких операторах 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, соединение текущего файла сохраняется;
- если спецификатор FILE задает другое имя файла, предыдущий файл закрывается, а с устройством соединяется новый файл.

При выполнении оператора OPEN положение указателя в файле не изменяется.

Значение спецификатора может определяться символьным выражением, значение которого вычисляется в процессе выполнения программы (символьное значение может содержать только младшие пробелы):

```
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.

Следующий оператор создает файл на магнитной ленте:

```
OPEN(UNIT = I, FILE = '/ dev/ rmt6', STATUS = 'NEW', RECL = 1024)
```

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;
- 'NEW' — создать новый файл. Если файл уже существует, возникает ошибка. При создании файла ему присваивается статус 'OLD';
- 'SCRATCH' — неименованный временный файл (scratch file). 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, он удаляется, а новый файл создается с тем же именем;
- 'UNKNOWN' — файл может существовать, а может не существовать. Если файл не существует, создается новый файл, а его статус изменяется на 'OLD'.

По умолчанию для спецификатора STATUS используется значение 'UNKNOWN'. Оно используется по умолчанию, если файл открывается неявно с помощью оператора WRITE. Если файл открывается неявно с помощью оператора READ, по умолчанию используется значение 'OLD'. Спецификатор STATUS может также использоваться в операторах CLOSE для определения статуса файла после его закрытия.

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.

В операторе CLOSE может быть указано устройство, которое не было открыто. В этом случае никакие действия не выполняются. Не обязательно и 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. Если параметр оператора CLOSE является выражением с вызовом функции, эта функция не должна выполнять операцию ввода-вывода, поскольку в этом случае результат непредсказуем.

В следующем примере файл, связанный с логическим устройством 17, удаляется после закрытия:

```
CLOSE(17, STATUS = 'DELETE', ERR = 100)
```

Если при выполнении операции произойдет ошибка, управление будет передано оператору с меткой 100.

Пример обработки ошибок при соединении файла дан в листинге 9.12.

**Листинг 9.12. Пример обработки ошибок при соединении файла**

```
PROGRAM IO_ERR
 CHARACTER(LEN = 20) :: FILE_NAME
 DO I = 1, 3
       FILE_NAME = ''
       WRITE(6, *)  'TYPE FILE NAME '
       READ(5, *) FILE_NAME
       OPEN(UNIT = 12, FILE = FILE_NAME, STATUS = 'OLD', &
& IOSTAT = IERR, ERR = 100)
   WRITE(6, *) 'OPENING FILE: ', FILE_NAME
      CLOSE(UNIT = 12)
      STOP
  100 WRITE(6, *) 'ИМЯ ФАЙЛА: ', FILE_NAME, &
& ' НЕВЕРНО, ВВЕДИТЕ ДРУГОЕ'
 END DO
 WRITE(6, *) 'ФАЙЛ НЕ НАЙДЕН. НАЙДИТЕ ЕГО И ЗАПУСТИТЕ ПРОГРАММУ ВНОВЬ'
END PROGRAM IO_ERR
```

# Операторы управления файловым указателем

Оператор REWIND устанавливает указатель в начало файла последовательного доступа:

```
REWIND([UNIT =]НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА [, ERR] [, IOSTAT])
REWIND НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА
```

Файл должен находиться на диске или на магнитной ленте. Если файловый указатель уже находится в начале файла, при выполнении оператора REWIND ничего не происходит. Ничего не происходит и в том случае, когда указанное устройство не соединено с файлом. Пример использования оператора REWIND:

```
REWIND(UNIT = 25, IOSTAT = ISTATUS, ERR = 1000)
```

В этом случае указатель устанавливается в начало файла, связанного с логическим устройством 25. Если выполнить операцию по какой-то причине нельзя, переменной ISTATUS присваивается положительное целое значение, а управление передается оператору с меткой 1000.

Оператор BACKSPACE устанавливает файловый указатель в начало предыдущей записи:

```
BACKSPACE([UNIT =]НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА [, ERR] [, IOSTAT])
BACKSPACE НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА
```

Номер логического устройства должен задавать открытый файл на диске или магнитной ленте. Назначение спецификаторов ERR и IOSTAT читателю уже известно.

Оператор BACKSPACE используется с файлами, открытыми для последовательного доступа, но его нельзя использовать для пропуска записей, которые были созданы с помощью управляемого списком или NAMELIST-форматирования. Он также не применяется с файлами, открытыми для прямого доступа.

Переход на предыдущую запись из текущей записи с номером $n$ выполняется в два этапа — сначала перемотка в начало файла, а затем $n - 1$ считывание. Если указатель расположен между последней записью и записью "конец файла", BACKSPACE устанавливает указатель в начало последней записи.

В результате выполнения любого из операторов:

```
BACKSPACE 15
BACKSPACE(15)
```

указатель в файле, связанном с логическим устройством 15, перемещается на одну запись назад. Оператор:

```
BACKSPACE(UNIT = 19, IOSTAT = WSTAT, ERR = 230)
```

перемещает указатель в файле, связанном с устройством 19, на предыдущую запись. При возникновении ошибки управление передается оператору с меткой 230, а в переменную WSTAT записывается положительное целое значение.

При выполнении оператора ENDFILE в последовательный файл добавляется запись "конец файла", а указатель устанавливается после этой записи. Для файлов прямого доступа этот оператор обрезает файл после текущей записи (Compaq Visual Fortran).

```
ENDFILE([UNIT =]НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА [, ERR] [, IOSTAT])
ENDFILE НОМЕР_ЛОГИЧЕСКОГО_УСТРОЙСТВА
```

Если логическое устройство не было открыто, файл открывается для неформатного доступа. Оператор ENDFILE может применяться только к форматным файлам последовательного доступа. Если файл создан программой на другом языке программирования, применять его нельзя. Спецификаторы оператора ENDFILE не должны включать вызовы функций, выполняющих операции ввода-вывода. Пример использования:

```
ENDFILE 2
```

# Оператор *INQUIRE*

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 скалярной 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 скалярного символьного выражения DEF. Каждый спецификатор запроса может использоваться в списке только один раз. Опрашиваемое устройство может и не быть связанным с файлом.

При вызове возвращаются значения, которые были текущими в момент выполнения оператора INQUIRE. Для получения характеристик файла этот оператор должен находиться после оператора OPEN. Управляющие спецификаторы (UNIT, ERR и IOSTAT) и спецификаторы запроса могут использоваться

как в ключевой, так и в неключевой формах. Действуют обычные правила их размещения.

Пример использования оператора INQUIRE:

```
INQUIRE(FILE = 'RES.DAT', EXIST = EXIST_CHECK)
```

Программа FILE_EXISTS_OR_NOT (листинг 9.13) запрашивает имя файла, затем проверяет, существует ли этот файл, и если его нет, запрашивается другое имя.

**Листинг 9.13. Пример использования оператора *INQUIRE***

```
PROGRAM FILE_EXISTS_OR_NOT
 CHARACTER(LEN = 10) :: FILE_NAME
 LOGICAL(1) :: EXISTS_OR_NOT
 DO WHILE(.NOT.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
```

# Оператор *NAMELIST*

Оператор NAMELIST связывает с набором переменных общее имя, поэтому он иногда называется именованным списком. Это групповое имя можно использовать в операциях ввода-вывода. Оператор NAMELIST имеет следующий вид:

```
NAMELIST /ИМЯ_ГРУППЫ/ СПИСОК_ПЕРЕМЕННЫХ [[,] /ИМЯ_ГРУППЫ/
СПИСОК_ПЕРЕМЕННЫХ] ...
```

Переменные, которые связаны с групповым именем, могут быть любого типа. Групповое NAMELIST-имя используется NAMELIST-операторами ввода-вывода вместо списка ввода-вывода. Одна переменная может использоваться в нескольких NAMELIST-группах.

Доступ к каждой переменной в списке должен быть организован посредством USE-ассоциирования или ассоциирования через носитель либо должен иметь тип явно или неявно заданный в том же модуле программы, в котором содержится именованный список. Если тип переменной определен неявно, она может появиться в следующем предложении описания типа, только если оно удовлетворяет правилам неявной типизации.

В NAMELIST-группах нельзя использовать:

- ❒ массивы с переменными границами;
- ❒ символьные переменные с подразумеваемой длиной;
- ❒ динамические массивы;
- ❒ автоматические объекты;
- ❒ указатели;
- ❒ переменные производного типа, который содержит указатель в качестве компонента;
- ❒ подобъекты любого из вышеупомянутых объектов.

При выполнении операции ввода-вывода можно считывать или записывать только переменные, указанные в NAMELIST. Порядок переменных в NAMELIST определяет порядок, в котором значения появляются при NAMELIST-выводе. Ввод значений из именованного списка может выполняться в любом порядке. Если групповое имя имеет атрибут PUBLIC, ни один из элементов в списке переменных не может иметь атрибут PRIVATE.

Одно групповое имя может быть указано в нескольких операторах NAMELIST, находящихся в одном модуле программы. Список переменных, указанный после каждого последующего появления одного и того же группового имени, добавляется к соответствующему списку. В примере:

```
NAMELIST /SPACETIME/ X, Y, Z
...
NAMELIST /SPACETIME/ TIME
```

переменная TIME добавляется к переменным X, Y и Z списка, соответствующего групповому имени SPACETIME.

В следующем примере определены два групповых имени:

```
NAMELIST /LST1/ YEAR, MONTH, DAY /LST2/ NAME, TITLE
```

Пример ввода с использованием именованного списка:

```
NAMELIST /INPARAMS/ NOM, FLAG, TEMPER, COMEN, AR
READ(*, NML = INPARAMS)
```

Данные для ввода этого списка могут иметь такой вид:

```
 &EXAMPLE
 NOM  = 10
 FLAG  = .TRUE.
 TEMPER = 299.5D0
 COMEN = (1.0D0, 0.10D0)
 AR(3) = 41, 42, 43
 /
```

NAMELIST-форматирование обозначается оператором ввода-вывода со спецификатором NML. Форматирование NAMELIST-записей такое же, как у записей, управляемых списком. Использование NAMELIST для неформатного ввода-вывода не допускается.

Пример использования NAMELIST-вывода приведен в листинге 9.14.

**Листинг 9.14. Пример использования *NAMELIST*-вывода**

```
PROGRAM NAMELIST_DEMO
 INTEGER :: A
 REAL :: B
 CHARACTER(LEN = 10) :: MESSAGE
 NAMELIST /NMLST/ A, B, MESSAGE
 A = 1
 B = 1.0
 MESSAGE = 'NAMELIST OUTPUT'
 OPEN(UNIT = 4, FILE = 'NM.DAT', DELIM = 'APOSTROPHE')
 WRITE(UNIT = 4, NML = NMLST)
 CLOSE(4)
END PROGRAM NAMELIST_DEMO
```

Вывод имеет следующий вид:

```
&NMLST
 A       =           1,
 B       =   1.000000,
 MESSAGE = 'NAMELIST O'
 /
```

# Повышение производительности ввода-вывода

Операции ввода-вывода принадлежат к числу наиболее медленных операций. Если доля операций ввода-вывода в прикладной программе велика, их оптимизация может привести к существенному повышению производительности и сокращению времени счета. В этом случае следует свести к минимуму излишнее форматирование данных и другие операции, потребляющие процессорное время, а также излишние передачи промежуточных данных. Приведем несколько рекомендаций по оптимизации операций ввода-вывода.

Всегда, когда возможно, следует использовать неформатные файлы. Неформатные данные не требуют преобразования при передаче. Отсутствие преобразований уменьшает потери точности при вводе и выводе. Неформатные данные более компактны. Если, например, имеется массив AR, то запись значений его элементов во внешний файл с помощью оператора:

```
WRITE(12) AR
```

более эффективна, чем с помощью оператора:

```
WRITE(12, '(5(5F9.3))') AR
```

Передавать массивы или строки следует целиком, а не поэлементно. Это связано с тем, что каждый элемент списка ввода-вывода порождает свою собственную вызывающую последовательность машинных команд. Неэффективными конструкциями являются и циклические списки.

Рекомендуется передавать элементы массива в том порядке, в котором они хранятся в памяти. Напомним, что естественным порядком элементов в Фортране является такой, когда крайний левый индекс изменяется быстрее всего, второй слева чуть медленнее и т. д. Используя математическую терминологию, можно сказать, что двумерные массивы (их математический аналог — матрицы) хранятся по столбцам. Если необходимо использовать порядок, отличный от естественного, иногда лучше сначала передать данные в память и переупорядочить их перед выполнением операции ввода-вывода.

Для хранения промежуточных результатов можно использовать оперативную память компьютера. Этот метод может не дать выигрыша в производительности, если объем данных велик по сравнению с физическим объемом памяти компьютера. В этом случае значительное время уходит на выполнение операций "свопинга" (запись на жесткий диск страниц виртуальной памяти).

Динамическое форматирование (форматирование времени выполнения) может существенно снизить производительность. В Compaq Visual Fortran имеется расширение, которое называется "форматным выражением". Это выражение, которое заключается в угловые скобки (< >) и которое можно использовать в операторе FORMAT. Форматные выражения более эффективны, чем динамическое форматирование. Пример использования форматного выражения:

```
WRITE(12, 100) (A(I), I = 1, N)
100 FORMAT(3X, < N > F7.3)
```

При выполнении операций ввода-вывода для ускорения передачи данных используется их предварительное сохранение в специальной области оперативной памяти — буфере ("кеширование"). Это позволяет минимизировать обмен с физическим диском. Если запись имеет небольшой размер, в буфере могут быть сохранены несколько записей. Запись на диск при этом происходит реже. Однако системная авария при использовании буферизованного вывода может привести к потере записей, поскольку в момент сбоя они

могут оказаться еще не записанными на диск. При программировании в среде Compaq Visual Fortran управлять кешированием вывода можно с помощью спецификаторов BUFFERED, BUFFERCOUNT и BLOCKSIZE оператора OPEN.

Рекомендуется выбирать оптимальное значение RECL. Значение RECL должно быть таким, чтобы обеспечить максимальное заполнение блоков. При программировании в среде Compaq Visual Fortran, например, значение RECL должно в целое число раз отличаться от значения спецификатора BLOCKSIZE. В этом случае система может обеспечить эффективное выполнение операций ввода-вывода, когда каждая операция перемещает максимально возможное количество данных. Значение RECL для форматных файлов задается в байтах, а для неформатных файлов RECL измеряется в 4-байтных единицах.

Следует также использовать оптимальный тип записи. Для последовательных файлов это записи с фиксированной длиной. Для последовательных неформатных файлов, когда размер записей не фиксирован, тип записей с переменной длиной дает наилучшую производительность.

## Вопросы и задания для самостоятельной работы

1. Что такое файл?
2. Перечислите виды файлов.
3. Для чего используются внешние файлы?
4. Для чего используются внутренние файлы?
5. Сравните форматные и неформатные файлы с точки зрения эффективности операций ввода-вывода.
6. Что такое запись? Чем она характеризуется?
7. Чем отличаются файлы последовательного и прямого доступа?
8. Для чего используются дескрипторы преобразования данных?
9. Перечислите дескрипторы форматирования числовых значений.
10. Перечислите дескрипторы форматирования логических значений.
11. Перечислите дескрипторы форматирования символьных значений.
12. Какие дескрипторы применяются для форматирования комплексных значений?
13. Перечислите дескрипторы управления форматом.
14. Опишите последовательность действий по записи данных в файл.
15. Перечислите основные спецификаторы, используемые в операторах OPEN и CLOSE.

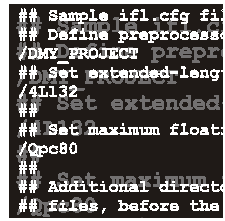
16. Для чего предназначен оператор `INQUIRE`?
17. Перечислите операторы управления положением файлового указателя.
18. Расскажите об именованных списках.
19. Расскажите об оптимизации операций ввода-вывода.
20. Что является результатом выполнения следующего оператора?

```
WRITE(*, "(HI)")
```

21. Что происходит при выполнении следующего фрагмента программы?

```
CHARACTER(LEN = 9) :: F
F = '(1PG14.6)'
WRITE(*, F) 0.001, 1.0, 1000000
```

22. 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.
23. Напишите программу, которая добавляет заданное количество записей в конец файла, созданного программой из задания 22.
24. Как будут выводиться нижеприведенные значения с указанным дескриптором:
    - −123 (`I4`);
    - −226 (`I3`);
    - −213.155 (`F6.1`);
    - −3456.789 (`E10.4`);
    - 0.000012013 (`E9.2`).
25. Напишите программу, которая считывает положительное целое значение, преобразует его в двоичное представление и выводит это представление на экран.




# Глава 10

# Встроенные функции и процедуры

Встроенные функции определяют возможности языка в той области программирования, для которой он предназначен. Язык, созданный для научных расчетов, должен включать в себя наиболее используемые математические функции, чтобы не вынуждать каждого программировать их для себя заново. Как правило, качество программ, поставляемых вместе с компилятором, много выше качества процедур, написанных прикладными программистами. Некоторые встроенные функции предоставляют программисту информацию, которую он не смог бы получить с помощью других средств. Например, справочные функции позволяют определить наличие необязательного параметра, состояние ссылки или длину текстовой константы. Такие средства расширяют возможности языка, способствуя разработке надежных и гибких приложений.

Набор встроенных функций и процедур Фортрана, предусмотренный стандартом языка, весьма широк, он включает в себя около ста функций и процедур (описания процедур отмечены оператором вызова `CALL` перед именем процедуры).

С целью создания удобного для чтения текста описания неформально разделены на смысловые группы, скомпонованные таким образом, чтобы число групп было хотя бы в несколько раз меньше числа функций. В группу "Справочные функции", например, внесены не только функции с результатом логического типа, но и функции поиска в массивах.

Полный перечень встроенных функций и процедур в алфавитном порядке приведен в *Приложении 2*.

## Оператор *INTRINSIC*

Оператор `INTRINSIC`, как и атрибут `INTRINSIC`, указывают на принадлежность определяемых ими переменных к именам встроенных функций. Оператор имеет вид:

```
INTRINSIC список_встроенных_имен
```

После этого оператора все имена, внесенные в список_встроенных_имен, рассматриваются как имена встроенных функций или процедур. Если среди встроенных процедур не находится процедура с указанным именем, выдается сообщение об ошибке. Если имя объявлено именем встроенной функции, любая внешняя функция с таким же именем становится недоступной; появление такого же имени в операторе EXTERNAL в том же блоке видимости недопустимо. 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: исключается всякая неоднозначность в толковании имен, нестандартные встроенные функции, используемые в программе, при переносе на систему, не поддерживающую их, точно диагностируются, исключается случайное использование имени встроенной функции в подпрограммах пользователя.

# Ключевые параметры встроенных функций и процедур

Почти все встроенные функции имеют необязательные параметры, которые можно задавать как ключевые, используя в качестве ключей имена формальных параметров. Некоторые функции не имеют ни одного строго обязательного параметра, но требуют присутствия одного из необязательных в ключевом варианте.

# Справочные функции

К справочным здесь отнесены все функции, являющиеся справочными по своему смыслу, а не по формальным признакам — то есть сообщающие информацию о состоянии объекта, особенностях его устройства, разновидности типа, о среде программирования и т. п. В результате, в раздел вошли описания функций, возвращающих параметры модели представления чисел в системе, разновидность типа для заданных степенных диапазонов и процедуры определения времени и даты.

Справочные функции не являются элементными, хотя некоторые из них равно принимают в качестве аргументов и скаляры, и массивы.

## Справочные функции для любого типа

Следующие справочные функции применимы к любому типу данных.

- ASSOCIATED(POINTER[, TARGET]) — возвращает переменную логического типа. Если аргумент TARGET опущен, анализируется состояние привязан-

ности ссылки POINTER. В этом случае ASSOCIATED возвращает .TRUE., если ссылка POINTER прикреплена к какому-либо адресату, и .FALSE. в противном случае. Если аргумент TARGET присутствует, то анализируется состояние привязанности ссылки POINTER к адресату TARGET. Значение .TRUE. возвращается, если адресатом ссылки POINTER является аргумент TARGET, во всех остальных случаях возвращается .FALSE. TARGET, в свою очередь, также может быть ссылкой; тогда выясняется, привязан ли аргумент POINTER к адресату TARGET. ASSOCIATED не возвращает никакого результата для аргументов, находящихся в состоянии неопределенности и приводит к ошибке времени исполнения. ASSOCIATED в применении к массивам требует совпадения формы аргументов POINTER и TARGET и соответствия порядка следования элементов ссылки элементам адресата.

- PRESENT(A) — анализирует наличие необязательного формального параметра A в списке фактических параметров подпрограммы. Возвращает скалярную переменную логического типа. .TRUE. возвращается в случае присутствия фактического параметра в текущем вызове, в противном случае возвращается .FALSE.
- KIND(X) — возвращает переменную целого типа стандартной разновидности, значение которой равно разновидности типа аргумента X.

## Числовые справочные функции

Справочные функции этого раздела связаны с моделями представления вещественных и целых чисел для каждой из реализованных разновидностей этих типов.

Разновидности целого типа могут быть выражены в виде суммы произведений вида:

$$i = s \times \sum_{k=1}^{n} a_k \times b^{k-1},$$

где:

- $s = \pm 1$ определяет знак числа;
- $n$ — положительное целое число, определяемое разрядностью числа;
- $b$ — основание модели — положительное целое число, большее 1 (как правило, 2);
- $a_k$ — положительные целые числа от нуля до $b$ ($0 \le a_k < b$, и если $b = 2$, то это нули или единицы).

Разновидности вещественного типа моделируются множеством:

$$x = 0 \bigcup x = s \times b^e \times \sum_{k=1}^{n} f_k \times b^{-k},$$

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 девять функций выдают характеристики модели, соответствующей их аргументу. Аргумент у этих функций всегда один, причем важен только его тип, а значение на результат функций не влияет. Возвращаемый результат не зависит от ранга аргумента: 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`, т. е. $b^{1-n}$. Возвращаемое значение имеет ту же разновидность типа, что и `X`; оно является минимальным положительным числом, сумма которого с единицей воспринимается как отличное от единицы число.
- `HUGE(X)` — возвращает максимальное значение в модели, представляющей вещественный или целый аргумент `X`, т. е. $(1-b^{-n}) \cdot b^{e_{\max}}$ для вещественных аргументов и $b^n - 1$ для целых.
- `MAXEXPONENT(X)` — возвращает в виде стандартного целого максимальный показатель степени в модели, представляющей вещественный аргумент `X`, т. е. $e_{\max}$.
- `MINEXPONENT(X)` — возвращает в виде стандартного целого минимальный показатель степени в модели, представляющей вещественный аргумент `X`, т. е. $e_{\min}$.
- `PRECISION(X)` — возвращает в виде стандартного целого 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.
- `RADIX(X)` — возвращает в виде стандартного целого основание в модели, представляющей вещественный или целый аргумент `X`, т. е. $b$.
- `RANGE(X)` — возвращает стандартное целое, 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` соответственно равны наибольшему и наименьшему положительному числам в модели.

- `TINY(X)` — возвращает наименьшее положительное значение в модели, представляющей вещественный или комплексный аргумент `X`, т. е. $b^{e_{\min}}$. Тип и разновидность типа возвращаемого значения совпадают с типом и разновидностью типа аргумента.

Следующие две функции определяют разновидность типа, обеспечивающую заданный аргументом диапазон значений. Они применимы только к скалярам и в результате возвращают скаляр. Возвращаемый параметр разновидности типа всегда является минимальным, т. е. определяется наименее "затратная" разновидность, удовлетворяющая предъявленным требованиям.

- `SELECTED_INT_KIND(R)` — возвращает в виде стандартного целого параметр такой разновидности целого типа, которая обеспечивает представление целого числа n в диапазоне $-10^R < n < 10^R$, где R — скаляр целого типа. Если находится несколько разновидностей, обеспечивающих этот диапазон, то выбирается наименьшая из них. Если подходящей разновидности нет, то возвращается $-1$.
- `SELECTED_REAL_KIND([P][,R])` — возвращает в виде стандартного целого параметр разновидности вещественного типа, обеспечивающего десятичную точность (в смысле функции `PRECISION`) не хуже `P` (в смысле функции `RANGE`) в степенном диапазоне не меньше `R`, где `P` и `R` — скаляры целого типа. Оба аргумента необязательные, но хотя бы один из них обязан присутствовать. Подходящей разновидности может и не быть. Тогда, если недостижима заданная точность, возвращается $-1$; если недоступен степенной диапазон, то возвращается $-2$; если недоступны одновременно и точность, и диапазон, то $-3$. Если существуют несколько разновидностей, удовлетворяющих заданным точности и диапазону, то выбирается наименьшая из них.

## Справочная функция для строк

`LEN(S)` — возвращает стандартное целое, равное числу символов в строке `S`. Если `S` — массив строк, то возвращается длина строки — элемента массива.

## Справочные функции для массивов

Пять функций этой категории поставляют информацию о состоянии и форме массивов.

- `ALLOCATED(ARRAY)` — отмечает состояние массива, описанного как `ALLOCATABLE`. Возвращает `.TRUE.` для массивов, память под которые

в настоящий момент выделена, и .FALSE., если память под массив ARRAY не выделена. Если аргумент находится в состоянии неопределенности, то и результат функции ALLOCATED будет неопределенным.

- LBOUND(ARRAY [, DIM]) — возвращает стандартное целое значение, равное значению нижней границы массива ARRAY по индексу DIM, или одномерный массив стандартных целых, содержащий значения нижних границ массива ARRAY, если аргумент DIM отсутствует.
- UBOUND(ARRAY [, DIM]) — возвращает стандартное целое значение, равное значению верхней границы массива ARRAY по индексу DIM, или одномерный массив стандартных целых, содержащий значения верхних границ массива ARRAY, если аргумент DIM отсутствует.
- SHAPE(SOURCE) — возвращает в виде массива стандартных целых 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 стандартного целого размер массива ARRAY или, если присутствует целый скалярный аргумент DIM, величину экстента массива ARRAY по индексу DIM.

## Функции для поиска в массиве

Следующие четыре функции позволяют находить положение и значение экстремальных элементов массивов.

- MAXLOC(ARRAY [, MASK]) — возвращает в виде одномерного массива стандартных целых индекс элемента целого или вещественного массива, имеющего 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. Значения индексов отсчитываются так, как будто все нижние границы массива ARRAY равны 1. Если максимальным значением обладает несколько элементов массива ARRAY, то возвращается индекс первого из них в порядке следования элементов массива.
- MAXVAL(ARRAY [, DIM] [, MASK]) — возвращает максимальное значение элемента целого или вещественного массива. Необязательный аргумент DIM приводит к поиску максимальных значений по всем одномерным сечениям, проходящим по индексу DIM. В результате 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, конформ-

ный исходному массиву. В случае присутствия этого аргумента, максимальный элемент выбирается только из тех элементов, которым соответствуют элементы массива MASK, имеющие значение .TRUE. Если исходный массив имеет нулевой размер, то возвращается наибольшее по абсолютной величине отрицательное число, допускаемое процессором.

- MINLOC(ARRAY [, MASK]) — возвращает в виде одномерного массива стандартных целых индекс элемента целого или вещественного массива, имеющего минимальное 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. Значения индексов отсчитываются так, как будто все нижние границы массива ARRAY равны 1. Если минимальным значением обладает несколько элементов массива ARRAY, то возвращается индекс первого из них в порядке следования элементов массива.
- MINVAL(ARRAY [, DIM] [, MASK]) — возвращает минимальное значение элемента целого или вещественного массива. Необязательный аргумент DIM приводит к поиску минимальных значений по всем одномерным сечениям, проходящим по индексу DIM. В результате возвращается массив минимальных 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, конформный исходному массиву. В случае присутствия этого аргумента, минимальный элемент выбирается только из тех элементов, которым соответствуют элементы массива MASK, имеющие значение .TRUE. Если исходный массив имеет нулевой размер, то возвращается наибольшее положительное число, допускаемое процессором.

# Процедуры определения даты и времени

Информацию часов реального времени можно получить с помощью двух следующих процедур. Первая из них соответствует стандарту 8601 Международной организации по стандартизации (International Standard Organization, ISO) представления времени и дат.

Обе процедуры предполагают наличие системного таймера, содержимое которого при каждом отсчете увеличивается на единицу до достижения некоторого максимального значения, после чего счетчик таймера сбрасывается в нуль и начинается следующий цикл. В тех системах, где таймера нет, процедуры возвращают не имеющие смысла значения, установленные по умолчанию: вместо текстовых величин возвращаются пробелы, вместо целых, если не указано другое, выдается число, равное -HUGE(0).

Все аргументы процедур необязательные, но хотя бы один из них должен присутствовать. Все аргументы имеют вид связи OUT.

- CALL DATE_AND_TIME([DATE] [, TIME] [, ZONE] [, VALUES]), причем:
  - DATE — текстовая скалярная переменная стандартной разновидности, содержащая дату в виде CCYYMMDD, где CC — две цифры, определяющие век, YY — год, MM — месяц и DD — день; например, 20030501 для 1-ого мая 2003 года;
  - TIME — текстовая скалярная переменная стандартной разновидности, содержащая время в виде HHMMSS.sss, где HH соответствует часу, MM — минутам, SS — секундам и sss — миллисекундам;
  - ZONE — стандартная текстовая переменная, соответствующая часовому поясу, выражающая разницу между универсальным (UTC) и местным временем;
  - VALUES — одномерный стандартный 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.
- CALL SYSTEM_CLOCK([COUNT] [, COUNT_RATE] [, COUNT_MAX]), где:
  - COUNT — стандартный целый скаляр, приблизительно отражающий значение текущего отсчета системного таймера;
  - COUNT_RATE — стандартный целый скаляр, содержащий число отсчетов таймера в секунду, или 0, если таймер отсутствует;
  - COUNT_MAX — стандартный целый скаляр, содержащий максимальное значение счетчика таймера, или 0, если таймер отсутствует.

# Элементные функции

Функции, представленные в этом разделе, для скалярного аргумента возвращают скаляр, для аргумента-массива — массив результатов. По смыслу функции разделяются на две группы — числовые функции, относящиеся к модельному представлению чисел, и математические функции.

## Числовые функции для действий над действительными числами

Ниже приведен список элементных числовых функций, используемых для определения элементов модельного представления чисел.

- EXPONENT(X) — возвращает в виде стандартного степенную часть модельного представления числа X, а именно *e*. Для X = 0 возвращает нуль.

- FRACTION(X) — возвращает в виде стандартного вещественного дробную часть модельного представления числа X, т. е. $X b^{-e}$.
- NEAREST(X, S) — возвращает вещественной 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 S — вещественного числа, не равного нулю.
- RRSPASING(X) — возвращает в виде вещественного значение той же, что и X, разновидности типа величину, обратную относительному расстоянию между числами модели в области X, а именно $|X b^{-e}| b^{n}$.
- SCALE(X,I) — возвращает вещественное значение с тем же параметром типа, что и X, равное по величине $X b^{I}$, где I — аргумент целого типа, а $b$ является основанием в модели представления X.
- SET_EXPONENT(X, I) — возвращает вещественное значение той же, что и X, разновидности, дробная часть которого равна дробной части X, а степенная часть равна I, а именно $X b^{I-e}$.
- SPACING(X) — возвращает в виде вещественного значения той же, что и X, разновидности типа абсолютное расстояние между числами модели в области X, т. е. $b^{e-n}$, или TINY(X), если $e - n$ выходит за пределы степенного диапазона.

## Математические функции

Ниже приведен список основных 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. Разновидность типа возвращаемой величины совпадает с разновидностью типа аргумента.
- ACOS(X) — для вещественного аргумента, не превосходящего по модулю 1 ($|X| \le 1$), возвращает главное значение арккосинуса, выраженное в радианах ($0 \le$ ACOS(X) $\le \pi$). Разновидность типа результата функции совпадает с разновидностью типа аргумента.
- AIMAG(Z) — возвращает вещественное значение, равное мнимой части комплексного аргумента Z. Разновидность типа результата функции совпадает с разновидностью типа аргумента.
- AINT(A [, KIND]) — возвращает в виде вещественного значения стандартной (или заданной необязательным аргументом KIND) разновидности ближайшее к A со стороны нуля целое число.
- ANINT(A [, KIND]) — возвращает в виде вещественного значения стандартной (или заданной необязательным аргументом KIND) разновидности 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 ($-\pi / 2 \le$ ASIN(X) $\le \pi / 2$). Разновидность типа результата функции совпадает с разновидностью типа аргумента.
- ATAN(X) — для вещественного аргумента возвращает главное значение арктангенса, выраженное в радианах ($-\pi / 2 \le$ ATAN(X) $\le \pi / 2$). Разновидность типа результата функции совпадает с разновидностью типа аргумента.
- ATAN2(Y, X) — для пары вещественных аргументов Y и X одной разновидности типа возвращает главное значение аргумента комплексного числа (X, Y) в интервале ($-\pi \le$ ATAN2(Y, X) $\le \pi$), выраженное в радианах. Разновидность типа результата функции совпадает с разновидностью типа аргументов.
- CEILING(A) — для вещественного аргумента A возвращает в виде стандартного целого наименьшее целое число, не меньшее A.
- CONJG(Z) — возвращает сопряженное значение комплексного аргумента Z с разновидностью типа, равной разновидности типа Z.
- COS(X) — для вещественного или комплексного аргумента возвращает значение косинуса, интерпретируя аргумент как значение в радианах. Разновидность типа результата совпадает с разновидностью типа аргумента.
- COSH(X) — для вещественного аргумента возвращает значение гиперболического косинуса той же, что и аргумент, разновидности типа.
- DIM(X, Y) — для двух целых или двух вещественных аргументов одинаковой разновидности возвращает max(X - Y, 0). Разновидность типа результата совпадает с разновидностью типов аргументов.
- DPROD(X, Y) — для двух вещественных скаляров стандартной разновидности возвращает их произведение с двойной точностью.
- EXP(X) — для вещественного или комплексного аргумента возвращает значение экспоненциальной функции. Разновидность результата совпадает с разновидностью аргумента.
- FLOOR(A) — возвращает в виде стандартного целого наибольшее целое число, не превышающее вещественный аргумент A. (Сравните с AINT.)
- LOG(X) — для вещественного положительного или отличного от нуля комплексного аргумента возвращает значение натурального логарифма. Мнимая часть результата находится в интервале $[-\pi, \pi]$. Разновидность типа результата совпадает с разновидностью типа аргумента.
- LOG10(X) — для вещественного положительного аргумента возвращает значение десятичного логарифма. Разновидность типа результата совпадает с разновидностью типа аргумента.

- `MAX(A1, A2 [, A3, …..])` — для двух или более целых или вещественных аргументов возвращает значение максимального из них; тип и разновидность типа результата совпадает с типом и разновидностью типа аргументов — все аргументы должны быть одного типа и разновидности.
- `MIN(A1, A2 [, A3, …..])` — для двух или более целых или вещественных аргументов возвращает значение минимального из них; тип и разновидность типа результата совпадает с типом и разновидностью типа аргументов — все аргументы должны быть одного типа и разновидности.
- `MOD(A, P)` — для пары целых или вещественных аргументов одинакового типа возвращает остаток от `A` по модулю `P`, а именно: `A - INT(A / P) * P`. Результат для `P = 0` зависит от процессора.
- `MODULO(A, P)` — для пары целых или вещественных аргументов одинакового типа возвращает `A` по модулю `P`, а именно: `A - FLOOR(A / P) * P` для вещественных и `A - FLOOR(A:P) * P` для целых аргументов. Результат для `P = 0` зависит от процессора.
- `SIGN(A, B)` — для пары целых или вещественных аргументов одинакового типа возвращает значение, равное `|A| * sign(B)`. Знак `B = 0` считается положительным.
- `SIN(X)` — для вещественного или комплексного аргумента, который интерпретируется как значение в радианах, возвращает значение синуса. Разновидность типа результата совпадает с разновидностью типа аргумента.
- `SINH(X)` — для вещественного аргумента `X` возвращает значение гиперболического косинуса. Разновидность типа результата совпадает с разновидностью типа аргумента.
- `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 интерпретируется как значение в радианах, возвращает значение тангенса. Разновидность типа результата совпадает с разновидностью типа аргумента.
- `TANH(X)` — для вещественного аргумента `X` возвращает значение гиперболического тангенса. Разновидность типа результата совпадает с разновидностью типа аргумента.

## Функции преобразования и переноса типов

Ниже приведен список дополнительных математических функций, используемых для согласования типов.

- `AIMAG(Z)` — возвращает вещественное значение, равное мнимой части комплексного аргумента `Z`. Разновидность типа результата функции совпадает с разновидностью типа аргумента.

- CMPLX(X [, Y] [, KIND]) — для аргумента X или пары аргументов (X, Y) возвращает комплексное значение стандартной или заданной аргументом KIND разновидности комплексного типа. Если аргумент Y отсутствует, 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 — целого или вещественного типа одной разновидности.
- INT(A [, KIND]) — преобразует аргумент к целому типу стандартной или заданной аргументом KIND разновидности. 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.
- LOGICAL(L [, KIND]) — преобразует аргумент логического типа к стандартной или заданной аргументом KIND разновидности логического типа.
- NINT(A [, KIND]) — возвращает ближайшее к вещественному аргументу A целое стандартной или заданной аргументом KIND разновидности.
- REAL(A [, KIND]) — преобразует аргумент A к вещественному типу стандартной или заданной аргументом KIND разновидности. Аргумент A может быть целого, вещественного или комплексного типа. Для аргумента комплексного типа возвращаемая величина равна вещественной части аргумента; разновидность типа комплексного аргумента, если параметр KIND отсутствует, сохраняется. Для целых и вещественных аргументов в таком случае разновидность типа стандартная.
- 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 и разновидности, 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, то 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 затравочного массива можно получить с помощью подпрограммы RANDOM_SEED, с помощью этой же процедуры можно установить "затравку" заново. Другая подпрограмма,

RANDOM_NUMBER, позволяет получить сгенерированную последовательность псевдослучайных чисел в виде скаляра или массива вещественного типа.

CALL RANDOM_NUMBER(HARVEST) — помещает псевдослучайное число из равномерно распределенных случайных значений в интервале [0, 1] или массив таких чисел в аргумент HARVEST. Аргумент HARVEST имеет вид связи OUT и должен быть вещественным скаляром или массивом.

CALL 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.

# Операции над массивами

В Фортране создан очень удобный комплекс встроенных функций для работы с массивами. Кроме справочных функций, в него входят 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.

## Функции для создания массивов и операций над ними

Ниже приведен список общих функций, используемых для создания и преобразования массивов.

- CSHIFT(ARRAY, SHIFT [, DIM]) — функция, циклически смещающая элементы массива на заданное число позиций. 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 B(I) = A(I + 1), а B(N) = A(1). CSHIFT возвращает массив, размеры и форма которого совпадают с размерами и формой массива ARRAY. DIM — скаляр целого типа, который определяет индекс сечений, в которых производится сдвиг. Если этот аргумент опущен, то по умолчанию принимается значение DIM = 1. SHIFT задает шаг сдвига; этот аргумент обязан быть скаляром, если ARRAY является одномерным масси-

вом. Если же ARRAY — многомерный массив, 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 и разновидность должны совпадать с типом и разновидностью 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 конформный исходным массивам массив, элементы которого совпадают, в зависимости от значения соответствующего элемента логического массива MASK, либо с элементами TSOURCE, либо с элементами FSOURCE. Тип и разновидность типа массивов FSOURCE и TSOURCE должны быть одинаковыми; оба этих аргумента могут быть скалярами, тогда они будут расширены до массивов требуемых размеров.
- PACK(ARRAY, MASK [, VECTOR]) — функция "упаковки" массива. Заданные логическим массивом MASK элементы массива ARRAY помещаются в одномерный массив. Если аргумент VECTROR опущен, 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 переформирования массива. Возвращает массив, форма которого задана одномерным массивом целого типа SHAPE. Если размер переформированного массива превосходит размер исходного массива SOURCE, то недостающие элементы будут подставлены из массива PAD. Порядок изменения индексов массива, если он отличается от обычного, 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 последо-

вательность, совпадающую с естественной последовательностью элементов массива SOURCE, за которой следуют элементы массива PAD. Если аргумент PAD отсутствует или имеет нулевой размер, результирующий массив не может превышать исходный массив. Если массив ORDER не задан, элементы массива SOURCE копируются в результирующий массив в порядке следования.

- SPREAD(SOURCE, DIM, NCOPIES) — функция репродуцирования 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. То есть элемент исходного массива $(s_1, ..., s_n)$ получается из элемента результирующего массива $(r_1, ..., r_n + 1)$ изъятием индекса DIM.
- UNPACK(VECTOR, MASK, FIELD) — функция распаковки "упакованного" массива (см. PACK). Возвращает массив того же типа и разновидности, что и одномерный массив VECTOR с формой логического массива MASK. Если элементу VECTOR соответствует (в порядке следования элементов массивов) истинный элемент массива MASK, 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 числа истинных элементов массива MASK. FIELD должен иметь те же тип и разновидность типа, что и VECTOR; если FIELD — массив, то его форма должна совпадать с формой массива MASK.

## Функции редукции массивов

Функции этого раздела принимают в качестве аргумента массив, а возвращают, в общем случае, одно значение. Этим же свойством обладают MINVAL и MAXVAL из раздела "Справочные функции". Все функции имеют необязательный аргумент DIM. Если DIM присутствует, действие функции применяется ко всем одномерным сечениям, проходящим по индексу DIM. В результате возвращается массив, форма которого совпадает с формой массива-аргумента за исключением индекса DIM, а тип определяется типом результата функции.

- ALL(MASK [, DIM]) — анализирует истинность логического массива MASK. Возвращает значение .TRUE. в том случае, когда все элементы массива MASK истинны.
- ANY(MASK [, DIM]) — анализирует наличие истинных элементов в логическом массиве MASK. Возвращает значение .TRUE., если хотя бы один элемент из массива имеет значение .TRUE.
- COUNT(MASK [, DIM]) — производит подсчет числа истинных элементов в логическом массиве MASK. Возвращает стандартное целое, равное числу элементов массива MASK, имеющих значение .TRUE.

- PRODUCT(ARRAY [, DIM] [, MASK]) — функция, вычисляющая произведение всех элементов массива. Для массивов числовых типов возвращает число того же типа и разновидности, что и массив ARRAY, равное произведению всех элементов массива. Третий необязательный аргумент MASK представляет собой логический массив, конформный массиву ARRAY, управляющий перемножением: перемножаются только те элементы массива ARRAY, которым соответствуют истинные элементы массива MASK.
- SUM(ARRAY [,DIM] [, MASK]) — функция, вычисляющая сумму всех элементов массива. Для массивов числовых типов возвращает число тех же типа и разновидности, что и массив ARRAY, равное сумме всех элементов массива. Третий необязательный аргумент MASK представляет собой логический массив, конформный массиву ARRAY, управляющий суммированием: суммируются только те элементы массива ARRAY, которым соответствуют истинные элементы массива MASK.

## Операции с векторами и матрицами

Скалярное произведение двух векторов, перемножение и транспонирование матриц — самые популярные операции векторной алгебры — тоже являются встроенными функциями Фортрана.

- DOT_PRODUCT(VECTOR_A, VECTOR_B) — функция вычисления произведения двух векторов. Аргументы DOT_PRODUCT — одномерные массивы одинаковой длины. Для вещественных аргументов функция возвращает сумму попарных произведений элементов SUM(VECTOR_A * VECTOR_B). Для комплексных аргументов возвращается значение SUM(CONJG(VECTOR_A) * VECTOR_B), для аргументов логического типа — ANY(VECTOR_A .AND. VECTOR_B).
- MATMUL(MATRIX_A, MATRIX_B) — функция, выполняющая перемножение матриц. Как и в алгебраическом перемножении матриц, форма результата зависит от формы исходных матриц. Для аргументов числовых типов возможны три варианта, приведенные ниже. Для аргументов логического типа форма результата будет такой же, а вычисление значений элементов с заданными индексами производится по аналогичным формулам, полученным заменой функции SUM на ANY, и оператора * на оператор .AND.
  - MATRIX_A имеет форму (n, m), MATRIX_B — форму (m, k). Результат имеет форму (n, k), элемент с индексом (i, j) имеет значение SUM(MATRIX_A(i, :) * MATRIX_B(:, j)).
  - MATRIX_A имеет форму (m), MATRIX_B — форму (m, k). Результат имеет форму (k); элемент с индексом (i) имеет значение SUM(MATRIX_A * MATRIX_B(:, i)).
  - MATRIX_A имеет форму (n, m), MATRIX_B — форму (m); результат имеет форму (n), элемент с индексом (i) имеет значение SUM(MATRIX_A(i, :) * MATRIX_B).

- TRANSPOSE(MATRIX) — функция транспонирования матрицы. Возвращает массив того же типа, что и двумерный массив MATRIX. Элемент с индексом (i, j) результирующего массива равен элементу с индексом (j, i) массива-аргумента.

# Текстовые функции

Текстовые функции Фортрана можно разделить на три группы — преобразования "символ-номер", функции лексического сравнения и функции для обработки строк.

- ACHAR(I) — функция поиска символа по номеру в диапазоне таблицы символов ASCII. Возвращает переменную символьного типа стандартной разновидности единичной длины, номер которой в сортирующей последовательности ASCII равен I. Здесь I должно находиться в интервале $0 \le I \le 127$, а результат для других значений I зависит от конкретной реализации. Например, для $128 \le I \le 255$ могут возвращаться символы из таблицы кодировки, соответствующей установленному стилю, а все остальные значения I приводятся к диапазону $0 \le I \le 255$ и возвращают результат для приведенного значения. Так, ACHAR(-1) = ACHAR(255) и ACHAR(256) = ACHAR(0).
- IACHAR(C) — поиск номера символа в сортирующей последовательности ASCII. Возвращает целое значение стандартного типа, соответствующее номеру символа C в сортирующей последовательности ASCII. Результат для аргументов, не содержащихся в таблице ASCII, зависит от процессора.
- CHAR(I [, KIND]) и ICHAR(C) — расширения функций ACHAR и IACHAR для аргументов любой (из поддерживаемых процессором) разновидности символьного типа. Аргумент KIND, если он задан, должен быть целым скалярным инициализирующим выражением, значение которого должно отвечать одному из параметров поддерживаемых процессором разновидностей символьного типа.
- LGE(STRING_A, STRING_B) — функция сравнения строк. Возвращает .TRUE. в том случае, когда STRING_A в сортирующей последовательности следует за строкой STRING_B, или совпадает с ней; .FALSE. возвращается в противном случае.
- LGT(STRING_A, STRING_B) — функция сравнения строк. Возвращает .TRUE. в том случае, когда строка STRING_A в сортирующей последовательности следует за строкой STRING_B; .FALSE. возвращается во всех остальных случаях.
- LLE(STRING_A, STRING_B) — функция сравнения строк. Возвращает .TRUE. в том случае, когда строка STRING_A в сортирующей последова-

тельности предшествует строке STRING_B или совпадает с ней; .FALSE. возвращается в противном случае.

- LLT(STRING_A, STRING_B) — функция сравнения строк. Возвращает .TRUE. в том случае, когда строка STRING_A в сортирующей последовательности предшествует строке STRING_B; .FALSE. возвращается во всех остальных случаях.
- ADJUSTL(STRING) — элементная функция преобразования строки. Возвращает строку, длина которой совпадает с длиной строки STRING, но все находящиеся слева пробелы смещены в конец строки.
- ADJUSTR(STRING) — элементная функция преобразования строки. Возвращает строку, длина которой совпадает с длиной строки STRING, но все находящиеся справа пробелы смещены в начало строки.
- INDEX(STRING, SUBSTRING [, BACK]) — элементная функция поиска подстроки в строке. Возвращает стандартное целое, равное номеру позиции первого символа подстроки SUBSTRING в строке STRING. Если STRING не содержит SUBSTRING, то возвращается нуль. Если SUBSTRING имеет нулевую длину, то возвращается значение, равное LEN(STRING) + 1. Логический параметр BACK задает направление поиска: если он не задан или задан со значением .FALSE., возвращается номер позиции первого от начала STRING вхождения SUBSTRING. В противном случае, поиск производится с конца строки STRING.
- LEN_TRIM(STRING) — элементная функция вычисления длины "усеченной" строки. Возвращает стандартное целое, равное длине строки STRING за вычетом начальных и конечных пробелов.
- REPEAT(STRING, NCOPIES) — не элементная функция расширения строки. Возвращает строку, полученную последовательной конкатенацией NCOPIES строк STRING. Функция не может быть элементной, так как размерность ее результата зависит от значения аргумента.
- SCAN(STRING, SET [, BACK]) — элементная функция поиска в строке элементов из заданного набора символов. Возвращает стандартное целое, равное номеру позиции символа из набора SET в строке STRING, или нуль, если ни один из элементов набора SET в строке STRING не найден. Логический параметр BACK задает направление поиска: если он не задан или задан со значением .FALSE., возвращается номер позиции первого от начала STRING вхождения одного из элементов SET. В противном случае, поиск производится с конца строки STRING.
- TRIM(STRING) — не элементная функция "усечения" строки. Возвращает строку, полученную из строки STRING удалением конечных пробелов. Не может быть элементной из-за зависимости длины результата от значения аргумента.

- VERIFY(STRING, SET [, BACK]) — элементная функция проверки наличия вхождений элементов строки STRING в символьный набор SET. Возвращает стандартное целое, значение которого 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 крайнего правого из символов STRING, не входящих в SET, во всех остальных случаях — крайнего левого.

# Процедуры для работы с битами

Набор битов — элементов, занимающих один разряд и принимающих значения 0 или 1 — можно рассматривать как целое число $I$, представленное в виде суммы

$$I = \sum_{k=0}^{s-1} b_k \times 2^k ,$$

где $b_k$ могут принимать значения 0 или 1. 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 ниже 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) — справочная функция: возвращает стандартное целое, равное числу битов, необходимых для представления целого числа с таким же параметром разновидности, как у аргумента I.
- BTEST(I, POS) — элементная функция проверки бита. Если бит аргумента I, занимающий позицию POS, имеет значение 1, возвращается .TRUE., в противном случае — .FALSE.
- IAND(I, J) — элементная функция логического побитового умножения (логическое "И"), причем оба аргумента должны быть одной разновидности целого типа. Возвращает значение, каждый бит которого получен по следующим правилам:

```
        I = 1 1 0 0
        J = 1 0 1 0
IAND(I, J) = 1 0 0 0
```

- IBCLR(I, POS) — элементная функция обнуления указанного бита. Возвращает целое значение той же разновидности, что и I, с нулевым битом в позиции POS.
- IBITS(I, POS, LEN) — элементная функция выборки со сдвигом вправо. Возвращает целое значение той же разновидности, что и I, полученное из LEN битов аргумента, начиная с позиции POS, сдвинутых вправо с обнулением всех оставшихся битов.
- IBSET(I, POS) — элементная функция установки заданного бита. Возвращает целое значение той же разновидности, что и I, полученное назначением 1 для бита в позиции POS.
- IEOR(I, J) — элементная функция логического побитового "исключающего ИЛИ", причем оба аргумента должны быть одной разновидности целого типа. Возвращает значение, каждый бит которого получен по следующим правилам:

```
          I = 1 1 0 0
          J = 1 0 1 0
IEOR(I, J) = 0 1 1 0
```

- IOR(I, J) — элементная функция логического побитового "ИЛИ", причем оба аргумента должны быть одной разновидности целого типа. Возвращает значение, каждый бит которого получен по следующим правилам:

```
          I = 1 1 0 0
          J = 1 0 1 0
 IOR(I, J) = 1 1 1 0
```

- ISHIFT(I, SHIFT) — элементная функция вытесняющего сдвига. Возвращает целое значение той же разновидности, что и I, 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]) — элементная функция циклического сдвига. Возвращает целое значение той же разновидности, что и I, полученное циклическим сдвигом на SHIFT позиций. Если аргумент SIZE присутствует, 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. Возвращает значение, каждый бит которого получен как дополнение до 1:

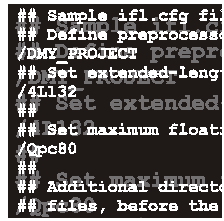
```
     I = 1 0
NOT(I) = 0 1
```

- CALL 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 той же разновидности типа, что и FROM. Входные аргументы должны удовлетворять требованиям сохранения границ разрядной сетки: LEN $\geq 0$, FROM $\geq 0$, FROMPOS $\geq 0$, TOPOS $\geq 0$, FROMPOS + LEN $\leq$ BIT_SIZE(FROM) и TOPOS + LEN $\leq$ BIT_SIZE(TO).

# Вопросы и задания для самостоятельной работы

1. С помощью встроенных функций определите параметры моделей представления целых и вещественных чисел.
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.
3. В матрице, заполненной случайными числами, смените на минус знак элементов, попадающих в интервал [0.3, 0.7]. Вычислите, не используя оператор цикла, число положительных элементов и упакуйте их в одномерный массив.




# Глава 11

# Компиляция, выполнение и отладка программ на Фортране

Свидетельством жизнеспособности Фортрана и его устойчивой популярности является большое и все возрастающее число компиляторов, а также инструментальных средств разработки программ. Ссылки на Web-сайты некоторых фирм, выпускающих компиляторы и среды программирования на Фортране, приведены в *Приложении 4*.

Имеются как коммерческие, так и свободно распространяемые пакеты. Свободно распространяется для некоммерческого использования, например, компилятор Фортран 95 фирмы Intel. Предлагаются также компиляторы, работающие с подмножествами языка. К их числу относится система F (The Fortran Company), до недавнего времени бесплатно распространялась и система ELF90.

Часто используются и средства разработки, которые можно отнести к числу вспомогательных. Это конвертеры (трансляторы) с Фортрана на другие языки, например, C, а также различные системы отладки программ, поддерживающие работу с Фортраном. Среди них следует упомянуть такие программы, как dbx, gdb, ElectricFence и другие.

В этой главе дается обзор некоторых наиболее известных средств разработки программ на языке Фортран. Кратко рассмотрев основные этапы генерации исполняемого файла программы, мы перейдем к знакомству с трансляторами и некоторыми средствами отладки. Количество компиляторов и систем программирования на Фортране велико и существенно превышает количество компиляторов, например, для такого популярного языка, как Паскаль. В одной книге невозможно дать полное описание даже одного компилятора. С учетом этого в данной главе дается лишь очень краткий обзор таких компиляторов, как Intel Fortran, Lahey Fortran, Compaq Visual Fortran и GNU Fortran.

# Основные этапы создания исполняемого файла программы

Создание исполняемого файла программы, вообще говоря, достаточно сложный и многоступенчатый процесс, в котором можно выделить следующие этапы.

1. Создание файла с исходным текстом программы с помощью текстового редактора.
2. Необязательный шаг обработки исходного текста программы препроцессором.
3. *Компиляция* — создание объектного файла (или нескольких объектных файлов, если программа достаточно сложна и ее исходный текст содержится в нескольких файлах). Объектный файл содержит исполняемый код программы на машинном языке (или подпрограммы), который еще не готов к выполнению. В него не включены код запуска программы (start-up code) и коды библиотечных подпрограмм. Для разных частей программы может быть подготовлен свой собственный объектный файл.
4. Создание исполняемого файла с помощью программы-компоновщика, которая собирает исполняемый файл из объектных файлов пользовательской программы и необходимых библиотек, а также включает в него код запуска. *Компоновку* (сборку) выделяют в отдельный процесс, потому что его можно выполнить независимо от собственно компиляции. Компиляторы обычно выполняют компоновку по умолчанию, и миновать этот шаг можно только с помощью специального *ключа компиляции*. Напомним, что ключи позволяют включить определенный режим выполнения программы. При запуске программы из командной строки ключи являются частью команды запуска.

Во время компиляции выполняется проверка синтаксиса программы и, если обнаружены ошибки, выдается диагностическое 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.

Кроме сообщений об ошибках, компилятор может выводить *предупреждения*. Предупреждения выводятся в том случае, когда исходный текст программы не содержит синтаксических ошибок, но какие-то конструкции вызывают у компилятора "подозрение" в правильности их использования. Приведем пример:

```
PROGRAM DOLOO
DO I = 1, K
```

```
J = J + 1
END DO
END PROGRAM DOLOO
```

При компиляции Compaq Visual Fortran выведет предупреждение:

```
Warning: Variable K is used before its value has been defined
DO I = 1, K
----------^
```

Иногда выводятся *информационные сообщения*, содержащие 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 1, 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. Исходный текст обычно содержится в файле, имя которого имеет суффикс **f90** или **f95**. Иногда совместно с программными блоками, написанными на Фортране, используются части кода, написанные на языке C. Исходные тексты хранятся в этом случае в файлах с расширением **c**. Файлы модулей имеют расширение **mod**. Объектный файл имеет расширение **o**. Статические библиотеки могут содержаться в файлах с расширением **lib** (MS Windows) или **a** (UNIX), а динамические — соответственно в файлах с расширениями **dll** или **so**. Имя исполняемого файла в MS Windows имеет суффикс **exe**, а в операционных системах (ОС) UNIX оно может быть любым (по умолчанию a.out). Иногда используются также фрагменты на языке ассемблера, им соответствуют файлы с суффиксом **asm**. Файлы с расширением **map** могут содержать информацию, используемую при отладке программы.

# Язык и система программирования F

Система программирования F, распространяемая The Fortran Company и NAG (является их торговой маркой), реализует подмножество языка Фортран 95 и содержит инструменты для разработки программ на этом языке. Подмножество это выбрано с учетом особенностей преподавания языка Фортран. С этим обстоятельством связаны некоторые особенности языка. В нем, например, отсутствует оператор эквивалентности `EQUIVALENCE`, изучение которого вызывает трудности и который является источником серьезных ошибок программирования. Удаление потенциально опасных конструкций Фортрана привело к тому, что язык F оказался достаточно "безопасным" и универсальным. Он может быть выбран в качестве базы для перехода с языка Фортран 77 на Фортран 95 или HPF.

Компиляторы F имеются для ОС Intel/Linux, Windows 95/98/NT/ME/XP и Solaris. В язык F, по сравнению с Фортраном 77, включены и некоторые дополнительные возможности. Среди них: доступ к аргументам командной строки и переменным окружения, возможность выполнения команд операционной системы непосредственно из программы, наличие элементов HPF, возможность использовать динамические массивы в качестве компонента производного типа. В F поддерживается модульное программирование.

В языке F отсутствуют: неявное объявление типа, конструкция `BLOCK DATA`, метки (есть, правда, исключение), `NAMELIST`-ввод-вывод, оператор `include`, оператор-функция, внешние процедуры и некоторые другие особенности Фортрана.

Отметим еще ряд особенностей языка F:

- все атрибуты переменных задаются только в предложениях описания;
- все значения параметра разновидности должны задаваться с помощью параметров;
- производные типы могут определяться только в модуле;
- управляющие переменные циклов `DO` и `FORALL` должны быть локальными;
- процедуры должны находиться в модуле или быть внутренними по отношению к главной программе;
- атрибут доступа для всех переменных, типов и процедур, описанных в модуле, должен быть указан явно.

Вызывается компилятор F следующим образом:

```
F [ключи] [список_файлов]
```

Он транслирует программы, написанные на языке F, в исполняемые файлы, модули, ассемблерный код или код C.

Действие, выполняемое компилятором F, определяется расширением (суффиксом) имени файла. Файлы, имена которых имеют расширение **f90** и **f95** считаются исходными текстами F-программ. Расширения **F90** и **F95** означают, что исходный файл должен быть обработан препроцессором fpp (Fortran PreProcessor — препроцессор Фортрана). Список файлов может содержать имена файлов любого вида. Если имя файла не имеет суффикса и файл с таким именем не обнаружен, компилятор F ищет файл с этим именем и с расширением **f95**, а если не находит и его, то файл с расширением **f90**.

Модули и включаемые файлы должны находиться в текущем каталоге или в каталоге, указанном после ключа `-I`. Те ключи, которые не распознаются компилятором F, передаются компоновщику.

В табл. 11.1 приведены описания некоторых ключей компилятора F. С полным списком можно ознакомиться в документации по системе. Заметим также, что не все из нижеперечисленных ключей работают на всех платформах.

***Таблица 11.1.*** *Некоторые ключи компилятора F*

| Ключ | Описание |
|---|---|
| `-c` | Завершить компиляцию после создания объектного файла |
| `-C` | Компилировать программу со всеми проверками времени выполнения |
| `-C = check` | Компилировать в соответствии со значением `check`:<br>`all` — выполнять все проверки;<br>`array` — проверять выход индекса за установленные границы;<br>`calls` — проверять ссылки на процедуры;<br>`do` — проверять циклы `DO` на нулевые значения шага;<br>`none` — не выполнять никаких проверок, действует по умолчанию;<br>`present` — проверка ссылок `OPTIONAL`;<br>`pointer` — проверка ссылок `POINTER` |
| `-dryrun` | Показывать, но не выполнять команды компилятора |
| `-F` | Выполнить обработку препроцессором файлов, имена которых имеют расширения F90 и F95. В результате обработки создаются файлы f90 и f95. Компиляция не выполняется |
| `-f77` | Обеспечить совместимость операций компоновки с "родным" транслятором f77 |
| `-g` | Генерация информации для интерактивной отладки отладчиком. Оптимизация должна быть отключена (ключ `-O0`) |
| `-g90` | Компиляция для отладчика dbx90, который совместим с Фортраном 90 и работает совместно с dbx. В результате такой компиляции для каждого файла с исходным текстом в файл с расширением g90 записывается отладочная информация |
| `-gc` | Включить автоматический "сбор мусора" выполняемой программой. Garbage collector или "сборщик мусора" — это программа чистки памяти, часть системы динамического распределения памяти |
| `-gline` | Компиляция с генерацией номеров строк, которые выводятся в сообщениях об ошибках времени выполнения (не всех). Размер исполняемого файла при этом увеличивается, а скорость выполнения программы уменьшается |
| `-hpf` | Позволяет использовать расширения Фортрана HPF |
| `-I маршрут` | Добавляет маршрут к списку каталогов, в которых выполняется поиск файлов с информацией о модулях (mod) и включаемых файлов. В первую очередь поиск выполняется в текущем каталоге, затем продолжается в каталогах, указанных после ключей `-I`. В последнюю очередь поиск выполняется в библиотечном каталоге компилятора, обычно это **/usr/local/lib/F** или **/opt/F/lib** |

***Таблица 11.1*** *(продолжение)*

| Ключ | Описание |
|---|---|
| `-ieee = режим` | Режим выполнения арифметических операций IEEE:<br>`full` — включает все возможности расширенной арифметики IEEE;<br>`nonstd` — запрещает постепенную потерю точности, приводя к нулевому значению вместо денормализованного числа. Это приводит к увеличению скорости выполнения программы. Расширенная арифметика также блокирована;<br>`stop` — включает все возможности арифметики IEEE кроме расширенной (non-stop), выполнение программы будет прервано при переполнении, делении на ноль или некорректном операнде.<br>При работе на процессоре Alpha под управлением ОС Digital UNIX (или Tru64 UNIX), ключ `-ieee` следует указывать при компиляции всех программных модулей. На других платформах и для других ОС ключ `-ieee` следует использовать при компиляции главной программы, его действие глобально. По умолчанию используется `-ieee = stop` |
| `-info` | Включает вывод информационных сообщений. Этот вывод по умолчанию отсутствует |
| `-kind = option` | Задает систему нумерации параметров разновидности. Возможные значения `option`:<br>`byte` — значения параметра разновидности для `INTEGER`, `REAL` и `LOGICAL` соответствует количеству байтов, отводимому для хранения элемента данных соответствующего типа (например, по умолчанию типу `REAL` соответствует значение 4);<br>`sequential` — (используется по умолчанию), параметры разновидности пронумерованы последовательно, начиная с 1, увеличиваясь с возрастанием точности (стандартному вещественному типу соответствует 1, а вещественному типу с двойной точностью — 2).<br>Этот ключ не влияет на интерпретацию спецификаторов длины типа в байтах (расширение Фортрана 77) |
| `-lx` | Использовать библиотеку `libx.a`. Поиск этой библиотеки выполняется в каталогах, заданных ключами `-L`*каталог* |
| `-L`*каталог* | Добавить указанный каталог к списку каталогов с библиотечными файлами |
| `-M` | Завершить компиляцию после создания файлов модулей (mod) |
| `-o файл` | Имя файла с результатом компиляции (вместо a.out). Данный ключ можно использовать и совместно с ключами `-c` и `-S` |
| `-On` | Различные уровни оптимизации — от ее отсутствия `-O0` и до максимального уровня `-O4` |

***Таблица 11.1*** *(продолжение)*

| Ключ | Описание |
|---|---|
| `-Ounroll = n` | Глубина развертки циклов |
| `-Ounsafe` | Выполнять потенциально опасную оптимизацию, которая может привести к численной неустойчивости программы |
| `-pg` | Скомпилировать программу с генерацией профилирующей информации. Эта информация во время выполнения программы записывается в файл, формат которого зависит от реализации, а имя обычно — gmon.out или mon.out. Профиль затем может быть построен с помощью утилиты gprof (на платформах Sun и IBM RISC) или prof (на рабочих станциях DEC и SGI). В некоторых реализациях этот ключ не работает |
| `-Qpath-путь` | Указать маршрутное имя библиотеки компилятора f95 (по умолчанию обычно **/usr/local/lib/F** или **/opt/F/lib**) |
| `-r8` | Удвоить размер типа `REAL/COMPLEX` по умолчанию, если на машине поддерживаются типы `REAL/COMPLEX` с учетверенной точностью, то удваивается размер типа `DOUBLE PRECISION`. Если переменные типов `REAL/COMPLEX` описаны с явным указанием значения `KIND`, длина в байтах не меняется, но поскольку внутренняя функция `KIND` возвращает правильные значения, `COMPLEX(KIND(0d0))` на машине с плавающей точкой учетверенной точности правильно выберет комплексный тип с учетверенной точностью. Не влияет на размер типа `INTEGER` |
| `-s` | Удалить таблицу имен (символов) из исполняемого файла. Этот ключ передается ld и действует только на шаге компоновки |
| `-S` | Выдать в качестве результата исходный текст на языке C. Результирующий файл с расширением **.c** следует компилировать с помощью команды f95, а не компилятором C |
| `-save` | Компиляция с этим ключом равносильна подстановке оператора `SAVE` во все подпрограммы, которые не описаны как `RECURSIVE`, что приведет к статическому размещению всех локальных переменных в таких подпрограммах |
| `-target=machine` | Задает аппаратную платформу (это, в основном, процессоры с архитектурой SPARC и ОС Solaris фирмы Sun Microsystems), для которой следует выполнить компиляцию и оптимизацию кода:<br>`V7` — SPARCstation 1 (по умолчанию);<br>`V8` — SPARCstation 2;<br>`super` — SuperSPARC;<br>`ultra` — UltraSPARC;<br>`native` — данная машина.<br>Последний флаг не работает при совместном использовании с компилятором gcc |

***Таблица 11.1*** *(окончание)*

| Ключ | Описание |
|---|---|
| `-tempdir-каталог` | Каталог размещения временных файлов. По умолчанию используется имя каталога, указанное в переменной окружения `TMPDIR`, а если она не задана, то каталог **/tmp** |
| `-time` | Вывод времени выполнения этапов компиляции |
| `-v` | Вывод имени каждого файла при его компиляции |
| `-V` | Вывести версию компилятора |
| `-xs` | Запись таблицы символов для использования отладчиком dbx прямо в исполняемый файл (только для платформы Sun/SPARC) |

Ограничения компилятора приведены в табл. 11.2.

***Таблица 11.2.*** *Ограничения компилятора F*

| Элемент | Максимальное количество |
|---|---|
| Вложенность включаемых файлов | 20 |
| Количество операторов `INCLUDE` для одного сеанса компиляции | 2047 |
| Вложенность циклов `DO` | 199 |
| Вложенность конструкции `CASE` | 30 |
| Вложенность неявных циклов в операторе `DATA` | 20 |
| Вложенность неявных циклов в конструкторе массива | 20 |
| Количество аргументов подпрограммы | 32767 |
| Количество аргументов в функциях `MIN` и `MAX` | 20 |

В операциях ввода-вывода длины записей по умолчанию: форматный ввод-вывод — 1024 символа, неформатный — 65 536 байт.

При создании модульных программ следует учесть следующее. Для того чтобы использовать модуль, его следует сначала откомпилировать, в результате чего создается файл с расширением **mod** и, обычно, объектный файл с расширением **o**. Файлы **mod** используются компилятором во время компиляции для того, чтобы сохранить необходимую информацию о содержимом модуля, если же создается файл **o**, он содержит код всех процедур модуля и должен быть указан при создании исполняемого файла. Имя **mod**-файла совпадает с именем модуля, а имя объектного файла такое же, как у исходного файла. Если имя модуля указано в предложении `USE`, компилятор F пытается найти соответствующий ему исходный файл и, если это удается,

проверяет время и дату модификации, выводя предупреждающее сообщение, если файл **mod** устарел.

# Компилятор фирмы Intel

Компилятор Фортрана фирмы Intel (Intel Fortran Compiler) версии 7.0, который в дальнейшем для краткости мы будем обозначать аббревиатурой IFC7.0, предназначен для архитектуры Intel IA-32 и Intel IA-64 (Itanium) и операционных систем Microsoft Windows 98/NT/2000. Имеется также версия компилятора для ОС Linux.

Среди особенностей компилятора IFC7.0 отметим поддержку компиляции для архитектуры Intel Itanium, поддержку спецификации OpenMP (MP, MultiProcessing, открытый стандарт многопроцессорной обработки), совместимость с Microsoft Visual Studio.Net (VS.NET), режим совместимости с Compaq Visual Fortran и поддержку технологии Hyper-Threading. В технологии Hyper-Threading один физический процессор представляется как несколько логических, что позволяет многопоточным приложениям использовать ресурсы компьютера более эффективно. Поддержка спецификации OpenMP позволяет создавать код для параллельного выполнения на системах с разделяемой памятью.

IFC7.0 можно использовать совместно с интегрированной средой Microsoft Visual C++, а также в среде разработки VS.NET для создания приложений на Фортране, включая статические (lib) и динамические (dll) библиотеки, а также исполняемые программы. В случае использования VS.NET создание приложений возможно только для платформы IA-32 Windows.

Компилятор IFC7.0 поддерживает расширения Streaming SIMD Extensions и Streaming SIMD Extensions 2 (сокращенно SSE и SSE 2 — потоковые расширения набора команд процессора, обеспечивающие расширенные возможности параллельной обработки мультимедийной информации, то есть изображений, трехмерной графики, потокового видео и аудио. Это дальнейшее развитие расширения MMX — MultiMedia Extension). Имеется также частичная совместимость с Compaq Visual Fortran. В дальнейшем мы ограничимся описанием возможностей компилятора IFC7.0 для платформы IA-32.

Основные компоненты комплекта поставки перечислены ниже.

- Компилятор. Компилятор IFC7.0 имеет три варианта: 32-разрядный компилятор, кросс-компилятор, работающий на платформе IA-32 и выполняющий компиляцию для платформы Itanium, и компилятор для платформы Itanium.
- Инструмент визуализации массивов (Intel Array Visualizer), который позволяет графически отобразить одно-, двух- и трехмерные массивы.
- Расширенный отладчик (Intel Enhanced debugger) с полной поддержкой Фортрана 95 и многопоточности.

С IFC7.0 используются также некоторые библиотеки и инструментальные средства.

- Ассемблер IA-32 — по умолчанию masm (Microsoft Macro Assembler), с IFC7.0 не поставляется.
- Ассемблер Itanium — по умолчанию Ассемблер Intel Itanium, поставляется вместе с компилятором.
- Компоновщик — по умолчанию компоновщик Microsoft linker, с IFC7.0 не поставляется.

По умолчанию, компилятор создает объектный файл, не вызывая ассемблер, поэтому ассемблер нужен, только если необходимо использовать ассемблерные части кода.

На платформе IA-32 можно использовать любой 32-разрядный ассемблер. В состав Microsoft Visual C++ и IFC7.0 не входит ассемблер, однако можно использовать masm версии 6.15 или выше.

Компилятор IFC7.0 предназначен для компьютеров с процессором Intel. Минимальные требования к аппаратной конфигурации включают наличие процессора Intel Pentium, Intel Xeon или другого процессора IA-32, оперативную память от 128 Мбайт. Рекомендуется процессор Pentium 4 или Xeon и 256 Мбайт оперативной памяти.

Кросс-компилятору для платформы IA-32 требуется процессор Pentium II или выше с тактовой частотой от 350 МГц, 256 Мбайт оперативной памяти. Он выполняет компиляцию для платформы Itanium.

Для компилятора Itanium требуется наличие процессора Itanium, от 512 Мбайт оперативной памяти (рекомендуется 1 Гбайт).

IFC7.0 может подключаться в качестве модуля к продукту Microsoft Visual C++ Professional Edition и использовать некоторые компоненты этой среды (заголовочные файлы, компоновщик и т. д.).

Объектные файлы могут применяться совместно с процедурами, написанными на языке ассемблера (рекомендуется использовать masm, версия 6.15 и выше).

В качестве компоновщика используется Microsoft Portable Executable Linker, который, в свою очередь, применяет библиотеки Microsoft Visual C++.

IFC7.0 представляет собой коммерческий продукт, для работы с которым необходимо приобрести лицензию.

## Этапы компиляции

Исполняемый файл создается в два этапа. Сначала выполняется компиляция, а затем редактирование связей (компоновка). Компилятор определяет фазу компиляции по суффиксу имени файла и по ключам командной строки. Различают 4 фазы компиляции.

1. *Обработка препроцессором* — это необязательная фаза, которая выполняется программой fpp.

2. *Компиляция* — выполняется программой f90com. Компилятор передает объектные файлы и файлы, которые ему не удалось распознать, компоновщику, который определяет, имеет он дело с объектным файлом (obj) или файлом библиотеки (lib). Компилятор распознает все типы входных файлов и может инициировать любую фазу компиляции.
3. *Сборка* — выполняется в версии для процессора Itanium программой ias.
4. *Редактирование связей, компоновка* — выполняется программой link.

## Ключи компилятора

Алфавитный список некоторых (но не всех) ключей компилятора приведен в табл. 11.3 (общие для платформ IA-32 и Itanium) и табл. 11.4 (только для платформы IA-32). В квадратных скобках приведены необязательные части ключей. Конструкция вида `/4I{2|4|8}` означает, что у данного ключа есть три варианта: `/4I2`, `/4I4` и `/4I8`, а конструкция вида `/W{n}`, что `n` — это одно из допустимых значений (они приводятся в столбце "Описание").

***Таблица 11.3.*** *Некоторые ключи компилятора IFC7.0*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| `/?`<br>`/help` | Вывод справочной информации | `OFF` |
| `/1`<br>`/Qonetrip` | Циклы `DO` выполняются по крайней мере один раз | `OFF` |
| `/4I{2\|4\|8}` | Параметр разновидности `KIND` для целых переменных и констант размером 2, 4 и 8 байт | `/4I4` |
| `/4L{72\|80\|132}` | Длина строки для фиксированного формата записи исходного текста | `/4L72` |
| `/4R{4\|8\|16}` | Параметр разновидности `KIND` для вещественных переменных размером 4, 8 и 16 байт.<br>Ключ `/4R8` устанавливает размер и точность стандартного вещественного типа (то есть типа, используемого по умолчанию) такими же, что и у типа с двойной точностью.<br>Ключ `/4R16` устанавливает размер и точность стандартного вещественного типа такими же, что и у типа `REAL (KIND = 16)` | `/4R4` |
| `/4{Y\|N}a` | Считать (или не считать) все переменные автоматическими | `ON` |

***Таблица 11.3*** *(продолжение)*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| /4{Y\|N}b | Включить (или отключить) расширенный контроль ошибок времени выполнения | OFF |
| /4{Y\|N}f | Определяет фиксированный формат исходного текста для следующих типов файлов:<br>/4Yf: .f, .for, .ftn;<br>/4Nf: .f90 | OFF |
| /c | Завершить процесс компиляции после создания объектных файлов (obj) | OFF |
| /cm | Отменить вывод сообщений с комментариями | OFF |
| /E | Выполнить обработку препроцессором, а ее результат направить в стандартный вывод | OFF |
| /EP | Выполнить обработку препроцессором и вывести ее результат в стандартный вывод, опуская директивы #line | OFF |
| /Fa[*файл*] | Создать файл с текстом программы на ассемблере | OFF |
| /Ge | Включить проверку стека для всех функций | OFF |
| /G{1\|2\|5\|6\|7} | Оптимизация кода для процессора:<br>/G1 — Itanium;<br>/G2 — Itanium 2;<br>/G5 — Pentium;<br>/G6 — Pentium Pro, Pentium II, Pentium III;<br>/G7 — Pentium 4, Xeon | /G2 для Itanium;<br>/G7 для IA-32 |
| /I*каталог* | Дополнительный каталог для поиска включаемых файлов, имена которых не начинаются с наклонной черты (/) | OFF |
| /LD | Результатом компиляции будет динамическая библиотека (dll) | OFF |
| /link | Ключи, указанные после данного ключа, передаются компоновщику | OFF |
| /list | Вывести исходный текст (без включаемых файлов), а также сообщения об ошибках и предупреждения в стандартный вывод | OFF |

***Таблица 11.3*** *(продолжение)*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| `/list /show:include` | Вывести исходный текст с подстановкой включаемых файлов в стандартный вывод | `OFF` |
| `/ML` | Компилировать и компоновать со статической, однопоточной, отладочной версией библиотеки времени выполнения | `ON` |
| `/module:путь`<br>`/nomodule` | Указывает каталог, в котором находятся файлы модулей (mod). Ключ `/nomodule` соответствует размещению модулей в каталоге, содержащем компилируемые файлы | `/nomodule` |
| `/O1` | Включает оптимизацию по производительности, но отключает некоторые виды оптимизации, которые увеличивают размер кода с небольшим увеличением производительности:<br>`/Og`, `/Oi`, `/Os`, `/Oy`, `/Ob1`, `/Gs`<br>Более подробную информацию по данной теме можно найти в Руководстве пользователя, которое включено в комплект поставки | `OFF` |
| `/O2` | Оптимизация по производительности | `ON` |
| `/O3` | Более "агрессивный" вариант оптимизации, чем с ключом `/O2`. Включает преобразования циклов, но существенный выигрыш дает не всегда | `OFF` |
| `/Od` | Запрещает оптимизацию. Рекомендуется использовать при избирательной оптимизации (например, для комбинации ключей `/Od` и `/Og`) | `OFF` |
| `/Og` | Включает глобальную оптимизацию | `ON` |
| `/Op[-]` | Включить (или отключить) более высокую точность выполнения операций с плавающей точкой, а также совместимость со стандартом IEEE 754 | `OFF` |
| `/Ot` | Включает все виды оптимизации по производительности | `ON` |
| `/P` | Обработка файлов препроцессором fpp | `OFF` |
| `/Qauto_scalar` | Делает все скалярные переменные автоматическими | `ON` |

***Таблица 11.3*** *(продолжение)*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| /Qd_lines | Компиляция отладочных операторов в строках, содержащих букву D в первой позиции строки исходного текста программы | OFF |
| /Qfpp{n} | Включает обработку исходного текста программы препроцессором fpp со следующими особенностями:<br>n = 0 — игнорировать директивы компиляции Compaq Visual Fortran (см. Руководство по Compaq Visual Fortran) и #;<br>n = 1 — обрабатывать директивы условной компиляции CVF и директивы #, используется по умолчанию, если обработка препроцессором выполняется;<br>n = 2 — обрабатывать только директивы #;<br>n = 3 — обрабатывать только директивы условной компиляции CVF | OFF |
| /Qlocation, инструмент, путь | Указывает альтернативный инструментарий и его расположение на диске. Здесь инструмент может быть следующим:<br>fpp — препроцессор Intel Fortran (fpp);<br>f — компилятор Фортрана (f90com);<br>asm — ассемблер (masm для IA-32 и ias для Itanium);<br>link — компоновщик.<br>Параметр путь задает положение компонента | OFF |
| /Qlowercase | Устанавливает нижний регистр символов в именах для внешнего компоновщика | OFF |
| /Qopenmp | Включает генерацию многопоточного кода, основанного на директивах OpenMP. С этим ключом должны использоваться ключи /MT и /Qfpp | OFF |
| /Qoption, инструмент, ключи | Передать ключи указанному инструменту | OFF |

***Таблица 11.3*** *(продолжение)*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| /Qparallel | Определяет циклы, которые можно безопасно распараллелить, и генерирует для них многопоточный код | OFF |
| /Qprec | Повышение точности операций с плавающей точкой. Скорость выполнения программы уменьшается | OFF |
| /Qprof_dir*каталог* | Каталог для хранения файлов с профилирующей информацией *.dyn и *.dpi | OFF |
| /Qprof_gen | Генерируется код для профилирования программы | OFF |
| /Qprof_file*файл* | Имя файла с информацией профилировщика | OFF |
| /Qsave | Все переменные считаются статическими | OFF |
| /Quppercase | Устанавливает верхний регистр символов в именах для внешнего компоновщика | ON |
| /Qzero | Неявно инициализирует нулевыми значениями все неинициализированные явным образом переменные. Используется совместно с ключом /Qsave | OFF |
| /S | Компиляция завершается созданием файлов с текстом программы на ассемблере (asm) | OFF |
| /us | Добавляет к именам внешних подпрограмм символ подчеркивания | OFF |
| /V*строка* | Вставляет заданную строку, например, с версией программы, в исполняемый код | OFF |
| /W{n} | n = 0 — отменяет вывод всех предупреждений.<br>n = 1 — включает вывод всех предупреждений | /W1 |
| /Zd | Отладочная информация в объектном файле содержит только номера строк | OFF |
| /Zi, /Z7 | В объектный код добавляются таблица символов и номера строк, что позволяет применять отладчики на уровне исходного текста | OFF |

***Таблица 11.3*** *(окончание)*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| /Zp{1\|2\|4\|8\|16} | Выравнивание структур на 1-, 2-, 4-, 8- и 16-байтную границу. Меньшее значение приводит к меньшему размеру секции данных, а большее значение увеличивает скорость работы программы. Ключ /Zp16 позволяет выполнить выравнивание и для common-блоков | IA-32: /Zp4 |
| /Zs | Выполняется только синтаксический анализ | OFF |

***Таблица 11.4.*** *Некоторые ключи компилятора IFC7.0 только для платформы IA-32*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| /CA | Генерируется код с проверкой корректности использования ссылок и динамических массивов (во время выполнения). Используется вместе с ключом /d{n} | OFF |
| /CB | Генерируется код, в котором выполняется проверка индексов массивов и ссылок на подстроки на выход за заданные границы. Используется вместе с ключом /d{n} | OFF |
| /CU | Генерируется код, выполнение которого с неинициализированными переменными приводит к ошибке. Используется вместе с ключом /d{n} | OFF |
| /CV | При входе в программу проверяется соответствие между фактическими и формальными параметрами. Вызывающий и вызываемый код должны быть скомпилированы с ключом /CV. Используется вместе с ключом /d{n} | OFF |
| /d{0\|1\|2\|>2} | Установка уровня диагностики:<br>/d0 — вывод заголовка процедуры;<br>/d1 — вывод локальных переменных;<br>/d2 — вывод локальных переменных и переменных из COMMON-блоков, при /d>2 вывод первых n элементов массивов и всех простых переменных | /d0 |

***Таблица 11.4*** *(продолжение)*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| /Qax{i\|M\|K\|W} | Генерирует код, использующий в дополнение к обычным для платформы IA-32, специфические для процессора команды:<br>I — Pentium Pro, Pentium II;<br>M — Pentium с технологией MMX;<br>K — Pentium III;<br>W — Pentium 4 и Xeon.<br>Так например, при работе на процессоре Pentium III, если основная часть операций выполняется с целыми числами, можно выбрать ключ /QaxM, потому что расширение MMX обеспечивает высокую производительность при работе именно с целыми значениями | OFF |
| /Qpc{32\|64\|80} | Управление точностью представления с плавающей точкой:<br>/Qpc32 — для 24-разрядной мантиссы;<br>/Qpc64 — для 53-разрядной мантиссы;<br>/Qpc80 — для 64-разрядной мантиссы | /Qpc64 |
| /Qprec_div | Запрещает оптимизацию посредством замены операций деления умножением, что повышает точность выполнения операции деления. Немного уменьшает производительность программы | OFF |
| /Qrcd | Увеличивает эффективность преобразования вещественных значений в целые значения | OFF |
| /Qsox[-] | Включает или отключает запись версии компилятора и его ключей в исполняемый код. Доступ к этой записи можно получить с помощью утилиты strings | /Qsox- |
| /Quse_vcdebug | Генерируется отладочная информация, которую может использовать отладчик Visual C++ | ON |
| /Qvec_report{n} | Уровень диагностики при векторизации кода:<br>n = 0 — диагностические сообщения не выводятся;<br>n = 1 — вывод сообщений о векторизованных циклах | /Qvec_report1 |

***Таблица 11.4*** *(окончание)*

| Ключ | Описание | Значение по умолчанию |
|---|---|---|
| /Qvec_report{n} | Уровень диагностики при векторизации кода:<br>n = 2 — то же, что n = 1 и невекторизуемые циклы;<br>n = 3 — то же, что n = 1 и информация о зависимостях;<br>n = 4 — вывод сообщений о невекторизованных циклах;<br>n = 5 — вывод сообщений о невекторизованных циклах с объяснением причины невозможности векторизовать | |

Компилятор IFC7.0 интерпретирует тип входного файла по суффиксу (расширению) его имени. Возможные варианты приведены в табл. 11.5.

***Таблица 11.5.*** *Интерпретация файлов компилятором IFC7.0*

| Суффикс | Интерпретация | Действия |
|---|---|---|
| lib | Библиотечный файл, содержащий объектный код | Обрабатывается компоновщиком |
| f<br>ftn<br>for | Исходный текст на Фортране в фиксированном формате | Обрабатывается компилятором |
| fpp | Исходный текст на Фортране в фиксированном формате | Обрабатывается препроцессором fpp, а затем компилируется IFC7.0 |
| f90 | Исходный текст программы на Фортране 90/95 в свободном формате | Обрабатывается компилятором |
| asm | Исходный текст на ассемблере IA-32 | Обрабатывается ассемблером |
| obj | Откомпилированный объектный файл | Обрабатывается компоновщиком |

## Запуск компилятора

Для вызова компилятора из текстовой консоли IFC7.0 должны быть заданы пути для всех необходимых файлов, включая исполняемые и включаемые файлы, а также библиотеки.

Запустить IFC7.0 в Windows можно следующим образом. Для 32-разрядного компилятора: зайти в **Главное меню** с помощью кнопки **Start** (Пуск), в разделе **Programs** (Программы) выбрать подраздел **Intel Software Development Tools**, войти в подменю **Intel(R) Fortran Compiler 7.0** и щелкнуть мышью на строке **Fortran for 32-bit applications**. Далее в открывшемся окне текстовой консоли запустить компилятор с помощью команды:

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

На платформе Itanium вместо команды `ifl` используется команда `efl`. Ключ представляет собой символьную строку, перед которой располагается наклонная черта (`/`), например:

```
/4I4
```

Ключи являются необязательной частью командной строки. Ключи нельзя объединять, то есть правильной конструкцией является:

```
/1 /4I4
```

и неправильной

```
/14I4
```

Регистр букв в ключах различается, поэтому `/c` и `/C` — разные ключи.

`Ключи`, указанные в одной команде, применяются ко всем файлам, которые используются в этой команде. Ключи могут использовать значения-аргументы. Это могут быть имена файлов, строки, буквы, численные значения. Если строковое значение содержит пробелы, оно должно быть заключено в кавычки.

`Имена_файлов` задают те файлы, которые будут обрабатываться компилятором. Если в строке используется несколько имен файлов, их следует разделить пробелами.

Ключ `/link` используется для того, чтобы, например, 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`, поскольку он передает все следующие после него ключи только компоновщику.

Для компиляции сборки программы в IFC7.0 из среды Microsoft Visual Studio на платформе IA-32 используется инструмент сборки Build Tool. Он может работать только со средой разработки Microsoft Visual C++ 6.0 и не работает с Microsoft Visual Studio.Net.

Инициализация Build Tool выполняется после установки IFC7.0. Для доступа к Build Tool необходимо открыть проект (рабочее пространство Workspace). Затем в текущий проект следует включить файлы с текстами программы на Фортране. Выполнить необходимые установки можно следующим образом.

1. В меню **Project** выполнить команду **Settings**.

2. В открывшемся одноименном диалоговом окне выбрать из списка любой фортрановский файл, при этом появится диалоговое окно с вкладкой **Custom Build**.
3. Выбрать вкладку **Custom Build** и просмотреть или изменить настройки (см. рис. 11.1).

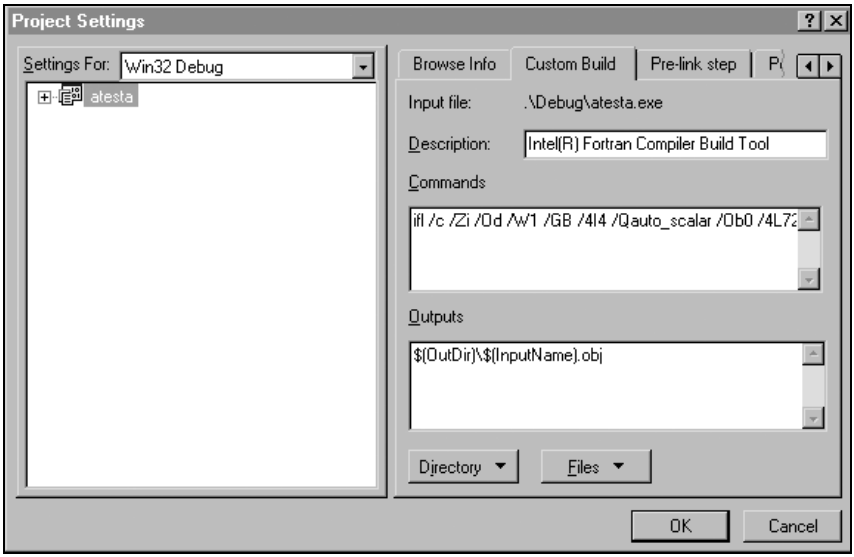



**Рис. 11.1.** Диалоговое окно **Settings**, используемое для подключения компилятора Intel

Отключить IFC7.0 можно с помощью флажка **Always Use Custom Build Step** на вкладке **General** (Общие) окна настройки свойств проекта.

Пользовательский интерфейс инструмента сборки Build Tool позволяет выбирать ключи компилятора на уровне проекта или на уровне конкретного файла. При вызове инструмента, в левой части диалогового окна отображается список фортрановских файлов, содержащихся в проекте. В правой части отображается список ключей компилятора, распределенных по их назначению: общие, диагностические, языковые, ключи оптимизации, ключи управления точностью арифметики с плавающей точкой, ключи генерации кода и ключи указания выходных файлов.

Инструмент сборки позволяет собирать разные конфигурации: Debug Mode и Release Mode. Первая из них соответствует режиму отладки, когда в объектные файлы включается дополнительная информация, необходимая для поиска ошибок. Это увеличивает размер исполняемого файла и уменьшает скорость его выполнения. Во второй конфигурации отладочная информация в объектные файлы не включается. Для каждой из них используются свои наборы ключей, действующие по умолчанию.

## Работа с проектом

Для того чтобы создать новое приложение в Microsoft Visual Studio Development Environment (C++ 6.0), необходимо выполнить действия.

1. Войти в меню **File**, перейти к подменю **New** и выбрать пункт **Projects**.
2. В открывшемся окне ввести путь и имя нового проекта.
3. Создать пустой проект.
4. Добавить в проект файлы с исходным текстом программы. Для этого нужно перейти из меню **Project** к подменю **Add To Project** и выбрать команду **Files**. В открывшемся диалоговом окне, в списке **Type of files**, по умолчанию не содержатся расширения f, fpp, for и f90, поэтому в поле ввода **File name** нужно ввести суффикс имени файла (например, .f), после чего нажать клавишу <Enter>. В окне отобразятся имена файлов с расширением f, расположенные в текущем каталоге. Остается выбрать файлы и нажать кнопку **OK**. В проект будут включены необходимые файлы.
5. Для того чтобы создать новый файл с текстом программы на Фортране, необходимо в меню **File** перейти к разделу **New** и выбрать команду **Files**. Затем в появившемся диалоговом окне выбрать в списке **Files of type** текстовый тип файла **Text File** и в поле ввода **File name** ввести имя файла с соответствующим расширением. В проект будет добавлен новый, пока еще пустой файл. В редакторе можно набрать текст программы.

Для сборки проекта необходимо включить инструмент сборки и ассоциировать компилятор IFC7.0 с файлами проекта. Если уже после вызова инструмента сборки в проект добавляются новые файлы, перед сборкой инструмент необходимо активизировать заново. Для того чтобы "запомнить" все файлы, содержащиеся в текущем проекте, инструмент сборки должен сохранить текущее рабочее пространство.

При работе в редакторе ключевые слова могут выделяться цветом. Эта возможность среды является настраиваемой.

## Настройка среды компиляции

По умолчанию компилятор создает исполняемый файл. Это происходит, разумеется, в том случае, когда исходный текст программы не содержит синтаксических ошибок. Пользователь может ограничить процесс компиляции выполнением определенного количества этапов. В этом случае вместо исполняемого файла могут создаваться объектные файлы, файлы на языке ассемблера и т. д. Так, например, с ключом `/c` выполняются только этапы компиляции, ассемблирования и создания объектного файла. Компоновка в этом случае не производится. С ключом `/S` работа компилятора завершается после создания файла с текстом программы на языке ассемблера. Можно менять и свойства исполняемого файла, в том числе использование

программой стека, проверки времени выполнения, использование регистров и т. д.

Следует отметить некоторые особенности "поведения" компилятора по умолчанию. Поиск всех файлов, в том числе и библиотечных, выполняется в текущем каталоге, а если их там нет, поиск библиотечных файлов продолжается в каталогах, заданных в *переменной окружения* `LIB` (см. ниже). Ключи, предназначенные для компоновки, а также расположение заданных пользователем библиотек, передаются компоновщику. Компилятор выводит сообщения об ошибках и предупреждения. Поддерживается расширение стандарта ANSI Фортрана (ANSI — American National Standard Institute, Американский Национальный Институт Стандартизации). Имеются установки и ключи, действующие по умолчанию. Для платформы IA-32, например, компилятор использует ключ `/G7` для оптимизации кода для процессоров Pentium 4 и Xeon, а для платформы Itanium компилятор использует ключ `/G2` оптимизации кода под процессор Itanium 2.

Для настройки среды компиляции можно использовать переменные окружения, ключи, конфигурационные файлы, файлы зависимостей (response files), которые используются для хранения ключей компиляции, файлы проекта и включаемые файлы.

Ниже перечислены некоторые из переменных окружения:

- `F_UFMTENDIAN` — задает номера логических устройств, используемых для преобразования из формата Little-endian (прямой порядок байтов) в формат Big-endian (обратный порядок байтов; разъяснение обоих способов адресации байтов см. далее в этой главе);
- `IFLCFG` — задает конфигурационный файл, который должен использоваться компилятором вместо файла по умолчанию (IA-32);
- `INCLUDE` — путь поиска каталога с включаемыми файлами;
- `LIB` — путь поиска библиотечных файлов. Используется компилятором и компоновщиком Microsoft (linker.exe);
- `PATH` — путь поиска каталога, содержащего исполняемые файлы компилятора;
- `TMP` — каталог хранения временных файлов. Если этот каталог не задан, временные файлы создаются в текущем каталоге.

В ОС Windows 98/NT переменные окружения можно задать в файле AUTOEXEC.BAT. В Windows 2000 последовательность шагов следующая.

1. Открыть **Control Panel** (Панель управления).
2. Дважды щелкнуть на пиктограмме **Administrative Tools** (Администрирование), а затем в появившемся одноименном окне выбрать из списка пункт **Computer Management** (Управление компьютером).

3. В открывшемся одноименном окне раскрыть папку **Systems Tools** (Служебные программы), в ней выбрать и раскрыть папку **System Information** (Сведения о системе) и, наконец, в ней раскрыть папку **Software Environment** (Программная среда) и щелкнуть на вложенной в нее папке **Environment Variables** (Переменные среды).

Ключи компилятора можно хранить в конфигурационном файле. Для платформы IA-32 по умолчанию это файл ifl.cfg, который должен находиться в каталоге **\bin** или по адресу, указанному в переменной окружения `IFLCFG`.

Конфигурационные файлы используются для того, чтобы сократить время на ввод часто используемой команды вызова компилятора. В конфигурационном файле можно использовать любые допустимые ключи. Сначала компилятор использует эти ключи и лишь затем ключи командной строки.

Пример конфигурационного файла для платформы IA-32 приведен в листинге 11.1. Символ `#` обозначает комментарий.

**Листинг 11.1. Конфигурационный файл компилятора IFC7.0 для платформы IA-32**

```
## Sample ifl.cfg file
## Define preprocessor macro MY_PROJECT
/DMY_PROJECT
## Set extended-length source lines.
/4L132
##
## Set maximum floating-point significand precision.
/Qpc80
##
## Additional directories to be searched for include
## files, before the default.
/Ic:\project\include
## Use the static, multithreaded C run-time library.
/MT
```

Для хранения ключей, а также имен файлов, которые используются при компиляции конкретных проектов, используются файлы зависимостей. Имена этих файлов указываются в командной строке. Ключи, содержащиеся в таком файле, подставляются в командную строку в том ее месте, где содержится имя файла. Файл зависимостей — это конфигурационный файл, привязанный к определенному проекту. Перед именем файла должен находиться символ `@`:

```
ifl @response_filename
```

Включаемые файлы подставляются в программу с помощью директивы препроцессора `#INCLUDE`. Стандартные включаемые файлы определены в каталогах, заданных в переменной окружения `INCLUDE`. Можно задать положение включаемых файлов с помощью ключей компилятора `/I` и `/X`.

## Препроцессор

С IFC7.0 поставляется препроцессор fpp. Обработка препроцессором является необязательной фазой компиляции и применяется для макроподстановок, условной компиляции и подстановки файлов. Управлять работой препроцессора можно с помощью ключей препроцессора из командной строки. Если ключи препроцессора не указаны, результат его работы не сохраняется, а напрямую передается компилятору. Если необходимо использовать другой препроцессор, он должен вызываться перед вызовом компилятора. Применение препроцессора, который не поддерживает Фортран 95, может "испортить" текст программы. Препроцессор fpp совместим с препроцессором cpp (для языка C) и распознает директивы в духе cpp.

Компилятор автоматически вызывает препроцессор, если имя файла имеет соответствующий суффикс. Для автоматического вызова препроцессора используются расширения fpp и F90.

Использование препроцессора может затруднить отладку. Для того чтобы этого избежать, можно сохранить результат обработки (ключ `/P`) и откомпилировать его отдельно.

Обработку препроцессором можно включить для любого файла с исходным текстом на Фортране с помощью ключа `/Qfpp{n}`. Ключ `/Qopenmp` вызывает препроцессор автоматически.

## Компиляция программ с модулями

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. Эти файлы содержат необходимую информацию о модулях.

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, содержащихся в разных каталогах, IFC7.0 использует для поиска mod-файлов ключ `/I`*каталог*, например:

```
ifl /c user_mod.f90 /I\usr\svromano\project
```

При работе с препроцессором можно использовать макроподстановки. Для этого используется ключ `/D` или директива препроцессора `#define`:

```
/Dимя[= value({#|text})]
```

где *имя* — имя макроопределения, `value({#|text})` — значение, которым заменяется имя. Если `value` содержит пробелы или специальные символы, оно должно быть заключено в кавычки. При обработке препроцессором каждый идентификатор *имя* в программе заменяется его значением `value`, например:

```
ifl /DSIZE = 100 evalprog.f
```

В этом случае в конструкциях вида:

```
REAL A(SIZE)
```

вместо `SIZE` будет подставлено значение 100.

Ключ /U*имя* запрещает макроподстановки. То же делает и директива `#undef`.

В IFC7.0 имеются предопределенные макросы, которые вызываются в определенных ситуациях. Со списком предопределенных макросов можно познакомиться в документации по компилятору.

## Преобразование из формата Little-endian в формат Big-endian

Английский термин Big-endian произошел от слова, которым Свифт в романе "Путешествия Гулливера" называл людей, полагавших, что яйцо следует разбивать с тупого конца, и которое обычно переводится как "тупоконечник". В программировании это слово обозначает тип компьютерной архитектуры, у которой в значениях, занимающих несколько байт, старший байт имеет наименьший адрес. В этом случае слово хранится по схеме "big-end-first". Платформы IBM, HP-UX, AIX, Solaris и Macintosh относятся к этому классу, а платформы VAX/VMS, AXP/VMS, Digital UNIX, Intel, OS/2 и Windows считаются Little-endian.

На платформах Big-endian единица (в шестнадцатеричной системе счисления) хранится: в однобайтовом формате как 01, в двухбайтовом — 00 01, в четырехбайтовом — 00 00 00 01. На платформах Little-endian представление единицы выглядит, соответственно, следующим образом: 01, 01 00 и 01 00 00 00.

Преобразование из формата Little-endian в формат Big-endian предназначено для операций неформатного ввода-вывода. Оно делает возможным обработку файлов с организацией данных по схеме Big-endian на платформе IA-32, на которой обычно используется формат Little-endian. Указанное преобразование не применяется к производным типам.

Для того чтобы использовать преобразование между форматами Little-endian и Big-endian, необходимо задать номер логического устройства,

используемого для преобразования. Это значение задается с помощью переменной окружения F_UFMTENDIAN. Операторы READ и WRITE, использующие заданные логические устройства, будут выполнять необходимые преобразования. Значение переменной задается следующим образом:

```
F_UFMTENDIAN = MODE | [MODE;] EXCEPTION
```

где:

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST, U
U = decimal | decimal -decimal
```

Параметр MODE задает текущий формат данных, он может быть опущен. Ключевое слово little означает, что данные уже имеют формат Little-endian. В этом случае преобразование не выполняется. На платформе IA-32 это значение используется по умолчанию. EXCEPTION позволяет задать список исключений для MODE, эта часть необязательна. Элементами списка U являются номера устройств.

После выполнения, например, команды:

```
export F_UFMTENDIAN = F_UFMTENDIAN = big
```

при выполнении операций ввода-вывода будет выполняться преобразование из формата Big-endian в формат Little-endian при вводе и из формата Little-endian в формат Big-endian — при выводе.

## Выходные файлы

Результатом выполнения команды:

```
ifl pr.f90
```

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

Допускается одновременная компиляция нескольких файлов:

```
ifl pr1.f90 pr2.f90
```

В результате выполнения этой команды будут откомпилированы и скомпонованы два файла, будут созданы два объектных файла pr1.obj и pr2.obj, будет создан исполняемый файл pr1.exe и все файлы будут сохранены в текущем каталоге. Имя исполняемого файла можно задать с помощью ключа /Fe*файл*. В качестве значения этого ключа может быть задано имя каталога, оно должно заканчиваться символом наклонной черты (/) или обратной наклонной черты (\).

## Ключи отладки

Для отладки программы иногда удобно включить в ее текст операторы, выводящие, например, 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, находится символ D. При компиляции программы с ключом /Qd_lines этот символ замещается пробелом и строка обрабатывается обычным образом, поэтому при выполнении команды

```
ifl /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 иногда необходимо запретить все виды оптимизации. Это можно сделать с помощью ключа /Od. Если оптимизация все-таки нужна, вместе с ключами /O1 или /O2 рекомендуется использовать ключ /Zi. Тем не менее, следует помнить, что одновременное использование ключей отладки и ключей оптимизации иногда приводит к непредсказуемым результатам.

## Оптимизация

Оптимизация на этапе компиляции может включаться соответствующими ключами, директивами препроцессора, а также поддерживается некоторыми библиотечными функциями и утилитами. Основными ключами оптимизации являются /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, оптимизируется конвейеризация и т. д. На третьем, более высоком, уровне, оптимизируются циклы, используется упреждающая выборка команд и выполняются другие действия. Эта оптимизация считается более "агрессивной" и применяется в тех программах, в которых велика доля операций с плавающей точкой. Она обычно приводит к увеличению быстродействия программы, хотя большой эффект не гарантирован. Ключи оптимизации могут действовать по-разному на платформах IA-32 и Itanium.

## Управление точностью операций с плавающей точкой

Ключ `/Op` запрещает некоторые виды оптимизации кода, которые могут привести к потере точности. Процессоры Intel обычно хранят результаты операций с плавающей точкой в 80-разрядных регистрах, что превышает разрядность значений с двойной точностью. 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 даже с двойной точностью выполняется округление до 32 (в первом случае) или до 64 (во втором случае) разрядов. С ключом `/Qop` соблюдаются и некоторые другие условия.

Ключ `/Qprec_div` запрещает оптимизацию посредством замены деления умножением на величину, обратную знаменателю. Это немного увеличивает точность вычисления результата деления, но, в то же время, уменьшает скорость выполнения программы.

Назначение ключа `/Qpc{32|64|80}` следующее. Алгоритмы некоторых операций с плавающей точкой для платформы IA-32 чувствительны к точности представления мантиссы вещественного числа. Выбор подходящего ключа позволяет выбрать разумный компромисс между точностью вычислений и скоростью работы программы.

## Другие виды оптимизации

При компиляции IFC7.0 может выполнять оптимизацию использования подпрограмм. Такая оптимизация называется межпроцедурной (Inter Procedural Optimization — IPO) и включает, в частности, подстановки кода подпрограмм, а также некоторые другие приемы. Для межпроцедурной оптимизации используются ключи `/Qip` и `/Qipo`.

Другой вид оптимизации называют оптимизацией, управляемой профилированием программы (по-английски это название короче — Profile-guided Optimization — PGO). В этом случае проводится предварительное исследование затрат времени на выполнение различных частей программы. Оптимизация проводится с учетом полученной информации.

Код при профилировании генерируется таким образом, что при каждом выполнении программы создается специальный информационный файл. Этот файл и используется компилятором при оптимизации тех частей программы, которые выполняются чаще всего.

Оптимизация высокого уровня (High Level Optimization — HLO) основана на свойствах алгоритмических конструкций, используемых в программе,

и включает такие преобразования, как перестановка циклов, слияние циклов и их развертка, предварительная выборка данных и т. д. Основной ключ высокоуровневой оптимизации `/O3`.

## Распараллеливание и векторизация

IFC7.0 поддерживает спецификацию параллельного программирования OpenMP (версия 2.0), а также автоматическое распараллеливание для симметричных многопроцессорных систем. При автоматическом распараллеливании создается многопоточный код.

Важнейшим инструментом распараллеливания является "векторизатор", который находит в программе операции, которые могут выполняться одновременно, а затем преобразует последовательную программу таким образом, чтобы обрабатывать 2, 4, 8 или 16 элементов за одну операцию. Распараллеливание и векторизацию иногда можно сочетать.

## Смешанное программирование на языках С и Фортран

При разработке относительно сложных программ иногда удобно некоторые подпрограммы запрограммировать на С, а вызывать их из программы, написанной на Фортране (или наоборот). Для этого следует учитывать соглашения об именах, принятые в этих языках.

По умолчанию компилятор Фортрана преобразует буквы в именах подпрограмм в верхний регистр. Компилятор С такое преобразование не выполняет никогда, следовательно, подпрограмме, написанной на С и вызываемой из программы на Фортране, следует присвоить имя с учетом правильного регистра:

```
CALL ABPROC(I, X)
```

Соответствующая данному вызову подпрограмма С должна содержать буквы в верхнем регистре.

При использовании ключа `/Qlowercase` символические имена для внешнего компоновщика, в том числе имена подпрограмм, используют только нижний регистр букв. К внешним именам добавляется символ подчеркивания (в конце имени). Это делается для того, чтобы избежать конфликтов имен с подпрограммами С. Данный ключ применяется при смешанном программировании на языках Фортран и С. Его совместное использование с некоторыми другими ключами запрещено.

Следующая проблема, которую необходимо решить при программировании на С и Фортране — передача параметров подпрограмм. По умолчанию, параметры подпрограмм в Фортране передаются по ссылке, то есть передается указатель на значение, а не само значение. В С наоборот, передаются

значения параметров. С учетом этого, если из программы на Фортране вызывается функция, на языке С, ее формальные параметры необходимо описать как указатели на соответствующие типы данных. Если наоборот, программа на С содержит вызов подпрограммы на Фортране, каждый фактический параметр необходимо явно описать как указатель.

IFC7.0 имеет ключи, предназначенные для оптимизации смешанных программ.

## Ограничения IFC7.0

Ограничения IFC7.0 приведены в табл. 11.6.

***Таблица 11.6.*** *Ограничения IFC7.0*

| Элемент | Максимальное количество |
|---|---|
| Вложенность интерфейсных блоков | 10 |
| Вложенность неявных циклов ввода-вывода | 20 |
| Вложенность неявных циклов в конструкторах массивов | 20 |
| Вложенность включаемых файлов | 10 |
| Длина символьной константы | 32767 |
| Число разрядов в числовой константе | 1024 |
| Вложенность спецификаторов форматов в скобках | 20 |
| Вложенность циклов и ветвлений | 100 |
| Количество аргументов в функциях `MIN` и `MAX` | 255 |
| Количество параметров | 256 |
| Количество строк продолжения | 99 |
| Ширина поля для числового спецификатора преобразования | 1024 |

## Утилита nmake

Для того чтобы задать многочисленные файлы, их пути, ключи компиляции и сборки, для последующей неоднократной компиляции, используют специальные файлы — makefile. Для того чтобы использовать этот файл при компиляции, необходимо ввести команду:

```
nmake /f psmproj.mak FPP = ifl.exe LINK32 = xilink.exe
```

Аргументы команды:

- `/f` — задать имя файла;
- psmproj.mak — makefile, который необходимо использовать для создания объектных и исполняемых файлов;

- FPP — команда вызова компилятора;
- LINK32 — задает программу-компоновщик.

Команда `nmake` при выполнении автоматически создает объектные (obj) и исполняемые (exe) файлы, указанные в файле psmproj.mak.

## Компилятор 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). Первоначально проект разрабатывался некоммерческой организацией Free Software Foundation (FSF), целью которой была разработка открытого, свободно распространяемого программного обеспечения. Система GNU Фортран 77 разрабатывалась в качестве свободно распространяемой замены компилятору UNIX f77, совместимой с другими средствами разработки проекта GNU. 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.

G77 состоит из фронтальной части, которая выполняет компиляцию исходного текста программы на Фортране, и выходной части, использующей результат трансляции для создания объектных и исполняемых файлов. Выходная часть — та же, что у GNU C, C++ и Objective-C, которые имеют свои собственные фронтальные части, выполняющие трансляцию с соответствующих языков. Существуют фронтальные части и для других языков, например для Pascal. G77 не конвертирует исходный фортрановский код в текст на языке C, это "настоящий" компилятор.

G77 содержит несколько компонентов.

- Модифицированную версию компилятора gcc. В состав G77 входит усовершенствованная версия gcc, которая не только распознает исходные файлы на языках C, C++ и Objective C, но и, разумеется, на Фортране. Установка версии gcc для работы с файлами на Фортране не влияет на работу с файлами программ на других языках, но без установки фортрановской версии GNU Fortran gcc работа с программами на Фортране невозможна.
- Команду g77, которая, собственно, и является фронтальной частью системы G77. Она "знает", какие библиотеки необходимы для компоновки программ на Фортране. Команда g77 выполняет анализ командной строки и после необходимой модификации передает ее команде gcc.
- Библиотеку времени выполнения libg2c, которая обеспечивает те возможности языка Фортран, которые не поддерживаются 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. Эта программа является наиболее интересным элементом G77.

Программа f771 состоит из двух частей. Одна из них называется GNU Back End (GBE) и "умеет" генерировать быстрый исполняемый код для ряда платформ. Та же выходная часть GBE используется программами C, C++ и программами компилятора Фортрана cc1, cc1plus и f771. Иногда GBE и gcc используются в качестве синонимов, но это, строго говоря, не так.

Вторая часть f771 является самой уникальной в G77. Она 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 (FFE).

Код основных арифметических операций подставляется в исполняемый файл программой g77, а обращения к тригонометрическим и иным функциям обрабатываются компилятором f771, который вызывается программой g77. Эти функции содержатся в библиотеке libg2c.

## Основные ключи G77

Перейдем к обзору основных ключей системы G77. Этот перечень, как обычно, будет 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:

```
-fключ
```

и

```
-fno-ключ
```

соответственно. В первом случае выполняется активизация некоторой функции или режима работы системы, а во втором эта функция (режим) отключаются.

Ключи можно разделить на следующие категории.

- Ключи общего назначения, которые определяют результат работы компилятора. Результатом могут быть исполняемые, объектные файлы, файлы с текстом на ассемблере, исходный файл, обработанный препроцессором.
- Ключи, управляющие диалектом Фортрана.
- Ключи, управляющие выводом предупреждений.
- Ключи отладки.

- Ключи оптимизации.
- Ключи препроцессора, управляющие макроопределениями, заголовочными файлами и т. д.
- Ключи, задающие каталоги с заголовочными файлами, библиотеками и исполняемыми файлами компилятора.
- Ключи генерации кода, управляющие использованием регистров процессора, определяющие соглашения о вызовах функций и т. д.

## Ключи, управляющие результатом компиляции

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, который не нуждается в обработке препроцессором. Такой текст не должен содержать директив препроцессора (`#include` и т. д.). Назначить обработку такого файла препроцессором cpp можно с помощью ключа `-x f77-cpp-input`.

Если суффикс имеет вид .F, .fpp или .FPP, то файлы содержат текст на Фортране, который должен быть обработан препроцессором cpp. Файлы, включенные директивой `INCLUDE`, препроцессор пропускает, поскольку понимает директиву `#include`.

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

Препроцессор cpp допускает использование C-подобных конструкций вида `#define` и `#include`, но в программах на Фортране это может приводить к различным ошибкам. Использование препроцессора C рекомендуют ограничивать применением директивы `#include`, а также совместно директив `#define` и `#if`. Эту рекомендацию следует соблюдать особенно строго при работе с фиксированным форматом исходного текста. При работе со свободным форматом записи текста источником проблем могут быть холлеритовские и символьные константы, особенно, если они занимают несколько строк. Подобные конструкции по-разному интерпретируются препроцессором C и компилятором Фортрана.

Источником ошибок могут быть комментарии в программе на Фортране. Так, например, если строка комментария заканчивается символом обратной

наклонной черты, препроцессор не воспринимает ее как комментарий, а обрабатывает в соответствии со своими собственными правилами: обратная наклонная черта в конце строки убирается, убирается и следующий за ней служебный символ конца строки, в результате чего текущая строка объединяется со следующей, которая может оказаться уже не строкой комментария.

Для того чтобы избежать подобных неприятностей, можно использовать ключи `-traditional` и `-undef` команды cpp (они используются по умолчанию), которые отключают некоторые особенности препроцессора ANSI C (такие как оператор `#`). Следует обратить внимание и на имена, используемые в директивах. Более подробно данный вопрос рассматривается в документации по G77, которая поставляется вместе с компилятором. Узнать, какие ключи используются при вызове препроцессора, можно с помощью ключа `-v`.

## Ключи, управляющие диалектом Фортрана

Некоторые ключи, управляющие диалектом Фортрана, приведены в табл. 11.7.

***Таблица 11.7.*** *Ключи G77, управляющие диалектом Фортрана*

| Ключ | Описание |
|---|---|
| `-ffree-form`<br>`-fno-fixed-form` | Определяет формат записи исходного текста программы — свободный или фиксированный |
| `-ff90` | Позволяет использовать некоторые конструкции Фортрана 90 |
| `-fno-backslash` | Запрещает интерпретацию символа `\` в составе символьных и холлеритовских констант в качестве специального символа. С ключом `-fbackslash` последовательность `A\nB`, например, интерпретируется как три символа, причем второй — символ новой строки. Та же последовательность при компиляции с ключом `-fno-backslash` считается четырьмя символами: `A`, `\`, `n` и `B` |
| `-fno-ugly-args` | Запрещает передачу холлеритовских или нетипизированных констант в качестве фактических аргументов подпрограмм (например, `CALL GREETING(5HHELLO)`) |
| `-fugly-assumed` | Считать, что любой массив — формальный параметр, для верхней границы индекса которого задано значение 1, является массивом с подразумеваемым размером, например `DIMENSION A(1)` трактуется как `DIMENSION A(*)` |
| `-fugly-complex` | Разрешает использование конструкций вида `REAL(EXPR)` или `AIMAG(EXPR)`, где выражение `expr` имеет комплексный тип с параметром разновидности, отличным от `KIND = 1`. Интерпретацией этой конструкции можно управлять с помощью ключа `-ff90` |

***Таблица 11.7*** *(окончание)*

| Ключ | Описание |
|---|---|
| `-fno-ugly-init` | Запрещает использование холлеритовских и нетипизированных констант в качестве начальных значений в операторах `PARAMETER` и `DATA`, а также использование символьных констант для инициализации числовых типов (например, `DATA A/'B'/`) |
| `-fugly-logint` | Позволяет переменным целого и логического типа заменять друг друга. Включает механизм автоматического преобразования значений |
| `-fonetrip` | При компиляции с этим ключом циклы `DO` выполняются не менее одного раза, даже если параметры цикла заданы таким образом, что число повторений оказывается меньше единицы. Этот ключ действует только на циклы `DO` и неявные циклы в списках ввода-вывода |
| `-ffixed-line-length-n` | Задает номер колонки, после которой при записи текста программы в фиксированном формате символы игнорируются. Обычно `n` равно 72 (по умолчанию), 80 или 132 |

## Ключи управления выводом предупреждений

Ключи управления выводом предупреждений обычно начинаются символами `-W`. У каждого из этих ключей есть как положительная, так и отрицательная форма, которая начинается с `-Wno-` и отключает вывод предупреждений определенного типа. Некоторые из этих ключей приведены в табл. 11.8.

***Таблица 11.8.*** *Ключи G77, управляющие выводом предупреждений*

| Ключ | Описание |
|---|---|
| `-fsyntax-only` | Выполнять только проверку на наличие синтаксических ошибок |
| `-pedantic` | Выводить предупреждения об использовании расширений ANSI FORTRAN 77. Правильно составленные программы, следующие этому стандарту, должны правильно компилироваться как с этим ключом, так и без него. Без этого ключа также поддерживаются некоторые расширения GNU и традиционный Фортран. С этим ключом многие из них будут отклонены. Этот ключ — средство соблюсти стандарт языка в программах пользователя |
| `-pedantic-errors` | Действие аналогично действию ключа `-pedantic`, но вместо предупреждений формируются сообщения об ошибках |

***Таблица 11.8*** *(окончание)*

| Ключ | Описание |
|---|---|
| `-fpedantic` | Действие этого ключа аналогично действию ключа — `pedantic`, но применяется только к конструкциям языка Фортран |
| `-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 |
| `-Wimplicit` | Выводить предупреждение о неявном объявлении переменной, массива или функции |
| `-Wunused` | Выводить предупреждение, если переменная не используется |
| `-Wuninitialized` | 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` | Объединение ключей `-Wunused` и `-Wuninitialized` |
| `-Wsurprising` | 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 |

## Ключи отладки и оптимизации

Некоторые из ключей отладки и оптимизации G77 приведены в табл. 11.9.

***Таблица 11.9.*** *Ключи G77, используемые для отладки и оптимизации*

| Ключ | Описание |
|---|---|
| `-g` | При компиляции с этим ключом генерируется отладочная информация, которая сохраняется в формате операционной системы (stabs, COFF, XCOFF, DWARF). 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` |
| `-malign-double` | Данный ключ применяется только на архитектуре Intel x86 и улучшает производительность программ, откомпилированных G77. Это связано с более быстрым выполнением программ на процессорах Pentium, Pentium Pro, 586 и 686, если значения с двойной точностью выравниваются на 64-битную границу в памяти |
| `-fforce-mem` | Оптимизация циклов |
| `-fforce-addr` | |
| `-ffast-math` | Включение "быстрой арифметики", что позволяет увеличить скорость выполнения некоторых программ. Активизируются ключи `-funsafe-math-optimizations`, `-ffinite-math-only` и `-fno-trapping-math` |
| `-funsafe-math-optimizations` | Включает режим "опасной" оптимизации, которая может привести к неправильным результатам |
| `-funroll-loops` | Оптимизация циклов `DO` с помощью метода "развертки циклов" |
| `-funroll-all-loops` | Развертка циклов применительно к циклам `DO` и циклам `DO WHILE` |

## Ключи поиска каталога

Основным ключом поиска каталога является:

`-I-`

`-I`*`каталог`*

после которого указывается имя каталога, содержащего включаемые файлы, имена которых указаны в операторе `INCLUDE`. Пробелы между `-I` и именем каталога не допускаются.

## Ключи генерации кода

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. 11.10.

***Таблица 11.10.*** *Ключи G77, управляющие генерацией исполняемого кода программы*

| Ключ | Описание |
|---|---|
| `-fno-silent` | При компиляции программы в стандартный вывод выводятся имена блоков программы |
| `-fno-automatic` | Каждый модуль программы обрабатывается так, как если бы он содержал оператор `SAVE` с указанием имен всех переменных и массивов, используемых в этом блоке. На `COMMON`-блоки действие данного ключа не распространяется |
| `-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. С этим ключом рекомендуется использовать ключ `-fno-automatic` |
| `-fno-second-underscore` | Запрещает присоединение к именам, использующимся в исходном тексте программы на Фортране, второго символа подчеркивания. Действие данного ключа отменяется ключом `-fno-underscoring` |
| `-fno-globals` | Запрещает вывод диагностических сообщений о проблемах взаимодействия между процедурами, такими, например, как несоответствие типов аргументов. Точнее говоря, сообщения переводятся в разряд предупреждений. Выполняются и некоторые другие действия |
| `-fbounds-check`<br>`-ffortran-bounds-check` | Проверка выхода значений индексов массива за установленные для них границы во время выполнения программы. Ключ `-ffortran-bounds-check` используется для программ на Фортране |

# Особенности диалекта Фортрана GNU

Фортран GNU включает и некоторые возможности Фортрана 90.

- Свободный формат записи исходного текста, который поддерживается при компиляции с ключом `-ffree-form`.
- Автоматические массивы в процедурах.
- Строка символов может иметь нулевую длину, допускаются также подстроки из символьных констант.

- Имена конструкций.
- Конструкции: `CYCLE`, `EXIT`, `DOUBLE COMPLEX`, `DO WHILE`, `KIND`, `IMPLICIT NONE`, оператор `INCLUDE`.
- Ввод-вывод, управляемый списком и `NAMELIST`-ввод-вывод для внутренних файлов.
- Двоичные, восьмеричные и шестнадцатеричные константы.
- Дескрипторы формата `O` и `Z`.
- Операции отношения `<`, `<=`, `==`, `/=`, `>` и `>=` вместо `.LT.`, `.LE.`, `.EQ.`, `.NE.`, `.GT.` и `.GE.` соответственно.
- Конструкция `SELECT CASE`, которая реализована частично.
- Предложения описания. Имеется `KIND`, хотя встроенные функции для работы с параметром разновидности типа отсутствуют.

## Отладка

Здесь мы не будем подробно обсуждать технику отладки при работе с G77, ограничившись одним вопросом. При компиляции главной программы G77 присваивает ей имя общей (public) процедуры `MAIN__`. Библиотека libg2c содержит процедуру `main()`, которая выполняет некоторые подготовительные действия, и лишь после этого вызывает `MAIN__`.

Система G77 и библиотека libg2c устроены таким образом, что главная программа не обязательно должна быть написана на Фортране. В этом случае рекомендуется внимательно изучить исходный текст процедуры gcc/libf2c/libF77/main.c, чтобы выяснить, что должна сделать ваша собственная процедура `main()` для правильной работы с кодом на Фортране.

Процедура `main()` из библиотеки libg2c, например, формирует информацию, которая используется встроенными функциями `IARGC` и `GETARG`. Если не позаботиться о соответствующих действиях, вызов `IARGC` и `GETARG` будет приводить к непредсказуемым последствиям.

При отладке следует помнить, что "стартовой" точкой программы является процедура `main()` из библиотеки libg2c.

Обойти эту проблему можно, установив точку останова в точке входа в процедуру `MAIN__`. Это можно сделать с помощью команды отладчика gdb:

```
tbreak MAIN__
```

добавив ее в файл .gdbinit в рабочем каталоге отладчика gdb. Это приведет к тому, что отладчик будет "считать" стартовой точкой главную программу.

# Система программирования Compaq Visual Fortran

Система программирования Compaq Visual Fortran продолжает линию, началом которой была система Microsoft Fortran Powerstation, а продолжением — Digital Visual Fortran. Она предназначена для работы в операционной системе MS Windows и включает в свой состав интегрированную среду разработки Microsoft Visual Studio, которую можно использовать совместно с Microsoft Visual C++. Для платформы Tru64 UNIX поддерживается параллельное выполнение с использованием спецификаций OpenMP или HPF. Compaq Fortran для платформы Linux Alpha распространяется бесплатно для некоммерческого использования. Все продукты Compaq Fortran 95 включают оптимизированную математическую библиотеку Compaq Extended Math Library.

По данной среде имеется разнообразная литература, поэтому мы ограничимся кратким обзором основных ее особенностей. Будучи достаточно мощным инструментом разработки, Compaq Visual Fortran достоин отдельной книги. В дальнейшем мы будем использовать сокращение CVF.

## Проекты CVF

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. С консольной программой можно использовать динамические и многопоточные библиотеки.

*Проект со стандартной графикой* использует одно графическое окно, в которое могут выводиться графические примитивы. Графика поддерживается библиотекой QuickWin. В программе можно использовать окна диалога. 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.

*Проект статической библиотеки* позволяет создавать статические библиотеки (файлы с расширением lib). Статическая библиотека содержит объектный код подпрограмм, которые при необходимости включаются в исполняемый код программы, использующей данную библиотеку.

*Проект динамической библиотеки* позволяет создавать динамические библиотеки (файлы с расширением dll). Динамическая библиотека разделяет код и адресное пространство данных с программой, которая ее вызывает. Размер исполняемого кода программы, использующей динамические библиотеки, меньше. К одной динамической библиотеке одновременно могут обращаться разные программы. При обращении к динамической библиотеке подпрограммы из нее загружаются в память по мере необходимости.

Любой из вышеперечисленных проектов может быть создан в одной из двух конфигураций — отладочной (debug) и окончательной (release). В первом случае проект содержит отладочную информацию, размер исполняемого файла увеличен, а скорость выполнения программы меньше. Во втором случае отладочная информация удалена, размер исполняемого файла меньше, а скорость выполнения программы больше.

Параметры конфигурации проекта хранятся в специальных файлах, имеющих расширения dsw (информация о рабочем пространстве проекта), dsp (файл проекта) и opt (параметры среды — геометрия окон, точки останова и т. д.). Эти файлы создаются и модифицируются с помощью интегрированной среды CVF.

## Обработка ошибок

При работе в интегрированной среде сообщения об ошибках компиляции и компоновки отображаются в окне Build. Сообщение об ошибке имеет следующий формат:

```
Имя файла(номер строки исходного текста) : степень "опасности" ошибки:
текст сообщения
          [фрагмент текста]
--------^
```

Указателем `---^` отмечается положение ошибки.

Найти строку исходного текста с ошибкой легко, для этого достаточно щелкнуть два раза по сообщению об ошибке или нажать клавишу <F4>. Каждое нажатие этой клавиши выводит сообщение о следующей ошибке.

В CVF имеются возможности для отладки, для этого используется собственный отладчик. Полезной утилитой является также инструмент визуализации массивов (Array Viewer).

В CVF имеются также функции (`CPU_TIME`, `SYSTEM_CLOCK`, `DATE_AND_TIME`, `TIME`, `ETIME` и `TIME`), которые можно использовать для профилирования программы.

## Ограничения компилятора

Ограничения CVF версии 6.6 приведены в табл. 11.11.

***Таблица 11.11.*** *Ограничения CVF6.6*

| Элемент | Максимальное количество |
|---|---|
| Размерность массива | 7 |
| Длина символьных констант | 2000 символов |
| Строки продолжения | 511 |
| Вложенность операторов `DO` и `IF` | 128 |
| Вложенность описаний формата | 8 |
| Длина строки исходного текста | 132 символа |
| Вложенность включаемых файлов | 20 |
| Именованные общие блоки | 250 |
| Вложенность скобок в выражениях | 40 |
| Вложенность структур | 20 |
| Длина имени | 63 символа |

# Компилятор Lahey/Fujitsu Fortran 95

Компилятор Lahey/Fujitsu Fortran 95 (LF95) предназначен для использования на платформе Linux с процессорами Intel от: 80486DX до Pentium 4. Это коммерческий продукт, который поставляется в разных вариантах: LF95 Linux PRO, LF95 Linux Express. Фирмой Lahey поставляется также компилятор Essential Lahey Fortran 90 (ELF90). Он представляет собой немного "урезанную" версию Фортрана 90, которая подходит для образовательных целей. Все вышеперечисленные компиляторы предъявляют весьма умеренные требования к ресурсам системы и могут работать в среде различных ОС Linux (Red Hat Linux, Slackware, Debian GNU/Linux, SuSE, Caldera Open Linux).

В состав системы входят:

- компилятор Фортрана 95;
- утилита архивации Linux — ar, предназначенная для работы со статическими библиотеками;
- компоновщик Linux — программа ld;
- отладчик fdb, совместимый с Фортраном 95;
- вспомогательные утилиты.

Компилятор (точнее говоря, система разработки программ) LF95 наделен всеми функциями, необходимыми для разработки, отладки и оптимизации программ на Фортране 95. Поддерживается смешанное программирование, а также генерация исполняемого кода для параллельного выполнения.

Возможна обработка файлов препроцессором C (файл должен иметь расширение F95, F90, FOR или F). Можно использовать ассемблер (исходные тексты содержатся в файлах с расширением s).

Поддерживаются оба формата записи исходного текста программ на Фортране: свободный (файл должен иметь расширение f90 или f95) и фиксированный (файлы с расширениями f или for).

Ограничения LF95 приведены в табл. 11.12.

***Таблица 11.12.*** *Ограничения CVF6.6*

| Элемент | Максимальное значение |
|---|---|
| Размер программы | 4 Гбайт |
| Размер символьной строки | 65 000 байт |
| Размер блока ввода-вывода | 65 000 байт |
| Длина записи файла | 2 147 483 647 байт |
| Размер файла | 2 147 483 647 байт |
| Количество записей в файле прямого доступа | 2 147 483 647, деленное на значение спецификатора `RECL` |
| Вложенность конструкций `DO`, `CASE`, `IF` | 50 |
| Вложенность неявных циклов `DO` | 25 |
| Вложенность включаемых файлов | 16 |
| Размерность массивов | 7 |

# Компиляторы Pro Fortran фирмы Absoft

Компиляторы и инструментарий разработчика программ на языке Фортран предлагает фирма Absoft. Наборы программиста, которые предлагаются под названием Pro Fortran, имеются для трех основных семейств операционных систем: Windows 95/98/ME/NT/2000/XP, Linux и Mac OS. Они включают в свой состав оптимизирующие компиляторы F77, F90/95 и C/C++, интегрированную среду, ориентированную на программистов на языке Фортран. Стандартный пакет Pro Fortran Standard Release предназначен для однопро-

цессорных систем, работающих под управлением операционных систем Microsoft Windows. Пакет Pro FortranMP дополнительно содержит средства параллельного программирования для двухпроцессорных систем, в том числе препроцессор VAST-F/Parallel для автоматического распараллеливания программ. Программы, оптимизированные для двухпроцессорной конфигурации, могут выполняться только под управлением ОС WinNT/2000/XP. Третий вариант поставки — академическое издание Pro Fortran. Это Standard Release, который предлагается по специальной цене образовательным учреждениям. Компиляторы Absoft Pro Fortran являются 32-разрядными.

В состав каждого пакета входит компилятор Фортран 95, соответствующий расширенному стандарту Cray Research CF90 ISO/ANSI. Поддерживается Win32 API. Поддерживается также большое количество платформ.

Имеется также компилятор Фортран 77, ориентированный на рабочие станции и поддерживающий расширение Cray POINTER из VAX/VMS (это расширение стандарта Фортран 77, позволяющее использовать указатели при работе на платформах Cray и VAX с ОС VMS), Sun, HP и IBM MVS и Microsoft. Поддерживается Win32 API. Оба типа компиляторов поддерживают размер исполняемого файла до 2 Гбайт.

В состав пакетов включаются оптимизаторы кода для процессоров Intel Pentium 4 и AMD, а также библиотеки численных методов линейной алгебры (BLAS, LAPACK95, ATLAS, IMSL), оптимизированные для процессоров Pentium 2, 3, 4 и AMD. Более подробную информацию об этих библиотеках можно найти в *Приложении 3*. Компиляторы C/C++ предназначены для смешанного программирования на разных языках, в том числе и на Фортране. Имеется редактор исходных текстов программ на Фортране, C и C++. Есть отладчик, работающий на уровне исходного текста. Он дополнен функциями отладки динамических библиотек (DLL). Имеется профилировщик программ, позволяющий строить графическое представление затрат процессорного времени на выполнение различных фрагментов программы. Все компоненты доступны из интегрированной среды, но можно работать и "из командной строки".

Все компиляторы Absoft совместимы с MS VC++ 5.0 и 6.0. Имеются также средства разработки графического интерфейса программ в духе MS Windows и библиотеки двумерной и трехмерной графики (OpenGL, SuperPlot).

Пакеты для Mac OS X включают, кроме прочего, средства векторизации, распараллеливания и выполнения на кластерных системах, работающих на основе процессора G4.

Компиляторы Absoft Fortran 95 и FORTRAN 77 для платформы Intel — Linux совместимы с набором инструментов GNU: gcc, g77, ld, as и make. Оба компилятора поддерживают оптимизацию для процессоров AMD и Intel, а также конвертацию порядка двоичных разрядов Little-endian/Big-endian,

что упрощает переносимость программ с рабочих станций на персональные компьютеры. Многоязыковый отладчик Absoft Fx поддерживает работу с языками Фортран 77/90/95, C и ассемблером. В комплекте имеется графическая библиотека высокого уровня DISLIN, а также откомпилированные библиотеки BLAS, ATLAS и LAPACK90. Имеется также библиотека IMSL.

Компилятор Absoft Fortran 95 соответствует стандарту ANSI/ISO Фортран 95. Фортран Absoft совместим на уровне исходного кода с Фортраном DEC, Cray, SGI, Sun, HP и IBM.

Компилятор Absoft Фортран 77 для Linux является оптимизирующим. Поддерживается стандарт ANSI X3.9-1978/ISO 1539-1980, а также стандарты DoD MIL-STD 1753, FIPS 69-1 и POSIX. Поддерживаются наиболее популярные расширения VAX/VMS, IBM/VS, Cray и Sun FORTRAN. Имеется пробная версия с двухнедельным сроком действия.

Расширения языка включают поддержку типа Integer*8, функции типа COMPLEX*16, тригонометрические функции с аргументом, заданным в градусах, функции типа INTEGER*2 и т. д.

## Компилятор FortranPlus

Компилятор FortranPlus является продуктом фирмы NASoftware Ltd. Он поставляется вместе с интегрированной средой и существует для платформ Windows и Linux.

Через Интернет доступна бесплатная пробная версия компилятора для Linux и Windows. В состав пакета входит математическая библиотека, которая содержит подпрограммы вычислительной алгебры, сортировки, быстрого преобразования Фурье, статистические, подпрограммы численного интегрирования, оптимизации, интерполяции и аппроксимации, решения обыкновенных дифференциальных уравнений и другие.

## Отладчик dbx

Для отладки программ в среде операционной системы UNIX можно использовать интерактивный отладчик dbx. Сам по себе он имеет довольно примитивный (текстовый, строчный) 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, например, xde для ОС AIX, 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 пе-

ременной. Можно также проследить "судьбу" отдельно взятой переменной. Поддерживается работа с языками Фортран, С и Паскаль.

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). Компиляция выполняется с ключом отладки `-g`. В makefile, например, можно добавить строки:

```
.f.o: ; xlf -c -g $*.f
```

для Фортрана или для С:

```
.c.o: ; cc -c -g $*.c,
```

или применить команду компиляции с этим ключом:

```
cc -g foo.c -o foo
xlf -g foo.f -o foo
```

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, 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 и лишь затем обращается к исходному тексту программы. Файл core должен находиться в текущем каталоге. Если он не найден, 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.

Для того чтобы при работе с отладчиком использовать интерфейс Emacs, следует использовать команду этого редактора:

M-x dbx <Enter>,

после которой вводится имя исполняемого файла, подлежащего отладке. Emacs создает новое окно и буфер для интерфейса оболочки. Команды

в этом окне являются командами отладчика dbx. Во время выполнения Emacs отображает исходный текст в другом окне, трассировка выполнения программы отображается в нем же.

Все команды 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`.

## Точки останова

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.

Выражения в dbx аналогичны выражениям в языке C, за исключением того, что имеется различие между операцией деления с плавающей точкой / и

целочисленным делением `div`, как в Паскале. Отладчик dbx может выполнить дизассемблирование объектного кода.

Кроме отладчика dbx в среде ОС UNIX имеется отладчик adb, но он уступает отладчику dbx в простоте использования. Единственное, что может делать adb и не может dbx — менять исполняемый код программы (выполнять patch).

***Таблица 11.13.*** *Команды dbx*

| **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, этот ввод необходимо выполнить. Выполнение продолжается до наступления одного из следующих событий:<br>достижение точки останова;<br>одновременное нажатие клавиш <Ctrl>+<C>;<br>нормальное завершение работы программы;<br>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 |
| `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 |
| `status` | "Просьба" отладчику вывести список команд, которые он "запомнил" |
| `delete номер` | Удаление команды с заданным номером из списка команд отладчика |

***Таблица 11.13*** *(продолжение)*

| Выполнение программы и ее трассировка | |
|---|---|
| `catch` | Выполнить перехват указанных сигналов |
| `ignore` | Остановить перехват указанных сигналов |
| `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` | Вывести значение выражения.<br>Данная команда используется для вывода значений переменных, например:<br>`(dbx) print a`<br>`1.2387997473787516e-05`<br>`(dbx)`<br>Переменной может быть массив:<br>`(dbx) print y`<br>`(1)  1.0`<br>`(2)  2.0`<br>и простое выражение:<br>`(dbx) print y(i + 1) + 2`<br>`3.0`<br>`(dbx)` |
| `whatis имя` | Вывести описание указанной переменной или типа |
| `which` | Вывести имя внешнего блока, связанного с переменной |
| `whereis` | Вывести все символы, соответствующие указанному идентификатору |
| `assign` | Присвоить значение переменной |
| **Работа с подпрограммами** | |
| `where` | 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 |

***Таблица 11.13*** *(окончание)*

| Работа с подпрограммами | |
|---|---|
| `down` | Перейти вниз по стеку по направлению к точке останова |
| `up` | Перейти вверх по стеку по направлению к главной программе |
| `call процедура` | Вызвать указанную подпрограмму |
| `dump` | Вывести 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` | Вызвать редактор для редактирования текущего исходного файла |
| `file` | Изменить текущий исходный файл |
| `func` | Изменить текущую подпрограмму |
| `list строка` | Вывести строки исходного текста. Каждой строке исходного текста присваивается номер. Если диапазон строк не указан, они выводятся порциями по 10, начиная с самой первой. Можно указать и диапазон номеров строк. В emacs номер строки можно получить, щелкнув по строке и выполнив команду `M-x what-line` |
| `use` | Задать список каталогов для поиска файлов с исходными текстами |
| `/.../` | Выполнить поиск по направлению к концу файла в соответствии с регулярным выражением |
| `?...?` | Выполнить поиск по направлению к началу файла в соответствии с регулярным выражением |
| **Вспомогательные команды** | |
| `sh` | Передать команду оболочке |
| `alias` | Изменить имя команды dbx |
| `help` | Справка по командам |
| `source` | Вводить команды из внешнего файла |
| `quit` | Завершить работу с отладчиком |

# Отладчик gdb

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
```

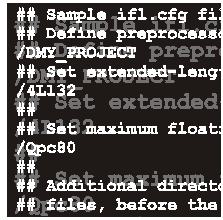
в среде ОС Linux.

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

# Вопросы и задания для самостоятельной работы

1. Проведите сравнение функциональных возможностей различных компиляторов.
2. Проведите сравнение функциональных возможностей различных отладчиков.
3. Проведите сравнение эффективности оптимизации различных компиляторов по быстродействию исполняемых кодов. В качестве тестового материала можно взять программу, выполняющую, например, матрично-векторные вычисления.

# Глава 12




# Фортран и объектно-ориентированное программирование

*Объектно-ориентированное программирование* (ООП) — это парадигма (парадигмой называют систему понятий, принцип, набор методов и приемов), которая позволяет сделать процесс разработки программного обеспечения более эффективным. Это особенно важно при создании сложных программ. Фортран 90 не является языком объектно-ориентированного программирования, но позволяет смоделировать основные концепции ООП. При этом сразу следует отметить, что в языках ООП соответствующие средства встроены и оптимизированы на уровне кода, и потому их применение значительно эффективнее, чем имитация средствами не ООП-языка.

Несмотря на то, что ООП завоевало прочные позиции почти во всех сферах деятельности, в области научно-технических расчетов оно используется реже, и это объясняется рядом причин. Часто прикладные программы среднего размера создаются для собственных нужд, их разработчики вполне комфортно чувствуют себя, используя "старые добрые" языки — Фортран 77 и C. Перевод сложной программы с Фортрана 77 на язык объектного программирования требует значительных усилий и затрат времени. Он означает не только изучение нового метода программирования, но и, как правило, нового языка. Такой язык объектно-ориентированного программирования, как C++, довольно сложен. Его изучение требует времени, которого, как правило, у научного работника или инженера нет. Существенными минусами современных языков объектно-ориентированного программирования являются недостаточно высокое быстродействие исполняемого кода, отсутствие стандартизации и стандартных классов для научных расчетов. C++ — мощный, гибкий и сложный язык. Фортран 90 более консервативен и стабилен. Он содержит большое количество встроенных функций, необходимых для научного программирования.

Фортран 90 позволяет использовать технологию ООП, ведь, вообще говоря, объектные программы можно писать на любом языке. Благодаря обратной совместимости Фортрана 90 с Фортраном 77 появляется возможность постепенного

перехода к новому стилю программирования без потери функциональности программного обеспечения. Согласно проекту, Фортран 2000 будет объектно-ориентированным языком. Данную главу при первом чтении книги можно пропустить.

# Объектно-ориентированное программирование

Развитие программирования идет по пути повышения *степени абстракции*, то есть возможности игнорировать детали выполнения основных операций. Стартовой точкой этого развития было программирование в машинных кодах, что соответствует низкому уровню абстракции. На смену программированию в машинных кодах пришло программирование на языках высокого уровня. Программа на языке высокого уровня состоит из операторов. Оператор соответствует обобщенной операции, состоящей из последовательности машинных команд. Размер исходного текста программы при этом значительно уменьшается, уменьшается вероятность ошибок, проще становится и процесс отладки.

Дальнейшее повышение уровня абстракции связано с введением подпрограмм — функций и процедур. Если подпрограмма уже написана и отлажена, можно отвлечься от деталей ее реализации, используя только ее интерфейс — имя, список параметров, возможно, глобальные переменные и, если надо, оператор вызова. Это дает возможность определять собственные обобщенные операции, объединяющие под общим именем набор операторов и предложений языка высокого уровня. Данная разновидность абстракции называется процедурной.

Следующим шагом стало появление производных типов данных. Это позволяет организовать информацию более естественным и оптимальным образом, учитывая особенности конкретной задачи. Это — абстракция данных.

Для обработки производных типов часто приходится использовать специально для этого предназначенные подпрограммы. Естественным шагом является объединение описаний типов данных с подпрограммами их обработки. Для этого можно использовать модули. В модулях несущественные для внешнего пользователя детали реализации скрыты. Это — модульное программирование.

При переходе на каждый последующий уровень абстракции упрощается структура программы, размер ее исходного текста уменьшается. Программа представляется меньшим числом блоков, уменьшается и количество связей между ними. Это увеличивает эффективность разработки, отладки и сопровождения сложных программ.

Объектно-ориентированное программирование является следующим шагом, который можно считать развитием идеи модульности. ООП позволяет соз-

давать хорошо структурированные, надежные и легко модифицируемые программы, его основная цель — повышение эффективности разработки сложных программ. Объектно-ориентированные языки позволяют эффективно использовать преимущества данного подхода. Среди них Simula, C++, Java и другие. Реализация объектно-ориентированного подхода возможна и в обычных языках — таких, например, как C или Фортран. Наоборот, на объектно-ориентированном языке можно написать программу без использования ООП. Многие задачи могут быть успешно решены и в рамках традиционного подхода. Это, прежде всего, задачи, требующие больших вычислений. В этом случае решающим фактором является высокое быстродействие программы, недостижимое в объектных языках.

В основе объектно-ориентированного подхода лежит наблюдение, что программа представляет собой описание действий, выполняемых над различными объектами. Это могут быть, например, записи в базах данных или совокупности числовых значений. В "традиционном" программировании (традиционным мы условно называем программирование на императивных языках) изменение данных или методов их обработки часто приводит к необходимости значительного изменения программы, а это увеличивает вероятность ошибок. Как следствие, возрастает время, необходимое для отладки программы. Использование ООП позволяет выйти из такой ситуации с минимальными потерями, сводя необходимую модификацию программы к ее расширению и дополнению. ООП не является панацеей от всех бед, но его ценность как технологии программирования несомненна. Изучение методов объектно-ориентированного программирования — непростая задача, однако применение ООП упрощает разработку и отладку сложных программ.

Основными составляющими ООП являются:

- *инкапсуляция*;
- *наследование*;
- *полиморфизм*.

Краткий обзор этих составляющих дается ниже.

## Классы и объекты

Базовыми в объектно-ориентированном программировании являются понятия *класса* и *объекта*. В отличие от модуля, класс жестко объединяет типы данных и функции их обработки. Класс используется только через свой интерфейс, т. е. методы, свойства, поля и события, поэтому детали его реализации скрыты от внешнего пользователя (принцип "черного ящика"). Класс содержит тип данных, который определяется программистом (похоже на обычный для Фортрана производный тип).

Класс является описанием объекта. Экземпляр объекта данного типа или сущность (instance) создается по этому описанию и состоит из набора опе-

раций над скрытыми от внешнего мира данными. Объекты предоставляют однородный интерфейс к множеству компонентов программной системы.

Объект обычно реализуется с помощью множества подобъектов, подобно тому, как при программировании "сверху-вниз" процедура реализуется с помощью набора более мелких процедур.

Структуры данных, использующиеся при программировании "сверху-вниз" сравнительно просты и остаются инвариантными при последовательной детализации программы. Процедуры при этом просто заменяются более детализованными версиями. Обычные языки структурного программирования вполне соответствуют такой методике программирования. При использовании сложных структур данных требуется одновременная модификация данных и функций их обработки, а в этом случае эффективность ООП велика.

Работа над программой с использованием ООП начинается с разработки объектной модели проектируемой системы. При разработке объектной модели выполняется следующая последовательность шагов.

1. Изучение предметной области и определение множества объектов реального мира на заданном уровне абстракции.
2. Определение требуемого поведения (семантики) объектов.
3. Определение отношений между объектами.
4. Реализация объектов.

Если, например, речь идет об отображении произвольной гладкой поверхности, можно считать, что она может быть приближенно представлена набором плоских элементов — треугольников. В этом случае объектами естественно считать треугольники. Треугольники могут различаться размерами сторон и углами, но они имеют определенное сходство, позволяющее объединить все треугольники в общий класс.

Объект характеризуется набором своих *атрибутов*. Атрибуты могут быть *постоянными* (статическими) или *переменными* (динамическими). Статические атрибуты характеризуют объект в его классе, т. е. доступны через класс (например, количество вершин многоугольника может быть описано прямо в классе). Текущие значения динамических атрибутов характеризуют состояние объекта, т. е. доступны только через объект и описаны в нем. При изменении значений этих атрибутов изменяется и состояние объекта (например, значения углов при вершинах многоугольника). Совокупность начальных значений этих атрибутов задает начальное состояние объекта.

Признаки, позволяющие объединить объекты реального мира в класс, определяются, прежде всего, контекстом задачи. Прямоугольники и прямоугольные треугольники могут характеризоваться одинаковым набором атрибутов — длинами двух смежных сторон, но при этом они будут разными геометрическими объектами.

Объединение объектов в классы позволяет рассмотреть задачу в более общей постановке. Класс имеет имя, которое относится ко всем объектам этого класса. Кроме того, в классе вводятся имена атрибутов, определенных для объектов.

Объект "знает", как решать определенные задачи, то есть располагает *методами* решения. Методам объекта доступны его *поля* (открытые данные). Программа, написанная с использованием ООП, состоит из объектов, которые могут взаимодействовать между собой.

## Инкапсуляция

*Инкапсуляция* не является исключительной особенностью ООП, она применяется и в языках структурного и модульного программирования. В ООП концепция инкапсуляции получила свое логическое завершение и является одним из базовых принципов объектного программирования.

Инкапсуляция основана на идее о том, что объект должен скрывать в себе детали, несущественные для его использования. Ее можно считать обобщением концепции подпрограмм. В традиционном программировании вынужденное использование глобальных переменных делает доступными все данные, связанные с этими переменными. В результате, процедура может ошибочно обратиться к такой переменной, которая связана с данными, не предназначенными для обработки в этой процедуре. "Жесткое" связывание данных и процедур их обработки в одном объекте позволяет избежать неприятностей такого рода. Инкапсуляция позволяет сгруппировать необходимые операции вокруг определенного типа данных. Таким образом, область действия этих операций ограничивается типом данных, размещенных в классе, что позволяет избежать ошибок неправильного обращения к процедурам обработки.

Инкапсуляция связана также с концепцией сокрытия информации. С ее помощью можно скрыть детали вычисления результата, допуская использование для обработки данных только хорошо определенные операции. Данные строкового типа, например, нельзя умножать друг на друга, но можно выделять подстроки и т. д.

> ***Определение***
>
> Инкапсуляцией называется объединение данных с функциями их обработки, одновременно с сокрытием несущественной для использования данных информации.

Описание предметной области обычно выглядит более естественно при использовании соответствующих ей типов данных, поэтому программисты определяют собственные *абстрактные типы данных*. Так например, можно определить тип данных, представляющий векторы. Подобный абстрактный тип

должен быть связан с описанием тех операций, которые можно применять к векторам. Это могут быть вычисление нормы, скалярного и векторного произведения и т. д.

Некоторые операции определены для операндов нескольких стандартных типов. Это свойство, как известно, называется перегрузкой операции, оно расширяет концепцию абстрактных типов данных. Операция умножения, например, может быть определена для простых целых и вещественных типов, а также для массивов. Перегрузка позволяет писать более короткие и логически организованные программы, благодаря тому, что в этом случае конструируется обобщенная операция, поведение которой определяется типом аргументов в момент компиляции.

Определение класса объединяет код допустимых операций с абстрактным типом данных и скрывает детали их реализации. Инкапсулированный код можно безопасно модифицировать, не изменяя при этом те фрагменты программы, в которых он используется. Образно говоря, инкапсуляция создает барьер, отделяющий часть кода программы от внешнего мира. Внешним миром в этом случае является остальная часть программы, она взаимодействует с инкапсулированным кодом через его интерфейс. До тех пор, пока этот интерфейс остается неизменным, неизменным остается и код программы, даже если программист изменил инкапсулированный код. Это упрощает процесс модификации программы. Интерфейс, с помощью которого инкапсулированный код взаимодействует с внешним миром, называется общим (public). Имеется еще один вариант взаимодействия — с кодом-"наследником" (о наследовании речь пойдет позже). Для такого взаимодействия используется защищенный (protected) интерфейс. Например, в объектно-ориентированном языке C++ программист явно описывает, какие компоненты объекта являются общими, какие защищенными, а какие — скрытыми (private), видимыми только в описании объекта.

Иногда говорят, что классы — это определяемые пользователем типы данных. Однако они являются более широким понятием, чем абстрактные типы данных и, в частности, допускают наследование.

## Наследование

*Наследование* позволяет определять новые объекты, используя свойства прежних, дополняя или изменяя их. Объект-наследник получает все поля и методы объекта-"родителя", к которым он может добавить свои собственные поля и методы или заменить ("перекрыть") их своими методами.

Из методов наследника можно вызывать методы родителя. Для создания наследника не требуется исходный текст объекта-родителя. Объект-родитель может находиться в составе уже оттранслированного модуля.

Привлекательность наследования заключается в том, что если некий объект уже был определен и отлажен, он может быть использован и в других про-

граммах. При этом может оказаться, что новая задача отличается от предыдущей, и возникает необходимость некоторой модификации данных и методов их обработки. Программисту приходится решать дилемму — создавать объект заново или использовать результаты предыдущей работы, применяя механизм наследования. Первый путь менее эффективен, так как требует дополнительных затрат времени на отладку и тестирование. Во втором случае часть этой работы оказывается выполненной, что сокращает время на разработку новой программы. Программист при этом может и не знать деталей реализации объекта-родителя.

Пример реализации наследования на некотором условном языке приводится в листинге 12.1.

**Листинг 12.1. Реализация наследования**

```
class FATHER =
private
val p = значение
public
fun f(x) = ... g(...) ...
fun g(y) = исходное определение
end;

class KID = extend FATHER with
private
val q = значение
public
fun g(y) = новое определение
end;
```

Связь между *виртуальными* методами и вызывающими их процедурами устанавливается не во время трансляции (это называется *ранним связыванием*), а во время выполнения программы (*позднее связывание*). Позднее связывание является одним из способов реализации полиморфизма. В этом случае полиморфизм заключается в том, что одно и то же обращение к методу порождает разные действия, в зависимости от типа объекта, вызвавшего метод.

Инициализация экземпляра объекта выполняется с помощью специального метода — *конструктора*. Конструктор присваивает полям объекта начальные значения и выполняет другие действия по инициализации объекта. Если у экземпляра объекта имеются виртуальные методы, то эти действия включают и создание указателя на таблицу виртуальных методов, которая в дальнейшем используется для поиска нужного метода. При вызове виртуального

метода по его имени определяется адрес, а затем по этому адресу передается управление.

У каждого объектного типа имеется своя собственная таблица виртуальных методов, что и позволяет одному и тому же оператору вызывать разные процедуры. Если имеется несколько экземпляров объектов одного типа, то недостаточно вызвать конструктор для одного из них, а затем просто скопировать этот экземпляр во все остальные. Каждый объект должен иметь свой собственный конструктор, который вызывается для каждого экземпляра.

Действия, обратные действиям конструктора, выполняет *деструктор*. Конструктор и деструктор могут быть "пустыми", то есть не содержать операторов.

Переменные объектного типа могут быть динамическими и размещаться в памяти только во время их использования.

Одна из форм наследования называется *подтипизацией*. Подтипизация заключается в следующем: если A является подтипом B, любое выражение типа A можно использовать в любом контексте, подразумевающем использование выражения типа B.

## Полиморфизм

*Полиморфизм* заключается в том, что в зависимости от типа объекта одно и то же имя метода может соответствовать различным действиям. Решение о том, какая операция должна быть выполнена в конкретной ситуации, принимается во время трансляции, если тип объекта известен заранее, или во время выполнения программы, если в ней каким-либо образом уточняется тип объекта.

## Объектная структура программы

Объектная структура программы часто позволяет использовать при ее разработке те же понятия, что и предметная область, послужившая источником задачи. Вместе с тем, трудоемкость разработки объектной модели может быть достаточно большой, за счет нового этапа — этапа разработки иерархии классов. Если при решении задачи нет необходимости строить иерархию классов, методы ООП могут оказаться излишними. В этом случае достаточно применить обычную модульную технологию программирования.

# Реализация основных принципов ООП в программах на языке Фортран 90

Рассмотрим возможность моделирования основных методов объектно-ориентированного программирования средствами Фортрана 90.

# Инкапсуляция данных с помощью массивов

Инкапсуляция данных в Фортране может быть реализована с помощью массивов. Рассмотрим пример подпрограммы, написанной на Фортране 77 и предназначенной для обработки одномерного вещественного массива. Пусть интерфейс этой подпрограммы организован следующим образом:

```
SUBROUTINE ARPROC(FDAT, TRANS, TMP, RECPROC, ILEN, NDAT, NELEM)
 INTEGER TRANS, ILEN, NDAT, NELEM
 REAL FDAT(NDAT)
 COMPLEX RECPROC(NELEM), TMP(NELEM)
...
```

В этом примере `FDAT` является входным параметром — массивом, который подвергается обработке, выходной параметр `TMP` является рабочим массивом, `RECPROC` — вспомогательный массив, `ILEN` является входным параметром, `NDAT` — размер массива `FDAT` и `NELEM` — размер остальных массивов.

Пусть процедура `ARPROC` предназначена для работы с данными переменной длины, и предполагается ее раздельная компиляция. Фортран 77 не имеет средств динамического размещения данных, поэтому массивы `TMP` и `RECPROC` описываются в главной программе и передаются в процедуру. Если вызов `ARPROC` располагается в других процедурах, массивы должны передаваться в эту процедуру "по цепочке" аргументов или через общий блок:

```
PROGRAM MAIN
 INTEGER TRANS, ILEN, NDAT, NELEM
 PARAMETER(ILEN = 10, NDAT = 21)
 REAL FDAT(NDAT)
 COMPLEX RECPROC(NELEM), TMP(NELEM)
 TRANS = 0
 CALL ARPROC(FDAT, TRANS, TMP, RECPROC, ILEN, NDAT, NELEM)
 STOP
END
```

Задачей инкапсуляции данных является приведение интерфейса процедуры `ARPROC` к виду:

```
CALL ARPROC(FDAT, TRANS),
```

где вспомогательные массивы и переменные, которые требуются только процедуре `ARPROC`, спрятаны внутри нее, а в остальной части программы не видны.

Одним из механизмов инкапсуляции является использование автоматических массивов, которые создаются при входе в процедуру и удаляются при выходе из нее. Имена таких массивов можно удалить из списка аргументов процедуры:

```
SUBROUTINE ARPROC(FDAT, TRANS, RECPROC, ILEN, NDAT, NELEM)
 INTEGER TRANS, ILEN, NDAT, NELEM
```

```
 REAL FDAT(NDAT)
 COMPLEX RECPROC(NELEM)
 COMPLEX, DIMENSION(NELEM) :: TMP
...
END SUBROUTINE ARPROC
```

Здесь автоматическим массивом является массив TMP.

Другой механизм инкапсуляции в Фортране 90 обеспечивается динамическими (размещаемыми) массивами. Они отличаются от автоматических тем, что программист полностью контролирует создание и удаление динамического массива. Размещение выполняется оператором ALLOCATE, а явное удаление — оператором DEALLOCATE. Такие массивы с атрибутом SAVE при выходе из процедуры не удаляются.

Предположим, что в промежутках между обращениями к ARPROC массив RECPROC не изменяется. В этом случае массив можно удалить из списка аргументов, явно размещая его оператором ALLOCATE:

```
SUBROUTINE ARPROC(FDAT, TRANS, ILEN, NDAT, NELEM)
 INTEGER TRANS, ILEN, NDAT, NELEM
 REAL FDAT(NDAT)
 COMPLEX, DIMENSION(:), ALLOCATABLE, SAVE :: RECPROC
 COMPLEX, DIMENSION(NELEM) :: TMP
 IF(TRANS == 0) ALLOCATE(RECPROC(NELEM))
...
END SUBROUTINE ARPROC
```

Массив в Фортране 90 содержит не только данные, но и информацию о своем размере. В Фортране 77 информацию о размере содержат только символьные массивы. В Фортране 90 размер массивов с подразумеваемой формой можно определить с помощью функции SIZE. Данное обстоятельство можно считать третьим механизмом инкапсуляции данных. В этом случае из списка аргументов можно удалить параметры NDAT и NELEM, определяя их в самой процедуре. Описание автоматического массива TMP следует изменить с помощью функции SIZE:

```
SUBROUTINE ARPROC(FDAT, TRANS, ILEN)
 REAL, DIMENSION(:) :: FDAT
 INTEGER :: TRANS, ILEN, NDAT, NELEM
 COMPLEX, DIMENSION(:), ALLOCATABLE, SAVE :: RECPROC
 COMPLEX, DIMENSION(SIZE(FDAT) / 2) :: TMP
 NDAT = SIZE(FDAT)
 NELEM = NDAT / 2
 IF(TRANS == 0) ALLOCATE(RECPROC(NELEM))
```

```
...
END SUBROUTINE ARPROC
```

Для того чтобы использовать массивы с подразумеваемой формой, компилятор должен "знать" тип фактических аргументов процедуры в момент ее вызова. Одним из способов получения такой информации является использование оператора INTERFACE (листинг 12.2).

**Листинг 12.2. Использование оператора интерфейса в примере с инкапсуляцией**

```
PROGRAM MAIN
 INTEGER :: TRANS
 INTEGER, PARAMETER :: ILEN = 11, NX = 20
 REAL, DIMENSION(NDAT) :: FDAT
 INTERFACE
       SUBROUTINE ARPROC(A, I, J)
              REAL, DIMENSION(:) :: A
              INTEGER :: I, J
       END SUBROUTINE ARPROC
 END INTERFACE
 TRANS = 0
 CALL ARPROC(FDAT, TRANS, ILEN)
 STOP
END
```

Дополнительным преимуществом использования явного описания интерфейса является то, что в этом случае компилятор выполняет проверку соответствия фактических аргументов процедуры числу и типу параметров, заданным в интерфейсе. Если аргумент TRANS случайно пропадет из списка параметров:

```
CALL ARPROC(FDAT, ILEN),
```

компилятор определит это.

От возможного источника ошибок, связанных с несоответствием описаний данных в блоке INTERFACE и фактических данных в процедуре, можно избавиться, включив процедуру в состав модуля. В этом случае компилятор создает интерфейсный блок автоматически:

```
MODULE ARPROC_MODULE
 CONTAINS
       SUBROUTINE ARPROC(FDAT, TRANS, ILEN)
              REAL, DIMENSION(:) :: FDAT
              INTEGER :: TRANS, ILEN, NDAT, NELEM
```

```
            COMPLEX, DIMENSION(:), ALLOCATABLE, SAVE :: RECPROC
            COMPLEX, DIMENSION(SIZE(FDAT) / 2) :: TMP
            NDAT = SIZE(FDAT)
            NELEM = NDAT / 2
            IF(TRANS == 0) ALLOCATE(RECPROC(NELEM))
…
      END SUBROUTINE ARPROC
END ARPROC_MODULE

PROGRAM MAIN
 USE ARPROC_MODULE
 INTEGER :: TRANS = 0
 INTEGER, PARAMETER :: ILEN = 11, NX = 20
 REAL, DIMENSION(NDAT) :: FDAT
 CALL ARPROC(FDAT, TRANS, ILEN)
 STOP
END
```

Фортран 90 содержит и другие средства безопасного программирования. Среди них предложение `IMPLICIT NONE`, а также атрибут `INTENT` для аргументов. Если аргументы `ARPROC` описаны следующим образом:

```
SUBROUTINE ARPROC(FDAT, TRANS, ILEN)
 IMPLICIT NONE
 REAL, DIMENSION(:), INTENT(INOUT) :: FDAT
 INTEGER, INTENT(IN) :: TRANS, ILEN,
```

то модификация переменных `TRANS` и `ILEN` в процедуре будет запрещена, поскольку они описаны как входные.

Инкапсуляция данных упрощает и совместную работу нескольких программистов над одним проектом. Подпрограмма `ARPROC` в нашем примере имеет простой интерфейс, вероятность изменения которого невелика даже в случае изменения ее кода. Пользователям этой процедуры менять код своих программ, скорее всего, не придется.

Можно упростить интерфейс еще больше, если заметить, что параметр длины `ILEN` требуется только во время инициализации вспомогательных массивов. При последующих вызовах использование другого значения `ILEN` без повторной инициализации массивов может привести к ошибке. Фортран 90 поддерживает использование необязательных (`OPTIONAL`) аргументов и функции `PRESENT`, позволяющей определить, переданы они в подпрограмму или нет. Можно сохранить параметр `ILEN`, переданный при инициализации, тогда его передача при последующих обращениях не потребуется. Кроме

того, можно ввести проверку значения сохраненного параметра ILEN с тем, чтобы не допустить некорректной инициализации массивов:

```
SUBROUTINE ARPROC(FDAT, TRANS, ILEN)
 INTEGER, INTENT(IN), OPTIONAL :: ILEN
 INTEGER, SAVE :: SAVED_ILEN = - 1
 IF(TRANS == 0) THEN
       IF(.NOT.PRESENT(ILEN)) THEN
              PRINT *, 'ЗНАЧЕНИЕ ILEN НЕ ОПРЕДЕЛЕНО ПРИ ИНИЦИАЛИЗАЦИИ!'
              STOP
       END IF
              SAVED_ILEN = ILEN
 ELSE
       IF(SAVED_ILEN < 0) THEN
              PRINT *, 'ИНИЦИАЛИЗАЦИЯ НЕ ВЫПОЛНЕНА!'
              STOP
       END IF
 END IF
...
```

В следующей версии подпрограммы процедура LIB_ARPROC предназначена для ссылки на библиотечную подпрограмму, в которой либо недоступен исходный текст, либо недопустимо его изменение:

```
SUBROUTINE ARPROC(FDAT, TRANS, ILEN)
 IMPLICIT NONE
 REAL, DIMENSION(:), INTENT(INOUT) :: FDAT
 INTEGER, INTENT(IN) :: TRANS
 INTEGER, INTENT(IN), OPTIONAL :: ILEN
 INTEGER :: NDAT, NELEM, IERR
 INTEGER, SAVE :: SAVED_ILEN = -1
 COMPLEX, DIMENSION(:), ALLOCATABLE, SAVE :: RECPROC
 COMPLEX, DIMENSION(SIZE(FDAT) / 2) :: TMP
 NDAT = SIZE(FDAT)
 NELEM = NDAT / 2
 IF(TRANS == 0) THEN
       IF(.NOT.PRESENT(ILEN)) THEN
              PRINT *, 'ЗНАЧЕНИЕ ILEN НЕ ОПРЕДЕЛЕНО ПРИ ИНИЦИАЛИЗАЦИИ!'
              STOP
       END IF
       IF(ILEN < 0) THEN
```

```
            PRINT *, 'ЗНАЧЕНИЕ ILEN ДОЛЖНО БЫТЬ НЕОТРИЦАТЕЛЬНЫМ!'
            STOP
      END IF
      SAVED_ILEN = ILEN
      IF(ALLOCATED(RECPROC)) DEALLOCATE(RECPROC)
      ALLOCATE(RECPROC(NELEM), STAT = IERR)
      IF(IERR /= 0) THEN
            PRINT *, 'ОШИБКА РАЗМЕЩЕНИЯ МАССИВА RECPROC'
            STOP
      END IF
 ELSE
      IF(SAVED_ILEN < 0) THEN
            PRINT *, 'ИНИЦИАЛИЗАЦИЯ НЕ ВЫПОЛНЕНА!'
            STOP
      END IF
 END IF
 CALL LIB_ARPROC(FDAT, TMP, TRANS, RECPROC, SAVED_ILEN, NDAT, NELEM)
END SUBROUTINE ARPROC
```

Вызов процедуры при инициализации имеет вид:

```
CALL ARPROC(FDAT, TRANS, ILEN),
```

а при последующих обращениях:

```
CALL ARPROC(FDAT, TRANS)
```

Таким образом, две различные операции — инициализацию и обработку массива — можно объединить в одной процедуре.

## Перегрузка функций

Продолжая развивать рассмотренный выше пример, допустим обработку массивов разных типов — вещественных, комплексных, одномерных и двумерных. В Фортране 77 для каждого из этих вариантов пришлось бы использовать свою процедуру. Механизм родовых функций позволяет использовать для работы со всеми этими типами одно имя, например, `ARPROC`. Компилятор автоматически будет выбирать правильный вариант процедуры в соответствии с типом фактически используемых аргументов.

Для обработки одномерного вещественного массива интерфейс процедуры может иметь вид:

```
SUBROUTINE ARPROC1(FDAT, TRANS)
 REAL, DIMENSION(:), INTENT(INOUT) :: FDAT
 INTEGER, INTENT(IN) :: TRANS
```

В случае обработки двумерного вещественного массива:

```
SUBROUTINE ARPROC2(FDAT, TRANS)
 REAL, DIMENSION(:, :), INTENT(INOUT) :: FDAT
 INTEGER, INTENT(IN) :: TRANS
```

Если обе эти процедуры разместить в одном модуле, можно определить родовую функцию ARPROC:

```
INTERFACE ARPROC
 MODULE PROCEDURE ARPROC1
 MODULE PROCEDURE ARPROC2
END INTERFACE
```

В главной программе, использующей модуль, оператор:

```
CALL ARPROC(FDAT, TRANS)
```

будет вызывать процедуру ARPROC1, если массив FDAT — вещественный одномерный массив, или ARPROC2, если FDAT — вещественный двумерный массив. В любом другом случае будет выводиться сообщение об ошибке компиляции.

Аналогично можно включить в общий интерфейс все прочие процедуры обработки массива, защитив тем самым программиста от ошибок вызова не той процедуры. При необходимости можно сделать специфические имена недоступными:

```
PRIVATE :: ARPROC1, ARPROC2
```

Это описание должно содержаться в модуле, тогда специфические имена можно независимо использовать в других модулях. Перегрузка функции называется иногда ad hoc-полиморфизмом.

## Производные типы, классы и объекты

Фортран 90 допускает определение пользовательских (производных или абстрактных) типов. Рассмотрим пример разработки интерфейса подпрограммы расчета движения заряженных частиц в электрическом поле:

```
SUBROUTINE CHARGE_MOVES(PARTICLES, FIELD, QBM, DT,&
& NP, DIMPHASE, NPARTS, NDAT)
 INTEGER NP, DIMPHASE, NPARTS, NDAT
 REAL QBM, DT, PARTICLES(DIMPHASE, NPARTS), FIELD(3, NDAT)
…
 RETURN
END
```

Здесь PARTICLES — массив координат и скоростей частиц, FIELD — массив значений электрического поля, DIMPHASE — размерность фазового пространства, NPARTS — максимально допустимое число частиц и NDAT — размер мас-

сива электрического поля. В Фортране 90 значения параметров DIMPHASE, NPARTS и NDAT в процедуру можно не передавать, так как они определяются с помощью функции SIZE, если PARTICLES и FIELD являются массивами с подразумеваемой формой. Параметр NP — это фактическое количество частиц в массиве PARTICLES, QBM — отношение электрического заряда к массе частицы, DT — шаг интегрирования уравнений движения по времени. Все скалярные величины, кроме шага по времени, описывают группу заряженных частиц и передаются вместе, если частицы обрабатываются одной процедурой. Совместное хранение этих величин можно организовать, определив производный тип PARTS_DESCRIPTOR:

```
TYPE PARTS_DESCRIPTOR
 INTEGER :: NUMBER_OF_PARTICLES
 REAL :: CHARGE, CHARGE_TO_MASS
END TYPE PARTS_DESCRIPTOR
```

Затем могут быть определены переменные производного типа, соответствующие разным сортам заряженных частиц:

```
TYPE(PARTS_DESCRIPTOR) :: ELECTRONS, IONS
```

Доступ к компонентам производного типа осуществляется с помощью символа %, например:

```
ELECTRONS%NUMBER_OF_PARTICLES = 250
ELECTRONS%CHARGE = 1.0
```

Определение производного типа следует разместить в секции описаний модуля, который содержит описание процедуры CHARGE_MOVING. Механизм производных типов Фортрана 90 позволил упростить интерфейс вызова процедуры:

```
CALL CHARGE_MOVING(PARTICLES, FIELD, ELECTRONS, DT)
```

Рассмотрим еще один пример. Для работы с комплексными значениями определим производный тип COMPLEXP и процедуры, реализующие необходимые операции. Комплексный тип в Фортране имеется, поэтому данный пример приводится только для иллюстрации:

```
TYPE COMPLEXP
 REAL :: REALP, IMAGP
END TYPE COMPLEXP
```

Описание комплексных переменных и присвоение им значений выглядит следующим образом:

```
TYPE(COMPLEXP) :: A, B
A%REALP = 1.14
A%IMAGP = 1.3
```

Функция MULTP предназначена для умножения значений типа COMPLEXP:

```
TYPE(COMPLEXP) FUNCTION MULTP(A, B)
```

```
 TYPE(COMPLEXP), INTENT(IN) :: A, B
 MULTP%REALP = A%REALP * B%REALP - A%IMAGP * B%IMAGP
 MULTP%IMAGP = A%REALP * B%IMAGP + A%IMAGP * B%REALP
END FUNCTION MULTP
```

Описание производного типа и процедур для работы с ним следует разместить в одном модуле (листинг 12.3).

**Листинг 12.3. Модуль комплексной арифметики**

```
MODULE COMPLEXP_MODULE
 TYPE COMPLEXP
       REAL :: REALP, IMAGP
 END TYPE COMPLEXP
 CONTAINS
       TYPE(COMPLEXP) FUNCTION MULTP(A, B)
             TYPE(COMPLEXP), INTENT(IN) :: A, B
             MULTP%REALP = A%REALP * B%REALP - A%IMAGP * B%IMAGP
             MULTP%IMAGP = A%REALP * B%IMAGP + A%IMAGP * B%REALP
       END FUNCTION MULTP
END MODULE COMPLEXP_MODULE
```

Программа, демонстрирующая умножение двух чисел типа `COMPLEXP`, приведена в листинге 12.4.

**Листинг 12.4. Программа, в которой используется модуль комплексной арифметики**

```
PROGRAM MAIN
 USE COMPLEXP_MODULE
 TYPE(COMPLEXP) :: A, B, C
 A%REALP = 1. ; A%IMAGP = -1.
 B%REALP = -1. ; B%IMAGP = 2.
 C = MULTP(A, B)
 PRINT *, 'C = ', C%REALP, C%IMAGP
 STOP
END PROGRAM MAIN
```

Размещение описания типа и процедур для работы с ним в отдельном модуле можно считать одним из вариантов инкапсуляции.

Можно инкапсулировать и отдельные компоненты производного типа, что является обычной практикой объектно-ориентированного программирова-

ния. В этом случае, если модуль используется в другой программной единице, можно определить тип COMPLEXP, но сделать так, чтобы отдельные компоненты, такие как A%REALP или A%IMAGP, были недоступны. В вышеприведенном примере доступ к отдельным компонентам осуществляется при инициализации данных и при печати результата умножения. Если компоненты инкапсулированы, модуль придется дополнить. Инкапсуляция обеспечивается добавлением атрибута PRIVATE в определение производного типа:

```
TYPE COMPLEXP
 PRIVATE
 REAL :: REALP, IMAGP
END TYPE COMPLEXP
```

Инициализация переменной типа COMPLEXP выполняется процедурой:

```
SUBROUTINE INITP(A, REALP, IMAGP)
 TYPE(COMPLEXP), INTENT(OUT) :: A
 REAL, INTENT(IN) :: REALP, IMAGP
 A%REALP = REALP
 A%IMAGP = IMAGP
END SUBROUTINE INITP
```

Процедура вывода комплексных значений:

```
SUBROUTINE DISPLAYP(A, C)
 TYPE(COMPLEXP), INTENT(IN) :: A
 CHARACTER * (*), INTENT(IN) :: C
 PRINT *, C, A%REALP, A%IMAGP
END SUBROUTINE DISPLAYP
```

Пример использования модуля комплексной арифметики приведен в листинге 12.5.

**Листинг 12.5. Пример использования модуля комплексной арифметики**

```
PROGRAM MAIN
 USE COMPLEXP_MODULE
 TYPE(COMPLEXP) :: A, B, C
 CALL INITP(A, 11.2, -21.31)
 CALL INITP(B, 1.22, 2.11)
 C = MULTP(A, B)
 CALL DISPLAYP(C, 'C = ')
 STOP
END PROGRAM MAIN
```

Наградой за наши усилия является то, что любые изменения внутреннего представления типа COMPLEXP будут ограничены модулем комплексной арифметики.

В Фортране 90 перегрузка операций выполняется с помощью оператора INTERFACE, который располагается в секции описаний модуля:

```
INTERFACE OPERATOR(*)
 MODULE PROCEDURE MULTP
END INTERFACE
```

В этом случае операции умножения * сопоставлено имя MULTP, а в главной программе можно умножать два числа типа COMPLEXP обычным образом:

```
C = A * B
```

Если в модуль добавить описание

```
PRIVATE :: MULTP
```

имя процедуры MULTP окажется недоступным.

Созданный нами модуль является аналогом *класса*. Наш класс содержит определение одноименного производного типа, а также процедуры, действующие на этом классе — методы (функции-члены класса). Компоненты производного типа общедоступны и поэтому соответствуют полям класса, а глобальные данные модуля соответствуют статическим полям класса. Фактическая переменная типа COMPLEXP соответствует объекту.

Следуя традициям программирования на языке C++, будем размещать аргумент производного типа на первом месте в списке параметров процедур модуля. Перегрузим функцию инициализации INITP и присвоим ей имя NEW с атрибутом доступа PUBLIC, сделав специфическое имя INITP скрытым. Еще один пример использования модуля комплексной арифметики дан в листинге 12.6.

**Листинг 12.6. Пример использования модуля комплексной арифметики**

```
PROGRAM MAIN
 USE COMPLEXP_CLASS
 TYPE(COMPLEXP) :: A, B, C
 CALL NEW(A, 1., 1.)
 CALL NEW(B, 1., 2.)
 C = A * B
 CALL DISPLAYP(C, 'C = ')
 STOP
END PROGRAM MAIN
```

## Наследование

Фортран 90 допускает различные способы моделирования наследования. В наиболее общем случае один класс используется другим. Такая форма наследования называется *композицией классов*. Создадим новый класс, поле которого является одномерным массивом типа COMPLEXP:

```
PROGRAM MAIN
 USE COMPLEXP_CLASS
 INTEGER, PARAMETER :: NDAT = 100
 TYPE(COMPLEXP), DIMENSION(NDAT) :: FDAT
```

Элементы этого типа можно инициализировать с помощью процедуры NEW из модуля COMPLEXP_CLASS:

```
DO I = 1, NDAT
 CALL NEW(FDAT(I), 0., 0.)
END DO
```

Процедура INITCARRAY преобразует два вещественных массива, содержащих вещественные и мнимые части комплексных значений, в один массив типа COMPLEXP. Разместим эту процедуру в модуле COMPLEX_ARRAY_CLASS (листинг 12.7).

**Листинг 12.7. Модуль COMPLEX_ARRAY_CLASS**

```
MODULE COMPLEX_ARRAY_CLASS
 USE COMPLEXP_CLASS
 CONTAINS
       SUBROUTINE INITCARRAY(THIS, REALP, IMAGP)
              TYPE(COMPLEXP), DIMENSION(:), INTENT(OUT) :: THIS
              REAL, DIMENSION(:), INTENT(IN) :: REALP, IMAGP
              DO I = 1, SIZE(THIS)
                     CALL NEW(THIS(I), REALP(I), IMAGP(I))
              END DO
              END SUBROUTINE INITCARRAY
END MODULE COMPLEX_ARRAY_CLASS
```

Модуль COMPLEXP_CLASS в этом примере используется ("наследуется") модулем COMPLEX_ARRAY_CLASS. Все общедоступные элементы первого модуля, который соответствует базовому классу, доступны второму модулю, то есть производному классу. Следует также перегрузить процедуру NEW:

```
INTERFACE NEW
 MODULE PROCEDURE INITCARRAY
END INTERFACE
```

В результате перегрузки, если аргументом процедуры NEW будет скалярная переменная типа COMPLEXP, фактически будет выполняться процедура INITP, а если ее аргументом является массив типа COMPLEXP, будет выполняться INITCARRAY. Пример программы, использующей модуль COMPLEX_ARRAY_CLASS, приведен в листинге 12.8.

**Листинг 12.8. Пример использования модуля комплексной арифметики**

```
PROGRAM MAIN
 USE COMPLEX_ARRAY_CLASS
 INTEGER, PARAMETER :: NDAT = 100
 TYPE(COMPLEXP) :: A
 TYPE(COMPLEXP), DIMENSION(NDAT) :: FDAT
 CALL NEW(A, 1.0, 1.0)
 CALL NEW(FDAT, (/(REAL(I), I = 1, NDAT)/), (/(-REAL(I), I = 1, NDAT)/))
 STOP
END PROGRAM MAIN
```

Добавим в модуль функцию умножения массивов. В Фортране 90 функции могут возвращать значения-массивы. Для этого функция описывается как массив, размерность которого определяется с помощью другого массива в списке параметров или с помощью функции SIZE. Поскольку операция * для скалярных переменных типа COMPLEXP уже определена в модуле COMPLEXP_CLASS, функция, значением которой является массив, использует циклический вызов этой операции:

```
FUNCTION MULTPCARRAY(THIS, B)
 TYPE(COMPLEXP), DIMENSION(:), INTENT(IN) :: THIS, B
 TYPE(COMPLEXP), DIMENSION(SIZE(THIS)) :: MULTPCARRAY
 DO I = 1, SIZE(THIS)
       MULTPCARRAY(I) = THIS(I) * B(I)
 END DO
END FUNCTION MULTPCARRAY
```

Наконец, можно перегрузить операцию * для того, чтобы в случае использования аргументов типа COMPLEXP фактически вызывалась процедура MULTPCARRAY:

```
INTERFACE OPERATOR(*)
 MODULE PROCEDURE MULTPCARRAY
END INTERFACE
```

В конечной версии производного класса COMPLEX_ARRAY следует перегрузить и функцию DISPLAY вывода элементов массива. Пример использования расширенного модуля комплексной арифметики приведен в листинге 12.9.

**Листинг 12.9. Пример использования расширенного модуля комплексной арифметики**

```
PROGRAM MAIN
 USE COMPLEX_ARRAY_CLASS
 INTEGER, PARAMETER :: NDAT = 100
 TYPE(COMPLEXP), DIMENSION(NDAT) :: FDAT, GDAT, HDAT
 CALL NEW(FDAT, (/(REAL(I), I = 1, NDAT)/), (/(-REAL(I), I = 1, NDAT)/))
 CALL NEW(GDAT, (/(REAL(I), I = 1, NDAT)/), (/(REAL(I), I = 1, NDAT)/))
 HDAT = FDAT * GDAT
 CALL DISPLAY(HDAT(1:3), 'HDAT = ')
 STOP
END PROGRAM MAIN
```

Наследование часто понимается в более узком смысле, чем композиция классов. Расширим класс `COMPLEXP` таким образом, чтобы он хранил историю выполненных операций. Это может оказаться полезным, например, при отладке программы. Определим производный тип:

```
TYPE MONITOR_COMPLEX
 TYPE(COMPLEXP) :: PC
 CHARACTER*15 :: LAST_OP
END TYPE MONITOR_COMPLEX,
```

который содержит один экземпляр типа `COMPLEXP`, а также дополнительный символьный компонент, используемый для записи операций. Модифицируем процедуры класса `COMPLEXP` (`NEW`, `*` и `DISPLAY`) так, чтобы они работали и в новом классе `MONITOR_COMPLEX`. Инициализация типа `MONITOR_COMPLEX` может быть выполнена с помощью оператора `NEW` в классе `COMPLEXP`:

```
TYPE(MONITOR_COMPLEX) :: X
CALL NEW(X%PC, 0.0, 0.0)
```

Дополнительный компонент инициализируется обычным образом:

```
X%LAST_OP = 'INIT'
```

Инициализацию можно включить в процедуру `INITMC`. Добавив интерфейс `NEW` в класс `MONITOR_COMPLEX`, перегрузим новую процедуру так, чтобы она вызывала `INITMC`, если аргумент имеет тип `MONITOR_COMPLEX`. Текст модуля, соответствующего классу `MONITOR_COMPLEX`, приведен в листинге 12.10.

**Листинг 12.10. Модуль комплексной арифметики**

```
MODULE MONITOR_COMPLEX_CLASS
 USE COMPLEXP_CLASS
 TYPE MONITOR_COMPLEX
```

```
 PRIVATE
 TYPE(COMPLEXP) :: PC
       CHARACTER * 15 :: LAST_OP
 END TYPE MONITOR_COMPLEX
 INTERFACE NEW
       MODULE PROCEDURE INITMC
 END INTERFACE
 CONTAINS
 SUBROUTINE INITMC(THIS, REALP, IMAGP)
       TYPE(MONITOR_COMPLEX), INTENT(OUT) :: THIS
       REAL, INTENT(IN) :: REALP, IMAGP
       CALL NEW(THIS%PC, REALP, IMAGP)
       THIS%LAST_OP = 'INIT'
 END SUBROUTINE INITMC
END MODULE MONITOR_COMPLEX_CLASS
```

Модифицируем и умножение, которое выполняется с помощью операции `*`, определенной в классе `COMPLEXP` на компоненте `COMPLEXP` типа `MONITOR_COMPLEX`:

```
TYPE(MONITOR_COMPLEX) :: X, Y, Z
Z%PC = X%PC * Y%PC
```

Эту операцию можно использовать в функции `MULTPC`:

```
TYPE(MONITOR_COMPLEX) FUNCTION MULTPC(THIS, B)
 TYPE(MONITOR_COMPLEX), INTENT(IN) :: THIS, B
 MULTPC%PC = THIS%PC * B%PC
 MULTPC%LAST_OP = 'MULTPIPLICATION'
END FUNCTION MULTPC
```

Пример использования модуля `MONITOR_COMPLEX_CLASS` приведен в листинге 12.11.

**Листинг 12.11. Пример использования модуля комплексной арифметики**

```
PROGRAM MAIN
 USE MONITOR_COMPLEX_CLASS
 TYPE(COMPLEXP) :: A, B, C
 TYPE(MONITOR_COMPLEX) :: X, Y, Z
 CALL NEW(A, 1., -1.)
 CALL NEW(B, -1., 2.)
 CALL NEW(X, 1., -1.)
 CALL NEW(Y, -1., 2.)
```

```
 C = A * B
 Z = X * Y
 CALL DISPLAY(C, 'C = ')
 CALL DISPLAY(Z, 'Z = ')
 Z = MC(C) * Z
 CALL LAST_OP(Z, 'Z')
END PROGRAM MAIN
```

В этом примере каждый объект производного класса содержит один объект базового класса. Расширение процедур заключалось в написании процедуры производного класса, которая вызывает процедуру базового класса, определенную на компоненте базового класса. Затем выполнялась перегрузка имени, так чтобы оно совпадало с именем процедуры базового класса. Базовый класс при этом не модифицировался. Такая форма наследования называется подтипизацией, потому что объекты производного типа могут использовать процедуры базового класса, как если бы они были одного типа. Хотя возможности наследования подтипизацией ограниченны, в языках ООП имеются механизмы автоматического расширения немодифицированных процедур базового класса на производные типы. Это делается с помощью автоматического преобразования типов. Фортран 90 не располагает такими механизмами, поэтому наследование подтипизацией должно конструироваться явным образом.

## Производные типы с указателями

Пусть в модуле `PARTS_MODULE` имеется процедура расчета движения заряженных частиц в электрическом поле `CHARGE_MOVING`:

```
SUBROUTINE CHARGE_MOVING(PARTICLES, FIELD, PARTS_ARGS, DT)
```

Аргумент `PARTS_ARGS` типа `PARTS_DESCRIPTOR` содержит характеристики группы частиц. Массив `PARTICLES` содержит координаты частицы из этой группы.

Упростим интерфейс этой процедуры, для чего введем производный тип:

```
TYPE PARTS
 TYPE(PARTS_DESCRIPTOR) :: DESCRIPTOR
 REAL, DIMENSION(DIMPHASE, NPARTICLES) :: COORDINATES
END TYPE PARTS
```

Неудобство такого определения заключается в том, что изменение значений `DIMPHASE` или `NPARTICLES` потребует перекомпиляции кода. Использование динамического массива:

```
TYPE PARTS
 TYPE(PARTS_DESCRIPTOR) :: DESCRIPTOR
 REAL, DIMENSION(:, :), ALLOCATABLE :: COORDINATES
END TYPE PARTS
```

проблему не решает, потому что в определении производного типа динамические массивы использовать нельзя. Выйти из положения можно с помощью указателей на массивы.

Определим вещественные массивы и указатели на массивы:

```
REAL, DIMENSION(4), TARGET :: A, B
REAL, DIMENSION(:), POINTER :: PTR1, PTR2
```

Указатель `PTR1` свяжем с массивом `A`:

```
PTR1 => A
```

После этого указатель `PTR1` можно использовать вместо массива `A` (это называется разыменованием указателя). Например, оператор:

```
PTR1(1) = 1.0
```

присвоит значение элементу `A(1)`. Для отмены связи указателя с массивом (его открепления) используется функция `NULLIFY`. Функция `ASSOCIATED` позволяет определить, связан ли указатель с массивом:

```
IF(ASSOCIATED(PTR1)) PRINT *, 'PTR1 АССОЦИИРОВАН!'
```

Ее можно использовать и для определения, не ссылаются ли два указателя на один адрес:

```
IF(ASSOCIATED(PTR1, PTR2)) PRINT *, 'УКАЗЫВАЮТ НА ОДИН АДРЕС!'
```

Разрешается связывать указатели и с неименованными переменными:

```
NULLIFY(PTR1)
ALLOCATE(PTR1(4))
```

Здесь сначала отменяется предыдущее ассоциирование указателя `PTR1`, а затем размещается неименованный массив из четырех вещественных элементов. Удаление массива выполняется обычным образом:

```
DEALLOCATE(PTR1)
```

Указатели на неименованные массивы ведут себя как динамические массивы, кроме того, их можно использовать в определении производного типа:

```
TYPE PARTS
 TYPE(PARTS_DESCRIPTOR) :: DESCRIPTOR
 REAL, DIMENSION(:, :), POINTER :: COORDINATES
END TYPE PARTS
```

Если объявить переменную `ELECTRONS`:

```
TYPE(PARTS) :: ELECTRONS,
```

то массив указателей можно разместить следующим образом:

```
ALLOCATE(ELECTRONS%COORDINATES(DIMPHASE, NPARTICLES), STAT = IERR)
```

Новый вариант процедуры `CHARGE_MOVING` вызывается так:

```
CALL CHARGE_MOVING(ELECTRONS, FIELD, DT)
```

При выполнении оператора присваивания:

```
TYPE(PARTS) :: ELECTRONS, IONS
IONS = ELECTRONS
```

происходит следующее:

```
IONS%DESCRIPTOR = ELECTRONS%DESCRIPTOR
IONS%COORDINATES => ELECTRONS%COORDINATES
```

Во второй строке копируется компонент указателя, а не данные, на которые он указывает. Оператор:

```
IONS%COORDINATES = ELECTRONS%COORDINATES
```

разыменовывает указатель и копирует данные. Он используется в процедуре:

```
SUBROUTINE COPY_PARTS(A, B)
 TYPE(PARTS), INTENT(OUT) :: A
 TYPE(PARTS), INTENT(IN) :: B
 A%DESCRIPTOR = B%DESCRIPTOR
 A%COORDINATES = B%COORDINATES
END SUBROUTINE COPY_PARTS
```

Свяжем эту процедуру с оператором присваивания:

```
INTERFACE ASSIGNMENT(=)
 MODULE PROCEDURE COPY_PARTS
END INTERFACE
```

Оператор:

```
IONS = ELECTRONS
```

в этом случае будет копировать данные, а не указатель.

## Динамическая диспетчеризация

Динамическую диспетчеризацию иногда называют полиморфизмом времени выполнения или динамическим полиморфизмом. Подтипизация статична, то есть уже при компиляции определяется, какую процедуру следует вызывать в заданной точке программы. Динамическая диспетчеризация позволяет одному имени объекта ссылаться на любой элемент иерархии наследования и позволяет процедуре во время выполнения решать, на какой фактический объект следует сослаться.

Процедура `WORK` использует методы в иерархии класса `COMPLEXP` и возводит в квадрат комплексное число:

```
SUBROUTINE WORK(A)
 TYPE(COMPLEXP), INTENT(INOUT) :: A
 A = A * A
 CALL DISPLAY(A, 'WORK:')
END SUBROUTINE WORK
```

Она написана для типа COMPLEXP, но должна правильно работать и со всеми типами, производными от COMPLEXP. Если описать и инициализировать объекты класса так:

```
TYPE(COMPLEXP), POINTER :: PC
TYPE(COMPLEXP), TARGET :: A
TYPE(MONITOR_COMPLEX), TARGET :: B
CALL NEW(A, 1., -1.)
CALL NEW(B, 1., -1.),
```

то должна появиться возможность выполнения следующих операций:

```
PC => A
CALL WORK(PC)
PC => B
CALL WORK(PC)
```

Третья строка этого примера является ошибочной, потому что указатель PC может указывать только на адресат типа COMPLEXP. Для того чтобы решить эту проблему, определим производный тип, который содержит указатель на каждый из возможных типов в иерархии наследования:

```
TYPE COMPLEX_SUBTYPE
 TYPE(COMPLEXP), POINTER :: PC
 TYPE(MONITOR_COMPLEX), POINTER :: MC
END TYPE COMPLEX_SUBTYPE
```

Этот подтип может указывать на любой комплексный тип:

```
TYPE(COMPLEX_SUBTYPE) :: GENERIC_PC
GENERIC_PC%PC => A
GENERIC_PC%MC => B
```

Динамическую диспетчеризацию можно реализовать, определив подтип класса, который наследует все классы в иерархии наследования и содержит новую версию каждого метода. Эта версия будет содержать проверку, какой из указателей в типе COMPLEX_SUBTYPE ассоциирован, и выполнять соответствующую версию функции. Можно определить новую процедуру вывода:

```
SUBROUTINE DISPLAY_SUBTYPE(A, C)
 TYPE(COMPLEX_SUBTYPE), INTENT(IN) :: A
 CHARACTER * (*), INTENT(IN) :: C
 IF(ASSOCIATED(A%PC)) THEN
       CALL DISPLAY(A%PC, C)
 ELSE IF(ASSOCIATED(A%MC)) THEN
       CALL DISPLAY(A%MC, C)
 END IF
END SUBROUTINE DISPLAY_SUBTYPE
```

Аргументом этой процедуры является переменная типа COMPLEX_SUBTYPE. Новая функция умножения также должна скрывать решение о вызове подходящей процедуры умножения. После перегрузки процедура WORK будет иметь тот же интерфейс, что и раньше. Изменится только тип аргумента:

```
SUBROUTINE WORK(A)
 TYPE(COMPLEX_SUBTYPE), INTENT(INOUT) :: A
 A = A * A
 CALL DISPLAY(A, 'WORK:')
END SUBROUTINE WORK
```

Подтипу класса необходим свой оператор присваивания. Компоненты класса COMPLEX_SUBTYPE будут скрытыми, поэтому нужна процедура для динамического присваивания одного из возможных указателей в иерархии класса и открепления остальных. Присваивание указателя COMPLEXP выполняется процедурой:

```
SUBROUTINE ASSIGN_PC(CS, PC)
 TYPE(COMPLEX_SUBTYPE), INTENT(OUT) :: CS
 TYPE(COMPLEXP), TARGET, INTENT(IN) :: PC
 CS%PC => PC
 NULLIFY(CS%MC)
END SUBROUTINE ASSIGN_PC
```

Процедура ASSIGN_MC присваивает значение указателю MONITOR_COMPLEX. Для того чтобы выражение:

```
A = A * A
```

давало правильный результат, следует определить новый оператор копирования COPY_SUBTYPE, который будет копировать данные, а не указатели. Операторы присваивания и копирования могут перегружаться оператором присваивания (=). Класс COMPLEX_SUBTYPE объединяет все эти возможности.

Программа, вызывающая WORK, приведена в листинге 12.12.

**Листинг 12.12. Пример использования процедуры WORK**

```
PROGRAM MAIN
 USE COMPLEX_SUBTYPE_CLASS
 TYPE(COMPLEX_SUBTYPE) :: PC
 TYPE(COMPLEXP), TARGET :: A
 TYPE(MONITOR_COMPLEX), TARGET :: B
 CALL NEW(A, 1., -1.)
 CALL NEW(B, 1., -1.)
 CALL ASSIGN(PC, A)
```

```
 CALL WORK(PC)
 CALL ASSIGN(PC, B)
 CALL WORK(PC)
END PROGRAM MAIN
```

За использование динамической диспетчеризации приходится расплачиваться быстродействием даже в ООП-языках.

Для имитации динамической диспетчеризации в Фортране 90 необходимо реализовать: во-первых, указатель, который может ссылаться на любой член иерархии наследования, во-вторых, механизм диспетчеризации, который может выбирать подходящую процедуру (метод), в зависимости от фактически используемого класса. В C++ это делается автоматически с помощью виртуальных функций.

Реализация динамической диспетчеризации в Фортране 90 сложнее, чем в языках объектно-ориентированного программирования, но используется она так же.

## Реализация классов

Рассмотрим еще один пример моделирования классов. Аналогом класса в Фортране 90 можно считать модуль, который содержит определение абстрактного (производного) типа данных и процедур, которые работают только с этим типом. Кроме того, модуль может содержать данные, соответствующие статическим членам класса в C++. Исходный текст модуля, соответствующего классу `PERSONNEL`, дан в листинге 12.13.

**Листинг 12.13. Модуль, соответствующий классу *PERSONNEL***

```
MODULE PERSONNEL_CLASS
 TYPE PERSONNEL
       INTEGER :: SSN
       CHARACTER*12 :: FIRSTNAME, LASTNAME
 END TYPE PERSONNEL
 CONTAINS
       SUBROUTINE NULL_PERSONNEL(THIS)
             TYPE(PERSONNEL), INTENT (OUT) :: THIS
             THIS%SSN = 0
             THIS%FIRSTNAME = ' '
             THIS%LASTNAME = ' '
       END SUBROUTINE NULL_PERSONNEL
END MODULE PERSONNEL_CLASS
```

Тип данных класса в этом примере содержит три компонента — переменную целого типа SSN и символьные переменные FIRSTNAME и LASTNAME. Этот класс содержит конструктор, инициализирующий SSN нулевым значением, а символьные поля — пробелами. Пример использования класса PERSONNEL_CLASS дан в листинге 12.14.

**Листинг 12.14. Пример использования класса *PERSONNEL_CLASS***

```
PROGRAM PERSONNEL_TEST
 USE PERSONNEL_CLASS
 TYPE(PERSONNEL) :: PERSON
 CALL NULL_PERSONNEL(PERSON)
END PROGRAM PERSONNEL_TEST
```

Оператор USE помещает класс в область видимости главной программы. Для процедур в модулях Фортрана 90 не требуются опережающие описания, они генерируются автоматически. Оператор INTERFACE позволяет обеспечить функциональность опережающих описаний для процедур, не располагающихся в модулях. Объект класса PERSON создается за два шага. Сначала описывается и создается переменная этого типа, затем применяется процедура-конструктор, инициализирующая объект:

```
TYPE(PERSONNEL) :: PERSON
CALL NULL_PERSONNEL(PERSON)
```

В C++ создание и инициализация могут быть объединены в один оператор. В процедуре NULL_PERSONNEL ссылка на объект класса всегда явная. Она является аргументом. Условимся делать его первым аргументом и называть THIS. В C++ возможна ссылка на объект, без явного его описания. Более того, доступ к полям организуется как к компонентам формального аргумента THIS.

В Фортране 90 есть два способа представления строк. Наиболее распространенный способ — использование переменных типа CHARACTER:

```
CHARACTER * 12 :: FIRSTNAME
```

Символьные переменные являются инкапсулированными объектами, которые содержат значение своей длины. Длина не может устанавливаться динамически.

Расширим класс PERSONNEL. Начнем с создания конструктора INIT_PERSONNEL для инициализации объекта:

```
SUBROUTINE INIT_PERSONNEL(THIS, S, FN, LN)
 TYPE(PERSONNEL), INTENT(OUT) :: THIS
 INTEGER, INTENT(IN) :: S
```

```
 CHARACTER * (*), INTENT(IN) :: FN, LN
 THIS%SSN = S
 THIS%FIRSTNAME = FN
 THIS%LASTNAME = LN
END SUBROUTINE INIT_PERSONNEL
```

Атрибут `INTENT(IN)` аргумента `S` соответствует ключевому слову `const` языка C++. При использовании в качестве формального аргумента процедуры, длина символьной переменной не описывается — она может быть определена с помощью функции `LEN`, аналогичной функции `strlen` в C++. Инициализация `PERSON` имеет вид:

```
TYPE(PERSONNEL) :: PERSON
CALL INIT_PERSONNEL(PERSON, 1, 'PAUL', 'JONES')
```

Символьные переменные Фортрана можно считать встроенным строковым классом. Они имеют фиксированную длину, следствием чего является неэффективное использование памяти. Другой подход основан на конструировании строк в духе языка C с помощью динамического массива символов:

```
CHARACTER * 1, DIMENSION(:), ALLOCATABLE :: FIRSTNAME
```

Память для такого массива отводится оператором `ALLOCATE`:

```
ALLOCATE(FIRSTNAME(LEN('PAUL'))),
```

который размещает массив однобайтовых символов, размер которого равен длине строки `PAUL`.

Строки в стиле языка C редко используются в Фортране 90, потому что с ними могут работать не все фортрановские строковые функции. Например, присваивание для строк в стиле C должно выполняться с помощью конструктора массива:

```
FIRSTNAME = (/'P', 'A', 'U', 'L'/)
```

вместо привычного оператора присваивания:

```
FIRSTNAME = 'PAUL'
```

Для того чтобы упростить присваивание строк, добавим процедуру, которая конвертирует символьную переменную фиксированного размера в массив однобайтовых символов:

```
SUBROUTINE STRCPY(S, C)
 CHARACTER, DIMENSION (:), INTENT(OUT) :: S
 CHARACTER * (*), INTENT(IN) :: C
 DO I = 1, MAX(SIZE(S), LEN(C))
       S(I) = C(I:I)
 END DO
END SUBROUTINE STRCPY
```

Для определения длины символьного массива S используется функция SIZE, а функция LEN позволяет определить длину символьной переменной C. С помощью процедуры STRCPY можно использовать динамические символьные массивы, сохраняя простой синтаксис присваивания:

```
CALL STRCPY(FIRSTNAME, 'PAUL')
```

Хотя динамические массивы и нельзя использовать в определениях производных типов, указатели на массивы могут заменить массивы, в том числе и при вызове процедур. Таким образом, выражение:

```
PTR(3) = 4
```

либо присвоит значение третьему элементу массива PTR, если PTR — массив, либо разыменует ссылку, если PTR — указатель на массив.

Добавим в класс PERSONNEL_CLASS конструктор и деструктор:

```
MODULE PERSONNEL_CLASS
 TYPE PERSONNEL
       INTEGER :: SSN
       CHARACTER, DIMENSION(:), POINTER :: FIRSTNAME, LASTNAME
 END TYPE PERSONNEL
 CONTAINS
       SUBROUTINE INIT_PERSONNEL(THIS, S, FN, LN)
              TYPE(PERSONNEL), INTENT(OUT) :: THIS
              INTEGER, INTENT(IN) :: S
              CHARACTER * (*), INTENT(IN) :: FN, LN
              THIS%SSN = S
              ALLOCATE(THIS%FIRSTNAME(LEN(FN)), THIS%LASTNAME(LEN(LN)))
              CALL STRCPY(THIS%FIRSTNAME, FN)
              CALL STRCPY(THIS%LASTNAME, LN)
 END SUBROUTINE INIT_PERSONNEL

       SUBROUTINE TERM_PERSONNEL(THIS)
              TYPE(PERSONNEL), INTENT (INOUT) :: THIS
              DEALLOCATE(THIS%FIRSTNAME, THIS%LASTNAME)
       END SUBROUTINE TERM_PERSONNEL

       SUBROUTINE STRCPY(S, C)
              CHARACTER, DIMENSION(:), INTENT(OUT) :: S
              CHARACTER * (*), INTENT(IN) :: C
              DO I = 1, MAX(SIZE(S), LEN(C))
                     S(I) = C(I:I)
```

```
            END DO
       END SUBROUTINE STRCPY
END MODULE PERSONNEL_CLASS
```

Перегрузим имя `NEW` и отождествим его с именами конструкторов `NULL_PERSONNEL` и `INIT_PERSONNEL`:

```
INTERFACE NEW
 MODULE PROCEDURE NULL_PERSONNEL, INIT_PERSONNEL
END INTERFACE
```

Можно перегрузить имя `DELETE` и отождествить его с `TERM_PERSONNEL`.

Деструктор обычно нужен только, если класс содержит указатели, которые необходимо открепить. Для класса `PERSONNEL` инициализация и удаление объекта выполняются так:

```
TYPE(PERSONNEL) :: PERSON
CALL NEW(PERSON, 1, 'PAUL', 'JONES')
CALL DELETE(PERSON)
```

Базовый класс `PERSONNEL` можно расширить. Прежде всего, добавим атрибут `PRIVATE`:

```
TYPE PERSONNEL
 PRIVATE
 INTEGER :: SSN
 CHARACTER, DIMENSION(:), POINTER :: FIRSTNAME, LASTNAME
END TYPE PERSONNEL
```

Этот атрибут делает компоненты типа `PERSONNEL` доступными только методам класса. По умолчанию используется атрибут общего доступа `PUBLIC`. В Фортране 90 отсутствует ключевое слово `PROTECTED`. Нельзя также делать некоторые элементы данных открытыми, а другие закрытыми.

Оператор:

```
PRIVATE :: INIT_PERSONNEL, TERM_PERSONNEL
```

сделает имена `INIT_PERSONNEL` и `TERM_PERSONNEL` скрытыми.

Добавим в класс процедуру вывода записи `PERSONNEL`:

```
SUBROUTINE PRINT_PERSONNEL(THIS, PRINTSSN)
 TYPE(PERSONNEL), INTENT(IN) :: THIS
 LOGICAL, OPTIONAL, INTENT(IN) :: PRINTSSN
 IF(PRESENT(PRINTSSN)) THEN
       IF(PRINTSSN) WRITE(*, '(I2, A2)', ADVANCE = 'NO') THIS%SSN, ': '
 END IF
 PRINT *, THIS%FIRSTNAME, ' ', THIS%LASTNAME
END SUBROUTINE PRINT_PERSONNEL
```

Оператор PRINT выполняет разыменование массивов, на которые указывают THIS%FIRSTNAME и THIS%LASTNAME. Предлагаем читателю самостоятельно разобрать работу этой процедуры.

В класс полезно добавить процедуру получения значения SSN:

```
FUNCTION GETSSN_PERSONNEL(THIS) RESULT(SSN)
 TYPE(PERSONNEL), INTENT (IN) :: THIS
 INTEGER :: SSN
 SSN = THIS%SSN
END FUNCTION GETSSN_PERSONNEL
```

Если перегрузить имя PRINT, добавив ссылку на PRINT_PERSONNEL, следующие операторы будут инициализировать запись PERSONNEL и выводить ее:

```
TYPE(PERSONNEL) :: PERSON
CALL NEW(PERSON, 1, 'BILL', 'JOE')
CALL PRINT(PERSON, .TRUE.)
```

В Фортране 90 метод вызывается с использованием синтаксиса:

```
CALL PRINT(PERSON, .TRUE.),
```

а в C++ это выглядело бы примерно так:

```
PERSON.PRINT(1);
```

Полезно добавить возможность определения количества созданных записей. Для этого вводится статическое поле NUM_FILES:

```
INTEGER, SAVE :: NUM_FILES = 0
```

Это предложение должно находиться перед оператором CONTAINS. Атрибут SAVE эквивалентен ключевому слову static в языке C++. Статические поля могут быть инициализированы в определении класса, что вообще говоря не допускается в C++. Количество записей можно учитывать, увеличивая значение глобальной переменной NUM_FILES в процедуре NEW и уменьшая его в процедуре DELETE.

В Фортране 90 область видимости переменных модуля контролируется оператором USE, а имена процедур должны быть либо уникальными, либо перегружаться оператором INTERFACE. В примере:

```
USE PERSONNEL_CLASS, LOCAL_NUM_FILES_NAME => NUM_FILES
PRINT *, LOCAL_NUM_FILES_NAME
```

имя LOCAL_NUM_FILES_NAME используется для ссылки на член статического класса NUM_FILES. С другой стороны, можно создать статическое поле, добавив атрибут PRIVATE.

```
INTEGER, SAVE, PRIVATE :: NUM_FILES = 0
```

В этом случае для чтения значения NUM_FILES следует добавить статический метод без ссылки на THIS:

```
INTEGER FUNCTION GET_NUM_FILES()
```

```
 GET_NUM_FILES = NUM_FILES
END FUNCTION GET_NUM_FILES
```

Пример использования класса PERSONNEL дан в листинге 12.15.

**Листинг 12.15. Пример использования класса *PERSONNEL***

```
PROGRAM PERSONNEL_TEST
 USE PERSONNEL_CLASS
 TYPE(PERSONNEL) :: PERSON
 CALL NEW(PERSON, 1, 'PAUL', 'JONES')
 CALL PRINT(PERSON)
 CALL DELETE(PERSON)
 PRINT *, 'NUM_FILES = ', GET_NUM_FILES()
END PROGRAM PERSONNEL_TEST
```

## Множественное наследование

*Множественное наследование* позволяет создавать композиционные классы, которые имеют свойства нескольких базовых классов. В качестве примера приведем класс STUDENTTEACHER, который наследует как из STUDENT, так и из TEACHER. В Фортране 90 композиционный класс можно реализовать, с помощью производного типа:

```
TYPE STUDENTTEACHER
 TYPE(STUDENT) :: STUDENT
 TYPE(TEACHER) :: TEACHER
END TYPE STUDENTTEACHER,
```

который включает компоненты типа STUDENT и типа TEACHER. Объявление этого типа можно включить в модуль:

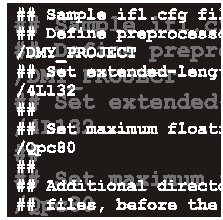
```
MODULE STUDENTTEACHER_CLASS
 USE STUDENT_CLASS
 USE TEACHER_CLASS
 TYPE STUDENTTEACHER
 TYPE(STUDENT) :: STUDENT
       TYPE(TEACHER) :: TEACHER
 END TYPE STUDENT
END MODULE STUDENTTEACHER_CLASS
```

Таким образом, эмуляция множественного наследования в Фортране 90 вызывает не больше трудностей, чем реализация простого наследования.

# Вопросы и задания для самостоятельной работы

1. Что такое объектно-ориентированное программирование?
2. Что такое инкапсуляция?
3. Что такое наследование? Чем простое наследование отличается от множественного?
4. Что такое полиморфизм?
5. Что такое объект и как объект связан с классом?
6. Расскажите о реализации композиции классов в Фортране 90.
7. Расскажите о том, как можно реализовать подтипизацию в Фортране 90.
8. Сделайте доклад о роли и необходимости объектно-ориентированного подхода в разработке вычислительных программ. Аргументируйте свои выводы.
9. Используя методы ООП, разработайте на Фортране 90 модуль для основных матрично-векторных операций и решения основных задач вычислительной алгебры (решение систем линейных уравнений и т. д.).
10. Проанализируйте те вычислительные задачи, которые вам приходится решать в своей предметной области, и сделайте вывод о целесообразности применения ООП.
11. Сделайте сравнительный обзор Фортрана 90 и любого языка объектно-ориентированного программирования с точки зрения удобства реализации ООП и быстродействия программ.

# Глава 13




# Фортран и высокопроизводительные вычисления

В этой главе мы расскажем о расширениях Фортрана для многопроцессорных вычислительных систем, к которым относятся суперкомпьютеры различной архитектуры и специализированные сетевые объединения персональных компьютеров — кластеры. Основным побудительным мотивом для создания параллельных расширений стандартных языков является стремление упростить пользователю переход от последовательной программы к параллельной. Поэтому разработчики расширений, как правило, стремятся добавить к стандартным операторам минимальное число максимально лаконичных директив, позволяющих получить оптимальный выигрыш от распараллеливания программы. Все производители суперкомпьютеров были вынуждены разрабатывать расширения стандартных языков программирования для своих компьютеров; необходимость стандарта была очевидной. Работы по созданию новых расширений продолжаются, так как совершенства нет, да оно и не возможно. К настоящему времени приняты два стандарта параллельных расширений Фортрана, ориентированные на системы с распределенной и разделяемой памятью соответственно: High Performance Fortran (HPF, высокопроизводительный Фортран) и OpenMP (Open specifications for MultiProcessing, открытая спецификация мультиобработки данных).

## High Performance Fortran

Первая версия стандарта HPF была разработана в рекордно короткие сроки (менее двух лет) и опубликована в мае 1993 г. Инициаторами разработки были крупные компании, выпускающие компьютеры: Convex, DEC, IBM, Sun Microsystems и другие. Вторая версия стандарта появилась в 1996 г.

HPF реализует парадигму параллелизма данных и модель обработки данных SIMD (Single Instruction — Multiple Data, один поток команд — несколько потоков данных). Поэтому основу HPF составляют директивы распределения данных. Они скрывают от разработчика приложений обращения к средствам

межпроцессорного обмена данными, специфическим для каждой аппаратной платформы, обеспечивая тем самым платформенную независимость прикладных программ. Однако кажущаяся простота директив HPF имеет обратную сторону: недостаточно продуманное применение директив может привести к неконтролируемому росту обмена данными и к обвальному падению эффективности программы.

Компилятор HPF действует как препроцессор Фортрана: на первом этапе компиляции директивы заменяются вызовами библиотечных подпрограмм, вид и содержание которых зависят от операционной системы, для которой компилятор был разработан.

Фортран 90 является базовым языком HPF, и программа на Фортран 90, без изменений перенесенная в HPF, может выиграть в эффективности только за счет встроенного параллелизма. Директивы HPF встраиваются в текст программы, написанной на Фортране 90, и с точки зрения последнего выглядят как комментарии различного стиля:

```
!HPF$ <директива HPF> —
```

стиль Фортрана 90 или в стиле Фортрана 77:

```
CHPF$ <директива HPF>,
```

где символ `C` располагается в первой позиции строки. Мы будем придерживаться стиля Фортрана 90. Если директива не помещается на одной строке, она может быть продолжена символом `&`, но каждая строка продолжения также должна начинаться символами `!HPF$`.

## Директивы распределения данных

Директивы распределения данных составляют основу HPF. Важным моментом распределения является описание псевдомассива процессоров, по которым данные предстоит распределить. Этот псевдомассив является программным представлением набора используемых процессоров, которое, вообще говоря, не связано с реальным числом процессоров компьютера и их аппаратной организацией. Программный псевдомассив процессоров, главным образом, задает ранг решетки процессоров, по которым предстоит распределять данные. Число процессоров тоже может быть задано, но такое ограничение редко бывает оправдано — программа создается в расчете на произвольное число процессоров, определяемых во время исполнения. Дело в том, что распределение массивов выполняется, кроме случаев исключения измерения, только по массиву процессоров того же ранга. Таким образом, в одной и той же программе могут понадобиться несколько массивов процессоров одного размера, но различной формы. По умолчанию на этапе исполнения всегда доступен одномерный массив процессоров, число элементов которого равно числу физических процессоров используемой вычислительной системы. Встроенные функции HPF `NUMBER_OF_PROCESSORS()` и

`PROCESSORS_SHAPE()` возвращают, соответственно, размер и форму массива реальных процессоров.

Программный массив процессоров описывается директивой `PROCESSORS`, например, так (листинг 13.1).

**Листинг 13.1. Пример директивы *PROCESSORS***

```
!HPF$ PROCESSORS P(8)
!HPF$ PROCESSORS, DIMENSION(:)   ::P1
!HPF$ PROCESSORS, DIMENSION(:,:) ::P2
```

Встроенные функции HPF можно использовать в исполняемых операторах:

```
NumOfProc = NUMBER_OF_PROCESSORS()
```

и в директивах HPF даже в блоке операторов описания, например,

```
!HPF$ PROCESSORS P22(2, NUMBER_OF_PROCESSORS()),
```

но нельзя использовать в операторах описания Фортрана:

```
INTEGER, PARAMETER :: NumOfProc = NUMBER_OF_PROCESSORS()!Это ошибка.
```

После определения псевдомассива процессоров можно приступать собственно к распределению данных, которое выполняется директивой `DISTRIBUTE`:

```
!HPF$ DISTRIBUTE [(способ_распределения)] &
!HPF$ [ONTO псевдомассив_процессоров] :: список_распределяемых_массивов
```

Псевдомассив процессоров должен быть определен предшествующей директивой `PROCESSORS`, массивы из списка распределяемых массивов должны быть точно (с указанием размера) описаны. Есть два способа распределения: блочный и циклический. При блочном распределении элементы массива разделяются на приблизительно равные блоки с идущими подряд элементами. В следующем примере массив из 100 элементов распределяется блочно по псевдомассиву из 5-ти линейно расположенных процессоров (листинг 13.2).

**Листинг 13.2. Пример директивы *DISTRIBUTE***

```
!REAL, DIMENSION(100) :: A
!HPF$ PROCESSORS P(5)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A
```

При таком распределении первые 20 элементов попадут первому процессору, следующие 20 — второму и так далее. Если число элементов не делится на число процессоров нацело, последний процессор получит столько элементов, сколько останется от деления нацело числа элементов массива на число

процессоров без одного. Так, если бы в псевдомассиве P было бы 6 процессоров, первые 5 получили бы по 20 элементов, а 6-ой не получил бы ни одного; при 7 процессорах — по 16 и 4 соответственно.

При циклическом распределении элементы массива распределяются как карты между игроками в тех карточных играх, где колоду раздают по кругу, выдавая по одной карте за каждый проход (листинг 13.3).

**Листинг 13.3. Пример циклического распределения**

```
!REAL, DIMENSION(15)::A
!HPF$ PROCESSORS P(9)
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: A
```

При таком распределении первый процессор получит 2 элемента массива A — первый и десятый, второй получит элементы с индексами 2 и 11 и так далее. Процессоры с восьмого по девятый получат по одному элементу.

Блочное распределение обеспечивает принадлежность элементов с близкими номерами одному процессору, что может избавить программу от лишнего обмена при итерациях с рекурсией по соседству, наподобие следующего оператора:

```
A(I) = A(I - 1) + 2 * A(I) + A(I + 1)
```

Циклическое распределение обеспечивает равномерную загрузку процессоров.

Двумерные массивы необходимо распределять по двумерному псевдомассиву процессоров, причем способы распределения в различных измерениях могут быть разными (листинг 13.4).

**Листинг 13.4. Пример смешанного распределения**

```
!REAL, DIMENSION(10, 100) :: A
!HPF$ PROCESSORS P(2, 2)
!HPF$ DISTRIBUTE (CYCLIC, BLOCK) ONTO P :: A
```

Если указание способа распределения опущено, то назначается блочный способ распределения. И циклическое, и блочное распределение допускают указание параметра — числа элементов в блоке или в порции выдаваемых циклически элементов (листинг 13.5).

**Листинг 13.5. Пример смешанного распределения с параметром**

```
!REAL, DIMENSION(10, 100) :: A
!HPF$ PROCESSORS P(2, 2)
!HPF$ DISTRIBUTE (CYCLIC(2), BLOCK(50)) ONTO P :: A
```

Указание параметра почти "сводит на нет" различие между блочным и циклическим распределением. Но при этом циклическое распределение с параметром выполнимо всегда, блочное же только тогда, когда значение параметра позволяет распределить все элементы массива полностью: "второго захода" при блочном распределении не бывает. Так что директива:

```
!HPF$ DISTRIBUTE (CYCLIC(2), BLOCK(2)) ONTO P :: A
```

в предыдущем примере была бы ошибочна.

Можно все элементы в одном из измерений привязать к распределению по другому измерению — так называемый коллапс измерения (листинг 13.6).

**Листинг 13.6. Коллапс измерения при распределении**

```
!REAL, DIMENSION(10, 100) :: A
!HPF$ PROCESSORS P(5)
!HPF$ DISTRIBUTE (CYCLIC, *) ONTO P :: A
```

В этом случае ранг псевдомассива процессоров понижается на единицу.

При распределении динамических массивов привязка элементов массива к процессорам происходит только тогда, когда выполнен соответствующий оператор `ALLOCATE`.

Кроме директивы `DISTRIBUTE`, для распределения данных используется еще одна директива HPF: `ALIGN` — директива выравнивания. Она обеспечивает распределение "по образцу" — по массивам или псевдомассивам, распределение которых производится директивой `DISTRIBUTE` (порядок следования директив `DISTRIBUTE` и `ALIGN` не имеет значения). Например, массив `B` можно распределить точно так же, как распределен массив `A` (листинг 13.7).

**Листинг 13.7. Пример применения директивы выравнивания**

```
!REAL, DIMENSION(10, 100)::A, B
!HPF$ PROCESSORS P(5)
!HPF$ DISTRIBUTE (CYCLIC, *) ONTO P :: A
!HPF$ ALIGN (:, :) WITH A(:, :) :: B
```

Специальная директива `TEMPLATE` позволяет определить псевдомассив-шаблон для последующего использования его в директиве `ALIGN` и тем самым полностью разделить описания Фортрана и директивы HPF (листинг 13.8).

**Листинг 13.8. Применение директивы *TEMPLATE***

```
!REAL, DIMENSION(100, 100) :: A
INTEGER, DIMENSION(20, 400) :: N
!HPF$ PROCESSORS, DIMENSION(:, :) :: P2
!HPF$ TEMPLATE, DIMENSION(100, 100) :: T1
!HPF$ TEMPLATE,DIMENSION(20, 400) :: T2
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO P2 :: T1
!HPF$ DISTRIBUTE (CYCLIC, BLOCK) ONTO P2 :: T2
!HPF$ ALIGN A(:, :) WITH T1
!HPF$ ALIGN N(:, :) WITH T2
```

Есть более гибкая индексная форма записи директивы `ALIGN`, она позволяет выравнивать не только конформные массивы (листинг 13.9).

**Листинг 13.9. Индексная форма директивы *ALIGN***

```
!REAL A(12), B(10), C(8)
!HPF$ DISTRIBUTE (CYCLIC) A
!HPF$ ALIGN (j) WITH A(j) :: B, C
```

Выравнивание может выполняться и более изощренными способами — с инверсией и заданным шагом (листинг 13.10).

**Листинг 13.10. Пошаговое инверсное выравнивание**

```
!REAL A(10), B(20)
!HPF$ DISTRIBUTE (CYCLIC) A
!HPF$ ALIGN B(:) WITH A(UBOUND(A)::-2),
```

или с коллапсом измерения (листинг 13.11).

**Листинг 13.11. Коллапс измерения при выравнивании**

```
!REAL A(10), B(10, 10)
!HPF$ DISTRIBUTE (CYCLIC) A
!HPF$ ALIGN(*, :) WITH A(:) :: B,
```

а также с репликацией элементов. Последнее требует дополнительных разъяснений, так как связано с очень ответственной и скрытой операцией. Дело в том, что массивы, не распределенные явно, дублируются (реплицируются)

на все процессоры как скалярные элементы. Это процесс, который должен выполняться по умолчанию; специальной директивы, указывающей на необходимость репликации, в HPF нет. Приведенный ниже пример показывает, как организовать принудительную репликацию с помощью директив TEMPLATE и ALIGN (листинг 13.12).

**Листинг 13.12. Принудительная репликация**

```
!REAL,DIMENSION(100,100)::A
!HPF$ PROCESSORS,DIMENSION(NUMBER_OF_PROCESSORS())::P
!HPF$ TEMPLATE,DIMENSION(NUMBER_OF_PROCESSORS())::T
!HPF$ ALIGN A(*,*) WITH T(*)
!HPF$ DISTRIBUTE::T
```

## Параллелизм исполнения в HPF

В HPF включены несколько инструкций, не связанных с привязкой данных. Это инструкции FORALL, INDEPENDENT, PURE и NEW, которые вносят в HPF элемент MIMD (Multiple Instructions — Multiple Data, несколько потоков команд — несколько потоков данных) модели обработки данных.

Инструкция FORALL так удачно дополнила базовый язык, что стала вноситься по инициативе разработчиков сначала в отдельные реализации Фортрана 90, а затем была включена в стандарт Фортрана *95 (см. гл. 8, разд. "Оператор и конструкция FORALL")*.

Директива INDEPENDENT применяется к циклам и конструкциям FORALL. Она означает, что итерации цикла могут выполняться в произвольном порядке, то есть они не зависят друг от друга. INDEPENDENT в применении к FORALL означает отмену синхронизации после третьего этапа из четырех этапов, составляющих FORALL. Напомним порядок действий в FORALL.

1. Вычисление индексного выражения.
2. Вычисление маски для индексов.
3. Вычисление правых частей оператора присваивания для элементов с логической маской .TRUE.
4. Выполнение присваиваний.

Синхронизация проводится после каждого этапа, INDEPENDENT дает возможность перейти к выполнению присваиваний тогда, когда еще не все процессы-исполнители закончили вычисление правых частей.

Директива INDEPENDENT помещается непосредственно перед началом оператора цикла или конструкции FORALL, к которым она относится (листинг 13.13).

**Листинг 13.13. Примеры директивы *INDEPENDENT***

```
!HPF$ INDEPENDENT
DO I = 1, N
       A(I) = B(I) + C(I) * G(I) - D(I)
END DO
.....
!HPF$ INDEPENDENT
FORALL (I = 1:N, J = 1:M, A(I, J) .GT. 0) A(I, J) = REAL(I + J)
```

И циклы, и конструкции FORALL могут содержать вызовы подпрограмм. Помеченные директивой INDEPENDENT циклы и конструкции FORALL могут содержать только вызовы PURE-процедур и PURE-функций. Чтобы подпрограмма могла быть объявлена PURE-подпрограммой, она должна удовлетворять следующим требованиям:

- ❒ не производить ввода/вывода и операций с памятью;
- ❒ не менять глобальных переменных;
- ❒ не содержать операторов PAUSE и STOP;
- ❒ не иметь локальных переменных с атрибутом SAVE;
- ❒ не выравнивать локальные данные по глобальным данным;
- ❒ обращаться только к PURE-подпрограммам.

PURE-функции имеют еще дополнительное ограничение: их формальные параметры должны быть входными, т. е. должны быть описаны с атрибутом INTENT(IN) (листинг 13.14).

**Листинг 13.14. Пример описания и применения *PURE*-подпрограммы**

```
PROGRAM PureExample
IMPLICIT NONE
INTEGER :: I, N, REINT
PARAMETER (N = 20)
INTEGER, DIMENSION(N) :: J
!HPF$ INDEPENDENT
DO I = 1, N
       J(I) = REINT(I, N)
END DO
write(*, *) J
CONTAINS
```

```
PURE INTEGER FUNCTION REINT(K, L)
INTEGER,INTENT(IN) :: K, L
REINT = ABS(L - K)
END FUNCTION REINT
END PROGRAM PureExample
```

Программист сам должен решать, может ли его подпрограмма быть объявлена как PURE: контроль компилятора сводится к формальной проверке корректности вызовов.

Заметим, что все перечисленные ограничения для PURE-подпрограмм не касаются распределения данных. При общем допущении перераспределения данных, PURE-подпрограммы не должны перераспределять глобальные данные, ограничиваясь формальными параметрами и локальными переменными.

Инструкция-дополнение NEW используется в INDEPENDENT-циклах тогда, когда создание копий переменных для каждого процесса делает итерации цикла независимыми (листинг 13.15).

**Листинг 13.15. Пример использования инструкции *NEW***

```
!HPF$ INDEPENDENT, NEW(R1, R2)
DO I = 1, N
       R1 = SIN(A(I))
       R2 = COS(A(I))
       A(I) = R1 * R1 - R2 * R2
END DO
```

HPF реплицирует обычные скалярные переменные. Каждое изменение значения такой переменной приводит к синхронизации ее значений на всех копиях программы, что может вызвать неконтролируемый и, скорее всего, совершенно не нужный обмен данными. Именно это и происходило бы, если в примере из листинга 13.15 для переменных R1 и R2 не создавались локальные копии. Переменные, помеченные как NEW, не могут:

- использоваться вне цикла без переопределения;
- быть необязательным аргументом или указателем;
- иметь атрибут SAVE.

В конструкциях FORALL инструкция NEW не имеет смысла.

## Подпрограммы и распределение данных

Информация о привязке входных параметров подпрограмм не может передаваться в подпрограммы явно — для этого в базовом языке нет средств.

Формально каждая подпрограмма может использовать основные директивы HPF точно так же, как они применяются в главной программе. По спецификации HPF-2.0 для подпрограмм, имеющих явный интерфейс с директивами распределения формальных параметров, фактические параметры при передаче управления подпрограмме будут распределены в соответствии с директивами подпрограммы. При возврате управления в вызывающий программный компонент будет выполнено обратное перераспределение, побочные эффекты в виде изменения привязки глобальных данных исключаются. Предполагается, что подпрограмма, не имеющая явного интерфейса, не требует перераспределения данных. Если же в действительности перераспределение требуется, подпрограмма становится источником ошибок.

Все директивы распределения данных в подпрограммах действительны; некоторые ограничения в этой области накладываются на распределение формальных аргументов PURE-процедур и PURE-функций (см. предыдущий раздел). В листинге 13.16 приведен пример обращения к внешней процедуре, радикально меняющей привязку фактических параметров.

**Листинг 13.16. Пример вызова процедуры с перераспределением данных**

```
PROGRAM MAIN
IMPLICIT NONE
INTEGER, PARAMETER :: N = 3000
REAL, DIMENSION(N, N) :: A, B, PR
!HPF$ PROCESSORS, DIMENSION(2, 2) :: P
!HPF$ TEMPLATE, DIMENSION(N, N) :: T
!HPF$ ALIGN (:, :) WITH T(:, :) :: A, B, PR
!HPF$ DISTRIBUTE(BLOCK, BLOCK) ONTO P :: T
INTERFACE
       SUBROUTINE ACTION(X, Y, RES)
       IMPLICIT NONE
       REAL, DIMENSION(:, :), INTENT(IN) :: X, Y
       REAL, DIMENSION(:, :), INTENT(OUT) :: RES
       !HPF$ PROCESSORS, DIMENSION(4) :: Q
       !HPF$ TEMPLATE, DIMENSION(SIZE(X, DIM = 2))::W
       !HPF$ ALIGN (:, *) WITH W :: X, Y, RES
       !HPF$ DISTRIBUTE (BLOCK) ONTO Q :: W
       END SUBROUTINE ACTION
END INTERFACE
       CALL RANDOM(A)
       CALL RANDOM(B)
```

```
       CALL ACTION(A, B, PR)
END PROGRAM MAIN
SUBROUTINE ACTION(X, Y, RES)
IMPLICIT NONE
REAL, DIMENSION(:, :), INTENT(IN) :: X, Y
REAL, DIMENSION(:, :), INTENT(OUT) :: RES
!HPF$ PROCESSORS, DIMENSION(4) :: Q
!HPF$ TEMPLATE, DIMENSION(SIZE(X, DIM = 2)) :: W
!HPF$ ALIGN (:, *) WITH W :: X, Y, RES
!HPF$ DISTRIBUTE (CYCLIC) ONTO Q :: W
       RES = (X + Y) * X * Y
END SUBROUTINE ACTION
```

INHERIT — специальная директива для подпрограмм — указывает на наследование распределения данных от вызывающего компонента (листинг 13.17).

**Листинг 13.17. Передача информации о распределении данных в подпрограмму директивой *INHERIT***

```
PROGRAM MAIN
IMPLICIT NONE
INTEGER, PARAMETER :: N = 3000
REAL, DIMENSION(N, N)::A, B, PR
!HPF$ PROCESSORS, DIMENSION(2, 2) :: P
!HPF$ TEMPLATE, DIMENSION(N, N) :: T
!HPF$ ALIGN (:, :) WITH T(:, :) :: A, B, PR
!HPF$ DISTRIBUTE(BLOCK, BLOCK) ONTO P :: T
       CALL RANDOM(A)
       CALL RANDOM(B)
       CALL ACTION(A, B, PR)
END PROGRAM MAIN
SUBROUTINE ACTION(X, Y, RES)
IMPLICIT NONE
REAL, DIMENSION(:, :),INTENT(IN) :: X, Y
REAL, DIMENSION(:, :),INTENT(OUT) :: RES
!HPF$ INHERIT X, Y, RES
       RES = (X + Y) * X * Y
END SUBROUTINE ACTION
```

Часть аргументов подпрограмм может наследовать распределение от вызывающего модуля при местном распределении остальных аргументов. Пример такого комбинированного распределения приведен в листинге 13.18.

**Листинг 13.18. Пример комбинированного распределения данных в подпрограмме**

```
SUBROUTINE ACTION(X, Y, RES)
IMPLICIT NONE
REAL, DIMENSION(:, :), INTENT(IN) :: X, Y
REAL, DIMENSION(:,:), INTENT(OUT) :: RES
!HPF$ INHERIT RES
!HPF$ ALIGN (i, j) WITH RES(j, i) :: X, Y
...
END SUBROUTINE ACTION
```

Во избежание возможных осложнений HPF-2.0 запрещает применение директив `INHERIT` и `DISTRIBUTE` к одним и тем же объектам.

Информации о наследовании распределения данных может передаваться в подпрограмму в другой, так называемой транскриптивной форме. При этом обычные директивы HPF включают знаки `*` для указания объектов, в отношении которых описываемое директивой действие считается уже выполненным (листинг 13.19).

**Листинг 13.19. Транскриптивная форма передачи информации о распределении данных**

```
SUBROUTINE ACTION(X, Y, RES)
IMPLICIT NONE
REAL, DIMENSION(:, :), INTENT(IN) :: X, Y
REAL, DIMENSION(:, :), INTENT(OUT) :: RES
!HPF$ DISTRIBUTE * ONTO * :: X, Y, RES
...
END SUBROUTINE ACTION
```

Возможно также сочетание траскриптивных директив с директивой `INHERIT`.

В целом директивы транскриптивной формы наследования уступают директиве `INHERIT`, так как применимы только к объектам, распределенным директивой `DISTRIBUTE`, и, как показывает практика, используются не часто.

### Замечания к практическому применению

Сведения о HPF, изложенные в этой книге, обобщают найденную в Интернете информацию о спецификации HPF и различных ее толкованиях. Примеры программ, приведенных полностью, проверены в действии, но, к сожалению, на весьма ограниченном наборе реализаций. По существу, это был пакет PGI HPF (Portland Group Compiler Technology, название фирмы) поверх mpich или pvm на Linux-кластере и HPF того же производителя, работающий через разделяемую память на 2-процессорном сервере. Возможности HPF в действительности гораздо шире.

Если читатель не располагает доступом к суперкомпьютерным центрам с реализациями HPF, поставляемыми производителем, рекомендуем ему пакет PGI Server, распространяемый с 15-дневной бесплатной лицензией (**http://www.pgroup.com**). Для применения в ознакомительных целях достаточно и одного процессора под управлением UNIX-подобной операционной системы (например, Linux). Если конфигурация именно такая, то потребуется еще пакет pvm (**http://www.csm.ornl.gov/pvm/pvm_home.html**) и/или mpich (**http://www-unix.anl.gov/mpi/mpich**). Установка всех пакетов не требует специальной подготовки и даже прав суперпользователя, достаточно знания основ администрирования системы Linux, а также утилит rpm и make.

## OpenMP

Стандарт HPF оказался "слишком" универсальным — действия, управляемые его директивами, ограничены пересечением возможностей массивно-параллельных вычислительных систем (MPP, Massively Parallel Processing, процессор с массовым параллелизмом, массовая параллельная обработка) и мультипроцессоров с разделяемой памятью (SMP, Symmetric MultiProcessor, симметричный мультипроцессор, обеспечивает симметричную многопроцессорную обработку (Symmetrical MultiProcessing), при которой и операционная система, и приложения могут использовать любой доступный процессор). Однако принципиальное различие систем MPP и SMP типов очевидно: первые хорошо приспособлены для физического разделения данных, но не имеют эффективного механизма для управления параллельными действиями над ними, вторые же не имеют возможности эффективно разделять данные, но обладают очень гибким и мощным механизмом управления параллельными действиями. Этот механизм — библиотека Thread, позволяющая в рамках одного процесса создавать несколько потоков инструкций — подпроцессов, имеющих доступ к одному сегменту данных. Однопроцессорные системы искусно имитируют "одновременное" выполнение нескольких программ, передавая управление от одного подпроцесса другому. SMP-системы "прикрепляют" каждый подпроцесс к заданному процессору, и тогда исполнение становится действительно параллельным.

Разработка стандарта OpenMP последовала сразу же за разработкой HPF. У этого стандарта уже была некоторая предыстория, так как все производители SMP-систем создавали собственные расширения к языкам программирования, чтобы дать потребителям возможность использовать преимущества многопроцессорности. Стандарт сложился относительно легко и практически так же быстро, как и HPF — первая версия была выпущена 1998 году, вторая — в 2000.

OpenMP так же, как и HPF, является надстройкой на фундаменте Фортрана 90 и сводится к набору директив, которые обрабатываются препроцессором и выглядят комментариями базового языка. Набор инструкций обеспечивает исключительную гибкость в управлении параллельным исполнением, эффективность которого сопоставима с эффективностью HPF.

## Общая форма директив OpenMP и общие принципы их использования

По правилам OpenMP в программе должна быть выделена область параллельного исполнения (блок параллелизации) — одна или несколько. Эта область должна быть областью замкнутого кода, т. е. она не должна содержать переходов к внешним операторам или адресов возврата от них. Для исполнения операторов блока параллелизации создается группа подпроцессов (или нитей, или потоков исполнения), число которых назначается до начала блока или в директиве, открывающей блок. Поток исполнения, создавший группу, становится ведущим подпроцессом (или мастером) группы. Внутри области параллельного исполнения число подпроцессов сохраняется неизменным. По выходу из блока параллелизации все подпроцессы группы, за исключением мастера, закрываются. Директивами OpenMP в блоке параллелизации могут быть выделены блоки операторов, исполнение которых может быть поручено одному из подпроцессов группы. Например, оператор цикла может быть разделен на части так, чтобы каждая из них выполнялась отдельным подпроцессом. Независимые не итеративные блоки операторов также могут быть поручены различным подпроцессам. Операторы, параллельное выполнение которых невозможно или нежелательно, могут исполняться либо одним из подпроцессов группы, либо подпроцессом-мастером. Все директивы, управляющие исполнением, применимы только к блокам замкнутого кода. Области параллельного исполнения могут вкладываться друг в друга. В этом случае каждый подпроцесс из группы исполнения создает свою группу, в которой становится мастером.

Процесс параллельного исполнения корректируется директивами дополнений, управляющими раздачей данных, синхронизацией подпроцессов и обработкой данных по выходу из блока разделения заданий.

Директивы управления данными определяют заданные списками переменные как частные (`PRIVATE`) или разделяемые (`SHARED`). Для `PRIVATE`-переменных

каждый подпроцесс создает свои экземпляры каждой переменной из списка. Разделяемые переменные существуют в одном экземпляре и доступны для всех подпроцессов. Директивы выходной обработки данных определяют ту версию набора переменных, растиражированных для параллельной обработки, с которой исполнение будет продолжено после завершения блока разделения заданий или блока параллелизации.

Встроенные функции и подпрограммы OpenMP несут информацию о свойствах среды исполнения и позволяют устанавливать некоторые ее параметры. Информация о свойствах среды исполнения также содержится в переменных окружения OpenMP.

Общая форма записи директив OpenMP такова:

```
стартовый_знак директива [дополнение[ [,]дополнение]…]
замкнутый_блок_кода
стартовый_знак END директива [модификатор_завершения]
```

Вид стартового знака зависит от формата записи текста программы, всего же есть три варианта его написания:

```
!$OMP
C$OMP
*$OMP
```

Первая форма может использоваться как в свободном, так и в фиксированном форматах, остальные — только в фиксированном формате. В нем директива всегда начинается с первой позиции строки, например:

```
C23456789
!$OMP PARALLEL DO SHARED(X, Y),
```

или

```
C$OMP PARALLEL DO
C$OMP+SHARED(X, Y),
```

или

```
*$OMP PARALLELDOSHARED(X, Y)
```

В свободном формате `!$OMP` может начинать директиву с любой позиции строки, но после пробела:

```
!23456789
        !$OMP PARALLEL DO &
!$OMP[ ][&][ ]SHARED(X, Y) !$OMP PARALLELDOSHARED(X, Y)
```

Строки продолжения определяются в соответствии с правилами записи строк продолжения Фортрана.

Некоторые директивы OpenMP содержат необязательные пробелы и могут быть записаны как директива в вышеприведенных примерах в три, два или одно слово.

В дальнейшем мы будем придерживаться формы `!$OMP` стартового знака и не будем пропускать необязательные пробелы в записи директив.

Основная директива OpenMP, директива определения области параллелизации, имеет вид:

```
!$OMP PARALLEL [дополнение[ [,]дополнение]…]
замкнутый_блок_кода
!$OMP END PARALLEL
```

Пример определения области параллельного исполнения приведен в листинге 13.20.

**Листинг 13.20. Директива *PARALLEL***

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X, Y) NUM_THREADS(2)
CALL WORK(A, B, C, IX_NEXT, X)
CALL WORK(A, B, C, IY_NEXT, Y)
!$OMP END PARALLEL
```

В этом примере определяется область параллелизации из двух операторов. В группе исполнения будет создано два (директива дополнения `NUM_THREADS(2)`) подпроцесса. Все переменные, встречающиеся в области, будут тиражироваться для каждого подпроцесса (директива дополнения `DEFAULT(PRIVATE)`), кроме переменных `X` и `Y` (директива дополнения `SHARED(X, Y)`).

Дополнениями к директиве `PARALLEL` могут быть следующие директивы.

- `PRIVATE(список)` — определяет список тиражируемых переменных.
- `SHARED(список)` — определяет список разделяемых переменных.
- `DEFAULT(PRIVATE | SHARED | NONE)` — определяет тип переменных по умолчанию. Если задан параметр `NONE`, все переменные области должны быть упомянуты в одном из (`PRIVATE` или `SHARED`) списков.
- `FIRSTPRIVATE(список)` — указывает на необходимость копирования `PRIVATE`-переменных из исходных объектов в начале блока параллелизации.
- `REDUCTION({оператор|имя_встроенной_подпрограммы}:список)` — указывает метод выходной обработки растиражированных переменных.
- `COPYIN(list)` — детализирует список тиражируемых переменных из именованных `COMMON`-блоков, отмеченных директивой `THREADPRIVATE`.
- `IF(скалярное_логическое_выражение)` — определяет условия для выполнения директивы `PARALLEL`.
- `NUM_THREADS(скалярное_выражение_целого_типа)` — задает (если это позволено) число подпроцессов группы исполнения.

Более детальное описание директив дополнения будет приведено в разделе "Директивы дополнения" этой главы, здесь же дается краткое представление допустимых директив, необходимое для понимания возможностей основной директивы.

## Директивы разделения заданий

Директивы разделения заданий встраиваются внутрь области параллелизации, например, как в листинге 13.21.

**Листинг 13.21. Директива *DO* внутри блока директивы *PARALLEL***

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
       DO I = 1, N
       A(I) = B(I) + C(I)
       END DO
!$OMP END PARALLEL
CALL REVERSE(I)
```

К директивам разделения заданий относятся директивы `DO`, `SECTIONS`, `SINGLE` и `WORKSHARE`. Они определяют способ распределения нагрузки по подпроцессам группы исполнения.

Директива `DO` может применяться только к циклам `DO … END DO`. Она имеет вид:

```
!$OMP DO [[дополнение[, дополнение]…]
       замкнутый_цикл
!$OMP END DO [NOWAIT]
```

Подчеркнем, что директива `DO` может применяться только к безусловным циклам, не имеющим досрочного выхода за пределы директивного блока. Переменная цикла автоматически определяется локальной и может не указываться в списках директивы дополнения `PRIVATE`. Модификатор завершения `NOWAIT` отменяет синхронизацию подпроцессов группы исполнения на выходе из блока директивы `DO`. Директивы дополнений указывают на метод распределения итераций цикла по подпроцессам. Для директивы `DO` в таком качестве могут выступать различные конструкции.

- `PRIVATE(список)`.
- `FIRSTPRIVATE(список)`.
- `LASTPRIVATE(список)` — сохраняет в переменных основного экземпляра значения, полученные последним выполнявшимся подпроцессом.
- `REDUCTION({оператор | имя_встроенной_подпрограммы} : список)`.

- ORDERED — указывает на последовательное выполнение итераций в заданном между директивами ORDERED … END ORDERED блоке операторов. Используется только как дополнение директивы DO.
- SCHEDULE(тип[, порция]) — указывает, каким образом итерации цикла будут распределены по подпроцессам. Параметр тип задает метод разделения работы, параметр порция определяет количественную характеристику распределения. Тип может быть одним из следующих.

  STATIC — итерации разделяются на равные порции, заданные параметром порция, если он присутствует, и распределяются между подпроцессами по принципу ромашки в порядке возрастания номеров подпроцессов. Если второй параметр отсутствует, то порция итераций определяется делением числа итераций на число подпроцессов.

  DYNAMIC — итерации делятся на порции, заданные параметром порция, и раздаются подпроцессам. Подпроцесс, завершивший свою порцию итераций, получает следующую динамически. Если второй параметр не задан, используется значение 1.

  GUIDED — порции итераций распределяются в порядке экспоненциального убывания. Порция определяет размер наименьшей порции итераций, исключая, может быть, последнюю из них. Размер начальной порции зависит от реализации. Если второй параметр не задан, его значение подразумевается равным 1.

  RINTIME — распределение итераций определяется переменной окружения OMP_SCHEDULE, поведение при неопределенной переменной зависит от реализации.

Пример параллельного цикла с редукцией переменных приведен в листинге 13.22.

**Листинг 13.22. Редуцирующий параллельный цикл**

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A, B)
DO I = 1, N
CALL WORK(ALOCAL, BLOCAL)
A = A + ALOCAL
B = B + BLOCAL
END DO
$OMP END PARALLEL DO
```

Директива SECTIONS распределяет между подпроцессами нагрузку не итеративного характера. Директива записывается следующим образом:

```
!$OMP SECTIONS [дополнение[ [,]дополнение]…]
[!$OMP SECTION]
```

```
замкнутый_блок_операторов
[!$OMP SECTION
замкнутый_блок_операторов]
...
!$OMP END SECTIONS [NOWAIT]
```

Дополнениями может быть одна из следующих директив:

- `PRIVATE(список)`;
- `FIRSTPRIVATE(список)`;
- `LASTPRIVATE(список)`;
- `REDUCTION({оператор | имя_встроенной_подпрограммы} : список})`.

Директива `SINGLE` отмечает блок операторов, который должен выполняться одним из подпроцессов группы исполнения. Остальные подпроцессы группы приостанавливаются у барьера в директиве `END SINGLE` в ожидании работающего подпроцесса:

```
!$OMP SINGLE [[дополнение[, дополнение]…]
замкнутый_блок_операторов
!$OMP END SINGLE [модификатор_завершения]
```

Дополнениями могут быть директивы `PRIVATE(список)` и `FIRSTPRIVATE(список)`. Модификатором завершения может быть либо директива `NOWAIT`, либо цепочка директив `COPYPRIVARE(список)` (см. следующий раздел). Пример директивы `SINGLE` приведен в листинге 13.23.

**Листинг 13.23. Депараллелизация ввода и вывода директивой *SINGLE***

```
!$OMP PARALLEL DEFAULT(SHARED)
CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
CALL OUTPUT(X)
CALL INPUT(Y)
!$OMP END SINGLE
CALL WORK(Y)
!$OMP END PARALLEL
```

Директива `WORKSHARE` применяется к блоку операторов, которые могут быть распределены по подпроцессам автоматически, в соответствии с определенными правилами. Директива имеет вид:

```
!$OMP WORKSHARE
замкнутый_блок_операторов
!$OMP END WORKSHARE [NOWAIT]
```

Чтобы директива WORKSHARE действительно привела к разделению исполнения, блок операторов, заключенный между директивами WORKSHARE – END WORKSHARE, должен содержать только лексически завершенные конструкции PARALLEL, CRITICAL и ATOMIC, присваивания массивов и скаляров, операторы и конструкции WHERE и FORALL. Если в блоке содержится обращение к внешней функции, эта функция должна быть элементной.

Операторы блока замкнутого кода будут разделены на элементы исполнения по следующим принципам.

- В выражениях с массивами в каждом операторе, включая обращения к встроенным функциям редукции массивов, элементами считаются:
  - вычисление каждого элемента массивного выражения;
  - вычисление встроенных преобразующих функций для массивов может быть разделено на элементы произвольным образом. К этим функциям относятся MATMUL, DOT_PRODUCT, SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, ALL, SPREAD, PACK, UNPACK, RESHAPE, TRANSPOSE, EOSHIFT, CSHIFT, MINLOC и MAXLOC.
- В присваиваниях массивов каждое присваивание считается отдельным элементом.
- Каждое скалярное присваивание считается отдельным элементом.
- В обращениях к элементным функциям вычисление значения функции для каждого элемента любого аргумента-массива рассматривается как элемент исполнения.
- В операторе и конструкции WHERE каждое вычисление маски и каждое присваивание под управлением маски считаются элементами исполнения.
- В операторе и конструкции FORALL каждое вычисление маски, вычисления индексных диапазонов и управляемые назначения рассматриваются как элементы исполнения.
- Все операторы блока директивы ATOMIC (см. следующий раздел) рассматриваются как элементы исполнения.
- Все операторы блока директивы CRITICAL (см. следующий раздел) рассматриваются как элементы исполнения.
- Все конструкции PARALLEL, заключенные в блоке операторов директивы WORKSHARE, рассматриваются как элементы исполнения; каждая такая область будет исполняться новой группой подпроцессов исполнения.

Если ни одно из этих правил не применимо к операторам блока WORKSHARE, весь блок рассматривается как один элемент исполнения.

**Примечание**

Для одиночных циклов и блоков операторов, подходящих для применения к ним директив SECTIONS или WORKSHARE, допускается альтернативная форма директив, комбинирующих директиву PARALLEL с одной из директив DO, SECTIONS или WORKSHARE. Например:

```
!$OMP PARALLEL WORKSHARE
блок_операторов
!$OMP END PARALLEL WORKSHARE
```

или

```
!$OMP PARALLEL DO [дополнение[ [,]дополнение]…]
DO … END DO цикл
 [!$OMP END PARALLEL DO [модификатор_завершения]]
```

## Директива *THREADPRIVATE* и директивы дополнений

Директива THREADPRIVATE(список_имен_COMMON-блоков) определяет глобальную политику в отношении переменных, входящих в именованные COMMON-блоки. COMMON-блоки, перечисленные в списке параметров этой директивы, тиражируются для каждого подпроцесса. Переменные этих COMMON-блоков, оставаясь глобальными в рамках одного подпроцесса, будут недостижимы из других подпроцессов. Директива THREADPRIVATE должна размещаться в области последовательного кода, вне блоков действия директив параллелизации. Так как параметрами этой директивы являются имена COMMON-блоков, она должна находиться в области их видимости.

Основная директива PARALLEL и почти все директивы разделения заданий имеют один или несколько разделов дополнение, где размещаются директивы управления данными. Директивы управления данными определяют политику и порядок использования переменных в блоке операторов, выполняемом под управлением директив параллелизации. Переменные могут быть глобальными (один экземпляр для всех подпроцессов группы исполнения) или локальными (каждому подпроцессу — свой экземпляр). При выходе из области параллелизации локальные переменные потоков могут быть проинтегрированы в экземплярах соответствующих переменных потока-мастера или просто скопированы в них. В дополнениях могут появляться следующие директивы управления данными.

- PRIVATE(список) — переменные, перечисленные в списке директивы PRIVATE, назначаются локальными: каждый поток исполнения в группе получит свой экземпляр всех переменных списка.
- SHARED (список) — переменные, перечисленные в списке директивы SHARED, назначаются глобальными: все потоки исполнения в группе будут использовать один и тот же экземпляр всех переменных списка.

- DEFAULT(PRIVATE | SHARED | NONE) — эта директива определяет политику использования переменных по умолчанию. Все переменные, не встречающиеся в списках директив PRIVATE и SHARED, будут причислены к категории, заданной параметром директивы DEFAULT. Если этот параметр имеет значение NONE, все переменные должны будут обязательно определяться явно.
- FIRSTPRIVATE(список) — может применяться только к PRIVATE-переменным директивного блока. Предполагает, что копии PRIVATE-переменных из списка должны быть созданы до начала блока параллелизации.
- LASTPRIVATE(список) — может применяться только к PRIVATE-переменным директивного блока. Указывает, что по завершению блока операторов директивы разделения заданий в переменные из списка остающегося подпроцесса будут скопированы экземпляры PRIVATE-переменных подпроцесса, выполнявшего последнюю итерацию цикла DO или последний раздел SECTION. При отсутствии назначений переменных списка в последней итерации или в последнем блоке SECTION переменные остающегося экземпляра окажутся в состоянии неопределенности. При совмещении LASTPRIVATE с модификатором завершения NOWAIT переменные остающегося экземпляра будут находиться в состоянии неопределенности до первого барьера (см. директиву *BARRIER* следующего раздела).

**Листинг 13.24. Пример директивы *LASTPRIVATE***

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
       DO I = 1, N
       A(I) = B(I) + C(I)
       END DO
!$OMP END PARALLEL
    CALL REVERSE(I)
```

Как и в последовательном исполнении цикла, переменная I после его окончания будет равна N + 1.

- REDUCTION({оператор | имя_встроенной_подпрограммы} : список}) — может применяться только к PRIVATE-переменным директивного блока. Все экземпляры переменных из списка будут обработаны указанным образом, переменные остающегося подпроцесса получат значения, равные результату обработки. Способ обработки задается либо оператором, либо встроенной функцией. Оператором может быть один из операторов: +, -, *, .AND., .OR., .EQV. или .NEQV. Встроенной функцией — MAX, MIN,

IAND, IOR или IEOR. Если REDUCTION дополняет директивы WORKSHARE, то операторы последней должны ограничиваться следующим набором.

- ◊ x = x оператор выражение.
- ◊ x = выражение оператор x (исключая вычитание).
- ◊ x = имя_встроенной_функции(x, список_выражений).
- ◊ x = имя_встроенной_функции(список_выражений, x).

Здесь приняты обозначения:

x — скалярная переменная встроенного типа;

выражение — скалярное выражение, не включающее x;

список_выражений — разделенный запятыми список скалярных выражений, не включающих x. Если имя_встроенной_функции — одно из IAND, IOR или IEOR, то список_выражений должен содержать только одно выражение;

имя_встроенной_функции — одно из MAX, MIN, IAND, IOR или IEOR;

оператор — один из +, *, -, .AND., .OR., .EQV. или .NEQV.;

операторы в выражении должны иметь приоритет не ниже, чем оператор;

выражение x оператор выражение должно быть математически эквивалентно выражению x оператор (выражение), и выражение оператор x должно быть математически эквивалентно выражению (выражение) оператор x.

- ❒ COPYIN(список) — используется только в присутствии директивы THREADPRIVATE, детализируя набор тиражируемых переменных из ее списка_имен_COMMON-блоков. Позволяет тиражировать не полные COMMON-блоки, а только необходимые переменные из них.
- ❒ COPYPRIVATE(список) — используется только в присутствии директивы THREADPRIVATE как модификатор завершения директивы SINGLE. Указывает на необходимость копирования переменных списка из экземпляра подпроцесса, исполнявшего SINGLE блок, в экземпляры подпроцессов группы исполнения. Применяется только к переменным из именованных COMMON-блоков, отмеченных директивой THREADPRIVATE.

## Директива *MASTER* и директивы синхронизации

Директива MASTER выделяет внутри области параллелизации блок операторов, исполнение которых поручается подпроцессу-мастеру:

```
!$OMP MASTER
замкнутый_блок_операторов
!$OMP END MASTER
```

Директива не предполагает синхронизации подпроцессов на выходе из блока; остальные подпроцессы просто пропустят этот блок, приступив к выполнению оператора, следующего за директивой END MASTER.

Директива CRITICAL выделяет внутри параллельной области блок операторов, выполняемых подпроцессами поочередно. То есть в любой момент только один подпроцесс может обращаться к блоку CRITICAL. Директива имеет вид:

```
!$OMP CRITICAL [(имя)]
замкнутый_блок_операторов
!$OMP END CRITICAL [(имя)]
```

Необязательный параметр имя должен, если присутствует, идентифицировать блок однозначно: в программе не должно быть второго блока CRITICAL с таким же именем, и имя в начале директивы должно точно совпадать с именем в ее завершении. Все неименованные блоки CRITICAL считаются одноименными.

Директива BARRIER синхронизирует все подпроцессы в группе. Все подпроцессы будут приостановлены в месте директивы; исполнение будет продолжено, когда последний подпроцесс из группы достигнет ее. Директива имеет вид:

```
!$OMP BARRIER
```

Внутри блоков разделения заданий необходимо выполнение директивы BARRIER всеми подпроцессами, иначе выйти из такого блока программе не удастся.

Директива ATOMIC управляет порядком перезаписи в памяти значения переменной, обновляемой оператором, следующим непосредственно за директивой. Например:

```
!$OMP ATOMIC
x = x + B
```

Директива гарантирует последовательное, "атомарное" обновление значения переменной в памяти, не допускающее конкурирующего обращения подпроцессов к записи в одну и ту же ячейку памяти. Операторы, следующие за директивой, должны удовлетворять требованиям, предъявляемым к параметрам директивы дополнения REDUCTION (см. предыдущий раздел). Пример применения директивы ATOMIC приведен в листинге 13.25.

**Листинг 13.25. Упорядочивание обновления разделяемой переменной в памяти**

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X, Y, INDEX, N)
DO I = 1, N
CALL WORK(XLOCAL, YLOCAL)
```

```
!$OMP ATOMIC
X(INDEX(I)) = X(INDEX(I)) + XLOCAL
Y(I) = Y(I) + YLOCAL
```

Директива FLUSH приводит в состояние консистентности доступные подпроцессу глобальные переменные, т. е. производит синхронизацию этих переменных в памяти подпроцесса с их экземплярами в глобальной памяти. К моменту выполнения этой операции (к точке сведения) подпроцесс должен завершить все действия по чтению и записи; он должен быть уверен в том, что все видимые ему переменные в точке сведения обновлены другими подпроцессами и будут извлечены из памяти до их первого использования после точки сведения.

К видимым подпроцессу глобальным переменным относятся следующие категории переменных:

- глобальные переменные (из COMMON-блоков и модулей);
- переменные, ассоциированные через носитель;
- локальные переменные с атрибутом SAVE;
- переменные, ассоциированные в памяти (EQUIVALENCE) с видимыми подпроцессу переменными;
- локальные переменные с захваченным адресом;
- локальные переменные без атрибута SAVE, но объявленные разделяемыми во внешнем блоке параллелизации;
- формальные параметры;
- все разыменования ссылок.

Директива FLUSH обеспечивает соответствие между памятью исполняющегося потока и глобальной памятью. Чтобы достичь соответствия для всех потоков, необходимо, чтобы все потоки выполнили операцию FLUSH.

Директива FLUSH может быть применена явно, и тогда она имеет вид:

```
!$OMP FLUSH [(список)],
```

где список — перечень разделенных запятыми переменных, требующих подтверждения консистентности; если список отсутствует, FLUSH будет применяться ко всем видимым подпроцессу глобальным переменным.

Но чаще всего эта директива применяется неявно, так как действие, производимое ею, включается в действие других директив. FLUSH подразумевается следующими директивами:

```
BARRIER;
CRITICAL;
END CRITICAL;
```

END DO;

END SECTIONS;

END SINGLE;

END WORKSHARE;

ORDERED;

END ORDERED;

PARALLEL;

END PARALLEL;

PARALLEL DO;

END PARALLEL DO;

PARALLEL SECTIONS;

END PARALLEL SECTIONS;

PARALLEL WORKSHARE;

END PARALLEL WORKSHARE;

и не подразумевается директивами:

DO;

MASTER;

END MASTER;

SECTIONS;

SINGLE;

WORKSHARE.

Директива FLUSH не подразумевается также в присутствии модификатора завершения NOWAIT. Пример явного применения директивы FLUSH приведен в листинге 13.26.

**Листинг 13.26. Явное обращение к директиве *FLUSH***

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
IAM = OMP_GET_THREAD_NUM()
ISYNC(IAM) = 0
NEIGH = GET_NEIGHBOR (IAM)
!$OMP BARRIER
CALL WORK()
!Я готов, синхронизируюсь с соседями
ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
```

```
!Жду соседей
DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
END DO
!$OMP END PARALLEL
```

Директивы ORDERED и END ORDERED выделяют в области параллелизации блок операторов, в котором итерации должны выполняться в порядке их следования в цикле. Директива имеет вид:

```
!$OMP ORDERED
замкнутый_блок_операторов
!$OMP END ORDERED
```

Такая директива может появляться только в поле действия директив DO или PARALLEL DO с дополнением ORDERED в заголовке. Пример применения директивы ORDERED приведен в листинге 13.26.

**Листинг 13.26. Упорядочивание записи на диск в параллельном цикле**

```
!$OMP DO ORDERED SCHEDULE(DYNAMIC)
DO I = LB, UB, ST
CALL WORK(I)
END DO
...
SUBROUTINE WORK(K)
!$OMP ORDERED
WRITE(*, *) K
!$OMP END ORDERED
END
```

## Встроенные процедуры и функции OpenMP

Встроенные процедуры и функции OpenMp обеспечивают интерфейс приложения с многопоточным окружением исполнения: позволяют получать информацию о нем и устанавливать его параметры. Первая группа встроенных подпрограмм управляет параметрами многопоточного окружения, в котором определяется число доступных процессоров, максимальное и текущее число подпроцессов. К ним относятся:

- OMP_GET_NUM_PROCS — функция, возвращающая целое стандартной разновидности, равное числу доступных процессоров. Формат вызова:

```
INTEGER OMP_GET_NUM_PROCS()
```

- OMP_GET_MAX_THREADS — функция, возвращающая целое стандартной разновидности, равное максимально возможному числу подпроцессов. Формат вызова:

```
INTEGER OMP_GET_MAX_THREADS()
```

- OMP_IN_PARALLEL — функция, возвращающая логическое .TRUE. стандартной разновидности, если вызов ее был сделан из области параллелизации, и .FALSE. для области последовательного исполнения. Формат вызова:

```
LOGICAL OMP_IN_PARALLEL()
```

- OMP_GET_NUM_THREADS — функция, возвращающая целое стандартной разновидности, равное числу подпроцессоров текущей группы исполнения. При вызове из области последовательного исполнения возвращает 1. Формат вызова:

```
INTEGER OMP_GET_NUM_THREADS()
```

- OMP_SET_NUM_THREADS — процедура, устанавливающая число подпроцессов для ближайшей следующей группы исполнения равным значению указанного скалярного целого выражения. Процедура имеет действие только в области последовательного исполнения при условии, что динамическое назначение числа подпроцессов внутри программы разрешено. Имеет преимущество перед переменной окружения OMP_NUM_THREADS. Внутри области параллелизации или при отсутствии разрешения менять число подпроцессов ее действие игнорируется. Формат вызова:

```
CALL OMP_SET_NUM_THREADS(скалярное_целое_выражение)
```

- OMP_GET_THREAD_NUM — функция, возвращающая целое стандартной разновидности, равное порядковому номеру подпроцесса, совершившего вызов. Имеет смысл только в области параллелизации, в области последовательного исполнения всегда возвращает единицу. Формат вызова:

```
INTEGER OMP_GET_THREAD_NUM()
```

- OMP_SET_DYNAMIC — процедура, управляющая разрешением динамического изменения числа подпроцессов в группе исполнения ближайшей следующей области параллелизации. Имеет преимущество перед переменной окружения OMP_DYNAMIC. Устанавливается равным значению указанного скалярного логического выражения. Процедура имеет действие только в области последовательного исполнения, в области параллелизации игнорируется. Формат вызова:

```
CALL OMP_SET_DYNAMIC(скалярное_логическое_выражение)
```

- OMP_GET_DYNAMIC — функция, возвращающая логическое значение стандартной разновидности, сообщающее о разрешении (или запрещении) динамического изменения числа подпроцессов в группе исполнения следующей области параллелизации. Формат вызова:

```
LOGICAL OMP_GET_DYNAMIC()
```

- OMP_SET_NESTED — функция, управляющая разрешением вложенного параллелизма. Разрешает вложенный параллелизм, если значение указанного в параметре вызова скалярного логического выражения равно .TRUE. Имеет преимущество перед переменной окружения OMP_NESTED. Формат вызова:

```
CALL OMP_SET_NESTED(скалярное_логическое_выражение)
```

- OMP_GET_NESTED — функция, возвращающая логическое значение стандартной разновидности, сообщающее о разрешении (или запрещении) вложенного параллелизма. Формат вызова:

```
LOGICAL OMP_GET_NESTED()
```

Следующая группа встроенных процедур связана с управлением переменными-локерами. Локеры дают дополнительные возможности управления подпроцессами. В качестве локеров используются переменные целого типа с разновидностью, достаточной для помещения в переменной адреса. Переменные-локеры не должны использоваться нигде, кроме процедур управления локированием. Сначала переменная должна быть объявлена (инициализирована) локером. Подпроцесс может захватить локер: тогда другой подпроцесс, встретивший процедуру, предписывающую занять локер, захваченный другим подпроцессом, будет ожидать его освобождения. Переменную-локер можно разрушить, после этого захватить ее будет невозможно. Различаются два вида локеров: простые и вкладывающиеся. Вкладывающийся локер может быть захвачен повторно; простой локер можно захватить только один раз.

К локирующим процедурам относятся:

- OMP_INIT_LOCK, OMP_INIT_NEST_LOCK — процедуры инициализации переменной-локера (простого и вкладывающегося соответственно). Переменная-параметр процедур должна быть переменной целого типа подходящей разновидности, не используемая нигде в качестве локера. Формат вызова:

```
CALL OMP_INIT_LOCK(svar)
CALL OMP_INIT_NEST_LOCK(nvar)
```

- OMP_DESTROY_LOCK, OMP_DESTROY_NEST_LOCK — процедуры ликвидации локеров. Переменная-параметр процедур должна быть инициализированным локером (простым и вкладывающимся соответственно). Формат вызова:

```
CALL OMP_INIT_LOCK(svar)
CALL OMP_INIT_NEST_LOCK(nvar)
```

- OMP_SET_LOCK, OMP_SET_NEST_LOCK — процедуры захвата локера. Подпроцесс, исполняющий процедуру, должен ждать, пока локер освободится, и затем занять его. Для простых локеров право собственности на локер дается исполняющему процедуру подпроцессу, если локер свободен. Для вкладывающихся локеров право захвата локера подпроцесс получает,

если локер свободен или если он уже принадлежит этому подпроцессу. Переменная-параметр процедур должна быть инициализированным локером (простым и вкладывающимся соответственно). Формат вызова:

```
CALL OMP_SET_LOCK(svar)
CALL OMP_SET_NEST_LOCK(nvar)
```

- OMP_UNSET_LOCK, OMP_UNSET_NEST_LOCK — процедуры освобождения локера. Переменная-параметр процедур должна быть инициализированным локером (простым и вкладывающимся соответственно), принадлежащим подпроцессу. Формат вызова:

```
CALL OMP_UNSET_LOCK(svar)
CALL OMP_UNSET_NEST_LOCK(nvar)
```

- OMP_TEST_LOCK, OMP_TEST_NEST_LOCK — функции тестирования локеров (простого и вкладывающегося соответственно). Функции проверяют, доступен ли исполняющему подпроцессу указанный параметром локер, и пытаются захватить его. Их действие аналогично действию процедур OMP_SET_LOCK, OMP_SET_NEST_LOCK соответственно, но без ожидания освобождения локера. OMP_TEST_LOCK возвращает .FALSE., если локер занят, в противном случае, она занимает локер и возвращает .TRUE. OMP_TEST_NEST_LOCK возвращает в виде стандартного целого счетчик вложений. Формат вызова:

```
LOGICAL OMP_TEST_LOCK(svar)
INTEGER OMP_TEST_NEST_LOCK(nvar)
```

К третьей группе встроенных процедур относятся функции отсчета времени. Их две.

- OMP_GET_WTIME — функция, возвращающая в виде вещественного числа повышенной точности время, отсчитанное от некоторого момента в прошлом. Гарантируется, что за время работы программы точка отсчета не изменится. Формат вызова:

```
DOUBLE PRECISION OMP_GET_WTIME()
```

- OMP_GET_WTICK — функция, возвращающая в виде вещественного числа повышенной точности число секунд между двумя соседними отсчетами системного таймера. Формат вызова:

```
DOUBLE PRECISION OMP_GET_WTIME()
```

## Переменные окружения

OpenMP поддерживает несколько переменных окружения, характеризующих среду параллельного исполнения.

- OMP_NUM_THREADS — содержит число подпроцессов в группе исполнения. Если программа устанавливает число подпроцессов самостоятельно, зна-

чение OMP_NUM_THREADS используется как максимально допустимое число. Пример:

```
export OMP_NUM_THREADS = 16
```

- OMP_SCHEDULE — управляет политикой распределения нагрузки в параллельных циклах. Определяется по правилам директивы SCHEDULE *(см. разд. "Директивы разделения заданий" и "Директивы дополнений" данной гл.)*. Пример:

```
export OMP_SCHEDULE = "GUIDED, 4"
```

- OMP_DYNAMIC — управляет разрешением динамического изменения числа подпроцессов в группах параллельного исполнения. Пример:

```
export OMP_DYNAMIC = TRUE
```

- OMP_NESTED — управляет разрешением вложенного параллелизма. Пример:

```
export OMP_NESTED = FALSE
```

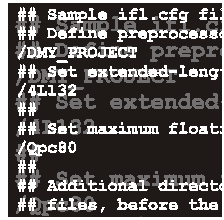
# Вопросы и задания для самостоятельной работы

1. Определите, как элементы массивов будут распределены по процессорам после выполнения следующих директив:

```
REAL, DIMENSION(16) :: A, C
REAL, DIMENSION(4,16) :: B, D
!HPF$ PROCESSORS(4) :: P
!HPF$ PROCESSORS(2, 2) :: Q
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: C
!HPF$ ALIGN (*, :) WITH A(:) :: B
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO Q :: B
!HPF$ ALIGN(:, *) WITH C(:) :: D
```

2. В листингах 13.16 и 13.17 приведены примеры различных способов передачи информации о распределении данных в подпрограмму для одной и той же задачи. Определите, какой из них содержит намеренно неэффективные инструкции. Убедитесь в правильности своего выбора на практике.

# Глава 14




# Фортран и другие языки программирования

В этой главе содержится краткий сравнительный обзор Фортрана и языков программирования Паскаль и C/C++. Читатель познакомится с некоторыми приемами смешанного программирования на Фортране и C. Здесь же дается описание конвертера f2c, который позволяет транслировать исходный текст программы на Фортране в текст на языке C.

## Фортран и Паскаль

Фортран и Паскаль относятся к одной группе языков структурного программирования. Их иногда называют *императивными языками*. Основными объектами в императивных языках являются переменные, операторы присваивания, стандартные алгоритмические конструкции. Императивные языки программирования привязаны к традиционной фон-неймановской архитектуре компьютера.

За длительную историю существования языка, прикладные программисты создали на Фортране огромное количество хорошо отлаженных, оптимизированных и прекрасно себя зарекомендовавших программ для научных и инженерно-технических расчетов. Имеется множество библиотек, предназначенных для численных расчетов и для работы с программами на Фортране. Компиляторы Фортрана, хотя и работают медленнее, чем многие из ныне применяющихся компиляторов Паскаля, генерируют хорошо оптимизированный исполняемый код. Эффективность компиляторов Фортрана — вне конкуренции. Следует отметить и доступность компиляторов Фортрана. Существуют не только коммерческие, но и свободно распространяемые компиляторы. Фортран — сравнительно простой язык. Он является основным языком на суперкомпьютерах, оснащен большим количеством встроенных математических функций, средствами управления точностью вычислений, разнообразными средствами работы с внешними файлами. Язык стандартизован, стандарты периодически обновляются с учетом тех достижений программирования, которые действительно представляют ценность для прикладного программиста-вычислителя. В отличие от Паскаля, Фортран поддерживает раздельную трансляцию.

В Фортране имеется встроенный комплексный тип. Фортран позволяет работать с символами национальных алфавитов как в комментариях, так и в операторах вывода. Специальные слова в Фортране являются ключевыми, а не зарезервированными. Программы на Фортране обычно хуже структурированы, чем программы на Паскале.

"Наследником" Паскаля является Оберон-2, однако и он уступает Фортрану в решении вычислительных задач.

# Фортран и С

Практика показывает, что тенденция к большему использованию в численных расчетах языков С и С++ поддерживается программистами, малоискушенными в решении вычислительных проблем, поэтому полезно знать о сильных и слабых сторонах каждого из этих языков. Сразу же следует отметить, что одним из важнейших достоинств Фортрана является то, что он лучше отвечает потребностям научных и технических расчетов, чем С или С++.

В Фортране аргументы подпрограмм могут быть массивами с переменной длиной, что дает возможность написания процедур без явного указания размерностей передаваемых им массивов. В стандартном С эта возможность отсутствует. В Фортране аргументы подпрограмм могут передаваться по ссылке, а это самый быстрый метод.

Фортран содержит богатый набор встроенных функций для работы с числовыми типами различной точности. Такие функции могут быть оптимизированы, например, написаны на ассемблере с оптимизацией использования кэш-памяти. Немаловажным является и наличие встроенной комплексной арифметики.

Операция возведения в степень в Фортране универсальна как по отношению к точности, так и к типу операндов. Ее реализация достаточно эффективна.

Фортран 90 поддерживает операции над сечениями массивов и использование векторных индексов. Он имеет набор встроенных функций для выполнения разнообразных операций с массивами. Границы изменения индексов массива могут быть произвольными, в отличие от С, где нижняя граница обязательно равна нулю. Средства работы с массивами делают Фортран удобным для параллельного программирования.

Фортран 90 поддерживает автоматический выбор типа числовых данных, имеющих заданную точность и диапазон, что делает программы на Фортране более универсальными.

Разнообразные средства ввода-вывода, например, неявные циклы, дают большую гибкость, чем стандартная библиотека С.

Параллельные расширения Фортрана 90 стандартизованы консорциумом High Performance Fortran и широко применяются при разработке параллельных программ для многопроцессорных вычислительных комплексов.

Фортран 90 поддерживает некоторые полезные возможности, заимствованные из языков C (свободный формат записи, указатели, динамические массивы и т. д.) и C++ (перегрузка операций, простые объекты).

В Фортране 77 нет явных указателей, что делает автоматическую оптимизацию более простой и эффективной. Это очень важно и для высокопроизводительных вычислений. Фортран 90 разрешает явно определенным указателям ссылаться только на переменные, описанные с атрибутом `TARGET`, что также облегчает автоматическую оптимизацию.

Фортран запрещает совмещение имен между переменными в `COMMON`-блоках и формальными аргументами.

Вышеперечисленные ограничения обеспечивают эффективную автоматическую оптимизацию кода компилятором, поэтому применение Фортрана позволяет добиться максимальной скорости выполнения программы.

На Фортране написано множество программ, причем значительная часть — это высококачественные, свободно распространяемые программы, предназначенные для решения разнообразных вычислительных задач. Программы для численных расчетов особенно сложно модифицировать, а их перевод на другие языки программирования является сложной и трудоемкой задачей, требующей больших затрат времени и средств.

Фортран 77 проще, чем C, поскольку подразумеваемая им модель компьютера проще. С другой стороны, Фортран 90 изменился в сторону увеличения разнообразия и сложности.

Стандарт языка C требует наличия только базовой математической библиотеки, в которой реализована лишь арифметика с двойной точностью. Если имеется только поддержка арифметических операций с двойной точностью, при выполнении операций со значениями, заданными с простой точностью, процессорное время расходуется нерационально. При этом может увеличиваться и погрешность выполнения арифметических операций. Стандарт Фортрана, с другой стороны, поддерживает арифметику как с простой, так и с двойной точностью. Иногда добавляется и четверная точность.

Фортран разрабатывался таким образом, чтобы сделать программирование численных расчетов более простым и ясным. Это обеспечивается следующими особенностями языка.

- Порядок вычисления арифметических выражений точно определен и может контролироваться.
- Возможность неявного описания типа позволяет сэкономить время на наборе исходного текста программы, хотя и увеличивает вероятность по-

явления ошибок, в том числе и таких, которые сложно определить и локализовать. Таким образом, один из недостатков языка можно считать одним из его достоинств. В отличие от многих других языков программирования, Фортран допускает два способа связывания переменной с ее типом — явное (с `IMPLICIT NONE`) и неявное.

- Нечувствительность к регистру позволяет избежать ошибок, связанных с нарушением регистра букв в именах.
- Отсутствие зарезервированных слов оставляет программисту полную свободу в выборе имен.
- Принцип размещения одного оператора в каждой строке делает исходные тексты программ более ясными и удобочитаемыми.
- Дополнительные пробелы несущественны (кроме, разумеется, строковых констант), что также повышает ясность исходных текстов программ на Фортране.
- Компоновка с математической библиотекой не требует специальных ключей компиляции. В C для этого требуется ключ `-lm`.
- Компиляторы Фортрана дают лучшую диагностику.

Можно считать, что различие между Фортраном и C — это различие между языком, предназначенным для численных расчетов, и языком системного программирования.

# Фортран и C++

Сравним только те особенности Фортрана 90 и C++, которые имеют значение для научно-технических расчетов, и не будем принимать во внимание поддержку объектно-ориентированного программирования (она рассматривалась в *гл.12*).

## Идентификаторы и специальные слова

В C++ специальные слова являются зарезервированными и набираются в нижнем регистре. В Фортране 90 используются ключевые слова, то есть смысл специального слова определяется контекстом, в котором оно используется.

## Записи

В C++ допускаются записи с вариантами, которые являются потенциально опасными конструкциями языка. Они, кроме того, неэффективны, так как требуют проверки вариантов во время выполнения программы. Имеются и агрегаты инициализации. В Фортране 90 есть записи и агрегаты инициализации. Оператор `SEQUENCE` позволяет хранить поля записи в порядке их определения.

## Многомерные массивы

Возможность работать с динамическими многомерными массивами значительно облегчает жизнь программиста. Статическое управление памятью в Фортране 77 приводило к необходимости создавать длинные списки параметров. В качестве параметров приходилось передавать временные рабочие массивы и размерности массивов.

В C++ имеются статические многомерные массивы, динамические одномерные массивы. Минимальное значение индекса всегда равно нулю.

В Фортране 90 допускаются статические и динамические многомерные массивы до семи измерений. Границы изменения индексов могут задаваться программистом. В списках параметров подпрограмм могут задаваться нижние границы интервала индексов.

## Неинициализированные переменные

Использование неинициализированных переменных может привести к неправильным результатам или ошибкам времени выполнения. Неинициализированные указатели могут стать причиной сбоя исполняющей системы. В обоих языках допускается использование неинициализированных переменных.

## Указатели

Указатели часто являются источником ошибок и снижают переносимость программы. В C++ вероятность такого рода ошибок велика. Указатели в Фортране 90 являются особыми объектами. Они более безопасны.

## Висячие ссылки

Указатели часто используются для организации динамических структур данных и абстрактных типов данных. Если язык допускает явное открепление объектов, непреднамеренное разыменование висячих указателей может привести к сбою операционной системы. В C++ и Фортране 90 такая ситуация возможна.

## Логические объекты и операции

В C++ имеется логический тип, а также битовые операции над символами, перечислениями, целыми значениями и битовыми полями. Логические выражения возвращают целые значения. Фортран 90 содержит тип `LOGICAL` и логические операции. Результат логических операций в Фортране также является логическим.

## Отношения

В C++ отношения определены для символов, перечислений, числовых типов и указателей. В Фортране 90 отношения определены для строк и числовых типов (кроме комплексных), а отношения равенства и неравенства определены и для комплексных типов.

## Арифметические операции

В C++ операции `+` и `-` применяются к символам, перечислениям, числовым типам и указателям. Операции `*` и `/` применяются к символьным и числовым типам, а также к перечислениям. Операция `%` применяется к символьным данным и перечислениям. Комплексного типа нет. В Фортране 90 операции `+`, `-`, `*`, `/` и `**` применяются к целым, вещественным и комплексным типам.

## Подпрограммы

В C++ соответствие типа и числа параметров проверяется, за исключением некоторых процедур (например, `printf`), где количество параметров может быть любым. Допускаются необязательные параметры. Передача параметров возможна по значению (модифицируется локальная копия параметра), по ссылке (модифицируется параметр) и по ссылке с префиксом `const` (только для чтения). Массивы нельзя передавать по значению. Количество элементов массива необходимо передавать отдельно. Процедурные переменные являются указателями на функции.

В Фортране 90 обязательно выполняется проверка соответствия формальных и фактических параметров подпрограмм по типу и числу. Разрешены необязательные параметры. Передача параметров возможна по значению и по ссылке. Программист может присвоить параметрам атрибуты: `IN` для немодифицируемых параметров, `OUT` для возвращаемых и `INOUT` для модифицируемых параметров. Количество элементов массива можно передавать отдельно (массивы с подразумеваемой формой, автоматические массивы). Процедурные переменные должны быть внешними или модульными процедурами.

## Рекурсия

В обоих языках рекурсия поддерживается, в Фортране 90 — если процедура объявлена рекурсивной.

## Перегрузка процедур и операций

В обоих языках разрешены: перегрузка процедур и операций, определение новых операций.

## Исключения и обработка ошибок

В C++ обработка ошибок выполняется обработчиком исключений. В Фортране 90 это невозможно.

## Поддержка параллелизма

Поддержка параллелизма необходима при решении сложных задач, требующих проведения трудоемких и объемных расчетов. В C++ нет поддержки параллелизма. В Фортране 90 также нет явной поддержки параллелизма, однако имеются параллельные расширения языка.

## Программы и единицы трансляции

Программа состоит из одной или нескольких *единиц трансляции*, каждая из которых содержит описания данных и код. Единицы трансляции могут компилироваться и храниться раздельно. Интерфейс единицы трансляции управляет доступом к ее объектам. Согласованность интерфейсов объектов программы означает совпадение интерфейсов.

В C++ единицами трансляции являются файлы. Описание интерфейсов содержится в заголовочных файлах. Согласованность интерфейсов не проверяется.

В Фортране 90 программа состоит из обязательной единицы трансляции — главной программы и необязательных единиц — модулей, внешних подпрограмм. Доступом к объектам модуля можно управлять с помощью атрибутов `PRIVATE` и `PUBLIC`. Согласованность интерфейсов не проверяется, но пользователь может скопировать интерфейсный блок непосредственно в единицу трансляции. В этом случае согласованность интерфейсов будет проверяться локально.

## Некоторые причины более низкой эффективности программ на языке C++

При выполнении операций с массивами для того, чтобы сохранить естественную нотацию, в C++ приходится использовать перегрузку операций, например:

```
z = w + x + y
```

где `w`, `x`, `y` и `z` являются одномерными массивами. Перегруженные операции в C++ всегда выполняются попарно. При вычислении выражения суммы трех векторов создаются два временных массива и три цикла. В этом случае для больших векторов проигрыш в производительности, по сравнению с Фортраном, может быть в 2–3 раза, а для маленьких векторов — почти на порядок, так как в этом случае преобладают потери на размещение временных объектов.

Язык C++ позволяет работать на высоком уровне абстракции. В этом случае компиляторы языка не выполняют некоторые виды оптимизации, что приводит к более низкой эффективности программ на C++. Это — "плата за абстракцию".

Компиляторы C++ учитывают возможность ссылки двух указателей на один элемент данных и не выполняют некоторые виды оптимизации. В Фортране такое совмещение отсутствует, что расширяет возможности оптимизации.

Мы перечислили лишь некоторые из причин более низкой эффективности программ на языке C++, более подробную информацию можно найти в специальной литературе.

# Конвертер f2c

Конвертер f2c позволяет транслировать исходный текст программы с языка Фортран 77 на язык C/C++ и создан в AT&T Bell Laboratories. Имеются версии для операционных систем UNIX и MS Windows. Автоматическое преобразование с Фортрана на C позволяет создавать переносимые программы на C, использующие исходный текст на Фортране. Если имеется исходный текст отлаженной и работающей программы на Фортране, а ее необходимо запустить на машине, на которой есть только компилятор C, можно использовать конвертер f2c.

Некоторую неуклюжесть результата преобразования разработчики объясняют строгим следованием стандарту Фортрана 77. Так, например, некоторые операторы ввода-вывода заменяются серией обращений к процедурам библиотеки ввода-вывода libI77. Результат преобразования обычно требует вмешательства программиста для ручной доработки текста.

## Имена

Конвертер f2c обрезает имена, содержащие более 50 символов. Имена могут содержать символы подчеркивания. Регистр букв в идентификаторах программы на Фортране преобразуется в нижний (если при вызове конвертера не используется ключ `-U`). К именам подпрограмм и общих блоков добавляется один символ подчеркивания, если они не содержат подчеркиваний, и два подчеркивания, если содержат. Так имена (Фортран) подпрограмм `ABC`, `A_B_C` и `A_B_C_` превращаются в имена функций (C) `abc_`, `a_b_c__` и `a_b_c___`.

## Типы

В табл. 14.1 приведено соответствие описаний типов на Фортране и C. Описания C используют типы, определенные в файле f2c.h, который используется при выполнении преобразований конвертером f2c.

***Таблица 14.1.*** *Соответствие описаний типов в Фортране и C*

| Фортран | Стандартный C | f2c.h |
|---|---|---|
| INTEGER * 2 X | shortint x; | short int x; |
| INTEGER X | integer x; | long int x; |
| LOGICAL X | long int x; | long int x; |
| REAL X | real x; | float x; |
| DOUBLE PRECISION X | doublereal x; | double x; |
| COMPLEX X | complex x; | struct {float r, i;} x; |
| DOUBLE COMPLEX X | doublecomplex x; | struct {double r, i;} x; |
| CHARACTER * 6 X | char x[6]; | char x[6]; |

В Фортране переменные целого, логического и вещественного типа занимают одинаковый объем памяти. Переменные типа `DOUBLE PRECISION` и `COMPLEX` занимают объем, в два раза больший. Тип C следует выбирать с учетом этого обстоятельства. От соблюдения этого правила зависит корректность преобразования операторов Фортрана `EQUIVALENCE` и `DATA`. Иногда для соблюдения этого правила требуется модификация файла f2c.h.

## Возвращаемые значения

Функциям Фортрана типа `INTEGER`, `LOGICAL` или `DOUBLE PRECISION` соответствуют функции C, возвращающие значение аналогичного типа. При вызове конвертера с ключом `-R` соответствие имеется и для функций типа `REAL`, в противном случае вещественной функции Фортрана отвечает функция C, возвращающая значение типа `doublereal`. Комплексные функции эквивалентны функциям C с дополнительным аргументом, указывающим положение возвращаемого значения в памяти. Так, комплексной функции Фортрана

```
COMPLEX FUNCTION F(...)
```

соответствует функция C:

```
void f_(temp, ...)
complex * temp;
...
```

Символьная функция Фортрана подобна функции C с двумя дополнительными аргументами, задающими адрес данных и их размер. Функции:

```
CHARACTER * 15 FUNCTION G(...)
```

соответствует функция C:

```
g_(result, length, ...)
```

```
char * result;
ftnlen length;
...
```

Она может вызываться следующим образом:

```
char chars[15];
...
g_(chars, 15L, ...);
```

## Списки аргументов

Аргументы подпрограмм в Фортране передаются по ссылке. Для каждого аргумента символьного типа, не являющегося функцией, передается аргумент, задающий длину значения. Длины строк являются величинами типа `long int`, передаваемыми по значению. Последовательность аргументов подпрограммы такова: основные аргументы, дополнительные аргументы для комплексных и символьных функций, адрес каждого значения или функции и длина каждого символьного аргумента. Таким образом, вызов подпрограммы на Фортране:

```
EXTERNAL F
CHARACTER * 7 S
INTEGER B(3)
...
CALL SAM(F, B(2), S)
```

соответствует такой последовательности операторов C:

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 7L);
```

Первый элемент массива в C всегда имеет нулевой индекс, а в Фортране по умолчанию это значение равно 1. Поскольку в Фортране массивы хранятся по столбцам, а в C — по строкам, f2c транслирует многомерные массивы Фортрана в одномерные массивы C с соответствующими индексными выражениями.

## Расширения Фортрана 77, поддерживаемые f2c

Конвертер f2c поддерживает расширения Фортрана 77. Одним из таких расширений является тип `DOUBLE COMPLEX` (`COMPLEX * 16`). Имена встроенных функций для комплексного типа с двойной точностью начинаются с буквы `Z`.

Вместо обозначения типа в операторе `IMPLICIT` можно использовать ключевое слово `UNDEFINED`. В этом случае переменные, имена которых начинаются с указанных далее букв, должны быть описаны явно. Конвертер f2c распознает и конструкцию Фортрана 90:

```
IMPLICIT NONE
```

как эквивалентную ей

```
IMPLICIT UNDEFINED(A - Z)
```

При вызове конвертера с ключом `-u` выполняется подстановка в начало каждой процедуры Фортрана предложения:

```
IMPLICIT NONE
```

В предложениях описания типа и операторе `IMPLICIT` можно использовать ключевые слова `STATIC` и `AUTOMATIC`. Они задают классы хранения переменных. У статической переменной имеется только одна копия, такие переменные сохраняют свои значения между вызовами процедуры, в которой они используются. Каждый вызов процедуры порождает новые копии автоматических переменных. Автоматические переменные нельзя использовать в операторах `EQUIVALENCE`, `DATA`, `NAMELIST` и `SAVE`. Ключ `-a` задает автоматический класс хранения для всех переменных, кроме тех, которые используются в общих блоках, а также в вышеперечисленных операторах.

Табуляция в первых шести позициях служит признаком свободного формата записи исходного текста. Символ `&` в первой позиции означает, что данная строка является строкой продолжения в свободном формате записи. Допускаются комментарии в стиле Фортрана 90.

Регистр букв игнорируется везде кроме строковых значений, если только при вызове f2c не используется ключ `-U`.

Оператор `INCLUDE 'svetl'`

заменяется содержимым файла svetl. Если имя файла не является абсолютным маршрутным именем, f2c сначала ищет svetl в рабочем каталоге. Если файл найти не удается, поиск продолжается в каталогах, указанных после ключа `-Idir`. Конструкции `INCLUDE` могут быть вложенными.

Транслятор f77 допускает использование двоичных, восьмеричных и шестнадцатеричных констант в операторах `DATA`. Конвертер f2c допускает использование таких констант в любом месте программы. Задаваться они могут в двух форматах. В первом случае сначала указывается буква, а затем значение в кавычках. Во втором случае сначала указывается основание системы (в десятичной форме), затем символ `#` и значение без кавычек. Двоичные константы обозначаются буквами `b` или `B`, восьмеричные — буквами `o` или `O` и шестнадцатеричные — буквами `x`, `X`, `z` или `Z`, например:

```
z'b5'
16#b5
o'351'
```

Строковые значения в программе на Фортране могут содержать Escape-последовательности, приведенные в табл. 14.2. Это требуется для совместимости с языком C.

***Таблица 14.2.*** *Escape-последовательности, распознаваемые конвертером f2c*

| Escape-последовательность | Описание | Escape-последовательность | Описание |
|---|---|---|---|
| `\0` | Пустой | `\n` | Новая строка |
| `\\` | `\` | `\r` | Возврат каретки |
| `\b` | Возврат | `\t` | Горизонтальная табуляция |
| `\f` | Прогон страницы | `\v` | Вертикальная табуляция |
| `\'` | Апостроф | `\"` | Кавычки |
| `\x` | `x`, где `x` — любой символ | | |

При вызове с ключом `-!bs` эти последовательности конвертером f2c не распознаются.

Имеются встроенные функции `AND`, `OR`, `XOR`, `LSHIFT`, `RSHIFT` и `NOT`, выполняющие поразрядные операции с целыми и логическими операндами.

Библиотека libF77 содержит две функции доступа к аргументам командной строки: `IARGC(DUMMY)` возвращает количество аргументов, а `GETARG(K, C)` присваивает строковое значение `C` `K`-му аргументу командной строки (или пробелы, если `K` находится за пределами допустимых значений).

Расширение Фортрана 77 для конвертера f2c включает рекурсию.

Переменные из общих блоков (`COMMON`) могут инициализироваться оператором `DATA` в любом модуле, а не только в `BLOCK DATA`.

Цикл может завершаться оператором `END DO`. Допускаются неименованные циклы `DO WHILE`.

Конвертер f2c распознает синонимы отношений в духе Фортрана 90: `<`, `<=`, `==`, `>=`, `>` и `<>`.

Допускается ввод-вывод, управляемый списком и `NAMELIST`. Конструкция `NAMELIST` работает так же, как в Фортране 90, за исключением того, что индексы должны иметь вид:

```
subscript [ : subscript [ : stride ] ]
```

В операторе `OPEN` конструкция `NAME =` трактуется как `FILE =`.

## Примеры

Конвертер запускается командой:

```
f2c [ключи] имя_файла.f
```

Результатом ее выполнения является файл с текстом на языке C. Его имя совпадает с именем исходного файла, а суффикс — `c`.

Если, например, конвертируется файл prog.f:

```
f2c prog.f
```

то результат записывается в файл prog.c. Для его компиляции следует использовать команду:

```
cc -o prog prog.c -lF77 -lI77 -lm
```

Библиотеки, подключаемые при использовании опций `-lF77` и `-lI77`, обеспечивают поддержку стандартного Фортрана.

Преобразование нескольких файлов с записью результата в единый файл выполняется с помощью конвейера команд:

```
cat *.f | f2c > progmy.c
```

Здесь f2c используется в качестве фильтра.

Познакомимся с назначением некоторых ключей f2c. При трансляции с ключом `-R` выходной файл содержит операции с простой точностью, а без него — операции с двойной точностью.

При трансляции с ключом `-A` конвертер f2c использует конструкции ANSI C. Для использования синтаксиса C++ необходим ключ `-C++`, например:

```
cat *.f | f2c -C++ -R > progmy.c
```

В ANSI C можно использовать прототипы, т. е. описания интерфейсов функций, включающие их имена, списки и типы аргументов. С ключом `-P` конвертер f2c создает файл file.P прототипов для подпрограмм, объявленных в каждом входном файле file.f. Можно объединить все файлы с прототипами в один заголовочный файл и включить его в файлы программы на языке C директивой `#include`.

Рассмотрим пример преобразования программы с помощью конвертера f2c. Исходный текст функции на Фортране:

```
FUNCTION DOT(N, X, Y)
 INTEGER N
 REAL X(N), Y(N)
 DOT = 0
 DO 10 I = 1, N
 10 DOT = DOT + X(I) * Y(I)
END
```

преобразуется в следующий текст на языке C:

```
/* dot.f -- translated by f2c (version 19950314).
You must link the resulting object file with the libraries:
-lf2c -lm (in that order)
*/
#include "f2c.h"
doublereal dot_(n, x, y)
integer *n;
real *x, *y;
{
/* System generated locals */
integer i__1;
real ret_val;
/* Local variables */
static integer i;
/* Parameter adjustments */
--y;
--x;
/* Function Body */
ret_val = (float)0.;
i__1 = *n;
for (i = 1; i <= i__1; ++i) {
/* L10: */
ret_val += x[i] * y[i];
}
return ret_val;
} /* dot_ */
```

Результирующий файл начинается строками комментария "…`translated by f2c`…" и директивы `#include`, включающей заголовочный файл f2c.h. Конвертер f2c преобразует имена в нижний регистр и добавляет символ подчеркивания к внешнему имени `dot`. С помощью операций `--x` и `--y` учитываются соглашения об индексации массивов в C. Метки сохраняются в комментариях для удобства ориентирования в тексте (`L10`).

Внутри функции ссылка на ее имя заменяется локальной переменной `ret_val`, которая хранит возвращаемое значение. Без ключа `-R` конвертер f2c преобразует возвращаемый тип вещественной функции в `doublereal`.

Конвертер f2c может удалять лишние скобки. Например, выражения в программе на Фортране:

```
X = A * (B * C)

Y = (A * B) * C
```

преобразуются в выражения на языке C:

```
x = a * (b * c);
y = a * b * c;
```

Конвертер f2c преобразует оператор ввода-вывода в вызовы подпрограмм библиотеки libI77 (эта библиотека входит в состав f2c). Если в программе на Фортране имеются описания:

```
INTEGER COUNT(10)
REAL VAL(10),
```

то оператор

```
READ(*, *) COUNT, VAL
```

порождает следующую последовательность описаний на языке C:

```
static integer c__3 = 3;
static integer c__10 = 10;
static integer c__4 = 4;
...
/* Builtin functions */
integer s_rsle(), do_lio(), e_rsle();
...
/* Fortran I/O blocks */
static cilist io__1 = { 0, 5, 0, 0, 0 };
```

и последовательность исполняемых операторов:

```
s_rsle(&io__1);
do_lio(&c__3, &c__10, (char *)&count[0], (ftnlen)sizeof(integer));
do_lio(&c__4, &c__10, (char *)&val[0], (ftnlen)sizeof(real));
e_rsle();
```

Неявные циклы в списках ввода-вывода, например:

```
READ(*, *) (COUNT(I), VAL(I), I = 1, 10),
```

порождают явные циклы C:

```
s_rsle(&io__4);
for (i = 1; i <= 10; ++i) {
do_lio(&c__3, &c__1, (char *)&count[i - 1], (ftnlen)sizeof(integer));
do_lio(&c__4, &c__1, (char *)&val[i - 1], (ftnlen)sizeof(real));
}
e_rsle();
```

Спецификаторы `END` и `ERR` в операторах ввода-вывода уменьшают читабельность результирующего кода, так как требуют дополнительных проверок. Текст на Фортране:

```
READ(*, *, ERR = 10) COUNT, VAL
10 CONTINUE
```

превращается в текст на C:

```
i__1 = s_rsle(&io__1);
if (i__1 != 0) {
goto L10;
}
i__1 = do_lio(&c__3, &c__10, (char *)&count[0], (ftnlen)sizeof(integer));
if (i__1 != 0) {
goto L10;
}
i__1 = do_lio(&c__4, &c__10, (char *)&val[0], (ftnlen)sizeof(real));
if (i__1 != 0) {
goto L10;
}
i__1 = e_rsle();
L10:
;
```

Общие блоки Фортрана транслируются в структуры C. Фрагмент текста на Фортране:

```
COMMON /NAMED/ C, D, R, I, M
COMPLEX C(10)
DOUBLE PRECISION D(10)
REAL R(10)
INTEGER I(10)
LOGICAL M(10)
IF(M(I(2))) D(3) = D(4) / D(5)
```

преобразуется в:

```
struct {
complex c[10];
doublereal d[10];
real r[10];
integer i[10];
logical m[10];
} named_;
#define named_1 named_
...
if(named_1.m[named_1.i[1] - 1]) {
named_1.d[2] = named_1.d[3] / named_1.d[4];
}
```

При использовании ключа `-p` в данном примере читабельность результата трансляции улучшается:

```
...
#define c (named_1.c)
#define d (named_1.d)
#define r (named_1.r)
#define i (named_1.i)
#define m (named_1.m)
...
if (m[i[1] - 1]) {
d[2] = d[3] / d[4];
```

Операции с символьными строками в программе на Фортране порождают вызовы функций в программе на C. Например, текст:

```
CHARACTER * (*) FUNCTION CAT(A, B)
CHARACTER * (*) A, B
CAT = A // B
END
```

преобразуется в текст:

```
...
static integer c_ _2 = 2;
/* Character */ int cat_(ret_val, ret_val_len, a, b, a_len, b_len)
char *ret_val;
ftnlen ret_val_len;
char *a, *b;
ftnlen a_len;
ftnlen b_len;
{
/* System generated locals */
address a__1[2];
integer i__1[2];
/* Builtin functions */
/* Subroutine */ int s_cat();
/* Writing concatenation */
i__1[0] = a_len, a__1[0] = a;
i__1[1] = b_len, a__1[1] = b;
s_cat(ret_val, a__1, i__1, &c__2, ret_val_len);
} /* cat_ */
```

Конвертер f2c преобразует главную программу на Фортране `MAIN` в функцию `MAIN__`, которая при завершении выполнения программы автоматически закрывает файлы, выводит сообщение об ошибках и обеспечивает правильную работу функций доступа к аргументам командной строки `IARGC` и `GETARG`.

Кроме конвертера f2c имеются программы, выполняющие обратное преобразование — трансляцию с языка C на Фортран, а также конвертеры с Фортрана на язык Java. Они не получили широкого распространения, поэтому рассматривать их здесь мы не будем.

# Смешанное программирование на Фортране и C/C++

Языки C/C++ и Фортран имеют различный синтаксис. Вместе с тем они являются основными языками программирования вычислений. Язык C часто применяется в системном программировании. C++ является объектно-ориентированным языком, поэтому он удобен для использования в больших проектах, где важны эффективность разработки программ и их надежность, а производительность не играет большой роли. Иногда оказывается удобным при решении сложной задачи разбить ее на части и при решении каждой из подзадач использовать наиболее подходящий для этого язык программирования. В этом случае важна возможность взаимодействия частей программы, написанных на разных языках программирования и откомпилированных по отдельности. Познакомимся с некоторыми примерами совместного использования языков C/C++ и Фортрана.

Напомним о тех различиях синтаксиса языков C/C++ и Фортран, которые относятся к программированию вычислений. В именах C/C++ различается регистр букв, а в Фортране он не имеет значения. В Фортране 77 длина строки не может превосходить 72 позиции, а первые 6 позиций зарезервированы для специальных целей. В C/C++ и Фортране 90 используется свободный формат записи исходного текста.

Обозначения арифметических операций совпадают в Фортране и C/C++. Обозначения отношений, логических операций и поразрядных операций с целыми числами различаются. Они приведены в табл. 14.3 и 14.4.

***Таблица 14.3.*** *Логические отношения и операции в Фортране и C/C++*

| Отношение, операция | Фортран 90 | C/C++ |
|---|---|---|
| Равенство | == | == |
| Неравенство | /= | != |
| Меньше | < | < |

***Таблица 14.3*** *(окончание)*

| Отношение, операция | Фортран 90 | C/C++ |
|---|---|---|
| Больше | > | > |
| Меньше или равно | <= | <= |
| Больше или равно | >= | >= |
| Логическое отрицание | .NOT. | ! |
| Логическое И | .AND. | && |
| Логическое ИЛИ | .OR. | \|\| |
| Логическая эквивалентность | .EQV. | |
| Логическая неэквивалентность | .NEQV. | |

***Таблица 14.4.*** *Битовые операции*

| Операция | Фортран 90 | C/C++ |
|---|---|---|
| Поразрядное И | IAND | & |
| Поразрядное ИЛИ | IOR | \| |
| Поразрядное исключающее ИЛИ | IEOR | ^ |
| Поразрядное отрицание | NOT | ~ |
| Сдвиг влево линейный | ISHFT | << |
| Сдвиг вправо линейный | ISHFT | >> |
| Циклический сдвиг | ISHFTC | |

Соответствие типов и объявлений функций Фортрана и C приведено в табл. 14.5.

***Таблица 14.5.*** *Соответствие типов и объявлений функций Фортрана и C*

| Фортран | C |
|---|---|
| INTEGER*2 I | short i; |
| INTEGER I | int i; |
| | long i; |
| LOGICAL L | int l; |
| | long l; |
| REAL X | float x; |
| DOUBLE PRECISION D | double d; |
| COMPLEX C | struct {float r, i;} c; |

***Таблица 14.5*** *(окончание)*

| Фортран | C |
|---|---|
| DOUBLE COMPLEX Z | struct {double dr, di;} z; |
| LOGICAL L | int l |
| CHARACTER*(N) C | char c[n]; |
| INTEGER FUNCTION RF(...) | int rf_(...) |
| DOUBLE PRECISION FUNCTION DF(...) | double df_(...) |
| DOUBLE COMPLEX FUNCTION ZF(...) | zf_(temp,...) |
| | struct {double r, i;} *temp; |
| | (значение возвращается через temp) |
| CHARACTER*8 FUNCTION CHF(...) | chf_(ret, len, ...) |
| | char ret[]; (значение chf_) |
| | int len; (размер ret[]) |
| SUBROUTINE SUB(...) | int sub_(...) |

## Вызов подпрограмм Фортрана из программы на C++

Пусть имеются процедура и функция, написанные на Фортране:

```
      SUBROUTINE FR1(N, M)
      M = N * N
      RETURN
      END

      INTEGER FUNCTION FF1(N)
      FF1 = N * N * N
      RETURN
      END
```

Вызов этих подпрограмм в C++ выполняется следующим образом:

```
#include <iostream.h>
extern "C"
{
 void __stdcall FR1(int *, int *);
 int __stdcall FF1(int *);
}
```

```
void main()
{
 int n = 10, nSq, nCube;
 FR1(&n, &nSq);
 cout << "The square is:" << nSq << endl;
 nCube = FF1(&n);
 cout << "The Cube is:" << nCube << endl;
}
```

В следующем примере символьная строка из программы на C++ передается в подпрограмму на Фортране. Текст программы на C++:

```
#include <iostream.h>
#include <string.h>
extern "C"
{
 void __stdcall FR1(int *, int *, char *);
}
void main()
{
 int n = 10, nSq;
 char szCtest[20];
 strcpy(szCtest, "teststring");
 FR1(&n, &nSq, szCtest);
 cout << "The square is:" << nSq << endl;
}
```

Текст процедуры на Фортране:

```
      SUBROUTINE FR1(N, M, CSTR)
      INTEGER * 4 CSTR(1)
      M = N * N
      WRITE(6, 20) (CSTR(L), L = 1, 3)
20    FORMAT(' CSTR = ', 3A4)
      WRITE(6, *) 'DONE'
      RETURN
      END
```

## Вызов подпрограмм C++ из программы на Фортране

Приведем пример вызова процедур и функций. Программа на Фортране:

```
      PROGRAM TESTC
      INTEGER CR2
```

```
      N = 10
      CALL CR1(N, M)
      WRITE(6, 20) N, M
20    FORMAT(' THE SQUARE OF', I3, ' IS', I4)
      K = CR2(N)
      WRITE(6, 30) N, K
30    FORMAT(' THE CUBE OF', I3, ' IS', I15)
      CALL EXIT
      END
```

Подпрограмма на C++:

```
extern "C"
{
 void __stdcall CR1(int *, int *);
 int __stdcall CR2(int *);
}
void __stdcall CR1(int *n, int *m)
{
 int k;
 k = *n;
 *m = k * k;
 return;
}
int __stdcall CR2(int *n)
{
int m, k;
 k = *n;
 m = k * k * k;
 return m;
}
```

Следующие замечания относятся к C и компилятору f77 в ОС UNIX. Если диалект C не поддерживает работу с комплексными значениями, можно определить соответствующий тип:

```
typedef struct {float r, i} complex;
complex alfa;
alfa.r = 1.0;
alfa.i = 2.0;
```

где суффиксы `r` и `i` обозначают соответственно вещественную и мнимую части числа.

Поскольку параметры в Фортране передаются по ссылке, а в С по значению, перед большинством параметров процедуры С, вызывающей процедуру на Фортране, следует ставить амперсанд `&`.

Компилятор f77 в ОС UNIX добавляет к именам общих блоков, функций и процедур знак подчеркивания. Это делается для того, чтобы отличать их от процедур С или внешних переменных с таким же именем, и связано с различием соглашений о способе передачи параметров.

Функция на Фортране, возвращающая результат типа `INTEGER`, `LOGICAL` или `DOUBLE PRECISION`, возвращает тот же результат, что и соответствующая функция С. Функции С не могут возвращать значение типа `float`, потому что оно сначала имеет тип с двойной точностью `double`. Функция комплексного типа в Фортране эквивалентна функции С с дополнительным аргументом, указывающим на адрес возвращаемого значения. Объявление в Фортране:

```
COMPLEX FUNCTION F(...)
```

эквивалентно следующему объявлению в С:

```
F_(temp, ...)
struct {float r, i;} *temp;
```

Аналогично, функция на Фортране, возвращающая символьное значение:

```
CHARACTER * 14 FUNCTION G(...)
```

эквивалентна функции С с двумя дополнительными аргументами, дающими адрес данных и их длину:

```
g_(result, length, ...)
char result[];
long length;
```

Она может вызываться из программы на языке С так:

```
char chars[15]
g_(chars, 15L, ...);
```

Символьные параметры требуют дополнительного указания длины строки. Вызов из программы на Фортране:

```
EXTERNAL FUNC
CHARACTER * 8 STR
INTEGER NUM(3)
CALL SAM(FUNC, NUM(2), STR)
```

эквивалентен вызову из программы на С:

```
int func_()
char str[8];
long num[3];
sam_(func_, &num[1], str, 8L)
```

# Глава 15

# Фортран 2003

Фортран четыре раза подвергался стандартизации. Первый стандарт был принят в 1966 году (ISO/IEC 1539:1972) и получил название *Фортран 66*. Затем стандарты принимались в 1978 (ISO/IEC 1539:1978 *Фортран 77*), 1991 (ISO/IEC 1539:1991, *Фортран 90*) и 1997 годах (ISO/IEC 1539:1997, *Фортран 95*). Изменения и дополнения, принятые в 1978 и 1997 годах, были сравнительно небольшими. Дополнения 1991 и 1997 годов оказались более серьезными. Основные усовершенствования языка, принятые в 1991 году, включали динамические массивы, структуры, производные типы, указатели, параметризацию типов, модули и дополнительные средства работы с массивами. В 1997 году язык был согласован со спецификацией Высокопроизводительного Фортрана. Затем принимались расширения языка, опубликованные в виде официальных отчетов. Первому из них присвоен код ISO/IEC TR 15581(E). Он включает расширенные возможности работы с типами. Второе дополнение (ISO/IEC TR 15580(E)) включает поддержку исключений IEEE, возникающих при выполнении операций с плавающей точкой.

Решением комитета ISO/IEC JTC1/SC22/WG5 в 1997 году начата работа над следующим стандартом языка, который учитывает пожелания программистов, принимаемые и обобщаемые национальными комитетами.

В данной главе приводится краткое описание особенностей грядущего стандарта Фортрана (*Фортран 2000* или *Фортран 2003*), основанное на его итоговом проекте. Этот стандарт является серьезным расширением Фортрана 95. Фортран 2003 отличается от Фортрана 95 гораздо больше, чем последний отличается от Фортрана 90. Основные изменения относятся к поддержке объектно-ориентированного программирования и совместного использования Фортрана и C. Второе дает программисту доступ к системным функциям, написанным на языке C. Имеются и другие изменения и дополнения. Окончательной датой выхода этого стандарта объявлен 2004 год.

# Новое в Фортране 2003

Приведем перечень основных дополнений, вводимых новым стандартом языка:

- расширения для работы с производными типами — параметризованные производные типы, улучшенное управление доступом к компонентам типа, усовершенствованные конструкторы структур и завершающие подпрограммы;
- поддержка ООП — расширение типа и наследование, полиморфизм, динамическое размещение типа и процедуры, привязанные к типу;
- расширения для работы с данными — размещаемые компоненты, отложенные параметры типа, атрибут `VOLATILE`, явные спецификации типа в конструкторах массива и операторах размещения, дополнительные средства работы с указателями, усовершенствованные выражения инициализации и расширения, относящиеся к встроенным процедурам;
- расширенные возможности ввода-вывода — асинхронная передача данных, потоковый доступ к файлам, определяемые пользователем операции передачи данных для производных типов, управление округлением во время преобразований формата, именованные константы для стандартных логических устройств ввода-вывода, оператор `FLUSH`, средства работы с диагностическими сообщениями;
- процедурные указатели;
- средства обработки исключений для операций с плавающей точкой стандарта IEEE 1989;
- расширенная поддержка смешанного программирования на Фортране и C;
- поддержка интернационального использования — возможность работы с символами 32-разрядной кодировки ISO 10646, выбора десятичной точки или запятой при форматном вводе-выводе числовых данных;
- дополнительные средства взаимодействия с операционной системой — доступ к аргументам командной строки, переменным окружения и сообщениям об ошибках процессора.

Есть и другие изменения. За небольшими исключениями, в Фортране 2003 соблюдается обратная совместимость с предыдущими стандартами. Изменения и дополнения учитывают пожелания прикладных программистов и позволяют найти разумный компромисс между современным состоянием программирования и большими инвестициями в разработку программного обеспечения, сделанными за многие годы.

## Типы

В Фортране 95 каждому встроенному типу сопоставлен параметр разновидности, а символьному типу — параметр длины. Это примеры параметризо-

ванных типов. Производные типы не допускали параметризации. В новом стандарте допускается использование произвольного числа любых параметров разновидности и длины. Параметр разновидности является константой и может использоваться в качестве разновидности встроенного или производного типа. Параметр длины моделируется с помощью параметра длины символьного типа и может использоваться для описания длины строковых компонент и границ массивов. Имена параметров объявляются в заголовке описания типа TYPE. В описании типа им сопоставляется атрибут KIND или LEN. В выражении инициализации может определяться начальное значение, например:

```
TYPE MATRIX(KIND, M, N)
 INTEGER, KIND :: KIND = KIND(0.0D0)
 INTEGER, LEN :: M, N
 REAL(KIND) :: ELEMENT(M, N)
END TYPE
```

Фактические значения параметров типа задаются в предложениях описания переменных или иных объектов этого типа, например:

```
TYPE(MATRIX(KIND(0.0D0), 15, 22)) :: AMATR
```

Можно использовать объявление типа с ключевым словом:

```
TYPE(MATRIX(KIND = KIND(0.0), M = 100, N = 20)) :: BM
```

Для того чтобы получить значение параметра типа, следует использовать конструкцию:

```
AMATR%KIND
BM%M
```

Изменять значение параметра типа во время выполнения программы нельзя.

## Процедурные указатели

Указатель может быть процедурным с явным или неявным интерфейсом, который не должен быть родовым или элементным. В примере:

```
PROCEDURE(PROC), POINTER :: P1 => NULL()
```

указатель P1 объявлен процедурным, его начальное значение пустое, а интерфейс такой же, как у процедуры PROC. Если нельзя подобрать в качестве шаблона уже имеющуюся процедуру, описывается абстрактный интерфейс:

```
ABSTRACT INTERFACE
 REAL FUNCTION F(X, Y)
 REAL, INTENT(IN) :: X, Y
 END FUNCTION
END INTERFACE
```

Абстрактному интерфейсу не соответствует никакая фактическая процедура. Процедурный указатель может быть инициализированным и неинициализированным. В примере:

```
PROCEDURE( ), POINTER :: P
```

указатель `P` не инициализирован и имеет неявный интерфейс. Оператор:

```
PROCEDURE(TYPE(MATRIX(KIND(0.0D0), M = 10, N = 20))), POINTER :: P
```

указывает, что указатель может быть связан только с функцией, имеющей указанный тип и параметры типа.

Результат функции может быть процедурным указателем. Присваивание для процедурного указателя аналогично присваиванию для указателя на данные, например:

```
P => PROC
```

Процедурные указатели заполняют пробел, имевшийся в Фортране 95, и позволяют реализовать механизм динамического связывания. Пример:

```
TYPE MATRIX(KIND, M, N)
 INTEGER, KIND :: KIND
 INTEGER, LEN :: M, N
 REAL(KIND) :: ELEMENT(M, N)
 PROCEDURE(LU), POINTER :: SOLVE
END TYPE
…
TYPE(MATRIX(KIND(0.0D0), M = 10, N = 20)) :: A
…
CALL A%SOLVE(…)
…
```

## Завершающие подпрограммы

У производного типа могут быть *завершающие* ("финальные") подпрограммы. Они предназначены для открепления указателей от адресатов, когда последние перестают существовать. Каждая завершающая подпрограмма является модульной процедурой с единственным аргументом производного типа. Этим аргументом является объект, готовый к удалению. Пример объявления завершающих подпрограмм:

```
TYPE T
 CONTAINS
 FINAL :: FINISH1, FINISH2
END TYPE T
```

Производный тип является *завершаемым*, если для него задана процедура завершения или он имеет компонент завершаемого типа. Последний

не может быть указателем или размещаемым объектом. Объект данных является завершаемым, если его тип завершаемый. Если такой объект прекращает существование и для него имеется процедура завершения с подходящим параметром разновидности и рангом, вызывается эта процедура. В противном случае вызывается элементная процедура завершения с подходящими параметрами разновидности. Затем завершается каждый завершаемый компонент. Если он является массивом, то отдельно завершается каждый завершаемый компонент каждого элемента.

## Процедуры, привязанные к типу по имени

Процедура может быть привязана к типу, а доступ к ней может быть организован с помощью выбора компонентов скалярного объекта этого типа, например:

```
TYPE T
 CONTAINS
 PROCEDURE :: PROC => MY_PROC
 PROCEDURE :: PROC2
END TYPE T
```

Здесь имя `MY_PROC` связывается с именем `PROC`, а `PROC2` — со своим собственным именем. Процедура должна быть модульной или внешней, с явным интерфейсом. Если `A` — скалярная переменная типа `T`, процедура, привязанная к типу, вызывается следующим образом:

```
CALL A%PROC(X, Y)
```

Несколько процедур могут вызываться с помощью общего родового имени. Для этого оператор `PROCEDURE` заменяется оператором `GENERIC`:

```
GENERIC :: GEN => PROC1, PROC2, PROC3
```

где `PROC1`, `PROC2` и `PROC3` являются именами специфических привязок.

## Атрибут PASS

Может оказаться так, что процедуре с доступом через компонент или с помощью привязки по имени необходимо иметь доступ к скалярному объекту, посредством которого она вызывалась. По умолчанию его значение передается первому аргументу. В случае вызова:

```
CALL A%PROC(X, Y)
```

значение `A` передается первому аргументу процедуры, `X` — второму и т. д. Для того чтобы это правило выполнялось, первый аргумент должен быть скалярным и иметь соответствующий тип. Считается, что по умолчанию процедура имеет атрибут `PASS`, который и обеспечивает такой способ передачи. Его можно отменить с помощью атрибута `NOPASS`, например:

```
PROCEDURE, NOPASS, POINTER :: P
```

Атрибут PASS можно назначить любому аргументу подпрограммы. Передаваемый объект не может быть размещаемым или указателем, а все параметры длины должны быть подразумеваемыми.

## Процедуры, привязанные к типу как операции

Процедура может быть привязана к типу как операция или как задаваемое присваивание. В этом случае доступ к ней возможен, если имеется доступ к объекту данного типа. Пример:

```
TYPE MATRIX(KIND, M, N)
 INTEGER, KIND :: KIND
 INTEGER, LEN :: M, N
 REAL(KIND) :: ELEMENT(M, N)
 CONTAINS
 GENERIC :: OPERATOR(+) => PLUS1, PLUS2, PLUS3
 GENERIC :: ASSIGNMENT(=) => ASSIGN1, ASSIGN2
END TYPE
…
TYPE(MATRIX(KIND(0.0D0), M = 10, N = 20)) :: A, B, C
…
A = B + C
…
```

Аргументы должны иметь тот тип, к которому привязана процедура.

## Расширение типа

Производный тип, не имеющий атрибутов SEQUENCE или BIND(C), является *расширяемым*. Тип, из которого произведено расширение, называется *родительским* типом. Все параметры родительского типа, его компоненты и связанные с ним процедуры наследуются расширенным типом и используются с теми же именами. Пример расширения типа MATRIX:

```
TYPE, EXTENDS(MATRIX) :: FACTORED_MATRIX
 LOGICAL :: FACTORED=.FALSE.
 REAL(MATRIX%KIND) :: FACTORS(MATRIX%M, MATRIX%N)
END TYPE
```

Объявление вещественной факторизованной матрицы может выглядеть следующим образом:

```
TYPE(FACTORED_MATRIX(KIND(0.0), 10, 10)) :: F
```

Значения параметров этого типа могут быть получены с помощью обращений F%KIND, F%M и F%N. На унаследованный компонент можно ссылаться следующим образом:

```
F%ELEMENT.
```

Расширенный тип может иметь параметры, компоненты и привязанные к нему процедуры, объявленные в его собственном описании. В нашем примере это дополнительные компоненты F%FACTORED и F%FACTORS.

Расширенный тип содержит специальный компонент — родительский компонент. Его имя, тип и параметры типа совпадают с именем, типом и параметрами родительского типа.

В конструкторах структур для расширенных типов используется следующий порядок компонентов: сначала идут унаследованные компоненты и именно в том порядке, в котором они заданы в родительском типе, затем следуют новые компоненты. В нашем примере это ELEMENT, FACTORED и FACTORS.

В конструкторе структуры значения могут задаваться для родительского компонента или для унаследованных компонентов. Компоненты нельзя задавать несколько раз, а любой компонент, не имеющий значения по умолчанию, задается обязательно.

## Перечисления

*Перечисление* является множеством целых констант-перечислителей, которое используется для взаимодействия с программами на языке C. Пример использования перечисления с константами 4, 9 и 10:

```
ENUM, BIND(C)
 ENUMERATOR :: RED = 4, BLUE = 9
 ENUMERATOR YELLOW
END ENUM
```

Если значение перечислителя явным образом не указано, оно берется на единицу большим, чем значение предыдущего перечислителя, или равным нулю, если перечислитель — первый по порядку.

## Блок *ASSOCIATE*

Блок ASSOCIATE связывает именованные объекты программы с выражениями или переменными на время выполнения этого блока, например:

```
ASSOCIATE(Z => EXP(-(X**2 + Y**2)) * COS(THETA))
 PRINT *, A + Z, A - Z
END ASSOCIATE

ASSOCIATE(XC => AX%B(I, J)%C, ARRAY => AX%B(I, :)%C)
 XC%DV = XC%DV + PRODUCT(XC%EV(1:N))
 ARRAY = ARRAY + 1.0
END ASSOCIATE
```

Каждое имя в этом случае называют *ассоциированным именем*. Ассоциированный объект нельзя использовать в ситуации, когда его значение может

измениться. Исключение составляют лишь переменные. Блоки ASSOCIATE могут быть вложенными.

## Полиморфные объекты

При объявлении полиморфного объекта вместо ключевого слова TYPE используется ключевое слово CLASS. Тип объекта в любой определенный момент выполнения программы называют его *динамическим* типом. Объект должен иметь атрибут указателя или размещаемого объекта и может быть формальным аргументом. Он получает динамический тип при размещении, присваивании указателя или связывании с фактическим аргументом.

Объект, объявленный с помощью спецификатора CLASS(*), называют *неограниченно полиморфным*. Он не обязан иметь тип какого-либо другого объекта и считается совместимым по типу со всеми объектами.

Формальный аргумент, являющийся указателем или размещаемым объектом, должен иметь тот же тип, что и соответствующий ему фактический аргумент, и если один из них полиморфный, то и другой должен быть полиморфным. Формальный аргумент с атрибутом PASS должен быть полиморфным, если его тип расширяемый.

Встроенное присваивание для производного типа должно быть расширено для того, чтобы его правая часть могла быть полиморфной. Правая часть оператора присваивания может иметь динамический тип, являющийся расширением типа левой части. В этом случае компоненты левой части копируются из соответствующих компонентов правой части.

Справочные функции SAME_TYPE_AS(A, B) и EXTENDS_TYPE_OF(A, MOLD) позволяют определить, имеют ли A и B одинаковый динамический тип и является ли динамический тип A расширением динамического типа MOLD.

## Блок *SELECT TYPE*

В зависимости от динамического типа выражения-*селектора*, блок SELECT TYPE позволяет выбрать для выполнения один из составляющих его блоков, например:

```
CLASS(MATRIX(KIND(0.0), 10, 10)) :: F
...
SELECT TYPE(FF => F)
 TYPE IS(MATRIX)
 TYPE IS(FACTORED_MATRIX)
END SELECT
```

В данном примере первый блок выполняется в том случае, когда переменная F имеет динамический тип MATRIX, а второй, если ее динамический тип — FACTORED_MATRIX. Связь селектора с его ассоциированным именем реализуется так же, как в блоке ASSOCIATE.

Рассмотрим еще один пример:

```
TYPE :: POINT
 REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: POINT_3D
 REAL :: Z
END TYPE POINT_3D

TYPE, EXTENDS(POINT) :: COLOR_POINT
 INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT), TARGET :: P
TYPE(POINT_3D), TARGET :: P3
TYPE(COLOR_POINT), TARGET :: C
CLASS(POINT), POINTER :: P_OR_C
P_OR_C => C

SELECT TYPE(A => P_OR_C)
 TYPE IS(POINT_3D)
 PRINT *, A%X, A%Y, A%Z
 CLASS IS(POINT)
 PRINT *, A%X, A%Y ! THIS BLOCK GETS EXECUTED
END SELECT
```

Здесь в каждом блоке ассоциированному имени ставится в соответствие или тип, заданный в операторах `TYPE IS`, или класс, заданный в операторах `CLASS IS`. Блок выбирается исходя из следующих правил:

- если тип селектора соответствует блоку `TYPE IS`, то выбирается этот блок;
- если класс селектора соответствует блоку `CLASS IS`, выбирается этот блок;
- если класс селектора соответствует нескольким блокам `CLASS IS`, выбирается блок для того типа, который является расширением остальных.

Необязательный блок `CLASS DEFAULT` выбирается в том случае, когда не выбран ни один другой блок. Ассоциированное имя в этом случае имеет тот же тип, что и селектор.

### Отложенное связывание и абстрактные типы

В некоторых ситуациях процедура, привязанная к типу, может вызываться только для расширений типа, переопределяющих ее. Это обычно происходит в случае, когда привязанная к типу процедура не имеет естественной реализации или реализации по умолчанию, а имеет только хорошо определенные назначение и интерфейс. В этом случае объявляется абстрактный тип с отложенной привязкой. Для привязанной к типу процедуры требуется абстрактный интерфейс. В следующем примере расширения типа имеют компоненты, содержащие информацию о файле, а оператор `OPEN` следует заменить процедурой, использующей эти данные для того, чтобы открыть данный файл:

```
TYPE, ABSTRACT :: FILE_HANDLE
 CONTAINS
       PROCEDURE(OPEN_FILE), DEFERRED, PASS :: OPEN
 …
END TYPE

ABSTRACT INTERFACE
 SUBROUTINE OPEN_FILE(HANDLE)
       CLASS(FILE_HANDLE), INTENT(INOUT) :: HANDLE
 END SUBROUTINE OPEN_FILE
END INTERFACE
```

Абстрактный тип могут иметь только полиморфные объекты. Отложенное связывание разрешено только для абстрактных типов.

## Разное

### Конструкторы структур

Список значений в конструкторе структуры может содержать ключевые слова, являющиеся именами компонент. Компоненты, значения которым даны по умолчанию (в определении типа), могут быть опущены. Конструкторы структур можно использовать для типов с приватными (`PRIVATE`) компонентами, имеющими значения по умолчанию. Явные значения не могут задаваться многократно, они переопределяют значения по умолчанию. Если тип параметризован, необходимо определить параметры, не имеющие значений по умолчанию, например:

```
A = MATRIX(KIND(0.0), M = 10, N = 20) (ELEMENT = 0.0)
```

Родовое имя может совпадать с именем производного типа, если оно является именем функции. Это дает эффект перегрузки конструктора данного типа.

## Оператор *ALLOCATE*

Атрибут ALLOCATABLE может использоваться не только в определении массивов. Фортран 2003 допускает объявление *исходной* переменной, с помощью которой можно присвоить значения отложенным параметрам типа и начальное значение самому объекту. Операторы:

```
TYPE(MATRIX(KIND(0.0D0), M = 10, N = 20)) :: A
TYPE(MATRIX(KIND(0.0D0), M = :, N = :)), ALLOCATABLE :: B, C
…
ALLOCATE(B, SOURCE = A)
ALLOCATE(C, SOURCE = A)
```

размещают переменные B и C, которые можно считать матрицами размером 10 на 20 со значением A. Если имеется ключевое слово SOURCE, один оператор ALLOCATE размещает только один объект. Значение SOURCE должно иметь тот же тип, что и размещаемый объект. Он может быть расширением типа размещаемого объекта, а его значение должно быть скаляром или массивом того же ранга, что и размещаемый объект.

Отложенные параметры типа могут быть заданы в объявлении типа, расположенном в операторе размещения:

```
ALLOCATE(MATRIX(KIND(0.0D0), M = 10, N = 20) :: B, C)
```

В этом случае инициализация с помощью исходной переменной невозможна.

В операторе размещения может объявляться динамический тип полиморфного объекта, например:

```
CLASS(MATRIX(KIND(0.0), 10, 10)) :: A, B, C, D
…
ALLOCATE(FACTORED_MATRIX(KIND(0.0), 10, 10) :: B, C)
ALLOCATE(D, SOURCE = A)
```

В операторах ALLOCATE и DEALLOCATE можно использовать необязательный спецификатор ERRMSG =, задающий символьную скалярную переменную. Этой переменной присваивается строковое значение с текстом диагностического сообщения, которое выводится в случае ошибки. Если исключительная ситуация не возникает, значение переменной не изменяется.

## Динамические массивы и оператор присваивания

Оператор присваивания для размещаемой переменной интерпретируется так же, как и для компонента размещаемого массива. Целевая переменная размещается с правильной формой, если она еще не была размещена или размещается заново с правильной формой, если она была уже размещена, но с другой формой. В следующем примере операторы:

```
S = SIZE(COMPRESS(B))
IF(ALLOCATED(A)) THEN
```

```
 IF(SIZE(A) /= S) DEALLOCATE(A)
END IF
IF(.NOT. ALLOCATED(A)) ALLOCATE(A(S))
A = COMPRESS(B)
```

и

```
A = COMPRESS(B)
```

дают одинаковый результат.

## Передача размещения

Встроенная подпрограмма `MOVE_ALLOC(FROM, TO)` позволяет передавать состояние размещения от одного объекта к другому. Оба аргумента должны быть размещаемыми и совместимыми по типу, а их ранг должен совпадать. Если один объект полиморфный, то и другой тоже должен быть полиморфным. Не отложенный параметр типа объекта `TO` должен иметь такое же значение, что и соответствующий параметр типа объекта `FROM`. После вызова подпрограммы объект `FROM` удаляется из памяти. Это дает возможность размещать объекты заново, управляя передачей старых данных новым объектам, например:

```
REAL, ALLOCATABLE :: GRID(:), TEMPGRID(:)
…
ALLOCATE(GRID(-N:N))
…
ALLOCATE(TEMPGRID(-2 * N:2 * N))
TEMPGRID(::2) = GRID
CALL MOVE_ALLOC(TO = GRID, FROM = TEMPGRID)
```

## Управление доступом

Атрибуты `PUBLIC` и `PRIVATE` можно назначать индивидуальным компонентам производного типа:

```
TYPE, EXTENDS(PERSON) :: S_PERSON
 CHARACTER(:), ALLOCATABLE, PUBLIC :: NAME
 INTEGER, PRIVATE :: AGE
END TYPE
```

Элемент может быть `PUBLIC`, даже если его тип имеет атрибут `PRIVATE`. Доступность компонентов типа не зависит от доступности самого типа.

Связывание с типом может быть объявлено `PRIVATE` или `PUBLIC`, например:

```
PROCEDURE, PUBLIC, PASS, POINTER :: P
GENERIC, PUBLIC :: GEN => PROC1, PROC2
GENERIC, PRIVATE :: OPERATOR(+) => PLUS1, PLUS2, PLUS3
```

Завершающая подпрограмма не может быть объявлена с атрибутами PRIVATE или PUBLIC. Она всегда доступна для удаления любой переменной данного типа.

Переменная или указатель, объявленные в модуле, могут иметь атрибут PROTECTED. В этом случае значение переменной или состояние указателя можно менять только в самом модуле. PROTECTED может быть атрибутом или оператором,— допустимы обе формы:

```
REAL, PROTECTED :: A(10)
```

или:

```
PROTECTED :: A, B
```

Данный атрибут распространяется на все подобъекты объекта. Если защищенным является указатель, то защищен статус ассоциации указателя, но не значение его адресата.

Возможность управления доступом позволяет увеличить надежность программного обеспечения. Она аналогична использованию атрибута INTENT(IN) для формального аргумента подпрограммы, запрещающего его изменение. Атрибут PROTECTED можно применять только в модуле исходного описания. Если модуль использует незащищенную переменную из другого модуля, она не может быть наделена атрибутом PROTECTED.

## Переименование операций в операторе *USE*

В Фортране 2003 появилась возможность переименовывать операции, которые не являются встроенными:

```
USE MY_MODULE, OPERATOR(.MYADD.) => OPERATOR(.ADD.)
```

Здесь новое имя .MYADD. дано операции .ADD.

## Указатель в операторах присваивания

В операторе присваивания с указателем в левой части можно задавать нижние границы для индексов. Они могут быть любым скалярным целочисленным выражением:

```
P(0:, 0:) => A
```

Разрешено изменение формы массива с переходом к массиву единичного ранга:

```
P(1:M, 1:2 * M) => A(1:2 * M * M)
```

Отображение элементов одного массива на элементы другого массива выполняется в порядке следования элементов первого массива, а целевой массив должен быть достаточно большим.

Параметры длины типа указателя могут быть отложенными (при описании с двоеточием). Присваивание указателя дает им значения соответствующих

параметров адресата. Все прочие параметры типа указателя должны иметь то же значение, что и соответствующие параметры типа адресата.

## Атрибут *INTENT* для указателя

В Фортране 95 атрибут `INTENT` не применяется с формальными параметрами, потому что в этом случае появляется неопределенность, должен ли он относиться к статусу ассоциирования указателя, к значению адресата или и к тому, и к другому. В Фортране 2003 атрибут `INTENT` можно использовать с формальными аргументами. Он относится к статусу ассоциирования и не влияет на значение адресата.

## Атрибут *VOLATILE*

В Фортране 2003 предусмотрена возможность изменения значений элементов данных программы (переменных, указателей и т. д.) извне, например, другой программой, выполняющейся одновременно с данной. Для этого компилятор должен располагать информацией о ячейках памяти, содержимое которых может модифицироваться непредсказуемым с точки зрения программы образом.

Атрибут `VOLATILE` назначается элементам данных, значения которых могут изменяться способом, не описанным в программе. В случае указателя атрибут `VOLATILE` относится только к статусу указателя. В случае размещаемого объекта атрибут `VOLATILE` относится ко всем характеристикам данного объекта, а также ко всем его подобъектам.

## Оператор *IMPORT*

В Фортране 95 из тела интерфейсного блока нет доступа к объектам модуля, в котором оно содержится. Тип, объявленный в модуле, также нельзя использовать в интерфейсном блоке, даже если описание интерфейса находится в том же модуле.

В Фортране 2003 оператор `IMPORT` позволяет объявить импортируемые из модуля объекты. Он имеет вид:

```
IMPORT :: NEW_TYPE, COMPUTE
```

Если список импорта отсутствует, доступными оказываются все объекты носителя. Если же список имеется, доступны только перечисленные в нем элементы.

## Определение параметров среды выполнения программы

Встроенный модуль `ISO_FORTRAN_ENV` содержит следующие константы:

- `INPUT_UNIT`, `OUTPUT_UNIT` и `ERROR_UNIT` — скалярные константы стандартного целого типа, содержащие логический номер устройства, используемый по умолчанию в операторах `READ` и `WRITE`;

- IOSTAT_END и IOSTAT_EOR — скалярные константы стандартного целого типа, содержащие значения, которые присваиваются переменной IOSTAT при достижении конца файла (End-Of-File) или конца записи (End-Of-Record);
- NUMERIC_STORAGE_SIZE, CHARACTER_STORAGE_SIZE и FILE_STORAGE_SIZE — скалярные константы стандартного целого типа, содержащие размер (в битах) устройств хранения числовых и символьных данных, а также файлов.

Добавлены встроенные процедуры:

- COMMAND_ARGUMENT_COUNT — возвращает количество аргументов командной строки;
- GET_COMMAND — возвращает команду вызова программы;
- GET_COMMAND_ARGUMENT — возвращает аргумент команды запуска программы;
- GET_ENVIRONMENT_VARIABLE — позволяет получить значение переменной окружения.

## Поддержка наборов интернациональных символов

В Фортране 2003 поддерживается стандарт ISO 10646, который определяет 4-байтовый код хранения символов и допускает использование практически всех национальных алфавитов.

Встроенная функция SELECTED_CHAR_KIND(NAME) возвращает значение параметра разновидности для указанного набора символов. Здесь NAME является скалярной величиной стандартного символьного типа, а ее значения DEFAULT, ASCII и ISO_10646 задают соответствующий набор символов.

Алфавит Фортрана 2003 дополнен новыми символами (см. табл. 15.1), но фактически в конструкциях языка используются только квадратные скобки.

***Таблица 15.1.*** *Новые символы алфавита Фортрана*

| Символ | Описание | Символ | Описание |
|---|---|---|---|
| ~ | Волна | \ | Обратная наклонная черта |
| [ | Левая квадратная скобка | ' | Диакритический знак |
| ] | Правая квадратная скобка | ? | Циркумфлекс |
| { | Левая фигурная скобка | } | Правая фигурная скобка |
| \| | Вертикальная черта | # | Знак номера |
| @ | Коммерческое "at" | | |

Стандартным способом представления 4-байтовых символов и строк 1-байтовых символов в файле является код UTF-8 (Unicode). Его поддержка обеспечивается следующим:

- в операторе `OPEN` для форматного файла можно использовать спецификатор `ENCODING = 'UTF-8'`;
- в операторе `INQUIRE` можно использовать запрос `ENCODING =`;
- разрешены ввод символьных значений из файла в кодировке UTF-8 и их присвоение символьным переменным стандарта ISO 10646;
- разрешен вывод числовых и логических значений в файл в кодировке UTF-8, а также их ввод из файла;
- разрешен вывод любых ASCII или символьных данных ISO 10646 в файл UTF-8, их ввод и присвоение символьным переменным той же разновидности;
- поддерживаются операции с внутренними файлами ISO 10646;
- разрешено присвоение стандартных символьных значений переменным ISO 10646;
- в функцию `ACHAR` введен добавочный аргумент `KIND`, а в функции `IACHAR` разрешается использовать символьный параметр любой разновидности.

В Фортране 2003 разрешены имена длиной до 63 символов и операторы длиной до 256 строк.

## Константы

В качестве аргумента встроенных функций `INT`, `REAL`, `CMPLX` и `DBLE` (но не в качестве разновидности типа) можно использовать двоичные, восьмеричные и шестнадцатеричные константы, например:

```
INT(O'345')
REAL(Z'1234ABCD')
```

В функции `INT` константа `'BOZ'` интерпретируется так, как если бы она была целочисленной константой разновидности с максимальным интервалом допустимых значений, поддерживаемым процессором. В других функциях эта константа интерпретируется так, как если бы она имела то же значение, что и переменная таких же типа и разновидности, что и заданный двоичный код. Интерпретация значения двоичного кода зависит от процессора. Если разновидность не указана, используется ее значение по умолчанию.

## Конструктор массива

В качестве ограничителей конструкторов массива вместо парных символов `(/` и `/)` можно использовать квадратные скобки. Конструктор может содержать объявление типа:

```
[CHARACTER(LEN = 7) :: 'PHYSICS', 'BIOLOGY', 'HISTORY']
[MATRIX(KIND(0.0), M = 10, N = 20) :: A, B, C],
```

которое позволяет задавать значения параметров длины. В конструкторах производных типов должно соблюдаться соответствие параметров разновидности и параметров длины.

## Выражения спецификации и инициализации

Ослаблены ограничения на использование выражений спецификации и инициализации. В частности, в них разрешено использовать любые стандартные встроенные процедуры.

## Комплексные константы

Для задания вещественной и мнимой частей комплексной константы можно использовать именованные константы:

```
REAL, PARAMETER :: PI = 3.1415926
COMPLEX :: C = (1.0, PI)
```

## Встроенные функции

Аргумент `COUNT_RATE` процедуры `SYSTEM_CLOCK` может иметь тип `REAL` или `INTEGER`. Это позволяет обеспечить более точное определение скорости работы современных процессоров.

Аргументы встроенных функций `MAX`, `MAXLOC`, `MAXVAL`, `MIN`, `MINLOC` и `MINVAL` могут иметь тип `CHARACTER`, `REAL` и `INTEGER`.

Усовершенствованы функции `ATAN2`, `LOG` и `SQRT`. Если аргумент равен положительному или отрицательному нулю, они возвращают предел соответствующей функции для значений аргумента, стремящихся к нулю с соответствующим знаком. Если аргумент `X` принимает отрицательное значение, функции `ATAN2(-0.0, X)` и `LOG(CMPLX(X, -0.0))` возвращают значение $-\pi$, а результат вычисления функции `SQRT(CMPLX(X, -0.0))` имеет отрицательную мнимую часть.

## Управление потерей значимости

Согласно стандарту IEEE потеря значимости должна быть "плавной", то есть исчезающе малые значения представляются с помощью минимального отрицательного порядка и минимальной точности, если это возможно. Некоторые процессоры заменяют такие значения нулями ("резкая" потеря точности). В таком режиме может ускоряться выполнение программы. В Фортране 2003 добавлена возможность определения режима потери значимости и его изменения на тех системах, которые это допускают.

В модуль `IEEE_ARITHMETIC` добавлены процедуры:

- `IEEE_SUPPORT_UNDERFLOW_CONTROL(X)` — возвращает значение `TRUE`, если процессор поддерживает режим управления потерей значимости в вы-

числениях с плавающей точкой и типом аргумента X. Аргумент X можно опустить, в этом случае проверка выполняется для всех вычислений с плавающей точкой;

- IEEE_GET_UNDERFLOW_MODE(GRADUAL) — присваивает аргументу GRADUAL стандартного логического типа значение TRUE, если включен режим плавной потери значимости;
- IEEE_SET_UNDERFLOW_MODE(GRADUAL) — процедура, которая переключает режимы потери значимости (если GRADUAL принимает значение TRUE, то включается режим плавной потери значимости). Эту процедуру можно использовать только в том случае, когда функция IEEE_SUPPORT_UNDERFLOW_CONTROL(X) для некоторых значений X возвращает значение TRUE.

# Ввод-вывод

Рассмотрим особенности ввода-вывода нового стандарта.

## Ввод-вывод для производных типов

В Фортране 2003 можно сделать так, что при наличии в списке ввода-вывода объекта производного типа будет вызываться подпрограмма, считывающая данные из файла или выводящая значение производного типа в файл. Такие подпрограммы могут быть привязаны к типу как родовые подпрограммы:

```
GENERIC :: READ(FORMATTED) => R1, R2
GENERIC :: READ(UNFORMATTED) => R3, R4, R5
GENERIC :: WRITE(FORMATTED) => W1
GENERIC :: WRITE(UNFORMATTED) => W2, W3,
```

что делает их доступными независимо от доступности объекта данного типа. Альтернативой этой конструкции является интерфейсный блок вида:

```
INTERFACE READ(FORMATTED)
 MODULE PROCEDURE R1, R2
END INTERFACE
```

При форматном и неформатном вводе-выводе используются разные процедуры:

```
SUBROUTINE FORMATTED_IO(DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG)
SUBROUTINE UNFORMATTED_IO(DTV, UNIT, IOSTAT, IOMSG)
```

Аргументы:

- DTV — скаляр производного типа. Он должен быть полиморфным. Любые параметры длины должны быть подразумеваемыми. При выводе они должны иметь атрибут INTENT(IN) и содержать выводимое значение. При вводе они наделяются атрибутом INTENT(INOUT) и изменяются в соответствии со считываемыми значениями;

- UNIT — скалярный аргумент стандартного целого типа с атрибутом INTENT(IN), задающий устройство ввода-вывода или внутренний файл;
- IOTYPE — скалярный аргумент типа CHARACTER(*) с атрибутом INTENT(IN). Его значением может быть 'LISTDIRECTED', 'NAMELIST' или 'DT'//STRING, где STRING является символьной строкой из дескриптора преобразования DT;
- V_LIST — массив ранга один стандартного целого типа с подразумеваемой формой и атрибутом INTENT(IN). Его значение выбирается из списка в скобках в дескрипторе преобразований;
- IOSTAT — скалярный аргумент стандартного целого типа с атрибутом INTENT(OUT). Если возникает ошибка, ему присваивается положительное значение. Если возникает условие End-Of-File или End-Of-Record, ему присваивается значение IOSTAT_END или IOSTAT_EOR соответственно. В любом другом случае значение этого аргумента равно нулю;
- IOMSG — скалярный аргумент типа CHARACTER(*) с атрибутом INTENT(INOUT). Если значение IOSTAT не равно нулю, IOMSG необходимо присвоить строковое значение с текстом сообщения об ошибке.

## Асинхронный ввод-вывод

Ввод-вывод может быть асинхронным, то есть во время его выполнения могут выполняться другие операторы. Такой режим допускается только для внешних файлов, открытых со спецификатором ASYNCHRONOUS = 'YES' в операторе OPEN или в операторах READ и WRITE. Выполнение оператора асинхронного ввода-вывода инициирует операцию "отложенного" ввода-вывода, которая завершается операцией ожидания для файла. Она выполняется явно оператором ожидания WAIT или неявно с помощью операторов INQUIRE, CLOSE или операторов позиционирования файлового указателя.

Остальные операторы асинхронного ввода-вывода могут выполняться и до оператора ожидания. Порядок выполнения операторов ввода-вывода в этом случае такой же, как и в случае синхронного ввода-вывода.

Выполнение оператора асинхронного ввода-вывода можно идентифицировать по скалярной целой переменной в спецификаторе ID =. При успешном выполнении оператора этой переменной присваивается значение, которое зависит от процессора и которое можно передавать следующим операторам WAIT или INQUIRE.

В операторе ожидания можно использовать спецификаторы END, EOR, ERR и IOSTAT. Они имеют такой же смысл, как и в операторах ввода-вывода, и относятся к ситуациям отложенного ввода-вывода. При наличии спецификатора ID завершается только заданная отложенная операция, другие спецификаторы относятся только к ней. В противном случае все отложенные операции для данного файла завершаются по очереди.

В операторе INQUIRE можно использовать спецификатор PENDING =. При наличии спецификатора ID он задает переменную, которой присваивается значение TRUE, если указанная операция ввода-вывода все еще находится в отложенном состоянии. Если спецификатора ID нет, переменной присваивается значение TRUE при наличии любых отложенных операций ввода-вывода для данного устройства. Если переменной присваивается значение FALSE, выполняются операции ожидания для одного или всех файлов. Для значения TRUE операции ожидания не выполняются.

Оператор позиционирования файлового указателя (BACKSPACE, ENDFILE, REWIND) выполняет операции ожидания для всех отложенных операций ввода-вывода, относящихся к данному файлу. Асинхронный ввод-вывод не допускается с пользовательскими производными типами.

Переменную называют *аффектором* операции отложенного ввода-вывода, если любая часть ее ассоциирована с любой частью элемента списка ввода-вывода, конструкции NAMELIST или спецификатора SIZE. Переопределять аффектор нельзя, пока операция ввода-вывода отложена. Он не может стать неопределенным или поменять статус прикрепления (для указателя). На аффектор нельзя ссылаться или ассоциировать его с формальным аргументом, имеющим атрибут VALUE.

Если переменная используется в исполняемом операторе или выражении спецификации и в то время, когда переменная является аффектором, выполняется какой-либо другой оператор, она должна быть наделена атрибутом ASYNCHRONOUS. Переменная автоматически получает этот атрибут, если она или ее подобъект являются элементом списка ввода-вывода, конструкции NAMELIST или спецификатора SIZE = в операторе асинхронного ввода-вывода. Переменную можно объявить с этим атрибутом явно:

```
INTEGER, ASYNCHRONOUS :: INT_ARRAY(10)
```

Можно использовать оператор ASYNCHRONOUS:

```
ASYNCHRONOUS :: INT_ARRAY, ANOTHER
```

Действие атрибута ASYNCHRONOUS распространяется на все подобъекты объекта.

## Оператор *FLUSH*

Оператор FLUSH для внешнего файла делает записанные в него данные доступными для других процессов. Данные, записанные в файл не из программы на Фортране, оказываются доступными для расположенных в ней операторов ввода READ. Синтаксис оператора FLUSH аналогичен синтаксису операторов управления файловым указателем. В сочетании со спецификатором ADVANCE = 'NO' или в режиме потокового доступа он позволяет программе получить доступ к посимвольному вводу с клавиатуры.

## Спецификатор *IOMSG*

Спецификатор IOMSG = в операторе ввода-вывода задает скалярную переменную стандартного символьного типа, в которую записывается сообщение об ошибке или условиях End-Of-File и End-Of-Record, возникающих при выполнении данного оператора.

## Потоковый ввод-вывод

Режим потокового доступа к внешнему файлу включается спецификатором ACCESS = 'STREAM' в операторе OPEN и может быть как форматным, так и неформатным. Положение файлового указателя задается в единицах хранения, обычно в байтах, начиная с 1. Текущее положение можно определить по значению скалярной целой переменной в спецификаторе POS = оператора INQUIRE для данного устройства. Требуемое положение можно указывать в операторах READ или WRITE с помощью того же спецификатора. При форматном вводе-выводе он должен быть равен 1 или значению, которое вернул оператор INQUIRE для данного файла. В отсутствие спецификатора POS положение указателя не изменяется.

## Спецификатор *ROUND*

Округлением во время форматного ввода-вывода можно управлять в операторе OPEN с помощью спецификатора ROUND =. Он принимает значения UP, DOWN, ZERO, NEAREST, COMPATIBLE или PROCESSOR_DEFINED, задающие способ округления. Более высокий приоритет имеют спецификаторы ROUND в операторах READ и WRITE. Значения NEAREST и COMPATIBLE относятся к ближайшему представимому значению. Если оба ближайших значения находятся на одинаковом расстоянии от того, которое подлежит округлению, выбирается значение, зависящее от процессора в случае NEAREST и ближайшее значение по направлению к нулю в случае COMPATIBLE. Режимом округления можно также управлять с помощью дескрипторов преобразования RU, RD, RZ, RN, RC и RP.

## Спецификатор *DECIMAL*

Символ, разделяющий целую и дробную части вещественного числа при форматном вводе-выводе, может быть задан с помощью спецификатора DECIMAL = в операторе OPEN. Он принимает значения COMMA или POINT. В первом случае вводимые или выводимые значения разделяются точкой с запятой, во-втором случае — точкой. Более высокий приоритет имеют спецификаторы DECIMAL в операторах READ и WRITE. Данным режимом можно управлять и с помощью спецификаторов преобразования DC и DP.

## Спецификатор *SIGN*

Спецификатор SIGN = в операторе OPEN управляет выводом знака "плюс" при форматированном выводе числовых значений. Он может принимать

значения SUPPRESS, PLUS или PROCESSOR_DEFINED. Более высокий приоритет имеют спецификаторы SIGN = в операторах WRITE. Данным режимом можно управлять и с помощью дескрипторов преобразования SS, SP и S.

### Рекурсивный ввод-вывод

*Рекурсивным* оператором ввода-вывода является оператор, который выполняется во время выполнения другого оператора ввода-вывода. Он применяется с производными типами данных.

### Встроенная функция для символа новой строки

Встроенная функция NEW_LINE(A) возвращает символ новой строки. Параметр A должен быть символьного типа, он определяет разновидность результата.

### Ввод-вывод исключений IEEE

В Фортране 2003 стандартизован ввод-вывод "бесконечностей" IEEE и значений NaN. Все дескрипторы преобразований для вещественных значений интерпретируют их одинаково, во внимание принимается только ширина поля w.

Выводиться могут следующие значения:

- -Inf или -Infinity — отрицательная бесконечность;
- Inf, +Inf, Infinity или +Infinity — положительная бесконечность;
- NaN — "не число", после которого может идти строковое значение в скобках и без пробелов, содержащее дополнительную информацию.

При вводе регистр букв не различается.

### Запятая после дескриптора преобразования *P*

Запятая после дескриптора преобразования P не обязательна, если после нее следует спецификатор повторения. Разрешена, например, конструкция:

```
1P2E12.4
```

## Фортран и C

В Фортране 2003 стандартизован механизм взаимодействия с C. Любой объект, используемый в смешанной программе, должен быть таким, чтобы в обоих языках для него имелись эквивалентные объявления. В Фортране все подобные объекты должны быть интероперабельными.

### Интероперабельность встроенных типов

Встроенный модуль ISO_C_BINDING содержит именованные константы со значениями параметра разновидности встроенных типов. Они приведены

в табл. 15.2. Процессор может некоторые из них не поддерживать, в этом случае значение константы отрицательно.

***Таблица 15.2.*** *Интероперабельность между типами Фортрана и С*

| Тип Фортрана | Именованная константа | Тип С |
|---|---|---|
| INTEGER | C_INT | int |
| | C_SHOR | short int |
| | C_LONG | long int |
| | C_LONG_LONG | long long int |
| | C_SIGNED_CHAR | signed char |
| | | unsigned char |
| | C_SIZE_T | size_t |
| | C_INT8_T | int8_t |
| | C_INT16_T | int16_t |
| | C_INT32_T | int32_t |
| | C_INT64_T | int64_t |
| | C_INT_LEAST8_T | int_least8_t |
| | C_INT_LEAST16_T | int_least16_t |
| | C_INT_LEAST32_T | int_least32_t |
| | C_INT_LEAST64_T | int_least64_t |
| | C_INT_FAST8_T | int_fast8_t |
| | C_INT_FAST16_T | int_fast16_t |
| | C_INT_FAST32_T | int_fast32_t |
| | C_INT_FAST64_T | int_fast64_t |
| | C_INTMAX_T | intmax_t |
| | C_INTPTR_T | intptr_t |
| REAL | C_FLOAT | float |
| | C_DOUBLE | double |
| | C_LONG_DOUBLE | long double |
| COMPLEX | C_FLOAT_COMPLEX | float _complex |
| | C_DOUBLE_COMPLEX | double _complex |
| | C_LONG_DOUBLE_COMPLEX | long double _complex |
| LOGICAL | C_BOOL | _bool |
| CHARACTER | C_CHAR | char |

Интероперабельность символьного типа требует, чтобы параметр длины типа был опущен или задан в выражении инициализации, значение которого 1. Имеются следующие именованные константы: C_NULL_CHAR, C_ALERT, C_BACKSPACE, C_FORM_FEED, C_NEW_LINE, C_CARRIAGE_RETURN, C_HORIZONTAL_TAB, C_VERTICAL_TAB.

## Интероперабельность с указателями С

В модуль ISO_C_BINDING включены производные типы C_PTR и C_FUNPTR, интероперабельные с указателями С на объекты и функции соответственно. Имеются именованные константы C_NULL_PTR и C_NULL_FUNPTR для соответствующих пустых указателей С.

Модуль ISO_C_BINDING содержит следующие процедуры:

- C_LOC — возвращает адрес своего аргумента в С;
- C_FUNLOC — возвращает адрес процедуры в С;
- C_ASSOCIATED — определяет, прикреплен ли указатель;
- C_F_POINTER — определяет адрес в С интероперабельных переменных и массивов;
- C_F_PROCPOINTER — определяет адрес в С интероперабельной процедуры Фортрана.

В С нельзя передавать адресат указателя Фортрана или массив с подразумеваемой формой. Размещенный динамический массив в С передавать можно. Массив, размещенный в С, может быть ассоциирован с указателем Фортрана.

## Интероперабельность производных типов

Для того чтобы производный тип был интероперабельным, в его описании необходимо явно указать атрибут BIND:

```
TYPE, BIND(C) :: MYTYPE
...
END TYPE MYTYPE
```

Каждый компонент производного типа должен иметь интероперабельный тип и параметры типа. Он не может быть указателем или размещаемым компонентом. Например, структура:

```
typedef struct {
int m, n;
float r;
} myctype
```

интероперабельна с типом:

```
USE ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
```

```
 INTEGER(C_INT) :: I, J
 REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

Для интероперабельности имя типа и имена компонентов не важны.

В Фортране нет типа, интероперабельного с объединениями C, структурами, содержащими битовые поля, и структурами, которые содержат массивы с переменными границами.

## Интероперабельность переменных

Скалярная переменная в Фортране интероперабельна, если интероперабелен ее тип и параметры типа, она не является указателем или размещаемой переменной. Массив интероперабелен, если он имеет интероперабельный тип и параметры типа и является массивом с явной формой или подразумеваемым размером. Он соответствует массиву C с таким же типом, параметрами типа и формой, но с обратным порядком индексов. Массив, объявленный в программе на Фортране:

```
INTEGER :: A(18, 3:7, *)
```

интероперабелен с массивом C, объявленным следующим образом:

```
int b[][5][18]
```

## Интероперабельность подпрограмм

Для поддержки интероперабельности скалярных формальных аргументов подпрограмм введен атрибут `VALUE`. При вызове подпрограммы создается копия фактического аргумента. Его значение может изменяться во время выполнения процедуры, но при возвращении результата обратное копирование не выполняется. Если тип является символьным, его длина должна быть равна 1. Процедура на Фортране интероперабельна, если она имеет явный интерфейс и объявлена с атрибутом `BIND`:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C)
```

Все формальные аргументы должны быть интероперабельными. Результат функции должен быть скалярным и интероперабельным.

Подпрограмма имеет *метку связывания* с глобальной областью действия. Она представляет собой имя, под которым подпрограмма известна процессору C. По умолчанию это имя подпрограммы в Фортране в нижнем регистре. Вышеприведенная функция, например, имеет метку связывания `func`. Может быть объявлена и альтернативная метка связывания:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C, NAME = 'C_Func')
```

В этом случае функция соответствует прототипу функции C с такой же меткой связывания. Результат функции должен быть интероперабельным с результатом прототипа. Для процедуры результат прототипа должен иметь тип

void. Формальный аргумент с атрибутом VALUE и типом, отличным от C_PTR, должен соответствовать формальному параметру прототипа, который не является указателем. Формальный аргумент без атрибута VALUE или с ним, но имеющим тип C_PTR, должен соответствовать формальному параметру прототипа — указателю.

## Интероперабельность глобальных данных

Атрибут BIND может быть назначен интероперабельной переменной модуля или COMMON-блоку с интероперабельными элементами:

```
USE ISO_C_BINDING
INTEGER(C_INT), BIND(C) :: C_EXTERN
INTEGER(C_LONG) :: C2
BIND(C, NAME = 'myVariable') :: C2
COMMON /COM/ R, S
REAL(C_FLOAT) :: R, S
BIND(C) :: /COM/
```

Такая переменная имеет метку связывания, определяемую по тем же правилам, которые действуют для процедур, и которая соответствует переменной C аналогичного типа.

## Примеры

Приведем пример вызова функции C из программы на Фортране. Прототип функции C имеет следующий вид:

```
int C_Library_Function(void* sendbuf, int sendcount, int *recvcounts)
```

Исходный текст модуля Фортрана приводится ниже:

```
MODULE FTN_C
 INTERFACE
        INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
        (SENDBUF, SENDCOUNT, RECVCOUNTS), &
        BIND(C, NAME = 'C_Library_Function')
               USE ISO_C_BINDING
               IMPLICIT NONE
               TYPE(C_PTR), VALUE :: SENDBUF
               INTEGER(C_INT), VALUE :: SENDCOUNT
               TYPE(C_PTR), VALUE :: RECVCOUNTS
        END FUNCTION C_LIBRARY_FUNCTION
 END INTERFACE
END MODULE FTN_C
```

Вызов процедуры из Фортрана имеет вид:

```
USE ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
 USE FTN_C
…
 REAL(C_FLOAT), TARGET :: SEND(100)
 INTEGER(C_INT) :: SENDCOUNT
 INTEGER(C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(:)
…
 ALLOCATE(RECVCOUNTS(100))
…
 CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
 C_LOC(RECVCOUNTS))
…
```

Следующий пример демонстрирует вызов подпрограммы на Фортране из программы на С. Исходный текст процедуры на Фортране:

```
SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS), BIND(C)
 USE ISO_C_BINDING
 IMPLICIT NONE
 INTEGER (C_LONG), VALUE :: ALPHA
 REAL (C_DOUBLE), INTENT(INOUT) :: BETA
 INTEGER (C_LONG), INTENT(OUT) :: GAMMA
 REAL (C_DOUBLE), DIMENSION(*), INTENT(IN) :: DELTA
 TYPE, BIND(C) :: PASS
        INTEGER (C_INT) :: LENC, LENF
        TYPE (C_PTR) :: C, F
 END TYPE PASS
 TYPE (PASS), INTENT(INOUT) :: ARRAYS
 REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
 REAL (C_FLOAT), POINTER :: C_ARRAY(:)
…
 CALL C_F_POINTER (ARRAYS%C, C_ARRAY, (/ARRAYS%LENC/))
…
 ARRAYS%LENF = 100
 ALLOCATE (ETA(ARRAYS%LENF))
 ARRAYS%F = C_LOC(ETA)
…
END SUBROUTINE SIMULATION
```

Описание структуры C имеет вид:

```
struct pass {
int lenc, lenf;
*c, float* f
}
```

Прототип функции C:

```
void simulation(long alpha, double *beta, long *gamma,
double delta[], struct pass *arrays)
```

Вызов из программы на C имеет вид:

```
simulation(alpha, &beta, &gamma, delta, &arrays);
```

# Удаленные и устаревшие элементы языка

Из нового стандарта удалены следующие элементы языка.

- Вещественные управляющие переменные цикла DO. При необходимости можно использовать цикл DO без параметра, но со вспомогательной переменной и проверкой условия завершения цикла.
- Переход на END IF извне условного оператора. При необходимости можно использовать оператор CONTINUE, расположенный сразу же после оператора END IF.
- Оператор PAUSE. При необходимости можно использовать вывод сообщения на соответствующее устройство и его последующий ввод.
- ASSIGN и назначаемый оператор перехода GO TO, а также назначаемые спецификаторы формата. Результат, аналогичный действию этих операторов, можно получить с помощью других операторов и конструкций языка.
- Дескриптор редактирования H. Вместо него можно использовать дескриптор редактирования символьной строки.

Стандарт определяет устаревшие, а потому нерекомендуемые элементы языка. Они сохраняются только для совместимости с предыдущими версиями. Ниже приведен перечень устаревших элементов.

- Арифметический условный оператор IF. Вместо него рекомендуется использовать оператор IF…THEN…ELSE….
- Завершение вложенных циклов DO общим оператором, а также завершение цикла оператором, отличным от END DO или CONTINUE. Для каждого цикла DO рекомендуется использовать отдельный оператор END DO или CONTINUE.
- Альтернативный возврат. При использовании альтернативного возврата список аргументов подпрограммы содержит метки, которые позволяют

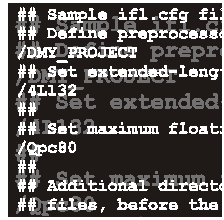
задать ход выполнения программы после завершения подпрограммы. Вместо этого можно использовать код завершения и оператор CASE, например:

```
CALL SUBRNAME(X, Y, Z, *100, *200, *300)
```

можно заменить следующим текстом:

```
CALL SUBRNAME(X, Y, Z, RETURN_CODE)
SELECT CASE(RETURN_CODE)
 CASE (1)
…
 CASE (2)
…
 CASE (3)
…
 CASE DEFAULT
…
END SELECT
```

- Вычисляемый оператор перехода GO TO. Его можно заменить конструкцией CASE.
- Оператор-функция. Для оператор-функций действует множество не всегда очевидных и удобных ограничений. Они являются потенциальным источником ошибок, так как их синтаксис схож с синтаксисом оператора присваивания. Внутренняя функция является более общей формой оператор-функции и полностью ее превосходит.
- Размещение операторов DATA среди исполняемых операторов. В стандартах Фортран 66/77/90 разрешалось размещать оператор DATA в любом месте программы после предложений описания. Эта возможность считается небезопасной.
- Символьные функции с подразумеваемой длиной. Такие функции можно заменить символьными функциями с автоматической длиной и объявлением длины результата в предложении описания. Формальные аргументы функции могут иметь подразумеваемую длину.
- Форма описания символьных переменных CHARACTER*. В Фортране 90 имеются две формы задания селектора длины в описаниях CHARACTER. Старая форма (CHARACTER*длина) является избыточным свойством языка.
- Фиксированный формат записи исходного текста программы. Этот формат был разработан для записи программ на перфокарты. Сейчас, когда исходный текст программы обычно вводится с клавиатуры, необходимость в нем отпала. Кроме того, фиксированный формат записи является потенциальным источником ошибок, например, вследствие неправильного использования позиций 6, 7 или 72.




# Приложение 1

# Перечень операторов языка

Все операторы разделены на две группы — неисполняемые (см. табл. П1.1, П1.2) и исполняемые (см. табл. П1.3–П1.7). Некоторые операторы, связанные с нерекомендуемыми свойствами языка, опущены (ENTRY, арифметический GOTO). Необязательные элементы заключены в квадратные скобки []. Если символ пробела не окружен квадратными скобками, то он является обязательным.

## Неисполняемые операторы

***Таблица П1.1.*** *Операторы программных компонент*

| Оператор | Комментарии | Раздел |
|---|---|---|
| PROGRAM имя_программы | Оператор заголовка главной программы | *Гл. 7, разд. "Главная программа"* |
| MODULE имя_модуля | Оператор заголовка модуля | *Гл. 7, разд. "Модули"* |
| END[ MODULE[ имя_модуля]] | Оператор завершения модуля | *Гл. 7, разд. "Модули"* |
| USE имя_модуля[, ONLY only_ список] | Оператор подключения модуля | *Гл. 7, разд. "Оператор USE"* |
| [RECURSIVE ]SUBROUTINE имя_подпрограммы [([список_формальных_ параметров])] | Оператор заголовка подпрограммы-процедуры | *Гл. 7, разд. "Внешние подпрограммы"* |
| [тип ][RECURSIVE ]FUNCTION имя_ функции ([список_ формальных_параметров]) [ RESULT(имя_результата)] | Оператор заголовка подпрограммы-функции | *Гл. 7, разд. "Внешние подпрограммы", "Рекурсивные процедуры и функции"* |

***Таблица П1.1*** *(окончание)*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `INTERFACE[ родовое_описание]` | Оператор заголовка интерфейса | *Гл. 7, разд. "Интерфейсы", "Именованные интерфейсы и совмещение процедур"* |
| `END[ ]INTERFACE` | Оператор завершения интерфейса | *Гл. 7, разд. "Интерфейсы"* |
| `CONTAINS` | Оператор содержания | *Гл. 7, разд. "Главная программа", "Внешние подпрограммы"* |
| `ENTRY` | | |
| `BLOCK[ ]DATA[ имя_блока_данных]` | Оператор заголовка блока данных | |
| `END[ ]BLOCK[ ]DATA[ имя_блока_данных]` | Оператор завершения блока данных | |

***Таблица П1.2.*** *Операторы описания и инициализации данных*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `MODULE PROCEDURE список_имен_модульных_процедур` | Оператор декларации модульных процедур | *Гл. 7, разд. "Именованные интерфейсы и совмещение процедур"* |
| `тип[[,атрибут][,атрибут…]….::] список_объектов` | Оператор описания | *Гл. 3* |
| где тип выбирается из списка: | | |
| `INTEGER[(KIND=]параметр_разновидности_типа)]` | | |
| `REAL[(KIND=]параметр_разновидности_типа)]` | | |
| `LOGICAL[(KIND=]параметр_разновидности_типа)]` | | |
| `COMPLEX[(KIND=]параметр_разновидности_типа)]` | | |
| `CHARACTER[список_параметров_типа]` | | |
| `DOUBLE[ ]PRECISION]` | | |
| `TYPE(имя_типа)` | | |

***Таблица П1.2*** *(продолжение)*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `тип[[,атрибут][,атрибут…]….::]список_объектов`<br>и атрибуты — совместимая комбинация из следующих значений:<br>`PARAMETER, PUBLIC, PRIVATE, POINTER, TARGET, ALLOCATABLE, DIMENSION(список_экстентов), INTENT(параметр_входа/выхода), EXTERNAL, INTRINSIC, OPTIONAL, SAVE` | Оператор описания | *Гл. 3* |
| `TYPE[, атрибут_доступа ::]имя_производного_типа атрибут_доступа` — `PUBLIC` или `PRIVATE` | Оператор определения производного типа, заголовок | *Гл. 3, разд. "Определение производных типов"* |
| `END[ ]TYPE[ имя_типа]` | Оператор определения производного типа: завершение | *Гл. 3, разд. "Определение производных типов"* |
| `IMPLICIT список`, где список — это `тип(список-букв)[, тип(список_букв)]…` или `NONE` | Оператор определения правил неявной типизации | *Гл. 3, разд. "Неявное определение типа"* |
| `ALLOCATABLE [::] имя_массива[(список_экстентов)][, имя_массива[(список_экстентов)]…]` | Оператор назначения атрибута `ALLOCATABLE` | *Гл. 3, разд. "Атрибуты* `DIMENSION` *и* `ALLOCATABLE`*…"* |
| `DIMENSION имя_массива(список_экстентов)[, имя_массива(список_экстентов)…]` | Оператор спецификации массивов | *Гл. 3, разд. "Атрибуты* `DIMENSION` *и* `ALLOCATABLE`*…"* |
| `POINTER [::] имя_объекта[(список_экстентов)][, имя_объекта[(список_экстентов)]…]` | Оператор назначения атрибута `POINTER` | *Гл. 3, разд. "Атрибуты* `POINTER` *и* `TARGET`*"* |
| `TARGET [::] имя_объекта[(список_экстентов)][, имя_объекта[(список_экстентов)]…]` | Оператор назначения атрибута `POINTER` | *Гл. 3, разд. "Атрибуты* `POINTER` *и* `TARGET`*"* |
| `PARAMETER (список_определений_именованных_констант)` | Оператор определения именованных констант | *Гл. 3, разд. "Атрибут* `PARAMETER`*. Именованные константы и константные выражения"* |
| `EXTERNAL список_внешних_имен` | Оператор назначения атрибута `EXTERNAL` | *Гл. 3, разд. "Атрибуты* `INTRINSIC` *и* `EXTERNAL`*"* |

***Таблица П1.2*** *(окончание)*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `INTRINSIC список_встроенных_имен` | Оператор назначения атрибута `INTRINSIC` | *Гл. 3, разд. "Атрибуты* `INTRINSIC` *и* `EXTERNAL`*"* |
| `INTENT(параметр_входа/выхода) список_формальных_параметров` | Оператор назначения атрибута `INTENT` | *Гл .3, разд. "Атрибуты* `OPTIONAL` *и* `INTENT`*. Описание формальных параметров"* |
| `OPTIONAL список_ формальных_параметров` | Оператор назначения атрибута `OPTIONAL` | *Гл. 3, разд. "Атрибуты* `OPTIONAL` *и* `INTENT`*. Описание формальных параметров"* |
| `SAVE[[::]список_сохраняемых_ объектов]` | Оператор назначения атрибута `SAVE` | *Гл. 3, разд. "Атрибут* `SAVE`*. Сохранение значений локальных переменных"* |
| `NAMELIST /имя_группы/ список_переменных[, /имя_группы/ список_переменных…]` | Оператор описания namelist-группы | *Гл. 9, разд. "Оператор* `NAMELIST`*"* |
| `COMMON /[имя_общего_блока] /список_переменных[, /имя_общего_блока/ список_переменных….]` | Оператор описания блока общей памяти | *Гл. 7, разд. "Области общей памяти"* |
| `EQUIVALANCE (объект, список_объектов)[, (объек, список_объектов)…]` | Оператор совмещения в памяти | *Гл. 7, разд. "Области общей памяти"* |
| `SEQUENCE` | Оператор назначения одноименного атрибута | *Гл. 7, разд. "Области общей памяти"* |
| `DATA список_объектов/ список_значений/[, список_ объектов/список_значений/…]` | Оператор инициализации объектов | *Гл. 3, разд. "Назначение исходных значений переменных"* |
| `FORMAT([список_дескрипторов])*` | Оператор спецификации формата преобразования данных при операциях ввода/вывода | *Гл. 9, разд. "Форматирование ввода-вывода"* |

* В отличие от других операторов описания, FORMAT может располагаться в любом месте программы.

# Исполняемые операторы

***Таблица П1.3.*** *Операторы передачи управления*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `END[ PROGRAM[ имя_программы]]` | Оператор завершения главной программы | *Гл. 7, разд. "Главная программа"* |
| `END[ подпрограмма[ имя_подпрограммы]]` где подпрограмма — `SUBROUTINE` или `FUNCTION` | Оператор завершения подпрограммы | *Гл. 7, разд. "Внешние подпрограммы", "Внутренние подпрограммы"* |
| `CALL имя_подпрограммы[(список_фактических_параметров)]` | Оператор передачи управления подпрограмме | *Гл. 7, разд. "Внешние подпрограммы", "Внутренние подпрограммы"* |
| `RETURN` | Оператор возврата управления вызывающему компоненту | *Гл. 7, разд. "Внешние подпрограммы"* |
| `STOP[ сообщение]` | Оператор останова программы | *Гл. 6, разд. "Оператор останова"* |

***Таблица П1.4.*** *Операторы присваивания*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `переменная = выражение` | Оператор присваивания для скалярных и массивоподобных объектов | *Гл. 4* |
| `ссылка => адресат` | Оператор прикрепления ссылки к адресату | *Гл. 4, разд. "Указатели (ссылки) в выражениях и операторах присваивания"* |

***Таблица П1.5.*** *Управляющие конструкции и условные операторы*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `IF(скалярное_логическое_выражение) исполняемый_оператор` | Условный оператор | *Гл. 6, разд. "Ветвления"* |
| `WHERE(логическое_выражение_массив) массив_переменная = выражение_массив` | Условное присваивание для массивов | *Гл. 8, разд. "Оператор и конструкция `WHERE`"* |

***Таблица П1.5*** *(окончание)*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `[if_имя:] IF(скалярное_логическое_выражение) THEN`<br>`ELSE[[ ]IF(скалярное_логическое_выражение) THEN[ if_имя]`<br>`END[ ]IF[ if_имя]` | Конструкция ветвления `IF_THEN_ELSE` | *Гл. 6, разд. "Ветвления"* |
| `WHERE(логическое_выражение_массив)`<br>`ELSEWHERE`<br>`END[ ]WHERE` | Конструкция ветвления в присваивании массивов | *Гл. 8, разд. "Оператор и конструкция* `WHERE`*"* |
| `[select_имя:] SELECT[ ]CASE (скалярное_выражение)`<br>`CASE (список_возможных_значений) [ select_имя]`<br>`CASE DEFAULT[ select_имя]`<br>`END[ ]SELECT[ select_имя]` | Конструкция `SELECT` | *Гл. 6, разд. "Ветвления"* |
| `GO[ ]TO метка` | Оператор перехода | *Гл. 6, разд. "Оператор перехода"* |
| `[do_имя:] DO[ метка] переменная= скалярное_целое_выражение1, скалярное_целое_выражение2[, скалярное_целое_выражение3]` | Заголовок оператора цикла | *Гл. 6, разд. "Организация циклов"* |
| `[do_имя:] DO[ метка] [,] WHILE(скалярное_лог_выражение)` | Заголовок оператора цикла в альтернативной форме | *Гл. 6, разд. "Организация циклов"* |
| `CYCLE[ do_имя]` | Оператор перехода к оператору завершения цикла с именем `do_имя` | *Гл. 6, разд. "Организация циклов"* |
| `EXIT[ do_имя]` | Оператор выхода из цикла с именем `do_имя` | *Гл. 6, разд. "Организация циклов"* |
| `CONTINUE` | Оператор перехода к следующему шагу цикла | *Гл. 6, разд. "Организация циклов"* |
| `END[ ]DO[ do_имя]` | Оператор завершения цикла с именем `do_имя` | *Гл. 6, разд. "Организация циклов"* |

***Таблица П1.6.*** *Операторы управления динамической памятью*

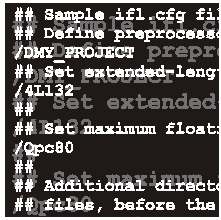
| Оператор | Комментарии | Раздел |
|---|---|---|
| `ALLOCATE (список_выделяемых_объектов[,STAT=статус])` | Оператор выделения динамической памяти для выделяемых объектов | *Гл. 8, разд. "Операторы* `ALLOCATE`, `DEALLOCATE` *и* `NULLIFY`*"* |
| `DEALLOCATE (список_выделенных_объектов[,STAT=статус])` | Оператор освобождения динамической памяти от выделенных объектов | *Гл. 8, разд. "Операторы* `ALLOCATE`, `DEALLOCATE` *и* `NULLIFY`*"* |
| `NULLIFY (список_ссылочных_объектов)` | Оператор открепления ссылочных объектов от адресатов | *Гл. 8, разд. "Операторы* `ALLOCATE`, `DEALLOCATE` *и* `NULLIFY`*"* |

***Таблица П1.7.*** *Операторы ввода/вывода*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `READ (список_управления_вводом) [список_ввода]`<br>`READ формат[, список_ввода]` | Оператор чтения данных | *Гл. 9, разд. "Операторы передачи данных"* |
| `WRITE (список_управления_выводом) [список_вывода]` | Оператор записи данных | *Гл. 9, разд. "Операторы передачи данных"* |
| `PRINT формат[, список_вывода]` | Оператор вывода данных на устройство стандартного вывода | *Гл. 9, разд. "Операторы передачи данных"* |
| `OPEN(список_спецификаций_соединения)` | Оператор соединения файла с логическим устройством ввода-вывода | *Гл. 9, разд. "Соединение файла с логическим устройством"* |
| `CLOSE(список_спецификаций_закрытия)` | Оператор закрытия файла | *Гл. 9, разд. "Соединение файла с логическим устройством"* |
| `INQUIRE(список_спецификаторов_запроса)`<br>`INQUIRE(IOLENGTH=длина) список_вывода` | Оператор сбора информации о состоянии устройства ввода-вывода | *Гл. 9, разд. "Оператор* `INQUIRE`*"* |

***Таблица П1.7*** *(окончание)*

| Оператор | Комментарии | Раздел |
|---|---|---|
| `REWIND номер_логического_устройства`<br>`REWIND(список_спецификаторов_позиционирования)` | Оператор позиционирования файлового указателя в начало файла последовательного доступа | *Гл. 9, разд. "Операторы управления файловым указателем"* |
| `BACKSPACE номер_логического_устройства`<br>`BACKSPACE(список_спецификаторов_позиционирования)` | Оператор позиционирования файлового указателя в начало предыдущей записи | *Гл. 9, разд. "Операторы управления файловым указателем"* |
| `END[ ]FILE номер_логического_устройства`<br>`END[ ]FILE(список_спецификаторов_позиционирования)` | Оператор установки метки "конец файла" в файле последовательного доступа | *Гл. 9, разд. "Операторы управления файловым указателем"* |




# Приложение 2

# Перечень встроенных подпрограмм

Список встроенных подпрограмм дан в алфавитном порядке (см. табл. П2.1). В третьем столбце таблицы указан раздел *гл. 10*, в котором приведено полное описание подпрограммы. Все подпрограммы-процедуры начинаются с ключевого слова `CALL`. Встроенные функции-двойники "двойной" точности, например, такие как `DIMAG` (вещественная часть комплексного числа с двойной точностью), опущены как устаревшие.

***Таблица П2.1.*** *Встроенные подпрограммы*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `ABS(A)` | Абсолютная величина | *"Математические функции"* |
| `ACHAR(I)` | I-ый символ сортирующей последовательности ASCII | *"Текстовые функции"* |
| `ACOS(X)` | Арккосинус в радианах | *"Математические функции"* |
| `ADJUSTL(STRING)` | Выравнивание строки влево | *"Текстовые функции"* |
| `ADJUSTR(STRING)` | Выравнивание строки вправо | *"Текстовые функции"* |
| `AIMAG(Z)` | Мнимая часть комплексного числа | *"Математические функции"* |
| `AINT(A[, KIND])` | Усечение до целого | *"Математические функции"* |
| `ALL(MASK [, DIM])` | Логическое перемножение массива | *"Функции редукции массивов"* |
| `ALLOCATED(ARRAY)` | Проверка выделенности динамического массива | *"Справочные функции для массивов"* |

***Таблица П2.1*** *(продолжение)*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `ANINT(A [, KIND])` | Ближайшее целое | *"Математические функции"* |
| `ANY(MASK [, DIM])` | Логическое сложение массива | *"Функции редукции массивов"* |
| `ASIN(X)` | Арксинус в радианах | *"Математические функции"* |
| `ASSOCIATED(POINTER [,TARGET])` | Проверка привязанности ссылки | *"Справочные функции для любого типа"* |
| `ATAN(X)` | Арктангенс в радианах | *"Математические функции"* |
| `ATAN2(Y, X)` | Аргумент комплексного числа | *"Математические функции"* |
| `BIT_SIZE(I)` | Число битов в целом данной разновидности типа | *"Процедуры для работы с битами"* |
| `BTEST(I, POS)` | Проверка заданного бита | *"Процедуры для работы с битами"* |
| `CALL DATE_AND_TIME([DATE][,TIME][,ZONE][,VALUES])` | Считывание времени и даты с часов реального времени | *"Процедуры определения даты и времени"* |
| `CALL MVBITS (FROM, FROMPOS,LEN, TO,TOPOS)` | Копирование цепочки битов | *"Процедуры для работы с битами"* |
| `CALL RANDOM_NUMBER(HARVEST)` | Случайное число в интервале [0,1] | *"Случайные числа"* |
| `CALL RANDOM_SEED ([SIZE] ,PUT][,GET])` | Получение/назначение затравочного массива | *"Случайные числа"* |
| `CALL SYSTEM_CLOCK([COUNT] [,COUNT_RATE] [,COUNT_MAX])` | Целочисленный отсчет часов реального времени | *"Процедуры определения даты и времени"* |
| `CEILING(A)` | Наименьшее целое не меньшее аргумента | *"Математические функции"* |
| `CHAR(I [, KIND])` и `ICHAR(C)` | I-ый символ в сортирующей последовательности процессора | *"Текстовые функции"* |

***Таблица П2.1*** *(продолжение)*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `CMPLX(X [, Y] [, KIND])` | Конструктор комплексного числа | *"Функции преобразования и переноса типов"* |
| `CONJG(Z)` | Комплексное сопряжение | *"Математические функции"* |
| `COS(X)` | Косинус | *"Математические функции"* |
| `COSH(X)` | Гиперболический косинус | *"Математические функции"* |
| `COUNT(MASK [, DIM])` | Число элементов массива, имеющих значение "истина" | *"Функции редукции массивов"* |
| `CSHIFT(ARRAY, SHIFT [, DIM])` | Циклический сдвиг элементов массива | *"Функции для создания массивов и операций над ними"* |
| `DIGITS(X)` | Число значащих цифр в модели представления аргумента | *"Числовые справочные функции"* |
| `DIM(X, Y)` | MAX(X-Y, 0) | *"Математические функции"* |
| `DOT_PRODUCT(VECTOR_A, VECTOR_B)` | Скалярное произведение векторов | *"Операции с векторами и матрицами"* |
| `DPROD(X, Y)` | Умножение с двойной точностью | *"Математические функции"* |
| `EOSHIFT(ARRAY, SHIFT [, BOUNDARY] [, DIM])` | Вытесняющий сдвиг элементов массива | *"Функции для создания массивов и операций над ними"* |
| `EPSILON(X)` | Наименьшее число в модели представления аргумента, не пренебрежимое по сравнению с единицей | *"Числовые справочные функции"* |
| `EXP(X)` | Экспонента | *"Математические функции"* |
| `EXPONENT(X)` | Степенная часть в модели представления аргумента | *"Числовые справочные функции"* |
| `FLOOR(A)` | Наименьшее целое, не превышающее аргумент | *"Математические функции"* |

***Таблица П2.1*** *(продолжение)*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `FRACTION(X)` | Дробная часть в модели представления аргумента | *"Числовые справочные функции"* |
| `HUGE(X)` | Наибольшее число в модели представления аргумента | *"Числовые справочные функции"* |
| `IACHAR(C)` | Индекс символа-аргумента в сортирующей последовательности ASCII | *"Текстовые функции"* |
| `IAND(I, J)` | Побитовое логическое "И" | *"Процедуры для работы с битами"* |
| `IBCLR(I, POS)` | Установка нуля в заданном бите | *"Процедуры для работы с битами"* |
| `IBITS(I, POS, LEN)` | Извлечение цепочки битов | *"Процедуры для работы с битами"* |
| `IBSET(I, POS)` | Установка единицы в заданном бите | *"Процедуры для работы с битами"* |
| `ICHAR(C)` | Индекс символа-аргумента в сортирующей последовательности процессора | *"Текстовые функции"* |
| `IEOR(I, J)` | Побитовое "ИСКЛЮЧАЮЩЕЕ ИЛИ" | *"Процедуры для работы с битами"* |
| `INDEX(STRING, SUBSTRING [,BACK])` | Индекс начала подстроки в строке | *"Процедуры для работы с битами"* |
| `INT(A [, KIND])` | Приведение к целому типу | *"Функции преобразования и переноса типов"* |
| `IOR(I, J)` | Побитовое логическое "ИЛИ" | *"Процедуры для работы с битами"* |
| `ISHIFT(I, SHIFT)` | Логический сдвиг битов | *"Процедуры для работы с битами"* |

***Таблица П2.1*** *(продолжение)*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `ISHIFTC(I, SHIFT [, SIZE])` | Логический циклический сдвиг части битов вправо | *"Процедуры для работы с битами"* |
| `KIND(X)` | Разновидность типа аргумента | *"Справочные функции для любого типа"* |
| `LBOUND(ARRAY [, DIM])` | Нижняя граница массива | *"Справочные функции для массивов"* |
| `LEN(S)` | Длина текстового аргумента | *"Справочная функция для строк"* |
| `LEN_TRIM(STRING)` | Длина строки без учета конечных пробелов | *"Текстовые функции"* |
| `LGE(STRING_A, STRING_B)` | Оператор сравнения (в смысле сортировки) строк: "больше или равно" | *"Текстовые функции"* |
| `LGT(STRING_A, STRING_B)` | Оператор сравнения (в смысле сортировки) строк: "больше" | *"Текстовые функции"* |
| `LLE(STRING_A, STRING_B)` | Оператор сравнения (в смысле сортировки) строк: "меньше или равно" | *"Текстовые функции"* |
| `LLT(STRING_A, STRING_B)` | Оператор сравнения (в смысле сортировки) строк: "меньше" | *"Текстовые функции"* |
| `LOG(X)` | Натуральный логарифм | *"Математические функции"* |
| `LOG10(X)` | Десятичный логарифм | *"Математические функции"* |
| `LOGICAL(L [, KIND])` | Приведение к заданной разновидности логического типа | *"Функции преобразования и переноса типов"* |
| `MATMUL(MATRIX_A, MATRIX_B)` | Перемножение матриц | *"Операции с векторами и матрицами"* |
| `MAX(A1, A2 [, A3, …..])` | Выбор максимального аргумента | *"Математические функции"* |
| `MAXEXPONENT(X)` | Наибольший показатель степени в модели представления аргумента | *"Числовые справочные функции"* |

***Таблица П2.1*** *(продолжение)*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `MAXLOC(ARRAY [, MASK])` | Индекс наибольшего элемента массива | *"Функции для поиска в массиве"* |
| `MAXVAL(ARRAY [, DIM ] [,MASK])` | Наибольший элемент массива | *"Функции для поиска в массиве"* |
| `MERGE(TSOURCE, FSOURCE, MASK)` | Слияние массивов под управление логического массива | *"Функции для создания массивов и операций над ними"* |
| `MIN(A1, A2 [, A3, …..])` | Выбор минимального аргумента | *"Математические функции"* |
| `MINEXPONENT(X)` | Наименьший показатель степени в модели представления аргумента | *"Числовые справочные функции"* |
| `MINLOC(ARRAY [, MASK])` | Индекс наименьшего элемента массива | *"Функции для поиска в массиве"* |
| `MINVAL(ARRAY [, DIM ] [,MASK])` | Наименьший элемент массива | *"Функции для поиска в массиве"* |
| `MOD(A, P)` | Остаток от деления по модулю | *"Математические функции"* |
| `MODULO(A, P)` | Деление по модулю | *"Математические функции"* |
| `MVBITS(FROM,FROMPOS, TO,TOPOS)` | См. *CALL MVBITS...* | |
| `NEAREST(X, S)` | Ближайшее машинное число в заданном направлении | *"Числовые функции для действий над действительными числами"* |
| `NINT(A [, KIND])` | Ближайшее целое | *"Функции преобразования и переноса типов"* |
| `NOT(I)` | Побитовое логическое дополнение | *"Процедуры для работы с битами"* |

***Таблица П2.1*** *(продолжение)*

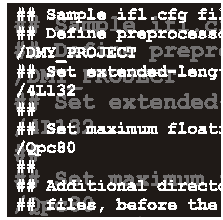
| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `PACK(ARRAY, MASK [,VECTOR])` | Упаковка массива | *"Функции для создания массивов и операций над ними"* |
| `PRECISION(X)` | Десятичная точность в модели представления аргумента | *"Числовые справочные функции"* |
| `PRESENT(A)` | Проверка присутствия необязательного аргумента | *"Справочные функции для любого типа"* |
| `PRODUCT(ARRAY [, DIM] [,MASK])` | Произведение элементов массива | *"Функции редукции массивов"* |
| `RADIX(X)` | Основание в модели представления аргумента | *"Числовые справочные функции"* |
| `RANDOM_ NUMBER(HARVEST)` | См. *CALL RANDOM_NUMBER...* | *"Случайные числа"* |
| `RANDOM_SEED([SIZE] [,PUT] [,GET])` | См. *CALL RANDOM_SEED...* | *"Случайные числа"* |
| `RANGE(X)` | Десятичный диапазон порядка в модели представления аргумента | *"Числовые справочные функции"* |
| `REAL(A [, KIND])` | Приведение к вещественному типу | *"Функции преобразования и переноса типов"* |
| `REPEAT(STRING, NCOPIES)` | "Размножение" строки | *"Текстовые функции"* |
| `RESHAPE(SOURCE, SHAPE [,PAD] [, ORDER])` | Изменение формы массива | *"Функции для создания массивов и операций над ними"* |
| `RRSPASING(X)` | Обратная величина относительного расстояния между числами в модели представления аргумента | *"Числовые функции для действий над действительными числами"* |

***Таблица П2.1*** *(продолжение)*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `SCALE(X,I)` | Умножение X на основание модели представления X в степени I | *"Числовые функции для действий над действительными числами"* |
| `SCAN(STRING, SET[,BACK])` | Индекс крайнего символа строки STRING в наборе SET | *"Текстовые функции"* |
| `SELECTED_INT_KIND(R)` | Параметр разновидности целого типа для заданного степенного диапазона | *"Числовые справочные функции"* |
| `SELECTED_REAL_ KIND([P][,R])` | Параметр разновидности вещественного типа для заданной точности и/или заданного степенного диапазона | *"Числовые справочные функции"* |
| `SET_EXPONENT(X, I)` | Модельное число порядка I со знаком и дробной частью X | *"Числовые функции для действий над действительными числами"* |
| `SHAPE(SOURCE)` | Форма аргумента | *"Справочные функции для массивов"* |
| `SIGN(A, B)` | Абсолютное значение A со знаком B | *"Математические функции"* |
| `SIN(X)` | Синус | *"Математические функции"* |
| `SINH(X)` | Гиперболический синус | *"Математические функции"* |
| `SIZE(ARRAY[,DIM])` | Размер массива | *"Справочные функции для массивов"* |
| `SPACING(X)` | Абсолютное расстояние между числами в модели представления аргумента | *"Числовые функции для действий над действительными числами"* |
| `SPREAD(SOURCE,DIM, NCOPIES)` | "Размножение" массива | *"Функции для создания массивов и операций над ними"* |

***Таблица П2.1*** *(окончание)*

| Имя подпрограммы | Возвращаемое значение или производимое действие | Глава 10, раздел |
|---|---|---|
| `SQRT(X)` | Квадратный корень | *"Математические функции"* |
| `SUM(ARRAY[,DIM][,MASK])` | Сумма элементов массива | *"Функции редукции массивов"* |
| `SYSTEM_CLOCK([COUNT][,COUNT_RATE][,COUNT_MAX]` | См. *CALL SYSTEM_CLOCK...* | *"Процедуры определения даты и времени"* |
| `TAN(X)` | Тангенс | *"Математические функции"* |
| `TANH(X)` | Гиперболический тангенс | *"Математические функции"* |
| `TINY(X)` | Наименьшее положительное число в модели представления аргумента | *"Числовые справочные функции"* |
| `TRANSFER(SOURCE, MOLD [,SIZE])` | Перенос данных из одного типа в другой без изменения их физического представления | *"Функции преобразования и переноса типов"* |
| `TRANSPOSE(MATRIX)` | Транспонирование матрицы | *"Операции с векторами и матрицами"* |
| `TRIM(STRING)` | Удаляет конечные пробелы в строке | *"Текстовые функции"* |
| `UBOUND(ARRAY [, DIM])` | Верхняя граница массива | *"Справочные функции для массивов"* |
| `UNPACK(VECTOR, MASK, FIELD)` | Распаковка массива | *"Функции для создания массивов и операций над ними"* |
| `VERIFY(STRING, SET[,BACK])` | Индекс крайнего символа строки STRING, отсутствующего в наборе SET | *"Текстовые функции"* |




# Приложение 3

# Библиотеки численных методов

В этом приложении приводится краткий перечень наиболее известных библиотек численных методов, которые можно использовать в программах на Фортране.

## BLAS

BLAS (Basic Linear Algebra System) представляет собой набор оптимизированных и высококачественных подпрограмм, реализующих основные матричные и векторные операции. Они служат строительными блоками для более сложных алгебраических вычислений.

Выделяют несколько уровней BLAS:

- первый уровень — основные векторно-векторные операции;
- второй уровень — матрично-векторные операции;
- третий уровень — матрично-матричные операции.

Иногда выделяется дополнительный уровень для работы с разреженными матрицами. Базовая версия библиотеки находится в Интернете по адресу **www.netlib.org/blas**, но она содержит не оптимизированные подпрограммы и играет роль описания интерфейсов подпрограмм и примеров их использования. Рекомендуется использовать более быстрые реализации BLAS, такие как ATLAS.

## ATLAS

Проект ATLAS (Automatically Tuned Linear Algebra Software) посвящен созданию и применению эмпирических методов обеспечения высокой производительности на разных платформах. В настоящее время имеются интерфейсы с C и Фортраном 77 для эффективных и переносимых реализаций BLAS, а также подпрограмм библиотеки LAPACK.

В состав пакета ATLAS входят следующие статические библиотеки:

- liblapack — полная библиотека функций LAPACK, оптимизированных в рамках проекта ATLAS;
- libcblas — не многопоточный интерфейс ANSI C к BLAS;
- libf77blas — не многопоточный интерфейс Фортрана 77 к BLAS;
- libptcblas — многопоточный интерфейс ANSI C к BLAS;
- libptf77blas — многопоточный интерфейс Fortran77 к BLAS;
- libatlas — основная библиотека ATLAS с процедурами низкого уровня для всех интерфейсных библиотек.

При компиляции кода Фортрана 77 с библиотекой BLAS требуются следующие флаги компиляции:

```
-L/usr/global/atlas -lf77blas -latlas
```

Описание пакета ATLAS можно найти на странице проекта в Интернете **math-atlas.sourceforge.net** и **www.netlib.org/atlas**.

# LAPACK

LAPACK (Linear Algebra PACKage Libraries) содержит подпрограммы для решения систем линейных алгебраических уравнений, метода наименьших квадратов, решения задач на собственные значения и вычисления сингулярных разложений. Имеются процедуры факторизации хорошо заполненных и ленточных матриц, но отсутствуют процедуры для работы с разреженными матрицами.

В состав библиотеки LAPACK входят подпрограммы-драйверы, вычислительные подпрограммы и вспомогательные подпрограммы. Драйверы предназначены для решения основных задач линейной алгебры, таких как решение систем линейных алгебраических уравнений, а вычислительные подпрограммы решают частные задачи, такие как LU-разложение матриц. Вспомогательные подпрограммы выполняют вычислительные операции низкого уровня.

Описание библиотеки и руководство пользователя можно найти в Интернете по адресу **www.netlib.org/lapack**.

# ScaLAPACK

Библиотека ScaLAPACK (масштабируемый LAPACK) включает подмножество подпрограмм LAPACK, реализованных для параллельных MIMD компьютеров с распределенной памятью. Они написаны с использованием явной модели обмена сообщениями и межпроцессорных коммуникаций. Информацию об этой библиотеке можно найти на сайте **www.netlib.org**.

# IMSL

Библиотека IMSL является коммерческим продуктом, который разрабатывает фирма Visual Numerics (**www.vni.com**). Библиотека IMSL представляет собой набор математических и статистических функций, которые можно использовать в прикладных программах. Она предназначена для разработки надежных и эффективных программ для сложных численных расчетов.

# Intel MKL

Библиотека MKL (Intel Math Kernel Library) содержит подпрограммы для решения задач линейной алгебры, операций с векторами. Подпрограммы оптимизированы для архитектур Intel Pentium 4 и Intel Itanium. MKL включает в свой состав библиотеку LAPACK и расширенный BLAS для работы с разреженными матрицами. Имеется набор быстрых преобразований Фурье с простой и двойной точностью, вещественными и комплексными типами данных и интерфейсами с Фортраном и C. MKL содержит также набор векторизованных трансцендентных функций, входящих в состав VML (Vector Math Library), дающих более высокую производительность и точность, чем функции из библиотеки libm.

# NAG

NAG (The Numerical Algorithms Group Ltd, Oxford) выпускает библиотеки численных и статистических методов, предназначенные для решения сложных научно-технических задач. Это коммерческие продукты. Имеются версии библиотек NAG для Фортрана 77/90, C, а также библиотеки для симметричных многопроцессорных и параллельных вычислительных систем с различной организацией памяти.

Библиотеки Фортрана носят название Mark, одной из последних версий является версия Mark 20. Библиотека состоит из разделов-глав. Ниже приведен список некоторых из них:

- A02 — комплексная арифметика;
- C02 — вычисление корней полиномов;
- C05 — вычисление корней нелинейных уравнений и их систем;
- C06 — суммирование рядов;
- D01 — вычисление квадратур;
- D02 — решение обыкновенных дифференциальных уравнений;
- D03 — решение дифференциальных уравнений в частных производных;
- D04 — численное дифференцирование;

- D05 — решение интегральных уравнений;
- D06 — генерация сеток;
- E01 — интерполяция;
- E02 — сглаживание кривых и поверхностей;
- E04 — вычисление экстремумов функций;
- F01 — факторизация матриц;
- F02 — вычисление собственных значений и собственных векторов;
- F03 — вычисление детерминантов матриц;
- F04 — решение систем линейных алгебраических уравнений;
- F05 — ортогонализация;
- F06 — вспомогательные подпрограммы для алгебраических вычислений;
- F07 — решение линейных уравнений (LAPACK);
- F08 — метод наименьших квадратов и проблема собственных значений (LAPACK);
- F11 — подпрограммы для работы с разреженными матрицами;
- G01 — простая статистическая обработка;
- G02 — корреляционный и регрессионный анализ;
- G03 — многофакторный анализ;
- G04 — дисперсионный анализ;
- G05 — генераторы псевдослучайных чисел;
- G07 — одномерное оценивание;
- G08 — непараметрическая статистика;
- G10 — сглаживание статистических данных;
- G13 — анализ временных рядов;
- H — исследование операций;
- M01 — сортировка;
- P01 — определение ошибок;
- S — аппроксимация спецфункций;
- X01 — математические константы;
- X02 — машинные константы;
- X03 — вычисление скалярных произведений;
- X04 — ввод-вывод;
- X05 — подпрограммы определения даты и времени.

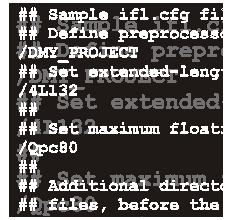
Имена подпрограмм, относящихся к каждому разделу, состоят из шести символов, первые три из которых — имя раздела, например: D01AJF.

Раздел F06 содержит все подпрограммы BLAS. Раздел F07 содержит подпрограммы, заимствованные из библиотеки LAPACK.

Реализации библиотеки NAG имеются для разных вычислительных платформ, компиляторов и режимов компиляции (например, скалярного и векторного). Поддерживается простая и двойная точность. Простая точность обозначается последней буквой имени E, а двойная — F, например: D01AJF, D01AJE.

Нестандартные целые типы (например, двухбайтовый) не поддерживаются.

Все подпрограммы библиотеки удовлетворяют стандарту Фортрана 90. Они могут использоваться в программах на других языках (C, Visual Basic). Имеются заголовочные файлы C, которые содержат описание соответствия типов данных Фортрана и C для различных компиляторов. Имеются документация и примеры. Полезную информацию об этой библиотеке можно найти на сайтах **www.nag.co.uk** и **www.nag.com**.




# Приложение 4

# Ресурсы Интернета, посвященные Фортрану

В Интернете имеется огромное количество ресурсов, посвященных программированию на Фортране. Получить их перечень можно, воспользовавшись поисковыми машинами и задав в качестве ключевого слова *Фортран* или *FORTRAN*. В данном приложении приведена лишь небольшая часть этого списка.

## Информация о компиляторах

- **http://www.intel.com/software/products/compilers/flin/noncom.htm** — свободно распространяемый для некоммерческого использования компилятор Intel для ОС Linux.
- **http://www.lahey.com** и **http://www.fortran.com/F** — семейство компиляторов Фортрана, а также компилятор языка F.
- **http://www.absoft.com** — компилятор фирмы Absoft для платформ Linux (Intel), Power Mac, UNIX и MS Windows.
- **http://www.cray.com/products/software/cf90.html** — компилятор CF90 для суперкомпьютеров Cray YMP и YMP-C90, Superserver 6400, Sparc Solaris и т. д.
- **http://www.compaq.com/fortran** — среда программирования и компилятор Compaq Fortran для MS Windows.
- **http://www.hp.com/go/hpfortran** — компилятор Hewlett Packard для Фортрана 90.
- **http://www-3.ibm.com/software/ad/fortran** — компилятор XL Fortran фирмы IBM, совместимый со стандартом Фортрана 95 и Фортраном OpenMp.
- **http://www.nasoftware.co.uk** — компилятор FortranPlus, поддерживающий Фортран 95 и некоторые расширения Фортрана 2003, графическую библиотеку OpenGL, интероперабельность с C. Включает графическую интегрированную среду.

- **http://www.nag.co.uk/nagware.asp** — компилятор NAGWare, поддерживает Фортран 95, HPF и некоторые конструкции Фортрана 2003.
- **http://www.pgroup.com** — компилятор Фортрана 90 и HPF для SGI, IBM SP2, HP/Convex, Linux.
- **http://www.salford.co.uk** — компилятор FTN90 Фортрана 90 для MS Windows.
- **http://www.sgi.com** — компилятор Фортрана и другие полезные программы для ОС IRIX и Linux.
- **http://g95.sourceforge.net** — компилятор Фортрана 95, разрабатываемый в рамках проекта GNU.
- **http://www.sun.com** — фирма Sun Microsystems выпускает компиляторы, в том числе и для Фортрана —The Sun ONE Studio 8, Compiler Collection (Forte Compiler Collection), а также другие программные продукты для программиста.
- **http://www.geocities.com/Athens/Olympus/5564/g77.htm** — компилятор g77 для Windows NT/2000/XP.
- **http://www.openwatcom.org/index.html** — компиляторы Open Watcom C/C++ и Фортрана.
- **http://www.idiom.com/free-compilers** — список свободно распространяемых компиляторов.

## Информация о стандартах

- **http://www.iso.ch** — официальный сайт ISO. Здесь можно найти информацию по стандартам.
- **http://www.j3-fortran.org** — комитет по стандартизации Фортрана США, работающий совместно с комитетом ISO/IEC/JTC1/SC22/WG5.
- **http://www.nag.co.uk/SC22WG5** — официальный сайт по стандартизации Фортрана.

## Информация общего характера

- **http://www.ifremer.fr/metocean/group/michel/michel_olagnon.htm** — персональная страница Мишеля Оланьона содержит впечатляющий список ссылок на ресурсы Интернета, посвященные программированию на Фортране и сопутствующим вопросам.
- **http://www.fortran.com/fortran/metcalf.htm** — часто задаваемые вопросы по Фортрану, а также другая полезная информация на странице Майкла Меткалфа.

- **ftp://rtfm.mit.edu/pub/usenet/comp.lang.fortran** — часто задаваемые вопросы Usenet группы comp.lang.fortran.
- **http://users.argonet.co.uk/users/fortran/index.htm** — клуб друзей Фортрана.
- **http://www-h.eng.cam.ac.uk/help/mjg17/f90/history.htm**, **http://sunsite.univie.ac.at/Fortran-Guide/ch1-1.html** — история Фортрана.
- **http://www.travelnotes.de/rays/fortran/fortran.htm**, **http://ifaq.wap.org/computers/realprogrammers.html**, **http://void.thunderteam.org/goodprog.html**, **http://www.pbm.com/~lindahl/mel.html**, **http://www.poppyfields.net/filks/00095.html** — шутки, посвященные Фортрану.
- **http://www.bcs.org.uk/siggroup/fortran** — раздел сайта Британского Вычислительного общества, посвященный Фортрану (BCS Fortran Specialist Group).
- **http://csep1.phy.ornl.gov** — сайт проекта Computational Science Education Project, который содержит полезную для программиста-вычислителя информацию.
- **http://sunsite.informatik.rwth-aachen.de/fortran** — заметки, посвященные программированию на Фортране.
- **http://sip.clarku.edu/links/programming/fortran.html** — коллекция ссылок, посвященных Фортрану.
- **http://www.nsc.liu.se/~boein/f77to90/fortran.html** — страница Бо Эйнарссона.
- **http://docs.sun.com** — полезная информация, в том числе и о том, что каждый программист должен знать об арифметике с плавающей точкой.
- **http://www.polyhedron.co.uk** — сравнительный обзор компиляторов Фортрана.
- **http://www.gis.rnd.runnet.ru/team/clubs/FClub.html** — "Клуб программистов на Фортране".
- **http://www.crpc.rice.edu/HPFF/home.html** — официальный сайт Высокопроизводительного Фортрана.
- **http://www.fortran-2000.com/ArnaudRecipes** — Арно Деситтер описывает свой опыт разработки программ.
- **http://wwwinfo.cern.ch/asdoc/f90.html** — статьи Майкла Меткалфа, посвященные программированию на Фортране.
- **http://www.nikhef.nl/~templon/fortran.html** — Фортран для ОС Linux.
- **http://users.bigpond.net.au/amiller** — сайт Алана Миллера, содержит исходные тексты программ.

# Информация о вспомогательных программах и утилитах

- **http://home.cfl.rr.com/davegemini** — конвертер из C в Фортран 90.
- **http://www.scitools.com/uf.html** — интегрированная среда для программирования на Фортране в ОС MS Windows и Unix.
- **http://www.cleanscape.net/products/fortranlint** — анализатор кода для Фортрана 77/90/95. Он помогает обнаружить такие ошибки, которые не выявляются во время компиляции.
- **ftp://ftp.ifremer.fr/ifremer/ditigo/fortran90/f90ppr.f90.gz** — препроцессор для Фортрана 90, аналогичный препроцессору cpp.
- **http://users.erols.com/dnagle/coco.html** — утилита для условной компиляции программ на Фортране 90.
- **http://www.promula.com/fortrantoc** — транслятор с Фортрана на C.
- **http://home.cfl.rr.com/davegemini/C2F.ZIP** — транслятор с C на Фортран.
- **http://www.cs.utk.edu/f2j** и **http://citeseer.nj.nec.com/fox97prototype.html** — трансляторы с Фортрана на Java.
- **http://www.cleanscape.net** — Fortranlint, анализатор кода от фирмы Cleanscape Software.
- **http://www.simulog.fr/is/2fore1.htm** — FORESYS, анализатор кода с удобным интерфейсом.
- **http://www.dsm.fordham.edu/~ftnchek** — FTNCHEK, свободно распространяемый анализатор кода для Фортрана.
- **http://www.simcon.uk.com** — анализатор кода.
- **http://hermes.desy.de/ww/f77prob.html** — инструментарий для решения проблем, возникающих при разработке программ на Фортране.
- **http://www.winternet.com/~gginc** и **ftp.winternet.com/users/gginc** — набор подпрограмм для работы с библиотекой Xlib.
- **http://www.canaimasoft.com/f90sql** — инструментарий для работы с базами данных.
- **http://sourceforge.net/projects/fiwizard** — The Fortran Interface Wizard, автоматический генератор кода для создания динамических библиотек в Фортране и интерфейсного кода для использования динамических библиотек из Borland Delphi.
- **http://cens.ioc.ee/projects/f2py2e** — FPIG, генератор интерфейса с Python.
- **http://www.ec-nantes.fr/DMN/WWW/Software/TkfPW** — TkfPW, среда для разработки программ на Фортране, ориентированная на работу в ОС Unix.

- **http://www.indowsway.com/home.htm** — RealWin, инструментарий для разработки Windows-приложений на Фортране.
- **http://math.nist.gov/f90gl** — интерфейс к графической библиотеке OpenGL для ОС Unix и Windows.
- **http://www.winteracter.com** — Winteracter, среда программирования с графическим интерфейсом.
- **http://www.forcheck.nl** — FORCHECK, анализатор кода программ на Фортране с разнообразными возможностями.
- **http://www.etnus.com** — отладчик TotalView.
- **http://www.intel.com/software/products/vtune/vpa** — анализатор производительности исполняемого кода VTune Performance Analyzer от фирмы Intel с разнообразными возможностями.

## Информация о библиотеках

- **http://www.nag.com/num_fortran_libraries.asp** — библиотеки численных методов на Фортране (а также C и Java) фирмы NAG.
- **http://www.nova.edu/ocean/psplot.html** — графическая библиотека.
- **http://www.linmpi.mpg.de/dislin/dislin.html** — библиотека научной графики.
- **http://www.pnl.gov/berc/flib/index.html** — математические библиотеки на Фортране 90. Распространяются свободно.
- **http://www.nr.com** — библиотека Numerical Recipes на Фортране и других языках.
- **http://www.ualg.pt/est/adec/csc/dxfortran** — векторная графическая библиотека на Фортране, распространяется свободно.
- **http://www.ancad.com** — MATFOR, библиотеки численных методов, а также графические библиотеки для Фортрана с разнообразными возможностями.
- **http://www.bradassoc.co.uk** — графические среды программирования и библиотеки для ОС MS Windows, Unix и OpenVMS.
- **http://www.netlib.org** — репозиторий свободно распространяемых текстов программ и библиотек.
- **http://www.fortranlib.com** — библиотека Фортрана.
- **http://www.nag.co.uk/nagware/Examples.asp** — репозиторий программного обеспечения на Фортране.
- **http://math.nist.gov/StopWatch** — измеритель быстродействия программ StopWatch.

Дополнительную информацию и обновляемый набор ссылок на ресурсы Интернета можно найти на персональной странице одного из авторов: **http://mph.phys.spbu.ru/~nemnugin**.

# Список литературы


1. Бартеньев О. В. Современный FORTRAN. — М.: Диалог-МИФИ, 1998.
2. Бартеньев О. В. FORTRAN для студентов. — М.: Диалог-МИФИ, 1999.
3. Бартеньев О. В. Фортран для профессионалов. Математическая библиотека Меткалф М., Рид Дж. Описание языка программирования Фортран 90. — М.: Мир, 1995.
4. Бен-Ари М. Языки программирования. Практический сравнительный анализ. — М.: Мир, 2000.
5. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб.: БХВ, 2002.
6. Голуб Дж., Ван Лоун Ч. Матричные вычисления. — М.: Мир, 1999.
7. Меткалф М. Оптимизация в Фортране. — М.: Мир, 1985.
8. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. — СПб.: БХВ, 2002.
9. Немнюгин С., Чаунин М., Комолкин А. Эффективная работа: UNIX. — СПб.: Питер, 2001.
10. Себеста Р. У. Основные концепции языков программирования. — М.: Вильямс, 2001.
11. Akin J. E. Object Oriented Programming in FORTRAN 90/95. — Cambridge University Press, 2003.
12. Bronson G. J., Walter K. Modern FORTRAN 77/90/2000. — Scott/Jones Publishing, 1999.
13. Brooks. Problem solving with Fortran 90: for scientists and engineers. — Springer, 1997.
14. Chivers I. D., Sleighthome J. Introducing FORTRAN 95: With Coverage of ISO TR 15580 and TR 15581. — Springer-Verlag, 2000.
15. Etzel M., Dickinson K. Digital Visual Fortran Programmer's Guide. — Butterworth-Heinemann, 1999.
16. Hahn B. D. Fortran 90 for Scientists and Engineers. — Arnold, 1997.

17. Koonin S. E. Computational Physics: FORTRAN Version. — Addison-Wesley, 2000.
18. Kupferschmid M. Classical FORTRAN: Programming for Engineering and Scientific Applications. — Marcel Dekker, 2002.
19. Lawrence N. COMPAQ Visual FORTRAN: A Guide to Creating Windows Applications. — Butterworth-Heinemann, 2001.
20. Metcalf M., Reid J. Fortran 90/95 Explained (2nd edition). — Oxford University Press, 1999.
21. Wille D. R. Advanced Scientific Fortran. — John Wiley and Sons, 1995.
22. ANSI: Programming Language Fortran, X3.9-1978, American National Standard.
23. ANSI: Programming Language Fortran 90, X3.198-1992, American National Standard.
24. ISML. — М.: Диалог-МИФИ, 2001.
25. ISO/IEC 1539:1991, Information Technology — Programming Languages — Fortran, Second Edition, 1991-07-01, ISO Publications Department.
26. ISO/IEC 1539-2:1994, Information Technology — Programming Languages — Fortran — Part 2: Varying length characater strings, ISO Publications Department.
27. ISO/IEC 1539-1:1997 Fortran, Part 1, Base Language, ISO Publications Department.
28. ISO/IEC 1539-2:2000 Fortran, Part 2, Varying length character strings, ISO Publications Department.
29. ISO/IEC 1539-3:1999 Fortran, Part 3, Conditional compilation, ISO Publications Department.
30. ISO/IEC TR 15580:2001 Fortran, Floating-point exception handling, ISO Publications Department.
31. ISO/IEC TR 15581:2001 Fortran, Enhanced data type facilities, ISO Publications Department.

# Предметный указатель


## A

ABS 110, 251
Absoft компиляторы 307
ACCEPT 205
ACCESS 207
ACHAR 259
ACOS 114, 251
ACTION 207
ADJUSTL 260
ADVANCE 206, 207, 221
AIMAG 104, 251, 253
AINT 110, 251
ALL 257
ALLOCATABLE 21, 110
ALLOCATE 193, 326
ALLOCATED 247
ANINT 251
ANY 257
APOSTROPHE' 207
APPEND' 207
ASIN 114, 252
ASIS 207
ASSIGN 206
ASSIGNMENT 169
ASSOCIATED 244, 341
ASSOCIATEVARIABLE 211
ATAN 114, 252
ATAN2 252
AUTOMATIC 395

## B

BACKSPACE 199, 202, 234
Big-endian формат хранения данных 289
BINARY 200, 207
BIT_SIZE 112, 261
BLANK 207
BLAS, библиотека 457
BLOCK DATA 107, 109, 396
BLOCKSIZE 211, 240
BTEST 261
BUFFERCOUNT 211
BUFFERED 211
BYTE 100

## C

CALL 157
CARRIAGECONTROL 211
CASE 141
CEILING 252
CHAR 259
CHARACTER 19, 23
CLOSE 202, 203, 232
CMPLX 104, 254
COMMON 109, 176, 396
Compaq Visual Fortran 101, 104, 199, 200, 211, 231, 235, 239, 267, 273, 304

COMPLEX 19, 46, 101
CONJG 110, 252
CONTAINS 155
CONTINUE 149
CONVERT 211
COS 114, 252
COSH 114, 252
COUNT 257
CSHIFT 255
CYCLE 147

## D

DATA 107, 108, 110, 395, 396
DATE_AND_TIME 250
DBLE 104
Dbx-отладчик 309
DEALLOCATE 194, 326
DEFAULTFILE 211
DELETE 232
DELIM 207, 221
Digital Visual Fortran 304
DIGITS 123, 246
DIM 252
DIMENSION 21, 24, 110
DIRECT 207
DISP 211
DISPOSE 211
DO 145
DOT_PRODUCT 258
DOUBLE PRECISION 44, 46
DPROD 110, 252

## E

ELSE 139
ELSE IF 139
ELSE WHERE 191
END 156, 206, 207
END DO 145
END FORALL 191
END WHERE 190
ENDFILE 198, 235
EOR 206, 207
EOSHIFT 256
EPSILON 123, 246
EQUIVALENCE 110, 177, 395
ERR 206, 207, 231
Escape-последовательность 396
Essential Lahey Fortran 90
  компилятор 306
EXIST 207
EXIT 34, 146
EXP 114, 252
EXPONENT 125, 250
EXTERNAL 21, 110

## F

FALSE. 20
F-система программирования 267
f2c 392
f77 395
FILE 207, 231
FLOAT 104
FLOOR 110, 252
FMT 205, 207
FORALL 191
FORM 200, 207
FORMAT 205, 222, 239
FORMATTED 200, 207
Fortran Company 267
FortranPlus компилятор 309
FRACTION 125, 251
FUNCTION 17, 157

## G

Gdb-отладчик 315
GETARG 396
GNU 77, компилятор 295
GO TO 144

## H

HPF 267
HUGE 123, 246

## I

IACHAR 259
IAND 261
IARGC 396
IBCLR 261
IBITS 262
IBSET 262
IEEE 117
IEEE 754, 102
IEOR 262
IF 137
IF...END IF конструкция 137
IFTHENEND IF 29
IMPLICIT 18, 20, 57, 110, 395
IMPLICIT NONE 60, 328
IN 163
INCLUDE 170, 395
INDEX 260
INOUT 163
INQUIRE 202, 205, 207, 235
INT 104, 254
INTEGER 19, 100
Intel Itanium 273
INTENT 21, 110, 163, 328
INTERFACE 160, 327, 335, 346
INTRINSIC 21, 110, 243
IOR 262
IOSTAT 206, 207, 231
ISHFT 112
ISHFTC 112
ISHIFT 262
ISHIFTC 262

## K

KEEP 232
KIND 104, 122, 245

## L

Lahey/Fujitsu Fortran 95 компилятор 306
LBOUND 248
LEN 247
LEN_TRIM 260
LGE 259
LGT 259
libF77, библиотека 396
Little-endian, формат хранения данных 289
LLE 259
LLT 260
LOG 114, 252
LOG10 114, 252
LOGICAL 19, 100, 254

## M

MATMUL 258
MAX 253
MAXEXPONENT 124, 246
MAXLOC 248
MAXREC 211
MAXVAL 248
MERGE 256
Microsoft Fortran Powerstation 304
MIN 253
MINEXPONENT 124, 246
MINLOC 249
MINVAL 249
MOD 32, 110, 253
MODE 211
MODULO 110, 253
MVBITS 262

## N

NAG 267
NAME 211
NAMED 207
NAMELIST 201, 206, 207, 219, 222, 234, 236, 395, 396
NaN 95, 119
NEAREST 125, 251
NEW 207, 232
NEXTREC 207
NINT 254

NML 207
NOT 262
NULL 207
NULLIFY 194
NUMBER 207

## O

OLD 207, 232
OMP_DESTROY_LOCK 381
OMP_DYNAMIC 383
OMP_GET_DYNAMIC 380
OMP_GET_MAX_THREADS 380
OMP_GET_NESTED 381
OMP_GET_NUM_PROCS 379
OMP_GET_NUM_THREADS 380
OMP_GET_THREAD_NUM 380
OMP_GET_WTICK 382
OMP_GET_WTIME 382
OMP_IN_PARALLEL 380
OMP_INIT_LOCK 381
OMP_NESTED 383
OMP_NUM_THREADS 382
OMP_SCHEDULE 383
OMP_SET_DYNAMIC 380
OMP_SET_LOCK 381
OMP_SET_NESTED 381
OMP_SET_NUM_THREADS 380
OMP_TEST_LOCK 382
OMP_UNSET_LOCK 382
ONLY 174
OPEN 199, 200, 202, 203, 207, 221, 230, 235, 396
OPENED 207
OpenMP 273, 293, 366
OPTIONAL 21, 110, 328
ORGANIZATION 211
OUT 163

## P

PACK 256
PAD 206, 207, 219
PARAMETER 18, 21, 110
POINTER 21, 25, 110
POSITION 207
PRECISION 124, 246
PRESENT 167, 245, 328
PRINT 205, 221, 222
PRIVATE 21, 110, 237, 334
PRODUCT 258
PROGRAM 17, 155
PUBLIC 21, 110, 237, 335

## R

RADIX 124, 246
RANDOM_NUMBER 255
RANDOM_SEED 255
RANGE 124, 246
READ 198, 202, 204, 205, 206, 207, 214, 220, 222
READONLY 211
READWRITE 207
REAL 19, 44, 101, 104, 254
REC 203, 207
RECL 199, 207, 235, 240
RECORDSIZE 211
RECORDTYPE 211
RECURSIVE 171
REPEAT 260
REPLACE 232
RESHAPE 256
RESULT 172
RETURN 157
REWIND 199, 202, 207, 234
RRSPACING 126, 251

## S

SAVE 21, 109, 110, 326
SCALE 126, 251
SCAN 260
SCRATCH 207, 232
SELECT CASE 31
SELECT-конструкция 137
SELECTED_INT_KIND 122, 247
SELECTED_INT_KIND(n) 43

SELECTED_REAL_KIND 121, 247
SELECTED_REAL_KIND(PR) 45
SEQUENCE 178
SEQUENTIAL 207
SET_EXPONENT 126, 251
SHAPE 182, 248
SHARE 211
SHARED 211
SIGN 110, 253
SIN 114, 253
SINH 114, 253
SIZE 182, 206, 207, 248, 326
SPACING 126, 251
SPREAD 257
SQRT 110, 253
STAT 193
STATIC 395
STATUS 207, 231, 232
STOP 143, 156
SUBROUTINE 17, 157
SUM 258
SYSTEM_CLOCK 250

## T

TAN 114, 253
TANH 114, 253
TARGET 21, 110
THEN 138
TINY 125, 247
TRANSFER 104, 254
TRANSPOSE 259
TRIM 260
TRUE 20
TYPE 24, 205

## U

UBOUND 248
UNDEFINED 395
UNFORMATTED 200, 207
UNIT 205, 207, 231
UNKNOWN 207, 232
UNPACK 257
USE 68, 110, 170, 173
USEROPEN 211
USE-ассоциирование 109

## V

VERIFY 260
VS.NET 273

## W

WHERE 190
WHILE 34, 149
WRITE 198, 202, 204, 205, 207, 214, 219, 222

---

## A

Абстрактная вычислительная машина 7
Абстракция:
  данных 318
  процедурная 318
Автоматические объекты 70, 183
Алгоритм 7
АЛУ 121
Алфавит:
  Фортрана 9
  языка 5
Арифметико-логическое устройство 121
  присваивание 99
Ассемблеры 7
Ассоциированные объекты 172
Атрибуты 60, 64
  ALLOCATABLE 56, 67
  DIMENSION 51, 66
  EXTERNAL 71
  INTENT 70
  INTRINSIC 71
  OPTIONAL 70
  PARAMETER 65

POINTER 54, 67
PRIVATE 63, 68
PUBLIC 63, 68
SAVE 69
TARGET 67

## Б

Библиотека:
динамическая 267
статическая 267
Блок интерфейсный 97
Блочное распределение 355
Буквальные константы 41, 42
вещественного типа 44
комплексного типа 47
символьного типа 48
Бэкус Джон 2

## В

Ввод-вывод:
для внутренних файлов 205
логическое устройство 197
непродвигающий 203, 206, 221
неформатный 205, 217, 220
последовательный 207
последовательный доступ 204, 205
преобразование данных 222
продвигающий 203, 206
простой список 214
прямой доступ 204
стандартное устройство 197, 203
статус завершения 206
управляемый NAMELIST-списком 205
управляемый списком 204
физическое устройство 197
форматный 204, 207, 217
циклический список 217
Векторный индекс 53, 186
Ветвления 137
Вещественные константы 45
Вещественный тип 41, 44
Внешние подпрограммы 156
Внутреннее представление числа 42
Внутренние подпрограммы 158
Встроенные операции 75
типы данных 41
Выравнивание порядков 118
Выражение 27, 75
арифметическое 95
выражение-массив 96
логическое 29
однородное арифметическое 100
скалярное 96
смешанное 100, 101
Выражения с массивами 89

## Г

Главная программа 16, 155
Глобальные объекты 172
Границы массива 51, 182
Группа исполнения 366

## Д

Двойная точность 117
Дескриптор:
A 226
EN 226
ES 226
L 226
масштабирования 224
обобщенного преобразования 226
позиционирования 227
преобразования данных 222
преобразования строк 222
управляющий 222, 227
Деструктор 324
Диапазон допустимых значений 42
Динамическая:
диспетчеризация 342
память 54, 55
массивы 67, 192

Директива:
ATOMIC 376
BARRIER 376
CRITICAL 376
DISTRIBUTE 355
DO 369
FLUSH 377
INHERIT 363
MASTER 375
ORDERED 379
PARALLEL 368
PROCESSORS 355
SECTIONS 370
SINGLE 371
TEMPLATE 357
THREADPRIVATE 373
WORKSHARE 371
выравнивания 357
распределения данных 354
Длина символьной константы 53

## Е

Единица трансляции 391

## З

Задаваемые операции 75, 86, 169
Запись 198
"конец файла" 198, 207
неформатная 198
потоковая 199
сегментированная 199
форматная 198
Значащие цифры 44

## И

Извлечение
элемента структуры 50
Именованные константы 41, 64
Императивные языки 8
Индекс 24
Инициализация
переменных 72, 107
Инициализирующее
выражение 60, 66
Инкапсуляция 319, 321, 325
Инструкция:
HPF FORALL 359
INDEPENDENT 359
NEW 361
PURE 360
Интероперабельность 430
Интерфейс 160
защищенный 322
общий 322
Интерфейсный блок 97
Информационные сообщения 267
Исполняемый файл 266

## К

Класс 319, 335
метод 335
поле 335
Ключевые параметры 166
Коллапс измерения 357
Комментарий 26
Компилятор Фортрана фирмы
Intel 273
Компиляция 266, 275
Комплексный тип 41, 46
Компоновка 266, 275
Конвертеры 392
Константа 6
буквальная 6, 22
именованная 6, 22
комплексная 221
символьная 221
Константные выражения 65
Константы:
восьмеричные 395
двоичные 395
шестнадцатеричные 395
Конструктор 323
массивов 184
структур 85

## Л

Лексема 5
Логика булева 20
Логические:
  массивы 189
  операции 83
  языки 8
  тип 47
Локальные объекты 172
Локеры 381

## М

Мантисса 117, 118
Массив 24, 50, 181
  автоматический 325
  динамический 326
  нулевой длины 181, 182
  сечение 109
  ссылок 189
Массивоподобные объекты 53
Машинная арифметика
  ограничения 121
Машинные коды 7
Метка 25
Метод Ньютона 127
Модель данных 115
Модули 155, 170
Модульные подпрограммы 168

## Н

Назначение формального
  параметра 70
Наследование 319, 322
  композиция классов 336
  множественное 351
  подтипизация 324, 340
Необязательные:
  параметры 166
  формальные параметры 70
Неопределенные переменные 76
Неформатная запись 198
Неявное определение типа 41, 57
  связывание 20
  цикл 73, 150
Нормализация 117

## О

Область:
  видимости 172
  параллелизации 366
Обработка препроцессором 274
Объект 319
  атрибуты 320
  виртуальный метод 323
  методы 321
Объектная модель 320
Объектно-ориентированное
  программирование 8, 317, 318
  класс 319
  объект 319
Объектно-ориентированные
  языки 8
ООП 8
Операнд 28, 95
Оператор 7
  ALLOCATE 56
  DATA 72
  ввода-вывода неформатного 198
  выбора 31
  описания 41, 57
  передачи данных 203
  присваивания 27, 77, 99, 107
  управляющий 7
  условный 29
  цикла 33
Операции:
  арифметические 390
  бинарная 96, 100
  ввода-вывода 197
  встроенная 96, 97
  задаваемая 96, 97
  многоместная 96
  одноместная 96
  перегружаемая 97

перегрузка 96, 97
приоритет 27, 98
ранг 100
тип результата 101
унарная 96
Описание производного типа 50
Определение типа по имени 20
Оптимизация 128
межпроцедурная 292
Отладка 309

## П

Парадигма программирования 317
Параллелизм 391
Параллельные расширения Фортрана 353
Параметр разновидности 42
Параметр-адресат 164
Параметр-подпрограмма 164
Параметры подпрограмм 161
Паскаль 385
Перегрузка 390
Перегрузка операции 322
Переменная 6
автоматическая 395
адрес 6
глобальная 6
значение 6
имя 6
инициализация 107
класс хранения 395
локальная 6
область видимости 6
статическая 395
тип 6
Переменные неинициализированные 389
Перенос строки 16
Подвыражение 99
Подобъекты массивов 186
Подпрограмма 16, 155, 390
Подразумеваемая форма массива 183
Подтипизация 324
Позднее связывание 323
Позиционные параметры 166
Полиморфизм 319, 324
ad hoc 331
динамический 342
Порядок операторов 179
Порядок следования элементов массива 52
Предложение описания 7, 107
Предупреждения 266
Прикрепление ссылки 68
Приоритет встроенных операций 78
Программа 7
Программа главная 16
Программирование модульное 318
Программирование смешанное 293, 402
Программный компонент 16
Производные типы данных 41, 49
Производный тип:
компоненты 63
определение 62
Простая точность 117
Процедура 16, 157
Процедурные языки 8

## Р

Раздел:
операторов 18
описаний 17
Размер массива 51, 181
Разновидности вещественного типа 45
Разновидность:
символьного типа 48
комплексного типа 47
типа 41
определение 43
Ранг 100
массива 51, 182
Раннее связывание 323

Редактирование связей 275
Рекурсия 171, 390, 396
Родовое имя 167

## С

Сборка 275
Свободный формат 11
Селектор оператора CASE 142
Семантика 97
Сечения массивов 53, 186
Символьные массивы 53
Символьный тип 48
Синтаксис 7
Скалярное числовое
присваивание 80
Скалярные операции отношения 81
Скалярные символьные
выражения 84
Скалярные числовые
выражения 78
Скрытый бит 118
Слово:
зарезервированное 6
ключевое 6
ключевое Фортрана 9
специальное 6
специальное Фортрана 9
Совместимые массивы 89, 182
Совмещение в памяти 176
Совмещение процедур 167
Сообщения об ошибках 266
Сортировка символов 49
Состояние неопределенности
ссылки 77
Спецификатор:
ветвления 206
ключевая форма 204
неключевая форма 204
смешанная форма 204
счетчик символов 206
управляющий 204
Спецификатор устройства 202
Ссылка 25
Ссылки (указатели) 54
Ссылки в выражениях 92
Ссылки-параметры 164
Стандарт IEEE 754, 102
Стандартная разновидность
целых 42
Статический полиморфизм 96
Строка продолжения 12
Строки 53
Структура программы 155
Схема Горнера 130

## Т

Тип:
базовый 24
вещественный 116
преобразование неявное 104
производный 24, 97, 109, 217
функции преобразования 104
целый 115
Тип выражения 79
Тип данных 19, 41
абстрактный 321
производный 318

## У

Указатели 389
Указатель 25, 54, 217
Управляющие символы 49
Устройство логическое 202

## Ф

Файл 197
атрибуты 197
внешний 107, 201
внутренний 201, 218
временный 232
двоичный 200
доступ по ключу 199
закрытие 232
*(окончание рубрики см. на стр. 481)*

Файл *(окончание)*
запись 198
исполняемый 266
неформатный 201, 239
объектный 266
открытие 230
отсоединение 232
последовательный доступ 199, 200, 207
прямой доступ 199, 200, 207, 220, 234
соединение 202, 230
указатель 198, 203
форматный 199, 200
Файловый указатель 198, 203, 207, 231, 234
Фактические параметры 161
Фиксированный формат 11
Форма массива 51, 181, 182
Формальные параметры 70, 161
Формат:
внутреннего представления 117
времени исполнения 228
динамический 206
записи программы 11
с плавающей точкой 23
спецификатор 198
Форматирование 222
Форматная запись 198
Форматное выражение 239
Фортран 2003 409
Функции:
битовые 112
встроенные математические 110
математические элементные 114
преобразования типов 104
справочные 122
Функциональные языки 8
Функция 16, 157
родовая 330

## Ц

Целое скалярное выражение 52
Целый тип 41
Цикл 33, 145
инвариант цикла 134
неявный 109
объединение 133
оптимизация 131
развертка 132
расщепление 133
с предусловием 34
со счетчиком 33
Циклическое распределение 356

## Ч

Частный указатель 187
Число нормализованное 118
Числовые типы 41

## Э

Экстент массива 51, 66
Элементные операции 184
Элементы массива 51, 66
индексы 52
обращение 52
расположение 51

## Я

Явное связывание 20
Языки:
высокого уровня 8
низкого уровня 7