Министерство науки и высшего образования Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Физико-механический институт

Высшая школа прикладной математики и вычислительной физики

А. Г. Абрамов М. А. Засимова М. Е. Фролов

ЦИФРОВЫЕ ТЕХНОЛОГИИ В ПРОФЕССИОНАЛЬНОЙ ДЕЯТЕЛЬНОСТИ

ЯЗЫК ПРОГРАММИРОВАНИЯ ФОРТРАН ДЛЯ НАУЧНЫХ И ИНЖЕНЕРНЫХ ВЫЧИСЛЕНИЙ

Ч а с т ь 1

Учебное пособие

ПОЛИТЕХ-ПРЕСС
Санкт-Петербургский
политехнический университет
Петра Великого

Санкт-Петербург
2024

УДК 004.43(519.6)
ББК 16:32.973.2:22.25я73
А16

Р е ц е н з е н т ы:
Доктор технических наук, профессор, заведующий кафедрой
высшей математики Тверского государственного
технического университета *В. Д. Горячев*
Кандидат физико-математических наук, доцент, доцент высшей школы
программной инженерии СПбПУ *С. П. Воскобойников*

Абрамов А. Г. **Цифровые технологии в профессиональной деятельности. Язык программирования Фортран для научных и инженерных вычислений** : учеб. пособие / А. Г. Абрамов, М. А. Засимова, М. Е. Фролов. В 2 ч. Ч. 1. – СПб. : ПОЛИТЕХ-ПРЕСС, 2024. – 154 с.

Учебное пособие соответствует содержанию рабочей программы дисциплины «Цифровые технологии в профессиональной деятельности» и посвящено программированию на языке Фортран с акцентом на научные и инженерные вычисления. В главе 1 рассматриваются некоторые аспекты разработки программного обеспечения, используемые на практике инструменты, вопросы развертывания среды и инструментов разработки. Глава 2 содержит общие сведения об алгоритмах, языках и парадигмах программирования. Остальные главы нацелены на изучение базовых принципов и методик программирования на Фортране, особенностей синтаксиса и семантики языка, включая условные операторы, циклы, операции ввода/вывода, введение в массивы и процедуры. Приводятся примеры исходных кодов программ вычислительной направленности, даются варианты заданий для компьютерного практикума по изучаемым темам.

Учебное пособие предназначено для студентов, обучающихся по направлениям подготовки бакалавров «Прикладные математика и физика» и «Прикладная механика», также может быть полезным и для других направлений подготовки и специальностей при обучении программированию на языках высокого уровня и вычислительным технологиям.

Табл. 19. Ил. 14. Библиогр.: 32 назв.

Печатается по решению
Совета по издательской деятельности Ученого совета
Санкт-Петербургского политехнического университета Петра Великого.

ISBN 978-5-7422-8586-1
doi:10.18720/SPBPU/2/id24-76

Ministry of Science and Higher Education of the Russian Federation

PETER THE GREAT ST. PETERSBURG
POLYTECHNIC UNIVERSITY

Institute of Physics and Mechanics
Graduate School of Applied Mathematics and Computational Physics

A. G. Abramov M. A. Zasimova M. E. Frolov

DIGITAL TECHNOLOGIES IN PROFESSIONAL ACTIVITIES

FORTRAN PROGRAMMING LANGUAGE FOR SCIENTIFIC AND ENGINEERING COMPUTING

Part 1

Training manual

POLYTECH PRESS
Peter the Great
St.Petersburg Polytechnic
University

Saint Petersburg
2024

R e v i e w e r s:
Doctor of Engineering, professor, Head of the Department of Mathematics
at Tver State Technical University *V. D. Goryachev*
Candidate (PhD) of Science in Physics and Mathematics, associate professor
of the Graduate School of Software Engineering, SPbPU *S. P. Voskoboinikov*

Abramov A. G. **Digital technologies in professional activities. Fortran programming language for scientific and engineering computing**: training manual / A. G. Abramov, M. A. Zasimova, M. E. Frolov. In 2 parts. Part 1. – St. Petersburg: POLYTECH-PRESS, 2024. – 154 p.

The training manual corresponds to the content of the discipline «Digital technologies in professional activities». It is devoted to programming in Fortran with an emphasis on scientific and engineering calculations. Chapter 1 deals with some aspects of software development, hands-on tools, issues of deployment environment, and development tools. Chapter 2 covers the overview of algorithms, languages, and paradigms for programming. Other chapters are aimed at studying basic principles and techniques of Fortran programming, specifics of the language syntax and semantics, including conditional statements, loops, I/O operations, introduction to arrays and procedures. The training manual provides examples of source codes for computational programs, options of assignments on relevant matters for the computer training.

It is intended for students studying in the Bachelor's majors «Applied mathematics and physics» and «Applied mechanics», as well as for other programs and majors in the field of high-level language programming, and computer science.

Tables 19. Figures 14. References: 32 titles.

Printed by the Publishing Council
of the Peter the Great St. Petersburg polytechnic university Academic Council.

ISBN 978-5-7422-8586-1
doi:10.18720/SPBPU/2/id24-76

ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ

АЛУ – арифметико-логическое устройство
БОК – блок операторов и конструкций
ИТ – информационные технологии
ЛВ – логическое выражение
RAM – Random Access Memory (оперативное запоминающее устройство)
ОС – операционная система
ПО – программное обеспечение
СЛАУ – система линейных алгебраических уравнений
УВВ – устройство ввода/вывода
ЦПУ – центральное процессорное устройство
CPU – Central Processing Unit
ЯВУ – язык программирования высокого уровня
API – Application Programming Interface (интерфейс прикладного программирования)
ASCII – American Standard Code for Information Interchange
BTS – Bug Tracking System (система отслеживания ошибок)
DDD – GNU Data Display Debugger (отладчик отображения данных GNU)
GCC – GNU Compiler Collection (коллекция компиляторов GNU)
GDB – GNU Project Debugger (отладчик проекта GNU)
GNU – GNU's Not Unix
HPF – High Performance Fortran (высокопроизводительный Фортран)
IDE – Integrated Development Environment (интегрированная среда разработки)
ISO – International Organization for Standardization (Международная организация по стандартизации)
MPI – Message Passing Interface (интерфейс передачи сообщений)
PMS – Project Management System (система управления проектами)
SDK – Software Development Kit (комплект для разработки ПО)
UTF – Unicode Transformation Format (формат преобразования Юникода)
VCS – Version Control System (система контроля версий)
WSL – Windows Subsystem for Linux (подсистема Windows для Linux)

ВВЕДЕНИЕ

Вычислительные системы и технологии глубоко и необратимо проникают в материально-производственную и социальную сферы. Без компьютеров в широком понимании этого термина, а также без эксплуатирующих их и актуальные информационные и технологические достижения киберфизических систем, которые интенсивно развиваются и внедряются в производственные процессы, трудно представить в настоящее время эффективное функционирование предприятий самых разных секторов экономики, равно как современный быт и социальные отношения.

Цифровая трансформация, массово затрагивающая как отдельные организации, так и отраслевые сегменты, сопровождается широким внедрением цифровых технологий и инновационных решений, преобразованием бизнес-моделей и организационных процессов, стратегическими и структурными изменениями. Целью этих изменений является повышение эффективности, производительности и конкурентоспособности, оптимизация деятельности, адаптация к быстроменяющейся технологической среде и удовлетворение постоянно растущих требований потребителей.

Рынок труда формирует и постоянно повышает уровень потребности в квалифицированных кадрах для отрасли информационных технологий (ИТ), которые призваны готовить в первую очередь образовательные организации высшего и дополнительного профессионального образования. В индустрии промышленного программирования, как одном из важнейших и востребованных направлений ИТ, особое место на протяжении всей истории развития программирования отводится проблематике, ориентированной на решение широкого класса научных и инженерных задач из самых разных областей.

В процессе постоянной эволюции появляются специалисты-вычислители, сфера профессиональных интересов которых связана с методами и алгоритмами, базирующимися на вычислительных подходах разной сложности и ресурсоемкости. Такие специалисты остро нуждаются в совершенствовании существующих и появлении новых программных инструментов и проблемно-ориентированных пакетов.

Вычислители глубоко погружены в свои области знаний (физика, математика, химия, биология и др.) и обычно не являются профессиональными программистами. Это объясняет их особую заинтересованность в технологиях и языках программирования, которые относительно несложны в освоении и позволяют эффективно и результативно решать задачи, базирующиеся на вычислениях, автоматизировать выполнение рутинных операций.

Широкий спектр развитых высокоуровневых языков программирования создает предпосылки для обоснованного выбора языка, наиболее подходящего для конкретной области применения. В контексте специализированных языков, нацеленных на научные и инженерные вычисления, обычно упоминают C, C++, Фортран, Python, Julia, R, MATLAB, Scala и некоторые др.

В этом наборе наиболее богатая история у языка Фортран, первая версия которого появилась в 1957 г. Языку, несмотря на свой «преклонный» по меркам ИТ возраст, удается сохранять своих приверженцев и актуальность с перспективами дальнейшего развития. Основные причины этого связаны с его выраженной целевой направленностью, высокой производительностью, мультипарадигменностью, наличием большого количества разработанных библиотек и прикладных пакетов, поддержкой современных технологий высокопроизводительных вычислений и развитием встроенных средств организации параллелизма.

Несмотря на приведенные доводы, выбор Фортрана как основного предмета изучения данного пособия, возможно, представляется неочевидным, особенно на фоне активного развития языков и технологий, которые более эффективны в таких критических научно-технологических приложениях, как искусственный интеллект и машинное обучение, генерация и работа с большими данными, технологии

блокчейна, Интернета вещей. Вместе с тем Фортран по-прежнему является первым кандидатом для изучения для многих специалистов, ориентированных на классические вычислительные методы и подходы, что в определенной степени оправдывает такой выбор.

Тематика учебного пособия обусловлена содержанием дисциплины «Цифровые технологии в профессиональной деятельности», преподаваемой в Санкт-Петербургском политехническом университете Петра Великого (СПбПУ) в рамках учебного плана по направлениям подготовки бакалавров 03.03.01 «Прикладные математика и физика» и 15.03.03 «Прикладная механика».

Учебное пособие состоит из двух частей, первая из которых посвящена основам программирования на языке Фортран, изложению полезных практик его использования для научных и инженерных вычислений.

В гл. 1 пособия в компактной форме обсуждаются аппаратное устройство компьютера и основные виды программного обеспечения, некоторые аспекты его разработки, используемые на практике инструменты, вопросы развертывания среды и инструментов разработки на языке Фортран. Глава 2 содержит общие сведения об алгоритмах, языках и парадигмах программирования, а также краткий исторический обзор развития Фортрана.

Остальные главы нацелены на изучение базовых принципов и методик программирования на Фортране, особенностей синтаксиса и семантики языка, включая объекты данных, выражения и операции, встроенные функции, условные операторы, циклы, операции ввода/вывода, введение в массивы и процедуры. Приводятся многочисленные примеры исходных кодов программ как демонстрационного характера, так и реализующих востребованные на практике алгоритмы вычислительной направленности. Даются варианты заданий для компьютерного практикума с разделением на математические и физические задачи, включая механику сплошных сред.

Учебное пособие завершается приложениями, в которые вынесена информация справочного характера, в том числе замечания к разработке программ, полезные ключи компиляторов, описание встроенных функций Фортрана.

В процессе подготовки учебного пособия использовались общепризнанные учебники, перечисленные в списке основной литературы. Кроме того, в списках дополнительной литературы приведены и другие полезные источники информации как по основной теме учебного пособия, так и по информационным технологиям, математике, механике, линейной алгебре, численным методам.

Учебное пособие предназначено для студентов физико-математических направлений подготовки, а также отдельных направлений в рамках укрупненных групп специальностей ИТ-профиля. В более узком понимании учебное пособие ориентировано на дисциплины направлений подготовки бакалавриата 03.03.01 «Прикладные математика и физика», 15.03.03 «Прикладная механика» и 01.03.02 «Прикладная математика и информатика», в которых изучаются математические и численные методы и предполагается кодирование соответствующих алгоритмов.

Вторая часть учебного пособия будет содержать описание расширенных возможностей современных стандартов языка Фортран, включая элементы объектно-ориентированного программирования, указатели и динамические структуры данных, средства организации параллельных вычислений, совместимости с языком C. Также будут рассмотрены такие вопросы, как производные типы данных, модули, расширенные возможности работы с встроенными типами данных, массивами, функциями и подпрограммами, дополнительные возможности ввода/вывода и ряд др.

Глава 1
НЕКОТОРЫЕ ВОПРОСЫ И ИНСТРУМЕНТЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Кратко об устройстве и работе компьютера

Компьютер (*computer* – «вычислитель») – функциональное устройство, предназначенное для автоматизации процесса обработки информации (выполнения вычислений) и получения результатов в понятной человеку или машине форме. Аппаратные

компоненты компьютера работают под управлением определяющих их действия программ, представляющих собой описание последовательностей выполнения требуемых операций.

Ключевые аппаратные компоненты (функциональные блоки) компьютера – это процессор, основная (оперативная) память, внешняя память и устройства ввода/вывода (рис. 1.1). Компоненты, за исключением устройств ввода/вывода, размещаются на *материнской плате*, которая служит основой электронной схемы компьютера, обеспечивая взаимодействие аппаратных компонентов, и содержит специальные технологические разъемы для установки внутренних и подключения внешних устройств.

Процессор (центральное процессорное устройство, ЦПУ) – главный функциональный компонент компьютера, выполненный в форме кристалла из полупроводникового материала со сверхвысокой плотностью размещения логических элементов (транзисторов), который совершает основные операции по обработке данных и управлению работой других компонентов компьютера. ЦПУ состоит из устройства управления, арифметико-логического устройства (АЛУ) и внутренней памяти (регистров).

Устройство управления контролирует и обеспечивает согласованную работу компонентов компьютера, а АЛУ выполняет арифметические и логические операции с данными. Внутренняя память состоит из регистров, используемых для временного хранения промежуточных результатов вычислений, а также кэш-памяти, обеспечивающей временное хранение данных, которые могут быть запрошены ЦПУ с наибольшей вероятностью.

Устройство управления вырабатывает распределенную последовательность внутренних и внешних управляющих сигналов,

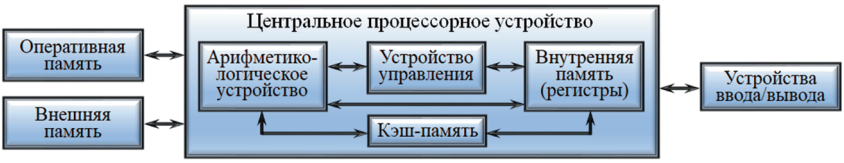

Рис. 1.1. Схема устройства типового компьютера и его основные компоненты

обеспечивающих выборку и выполнение команд компьютерной программы. В процессе выборки производится интерпретация (дешифрация) команды, извлеченной из оперативной или кэш-памяти, и ее сохранение в регистрах памяти. На этапе выполнения формируется микропрограмма в виде последовательности элементарных действий (микрокоманд), которые задают последовательность простейших низкоуровневых операций для АЛУ.

Высокая производительность современных ЦПУ с классической архитектурой обеспечивается наличием нескольких вычислительных ядер на одном процессорном кристалле (передовые модели включают десятки ядер), технологией *многопоточности* (физическое ядро распознается операционной системой компьютера в виде двух логических ядер), а также свойством *суперскалярности* (способность выполнять несколько команд одновременно за счет включения в состав ядра набора одинаковых АЛУ).

Память компьютера в широком понимании этого термина делится на три типа – кэш-память, оперативная память (оперативное запоминающее устройство, ОЗУ) и внешняя память.

Кэш-память обычно делится на несколько уровней и представляет собой быстродействующую память ограниченного объема, служащую промежуточным буфером между регистрами ЦПУ и ОЗУ. Устройство управления просматривает код программы для отбора данных, которые потребуются в ближайшем будущем, и предварительно загружает их из ОЗУ в кэш-память для возможности использования с минимальными задержками. Устройство управления также копирует данные из кэш-памяти обратно в ОЗУ, когда они больше не требуются.

Оперативная память – энергозависимая часть памяти (информация не сохраняется при отключении питания), в которой во время работы компьютера временно хранится выполняемый ЦПУ машинный код (программы) и требуемые программе данные. ОЗУ состоит из полупроводниковых микросхем и соединяется с ЦПУ с помощью компьютерной шины, представляющей собой набор расположенных на материнской плате сигнальных линий. Объем ОЗУ кратно превышает объем кэш-памяти, при этом хранящиеся

в ОЗУ данные извлекаются для обработки на ЦПУ с существенно меньшей скоростью по сравнению с кэш-памятью.

Внешняя память представлена энергонезависимыми запоминающими устройствами – жесткими и/или твердотельными дисками и USB-флеш-накопителями, характеризуемыми значительно меньшей по сравнению с ОЗУ скоростью доступа к данным при на порядки превышающем объеме.

Устройства ввода/вывода служат для соответствующих их названиям операций с данными при взаимодействии с внутренними компонентами компьютера. Наиболее распространенными устройствами ввода являются клавиатура и компьютерная мышь. Среди других используемых типов устройств ввода – сенсорные экраны, сканеры, микрофоны, камеры, геймпады и джойстики. Устройства вывода позволяют использовать данные, хранящиеся на компьютере, и могут быть представлены мониторами, принтерами, аудиодинамиками, мультимедиапроекторами.

1.2. Программное обеспечение компьютера

Общепринятая в информатике и программной инженерии классификация программного обеспечения (ПО) вычислительных систем (компьютеров) по назначению и функциональному признаку подразделяет его на три основных вида – *системное*, *инструментальное* и *прикладное*.

Системное ПО подразделяется на *базовое* и *сервисное*. *Базовое системное ПО* представляет собой операционную систему (ОС) с развитым текстовым и/или графическим интерфейсом пользователя и может поставляться вместе с компьютером в режиме предустановки. В составе ОС различают три основные группы компонентов – ядро, системные библиотеки и оболочку с утилитами.

Сервисное системное ПО включает программы и комплексы, которые расширяют функциональные возможности базового системного ПО, выполняют вспомогательные служебные задачи и организуют более удобный интерфейс работы пользователя при их решении.

Массово используемыми семействами ОС являются проприетарные (являющиеся частной собственностью авторов или правообладателей) ОС Microsoft (MS) Windows и Unix-подобные ОС, которые могут являться как свободными (преимущественно), так и проприетарными.

В настоящее время Unix-подобные ОС, существенная часть которых распространяется бесплатно на основании свободных и открытых лицензий, уступают MS Windows по масштабам использования. Вместе с тем Unix является основным и, по сути, безальтернативным решением для развертывания и эксплуатации компьютерных платформ, обеспечивающих функционирование сервисных решений на базе глобальных телекоммуникационных сетей, а также устанавливается на 100 % высокопроизводительных компьютеров.

Пользовательские (десктопные) версии Unix-подобных ОС для персональных и портативных компьютеров обладают развитым графическим интерфейсом и широким набором инструментального и прикладного ПО. Здесь доминирует богатое по числу дистрибутивов (популярная форма распространения ПО) семейство ОС GNU/Linux (Ubuntu, Debian, CentOS, openSUSE, Fedora, Gentoo и др.). Следует отметить некоторые отечественные дистрибутивы – Астра Линукс, РЕД ОС, ROSA Linux, Альт Линукс и др.

Ведущими мировыми ИТ-вендорами разработаны и успешно развиваются проприетарные Unix-подобные ОС (MacOS, Chrome OS, HP-UX, AIX и некоторые другие), распространяемые с соответствующими компьютерными архитектурами. Доминирующие на рынке платформы для мобильных, портативных, мультимедийных и иных типов устройств Android (Google) и iOS (Apple) представляют собой реализации Unix-подобных ОС.

В отношении возможностей совместного использования на одном компьютере ОС MS Windows и GNU/Linux можно указать на специализированное ПО, позволяющее работать с реализациями Linux в окружении MS Windows (WSL, Cygwin, MinGW), а также на средства виртуализации и контейнеризации (Virtualbox, Docker).

Инструментальное ПО предназначено для использования в ходе проектирования, разработки, сопровождения ПО и представлено

редакторами исходного кода, трансляторами программ (компиляторы, интерпретаторы), отладчиками, профилировщиками, средствами автоматизации сборки, библиотеками подпрограмм, системами управления версиями и др.

Прикладное ПО – отдельные прикладные программы и их пакеты, предназначенные для решения конкретных практических задач и имеющие интерфейсы для непосредственного взаимодействия с человеком.

Основные виды прикладного ПО:

– *общего назначения* (офисные пакеты, файловые менеджеры, средства просмотра файлов разных форматов, текстовые, табличные и графические редакторы, веб-браузеры, почтовые приложения и др.);

– *специального назначения* (системы работы с мультимедиа, настольные издательские системы, системы компьютерной верстки, пакеты виртуализации ОС, лингвистические системы, поисковые системы и др.);

– *профессиональное* (математическое ПО, геоинформационные системы, системы автоматизированного проектирования, системы искусственного интеллекта и машинного обучения, проблемно-ориентированные пакеты прикладных программ и др.).

1.3. Некоторые вопросы разработки программного обеспечения

1.3.1. Основные методологии и этапы разработки программ

Разработка программного обеспечения, под которой в наиболее общем смысле понимается деятельность по созданию нового ПО, является составной частью *программной инженерии* вместе с дисциплинами, отвечающими за функционирование и сопровождение программных продуктов.

Процесс разработки ПО обычно состоит из совокупности подпроцессов (этапов, итераций), формулируемых разными эксплуатируемыми на практике методологиями разработки. К имевшим на определенных этапах эволюции достаточно широкое распространение условно классическим методологиям относятся

каскадная (водопадная), итеративная, спиральная, V-модель и некоторые др.

В настоящее время лидеры отрасли промышленного программирования и многие команды разработки ориентируются на использование подходов, базирующихся на гибкой методологии Agile и применяющих методы управления проектами Lean, Kanban, Scrum, а также их комбинации.

В упрощенном представлении, пригодном для использования в контексте целевой аудитории и задач данного учебного пособия, к основным этапам создания компьютерных программ будем относить следующие: анализ; проектирование; кодирование (программирование); тестирование и отладка; оптимизация; документирование.

В процессе *анализа* исследуется предметная область, описывается исходная информация, определяются условия и эффекты действия программы, по результатам в формальном виде специфицируются наиболее существенные требования к программе.

На этапе *проектирования* продумывается структура будущей программы, задача разбивается на подзадачи (фрагменты), выбираются или разрабатываются алгоритмы и подходящие для их реализации программные единицы, решается вопрос организации хранения, обработки и обмена данными. В частности, разрабатывается интерфейс для программных единиц.

Этап *кодирования* предполагает запись программы на подходящем для ее функциональности языке программирования. Результатом являются программы в исходном тексте и (для компилируемых языков) в готовом к исполнению бинарном виде.

На этапе *тестирования* и *отладки*, который может занимать существенную часть общего времени, выполняется запуск отдельных фрагментов или всей программы в целях обнаружения ошибок (тестирование) и их устранения (отладка). Крайне ценно и востребованно умение качественно тестировать программу (разработать и задействовать полноценный набор тестов), а также максимально быстро локализовать обнаруженные ошибки.

Здесь полезно ввести тесно связанные с тестированием термины «верификация» и «валидация». *Верификация* – процесс

проверки соответствия программы установленным требованиям, правильности программирования и вычислительной реализации заложенных моделей (например, математических). *Валидация* – процесс оценки соответствия разработанной программы ожиданиям пользователей в контексте предполагаемых способов использования; в численном моделировании служит для оценки точности, с которой вычислительная модель воспроизводит явления реального мира.

Следующий этап нацелен на определение областей исходного кода (*bottlenecks* – узких мест), оптимизация которых может существенно ускорить работу программы. Улучшение кода обычно производится с учетом особенностей и средств используемого языка программирования.

Документирование предполагает создание текстовых и графических материалов по использованию ПО для различных категорий – пользователей, программистов, менеджеров (описание возможностей, пользовательского интерфейса, методик использования, принципов разработки и т. д.).

Некоторые замечания к разработке программ на языке Фортран, собранные из нескольких авторитетных литературных источников на основании собственного опыта авторов, приведены в прил. А.

1.3.2. Открытые библиотеки научных и инженерных вычислений

В ходе разработки программ вычислительной направленности могут оказаться весьма полезными готовые, отлаженные в процессе многолетнего использования и оптимизированные подпрограммы, входящие в состав открытых *библиотек научных* и *инженерных вычислений*.

Примерами широко известных и востребованных в профессиональной среде библиотек являются:

– коллекция математического ПО, библиотек, баз данных и публикаций в области научных вычислений Netlib (*https://netlib.org*);

– библиотека (стандарт) линейной алгебры BLAS (*https://netlib.org/blas/*) и ее реализации BLAS (эталонная), ATLAS, OpenBLAS и др.;

– библиотека линейной алгебры LAPACK (*https://netlib.org/lapack/*);

– коммерческая международная математическая библиотека подпрограмм IMSL (*https://help.imsl.com*);

– библиотека оптимизированных математических процедур для научных, инженерных и финансовых приложений Intel oneAPI MKL;

– библиотека математических и статистических подпрограмм общего назначения SLATEC (*https://netlib.org/slatec/*);

– коллекция пакетов и подпрограмм для крупномасштабных научных вычислений HSL (*https://www.hsl.rl.ac.uk*);

– набор структур данных и подпрограмм для параллельного решения научных задач, описываемых дифференциальными уравнениями в частных производных PETSc (*https://petsc.org*);

– набор алгоритмов, вспомогательных технологий и библиотек для решения крупномасштабных инженерных и научных задач в области физики Trilinos (*https://trilinos.github.io*).

Следует отметить, что перечисленные библиотеки либо непосредственно написаны на языке Фортран, либо имеют интерфейсы для возможности взаимодействия с программами и модулями на других языках.

Библиотеки и их отдельные части могут использоваться для внедрения в собственные вычислительные коды, а также в процессе верификации разработанных программ посредством сопоставления результатов с полученными аналогичными методами с помощью тех или иных библиотек.

1.3.3. Тестирование производительности компьютеров

Тестирование производительности компьютерных систем и их компонентов производится в целях сопоставительной оценки эффективности работы разных архитектур и аппаратно-программных комплексов и предполагает задействование готовых специализированных кодов разного уровня сложности, контрольных задач (так называемые *benchmarks*).

Среди наиболее востребованных подборок тестов с открытым исходным кодом (в том числе для параллельных архитектур) можно отметить:

– LINPACK – тесты, основанные на классических методах решения СЛАУ с плотной матрицей (*https://www.netlib.org/benchmark/hpl/*);

– Dhrystone – тесты для оценки производительности целочисленных вычислений (https://github.com/sifive/benchmark-dhrystone);

– Livermore Loops Benchmark – набор тестовых программ, основывающихся на циклах, для оценки производительности многопроцессорных компьютеров (*http://www.netlib.org/benchmark/livermore*);

– NAS Parallel Benchmarks – тесты производительности многопроцессорных компьютеров (*https://www.nas.nasa.gov/software/npb.html*);

– Phoronix Test Suite – кроссплатформенный набор тестов для разных ОС (*https://openbenchmarking.org*).

1.4. Инструменты разработки программного обеспечения

К основным видам используемого в процессе разработки *инструментального ПО* относятся:

– *редактор исходного кода* – текстовый редактор для создания и редактирования исходного кода программ, обладающий возможностями, которые повышают эффективность работы с кодом, такие как подсветка синтаксиса, автодополнение и отступы, контекстная помощь;

– *компилятор* – программа, переводящая исходный код программы, написанной на одном или нескольких ЯВУ, в эквивалентную программу на низкоуровневом языке, близком к машинному коду, или непосредственно на языке машинных команд;

– *интерпретатор* – программа, анализирующая команды или операторы исходного кода и преобразующая их в язык машинных команд всякий раз, когда программа запускается на выполнение;

– *библиотека подпрограмм* – набор готовых подпрограмм или объектов, используемых для разработки;

– *компоновщик* – редактор связей, программа, производящая компоновку ПО, которая принимает на вход один или несколько объектных модулей и собирает из них исполняемый файл;

– *отладчик* – программа, предназначенная для поиска ошибок в ПО, позволяет выполнить трассировку, отслеживать, устанавливать и изменять значения переменных в процессе выполнения, задавать контрольные точки или условия остановки и др.;

– *профилировщик* – программа, осуществляющая анализ производительности ПО с оценкой времени выполнения отдельных фрагментов кода и поиском узких мест.

Комплексные инструментальные программные системы, используемые в промышленной разработке ПО, могут включать в себя:

– набор средств разработки (Software Development Kit, SDK);

– интегрированную среду разработки (Integrated Development Environment, IDE);

– систему управления версиями (Version Control System, VCS);

– систему управления проектами (Project Management System, PMS);

– систему отслеживания ошибок (Bug Tracking System, BTS);

– систему непрерывной интеграции/непрерывной разработки (Continuous Integration/Continuous Delivery, CI/CD).

Набор средств разработки позволяет создавать приложения для определенного пакета программ, ПО базовых средств разработки, аппаратной платформы, компьютерной системы, операционной системы и т. д.

Интегрированная среда разработки – комплекс программных средств, используемый для разработки ПО и обычно включающий редактор исходного кода, компилятор и/или интерпретатор, средства автоматизации сборки и отладки. Развитые IDE могут содержать также средства для интеграции с системами управления версиями и специальные инструменты для упрощения конструирования графического интерфейса пользователя. IDE могут использоваться как для одного конкретного языка программирования, так и для нескольких (примеры IDE: Eclipse, Microsoft Visual Studio, Visual Studio Code, Apache NetBeans, Code::Blocks, ATOM, Geany, Sublime Text).

Система управления версиями – программное решение для облегчения работы с изменяющейся информацией (в частности, с ПО) – позволяет хранить несколько версий продукта, возвращаться к более ранним версиям, определять, кем и когда были сделаны изменения в коде, и т. д. (примеры систем данного типа: Git, Mercurial SCM, Apache Subversion).

Система управления проектами – следующая ступень развития функционала VCS, представляющая собой комплекс совместно используемых для управления крупными проектами разработки ПО профессиональных программных средств, который включает приложения для управления версиями, отслеживания ошибок, рецензирования/ревизии патчей («заплаток»), систему интеграции, средства планирования задач, составления расписания их решения, распределения ресурсов, совместной работы, документирования и администрирования (примеры систем данного типа: GitHub, GitLab, Bitbucket, Jira, Azure DevOps Server, Launchpad).

Система отслеживания ошибок – специализированное ПО, нацеленное на оказание помощи разработчикам в учете и контроле найденных в программах ошибок и неполадок, пожеланий и запросов пользователей, позволяющее следить за процессами устранения ошибок и выполнения пожеланий.

Система непрерывной интеграции – профессиональное ПО, позволяющее реализовывать регулярное (вплоть до нескольких раз в день, в зависимости от проекта) слияние рабочих копий совместно разрабатываемых программ в основную единую ветвь разработки и выполнять автоматизированные сборки проекта для скорейшего обнаружения потенциальных дефектов и решения возможных проблем интеграции (примеры популярных систем данного типа: GitHub, GitLab, Azure DevOps Server, CircleCI).

Следует отметить, что для большинства современных языков программирования и IDE (в том числе Eclipse, Visual Studio Code, Microsoft Visual Studio, ATOM) доступны собственные менеджеры пакетов, позволяющие производить обновление ПО, автоматизировать процесс сборки программ, управлять зависимостями и т. п.

1.5. Развертывание среды и инструментов разработки программ на языке Фортран

Рассмотрим варианты решений и особенности установки на локальный компьютер пользователя и последующей настройки среды и инструментов разработки программ на языке Фортран.

В качестве базового компилятора можно рекомендовать свободно распространяемый компилятор с открытым исходным кодом GNU Fortran Compiler (или gfortran), который входит в состав набора GNU Compiler Collection (GCC, *https://gcc.gnu.org*) и может быть тем или иным способом установлен фактически на любую ОС.

Примером широко используемого коммерческого компилятора для классических архитектур ЦПУ является Intel Fortran Compiler, который оптимизирован для процессоров Intel и интегрируется в популярные IDE (Visual Studio Code, Microsoft Visual Studio, Eclipse, Code::Blocks и др.).

Другие примеры находящихся в стадии развития свободно распространяемых компиляторов – это LFortran (https://lfortran.org) и LLVM Flang (*https://github.com/llvm/llvm-project/tree/main/flang*). Эти продукты базируются на технологии LLVM, в основе которой лежит принцип промежуточного представления кода с возможностью его трансформации во время компиляции, компоновки и выполнения программы. В результате из такого представления генерируется оптимизированный бинарный код для ряда компьютерных архитектур и языков программирования.

Отметим, что современные компиляторы обладают средствами *оптимизации кода*, которые рекомендуется задействовать при выполнении программ. Активация оптимизации для компилятора gfortran производится посредством добавления к команде компиляции (или сборки) ключа `-O` с указанием уровня оптимизации `-O0`, `-O1`, `-O2`, `-O3` (`0` означает отсутствие оптимизации и применяется по умолчанию, положительные целые числа соответствуют разным уровням оптимизации).

Некоторые полезные ключи компилятора gfortran приведены и прокомментированы в прил. Б.

Отладка программ может производиться с помощью встроенных средств той или иной IDE, а также с использованием свободно распространяемых инструментов из коллекции GNU, таких как GNU Project Debugger (GDB, *https://www.gnu.org/software/gdb/*), графического интерфейса к нему Data Display Debugger (DDD, *https://www.gnu.org/software/ddd/*).

Обсуждаемое далее различие вариантов развертывания среды разработки связано с используемым на компьютере типом ОС – MS Windows, GNU/Linux или macOS.

1.5.1. Работа в операционной системе GNU/Linux

Ситуация, при которой на компьютере установлен один из дистрибутивов ОС GNU/Linux с графическим пользовательским интерфейсом, является в настоящее время в Российской Федерации не слишком распространенной, но при этом наиболее простой в отношении обсуждаемых вопросов.

Установка компилятора (при необходимости IDE и иных инструментов) разработки в общем случае предполагает использование стандартного менеджера пакетов ОС. Здесь и далее будем предполагать, что компьютер работает под управлением ОС, относящейся к десктопной, т. е. предназначенной для настольных компьютеров и ноутбуков, версии популярного дистрибутива GNU/Linux – Ubuntu (*https://ubuntu.com*).

В режиме командной строки терминала ОС Ubuntu компилятор `gfortran` можно установить командой (требуются права суперпользователя):

```
sudo apt install gfortran
```

Компиляция и запуск программ на Фортране на исполнение в наиболее простом случае осуществляется следующей последовательностью команд:

```
gfortran program.f90
./a.out
```

В этом примере `program.f90` – файл с исходным кодом программы (`f90` – рекомендуемое расширение), `a.out` – имя созданного в результате компиляции и сборки бинарного файла

программы, символы `./` в команде запуска указывают на расположение файла в текущем каталоге (например, в домашнем каталоге пользователя /`home/user`, где `user` – имя пользователя).

Вместо обезличенного имени файла `a.out` можно добавить к команде компиляции ключ `-o` для сохранения результатов работы в бинарный файл с заданным именем (например, `program`):

```
gfortran -o program program.f90
./program
```

Изучение базовых принципов работы в командной строке Linux, основных команд и утилит работы с файловой системой, процессами, пользователями, системными ресурсами и т. п. выходит за рамки данного учебного пособия.

Установка дополнительных инструментов среды разработки (IDE, редакторы исходного кода, библиотеки подпрограмм) с помощью менеджера пакетов не должна вызывать каких-либо трудностей.

В качестве среды для разработки программ при отсутствии необходимости использовать расширенный функционал тяжеловесных IDE можно рекомендовать установить один из следующего набора свободно распространяемых пакетов: Geany, ATOM, Code::Blocks, Sublime Text (имеют версии для Linux, MS Windows и macOS), например:

```
sudo apt install geany
```

Все перечисленные пакеты позволяют производить компиляцию и сборку программ с использованием графического меню, устанавливать соответствующие команды и ключи компилятора (при наличии установленного компилятора), а также запускать бинарные файлы на исполнение.

В случае необходимости задействования удаленного (расположенного в локальной или глобальной компьютерной сети) узла в составе высокопроизводительной вычислительной системы для запуска программ, требующих значительных ресурсов, можно воспользоваться стандартными средствами ОС – командой `ssh` (в составе пакета `openssh-client`) или пакетами с графическим интерфейсом PuTTY, EasySSH, Remmina и др.

1.5.2. Работа в операционной системе MS Windows

Использование подсистемы WSL. Развертывание полноценной среды разработки в ОС MS Windows 10/11 (и новее) без использования средств эмуляции (например, Cygwin, *https://cygwin.com*) и виртуализации (например, Virtualbox, *https://www.virtualbox.org*) базируется на подсистеме WSL (Windows Subsystem for Linux). WSL представляет собой совместно развиваемый компаниями Microsoft и Canonical экземпляр ОС Ubuntu GNU/Linux, который работает нативно в MS Windows. Для установки подсистемы необходимо запустить на компьютере оболочку Windows PowerShell от имени администратора и выполнить команду:

```
wsl –install –d ubuntu
```

После завершения установки следует перезагрузить компьютер, запустить приложение Ubuntu on Windows и задать в интерфейсе командной строки учетную запись пользователя и пароль для ОС Ubuntu (это потребуется сделать только при первом запуске).

Далее следует обновить списки программных пакетов из базы данных, доступных в репозиториях ПО, с учетом актуальных изменений и установить последние обновления командами:

```
sudo apt update
sudo apt upgrade
```

Установка программных пакетов осуществляется командой `apt install`, в частности, для компилятора `gfortran`:

```
sudo apt install gfortran
```

Типичный набор команд компиляции и сборки программы на языке Фортран, а также ее запуска на исполнение из командной строки:

```
gfortran -o program program.f90
./program
```

В данном случае предполагается, что каталог, в котором хранятся файлы с исходными кодами программ и исполняемые бинарные файлы, находится в файловой системе MS Windows (например, `C:/opt/fortran`), а переход в него осуществляется с помощью следующей команды (`/mnt` – каталог присоединения логических

дисков файловой системы MS Windows в ОС Ubuntu, c – имя логического диска):

```
cd /mnt/c/opt/fortran
```

Подсистема WSL (версия 2 и новее) поддерживает «естественную» работу приложений с графическим пользовательским интерфейсом в полностью интегрированном интерфейсе рабочего стола. WSL позволяет запускать графические Linux-приложения прямо из меню «Пуск» MS Windows, закреплять их на панели задач, переключаться между приложениями ОС Linux и MS Windows и использовать общий буфер обмена.

Для запуска приложений с графическим интерфейсом ОС Linux необходимо установить драйвер, соответствующий видеокарте компьютера, что позволит использовать виртуальную видеокарту для получения преимуществ отрисовки графики OpenGL с аппаратным ускорением.

Установка пакета Geany и последующий запуск осуществляются в подсистеме WSL следующими командами:

```
sudo apt install geany xterm
geany
```

Для компиляции и сборки программы в пакете Geany (рис. 1.2) необходимо в пункте основного меню пакета «Сборка» (Build) выбрать подпункт с таким же названием (можно использовать клавиатурную кнопку F9); запуск программы на выполнение инициируется с помощью подпункта меню «Выполнить» (или кнопкой F5).

Стандартные инструменты и протоколы для реализации графического пользовательского интерфейса в Unix-подобных ОС обеспечиваются оконной системой X Window System. Система базируется на клиент-серверной архитектуре, в рамках которой X-сервер обменивается служебными сообщениями и данными с клиентскими программами, имеющими оконный интерфейс. Сервер принимает от таких программ запросы на вывод графических объектов (окон) и отправляет обратно пользовательский ввод.

Востребованной на практике является реализованная в X Window System технология сетевой прозрачности, которая позволяет графическим приложениям исполняться на удаленном компьютере,

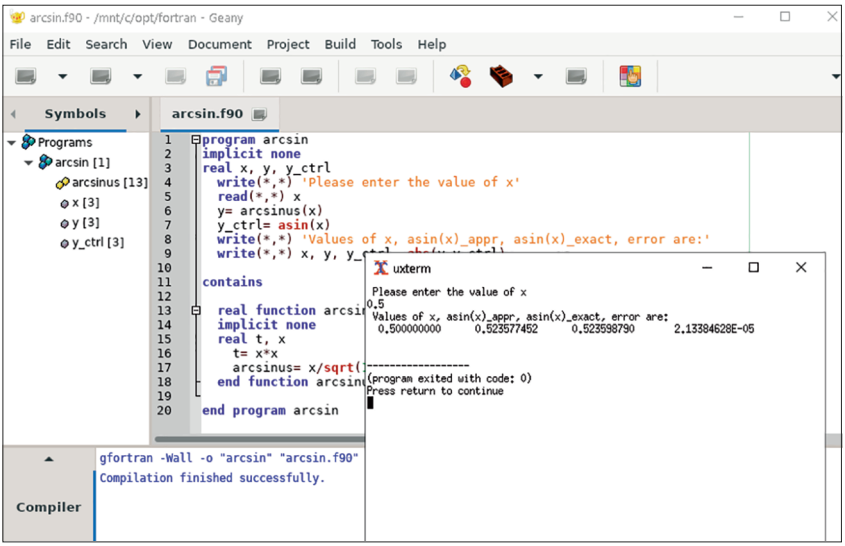

Рис. 1.2. Компиляция, сборка и запуск программы на языке Фортран в пакете Geany при работе в подсистеме WSL (ОС Ubuntu GNU/Linux, компилятор gfortran)

а оконный интерфейс приложений при этом транслируется по сети и отображается на локальном рабочем компьютере пользователя.

Среди разработанных для ОС MS Windows реализаций X-сервера присутствуют как коммерческие (OpenText Exceed, Reflection X, Xming), так и свободно распространяемые (Cygwin/X, VcXsrv) решения.

В случае отсутствия естественной поддержки подсистемой WSL работы приложений с графическим интерфейсом рекомендуется задействовать пакет VcXsrv. Процесс установки пакета предполагает скачивание соответствующего архитектуре компьютера компактного установщика с сайта *https://sourceforge.net/projects/vcxsrv/*.

Использование пакета Cygwin. Подход, альтернативный WSL, основан на использовании пакета Cygwin, который представляет собой комплекс свободно распространяемых инструментов, эмулирующих интерфейс командной строки Unix/Linux для ОС MS Windows. Cygwin является инструментом переноса (портирования)

ПО Unix в MS Windows, обеспечивающим интеграцию Windows-приложений, данных и ресурсов с соответствующими компонентами Unix-подобной среды. В состав эмулятора входит коллекция приложений, предоставляющая привычную Unix-среду, включая командные интерпретаторы, инструменты разработки ПО, библиотеки и прикладные пакеты.

Подробно ознакомиться с методиками работы с пакетами, входящими в состав Cygwin, можно с помощью официальной документации (*https://cygwin.com/docs.html*).

Компиляция, сборка и запуск программы на исполнение в среде Cygwin осуществляется с помощью следующих команд:

```
gfortran.exe -o program.exe program.f90
./program.exe
```

Стоит обратить внимание на наличие стандартного для исполняемых файлов MS Windows расширения `exe` у программы-компилятора и бинарного файла, что указывает на эмуляционную природу Cygwin в отличие от близкой к нативной реализации ОС Linux в WSL.

Бинарная совместимость между WSL и Cygwin отсутствует. Программы, скомпилированные в Cygwin, можно запустить в командной оболочке MS Windows в отличие от бинарных файлов из WSL. Это позволяет в том числе подключить компилятор из пакета Cygwin к IDE, например, к Geany, ATOM, Code::Blocks, Sublime Text.

Снимки экрана, демонстрирующие работу в пакете Geany с автоматически подключенным компилятором gfortran в составе эмулятора Cygwin, показаны на рис. 1.3.

Путь к исполняемым файлам Cygwin, включая и компилятор, можно установить или изменить с помощью переменной среды MS Windows с именем `Path` (например, `C:\opt\gcc\bin`).

Вместо Cygwin можно установить и подключить к IDE более компактный пакет MinGW, который представляет собой набор инструментов разработки ПО для создания приложений под MS Windows. Пакет содержит компилятор, нативный программный порт GNU GCC с набором свободно распространяемых библиотек и заголовочных файлов для Windows API, а также отладчик

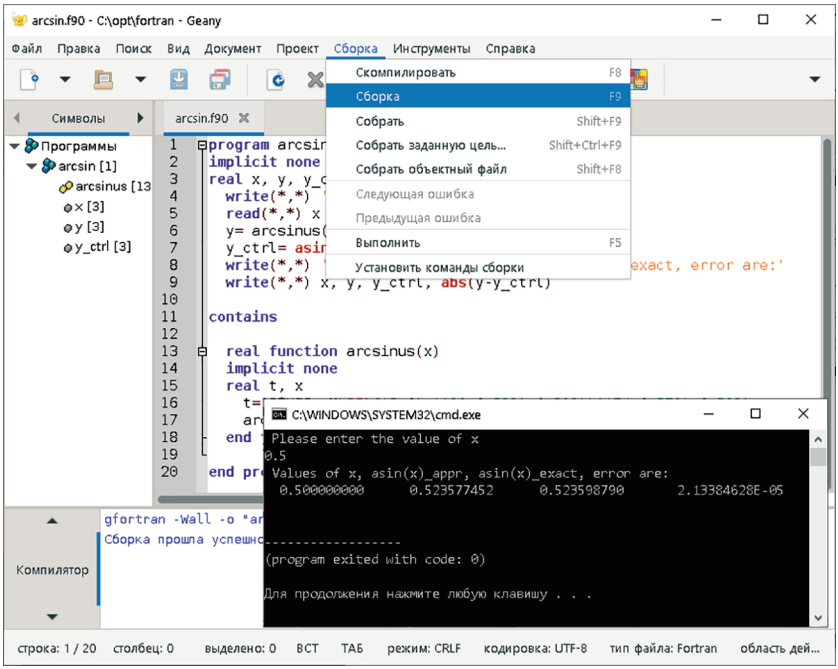

Рис. 1.3. Компиляция, сборка и запуск программы на языке Фортран в пакете Geany при работе в MS Windows с установленным Cygwin и компилятором gfortran

GDB. MinGW включен в специальные бинарные сборки IDE Code::Blocks (*https://www.codeblocks.org/downloads/binaries/*).

Использование интегрированной среды разработки. Перейдем к вопросам выбора, установки, настройки и общим принципам работы в интегрированных средах разработки как профессиональным инструментам с расширенными возможностями по написанию исходного кода, компиляции, автоматизации сборки и отладки, подключению сторонних библиотек и др.

В качестве примера укажем на свободно распространяемую IDE Eclipse (*https://eclipse.org*). Eclipse позволяет устанавливать программные расширения для работы со многими популярными ЯВУ, средствами разработки и структурирования информации

(языки программирования Фортран, C, C++, Java, Python и др., языки разметки HTML, XML, технологии параллельного программирования MPI и OpenMP).

Некоторые примеры расширений Eclipse:

– *Eclipse Parallel Tools Platform* (*PTP*) – IDE с поддержкой создания параллельных программ, написанных на языках C, C++ и Фортран;

– *Fortran Development Tools* (*Phortran*) – IDE с набором инструментов для разработки программ на Фортране (является частью PTP);

– *C/C++ Development Tooling* (*CDT*) – набор инструментов для разработчиков на языках C/C++.

Установка Eclipse на рабочем компьютере может быть выполнена после предварительной загрузки одной из размещенных на сайте коллекций сборок (*packages*), подготовленных для конкретной компьютерной платформы (ОС MS Windows, Linux, macOS, архитектура x86_64 или AArch64).

Рекомендуемая для установки сборка – *Eclipse IDE for Scientific Computing* (*https://eclipse.org/downloads/packages/*), в которую входит набор инструментов для разработчиков на языках C, C++, Фортран, включая отладчики, возможности удаленного создания, запуска и контроля приложений.

В общем случае процесс установки предполагает предварительную загрузку с сайта универсальной программы-установщика, которая позволяет в интерактивном режиме выбрать требуемые сборки, каталог установки и выполнить загрузку и локальную распаковку набора ПО среды.

Укажем на еще один, наиболее простой, способ, предполагающий работу с совместимой на бинарном уровне с MS Windows сборкой компилятора `gfortran`. Для этого потребуется загрузить файл с самораспаковывающимся архивом сборки с сайта *http://www.equation.com*, выполнить установку компилятора, сопутствующих библиотек, утилит и подключить его к той или иной IDE.

Таким образом, развертывание полноценной, не требующей покупки лицензий среды разработки на языке Фортран в MS Windows предполагает использование пакетов WSL и Eclipse.

При отсутствии потребности в тяжеловесных средствах разработки, отладки и тестирования программ можно остановиться на комбинации IDE Geany и пакета Cygwin (или упомянутой ранее нативной сборке компилятора). Минималистичный режим, предполагающий работу только в командной строке, базируется на использовании WSL или пакетов Cygwin и MinGW.

Получить в свое распоряжение полноценную реализацию ОС Linux можно с помощью пакета виртуализации Virtualbox (*https://virtualbox.org*). Пакет позволяет создавать виртуальные машины с выделением им аппаратных ресурсов, запускать их и работать в поддерживаемых гостевых ОС, в том числе и Linux. Данный подход предполагает выделение виртуальным машинам в монопольное пользование одного или нескольких ядер ЦПУ, объема ОЗУ и дискового пространства (которые будут монопольно использоваться гостевой ОС), скачивание ISO-образа интересующей ОС, установку и настройку.

Отметим, что доступ из MS Windows на удаленный Linux-узел может осуществляться, например, с помощью свободно распространяемого пакета с графическим интерфейсом PuTTY (*https://putty.org*).

1.5.3. Работа в операционной системе macOS

Для компьютеров Apple с операционной системой macOS можно указать на следующие варианты установки компилятора `gfortran` – использование менеджера пакетов Homebrew (*https://brew.sh*) или установщиков из проекта на GitHub (*https://github.com/fxcoudert/gfortran-for-macOS/*). В первом случае в терминале ОС необходимо подать следующие команды:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/
install/HEAD/install.sh)"
brew install gfortran
```

Дальнейшая работа в режиме командной строки терминала типична для Unix-подобных ОС. Можно также установить и использовать кроссплатформенные среды разработки Geany, ATOM, Code::Blocks, Sublime Text, Visual Studio Code или Eclipse.

1.5.4. Онлайн-компиляторы для языка Фортран

В условиях бурного развития средств телекоммуникаций и технологий Интернета, а также обеспечиваемых на их основе разнообразных сервисов создаются и становятся доступными в том числе веб-ориентированные инструменты для разработки программ на разных языках. Такие решения предполагают интерактивную работу в веб-браузере без необходимости установки специализированного ПО на локальный компьютер пользователя.

Типичные возможности онлайн-компиляторов – это редактор исходного кода с подсветкой синтаксиса; инструменты активации компиляции, сборки и запуска программ на исполнение; возможности подключения внешних библиотек, задания версии и ключей компилятора, входных данных для программы; поле вывода результатов работы.

Некоторые примеры онлайн-компиляторов:

– LFortran (рис. 1.4), *https://dev.lfortran.org* (использует популярную интерактивную веб-платформу Jupyter, *https://jupyter.org*);

– GodBolt Compiler Explorer, *https://godbolt.org*;

– OnlineGDB, *https://www.onlinegdb.com/online_fortran_compiler*;

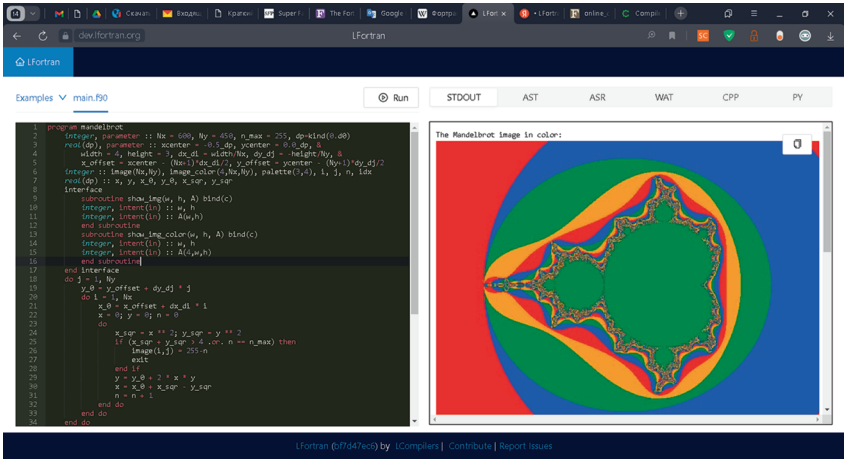

Рис. 1.4. Веб-интерфейс онлайн-компилятора LFortran

– OneCompiler, *https://onecompiler.com/fortran*;

TitorialsPoint,
https://www.tutorialspoint.com/compile_fortran_online.php.

Следует отметить, что онлайн-компиляторы имеют в целом менее развитый функционал по сравнению с локально установленными IDE, в частности, могут отсутствовать или быть неполноценными средства отладки и профилировки кода. Очевидно также, что возможности выполнения ресурсоемких вычислений в таких средах ограничены.

Глава 2
АЛГОРИТМЫ, ЯЗЫКИ И ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

2.1. Алгоритмы и их преставление

Алгоритм – одно из базовых понятий математики и информатики, связанных с программированием, которое не имеет единого строгого определения. Алгоритм отражает последовательность действий, направленных на решение некоторой задачи, и состоит из трех этапов: ввод исходных данных, их обработка и вывод результатов.

Из присущих алгоритму свойств выделим следующие три:

– *результативность, или конечность*, – алгоритм должен приводить к решению задачи за конечное число шагов;

– *массовость* – алгоритм должен быть применим для некоторого класса задач, различающихся лишь исходными данными;

– *быстродействие* – время работы алгоритма в определенном смысле должно быть минимальным.

В специальной литературе алгоритмам могут приписываться разные характеристики, но даже приведенный список указывает на то, что свойства алгоритма могут вступать друг с другом в противоречие (наглядность и эффективность, массовость и быстродействие, надежность и простота, простота и быстродействие и т. п.).

Алгоритмы представляются в следующих основных видах:

– *словесном* – запись на естественном (разговорном) языке;

– *графическом* – изображение с помощью графических блоков и символов;

– *псевдокод* – полуформализованное описание алгоритма на условном языке, включающем как элементы языка программирования, так и фразы естественного языка;

– *программы* – тексты на языках программирования.

2.2. Псевдокод и блок-схемы

В процессе проектирования необходимо описать подлежащий реализации алгоритм. Описание должно быть представлено в общепринятой форме, понятной и легко воспринимаемой, что должно упростить воплощение концепции в программный код. Стандартные формы, используемые для описания алгоритмов, принято называть конструкциями, а описываемый с помощью этих конструкций алгоритм – структурированным.

Конструкции, используемые для построения алгоритмов, можно представить двумя способами – псевдокодом или блок-схемами. *Псевдокод* – это гибридная смесь языка программирования и естественного языка; имеет структуру, подобную языку программирования, с отдельной строкой для каждой идеи или сегмента кода; при этом описания в строках приводятся на естественном языке. Каждая строка псевдокода должна описывать идею простым и понятным языком. Псевдокод имеет собственный словарь и полезен для разработки алгоритмов в силу своей гибкости и легкости модификации.

Напишем псевдокод для алгоритма перевода вводимого пользователем значения температуры в градусах Цельсия в градусы Фаренгейта, осуществляемого по следующей формуле (символ «←» означает в данном случае команду присваивания):

```
Tf = 32 + 9 · Tc / 5.
```

```
Начало
Ввести температуру в градусах Цельсия
```

№	Обозначение	Символ	Описание
(а)		*Овал*	Начало или конец алгоритма
(б)		*Прямоугольник*	Выполнение действий
(в)		*Параллелограмм*	Операции ввода или вывода
(г)		*Ромб*	Разветвление алгоритма
(д)		*Прямоугольник с двойной линией*	Обращение/вызов подпрограммы
(е)		*Шестиугольник*	Циклические операции
(ж)		*Стрелка*	Направление хода программы

Рис. 2.1. Геометрические фигуры, используемые в блок-схемах

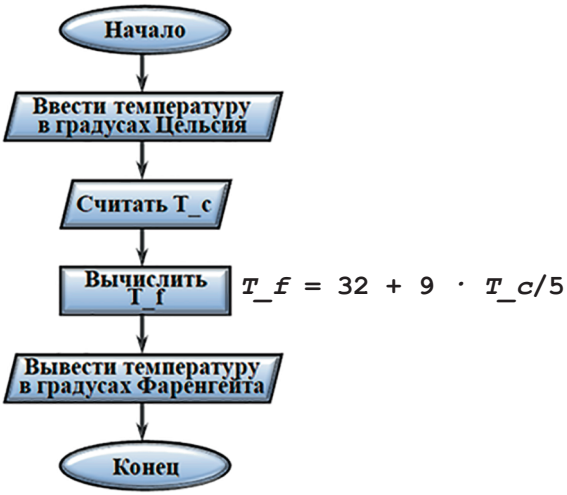

Рис. 2.2. Блок-схема алгоритма перевода градусов Цельсия в градусы Фаренгейта

```
Считать температуру в градусах Цельсия (T_c)
Вычислить T_f в градусах Фаренгейта ← 32 + 9 · T_c / 5
Вывести температуру в градусах Фаренгейта
Конец
```

Или в сокращенном виде:

```
Начало
Ввести T_c
T_f ← 32 + 9 · T_c / 5
Вывести T_f
Конец
```

Блок-схема – способ графического описания алгоритма, при котором разные геометрические фигуры и графические символы представляют собой разные операции, а стандартные конструкции состоят из наборов таких объектов.

Блок-схемы требуют более существенных усилий для модификации по сравнению с псевдокодами и не всегда эффективны на предварительных этапах определения алгоритма, когда могут происходить частые изменения. Полезная практика использования блок-схем – описание реализованного в программе алгоритма после завершения разработки.

Наиболее распространенные фигуры (блоки) и символы, используемые в блок-схемах, показаны на рис. 2.1, а блок-схема обсуждаемого алгоритма перевода температуры между единицами измерения – на рис. 2.2.

2.3. Типы алгоритмов. Проектирование «сверху вниз»

Типы алгоритмов принято разделять на линейные, разветвляющиеся и циклические. *Линейные алгоритмы* состоят из последовательности выполняемых друг за другом простых команд (присваивания и ввода/вывода), ветвления, циклы при этом отсутствуют (см. пример на рис. 2.2).

Алгоритмы, имеющие несколько ветвей (направлений вычислений, действий), называются *нелинейными*, к ним относятся *разветвляющиеся* и *циклические* алгоритмы. Ветвление алгоритма (рис. 2.3) происходит с помощью проверки выполнения некоторого условия, формулируемого в коде программы в виде *логического высказывания* (утверждения).

Условие может быть связано с любой характеристикой объекта, принимающей значения «*истина*» или «*ложь*». Например,

утверждение `x` – четное число истинно, если `x = 186`, и ложно, если `x = 123`. Отношение вида `x < y` следует трактовать как вопрос: «Верно ли, что `x < y`?»

Циклы – важнейшая составная часть нелинейных алгоритмов и большинства программ вычислительной направленности. Циклом называется блок последовательных действий, завершающийся переходом на первую команду блока операторов и конструкций или выходом из него при выполнении некоторого условия. Существует два основных вида циклов – *арифметический* и *итеративный*.

В *арифметическом цикле* явно присутствует параметр, управляющий количеством повторений (рис. 2.4). Этот параметр возникает, например, при вычислении конечных сумм и произведений, табулировании значений функций, работе с одномерными и многомерными массивами.

Итеративным называют цикл с заранее неизвестным числом повторений блока операторов и конструкций, содержащегося в нем (рис. 2.5). Обязательно должно быть корректно сформулировано условие выхода из такого цикла, не приводящее к бесконечной работе алгоритма (зацикливанию). Такой вид цикла является составной частью итерационных методов – последовательных приближений, методов решения СЛАУ, вычисления бесконечных сумм и произведений и т. п.

Разработка алгоритмов осуществляется, как правило, по принципу *«сверху вниз»*. Суть этого подхода состоит в разбиении

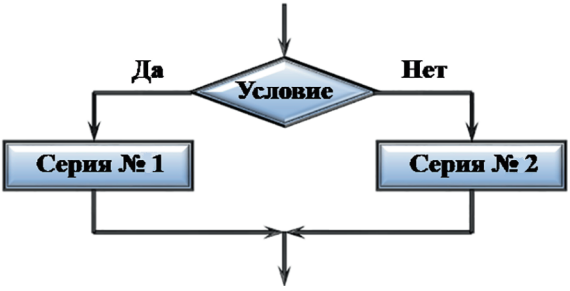

Рис. 2.3. Блок-схема разветвляющегося алгоритма

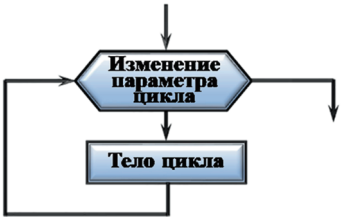

Рис. 2.4. Блок-схема арифметического цикла

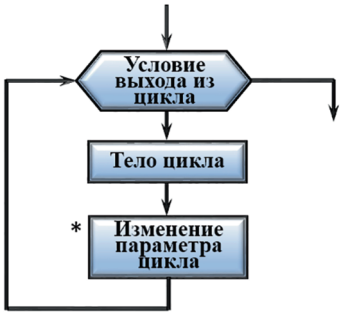

Рис. 2.5. Блок-схема итеративного цикла (* – параметра цикла может не быть)

(декомпозиции) исходной задачи на ряд более простых подзадач (фрагментов) и последующей работе с выделенными логическими фрагментами.

В процессе разбиения задачи на фрагменты целесообразно придерживаться следующей типовой схемы:

1. Четко сформулировать, проанализировать решаемую задачу и выделить в ней фрагменты.

2. Для каждого фрагмента определить, какие данные он получает (*входные данные*) и какие возвращает (*выходные данные*); установить связи между выделенными фрагментами.

3. Рассмотреть каждый фрагмент независимо и разработать для него *вспомогательный алгоритм*, при необходимости подвергнуть фрагмент дальнейшему разбиению на более мелкие.

4. Реализовать выделенные фрагменты при помощи подходящих средств используемого языка программирования (с учетом связей).

5. Протестировать полученные фрагменты (сначала независимо, а затем в составе единой программы).

Вспомогательные (*подчиненные*) *алгоритмы* вместе образуют систему, управление которой берет на себя алгоритм-диспетчер, который принято называть главным (головным). В состав функций головного алгоритма обычно входит ввод исходных данных,

активизация по мере необходимости подчиненных алгоритмов и использование результатов их работы, вывод результатов, выполнение служебных операций и т. д.

2.4. Языки программирования

Язык программирования – специализированный искусственный язык, позволяющий записать алгоритм решения задачи в виде программы. Для этой цели можно использовать и любой естественный язык, но на языки программирования наложено одно существенное ограничение – записанный на нем текст должен легко преобразовываться во внутренний «язык» компьютера, на котором он будет обрабатываться. Это, кроме всего прочего, предполагает, что на компьютере установлена программа-переводчик с языка программирования в машинные коды – *транслятор*.

Транслятор обязательно должен входить в состав средств программирования высокого уровня. Современные трансляторы – это не только программа-переводчик, а мощная среда, обеспечивающая весь цикл обработки программы. Транслятор может работать в режимах *компиляции* или *интерпретации*.

Компиляция – этап перевода модулей программы (или ее целиком) в машинные коды и их сборка в единое целое с привязкой к определенному адресному пространству. Создание исполняемого файла из исходного текста программы предполагает выполнение процессов компиляции и *компоновки* (линковки). Программа-компоновщик принимает на вход один или несколько модулей и собирает из них исполняемый или библиотечный файл.

В случае *интерпретации* перевод исходного кода программы в машинный осуществляется поочередно, команда за командой, переведенная команда выполняется, после чего перевод продолжается.

В процессе эволюции был предложен целый ряд *классификаций языков программирования*. Одна из них основывается на степени близости языка к машинным кодам, где нижний уровень – это машинные коды, а уровень выше – *ассемблер* (машинно-ориентированный язык) и т. д. В общем случае чем ниже уровень языка, тем

меньшая адресуемая единица памяти оказывается доступной, тем большие возможности предоставляются разработчику.

Системное программирование требует доступа к минимальной адресуемой единице в памяти, и для него в наибольшей степени подходят языки ассемблера и C. *Прикладное программирование* обычно производится на языках высокого уровня (ЯВУ), которые имитируют машинные языки, используя слова естественного языка и математические символы.

Примерами наиболее востребованных на практике ЯВУ являются сегодня Python, C, C++, Java, C#, JavaScript, Visual Basic, PHP, Go, Delphi/Object Pascal, MATLAB, Swift, Фортран, R, Kotlin, Ruby и др. (см., например, TIOBE Index: *https://www.tiobe.com/tiobe-index/*).

Другая возможная классификация основана на областях применения (например, упоминавшиеся системное программирование, научные вычисления, обработка и аналитика данных, разработка веб-приложений, приложений для мобильных устройств, настольных приложений, компьютерных игр, разработка программ, базирующихся на технологиях искусственного интеллекта и машинного обучения и т. д.).

В связи с этим можно (весьма условно) разделить языки на *универсальные* и *специализированные*. *Универсальные языки* (или языки общего назначения) могут использоваться для разработки программ без привязки к конкретным областям и задачам (например, C и C++). *Специализированные языки* ориентированы на решение какого-то одного или нескольких близких видов задач (например, научные и инженерные вычисления – Фортран, R, Python, MATLAB, Julia; веб-разработка – Python, PHP, SQL, JavaScript; мобильные приложения – Swift, Objective-C, Java, Kotlin).

Примеры компилируемых ЯВУ – C, C++, Фортран, Паскаль, Go. К интерпретируемым языкам относятся такие языки, как Python, Julia, R, JavaScript, PHP, MATLAB, Ruby, Perl.

Промежуточный между компиляцией и интерпретацией подход предполагает перевод программы не напрямую в машинный язык, а в *машинно-независимый код низкого уровня* (байт-код)

и последующее выполнение этого кода виртуальной машиной. В роли виртуальной машины обычно выступает интерпретатор (иногда дополненный JIT/AOT-компилятором), при этом отдельные части для ускорения работы могут быть транслированы в машинный код во время выполнения программы по технологии компиляции «на лету» (динамическая компиляция).

Преимущество байт-кода состоит в большей эффективности и переносимости (один и тот же код может исполняться на разных платформах и архитектурах, для которых реализован интерпретатор). Примеры классических языков, использующих байт-код в качестве базового подхода: Java, C#.

Следует отметить, что ряд интерпретируемых языков (в частности, Python, Perl, PHP, Ruby) в отдельных реализациях задействует концепцию байт-кода для ускорения и облегчения работы интерпретатора.

2.5. Парадигмы программирования

Парадигма программирования – устоявшаяся система взглядов, совокупность идей и понятий, специфицирующих стиль написания программ, способ концептуализации, определяющий организацию вычислений и структурирование выполняемых компьютером действий, а также способ классификации языков программирования на основе их особенностей. Верхнеуровневый классифицирующий подход выделяет среди разработанных парадигм *императивную* и *декларативную*.

Императивная парадигма представляет программу в виде точно описанной разработчиком последовательности действий (инструкций, операторов), преобразующих состояние программы для получения требуемого результата. Допустимые операторы и типы данных определяются конкретным языком программирования. Основными элементами в императивных языках являются именованные переменные, операторы присваивания, ввода/вывода, составные выражения, стандартные алгоритмические конструкции (такие, как ветвления и циклы), а также подпрограммы.

К классу императивных парадигм принято относить процедурный и объектно-ориентированный подходы к разработке программ. В *процедурной парадигме* отдельные группы часто повторяемых операторов выделяются в подпрограммы, которые содержат подлежащие выполнению последовательности действий (коды для конкретных подпрограмм). Примеры языков, базирующихся на процедурной парадигме: C, Фортран, Паскаль.

Процедурная парадигма иногда рассматривается в составе *структурного подхода*, основой которого является представление программы в виде иерархической структуры блоков (логически сгруппированных частей кода, например, записанных подряд наборов инструкций). Программы строятся без использования оператора безусловного перехода, включают три базовые управляющие конструкции (последовательность, ветвление, цикл), которые могут быть вложены друг в друга произвольным образом. Повторяющиеся фрагменты можно (но необязательно) оформлять в виде подпрограмм. Разработка ведется пошагово на основе метода «сверху вниз».

Объектно-ориентированная парадигма основана на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса. Классы – комплексные типы данных, состоящие из тематически единого набора полей (переменных более простых типов) и методов (функций для работы с этими полями). Классы образуют иерархию наследования, которая позволяет описать новый класс на основе свойств уже существующего с частично или полностью заимствованной функциональностью.

Эта парадигма возникла в результате развития процедурной идеологии, в которой данные и подпрограммы их обработки формально не связаны. Она позволила существенно повысить скорость написания кода и сделать его легкочитаемым, повысить управляемость программами, быстрее внедрять изменения и масштабировать код под различные задачи, что особенно важно при реализации крупных и сложных проектов. Примеры объектно-ориентированных языков: Python, C++, Java, C#, Ruby, PHP, JavaScript.

Декларативная парадигма основана на задании спецификации решения задачи – описывается, что представляют собой проблема и ожидаемый результат без предложения способа его получения. Декларативные программы не используют понятие состояния, не содержат переменных и операторов присваивания. Соответствующие парадигме языки программирования описывают то, что должна выполнить программа, с позиции предметной области, а не в виде последовательности примитивов языка.

В составе декларативной парадигмы обычно выделяют функциональное и логическое программирование. *Функциональное программирование* базируется на использовании функций, значения которых определяются по заданным параметрам. Вычислительный процесс трактуется как вычисление значений функций в их математическом понимании (в отличие от функций как подпрограмм в процедурном подходе). Роль основной конструкции в данном случае играют выражения, к которым относятся скалярные константы, структурированные объекты, функции, их тела и вызовы.

Программа представляет собой совокупность описаний функций и выражения, которые необходимо вычислить. Вычисление описывается как комбинация вызовов функций того же или более низкого уровня (часто рекурсивных). Каждая следующая функция в этой комбинации описывается аналогичным образом, до тех пор, пока процесс не сведется к предопределенным функциям, вычисление которых считается заданным.

Функциональное программирование не использует концепцию памяти как хранилища значений переменных. Операторы присваивания отсутствуют, вследствие чего «переменные» обозначают не области памяти, а объекты программы, что соответствует понятию переменной в математике. Примеры функциональных языков: Lisp, Scheme, Erlang, Haskell, OCaml, F#.

Логическое программирование – парадигма, основанная на математической логике, автоматическом доказательстве утверждений, с использованием механизмов логического вывода информации на основе заданных фактов и правил вывода. В программах на логических языках нет определенного (фиксированного) порядка

выполнения правил и шагов алгоритма, а выбор подходящей последовательности возлагается на компьютер.

Логическое программирование используется для создания баз знаний и экспертных систем, проведения исследований в сфере искусственного интеллекта и машинного обучения (на основе логических моделей баз знаний и логических процедур вывода и принятия решений). Примеры логических языков: Prolog, Datalog, Mercury, Planner, Popler.

Помимо уже перечисленных в процессе эволюции был разработан целый ряд других парадигм и подходов, в том числе автоматное, реактивное, событийно-ориентированное, нейролингвистическое, языково-ориентированное программирование, метапрограммирование, программирование, основанное на данных, и др.

Подход, известный как *мультипарадигменное программирование* предполагает задействование при разработке сразу нескольких разных парадигм и подходов. Основная цель данного подхода – использовать лучшие практики и инструменты для работы, отталкиваясь от того, что никакая парадигма не решает все проблемы самым легким и/или самым эффективным способом.

2.6. Язык Фортран: прошлое, настоящее, будущее

Фортран – де-факто первый и до сих пор широко используемый в практике разработки программного обеспечения (ПО) стандартизованный императивный компилируемый ЯВУ со статической типизацией, ориентированный на научные и инженерные вычисления. Англоязычное исходное название языка – FORTRAN (FORmula TRANslator или TRANslation в разных источниках – переводчик формул). В настоящее время принято использовать вариант Fortran.

Выраженная целевая направленность языка, высокая производительность выполнения программ, относительно несложное освоение, интуитивно понятный синтаксис, наличие большого количества написанных на нем и используемых на протяжении многих лет готовых библиотек и пакетов прикладных программ

(в том числе с открытым исходным кодом), поддержка современных технологий высокопроизводительных вычислений делают Фортран востребованным в профессиональной среде и актуальным в перспективе еще долгое время (особенно в областях вычислительной математики, физики, инженерных расчетов).

Фортран имеет большой набор встроенных функций разных типов, поддерживает работу с целыми, вещественными и комплексными числами высокой точности, имеет типовые лексические конструкции (условия, циклы), средства форматного ввода/вывода (в том числе файлового), эффективно работает с многомерными массивами.

Специалистам известны перечисленные ранее библиотеки научных и инженерных вычислений (BLAS, LAPACK, Netlib, IMSL и др.), библиотеки и технологии, нацеленные на организацию параллельных и распределенных вычислений (MPI, OpenMP), построение графических интерфейсов (Quickwin, Fortran/Tk) и др.

Фортран, первая версия которого была разработана коллективом Дж. Бэкуса и представлена компанией IBM в 1957 г. (*Фортран I*), открыл эру автоматизации программирования, предоставив инженерам и исследователям возможность кодировать математические формулы в привычном для них виде и не прибегать к услугам профессиональных программистов, разрабатывающих программы вручную на машинных языках. Компилятор языка Фортран преобразовывал написанные человеческим языком коды в машинный язык, который компьютер мог распознавать и выполнять.

Начальные версии Фортрана были компактными по сравнению с современными версиями, содержали ограниченное количество операторов и поддерживали только целочисленные и вещественные типы данных.

Дальнейшие разработки продолжались до 1962 г., когда был выпущен *Фортран IV*, который стал значительным усовершенствованием и стандартной версией языка на 15 лет. В 1966 г. язык был принят в качестве ANSI-стандарта и стал известен как *Фортран 66*.

В 1977 г. язык получил еще одно серьезное обновление. Версия *Фортран 77* включила множество новых функций, призванных

упростить написание и поддержку структурированных программ, были добавлены конструкция условия `IF`, возможности работы с символьными переменными, усовершенствованы возможности ввода/вывода (включая файловый) и т. д.

Следующим существенным обновлением языка стал *Фортран 90* (стандарт был выпущен в 1991 г.), включивший в себя целиком Фортран 77 в качестве подмножества и расширенный во многих новых направлениях. Среди основных улучшений – свободный формат записи кода, операции над массивами как аргументами встроенных функций и арифметических действий, сечения массива, модули как единицы совместной компиляции, новые встроенные и рекурсивные процедуры и функции, производные типы данных и некоторые элементы объектно-ориентированного программирования, указатели, операторные скобки `DO ... END DO`, операторы `DO WHILE`, `WHERE`, `CYCLE` и `EXIT`, оператор выбора `SELECT CASE`, улучшенные возможности ввода/вывода и др.

Фортран 90 был радикальным обновлением по сравнению с более ранними версиями, окончательно обозначился ориентир на специализацию в области написания вычислительных программ с поддержкой параллелизма (с сохранением претензий на универсальность). Фортран стал позиционироваться как язык для удобной и высокоэффективной работы с массивами.

Достойно упоминания разработанное в 1993 г. расширение Фортран 90 (High Performance Fortran, HPF) с введением конструкций, реализующих параллельные вычисления. HPF основывался на синтаксисе массивов из базового стандарта и использовал модель параллелизма данных для распределения операций с массивом на несколько процессоров.

В 1994 г. был опубликован стандартизированный и переносимый интерфейс для передачи информации (сообщений) MPI (Message Passing Interface), предназначенный для многопроцессорных архитектур. В настоящее время MPI де-факто является основной технологией разработки параллельных приложений для вычислительных систем с распределенной памятью и реализует механизмы обмена данными между процессами, совместно

выполняющими некоторую задачу. Стандарт сформулирован для Фортран и C, а реализован (в виде библиотеки) также для C++, Java, Python, Julia, C#.

В 1997 г. последовало незначительное обновление языка под названием *Фортран 95*, добавившее в него ряд новых функций (в том числе из HPF), таких как конструкция `forall` и вложенные where, поэлементные процедуры, атрибуты `pure` и `impure`, специфицирующие наличие побочного действия у процедур, возможности использования размещаемых массивов в качестве компонентов структур, фиктивных аргументов, результатов функций и некоторые новые внутренние процедуры.

Следующим содержательным обновлением стал *Фортран 2003*, включивший такие новые функции, как расширенные производные типы, полноценная поддержка объектно-ориентированного программирования, стандартизация асинхронного ввода/вывода, улучшения манипулирования данными, указатели процедур, поддержка интернационализации и совместимость с языком C, включая механизмы обмена данными. В этот период была разработана и реализована для языка Фортран поддержка технологии OpenMP, позволяющей управлять исполнением кода на параллельных архитектурах с общим полем оперативной памяти.

Стандарт языка *Фортран 2008* (принят в 2010 г.) в качестве основного обновления получил поддержку модели параллельных вычислений Coarray Fortran, в рамках которой могут запускаться несколько одинаковых исполняемых файлов (образов) программы с приватным адресным пространством и coarray-переменными, доступными для чтения/записи из любого образа (включая распределенные массивы). Кроме того, были введены оператор параллельного цикла `do concurrent`, атрибут `asynchronous` для асинхронных переменных в массивно-параллельных средах, операторные скобки `block... end block`, позволяющие повысить уровень локализации переменных, атрибут `contiguous` для более эффективного управления памятью, подмодули, новые встроенные процедуры и математические функции и др.

Фортран 2018 (принят в 2018 г.) стал незначительным расширением предыдущего стандарта. Основные изменения коснулись функций, определенных в технических спецификациях, были введены дополнительные параллельные функции, дальнейшие расширения совместимости с языком C и довольно большое количество небольших улучшений, в том числе устраняющих обнаруженные недостатки и несоответствия.

В 2023 г. был принят актуальный на момент написания пособия стандарт языка (*Фортран 2023*) с основными нововведениями в части развития параллельных функций, включая конструкцию `notify` для синхронизации обмена данными между образами программы, атрибут `simple` подпрограмм и функций для оптимизации параллельных вычислений, возможность оператора `do concurrent` включать спецификацию редукции, а также оператор `@` для замыкания индексации массивов, условные выражения, перечисления, новые полезные встроенные функции.

Таким образом, современные версии Фортрана позволяют разрабатывать программы на основе парадигмы, наилучшим образом соответствующей особенностям решаемой задачи, – процедурной, структурной, объектно-ориентированной или основанной на массивах, использовать возможности организации параллелизма на многопроцессорных компьютерах.

Отметим, что примеры кодов в данном учебном пособии приводятся в современных версиях Фортрана (2008/2018/2023), хотя и не используют многие новые возможности из этих стандартов.

Глава 3
ОСНОВЫ ЯЗЫКА ФОРТРАН

3.1. Алфавит языка

Алфавит, используемый в языке Фортран, известен как «набор символов Фортрана» и состоит из символов, приведенных в табл. 3.1.

Таблица 3.1

Символы языка Фортран

Количество символов	Тип	Значения
26	Заглавные буквы	A-Z
26	Строчные буквы	a-z
10	Цифры	0-9
1	Символ подчеркивания	_
5	Арифметические символы	+ - * / **
28	Специальные символы	( ) . = , ' $: ! " % & ; < > ? ~ \ [ ] ' ^ { } | # @ и пробел

Стоит обратить внимание на то, что Фортран нечувствителен к регистру – прописные буквы алфавита в наборе символов эквивалентны строчным. Это контрастирует, к примеру, с такими регистрозависимыми языками, как C, C++, Java, Python.

3.2. Общая структура программы

Программа на Фортране представляет собой последовательность следующих друг за другом в определенном порядке *операторов* и *конструкций* (последовательности могут оформляться в виде отдельных *подпрограмм*). Операторы разделяются на *исполняемые* (описывающие действия, которые должны быть произведены программой) и *неисполняемые* (непосредственно не участвующие в вычислениях). Конструкции состоят из нескольких операторов и применяются для выполнения управляющих действий (например, условных операторов или циклов).

Фортран поддерживает возможность записи кода в фиксированной (устаревшей) и в свободной (основной для современных версий языка) формах. В *свободной форме* операторы можно вводить в любом месте строки, ее длина может достигать 132 символов,

в пределах одной строки могут размещаться несколько операторов, разделенных точкой с запятой. В том случае, если строка текста завершается символом амперсанда «&», последующая строка рассматривается как строка продолжения. Любые расположенные между восклицательным знаком и концом строки символы рассматриваются как комментарии (игнорируемые компилятором).

Основными программными единицами языка являются головная программа, модуль, блок данных, а также внешняя и внутренняя подпрограммы и функции, которые будут обсуждаться по ходу изложения.

Приведем в целях начального погружения в вид исходного кода на Фортране пример программы, записанной в свободной форме и выполняющей вычисление арксинуса по следующей приближенной формуле:

$$\arcsin(x) \approx \frac{x}{\sqrt{1-x^2}} \frac{50x^4 - 155x^2 + 105}{24x^4 - 120x^2 + 105}.$$

```
program arcsin_approximate          ! тело головной программы
! Программа вычисления арксинуса по приближенной формуле
implicit none
real :: x, y, y_exact
  write(*,*) 'Введите значение x:'; read(*,*) x
  y = arcsinus(x)                   ! вызов внутренней функции
  y_exact = asin(x)                 ! вызов встроенной функции
  write(*,*) 'Значения x, asin(x) прибл, asin(x) точн, ошибка: '
  write(*,*) x, y, y_exact, abs(y - y_exact)
contains
  real function arcsinus(x)          ! тело внутренней функции
  implicit none
  real :: t, x
    t = x * x
   arcsinus = x / sqrt(1 - t) * ((50.*t - 155.) * t + 105.) / &
                                ((24.*t - 120.) * t + 105.)
  end function arcsinus
end program arcsin_approximate
```

Отметим, что в этой программе используются рассматриваемые далее операторы начала и завершения программы (**program, end program**); оператор объявления типа данных (**real**); оператор,

предписывающий обязательность явного объявления всех используемых в программных единицах объектов данных (**implicit none**), операторы ввода/вывода (**read, write**); оператор присваивания (=), вычисления выражений (арифметические операции), задания функции (**function**); оператор, отделяющий тело головной программы от внутренней функции (**contains**), и встроенные элементарные функции (asin, abs, sqrt).

Имя программы (и других программных единиц), указываемое после операторов начала и завершения программы, может содержать до 63 буквенно-цифровых символов (начиная с буквы) и символ подчеркивания.

Отметим, что в приводимых в пособии исходных кодах программ ключевые (служебные) слова (операторы) языка Фортран выделяются полужирным шрифтом, а также для лучшей читаемости кода используются горизонтальные и вертикальные отступы.

Программы (и программные единицы) разделяются на следующие основные секции, расположенные в коде в указанном порядке:

1. *Секция объявления* состоит из группы обычно неисполняемых операторов в начале программы, включая ее имя, а также имена и типы используемых переменных (может производиться присвоение начальных значений переменным и константам).

2. *Секция исполнения* – группа операторов и конструкций, описывающих действия, которые должна выполнить программа.

3. *Секция завершения* включает оператор или операторы, останавливающие выполнение программы и сообщающие компилятору, что программа завершена.

3.3. Объекты данных. Стандартные типы данных

Обратимся к вопросам *организации данных* в языке Фортран. Для каждого применяемого внутри программы *объекта данных* должны быть определены *тип*, *диапазон изменения* (для числового типа), а также *форма представления* – скаляр или массив.

Данные разделяются на изменяемые (*переменные*) и не подлежащие изменению (*константы*). Термины «переменная»

и «константа» распространяются на скалярные объекты, массивы и их подобъекты (см. гл. 7).

Типы данных подразделяются на *встроенные* и *производные* (создаваемые пользователем). Каждый *встроенный тип* данных характеризуется параметром *разновидности* (**kind**). Для числовых типов данных этот параметр описывает точность представления и диапазон изменения. Каждый встроенный тип данных имеет стандартную, задаваемую по умолчанию разновидность. Встроенный тип с задаваемой по умолчанию разновидностью называется *стандартным* типом данных.

Стандартные типы данных и соответствующие операторы объявления следующие: целый – **integer**; вещественный – **real**; комплексный – **complex**; логический – **logical**; символьный – **character**.

3.4. Числовые объекты данных. Оператор присваивания

Число байтов, отводимых под хранение объектов целого, вещественного и комплексного (вещественной и мнимой частей) типов данных, совпадает с разновидностью (по умолчанию, без явного указания значения разновидности, равно четырем, если это содержится в настройках системы).

Информация о диапазонах и точности изменения данных *целого типа* с доступными параметрами разновидности (записываются в скобках) приведена в табл. 3.2.

Таблица 3.2

Точность и диапазоны изменения данных целого типа

Обозначение типа данных и параметр разновидности	Диапазон и точность
integer(1)	-128… 127
integer(2)	-32 768… 32 767
integer(4)	-2 147 483 648… 2 147 483 647
integer(8)	-9 223 372 036 854 775 808… 9 223 372 036 854 775 807

Таблица 3.3

Точность и диапазоны изменения данных вещественного типа

Обозначение типа данных и параметр разновидности	Точность в десятичных цифрах	Машинное эпсилон (точность)	Диапазон изменения
real(4)	7-8	10^{-7}	10^{-38}... 10^{+38}
real(8)	15-16	10^{-16}	10^{-308}... 10^{+308}
real(16)	33-34	10^{-34}	10^{-4932}... 10^{+4932}

Сведения о точности и диапазонах изменения данных *вещественного типа* с доступными параметрами разновидности (для *комплексного типа* значения аналогичны) приведены в табл. 3.3 (границы указаны приблизительно, с округлением; числа могут быть отрицательными, положительными и нулем).

Вычисления в Фортране инициируются с помощью *оператора присваивания* (знака равенства) в следующей общей форме:

```
variable_name = выражение
```

Работа оператора присваивания, определенного для всех стандартных типов данных, приводит к изменению значения переменной `variable_name`, которое будет возвращено в результате вычисления выражения. Следует обратить внимание, что знак равенства не означает здесь равенства в его математическом понимании (как запись уравнения).

Присвоение переменной начального значения называется *инициализацией*. Приведем примеры объявления данных числовых типов (с одновременной инициализацией и без нее):

```
integer :: a
integer(2) :: a = 2, b = 3
integer(8) a /2/, b /3/
real :: b
real b /1.1/                                  ! b = 1.10000002
real(8) :: c = 123.12_8              ! c = 123.12000000000000
complex(8) :: z
complex :: z = (-3.7, 1.)   ! действительная и мнимая части
```

В Фортране различают *именованные* и *буквальные константы*. Возможно использование арифметических, логических и символьных констант.

Пример объявления целых именованных констант с помощью атрибута **parameter** (после знака «_» здесь и далее указывается разновидность типа):

```
integer, parameter :: a = 1, b = -123_2, c = +284_4
```

Вещественные константы могут быть представлены в *F*- (с фиксированной точкой), *E*- или *D*-формах (с плавающей точкой). Примеры объявления вещественных констант в *F*-форме:

```
real(4), parameter :: a = 2.7_4, b = -.02
real(8), parameter :: c = -12.45643_8
```

Вещественные константы в *E*- и *D*-формах (только для двойной точности **real(8)**) имеют следующий вид:

```
± [мантисса] E или e ± порядок[_разновидность типа]
± [мантисса] D или d ± порядок
```

например:

```
real(8), parameter :: a= 1.83e12,   b= 18.3e+11_8,  c= 18.3d11
real(8), parameter :: a= -18.3e-05, b= -18.3e-05_8, c= -1.83d-4
```

Необходимо отметить, что наличие в секции объявления кода программы оператора **implicit none** (рекомендуется использовать в современных версиях) указывает на то, что все используемые в программной единице переменные и константы должны быть введены *явно* (с помощью операторов объявления типов данных). Присутствие в программе невведенных объектов данных будет приводить к возникновению ошибок на этапе компиляции, что убережет разработчика от опечаток и части ошибок.

Не рекомендуется применять исторически присутствующие в Фортране правила неявного объявления типа переменной или функции по первой букве имени (например, начинающиеся с `i`, `j`, `k`, `l`, `m`, `n` имеют целый тип).

3.5. Выражения, операции и арифметические вычисления

Операции языка Фортран разделяются на *встроенные* и *задаваемые* разработчиком (перегружаемые). К *встроенным* относятся операции следующих видов (описание операций приведено в прил. В): арифметические; символьная операция конкатенации (см. п. 3.7); операции отношения; логические.

Операции применяются для создания *выражений*, которые используются в операторах. Выражения могут оперировать со скалярами, массивами и их сечениями (см. гл. 7), а также вызовами функций.

Результатом арифметического выражения может быть величина целого, вещественного, комплексного типа или массив одного из этих типов. Операндами *арифметического выражения* могут быть арифметические константы, скалярные числовые переменные, числовые массивы и их сечения, а также вызовы функций числовых типов данных.

Арифметические операции имеют разный приоритет при совместном использовании (приведены далее по мере уменьшения приоритета): возведение в степень; умножение и деление; одноместные (унарные) плюс/минус (смена знака переменной); сложение и вычитание.

Все арифметические операции, за исключением возведения в степень, выполняются *слева направо* в соответствии с приоритетом. Операции возведения в степень выполняются *справа налево*. Заключенные в круглые скобки подвыражения вычисляются в первую очередь.

Примеры записи арифметических операций с использованием оператора присваивания и их результаты:

```
integer :: k
k = 2 * 2**2 / 2 / 2      ! 2
k = ((2* (2**2))/2) / 2   ! 2
k = 2**8 / 2 + 2          ! 130
k = 2**(8 / 2 + 2)        ! 64
k = 2**(8 / (2 + 2))      ! 4
```

В арифметических выражениях запрещается делить на нуль, возводить равный нулю операнд в отрицательную или нулевую степень, возводить отрицательный операнд в нецелочисленную степень.

Целочисленное деление производится в том случае, если операция деления выполняется над целыми числами. Примеры выполнения такой операции и соответствующие результаты:

```
real :: k
k = 1/2                  ! 0
k = -4/3                 ! -1
k = 1/4 + 1/4 + 1/4      ! 0
k = 2 / (3/4)            ! ошибка, деление на 0
```

3.6. Операции с разными типами данных

В Фортране допускается использование в арифметических выражениях операндов разных типов и их разновидностей, результат выполнения арифметической операции определяется при этом по следующим правилам:

– если операнды имеют одинаковый тип, результат операции имеет тот же тип;

– если операнды имеют разный тип, результат операции имеет тип операнда наивысшего ранга.

Ранг типов арифметических операндов (в порядке его убывания):

```
complex(16), complex(8), complex(4)
real(16), real(8), real(4)
integer(8), integer(4), integer(2), integer(1)
```

В ряде случаев в выражениях (например, вещественного типа с целочисленными операндами) для того, чтобы избежать целочисленного деления, следует выполнить явное преобразование типов данных с учетом разновидности типа (см. описание встроенных функций в прил. Г):

```
integer(2) :: a = 8, b = 3
real :: c
c = 2. ** (a / b)                    ! 4.00000000
c = 2. ** (real(a) / real(b))        ! 6.34960461
a = int2(c)                          ! 6
```

При преобразованиях разновидностей типов возможно искажение точности. Например, при преобразовании меньшей разновидности к большей:

```
real(4) :: a = 1.23_4              ! 1.2300000
real(8), parameter :: b = 1.23_4   ! 1.2300000190734863
real(8) :: c, d
c = a            ! 1.2300000190734863 (незначительное искажение)
d = 1.23_8       ! 1.2300000000000000
```

или, наоборот, большей разновидности к меньшей:

```
integer(2) :: i2 = 321
integer(1) :: i1                              ! -128 <= i1 <= 127
i1 = i2              ! i1 = 65  i2 = 321 (исчезают биты данных!)
```

Арифметические выражения с вещественными операндами вычисляются неточно, с возникновением ошибок округления. Вещественные числа не рекомендуется сравнивать на предмет точного равенства (неравенства), выполнять такие операции следует с указанием заданной точности. Влияние ошибок можно снизить, правильно выстраивая порядок вычислений:

```
real(4) :: x = 1.0e+30, y = -1.0e+30, z = 5.
write(*,*) (x + y) + z      ! 5.000000
write(*,*) x + (y + z)      ! 0.000000E+00
```

3.7. Символьные объекты данных

Символьный тип данных в Фортране могут иметь переменные и константы, которые принято называть строками, а также массивы и функции.

Символьные объекты данных объявляются оператором `character`:

```
character(len = 15) :: s1 = 'example'
character(len = *), parameter :: s2 = 'exam'
```

Параметр `len` задает длину объекта данных, а его значение может быть звездочкой (`*`), целой константой в диапазоне значений `1... 32767` или целочисленным константным выражением, вычисляемым со значением в том же диапазоне. Длина строки `s2` в примере равна числу символов именованной константы. Под хранение одного символа латинского алфавита отводится 1 байт, кириллицы – 2 байта.

В Фортране можно выделить из строки любую подстроку с помощью оператора следующего вида (`first` и `last` – арифметические выражения целого или вещественного типа):

```
s([first]: [last])
```

Выражение first задает первый символ в подстроке (по умолчанию `1`; если не задано, то подстрока начинается с первого символа

строки s), а last – последний символ в подстроке (по умолчанию равно длине строки; если не задано, то подстрока оканчивается последним символом строки s).

Примеры выделения подстроки:

```
a = 'abcdefghij';    b = '12345678'
c = a(5:7); b(7:8) = a(2:3)   ! c = 'efg'  b = '123456bc'
```

Символьная *операция конкатенации* предполагает объединение двух или более строк (или подстрок) в одну строку и обозначается как //. Длина результирующей строки равна при этом сумме длин строк-операндов:

```
a = 'abcdefghij'; b = '12345678'
c = a(1:3) // b(4:5) // a(6:8)        ! c = 'abc45fgh'
```

3.8. Встроенные функции

В научных и инженерных вычислениях часто задействуются функции, более сложные по сравнению с простейшими арифметическими операциями. Некоторые из функций широко распространены и используются во многих приложениях, другие встречаются реже и специфичны для какой-то одной задачи или класса задач. Распространенными примерами являются тригонометрические функции, логарифмы, квадратный корень и т. д.

Фортран имеет механизмы для поддержки широкого набора функций, погруженных непосредственно в стандарт языка (так называемые *встроенные функции*). Встроенная функция (наравне с внешними и внутренними, см. гл. 8) принимает одно или несколько входных значений и вычисляет на их основе единственное выходное значение.

Входные значения называются *аргументами* и задаются в круглых скобках после имени функции. Аргументом могут быть константа, переменная, массив или его сечение, выражение или результат другой функции. Выходное значение – число, логическое значение или строка символов, которые можно использовать вместе с другими функциями, константами и переменными в выражениях. Реализуемая пользователем функция может возвращать более сложный объект – элемент *производного типа данных*.

Встроенные функции вызываются посредством их явного именования в выражении, например, функция `asin` позволяет вычислить арксинус заданного числа (x – аргумент, y – выходное значение):

```
y = asin(x)
```

Среди других функций можно упомянуть числовые функции преобразования типов, округления; математические функции (модуль, квадратный корень, натуральный и десятичный логарифмы, экспоненциальная функция); функции работы с массивами (размер, максимальные/минимальные значения, скалярное произведение, транспонирование, умножение матриц); функции работы с символьными строками; функции времени, случайных чисел и др.

Встроенные функции могут быть *универсальными*, позволяющими использовать более одного типа входных данных, или *специальными*, требующими конкретного типа входных данных и создающими один тип выходного значения.

Некоторые встроенные функции с разделением на подгруппы и их краткие описания приведены в прил. Г.

3.9. Простой ввод/вывод

Стандартные средства языка Фортран поддерживают четыре вида *ввода/вывода*: управляемый списком, форматный, неформатный и двоичный. Введем сначала в рассмотрение наиболее простой вид – *управляемый списком*, для которого формат ввода и вывода определяют типы задействованных в них данных. Другие виды будут рассмотрены позднее.

Оператор ввода считывает одно или несколько значений со стандартного устройства ввода и сохраняет их в переменных, указанных в коде программы. Устройством ввода может быть клавиатура (в интерактивном режиме) или файл во внешней памяти (в пакетном режиме). Оператор вывода записывает одно или несколько значений в стандартное устройство вывода, которым может быть экран монитора, файл, принтер и т. п.

Оператор ввода (чтения) для управляемого списком действия имеет следующий вид (`список_ввода` – список разделенных

запятыми или запятыми с пробелами переменных, в которые помещаются считываемые значения):

```
read(*,*) список_ввода
```

Круглые скобки содержат управляющую информацию для чтения. Первое поле в скобках задает устройство ввода, из которого должны быть считаны данные (см. пояснения в гл. 6). Звездочка в этом поле означает, что данные необходимо считывать со стандартного устройства ввода (обычно с клавиатуры). Второе поле в скобках определяет формат, в котором данные должны быть считаны, а звездочка указывает на необходимость использования вида ввода, – управляемый списком.

Пример записи оператора ввода:

```
integer :: i, j
real :: a
character(len=12) :: s
read(*,*) i, j, a, s    ! считываются значения разных типов
```

Оператор вывода (записи) может иметь две эквивалентные формы (далее будет использоваться рекомендуемая форма `write`):

```
write(*,*) список_вывода
print *,   список_вывода
```

В данном случае `список_вывода` – это список элементов данных (переменных, констант или выражений), которые требуется вывести. Круглые скобки содержат управляющую информацию для вывода, а звездочки играют те же роли, что и в операторе ввода.

Пример записи операторов вывода и результаты выполнения (оценка числа π с точностью до 15 знаков после точки: 3.141592653589793):

```
integer :: i = 1
real(8), parameter :: pi1 = acos(-1.)      ! снижение точности
real(8), parameter :: pi2 = acos(-1.d0)    ! сохранение точности
write(*,*) ' i   = ', i
write(*,*) ' pi1 = ', pi1, ' cos(pi1) = ', cos(pi1)
write(*,*) ' pi2 = ', pi2, ' cos(pi2) = ', cos(pi2)
write(*,*) real(i), nint(pi1), nint(pi2)  ! преобразование типов
  i =            1
  pi1 = 3.141592 7410125732    cos(pi1) =  -0.99999999999999623
  pi2 = 3.141592 6535897931    cos(pi2) =  -1.00000000000000000
  1.00000000             3           3
```

3.10. Первые примеры программ

Напомним, что вопросы написания исходного кода, компиляции и запуска программ, применения специальных программных инструментов на разных платформах подробно рассмотрены в п. 1.5. Приводимые далее примеры кодов программ реализуют линейные алгоритмы.

Напишем сначала простейшую *программу-приветствие* на Фортране с комментариями непосредственно в коде:

```
program hello_world                ! начало программы и ее имя
! Программа-приветствие на языке Фортран
  write(*,*) 'Привет, мир!'                ! печать приветствия
end program hello_world                      ! конец программы
```

Здесь и далее предполагается использование стандарта кодирования UTF-8, обеспечивающего корректную запись кириллических символов.

Выполним компиляцию и запуск этой программы на исполнение в режиме командной строки при работе в Unix-подобной ОС:

```
gfortran program.f90
./a.out
Привет, мир!
```

Программирование формул – востребованный на практике пример линейного алгоритма. Напишем и прокомментируем программу пересчета вводимого значения температуры в градусах Цельсия в градусы Фаренгейта (псевдокод и блок-схема алгоритма приведены в п. 2.2).

```
program Celsius2Fahrenheit
! Программа перевода градусов Цельсия в градусы Фаренгейта
implicit none      ! необходимость явного объявления переменных
real :: T_c, T_f                     ! переменные для температур
  write(*,*) 'Введите температуру в градусах Цельсия:'
  read(*,*)  T_c    ! считывание температуры в градусах Цельсия
  T_f = 32. + 9. * T_c / 5.               ! выполнение пересчета
  write(*,*) 'Tc =', T_c, 'Tf =', T_f       ! вывод результатов
end program Celsius2Fahrenheit
```

В следующем примере приведен код программы *решения квадратного уравнения* с коэффициентами `a`, `b` и `c`: `ax2 + bx + c = 0`.

```
program quad_equation_explicit
! Программа решения квадратного уравнения
! с явным заданием коэффициентов
implicit none
real :: a, b, c, a2    ! переменные для коэффициентов уравнения
complex :: sqd, x1, x2        ! переменные для корней уравнения
  a   = 4.; b = 2.; c = 1.  ! значения коэффициентов уравнения
  a2  = 2 * a
  sqd = sqrt(cmplx(b**2 - 4 * a * c))  ! корень из дискриминанта
  x1 = (-b + sqd) / a2; x2 = (-b - sqd) / a2   ! корни уравнения
  write(*,*) 'Корни уравнения: x1 = ', x1, 'x2 = ', x2
end program quad_equation_explicit
```

Значения коэффициентов задаются явно с помощью операторов присваивания и подобраны таким образом, чтобы уравнение имело два комплексных корня (см. другую версию кода в п. 4.4).

Последний пример относится к работе с символьными объектами данных – приведенная далее программа подсчитывает количество символов во вводимом пользователем имени:

```
program str_length
! Программа подсчета количества символов во вводимом имени
implicit none
character(len=28) :: fname
  write(*,*) 'Введите имя:'; read(*,*) fname
  write(*,*) 'Количество символов в имени:'
  write(*,*) len_trim(fname)/2        ! см. комментарий ниже
end program str_length
```

При использовании стандарта UTF-8 результат, возвращаемый встроенной функцией подсчета количества символов в строке (`len_trim`), нужно поделить на два, поскольку для хранения кириллических символов задействуются 2 байта, а не один, как для символов латинского алфавита.

3.11. Варианты заданий

Приводимые в конце этой и следующих глав данного учебного пособия задания для компьютерного практикума предполагают разработку программ, выполняющих соответствующие вычисления. Некоторые задания к этой и следующей главам

заимствованы из учебника по программированию В. Д. Шелеста, в котором можно также найти дополнительные варианты заданий для математического блока.

3.11.1. Математический блок

Выполните алгебраические вычисления с использованием встроенных функций языка Фортран (`sqrt`, `abs`, `log`, `exp`, `sin`, `cos`).

2. Вычислите логарифм с произвольным основанием по формуле

$$\log_b(x)=\frac{\log_{10}(x)}{\log_{10}(b)}.$$

3. Вычислите гиперболический косинус, используя формулу

$$\cosh(x)=\frac{e^{x}+e^{-x}}{2}.$$

4. Вычислите гипотенузу прямоугольного треугольника и его площадь при заданных катетах.

5. Вычислите расстояние между двумя точками с координатами $(x_1,\ y_1)$ и (x_2, y_2), используя формулу

$$d=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}.$$

6. Определите площадь треугольника для заданных координат вершин.

7. Вычислите векторное произведение заданных векторов по формуле

$$\vec{a}\times\vec{b}=\begin{vmatrix}\vec{i} & \vec{j} & \vec{k}\\ a_x & a_y & a_z\\ b_x & b_y & b_z\end{vmatrix}=\vec{i}(a_yb_z-a_zb_y)+\vec{j}(a_zb_x-a_xb_z)+\vec{k}(a_xb_y-a_yb_x).$$

8. Вычислите число π, используя формулу Гаусса

$$\pi=48\,\mathrm{arctg}(1/18)+32\,\mathrm{arctg}(1/57)-20\,\mathrm{arctg}(1/239).$$

Результат сравните со значением π, полученным по формулам

$$\pi\approx 355/113 \text{ и } \pi\approx 16\,\mathrm{arctg}(1/5)-4\,\mathrm{arctg}(1/239).$$

9. Для чисел a, b, c найдите произведение p двух наибольших чисел, используя формулу

$$p = \frac{abc}{\min\left(\min(a,b),\min(b,c)\right)}, \text{ где } \min(u,v) = \left[(u+v) - |u-v|/2\right].$$

10. Вычислите факториал заданного числа n по формуле Стирлинга

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} - \frac{571}{2488320n^4} + \ldots\right).$$

Дополнительные варианты заданий можно найти в книге В. Д. Шелеста на с. 45–49.

3.11.2. Физический блок

1. Выполните конвертацию следующих единиц измерения:
а) величина угла в радианах и градусах, минутах и секундах;
б) градус Фаренгейта, градус Кельвина и градус Реомюра;
в) фунт и килограмм;
г) дюйм и сантиметр;
д) миля и километр;
е) кубический метр и литр;
ж) метр в секунду и километр в час;
з) Паскаль и фунт на квадратный дюйм.

2. Вычислите потенциальную и кинетическую энергии движущегося объекта, используя формулы $E_p = mgh$ и $Ek = 0.5mV^2$, где m – масса объекта; g – ускорение свободного падения; h – высота; V – скорость.

3. Вычислите полную энергию физического объекта по формуле Эйнштейна $E = mc^2$, где m – масса объекта; c – скорость света в вакууме.

4. Вычислите период колебаний маятника $T = 2\pi(L/g)^{0.5}$, где L – длина маятника; g – ускорение свободного падения.

5. Определите скорость отрыва объекта от поверхности планеты или спутника планеты, используя формулу $V_e = (2GM)^{0.5}/R$, где R – радиус; M – масса; G – гравитационная постоянная.

6. Вычислите центробежное ускорение, используя формулу $a = V^2/R$, где V – скорость; R – радиус.

7. Вычислите соотношения двух измерений мощности в децибелах по формуле $dB = 10 \cdot \log_{10}(P_2/P_1)$, где P_2 – измеряемый уровень мощности; P_1 – некоторый эталонный уровень.

8. Определите скорость V истечения жидкости из малого отверстия в открытом сосуде с заданной высотой жидкости h над отверстием, используя формулу Торричелли $V = (2gh)^{0.5}$, где g – ускорение свободного падения.

9. Вычислите следующие гидродинамические числа подобия для заданных значений, входящих в формулы величин (для воздуха и воды) [23]:

а) число Маха $M = V/a$, где a – скорость звука; V – скорость жидкости;

б) число Рейнольдса $Re = \rho VL/\mu$, где ρ – плотность; V – скорость жидкости; L – характерная длина; μ – динамический коэффициент вязкости.

10. Вычислите плотность жидкости ρ, используя уравнение Менделеева–Клапейрона $p/\rho = TR_{ун}/M_m$, где p – давление; ρ – плотность; T – температура; $R_{ун}$ – универсальная газовая постоянная; M_m – молярная масса.

11. Вычислите давление жидкости p на заданной глубине h по формуле $p = p_0 + \rho gh$, где p_0 – давление над поверхностью; ρ – плотность; g – ускорение свободного падения.

Глава 4
УСЛОВНЫЕ ОПЕРАТОРЫ И ВЕТВЛЕНИЯ

4.1. Логические константы, переменные и операторы

4.1.1. Логические константы, переменные и вычисления

Логический тип данных в языке Фортран содержит одно из двух возможных значений – истина (*true*) или ложь (*false*). Логическая константа может принимать одно из двух значений – `.true.` или

`.false.` (точки необходимы по обе стороны от значений, чтобы отличать их от имен переменных).

Логическая переменная содержит значение логического типа данных и указывается в секции объявления программы с помощью оператора **`logical`**:

```
logical :: f = .false.
```

Как и арифметические, логические вычисления выполняются с помощью оператора присваивания в следующей общей форме:

```
logical_variable_name = логическое_выражение
```

Выражение может представлять собой любую комбинацию допустимых логических констант, переменных и операций. Последние подразделяются на два вида – операции отношения и логические операции.

4.1.2. Операции отношения

Операции отношения представляют собой операции с двумя числовыми или символьными операндами, дающие логический результат. Общая форма записи операции отношения имеет вид:

a_1 `op` a_2,

где a_1 и a_2 – арифметические выражения, переменные, константы или строки; `op` – одна из операций отношения, перечисленных в табл. 4.1.

Таблица 4.1

Операции отношения и их запись в Фортране

Запись операции в Фортране		Значение
Новый стиль (введен в Фортран 90)	Старый стиль	
==	.eq.	Равно
/=	.ne.	Не равно
>	.gt.	Больше
<	.lt.	Меньше
>=	.ge.	Больше или равно
<=	.le.	Меньше или равно

Таблица 4.2

Примеры операций отношения

Операция	Результат
`4 <= 5`	`.true.`
`4 == 5`	`.false.`
`4 > 5`	`.false.`
`4 <= 4`	`.true.`
`'a' < 'b'`	`.true.`

В случае если выраженное операцией отношение между a_1 и a_2 истинно, будет возвращено значение `.true.`, иначе – значение `.false`.

Примеры операций отношения и результаты приведены в табл. 4.2.

Результат последней операции – `.true.`, поскольку символы сравниваются в алфавитном порядке. В сложных конструкциях операции отношения выполняются после арифметических. При сравнении вещественных и целочисленных значений последнее предварительно преобразуется в вещественное. Сравнение числовых и символьных данных недопустимо.

4.1.3. Логические операции

Логические операции – это операции с одним или двумя логическими операндами (утверждениями), дающие логический результат. Существуют три базовые логические операции связывания простых операций отношения в составные – *конъюнкция* (логическое И), *дизъюнкция* (логическое ИЛИ), *инверсия*, или отрицание (логическое НЕ), а также дополнительные – логические *эквивалентность* (*неэквивалентность*) и *импликация*.

Общая форма записи бинарной логической операции в Фортране:

f_1 `op` f_2,

где f_1 и f_2 – логические выражения, переменные или константы; op – одна из связывающих их логических операций, перечисленных в табл. 4.3.

Точки являются частью операции и обязательно должны присутствовать. В случае если выраженное операцией отношение между f_1 и f_2 истинно, будет возвращено значение `.true.`, иначе – значение `.false.`

Результаты выполнения операций для всех возможных комбинаций f_1 и f_2 сведены в *таблицы истинности* – в табл. 4.4, 4.5.

Таблица 4.3

Логические операции

Операция	Функция	Определение
f_1 `.and.` f_2	Конъюнкция	Результат – `.true.` тогда и только тогда, когда f_1 и f_2 – .true.
f_1 `.or.` f_2	Дизъюнкция	Результат – `.true.` тогда и только тогда, когда хотя бы один f_1 или f_2 – `.true.`
f_1 `.eqv.` f_2	Эквивалентность	Результат – `.true.`, если f_1 имеет ту же таблицу истинности, что и f_2
f_1 `.neqv.` f_2	Неэквивалентность	Отрицание эквивалентности
`.not.` f_1	Инверсия	Результат – `.true.`, если f_1 – `.false.`, и результат – `.false.`, если f_1 – `.true.`

Таблица 4.4

Таблица истинности для бинарных логических операций

f_1	f_2	f_1 `.and.` f_2	f_1 `.or.` f_2	f_1 `.eqv.` f_2	f_1 `.neqv.` f_2
`.false.`	`.false.`	`.false.`	`.false.`	`.true.`	`.false.`
`.false.`	`.true.`	`.false.`	`.true.`	`.false.`	`.true.`
`.true.`	`.false.`	`.false.`	`.true.`	`.false.`	`.true.`
`.true.`	`.true.`	`.true.`	`.true.`	`.true.`	`.false.`

Таблица 4.5

Таблица истинности для унарной логической операции

№ п/п	f	.not. f
1	.false.	.true.
2	.true.	.false.

Таблица 4.6

Примеры логических операций

№ п/п	Операция	Результат
1	`.not. f1`	`.false.`
2	`f1 .or. f3`	`.true.`
3	`f1 .and. f3`	`.false.`
4	`f2 .neqv. f3`	`.true.`
5	`f1 .and. f2 .or. f3`	`.true.`
6	`f1 .or. f2 .and. f3`	`.true.`
7	`.not. (f1 .eqv. f2)`	`.false.`

В сложных конструкциях логические операции выполняются после того, как выполнены все арифметические операции и операции отношения.

Введем в рассмотрение еще одну логическую операцию – *импликацию*, которая связывает два утверждения отношением логического следования «`если а, то б`». Импликация истинна всегда, кроме случая, когда утверждение `а` истинно, а утверждение `б` – ложно.

Существует определенная последовательность выполнения логических операций при преобразовании сложных утверждений (если она не задана явно наличием скобок). Общий порядок по убыванию приоритета – инверсия, конъюнкция, дизъюнкция, импликация и эквивалентность (для бинарных операций выполнение происходит слева направо).

Примеры логических операций и их результаты приведены в табл. 4.6 (значения переменных f_1 `= .true.`, f_2 `= .true.` и f_3 `= .false.`).

4.2. Операторы ветвления

Ветвления в большинстве языков программирования реализуются с помощью условного оператора `if` (и связанных с ним *операторов ветвления*), указывающего на то, что соответствующий блок операторов и конструкций (<БОК>) будет выполнен тогда и только тогда, когда сопутствующее оператору логическое выражение (ЛВ) истинно.

Конструкции ветвлений вида «если–то», «если–то–иначе» и многозначного ветвления «если–то–иначе–если» записываются в Фортране так, как показано в табл. 4.7.

В ветвлении вида «если – то», если ЛВ – истинно, будут выполнены операторы в блоке <БОК 1> между операторами **if** и **end if**, в противном случае все операторы в блоке между этими операторами будут пропущены и программа перейдет к выполнению операторов в блоке <БОК 2>, следующем после оператора **end if**.

Сокращенная форма записи оператора ветвления (без **then** и **end if)** в одной строке возможна, если блок состоит из единственного исполняемого оператора (за исключением **if**, **else**, **else if**, **end if**, **do**, **end**).

В ветвлении вида «если–то–иначе», если ЛВ – истинно, будут выполнены операторы в блоке <БОК 1>, в противном случае – операторы в блоке <БОК 2>, в любом случае после выполнения того или иного блока программа перейдет к выполнению операторов в блоке <БОК 3>.

В *многозначном ветвлении* (см. блок-схему на рис. 4.1), если ЛВ 1 – истинно, будут, как и ранее, выполнены операторы в блоке <БОК 1>, если ЛВ 1 – ложно, а ЛВ 2 – истинно, то будут выполнены операторы в блоке <БОК 2>. Если оба логических выражения ЛВ 1 и ЛВ 2 – ложны, будут выполнены операторы в блоке <БОК 3>.

Таблица 4.7

Запись ветвлений в языке Фортран

Ветвление вида «если–то»	Ветвление вида «если–то–иначе»	Многозначное ветвление

if (ЛВ) **then**
  <БОК 1>
end if
<БОК 2>
 	
if (ЛВ) оператор
	if (ЛВ) **then**
	  <БОК 1>
	else
	  <БОК 2>
	end if
	<БОК 3>
		if (ЛВ 1) **then**
		  <БОК 1>
		else if (ЛВ 2)
		then
		  <БОК 2>
		else
		  <БОК 3>
		end if
		<БОК 4>

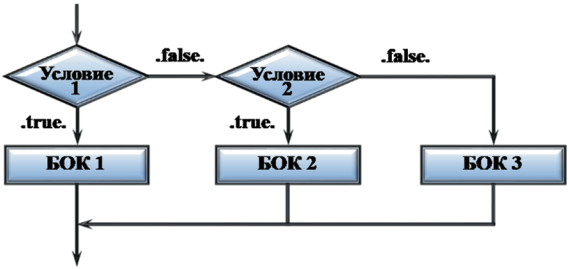

Рис. 4.1. Блок-схема операции многозначного ветвления

После выполнения того или иного из трех блоков программа перейдет к выполнению операторов в блоке <БОК 4>.

Пример фрагмента кода с использованием многозначного ветвления, в котором определяется знак вводимого целого числа:

```
write(*,*)   'Введите целое число:'; read(*,*) i
if (i > 0) then
  write(*,*) 'Введенное число - положительное'
else if (i < 0) then
  write(*,*) 'Введенное число - отрицательное'
else
  write(*,*) 'Введенное число равно нулю'
end if
```

4.3. Оператор многозначного выбора

Оператор многозначного выбора – форма конструкции ветвления, позволяющая выбирать конкретный блок кода для выполнения на основе значения одной константы или списка нескольких. Общая форма записи следующая:

```
select case (тест_выражение)
case (список_констант 1)
  <БОК 1>
case (список_констант 2)
  <БОК 2>
...
```

```
case (список_констант 2)
  <БОК N>
case default
  <БОК D>
end select
```

В данном случае `тест_выражение` – целочисленное, символьное или логическое скалярное выражение, `список_констант` – список констант, тип которых должен соответствовать типу `тест_выражение`.

Работа конструкции **`select case`** (см. блок-схему на рис. 4.2) стартует с вычисления значения `тест_выражение`, после чего выполняются последовательные проверки наличия этого значения в списках констант в круглых скобках после операторов выбора **case** (`<БОК 1>`, `<БОК 2>`, … `<БОК N>`). Нахождение значения в одном из списков приводит к выполнению соответствующего блока операторов и последующего перехода к первому оператору, расположенному после конструкции **`end select`**.

В случае если значение `тест_выражение` не найдено ни в одном из обозначенных списков и присутствует оператор **`case default,`** будет выполнен блок операторов `<БОК D>`, а затем – переход к первому оператору после **`end select`**. Если значение `тест_выражение` не найдено ни в одном из списков и **`case default`** отсутствует, то ни один из блоков операторов выполнен не будет, управление будет передано на следующий за **`end select`** оператор.

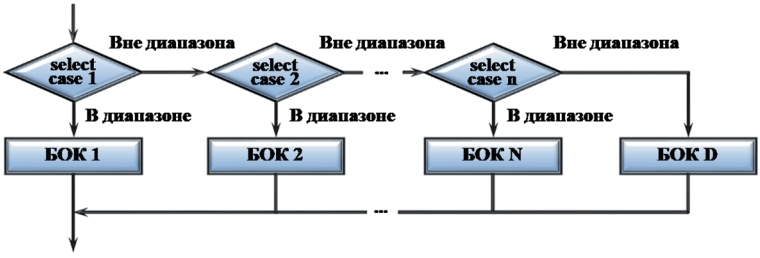

Рис. 4.2. Блок-схема операции многозначного выбора

Список констант может содержать одно значение, состоять из разделенного запятыми набора констант, может быть задан как диапазон разделенных двоеточием значений. В последнем случае нижняя граница должна быть меньше верхней (например, `10:15`), а для диапазона символов код первого символа должен быть меньше кода второго. При отсутствии нижней границы (например, `:15`) считается, что в списке констант содержатся все значения, меньшие или равные верхней границе. Наоборот, если опущена верхняя граница (например, `10:`), то считается, что в список констант попадают все значения, большие или равные нижней границе. Список может включать также смесь отдельных значений и диапазонов, разделенных запятыми, например:

```
case (1, 5, 10:15, 25)
```

Приведем пример фрагмента кода с использованием многозначного выбора, в котором определяется знак задаваемого целого числа (см. вариант с использованием операторов ветвления в п. 4.2):

```
write(*,*) 'Введите целое число:';  read(*,*) i
select case (i)
case (1:)
  write(*,*) 'Введенное число - положительное'
case (:-1)
  write(*,*) 'Введенное число - отрицательное'
case default
  write(*,*) 'Введенное число равно нулю'
end select
```

4.4. Примеры программ

Напишем программу вычисления функции `f(x,y)` для любых двух задаваемых значений независимых переменных `x` и `y`:

$$f(x,y)=\begin{cases} x+y\ , & x\geq 0, y\geq 0, \\ x+y^2\ , & x\geq 0, y<0, \\ x^2+y\ , & x<0, y\geq 0, \\ x^2+y^2, & x<0, y<0. \end{cases}$$

```
program f_xy
! Программа вычисления значения функции двух переменных
implicit none
real :: x, y, f    ! независимые переменные и результат вычисления
  write(*,*) 'Введите значения переменных:'; read(*,*) x, y
! Вычисление значения функции в зависимости от знаков переменных
  if ( (x >= 0.) .and. (y >= 0.) ) then
    f = x + y
  else if ( (x >= 0.) .and. (y < 0.) ) then
    f = x + y**2
  else if ( (x < 0.) .and. (y >= 0.) ) then
    f = x**2 + y
  else
    f = x**2 + y**2
  end if
  write(*,*) 'Значение функции: ', f
end program f_xy
```

Приведем версию программы нахождения корней квадратного уравнения с произвольно вводимыми коэффициентами и с использованием условных операторов (см. вариант кода в п. 3.10):

```
program quad_equation_general
! Программа решения квадратного уравнения в общем виде
implicit none
real :: a, b, c          ! переменные для коэффициентов уравнения
real :: discr            ! переменная для дискриминанта уравнения
real :: imag_part, real_part  ! вещественная и целая части корней
real :: x1, x2        ! корни уравнения (для вещественных корней)
  write(*,*) 'Введите коэффициенты a, b и c:'
  read(*,*)   a, b, c
  discr = b**2 - 4 * a * c              ! вычисление дискриминанта
! Нахождение корней в зависимости от значения дискриминанта
  if ( discr > 0 ) then
    x1 = ( -b + sqrt(discr) ) / (2 * a)
    x2 = ( -b - sqrt(discr) ) / (2 * a)
    write(*,*) 'Уравнение имеет два вещественных корня:'
    write(*,*) 'x1 = ', x1, 'x2 = ', x2
  else if ( discr == 0 ) then
    x1 = (-b) / (2 * a)
    write(*,*) 'Уравнение имеет два одинаковых веществ. корня:'
    write(*,*) 'x1 = x2 = ', x1
  else
    real_part = (-b) / (2 * a)
    imag_part = sqrt (abs(discr)) / (2 * a)
    write(*,*) 'Уравнение имеет два комплексных корня:›
    write(*,*) 'x1 = ', real_part, ' +i ', imag_part
    write(*,*) 'x2 = ', real_part, ' -i ', imag_part
  end if
end program quad_equation_general
```

4.5. Варианты заданий

4.5.1. Математический блок

1. Вычислите значения следующих функций:

$$\text{а) } f(x)=\begin{cases} x^3, & x<0, \\ x^2, & 0\le x\le 3, \\ 9, & x>3; \end{cases} \qquad \text{б) } f(x)=\begin{cases} \sin(x), & x<0, \\ \cos(x)-1, & x\ge 0. \end{cases}$$

2. Вычислите значения полиномов Лагерра $L_n(x)$ для заданных x и n:

$$L_n(x)=\begin{cases} 1, & n=0; \\ -x+1, & n=1; \\ x^2-4x+2, & n=2; \\ -x^3+9x^2-18x+6, & n=3; \\ x^4-16x^3+72x^2-96x+24, & n=4. \end{cases}$$

3. Определите, попадает ли заданная точка A с координатами (x_A, y_A, z_A) в область, ограниченную:

а) окружностью $x^2 + y^2 = a^2$, где a – заданный радиус круга ($z_A = 0$);

б) конусом $x^2/a^2 + y^2/b^2 = z^2/c^2$.

4. По заданным сторонам треугольника a, b, c найдите его площадь S и радиусы r_1 описанной и r_2 вписанной окружностей.

5. Пусть a, b, c, d – стороны четырехугольника. Вычислите площадь четырехугольника S, используя формулу:

$$S=\sqrt{(p-a)(p-b)(p-c)(p-d)},\ p=(a+b+c+d)/2.$$

Выясните, можно ли в четырехугольник вписать окружность и можно ли около него описать окружность.

6. Определите, является заданное целое число четным или нечетным.

7. Присвойте переменной R единицу, если ближайшее к значению x целое число – четное и отличное от нуля, в противном случае R равно нулю.

8. Присвойте переменной R единицу, если переменная n делится нацело на m или на k, или при ее делении на n получается остаток, кратный m или k, в противном случае R равно нулю.

9. Определите, является ли заданный год високосным.

10. Осуществите перевод заданного числа из десятичной системы счисления в двоичную.

Дополнительные варианты заданий можно найти в книге В. Д. Шелеста на с. 58–62.

4.5.2. Физический блок

1. К баку подсоединены краны с различной формой среза – квадрат со стороной h, круг диаметра h и равносторонний треугольник со стороной h. Определить среднюю скорость движения жидкости V_{cp} на выходе из каждого крана, если за время t вытекает заданный объем жидкости Vol. Общий объемный расход определяется по формуле Q = Vol/t; расход через каждый кран $Q_i = V_{cp}S_i$, где S_i – площадь i-го сечения (i = 1, n). В программу добавить возможность выбора количества и типа для рабочих кранов.

2. Распределения температуры T, плотности ρ и давления p в стандартной атмосфере Земли, состоящей из двух слоев совершенного газа – нижнего, в котором температура убывает по линейному закону (тропосфера), и верхнего с постоянной T (стратосфера), можно рассчитать по следующим формулам:

Тропосфера ($0 < h < H_1$)

$$\begin{cases} T = T_0\left(1+\left(T_1/T_0-1\right)h/H_1\right), & E_1 = -H_1T_0/\left[H^*\left(T_1-T_0\right)\right]-1, \\ \rho = \rho_0\left(1+\left(T_1/T_0-1\right)h/H_1\right)^{E_1}, & E_2 = -H_1T_0/\left[H^*\left(T_1-T_0\right)\right], \\ p = p_0\left(1+\left(T_1/T_0-1\right)h/H_1\right)^{E_2}, & H^* = \rho_0/\rho_0 g. \end{cases}$$

Стратосфера ($H_1 < h$)

$$\begin{cases} T = T_1, \\ \rho = \rho_1 e^{E_1}; \quad E = -(h-H_1)\rho_1 g/p_1, \\ p = p_1 e^{E_2}. \end{cases}$$

Высота атмосферы варьируется в диапазоне 0…25 км. Значения на поверхности Земли: температура $T_0 = 15$ °C; давление $p_0 = 105$ Па; плотность $\rho_0 = 1.3$ кг/м^3; на границе тропосферы и стратосферы ($H_1 = 11$ км): $T_1 = 56.5$ °C, $p_1 = 2.27 \cdot 10^4$ Па, $\rho_1 = 0.36$ кг/м^3.

а) рассчитайте давление и плотность в стандартной атмосфере Земли;

б) рассчитайте температуру в стандартной атмосфере Земли.

3. Рассматривается установившееся течение вязкой жидкости в круглой трубке диаметром $D = 1$ см. В качестве жидкости рассмотреть воздух (плотность $\rho = 1.2$ кг/м^3; динамический коэффициент вязкости $\mu = 1.8 \times 10^{-5}$ кг/мс) и воду ($\rho = 10^3$ кг/м^3; $\mu = 8.9 \times 10^{-4}$ кг/мс). Напишите программу, определяющую, при каких значениях скорости V течение жидкости ламинарное, а при каких – турбулентное. Критическое число Рейнольдса, при котором происходит переход от ламинарного режима течения к турбулентному, $Re_{кр} = \rho VD/\mu \approx 2300$.

4. Самолет движется в воздухе при следующих условиях: температура $T = 223$ K; показатель адиабаты $k = 1.4$; молярная масса $Mol = 29$ г/моль; универсальная газовая постоянная $R_{ун} = 8.31$ Дж/(моль · K). Определите, при каких значениях скорости V движение воздуха дозвуковое ($M < 1$), звуковое ($M = 1$) и сверхзвуковое ($M > 1$); число Маха $M = V/a$, где $a = (kRунT/Mol)0.5$ – скорость звука.

5. Рассматривается установление течения вязкой жидкости (рассмотреть воздух, воду, мед) в следующих конструкциях:

а) плоский щелевой канал высоты h;

б) цилиндрическая труба диаметра h.

На входной границе задано однородное распределение скорости $V_{ср}$. Определите длину канала $L_н$, на котором течение устанавливается, если для ламинарного течения в плоском канале $L_н = 0.04\ Re$, для круглой трубы $L_н = 0.06Re$. Задано число Рейнольдса $Re = \rho Vcph/\mu$, где ρ – плотность; μ – динамический коэффициент вязкости.

6. Рычажные весы имеют четыре неравноплечных рычага, свободно качающихся на опоре; длины плеч равны l, l_1, l_2 и l_3

соответственно. На одном из плеч рычага устанавливается груз массой m. Определите массу груза, если система уравновешена одним, двумя или тремя рычагами, на которых установлены грузики заданной массы. Выражение для определения массы груза определяется в соответствии с законом сохранения моментов:

$$mgl = \sum_{i=1}^{3} m_i g l_i.$$

Глава 5
ЦИКЛЫ

5.1. Арифметические циклы

Напомним, что *циклические алгоритмы* относятся к типу *нелинейных* и являются одним из важных и наиболее ресурсоемких элементов вычислительных программ. В цикле многократно выполняется связанный блок операторов и конструкций, при этом количество повторений может быть либо предопределено заранее (*арифметические циклы*), либо неизвестно с обязательной формулировкой условия выхода (*итеративные циклы*).

Арифметический цикл (цикл с параметром, перечисляемый цикл) используется для повторения блока <БОК> заранее определенное количество раз (шагов) и записывается в Фортране в следующем общем виде:

```
do i = istart, iend [, incr]
   <БОК>
end do
```

В этой записи присутствует окружающая блок операторная конструкция **do** … **end do** и определены параметры: i (индекс, параметр или шаг цикла, целая или вещественная переменная; второй вариант не рекомендуется); istart (начальное значение индекса), iend (конечное значение индекса) и incr (приращение индекса цикла) – параметры, контролирующие значения индекса во время выполнения цикла. Они могут быть константами, переменными

или скалярными выражениями (переменные и выражения вычисляются перед началом цикла). Параметр `incr` является необязательным, а его значение по умолчанию равно `1`.

В начале выполнения арифметического цикла происходит присваивание переменной `i` значения `istart`, и, если на текущем шаге выполняется условие `i ≤ iend`, будет исполнен блок внутри тела цикла.

В качестве примера приведем фрагмент программы вычисления функции факториала для заданного значения целого числа по формуле

$$\mathtt{fact}(\mathtt{n}) = \begin{cases} 1, & \mathtt{i}=0, \\ \mathtt{i}\times\mathtt{fact}(\mathtt{n}-1), & \mathtt{i}>0. \end{cases}$$

```
integer, parameter :: n = 10
integer :: i, fact
  fact = 1     ! важно инициализовать единицей произведения
  do i = 1, n                          ! вычислительный цикл
    fact = fact * i                     ! результат: 3628800
  end do
```

Внутри тела цикла могут присутствовать дополнительные управляющие операторы:

– **`cycle`** – *оператор продолжения*; выполнение текущего шага цикла останавливается, и управление передается следующему шагу цикла;

– **`exit`** – *оператор прерывания*; управление передается из тела цикла первому следующему после него оператору (фактически выход из цикла).

Приведем фрагменты программ, печатающих четные положительные числа, значения которых не превышают `10`, с использованием заданного приращения индекса цикла, операторов **`cycle`** и **`exit`**.

```
  do i = 2, 10, 2
    write(*,*) i
  end do
  do i = 10, 2, -2
    write(*,*) i
  end do
  do i = 1, 10
```

```
  if (mod(i,2) /= 0) cycle ! mod возвращает остаток от деления
   write(*,*) i
end do
do i = 2, 20, 2
  if (i > 10) exit
  write(*,*) i
end do
```

Циклы могут иметь сложную структуру и состоять из нескольких вкладывающихся друг в друга циклов. Такой цикл называется кратным, окаймляющий – внешним, а вложенные в него – внутренними.

5.2. Итеративные циклы

В языке Фортран определены два вида итеративных циклов – цикл «пока» и цикл «до», общая форма записи которых приведена в табл. 5.1.

В *цикле «пока»*, если связанное логическое выражение (ЛВ) истинно, будет исполнен <*БОК*>, а затем управление вернется к оператору **do while** с повторением этого процесса до тех пор, пока ЛВ не станет ложным, после чего будет осуществлен переход к первому оператору после оператора **end do**.

В *цикле «до»* исполнение блоков операторов в теле цикла будет повторяться, пока ЛВ не станет истинным и не будет выполнен оператор **exit** с передачей управления первому оператору после **end do**.

Очевидно, что у двух видов итеративных циклов отличается очередность проверки условия выхода – до или после выполнения <БОК> (предусловие или постусловие, хотя в общем случае цикл «до» может содержать проверку условия в любом месте, даже в начале).

Таблица 5.1

Запись итеративных циклов

№ п/п	Запись цикла «пока»	Запись цикла «до»
1	**do while** (ЛВ) <БОК> **end do**	**do** <БОК> **if** (ЛВ) **exit** **end do**

5.3. Именование циклов

В языке Фортран циклам всех видов можно присваивать имена:

```
[имя:] do index = istart, iend [, incr]
          <БОК>
end do [имя]
[имя:] do while (ЛВ)
          <БОК>
end do [имя]
[имя:] do
          <БОК>
          if (ЛВ) exit [имя]
end do [имя]
```

Имя цикла (параметр `[имя]`) должно быть уникальным в пределах каждой программной единицы и указано в операторе **`end do`**. Имена не являются обязательными для расположенных в теле цикла операторов **`cycle`** и **`exit`**, но при явном использовании должны совпадать с именем в операторе **`do`**.

Основная мотивация для именования циклов — помочь компилятору (и разработчику) идентифицировать циклы и связанные операторы в случаях, когда они становятся сложными. Присвоение имен частям цикла позволяет легко определить, к какой конструкции принадлежит конкретный оператор **`end do`**, **`cycle`** или **`exit`**. Кроме того, имена конструкций могут помочь компилятору (отладчику) локализовать места возникновения ошибок.

5.4. Примеры программ

В качестве первого примера приведем код программы, вычисляющей *сумму первых* `n` *натуральных чисел* в арифметическом цикле:

```
program sum_natural
! Программа вычисления суммы первых n натуральных чисел
implicit none
integer, parameter :: n = 10
integer :: i, s             ! индекс цикла и переменная суммы
  s = 0             ! важно обнулять суммы перед вычислениями
  do i = 1, n                              ! вычислительный цикл
```

```
    s = s + i
  end do
end program sum_natural
```

Приведем код программы приближенного *вычисления квадратного корня методом* Ньютона для задаваемого вещественного числа a с использованием арифметического цикла по формуле (x_i при i, стремящемся к бесконечности, сходится к значению $a^{0.5}$):

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right),\ i = 0,\ 1,\ldots;\ x_0 = 1.$$

```
program squre_root_Newton
! Приближенное вычисление квадратного корня методом Ньютона
implicit none
integer(2), parameter :: n = 10              ! число шагов цикла
integer(2) :: i                                  ! индекс цикла
real :: a, x                          ! переменные для вычислений
  write(*,*) 'Введите неотрицательное число: '; read(*,*) a
  x = 1.
! Основной цикл приближенного вычисления квадратного корня
  do i = 1, n
    x = (x + a / x) / 2.   ! формула Ньютона
   write(*,*) x, sqrt(a) ! вывод приближения и точного значения
  end do
end program squre_root_Newton
```

Напишем программу, печатающую *таблицу соответствия между градусами Цельсия и Фаренгейта* в заданном диапазоне изменения температуры, который определяется начальным и конечным значениями индекса арифметического цикла:

```
program Celsius_2_Fahrenheit_table
! Печать таблицы соответствия градусов Цельсия и Фаренгейта
implicit none
integer(2), parameter :: n= 20   ! конечное значение индекса цикла
integer(2) :: i                                  ! индекс цикла
real :: T_c, T_f             ! переменные для значений температур
  write(*,*) 'Таблица соответствия между температурными шкалами›
! Цикл с пересчетом температуры и печатью значений
  do i = 0, n
    T_c = 5. * i
    T_f = 32. + 9. * T_c / 5.
    write(*,*) 'Tc =', T_c, 'Tf =', T_f
  end do
end program Celsius_2_Fahrenheit_table
```

Приведем пример использования кратных циклов, вычислив с их помощью следующую двойную сумму:

$$S=\sum_{i=1}^{m}\sum_{j=1}^{n}\frac{i+j}{i^2+j^2}.$$

```
program double_sum
! Программа вычисления суммы в кратном цикле
implicit none
integer(2), parameter :: m = 10, n = 20       ! пределы суммы
integer(2) :: i, j                            ! индексы цикла
real(8) :: s
  s = 0.
! Кратный цикл вычисления суммы
  do i = 1, m
    do j = 1, n
      s = s + real(i + j,8) / real(i**2 + j**2,8)
    end do
  end do
 write(*,*) 'Значение суммы: ', s          ! 27.0586037289158
end program double_sum
```

Табулирование функций – процесс составления таблиц значений функции для некоторого заданного диапазона изменения значений ее аргументов. Шаг продвижения по диапазону обычно выбирается постоянным. Приведем код программы с арифметическим циклом, в которой производится табулирование функции следующего вида:

$$f(x)=\begin{cases}-1, & x\le -1;\\ -x^2, & -1<x\le 0;\\ \arcsin(x), & 0<x\le 1;\\ \frac{\pi}{2}+\sqrt{x-1}, & x>1.\end{cases}$$

```
program tabulation_func
! Программа табулирования функции в заданном диапазоне
implicit none
integer(2) :: alg, i, n  ! алгоритм, индекс цикла, число шагов
real(8) :: a, b, h, f, x    ! диапазон, шаг, функция, аргумент
! Выбор алгоритма и определение параметров диапазона
  write(*,*) 'Выберите алгоритм (1 или 2):'; read(*,*) alg
```

```
  select case (alg)
  case(1)
    write(*,*)       'Введите границы диапазона и число шагов:'
    read(*,*) a, b, n
    h = (b - a) / n
  case(2)
    write(*,*)     'Введите границы диапазона и величину шага:'
    read(*,*) a, b, h
    n = ceiling((b - a) / h)                    ! см. прил. Г
  case default
    a = -2; b =  2; n = 40
    h = (b - a) / n
  end select
  write(*,'(a4, 2a15)') 'i', 'x', 'f'           ! печать заголовка
! Цикл вычисления значений функции в заданном диапазоне
  do i = 0, n
    x = a + i * h
    if (x <= -1.) then
      f = -1.
    else if (x <= 0.) then
      f = -x * x
    else if (x <= 1.) then
      f = asin(x)
    else
      f = asin(1.0_8) + sqrt(x - 1)
    endif
    write(*,'(i4, 2f15.7)') i, x, f             ! печать таблицы
  end do
end program tabulation_func
```

В программе использованы два алгоритма табулирования, в которых пользователем задаются границы диапазона `[a, b]`, а также число шагов в пределах заданного диапазона `n` (алгоритм 1) или величина шага продвижения по диапазону `h` (алгоритм 2). В случае ввода на запрос выбора алгоритма целого числа, отличного от 1 и 2, выполняется вариант по умолчанию.

Операторы многозначного выбора и ветвления применяются здесь на этапе выбора алгоритма и вычисления параметров и в процессе вычислений внутри тела цикла соответственно. В операторах вывода используется форматирование для аккуратной печати данных в таблице (см. гл. 6).

Статистические вычисления являются постоянно востребованными на практике и реализуются во многих программных пакетах.

Напишем программу, которая использует операторы цикла для нахождения *среднего арифметического* и *среднего квадратического значений* (RMS) для набора входных данных вещественного типа по следующим формулам:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \quad , \quad s = \sqrt{\frac{\sum_{i=1}^{N} x_i^2 - \left(\bar{x}\right)^2}{N-1}} \; ; \; \mathrm{RMS}(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \, .$$

```
program stats_calculations
! Программа для выполнения статистических вычислений
implicit none
integer(2) :: i, n
real(8) :: sum_x, sum_x2, x, x_bar, x_rms
! Считывание данных и вычисление сумм
  write(*,*) 'Введите количество чисел в наборе:'; read(*,*) n
  sum_x = 0.; sum_x2 = 0.
  do i = 1, n
    write(*,*) 'Введите число:'
    read(*,*) x
    sum_x  = sum_x + x
    sum_x2 = sum_x2 + x**2
  end do
! Расчет статистики и печать результатов
  x_bar = sum_x / n
  x_rms = sqrt( sum_x2 / n )
  write(*,*) 'Среднее арифметическое значение: ', x_bar
  write(*,*) 'Среднее квадратическое значение: ', x_rms
end program stats_calculations
```

Приведем и обсудим код программы, использующей итеративные циклы обоих видов для приближенного вычисления бесконечного произведения на примере гиперболического косинуса по формуле:

$$\cosh(x) = \prod_{n=1}^{\infty}\left(1 + a_n\right) \quad , \quad a_n = \frac{x^2}{\left(\frac{\pi}{2}(2n-1)\right)^2} .$$

```
program inf_product
! Программа вычисления cosh через бесконечное произведение
implicit none
```

```
character(len=2) :: sym
real(8) :: x, eps, an, tmp, cosh_iprod, cosh_ifunc
integer :: n, nmax
! Цикл, выполняющийся до ввода пользователем символа завершения
  input_output: do
    write(*,*) 'Введите значения x, eps, nmax:'
    read(*,*) x, eps, nmax
    cosh_iprod = 1.0_8; n = 1
    tmp = (x / 1.5707963267948966_8)**2; an = tmp
! Основной вычислительный цикл
    calculations: do while ( (an >= eps) .and. (n <= nmax) )
      cosh_iprod = cosh_iprod * (1. + an)
      n = n + 1
      an = tmp / real((2*n - 1)**2,8)
    end do calculations
    cosh_ifunc = cosh(x)
    write(*,*) 'n =', n-1, 'an =', an, 'eps =', eps
    write(*,*) 'x =', x, ' cosh(inf_product) =', cosh_iprod, &
               ' cosh(intr_func) =', cosh_ifunc
    write(*,*) 'Завершить выполнение программы Д/Н?'
    read(*,*) sym
    if ( (sym == 'д') .or. (sym == 'Д') ) exit input_output
  end do input_output
end program inf_product
```

В этой программе после секции объявления располагается «бесконечный» цикл с именем `input_output`, который повторяется до момента, пока пользователь вводит в качестве ответа на запрос необходимый символ. Внутри цикла производится считывание значений аргумента `x`, погрешности `eps`, участвующей в задании условия выхода из вычислительного цикла, и максимального числа частичных произведений `nmax`. Последнее ограничение необходимо для страховки от отсутствия сходимости.

Основной вычислительный цикл с именем `calculations` выполняется до удовлетворения условия выхода при достижении `an` значения, меньшего `eps`, или при превышении числом частичных произведений `n` значения `nmax`. После выполнения очередного этапа вычислений для сопоставления производится вывод рассчитанного значения произведения и эталонного значения, полученного с помощью встроенной функции `cosh`.

5.5. Варианты заданий

5.5.1. Математический блок

1. Выполните табулирование функций следующего вида:

$$\text{а) } f(x)=\begin{cases}(x+6)(x+4), & x\le -5,\\ x/5, & -5<x\le 0,\\ \sqrt{x}, & 0<x\le 25,\\ 5, & x>25;\end{cases}$$

$$\text{б) } f(x)=\begin{cases}-x-2\pi, & x\le -2\pi,\\ \sin(x), & -2\pi<x\le 0,\\ \sin(x/2), & 0<x\le 4\pi,\\ x-4\pi, & x>4\pi.\end{cases}$$

2. Вычислите значение функций по следующим формулам:

$$\text{а) } \sin(x)=\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n+1}}{(2n+1)!};\ \text{б) } \operatorname{arctg}(x)=\sum_{n=1}^{\infty}\frac{(-1)^{n-1}}{2n-1}x^{2n-1};$$

$$\text{в) } \operatorname{sh}(x)=\sum_{n=0}^{\infty}\frac{x^{2n+1}}{(2n+1)!}.$$

Бесконечный числовой ряд Тейлора следует заменить на конечный, подобрав подходящее значение количества членов. Сравните полученное значение на основе ряда с рассчитанным с помощью встроенной функции.

3. Вычислите значение функции `f(x)=ln[1/(1-x)]` для вводимых `x`. Используйте бесконечный цикл с повторением вычислений для допустимых значений `x`, при вводе некорректного значения – завершите программу.

4. Определите, является ли введенное число простым. Если число не простое, найдите и выведите все его делители.

5. Определите и выведите четные числа в диапазоне `0...50`, а также их сумму, квадраты и квадратные корни.

6. Вычислите биномиальный коэффициент для заданных n и k по следующей формуле:

$$(1+x)^n = \binom{n}{0} + \binom{n}{1}x + \binom{n}{2}x^2 + \ldots + \binom{n}{n}x^n = \sum_{k=0}^{n}\binom{n}{k}x^k.$$

7. Выведите таблицу углов α от 0 до 360° с заданным шагом изменения угла, значений угла в радианах и тригонометрических функций sin(α), cos(α) и tg(α).

8. Выполните перевод заданного числа из десятичной системы счисления в двоичную и из двоичной в десятичную.

9. Вычислите для заданного значения n числа Фибоначчи f и Леонардо L, используя следующие формулы:

$$\text{а) } f(n) = \begin{cases} 1, & n = 1, \\ 2, & n = 2, \\ f(n-1) + f(n-2), & n \geq 3; \end{cases}$$

$$\text{б) } L(n) = \begin{cases} 1, & n = 0, \\ 1, & n = 1, \\ L(n-1) + L(n-2) + 1, & n > 1. \end{cases}$$

10. Вычислите среднее геометрическое, среднее гармоническое и среднее квадратическое значения для набора введенных данных, используя следующие формулы (сравните со средним арифметическим значением):

$$\text{а) } x_{geom} = \sqrt[n]{x_1 x_2 x_3 \ldots x_n}; \quad \text{б) } x_{harm} = \frac{N}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_N}};$$

$$\text{в) } RMS(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}.$$

11. Вычислите значения следующих бесконечных последовательностей с заданной точностью $|x_{n+1} - x_n| \leq \varepsilon$ ($\varepsilon = 10^{-7}$):

а) $x_{n+1} = \sqrt[3]{1000 - x_n}$ и $y(x) = x^3 + x - 1000$;

б) $x_{n+1} = x_n - \frac{y(x_n)}{y'(x_0)}$ и $y(x) = \sin(x) - x\cos(x)$.

12. Вычислите основание натуральных логарифмов e по следующим формулам с заданной точностью $e_{n+1}-e_n \le \varepsilon$ ($\varepsilon = 10^{-7}$):

$$\text{а) } e=\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n; \quad \text{б) } e=\lim_{n\to\infty}\frac{n}{\sqrt[n]{n!}}.$$

13. Вычислите число π по следующим формулам с заданной точностью $|\pi_{n+1} - \pi_n| \le \varepsilon$ ($\varepsilon = 10^{-7}$):

$$\text{а) } \pi=2\prod_{n=1}^{\infty}\frac{4n^2}{4n^2-1}; \quad \text{б) } \pi=4\prod_{n=1}^{\infty}\left[1-\frac{1}{(2n+1)^2}\right];$$

$$\text{в) } \pi=6\Bigg/\left[e\prod_{n=2}^{\infty}\left(1+\frac{2}{n}\right)^{k(n)}\right], \; k(n)=n(-1)^n.$$

14. Выполните следующие операции:

$$\text{а) } S=\prod_{i=1}^{n}\left[x_{n-i-1}\sum_{j=2}^{n-1}x_j\right], \quad x_{k+1}=\sqrt[3]{1000-x_k};$$

$$\text{б) } S=\frac{1}{n!\,m!}\prod_{i=1}^{m}\prod_{j=1}^{n}(i+j-n)^{ij}.$$

5.5.2. Физический блок

1. Используя уравнение Менделеева–Клапейрона $p/\rho = TR_{ун}/M_m$, где p – давление; ρ – плотность; T – температура; $R_{ун}$ – универсальная газовая постоянная; M_m – молярная масса (рассмотреть воздух с $\rho = 1.2$ кг/м^3):

а) для заданного диапазона изменения давления среды p выведите на экран изменение температуры T при постоянном значении плотности ρ;

б) для заданного диапазона изменения температуры среды T выведите на экран изменение давления p при постоянном значении плотности ρ.

2. Рассматриваются механические весы с двумя чашами. Расстояние от чаш до центра весов может варьироваться от 10 до 50 см. На одну из чаш весов по очереди устанавливаются

грузики массой 1...10^3 г. Используя закон сохранения момента, составьте таблицу соотношений между расстояниями от чаш до центра весов и массой грузиков, устанавливаемых на каждой чаше с заданным шагом. Интерес представляет случай уравновешенных чаш.

3. Найдите положение ядра, двигающегося в поле притяжения Земли недалеко от ее поверхности, в заданные моменты времени. Зависимость координат x и y от времени t имеет следующий вид: $x(t) = x_0 + V_{x,0}t$, $y(t) = y_0 + V_{y,0}t - gt^2/2$. Здесь x_0 и y_0 – начальные координаты ядра; $V_{x,0}$ и $V_{y,0}$ – компоненты начальной скорости движения ядра.

4. Вычислите соотношения двух измерений мощности в децибелах по формуле $dB = 10 \cdot \log10(P_2/P_1)$, где P_2 – измеряемый уровень мощности (изменяется в диапазоне от 1 до 20 Вт с шагом 0,5 Вт); P_1 – некоторый эталонный уровень мощности (равный 1 Вт).

5. Рассматривается установившееся ламинарное течение несжимаемой жидкости сквозь трубу квадратного сечения с длиной сторон h = 2 (безразмерное). Распределение продольной скорости жидкости по сечению трубы может быть записано в следующей форме (в безразмерном виде):

$$V = \frac{16}{\pi^3} \sum_{p=0}^{\infty} \frac{(-1)^p}{(2p+1)^3} \left[1 - \frac{ch\left(\frac{2p+1}{2}\pi y\right)}{ch\left(\frac{2p+1}{2}\pi\right)} \right] \cos\left(\frac{2p+1}{2}\pi x\right),$$

где x, y – безразмерные координаты.

Вычислите и выведите на экран значения ряда для набора N×N координат x, y из диапазона [-1…1], изменяющихся с равномерным шагом.

6. Выполните пересчет единиц измерения (фунты – килограммы, дюймы – сантиметры, мили – километры, метры в секунду – километры в час и т. д.) для заданного диапазона и шага и выведите соответствующую таблицу.

Глава 6

БАЗОВЫЕ КОНЦЕПЦИИ ВВОДА/ВЫВОДА

6.1. Форматирование ввода/вывода данных

В п. 3.9 был рассмотрен и преимущественно использовался при дальнейшем изложении ввод/вывод, управляемый списком, где формат определяется типами задействованных данных. В п. 6.1 будет рассмотрен форматный вид операций ввода/вывода в языке Фортран, а также особенности выполнения ввода/вывода при работе с файлами.

Форматный вид используется для явного указания способа вывода переменных, может специфицировать горизонтальное и вертикальное положение, количество значащих цифр и т. д. Соответствующие модификации производятся с помощью *дескрипторов преобразований (формата)*, ответственных за контролируемый перевод данных из внутреннего представления компьютера (бинарного) во внешнее (текстовое).

Дескрипторы содержатся в спецификации формата, которая, являясь необязательной, тем не менее может быть отдельно задана во втором поле в списке параметров оператора **`write`** или оператором **`format`**. Основные дескрипторы преобразований представлены в табл. 6.1.

Таблица 6.1

Основные дескрипторы преобразований

Дескриптор	Тип аргумента	Внешнее представление
`iw`	Целый	Целое число
`bw`	Целый	Двоичное представление
`ow`	Целый	Восьмеричное представление
`zw`	Целый	Шестнадцатеричное представление
`fw.d`	Вещественный	Вещественное число в F-форме
`ew.d[ee]`	Вещественный	Вещественное число в E-форме
`esw.d[ee]` `enw.d[ee]`	Вещественный	Научная нотация, формат аналогичен предыдущему, но при этом число

Окончание табл. 6.1

Дескриптор	Тип аргумента	Внешнее представление
		преобразуется таким образом, чтобы мантисса лежала в диапазоне 1...10 и 1...1000 соответственно
dw.d	Вещественный	Вещественное число двойной точности
lw	Логический	Логическая переменная
aw	Символьный	Строка символов
nx	–	Пробельные символы (n – количество пробелов)
tc	–	Переход к заданной позиции в строке (c – номер позиции)

В табл. 6.1 введены следующие обозначения: w – длина поля (общее количество символов, используемых для ввода или вывода); d – число десятичных знаков; e – число позиций при выводе показателя степени.

Приведем несколько примеров. Начнем с вывода вещественного числа в поле длиной в шесть символов, в котором три отведены для представления дробной части:

```
real(8) a = 12.3456_8
write(*, '(f6.3)') a    ! 12.346
write(*, 100) a         ! 12.346
100 format(f6.3)
```

Вывод вещественных чисел с использованием научной нотации:

```
real :: a = 1.2346e6, b = 0.001
write(*, '(es10.4, es8.1)') a, b  ! 1.2346E+06 1.0E-03
```

Особый случай использования дескриптора нулевой длины приводит к тому, что целое число записывается с переменной шириной поля, достаточной для хранения содержащейся в нем информации:

```
integer :: i = -12, j = 4, k = -12345
write(*, '(i0,1x,i0,1x,i0)') i, j, k   ! -12 4 -12345
```

Форматный вывод переменных логического и символьного типов:

```
logical :: output = .true., debug = .false.
character(len=19) :: s = 'Это строка'
```

```
write(*, '(l1, l2)') output, debug    ! T F
write(*, '(a, 1x, a6)') s, s                ! Это строка
```

В последнем случае будет выведено только три начальных символа строки, поскольку для хранения кириллических символов при использовании стандарта UTF-8 требуется 2 байта.

Вывод набора данных с одинаковым форматом может производиться с использованием *коэффициента повторения* для дескриптора или группы:

```
real(8) :: a = 1.23, b = 3.45, c = 5.67
write(*, '(1x,f4.2, 1x,f4.2, 1x,f4.2)') a, b, c    ! 1.23 3.45 5.67
write(*, '(3(1x, f4.2))') a, b, c                  ! 1.23 3.45 5.67
write(*, '(*(1x, f4.2))') a, b, c                  ! 1.23 3.45 5.67
```

Знак звездочки в последнем примере позволяет не указывать явно значение коэффициента повторения (снято ограничение на число повторений).

Следует отметить, что компилятор Intel и некоторые другие позволяют использовать целочисленные переменные и константы (в угловых скобках) для задания динамически корректируемого формата:

```
integer, parameter :: i = 3
write(*, '(<i>(1x, f4.2))') a, b, c    ! 1.23 3.45 5.67
```

Компилятор gfortran не поддерживает этот функционал, который можно воспроизвести, используя более мощную (и стандартную) комбинацию внутреннего вывода и строковых форматов:

```
integer, parameter :: i = 3
character(len=20) :: fmt
write(fmt, *) i
write(*, '('//fmt//'(1x, f4.2))') a, b, c    ! 1.23 3.45 5.67
```

Специальный символ (косая черта) в представлении формата отвечает за инициализацию перехода вывода данных на новую строку (/, не путать со знаком // – конкатенацией):

```
integer :: i = 123, j = 4567
write(*, '(a, i3 / a, i4)') 'i = ', i, 'j = ', j
! i = 123
! j = 4567
```

В дескрипторе может также использоваться специальный символ (знак доллара) $, который, наоборот, подавляет перевод строки (в некоторых компиляторах – \).

Форматный ввод данных (оператор **read**) базируется на тех же принципах и дескрипторах, что и вывод, поэтому в данном случае не рассматривается.

6.2. Файловый ввод/вывод

Компьютеры и работающие на них ОС эксплуатируют стандартные структуры хранения данных, которые основываются на понятии *файла*. Файл состоит из множества строк данных, расположенных последовательно, связанных друг с другом и доступных как единое целое (на основании присвоенного файлу имени).

Фортран обладает гибкими средствами работы с файлами в режиме чтения или записи информации, не зависящими от устройства хранения (типа носителя). Реализующий эти возможности универсальный механизм называется *устройством ввода/вывода* (УВВ, или *логическим устройством*).

Система ввода/вывода в Фортране основывается на понятии *записи* как логически завершенной группе данных (например, строке, заканчивающейся символом новой строки). Записи бывают форматные (текстовые), неформатные (создаются без преобразования данных) и указывающие на конец файла.

Для управления вводом и выводом из файлов могут использоваться рассмотренные далее операторы языка, включенные в табл. 6.2.

Таблица 6.2

Операторы Фортрана для работы с файлами

№ п/п	Имя оператора	Выполняемая функция
1	**open**	Привязка определенного файла на носителе к заданному номеру УВВ («открытие файла»)
2	**close**	Завершение привязки файла на носителе к заданному номеру УВВ («закрытие файла»)
3	**read**	Чтение данных из УВВ, указанного с помощью номера
4	**write**	Запись данных в УВВ, указанное с помощью номера
5	**rewind**	Перемещение к началу файла
6	**backspace**	Перемещение в файле на одну запись назад

Номера УВВ (переменная целого типа) присваиваются определенным файлам на носителе или устройствам с помощью оператора **open** и отсоединяются от них оператором **close**. После выполнения привязки к УВВ оператором **open** («открытия») можно осуществлять чтение и запись данных.

Номер УВВ указывается в первом поле в списке параметров операторов **read** и **write** (вместо звездочки). В остальном эти операторы записываются и используются так же, как и при работе со стандартными УВВ. Операторы **rewind** и **backspace** могут использоваться для изменения текущей позиции чтения или записи в файле, пока он открыт.

Оператор **open** связывает файл с заданным номером УВВ и записывается в следующем виде:

```
open(open_list),
```

где `open_list` содержит набор разделенных запятыми спецификаторов, с помощью которых задаются номер УВВ, имя файла и информация о том, как получить к файлу доступ.

Введем на данном этапе в рассмотрение шесть наиболее востребованных на практике спецификаторов (табл. 6.3).

В случае если оператор выполнен успешно, в переменную `int_var` спецификатора `iostat` будет возвращен `0`, а содержимое символьной переменной `chart_var` спецификатора `iomsg` не изменится, в противном случае будут возвращены целое положительное число, соответствующее коду системной ошибки, и описательное сообщение об ошибке соответственно.

Приведем примеры записи операторов открытия файлов для записи и для чтения:

```
integer :: un = 8, ierror
character(len=10) :: fn = 'input.dat'
character(len=80) :: err_msg
open(unit=un, file=fn, status='old', action='read', &
iostat=ierror, iomsg=err_msg)
open(9, file='output.dat', status='new', action='write', &
iostat=ierror, iomsg=err_msg
```

Значение `'old'` спецификатора `status` указывает, что открываемый файл уже существует, `'new'` – что должен быть создан новый

Таблица 6.3

Спецификаторы оператора `open`

Имя спецификатора	Описание действия	Форма, тип и возможные варианты значений
`unit`	Указывает номер УВВ, который будет связан с интересующим файлом	`unit = int_expr` целое неотрицательное число
`file`	Задает имя файла, который необходимо открыть	`file = char_expr` символьная строка

`status`	Определяет статус подлежащего открытию файла	`status = char_expr`
		символьная строка
		`'old'`, `'new'`, `'replace'`, `'scratch'`, `'unknown'`

`action`	Указывает, должен ли файл быть открыт только для чтения, только для записи или для чтения и записи одновременно	`action = char_expr`
		символьная строка
		`'read'`, `'write'`, `'readwrite'`

`iostat`	Определяет имя переменной, в которую может быть возвращен статус операции открытия	`iostat = int_var` целочисленная переменная
`iomsg`	Определяет имя переменной, которая будет содержать сообщение в случае возникновения ошибки	`iomsg = chart_var` символьная переменная

файл, а `'replace'` – что новый файл должен быть открыт для вывода независимо от того, существует файл с таким именем или нет. Значение `'read'` спецификатора `action` указывает, что файл должен быть доступен только для чтения, `'write'` – только для записи.

Следующий оператор открывает временный (*scratch*) файл и присоединяет его к УВВ:

```
open(unit=10, status='scratch', iostat=ierror)
```

Временный файл создается программой и автоматически удаляется при закрытии файла или при завершении программы; его

можно использовать для хранения промежуточных результатов во время работы программы.

Оператор **close** закрывает файл, освобождает номер связанного с ним УВВ и записывается в следующем общем виде:

```
close(close_list),
```

где close_list должен содержать спецификатор, определяющий номер УВВ, а также и некоторые другие параметры.

Приведем примеры чтения и записи данных при работе с файлами:

```
open(8,  file='input.dat',  status='old',  action='read',  &
iostat=ierror, iomsg=err_msg)
read(8,*) x, y, z
open(9,  file='output.dat',  status='replace',  action='write',  &
iostat=ierror, iomsg=err_msg)
write(9,'(3(a, f8.2))') 'x =', x, 'y =', y, 'z =', z
```

К оператору **read** могут быть добавлены спецификаторы iostat и iomsg (см. табл. 6.2). Логика работы спецификаторов аналогична описанной для оператора **open**, за исключением ошибки чтения по причине достижения конца входного файла – в переменной int_var в этом случае будет возвращено отрицательное число.

Фортран предоставляет два оператора, которые помогают перемещаться по файлу:

```
backspace(unit=un)
rewind(unit=un)
```

Оператор **rewind** выполняет переход к началу файла, а **backspace** перемещает на одну запись назад при каждом вызове.

6.3. Примеры программ

Наглядным примером использования форматного вывода является генерация отформатированной таблицы (в данном случае производятся вычисления и вывод в строках значений целого операнда, квадратного корня от него, а также квадрата и куба):

```
program table_formatted
! Программа генерации отформатированной таблицы с данными
implicit none
```

```
integer :: i, cube, square      ! переменные для целых вычислений
real :: sqroot                  ! переменная для квадратного корня
! Вывод заголовков столбцов таблицы
  write(*, 100)
  100 format(t4,'number',t13,'sqroot',t27,'square',t37,'cube')
  write(*, 110)
  110 format(t4,'=======', t13,'============', t27,'========', &
             t37,'======'/)
! Вычисление и вывод значений в столбцах таблицы с разделителями
  do i = 1, 10
    sqroot = sqrt( real(i) );  square = i**2;  cube = i**3
    write(*, 120) i, sqroot, square, cube
    120 format(1x, '|', t4, i4, t11, '|', f10.6, t25, '|', i6, &
               t35, '|', i7, '|')
  end do
end program table_formatted
```

Напишем программу, выполняющую аппроксимацию набора входных данных методом наименьших квадратов прямой линией и выводящую полученные значения наклона линии и точки пересечения с осью ординат:

```
program least_squares_fit
implicit none
integer :: ierror                              ! статус ввода/вывода
integer :: n                  ! количество пар входных данных (x,y)
character(len=24) :: fn = 'input.dat'    ! файл с входными данными
character(len=80) :: err_msg                   ! сообщение об ошибке
real :: sum_x, sum_x2, sum_xy, sum_y
real :: x, x_bar, y, y_bar, y_int, slope
open(8, file=fn, status='old', iostat=ierror, iomsg=err_msg)
! Проверка корректности открытия файла
errorcheck: if ( ierror > 0 ) then
  write(*,1010) fn
  1010 format ('error: file ',a,' does not exist!')
  write(*,'(a)') err_msg
else
  n = 0; sum_x = 0.; sum_x2 = 0.; sum_xy = 0.; sum_y = 0.
! Считывание пар значений (x,y) из входного файла
  do
    read(8,*,iostat=ierror) x, y
    if ( ierror /= 0 ) exit
    n = n + 1
    sum_x = sum_x + x; sum_y = sum_y + y
    sum_x2 = sum_x2 + x**2; sum_xy = sum_xy + x * y
  end do
! Вычисление наклона линии и точки пересечения с осью y
```

```
  x_bar = sum_x / real(n);  y_bar = sum_y / real(n)
  slope = (sum_xy - sum_x * y_bar) / ( sum_x2 - sum_x * x_bar)
  y_int = y_bar - slope * x_bar
  write(*, 1020) slope, y_int, n
  1020 format('Коэффициенты регрессии для линии МНК:', &
  /,' slope(m) = ', f12.3, &
  /,' intercept(b) = ', f12.3, &
  /,' no of points = ', i12)
  close(8)
end if errorcheck
end program least_squares_fit
```

Рассмотрим фрагмент кода, в котором используется конструкция error stop, завершающая работу программы и выводящая информацию об ошибке (здесь – об ошибке открытия файла с входными данными):

```
integer :: ierror                          ! статус ввода/вывода
character(len=24) :: fn = 'input.dat'  ! файл с входными данными
character(len=80) :: err_msg               ! сообщение об ошибке
! Попытка открытия файла с входными данными
  open(8, file=fn, status='old', iostat=ierror, iomsg=err_msg)
  if ( ierror /= 0 ) then
    write(*, 100) fn, ierror
    100 format ('Ошибка открытия файла ',a,'. iostat = ',i6)
    write(*, '(a)') err_msg
    error stop  'Ошибка открытия файла с входными данными'
  end if
```

Глава 7
ВВЕДЕНИЕ В МАССИВЫ

7.1. Массивы. Основные термины и понятия

Массивы – ключевые компоненты большинства алгоритмов вычислительной направленности, представляющие собой упорядоченные наборы из конечного числа переменных или констант одного типа, доступ к которым производится по общему имени. Отдельное значение в массиве называется элементом, который идентифицируется именем массива и индексом (или набором индексов), указывающим на его конкретное место в массиве.

Массивы являются мощным инструментом для работы с данными и подразделяются на *статические* и *динамические*. Объем оперативной памяти, выделяемой под статические массивы, определяется на этапе компиляции программы. Динамические массивы получают необходимые ресурсы непосредственно в процессе выполнения программы, при этом объем выделенной памяти может быть изменен или память может быть вообще освобождена.

Массивы характеризуются *рангом* (или *размерностью*, числом измерений), *размером* (числом элементов) и *формой* (или *экстентом*, протяженностью вдоль каждого из измерений). Массивы с одинаковыми значениями всех трех перечисленных характеристик называются *согласованными*.

Границами массива в заданном измерении называется диапазон изменения в нем индекса массива. Индексы могут быть любыми следующими подряд целыми числами, в том числе и отрицательными. В языке Фортран нумерация элементов массива в каждом измерении по умолчанию начинается с единицы, а заканчивается числом, равным соответствующей протяженности массива. В случае если нижняя граница начинается с единицы, при объявлении массива ее можно опустить (иначе границы необходимо указывать явно, используя разделитель между границами – двоеточие).

Как будет показано далее, в современном Фортране можно манипулировать массивами и выполнять вычисления с отдельными элементами массивов «один за другим», с именованными массивами целиком или с их сечениями.

7.2. Статические массивы. Объявление и инициализация. Поэлементная работа с массивами

Перед использованием статического массива требуется явным образом объявить его тип и количество элементов (атрибут **dimension**), например, для массивов ранга 1 (одномерного массива, вектора):

```
integer, parameter :: n = 5                    ! размер массива
integer, dimension(n) :: a1                ! массив из переменных
integer(2), parameter, dimension(n) :: a2  ! массив из констант
```

```
real, dimension(-5:2) :: a3          ! объявление границ массива
real, dimension(5:2) :: a4             ! массив нулевого размера
character(len = 2), dimension(7) :: s            ! массив строк
logical, dimension(n) :: f       ! массив логических переменных
```

Массив, форма которого явно объявлена в операторе объявления типа, называется *массивом заданной формы*. Массив может иметь нулевой размер (нижняя граница превосходит верхнюю); работа с такими массивами происходит по общим правилам. При объявлении рекомендуется использовать именованные константы, поскольку в этом случае легко изменять размеры массивов при компиляции, что особенно ценно для больших и сложно устроенных программ.

Элементы массива могут быть инициализированы операторами присваивания, вызовами встроенных функций, в операторах объявления типов или с помощью операторов ввода **read** (см. п. 7.5). Обращение к элементам массива выполняется с использованием круглых скобок.

Приведем примеры различных вариантов инициализации векторов с помощью операторов присваивания:

```
integer, parameter :: n = 5
real, dimension(n) :: a1, a2, a3, a4
  do i = 1, n
    a1(i) = real(i)              !  1.0  2.0  3.0  4.0  5.0
  end do
  a2 = [ (real(i), i = 1,n) ]    !  1.0  2.0  3.0  4.0  5.0
  a3 = [1., 2., 3., 4., 5.]      !  1.0  2.0  3.0  4.0  5.0
  a4 = 0.                        !  0.0  0.0  0.0  0.0  0.0
```

Массивы можно инициализировать непосредственно при объявлении их типов, например, так:

```
integer, parameter :: n = 5
integer :: i, j
integer, dimension(n) :: a1 = [ 1, 2, 3, 4, 5 ]
integer(2), dimension(n) :: a2 = [ (i, i = 1, n) ]
character(len = 4), dimension(7) :: s = ['Пн', 'Вт', 'Ср', & 'Чт', 'Пт',
'Сб', 'Вс']    ! стандарт кодирования UTF-8, len = 4
real(8), dimension(n) :: a3 = 1.
real, dimension(n) :: a4 = [0., (exp(real(i)*.5), i = 1, 3), 9.]
integer, dimension(15) :: a5 = [ ((0, i = 1, 4), 5*j, j =1, 3) ]
! Результат:  0  0  0  0  5  0  0  0  0  10  0  0  0  0  15
```

Для создания сложных шаблонов инициализации могут использоваться выражения с переменными, неявные циклы и некоторые встроенные функции. Так, в последнем примере элементы массива инициализируются нулем, если не делятся на 5, и значением номера элемента в противном случае.

Схема расположения элементов некоторого вектора a размера 5 с границами 1 и 5 в линейной по своей структуре оперативной памяти компьютера показана на рис. 7.1.

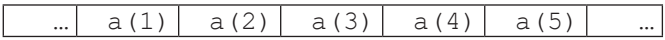

Рис. 7.1. Расположение элементов вектора в оперативной памяти

Приведем примеры фрагментов программ с поэлементной обработкой векторов в арифметических циклах – вычисление функции факториала для заданного набора значений целых чисел (см. вариант кода без массива в п. 5.1); присвоение четным элементам массива нечетных значений индексов и наоборот; возведение в квадрат набора значений целых чисел:

```
integer, parameter :: n = 10
integer :: i                                    ! индекс массива
integer, dimension(0:n) :: fact              ! объявление массива
  fact(0) = 1
  do i = 1, n
    fact(i) = i * fact(i-1)
  end do
! Элементы массива fact: 1 1 2 6 24 120 720 5040 40320 362880
integer, dimension(n) :: even_odd            ! объявление массива
  do i = 1, n/2
    even_odd(2*i)   = 2 * i-1
    even_odd(2*i-1) = 2 * i
  end do
! Элементы массива even_odd: 2  1  4  3  6  5  7  6  8  7  10  9
real, dimension(-2:2) :: a = [ (real(i), i = -2,2) ], square
  do i = -2, 2 ! цикл можно заменить на одну команду square=a**2
    square(i) = a(i)**2
  end do
! Элементы массива square: 4.0  1.0  0.0  1.0  4.0
```

Современные компиляторы (в том числе и gfortran) на этапе компиляции выполняют проверку кода программы на возможный

выход индексов за границы массивов, предупреждая об этом разработчика:

```
integer, parameter :: n = 5
integer :: i
real, dimension(n) :: a = [ (real(i), i = 1,n) ]
  do i = 1, n+1
    a(i) = a(i) * 2.
  end do
! Warning: Array reference out of bounds (6 > 5) in loop
```

7.3. Именованные массивы. Сечения массива

Термин *именованный* (*named*) по отношению к массиву означает, что его имя не сопровождается индексами, а операции выполняются сразу над всеми или над явно указанными элементами одновременно, как если бы они были обычными переменными (также в специальной литературе используется термин *whole array, массив целиком*).

Приведем код программы и иллюстрацию (рис. 7.2) выполнения операции сложения двух согласованных векторов двумя способами – поэлементно и с использованием именованных массивов:

```
program sum_vectors
implicit none
integer, parameter :: n = 4
integer :: i
real, dimension(n) :: a = [ 1., 2., 3., 4. ]
real, dimension(n) :: b = [ 5., 6., 7., 8. ], c, d
! Поэлементное сложение векторов
  do i = 1, n
    c(i) = a(i) + b(i)                ! 6.0   8.0   10.0   12.0
  end do
! Сложение именованных массивов целиком
    d = a + b                         ! 6.0   8.0   10.0   12.0
end program sum_vectors
```

Массивы можно использовать в качестве операндов, только если они согласованы, диапазон индексов в каждом измерении при этом необязательно должен быть одинаковым.

Многие встроенные функции Фортрана, используемые со скалярными величинами, могут принимать массивы как входные аргументы и возвращать их в качестве результатов массива.

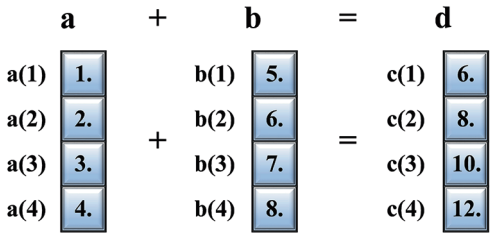

Рис. 7.2. Иллюстрация выполнения операции сложения векторов

Возвращенные массивы будут содержать результат поэлементного применения функции к входному массиву. Такие функции называются элементными, например, `abs`, `mod`, `floor`, `sign`, `int`, `real`, `sin`, `cos`, `exp`, `log`, `sqrt` и др. (см. прил. Г).

Приведем код программы с примерами работы с именованными массивами, в том числе с привлечением элементных встроенных функций:

```
program whole_arrays
implicit none
integer, parameter :: n = 5
real, dimension(n) :: a = [ 1., 2., 3., 4., 5. ]
real, dimension(n) :: b = 2., c, d, e, f
! Прибавление ко всем элементам массива вещественного числа
  c = a + 1.                              ! 2.0  3.0  4.0  5.0  6.0
! Умножение массивов
  d = a * b                             ! 2.0  4.0  6.0  8.0  10.0
! Операция с массивами с использованием встроенных функций
  e = sqrt (log(a) / sin(b))        ! 0.0  0.87  1.1  1.23  1.33
! Операция с массивами - модуль a, умноженный на знак -b
  f = sign (a, -b)                ! -1.0  -2.0  -3.0  -4.0  -5.0
end program whole_arrays
```

Помимо обращений к элементам массивов поэлементно или целиком в Фортране можно использовать подмножества элементов, называемые *сечениями*. Сечения позволяют формировать подмножества регулярной структуры. Задание сечения осуществляется при помощи индексного *триплета* – тройки целых чисел – и в наиболее общем виде выглядит так:

```
[нижняя граница] : [верхняя граница] : [шаг]
```

Приведем примеры различных вариантов использования сечений заданного одномерного массива и результаты выполнения операций:

```
integer :: i = 3, j = 7
real, dimension(8) :: a = [ 1., -2., 3., -4., 5., -6., 7., -8. ]
a(:)       ! 1.0  -2.0  3.0  -4.0  5.0  -6.0  7.0  -8.0
a(i:j)     ! 3.0  -4.0  5.0  -6.0  7.0
a(i:j:i)   ! 3.0  -6.0
a(:j:i)    ! 1.0  -4.0  7.0
a(i:)      ! 3.0  -4.0  5.0  -6.0  7.0  -8.0
a(:j)      ! 1.0  -2.0  3.0  -4.0  5.0  -6.0  7.0
a(::i)     ! 1.0  -4.0  7.0
```

Другой, более гибкий, способ основан на задании векторного индекса. *Векторный индекс* – одномерный целочисленный массив, содержащий значения индексов, попадающих в пределы границ исходного массива, из которого будет извлекаться подмножество элементов. Массив, заданный с помощью векторного индекса, будет содержать по одному элементу для каждого указанного индекса, например:

```
integer, dimension(5) :: v = [ 2, 6   , 4, 2, 1 ]
real, dimension(6) :: a = [ 1., -2., 3., -4., 5., -6. ]
  write(*,*) a(v)      ! a(v) = -2.0  -6.0  -4.0  -2.0  1.0
```

В отличие от индексного триплета элементы в векторном индексе могут располагаться в произвольном порядке и повторяться.

Пример программы, использующей сечения для подсчета суммы элементов массива с четными и нечетными индексами:

```
program even_odd_sections
implicit none
integer, parameter :: n = 6
real, dimension(n) :: a = [ 1., -2., 3., -4., 5., -6. ]
  write(*,*) 'Сумма нечетных элементов:', sum(a(1:n:2))  ! 9.0
  write(*,*) 'Сумма четных элементов:',   sum(a(2:n:2))  ! -12.0
end program even_odd_sections
```

Отметим, что среди широкого набора *встроенных функций* Фортрана, представленных в прил. Г, часть предназначена исключительно для работы с массивами, в том числе справочные функции, функции заполнения и преобразования, поиска экстремальных элементов и некоторые др.

7.4. Многомерные массивы и составные циклы

Многомерные массивы имеют ранг больше единицы, а обращение к их элементам производится с помощью указания (в круглых скобках через запятую) количества индексов, равного рангу. Примеры объявления:

```
integer, dimension(3,2) :: A1                    ! массив ранга 2
real(8) A2(4,3,2)    ! массив ранга 3 (другой вариант описания)
real, dimension(-5:2,8:10,0:5) :: A3 ! объявление границ массива
character(len = 4), dimension(-3:3,6) :: s        ! массив строк
```

В языке Фортран, если применить аналогию с массивами ранга 2 (матрицами), которые адресуются двумя индексами, элементы располагаются в оперативной памяти *по столбцам* (рис. 7.3, матрица формы 3 на 2). Таким образом, «пробегание» по многомерному массиву начинается с первого индекса, а заканчивается последним (в языке С – наоборот). Этот факт следует учитывать при формировании порядка вложенности циклов в коде программ (см. пояснения в п. 9.8).

Примеры инициализации матрицы с помощью составного (тесно вложенного) цикла и оператора присваивания (рис. 7.3):

```
integer, parameter :: m = 3, n = 2
integer :: i, j
integer, dimension(m,n) :: A
  do i = 1, m
    do j = 1, n
      A(i,j) = i+3*(j-1)+1  ! расположение в памяти: 2 3 4 5 6 7
    end do
  end do
```

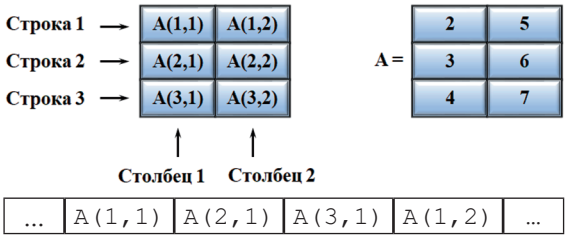

Рис. 7.3. Расположение элементов матрицы в оперативной памяти

Встроенная функция **reshape** позволяет менять форму массива при объявлении типов или в коде программы, сохраняя количество элементов массива, например, для инициализации той же матрицы:

```
integer, parameter :: m = 3, n = 2
integer, dimension(m,n) :: A(m,n) = &
        reshape ( [ 2, 3, 4, 5, 6, 7 ], [m,n] )        ! или
     A = reshape ( [ 2, 3, 4, 5, 6, 7 ], [m,n] )
```

Первый вектор в параметрах функции содержит элементы для изменения формы, второй – описывает целевую форму. Количество элементов во втором векторе задает число измерений в выходном массиве, а значение элементов в нем – размер каждого из измерений.

Сечения многомерных массивов могут быть отобраны с использованием индексных триплетов или векторных индексов. Например, создание сечений, соответствующих первому столбцу и третьей строке матрицы `A`, производится обращениями `A(:,1)` и `A(3,:)`.

Индексы массива можно использовать независимо в каждом измерении. Так, сечение массива `A(1:3:2, 1:2)` представляет собой матрицу, состоящую из строк 1 и 3 и столбцов 1 и 2 исходной матрицы, т. е. с исключением из нее элементов `A(2,1)` и `A(2,2)`.

Задание сечения при помощи векторного индекса позволяет гибко оперировать с элементами многомерных массивов (повторять, располагать в произвольном порядке), например:

```
i = [2, 6, 4, 7, 15];  j = [3, 12, 6, 18]
A(i,j) = 4!    20 выбранных элементов массива примут значение 4
```

Фортран поддерживает и более сложные многомерные массивы, содержащие до 15 индексов, которые хранятся в памяти, объявляются, инициализируются и используются по тем же правилам, что и матрицы. На практике часто используются массивы ранга 3, например при численном моделировании физических и иных процессов в трехмерном пространстве.

7.5. Операции ввода/вывода с массивами

Операции ввода/вывода с массивами в Фортране можно выполнять как с отдельными элементами, так и с массивами целиком. В следующих примерах вывод массива производится в одну строку поэлементно и с помощью неявного арифметического цикла (результат – это `1.0 2.0 3.0 4.0 5.0`):

```
integer, parameter :: n = 5
character(len=20) :: fmt
real, dimension(n) :: a = [ ( real(i), i = 1, n ) ]
write(fmt, *) n
write(*,'('//fmt//'(1x,f3.1))') a(1), a(2), a(3), a(4), a(5)
write(*,'(*(1x,f3.1))') a(1), a(2), a(3), a(4), a(5)
write(*,'(*(1x,f3.1))') ( a(i), i = 1, n )
write(*,'(*(1x,f3.1))') a
```

В общем случае операторы ввода/вывода с неявным циклом могут содержать набор значений и границы изменения индекса цикла, при этом каждый аргумент в списке значений задействуется однократно при выполнении шага цикла, например (результат вывода – `1 2.0 3 6.0 5 10.0`):

```
write(*,'(*(i3,1x,f4.1))') (i, a(i)*2., i = 1, 5, 2)
```

Неявные циклы, как и обычные циклы `do`, могут быть вложенными, при этом внутренний цикл будет выполняться целиком для каждого внешнего шага, например (результатом вывода будет матрица, как на рис. 7.3):

```
integer, parameter :: m = 3, n = 2
A = reshape ( [ 2, 3, 4, 5, 6, 7 ], [m,n] )
write(*,'(i5,1x,i5)') ((A(i, j), i = 1, m), j = 1, n)
```

Другой вариант вывода матрицы в таком же виде основан на использовании цикла `do` и сечения:

```
do i = 1, m
  write(*,'(*(i5,1x))') A(i,:)
end do
```

Вывод массивов целиком производится посредством указания имени массива в качестве параметра оператора **`write`**, например:

```
write(*,*) a
write(*,'(*(1x,f4.1))') A
```

В следующем примере выполняется вывод сечения массива, выбранного с помощью триплета или векторного индекса:

```
integer :: i = 3, j = 5
real, dimension(6) :: a = [ 1., -2., 3., -4., 5., -6. ]
integer, dimension(4) :: v = [ 2, 6     , 4, 2 ]
write(*,'(*(f4.1,1x))') a(i:j)       ! 3.0  -4.0  5.0
write(*,'(*(f4.1,1x))') a(v)         ! -2.0  -6.0  -4.0  -2.0
```

Массивы можно инициализировать с помощью оператора **read**. При отсутствии индексов у массивов-аргументов оператора программа попытается прочитать все доступные значения и присвоить их элементам массива в порядке, соответствующем хранению в памяти. Например, если файл с именем `init.dat` содержит набор записанных в одну строку числовых значений (`2 3 4 5 6 7`), то в результате выполнения следующих блоков операторов будут инициализированы вектор `x` и матрица `A`, как на рис. 7.3:

```
integer, parameter :: n = 6
integer, dimension(n) :: x
open(8, file='init.dat', status='old', action='read')
read(8,*) x
integer, parameter :: m = 3, n = 2
integer, dimension(m,n) :: A
open(8, file='init.dat', status='old', action='read')
read(8,*) A    ! 2  3  4  5  6  7
```

Неявные циклы могут использоваться в операторе ввода для изменения порядка инициализации элементов массива или для инициализации только его части. Например, операторы, представленные далее, инициализируют матрицу с другим порядком расположения элементов (по столбцам — `2 4 6 3 5 7`):

```
integer :: i, j
integer, parameter :: m = 3, n = 2
integer, dimension(m,n) :: A
open(8, file='init.dat', status='old', action='read')
read(8,*) ( (A(i,j), j=1, n), i=1, m )
```

Пусть теперь в файл с входными данными записан набор чисел, как на рис. 7.3, в многострочном виде (три строки и два столбца), следующие операторы считают данные в нужном порядке, инициализировав ими матрицу `A`:

```
open(8, file='init2D.dat', status='old', action='read')
read(8,*) ( (A(i,j), j=1, n), i=1, m )
```

7.6. Маскирование присваивания. Операторы и конструкции where и forall

В Фортране можно выполнять присваивание только тем элементам массива, которые удовлетворяют заданным условиям. Такая операция называется *маскированием присваивания* и реализуется, в частности, с помощью оператора или конструкции **where**, которые можно рассматривать как аналог условного оператора **if** для массивов. Общая форма записи конструкции следующая:

```
[имя:] where (ЛВ 1 — массив)
 операторы присваивания массивов     ! <Блок 1>
elsewhere (ЛВ 2 — массив) [имя]
 операторы присваивания массивов     ! <Блок 2>
elsewhere [имя]
 операторы присваивания массивов     ! <Блок 3>
end where [имя]
```

В данном случае каждое выражение в круглых скобках представляет собой маску в виде логического массива той же формы, что и массивы, с которыми производятся манипуляции в операторах присваивания.

Конструкция применяет операцию или набор операций в `<Блоке 1>` ко всем элементам массива, для которых `ЛВ 1` истинно. Затем операции в `<Блоке 2>` применяются ко всем элементам, для которых `ЛВ 1` ложно, а `ЛВ 2` истинно. В конце происходит выполнение операций из `<Блока 3>` по отношению ко всем элементам массива, для которых и `ЛВ 1`, и `ЛВ 2` ложны.

Можно использовать также однострочный оператор **where**, выполняющий операции присваивания применительно к тем элементам массива, для которых выражение маски в круглых скобках истинно:

```
where (ЛВ — массив) оператор присваивания массивов
```

Примеры использования оператора и конструкции where с задействованием различных логических выражений и их комбинаций:

```
integer, dimension(5) :: a = [ 1, -1, 1, -1, 1 ]
where( a > 0 ) a = a * 2    ! 2  -1  2  -1  2
real, dimension(5) :: b = [ 1., -1., 2., -1., 3. ]
```

```
where ( b > 0 ) b = log(b) !  0.0  -1.0  0.693147  -1.0  1.09861
real, dimension(5) :: c = [ 5., -0.5, -4., 0.1, 3. ]
where ( c > 1. )
  c = 1.
elsewhere ( c < -1. )
  c = -1.                    ! 1.0  -0.5  -1.0  0.1  1.0
end where
integer, dimension(6) :: d = [1, -1, 1, -1, 1, 0], e(-2:3) = 0
where ( d > e )              ! массивы должны быть согласованы
    e = e + 2
elsewhere
    e = e - 3                ! 2 -3  2 -3  2 -3
    d = d + e                ! 1 -4  1 -4  1 -3
endwhere
```

Фортран содержит специальную конструкцию **forall**, позволяющую применять набор операций поэлементно к подмножеству элементов массива. Элементы, над которыми требуется осуществлять операции, могут выбираться как по индексу, так и по логическому условию. Операции будут применены только к тем элементам, которые одновременно удовлетворяют ограничениям индексов (в спецификации триплета) и логическому условию.

Общая форма записи конструкции **forall**:

```
[имя:] forall (спецификация триплета [, спецификация триплета] &
               ... [, ЛВ — массив]))
  операторы конструкции
end forall [имя]
```

Операторами конструкции могут быть операторы присваивания, а также операторы или конструкции **where** и **forall**. Переменная, наделяемая значением, должна быть элементом массива или его сечением. Спецификация должна содержать имена всех индексов, включенных в триплеты.

Аналогично оператору **where** возможно использование однострочного оператора **forall**, выполняющего операции присваивания для тех индексов и логических выражений, которые удовлетворяют управляющим параметрам:

```
forall спецификация триплета [, спецификация триплета] &
                ... [, ЛВ — массив])) оператор присваивания
```

Приведем примеры использования конструкции и оператора **forall**, в первом из которых создается матрица с одинаковыми

элементами на главной диагонали и нулями на остальных местах, а во втором вычисляется обратная величина для всех положительных элементов матрицы:

```
integer, parameter :: n = 5
integer :: i, j
real :: eps = 1e-8
real, dimension(n,n) :: A = 0.
  forall ( i=1:n ) A(i,i) = 5.
  forall ( i=1:n, j=1:n, A(i,j) > eps )
    A(i,j) = 1. / A(i,j)
  end forall
```

Обработка операторов в теле конструкции **forall** происходит строго последовательно. В примере далее значения элементов матрицы A, рассчитанные в первом операторе, используются для вычисления элементов матрицы B во втором операторе, пересчет всех удовлетворяющих условию элементов A будет произведен до начала вычисления B:

```
forall ( i=2:n-1, j=2:n-1, A(i,j) > eps )
  A(i,j) = sqrt(A(i,j))
  B(i,j) = 1. / A(i,j)
end forall
```

В целом при использовании только что рассмотренных конструкций код программы выглядит более компактно по сравнению с поэлементными операциями, особенно при работе с многомерными массивами. Также уменьшается вероятность совершения ошибки.

7.7. Динамические массивы

Использование статических массивов ставит разработчика перед необходимостью предварительно задать их размеры. Во многих случаях эффективнее использовать *динамические (размещаемые) массивы*, размеры которых можно задавать и изменять в процессе выполнения программы.

Динамические массивы объявляются со специальным атрибутом **allocatable**, который добавляется к оператору объявления типа, например, для одномерного и двумерного массивов:

```
integer, allocatable, dimension(:) :: x      ! одномерный массив
real(8), allocatable, dimension(:,:) :: A    ! двумерный массив
```

Массив, объявленный с двоеточиями для обозначения размеров, называется *размещаемым* (с отложенной формой), поскольку задание фактической формы массива откладывается до момента выделения памяти.

Фактический размер массива указывается в операторе **allocate**, когда производится выделение памяти. Две возможные формы оператора:

```
allocate(список массивов [, stat=ier, errmsg=err_msg])
allocate(массив, source=src_array [,stat=ier, errmsg=err_msg])
```

В первой форме оператора параметр `список массивов` содержит одно или несколько разделенных запятыми имен массивов и их размеры в круглых скобках. Во второй форме `массив` получит форму как у массива `src_array`, а данные из исходного массива будут скопированы в только что размещенный массив.

Необязательный параметр `stat` (поэтому заключенный в квадратные скобки) возвращает целочисленный статус выполнения операции. В случае успешного размещения будет возвращен `0`, а символьная переменная в параметре `errmsg` изменена не будет, в противном случае возвратится ненулевой код ошибки, а в символьную переменную будет помещено описательное сообщение о возникшей проблеме.

Примеры выполнения оператора **allocate**:

```
integer :: ierror                          ! статус операции
character(len=80) :: err_msg           ! сообщение об ошибке
integer, allocatable, dimension(:) :: x, y
real(8), allocatable, dimension(:,:) :: A
 allocate(x(10))  ! выделение памяти для одномерного массива
 allocate(y, source=x)    ! выделение памяти и инициализация
  allocate(A(100, 0:10), stat=ierror, errmsg=err_msg)
```

В завершение программы или процедуры, в которой используется динамический массив, следует освободить память, чтобы сделать ее доступной для повторного использования. Освобождение памяти осуществляется оператором **deallocate** следующего вида:

```
deallocate(список массивов [, stat=ierror])
```

Параметр «список массивов», как и в операторе **allocate**, содержит одно или несколько разделенных запятыми имен массивов, а **stat** возвращает статус выполнения операции. После освобождения памяти массив может быть размещен повторно с новым или с прежним размером. Ранг при этом меняться не может.

Динамический массив нельзя использовать в программе, пока для него не будет выделена память. Любая такая попытка приведет к ошибке во время выполнения и к аварийному завершению программы. Внутренняя логическая функция `allocated` позволяет проверять состояние размещения массива перед использованием. Функция возвращает значение `.true.`, если массив в данный момент размещен, `.false.` – в противном случае:

```
real, allocatable, dimension(:,:) :: A
  allocate(A(5, 10))
  if (allocated(A)) then
    read(8,*) input_data
  else
    write(*,*) 'Ошибка выделения памяти!'
  end if
```

Причиной ошибки выделения памяти может быть недостаток памяти или попытка разместить ранее размещенный и не освобожденный с помощью **deallocate** объект.

7.8. Примеры программ

Приведем код программы, вычисляющей первую норму вектора как сумму модулей всех его элементов в двух вариантах – с помощью арифметического цикла и встроенной функции:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|.$$

```
program sum_vector_elements
! Программа вычисления суммы модулей всех элементов массива
integer :: n, i                ! размер массива, индекс цикла
real(8) :: x1, x2                    ! результаты вычислений
real(8), allocatable, dimension(:) :: x ! объявление массива
  write(*,*) 'Введите размер массива: '; read(*,*) n
```

```
  allocate(x(n))                              ! выделение памяти
  x = -1.; x1 = 0.                              ! инициализация
! Вариант с использованием арифметического цикла
  do i = 1, n
    x1 = x1 + abs(x(i))
  end do
  100 format(a,1x,f12.1)
  write(*,100) 'Первая норма вектора (1):', x1
! Вариант с использованием встроенной функции компактнее
  x2 = sum(abs(x))
  write(*,100) 'Первая норма вектора (2):', x2
  deallocate(x)                             ! освобождение памяти
end program sum_vector_elements
```

Напишем новую версию программы, печатающей *таблицу соответствия между градусами Цельсия и Фаренгейта* в заданном диапазоне изменения температуры, в которой задействуются одномерные массивы (см. код без использования массивов в п. 5.4).

```
program Celsius_2_Fahrenheit_array
! Печать таблицы соответствия градусов Цельсия и Фаренгейта
implicit none
integer(2), parameter :: n= 20  ! конечное значение индекса цикла
integer(2) :: i                                    ! индекс цикла
real, dimension(n+1) :: T_c, T_f          ! массивы для температур
 write(*,*) 'Таблица соответствия между температурными шкалами &
             Цельсия и Фаренгейта'
! Цикл с пересчетом температуры и печатью данных
  do i = 1, n+1
    T_c(i) = 5. * (i-1)
    T_f(i) = 32. + 9.* T_c(i) / 5.
    write(*,'(2(a5,1x,f5.1))') 'Tc =', T_c(i), 'Tf =', T_f(i)
  end do
end program Celsius_2_Fahrenheit_array
```

Напишем программу расчета *среднего арифметического, среднего квадратического* и *медианного значений* для набора входных данных (см. код без использования массивов в п. 5.4). Алгоритм программы предполагает последовательное выполнение следующих действий: открытие входного файла с данными и их считывание в массив, сортировка данных в порядке возрастания, вычисление и вывод интересующих значений:

```
program stats_calculations_array
! Программа для выполнения статистических вычислений
implicit none
```

```
integer, parameter :: max_size = 100      ! максимальный размер
real, dimension(max_size) :: a  ! массив данных для сортировки
logical :: exceed = .false.
character(len=24) :: fn = 'input.dat'  ! файл с входными данными
character(len=80) :: err_msg              ! сообщение об ошибке
integer(2) :: ierror, i, j, iptr, nvals
real(8) :: temp, sum_x, sum_x2, x_bar, x_rms, median
  nvals = 0
! Считывание набора данных из файла
  open(8, file=fn, status='old', action='read', &
       iostat=ierror, iomsg=err_msg)
 fileopen: if ( ierror == 0 ) then        ! файл открыт успешно
    do
      read(8,*, iostat=ierror) temp
     if ( ierror /= 0 ) exit   ! выход, когда закончатся данные
      nvals = nvals + 1
      checksize: if ( nvals <= max_size ) then
       a(nvals) = temp          ! сохранение значения в массиве
      else
       exceed = .true.                   ! переполнение массива
      end if checksize
    end do
    close(8)
! Проверка превышения размера массива, сообщение и выход
    toomuchdata: if ( exceed ) then
      write(*,*) 'Достигнут максимальный размер массива'
    else
! Предел не превышен: сортировка данных
      outer: do i = 1, nvals-1
! Поиск минимального значения в диапазоне от a(i) до a(nvals)
        iptr = i
        inner: do j = i+1, nvals
          minval: if ( a(j) < a(iptr) ) then
            iptr = j
          end if minval
        end do inner
! iptr теперь указывает на минимальное значение, смена мест
        swap: if ( i /= iptr ) then
          temp = a(i); a(i) = a(iptr); a(iptr) = temp
        end if swap
      end do outer
! Расчет статистики и печать результатов
      sum_x  = 0.; sum_x2  = 0.
      sums: do i = 1, nvals
```

```
        sum_x  = sum_x + a(i)
        sum_x2 = sum_x2 + a(i)**2
      end do sums
      x_bar = sum_x / nvals
      x_rms = sqrt( sum_x2 / nvals )
      even: if ( mod(nvals,2) == 0 ) then
        median = ( a(nvals/2) + a(nvals/2+1) ) / 2.
      else
        median = a(nvals/2+1)
      end if even
      write(*,'(a,f6.3)') 'Среднее арифметическое:', x_bar
      write(*,'(a,f6.3)') 'Среднее квадратическое:', x_rms
      write(*,'(a,f6.3)') 'Медиана:', median
      write(*,'(a,i3)')   'Размер массива:', nvals
    end if toomuchdata
  else fileopen
    write(*,*) 'Ошибка открытия файла, error = ', trim(err_msg)
  end if fileopen
end program stats_calculations_array
```

Для проверки правильности работы программы предлагается взять набор входных значений (`5., 3., 4., 1., 9.`), результаты при этом должны быть следующими: среднее арифметическое значение — `4.4`, медианное значение — `4.0`, среднее квадратическое значение приближенно равно `5.138`.

Приведем код программы, которая считывает из файла квадратную матрицу и заменяет нулями все элементы ниже главной и побочной диагоналей. В коде выполняется файловый ввод/вывод матриц, динамическое выделение памяти, а для преобразования матрицы используются сечения:

```
program matrix_change_elements
! Программа замены нулями всех элементов ниже диагоналей матрицы
implicit none
integer :: i, j, n                ! индексы цикла и размер матрицы
integer, allocatable, dimension(:,:) :: A   ! объявление матрицы
! Открытие файлов и считывание размера и элементов матрицы
  open(8,file='input.dat', status='old', action='read')
  open(9,file='output.dat', status='replace',action='write')
  read(8,*) n; allocate(A(n,n))
  read(8,*) ( (A(i,j), j=1, n), i=1, n )
  close(8)                    ! закрытие файла с исходной матрицей
   ! Цикл замены нулями элементов ниже главной и побочной
```

```
    ! диагоналей матрицы
      do i = n/2 + 2, n
        A(i, n-i+2:i-1) = 0
      end do
    ! Вывод размера и элементов результирующей матрицы в файл
      write(9,'(i2)') n
      do i = 1, n
        write(9,'(*(i2))') (A(i, 1:n))
      end do
      deallocate(A)                          ! освобождение памяти
      close(9)          ! закрытие файла с результирующей матрицей
end program matrix_change_elements
```

Напишем программу сложения двух квадратных матриц с формированием верхней треугольной матрицы, ограничением модуля значений элементов (замена нулем при превышении) и умножением элементов на диагонали на число с использованием операторов маскирования присваивания:

```
program forall_where_matrix
! Работа с матрицами с маскированием присваивания
integer :: i, j, n                  ! индексы цикла и размер матриц
real, allocatable, dimension(:,:) :: A, B, C
character(len=20) :: fmt
  write(*,*) 'Введите размер матриц: '; read(*,*) n
  allocate(A(n,n), B(n,n), C(n,n))
! Инициализация матриц
  forall (i = 1:n, j = 1:n) A(i, j) = 2. / real(i + j)
  forall (i = 1:n, j = 1:n) B(i, j) = 1. / real(i + j)
  C = 0.
! Выполнение вычислений и преобразований элементов матрицы
  forall (i = 1:n, j = 1:n, i <= j) C(i, j) = A(i, j) + B(i, j)
  where (abs(C) > 0.8) C = 0.
  forall (i = 1:n) C(i, i) = C(i, i) * 5.
  write(fmt,*) n
  write(*,'('//fmt//'(f12.4,1x))') ((C(i, j), j = 1,n), i = 1,n)
end program forall_where_matrix
```

7.9. Варианты заданий

Напишите программы для приведенных далее вариантов заданий, используя (при наличии возможности) все имеющиеся способы работы с массивами — поэлементно в арифметических циклах, с именованными массивами целиком и встроенными функциями.

Инициализацию массивов выполнить любым способом. Вычисления произвести для двух размеров массивов, отличающихся на порядок (например, для векторов – 10^8 и 10^9 элементов). Выполнить с помощью встроенной функции `cpu_time` замеры времени работы вычислительных частей кода и сопоставить их для разных реализаций.

7.9.1. Математический блок

1. В декартовой системе координат двумерный вектор представляется суммой его проекций на координатные оси:

$$\vec{V} = V_x\vec{i} + V_y\vec{j}.$$

Для записи вектора в полярных координатах осуществляется перевод:

$$V_x = \rho\sin(\theta),\ V_y = \rho\cos(\theta),$$

где ρ и θ – полярные координаты. Модуль вектора и угол его наклона можно определить по следующим формулам:

$$\rho = |V| = \sqrt{V_x^2 + V_y^2},\ \theta = \begin{cases} \operatorname{arctg}\left(\dfrac{V_y}{V_x}\right), V_x \neq 0, \\ \dfrac{\pi}{2}, V_x = 0. \end{cases}$$

Напишите программу, осуществляющую перевод вектора, заданного в полярной системе координат, в декартову, и наоборот, перевод вектора, заданного в декартовой системе координат в полярную.

2. Заданы два вектора $\vec{a} = a_x\vec{i} + a_y\vec{j} + a_z\vec{k}$, $\vec{b} = b_x\vec{i} + b_y\vec{j} + b_z\vec{k}$.

Вычислите скалярное произведение этих векторов по формуле

$$\vec{a}\cdot\vec{b} = a_xb_x + a_yb_y + a_zb_z.$$

3. Вычислите минимальное и максимальное значения функции при движении с указанным шагом h по заданному интервалу:

а) $f(x) = x^3 - 2.5x^2 - 2x + 1.5,\ x \in [-1, 3],\ h = 0.01$;

б) $f(x) = \cos(x),\ x \in [-\pi/4,\ 5\pi/4],\ h = \pi/30$;

в) $g(x, y) = x^2 + y^2,\ x \in [-1,\ 1],\ y \in [-2,\ 2],\ h = 0.05$.

4. Выполните умножение всех элементов вектора на вещественное число (здесь и далее под вектором понимается одномерный массив).

5. Вычислите вторую (евклидову) и чебышевскую нормы вектора:

а) $\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$; б) $\|x\|_\infty = \max_{i=1,N} \{|x_i|\}$.

6. Вычислите произведение всех элементов вектора.

7. Вычислите среднее значение для элементов вектора.

8. Найдите минимальное и максимальное значения среди всех элементов вектора и их порядковые номера.

9. Вычислите среднеквадратическое отклонение для вектора.

10. Прибавьте ко всем элементам вектора ненулевое вещественное число.

11. Подсчитайте количество ненулевых элементов матрицы.

12. Подсчитайте количество ненулевых элементов вектора.

13. Выполните суммирование (вычитание) двух векторов.

14. В каждой строке матрицы найдите наибольшее значение и из всех найденных значений выберите наименьшее.

15. Поменяйте местами строки матрицы, содержащие наименьшее и наибольшее значения.

16. Выполните сложение (поэлементно) двух матриц.

17. Прибавьте ко всем элементам матрицы ненулевое вещественное число.

18. Выполните умножение всех элементов матрицы на ненулевое вещественное число.

19. Вычислите первую, евклидову и чебышевскую нормы матрицы:

а) $\|A\|_1 = \max_j \left\{ \sum_{i=1}^{n} |a_{ij}| \right\}$; б) $\|A\|_E = \sqrt{\sum_{i,j=1}^{n} |a_{ij}|^2}$;

в) $\|A\|_\infty = \max_i \left\{ \sum_{j=1}^{n} |a_{ij}| \right\}$.

20. Вычислите среднее значение по всем элементам матрицы.

7.9.2. Физический блок

Выполните задания 1–5 из пп. 5.5.2, задействуя массивы.

1. Соотношение между вектором угловой скорости $\vec{\omega}$ движущегося объекта и его линейной скоростью $\vec{V}$ записывается в виде $\vec{V}=\vec{\omega}\times\vec{r}$, где $\vec{r}$ – радиус-вектор, построенный от точки вращения до объекта. Напишите программу, которая для заданных значений $\vec{\omega}=\omega_x\vec{i}+\omega_y\vec{j}+\omega_z k$ и $\vec{r}=x\vec{i}+y\vec{j}+z\vec{k}$ определяет вектор $\vec{V}=V_x\vec{i}+V_y\vec{j}+V_z\vec{k}$. Формула для векторного произведения приведена в задании 7 из пп. 3.11.1.

2. Рассматривается плоское ламинарное течение вязкой несжимаемой жидкости (с плотностью ρ и коэффициентом динамической вязкости μ), движущейся между двумя плоскостями $y = \pm h$ (течение жидкости сквозь плоский канал). Распределение скоростей w для такого течения представляется параболой:

$$w=\frac{\Delta p h^2}{2\mu L}\left[1-\left(\frac{y}{h}\right)^2\right],$$

где Δp – перепад давления в канале длины L. В расчетах следует взять значения Δp из диапазона $0.5\ldots2$ Па, $L = 1\ldots40$ м и высоты трубы $2h = 0.01$ м:

а) выведите таблицу значений w в поперечном сечении плоского канала;

б) рассчитайте максимальную скорость w_{max} в плоском канале и соответствующее значение координаты y; сравните полученное значение с точным $w_{max} = 0.5\ \Delta p h^2/(\mu L)$;

в) вычислите среднюю по перечному сечению скорость w_{cp}; сравните полученное значение с точным $w_{cp} = 2/3\ w_{max}$;

г) вычислите секундный объемный расход Q жидкости:

$$Q=\int_S \rho\left(\vec{V}\cdot\vec{n}\right)dS,$$

сравните полученное значение с точным $Q = 2/3\ \Delta p h^3/(\mu L)$;

д) вычислите коэффициент сопротивления движению жидкости λ сквозь щель между плоскостями, где λ = 24/Re и Re = = 2hρwcp/μ – число Рейнольдса.

3. Рассматривается установившееся ламинарное течение вязкой несжимаемой жидкости, движущейся с постоянной скоростью w по цилиндрической трубе эллиптического сечения с полуосями a и b. Распределение скоростей в сечении для такого течения имеет следующий вид (см. [23]):

$$w = \frac{\Delta\rho}{2\mu L}\frac{a^2b^2}{a^2+b^2}\left[1-\frac{x^2}{a^2}-\frac{y^2}{b^2}\right].$$

В расчетах следует взять значения перепада давления Δp в диапазоне 0.5…2 Па, длины трубы L – 1…40 м и полуосей a и b – 0.5…1 см.

а) выведите таблицу значений w в поперечном сечении трубы;

б) рассчитайте максимальную скорость w_{max} и соответствующие (x, y); сравните полученное значение с точным w_{max} = Δpa2b2/(2μL(a2+b2));

в) вычислите среднюю по перечному сечению скорость w_{cp}; сравните полученное значение с точным w_{cp} = 0.5 w_{max};

г) вычислите секундный объемный расход Q жидкости;

д) вычислите коэффициент сопротивления движению жидкости λ по формуле λ = 64/Re, где Re = 0.5(a+b)ρw_{cp}/μ – число Рейнольдса.

4. Для заданного коэффициента a вычислите циркуляцию скорости жидкости по отрезку L прямой, соединяющей точки A(1,0) и B(0,1); заданы проекции вектора скорости:

$$V_x = -\frac{ay}{\left(x^2+y^2\right)},\ V_y = \frac{ax}{\left(x^2+y^2\right)},\ V_z = 0.$$

Циркуляция вычисляется по такой формуле:

$$\int_L \vec{V}\cdot d\vec{r}.$$

Глава 8

ВВЕДЕНИЕ В ФУНКЦИИ И ПОДПРОГРАММЫ

8.1. Процедуры в языке Фортран. Общие сведения

В п. 2.3 был рассмотрен востребованный при разработке программ принцип «сверху вниз», предполагающий разбиение исходной задачи на подзадачи (фрагменты), с которыми далее выполняются необходимые операции. Кодирование фрагментов может производиться с помощью таких программных единиц, как головная программа, модули, функции и подпрограммы.

Функции и подпрограммы представляют собой именованные программные единицы и называются в Фортране *процедурами*. Синтаксис программных единиц приведен в табл. 8.1.

Процедуры могут быть внешними, внутренними и модульными. *Внешняя процедура* – самостоятельная программная единица, существующая независимо от вызывающих ее единиц. К интересующей внешней процедуре можно обратиться из головной программы или из другой внешней процедуры.

Таблица 8.1

Синтаксис программных единиц в языке Фортран

Головная программа	Подпрограмма	Функция
program имя <операторы объявления> <исполняемые операторы> **contains** <внутренние процедуры> **end program** [имя]	**subroutine** имя (список формальных параметров) <операторы объявления > <исполняемые операторы> [**contains** <внутренние процедуры>] **end subroutine** [имя]	тип **function** имя (список формальных параметров) <операторы объявления > <исполняемые операторы> имя = <выражение> [**contains** <внутренние процедуры>] **end function** [имя]

Внутренние процедуры могут задаваться в головной программе, внешней или модульной процедуре (называемые в этом контексте *носителями*), а их код следует после оператора **contains**. Вызов внутренних процедур возможен только изнутри своих носителей, а они не могут содержать в себе другие программные единицы.

Как и любая другая программная единица, внешняя процедура содержит секции объявления и исполнения. Когда осуществляется вызов внешней процедуры, выполнение вызывающей ее программы приостанавливается и запускается раздел с кодом процедуры. После достижения оператора завершения (возврата) программа продолжит работу со строки, следующей за вызовом внешней процедуры.

Процедуры взаимодействуют с вызывающими их программными единицами через перечисляемый в описании список скалярных переменных и/или массивов, которые называются *параметрами* и играют роль входных данных и результатов работы. Эти параметры носят название *формальных*, так как они используются в процедуре только для кодирования алгоритма, и память для них не выделяется.

Вызов процедуры сопровождается подстановкой на место формальных параметров переменных с заданными значениями или констант – *фактических* параметров (в виде указателей на местоположение в памяти). Такое связывание называется *ассоциированием*. Процедура просматривает области памяти, на которые указывает вызывающая программная единица, для получения значений формальных параметров. Порядок следования и типы формальных и фактических параметров должны совпадать (если не заданы ключевые слова), имена переменных при этом могут отличаться от имен соответствующих формальных параметров. Поскольку каждая программная единица компилируется отдельно, имена формальных параметров могут повторно использоваться в различных программных единицах, не вызывая ошибок.

Вид связи формальных и фактических параметров задается атрибутом оператора объявления типа **intent** со следующими возможными значениями и особенностями использования соответствующего формального аргумента:

– **intent** (in) – для передачи входных данных в процедуру;

– **intent**(out) – для возврата результатов вызывающей программе;

– **intent**(inout) – как для передачи входных данных в процедуру, так и для возврата результатов в вызывающую программу.

При помощи этого атрибута можно задать компилятору ограничения на модификацию фактических параметров и т. п.

Преимуществами использования процедур являются возможности их независимого проектирования, кодирования, тестирования и отладки, повторного использования кода в случаях, когда решение одной и той же подзадачи требуется во многих частях программы, а также определенная степень изоляции от возможных проблем с учетом влияния процедур только на переменные, присутствующие в списке параметров.

8.2. Подпрограммы и функции

Фрагмент алгоритма, возвращающий более одного скаляра и/или массива или модифицирующий какие-либо из входных данных, оформляется как процедура типа *подпрограмма*. Начало подпрограммы объявляется с помощью оператора типа **subroutine**, который определяет ее имя и связанный список формальных параметров. Вызов подпрограммы осуществляется оператором call по имени с указанием списка фактических параметров:

call имя_подпрограммы(фактические_параметры).

Приведем пример программы с вызовом подпрограммы вычисления гипотенузы прямоугольного треугольника по двум заданным катетам:

```
program hypotenuse
! Программа вычисления гипотенузы прямоугольного треугольника
implicit none
real :: s1, s2, h                    ! длины катетов и гипотенузы
  write(*,*) 'Введите длины катетов: '
  read (*,*) s1, s2                          ! ввод длин катетов
  call calc_hypotenuse( s1, s2, h )        ! вызов подпрограммы
  write(*,'(f12.10)') h
end program hypotenuse
```

```
! Вычислительная подпрограмма
subroutine calc_hypotenuse( side1, side2, hypot )  ! объявление
implicit none
real, intent(in) :: side1, side2   ! входные параметры (катеты)
real, intent(out) :: hypot      ! выходной параметр (гипотенуза)
  hypot = sqrt( side1**2 + side2**2 )
end subroutine calc_hypotenuse
```

Следует отметить, что возведение в целую степень производится существенно быстрее, чем в вещественную, поскольку последнее выполняется через тождество вида `x**y = exp(y * log(x))`.

В случае использования в качестве формального параметра символьной переменной (строки) ее длина объявляется символом звездочки:

```
subroutine sub_character ( s )
character(len=*), intent(in) :: s
  write(*,'(a,i3)') 'Длина строки: ', len(s)
end subroutine sub_character
```

Поскольку память для формального параметра не выделяется, то нет необходимости знать длину строки на этапе компиляции подпрограммы.

Функция – это тип процедуры, результатом выполняемых вычислений в которой является единственное число, логическое значение, строка или массив (в более общем случае – единственный объект). Существует два типа функций – *встроенные* (см. п. 3.8) и *пользовательские*.

Пользовательская функция объявляется оператором **function** с указанием типа возвращаемого результата, имени и списка формальных параметров. Вызов функции производится по имени из выражения, фигурирующего в правой части оператора присваивания или оператора вывода.

Имя функции обязательно должно появиться в теле функции слева в операторе присваивания. Присвоенное таким образом значение будет являться значением функции при возврате в вызывающую программную единицу. Список параметров функции может быть пустым, если функция может выполнять вычисления без входных параметров.

Таблица 8.2

Объявление типа функции

№ п/п	Вариант с объявлением перед оператором **function**	Вариант с указанием типа на отдельной строке
1	`integer function f( i, j )`	`function f( i, j )` `integer :: f`

Поскольку функция возвращает значение, ей необходимо присвоить тип. В случае указания оператора **implicit none** тип функции должен быть объявлен в функции. Объявление типа функции может принимать одну из двух эквивалентных форм, приведенных в табл. 8.2.

В следующем примере приведен код программы вычисления квадратичного полинома `f(x)=ax2 + bx + c` с вызовом пользовательской функции:

```
program quad_polynomial
! Программа вычисления квадратичного полинома
implicit none
real :: a, b, c, x                ! объявление локальных переменных
  write(*,*) 'Введите коэффициенты полинома a, b, и c: '
  read (*,*) a, b, c
  write(*,*) 'Введите точку вычисления значения полинома: '
  read (*,*) x
  write(*,'(a,f10.4,a,f12.4)') 'qpol(',x,') =', qpol(x, a, b, c)
end program quad_polynomial
! Вычислительная функция
real function qpol( x, a, b, c )              ! объявление функции
implicit none
real, intent(in) :: x, a, b, c
  qpol = a * x**2 + b * x + c
end function qpol
```

Функции и подпрограммы могут передаваться в качестве формальных параметров при их объявлении *внешними* в вызывающих и вызываемых программных единицах с помощью оператора или атрибута **external**:

```
external :: sub
real, external :: fun
```

Дополнительные примеры кодов программ, в которых используются подпрограммы и пользовательские функции, приведены в гл. 9.

8.3. Варианты заданий

8.3.1. Математический блок

Напишите программы, в которых используются функции и подпрограммы, для заданий из пунктов: 1) 3.11.1 – № 2, 3, 5, 6, 7, 9, 10; 2) 4.5.1 – № 1–5; 3) 5.5.1 – № 2, 3, 6, 10–14; 4) 7.9.1 – все задания.

Выполните следующие задания с использованием в кодах программ внутренней и внешней пользовательской функции.

1. Вычислите на некотором отрезке с заданным шагом все значения аргумента и функции:

а) $\mathrm{sh}(x)=\frac{e^{x}-e^{-x}}{2}$; б) $\mathrm{ch}(x)=\frac{e^{x}+e^{-x}}{2}$; в) $\mathrm{th}(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$.

2. Численно решите уравнение `f(x) = 0` для нелинейной функции на заданном интервале методами: 1) половинного деления; 2) простых итераций; 3) хорд; 4) секущих:

а) $2\mathrm{arcctg}(x)-x-3=0$; б) $x^{2}-20\sin(x)=0$; в) $x^{3}-2x^{2}-4x+7=0$.

3. Найдите точку пересечения двух функций на заданном интервале:

а) $f(x)=\sqrt{x}$, $g(x)=\frac{1}{x}$; б) $f(x)=x^{3}$, $g(x)=x+7$;

в) $f(x)=x$, $g(x)=\sin(x)$.

4. Приближенно вычислите значения определенного интеграла функции на заданном интервале: 1) методом средних прямоугольников; 2) по формуле трапеций; 3) по формуле Симпсона:

а) $\int_{0}^{1}\frac{dx}{\sqrt{x^{3}+1}}$; б) $\int_{0}^{\frac{\pi}{2}}x\sin(x)dx$; в) $\int_{0}^{1}\left(x^{2}+x\right)dx$.

8.3.2. Физический блок

Напишите программы с использованием функций и подпрограмм для заданий из пунктов: 1) 4.5.2 – № 1, 2, 5; 2) 5.5.2 – № 3, 4, 5; 3) 7.9.2 – все задания.

Глава 9
ПРИМЕРЫ АЛГОРИТМОВ И ПРОГРАММНЫХ РЕАЛИЗАЦИЙ НА ЯЗЫКЕ ФОРТРАН

В данной главе приведены дополнительные примеры исходных кодов вычислительных программ на языке Фортран, включающие работу со скалярными переменными и массивами разной размерности.

В качестве вычислительных экспериментов, сопровождающих изучение кодов, можно предложить проведение замеров времени работы программ с помощью команды Unix `time` или встроенной функции Фортрана `cpu_time` при варьировании размеров операндов, точности вычислений, использовании разных вариантов реализации операций, уровней оптимизации (`-O0/1/2/3`) и др. Встроенная функция позволяет относительно грубо оценить время выполнения вычислительных частей программ без учета служебных и инициализирующих операций.

В некоторых примерах работы с массивами использованы разные варианты реализации алгоритмов, в том числе с помощью арифметического цикла, векторной формы, встроенной функции, оператора маскирования присваивания. Память для массивов выделяется преимущественно динамически.

Инициализация массивов может производиться с помощью встроенной подпрограммы `random_number`, возвращающей массив (или скаляр) вещественных псевдослучайных чисел из равномерного распределения по диапазону `[0, 1]`. Массив генерируется на основе затравочной (seed) последовательности целых чисел, переустанавливаемой или возвращаемой подпрограммой `random_seed`.

9.1. Нахождение корней уравнения

Программа нахождения корня заданной функции итерационным численным методом Ньютона (касательных). Вид функции: `f(x)= cos(x)- x`^{3}; начальное приближение и погрешность вычислений задаются в качестве параметров вызова программы

(например, `0.5` и `1e-8` соответственно). Формула вычисления очередного приближения для корня функции:

$$x_{k+1} = x_k - \frac{f\left(x^k\right)}{f'\left(x^k\right)},\ k = 0,1,\ldots,\ f'(x) \equiv \frac{df}{dx}(x).$$

```
program Newton_root
! Программа численного решения уравнения методом Ньютона
implicit none
character(len = 20), dimension(2) :: args
real(8) :: x, dx, fx, dfx, eps       ! переменные для вычислений
! Проверка наличия аргументов вызова программы и их считывание
  if (command_argument_count() /= 2) then
    write(*,*) 'Ошибка: требуются два аргумента &
        вызова программы - начальное приближение и погрешность'
    stop
  end if
  call get_command_argument(1, args(1))  ! начальное приближение
  call get_command_argument(2, args(2))           ! погрешность
  read(args(1),*) x; read(args(2),*) eps ! считывание аргументов
! Основной вычислительный цикл с вызовом функций
  do
    fx = f(x);  dfx = df(x)        ! вычисление f(x) и f'(x)
    write(*,'(2(f14.12,1x))') x, fx          ! вывод приближения
    dx = -fx / dfx; x = x + dx   ! вычисление нового приближения
    if ( abs( fx ) < eps ) exit    ! проверка условия сходимости
  end do
  write(*,'(a,f13.10)') 'Корень уравнения:', x     ! 0.8654740331
contains
! Функция вычисления значения функции f(x) = cos(x)-x3
  real(8) function f(x)
  implicit none
  real(8), intent(in) :: x
    f = cos(x) - x**3
  end function f
! Функция вычисления значения производной f'(x) = -sin(x)-3x2
  real(8) function df(x)
  implicit none
  real(8), intent(in) :: x
    df = -sin(x) - 3 * x**2
  end function df
end program Newton_root
```

9.2. Интерполирование функции

Программа интерполирования функции с помощью интерполяционного полинома в форме Лагранжа (задана таблица с дискретным набором узлов (x_i, y_i) и точка x для вычисления значения функции):

$$L_n(x) = \sum_{i=0}^{n} \left[y_i \prod_{j=0, j \neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)} \right].$$

```
program iterpolation_Lagrange
! Программа интерполяции функции полиномом в форме Лагранжа
implicit none
integer, parameter :: n = 4            ! число дискретных узлов
integer :: i, j                                ! индексы цикла
real, dimension(n) :: x, y                     ! массивы узлов
real :: Ln, xp, p                  ! переменные для вычислений
! Основной вычислительный цикл
  x = [ 3.2, 2.7, 1.0, 4.8 ]; y = [ 22.0, 17.8, 14.2, 38.3 ]
  Ln = 0.; xp = 3.5
  do i = 1, n                   ! цикл по числу дискретных узлов
    p = 1.                     ! стартовое значение произведения
    do j = 1, i-1                 ! цикл вычисления произведения
      p = p * ( xp - x(j) ) / ( x(i) - x(j) )
    end do
    do j = i+1, n                 ! цикл вычисления произведения
      p = p * ( xp - x(j) ) / ( x(i) - x(j) )
    end do
    Ln = Ln + y(i) * p
  end do
  write(*,'(a,f3.1,a,f6.3)') 'f(', xp, ') =', Ln ! f(3.5)=24.889
end program iterpolation_Lagrange
```

9.3. Численное дифференцирование

Программа приближенного вычисления значения производной функции одной переменной со вторым порядком точности на основе центрально-разностной схемы:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}.$$

Дифференцируемая функция `sin(x)`; дискретный набор значений переменной x для вычисления производной берется из интервала [0, π/2]:

```
program numerical_differentiation
! Программа численного дифференцирования функции
implicit none
integer, parameter :: n = 4                    ! число значений x
integer :: i                                   ! индекс цикла
real(8) :: x, step, h, df     ! переменные для вычислений
  h = 1.e-5; step = 2. * atan(1._8) / n        ! π/2 = 2·arctg(1)
! Основной цикл численного дифференцирования с вызовом функции
  do i = 0, n
    x = i * step                                ! значение x
    df = ( f(x + h) - f(x - h) ) / ( 2. * h )   ! производная
    write(*,'(3(f12.10,1x))') x, df, cos(x)  ! вывод результатов
  end do
contains
! Дифференцируемая функция: sin(x)
  real(8) function f(x)
  implicit none
  real(8), intent(in) :: x
    f = sin(x)
  end function f
end program numerical_differentiation
```

9.4. Численное интегрирование

Программа вычисления приближенного значения определенного интеграла функции `f(x) = 4/(1+x²)` на интервале `[0, 1]` методом средних прямоугольников по формуле (результат интегрирования равен числу π):

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f\left(x_{i-1/2}\right)h.$$

```
program numerical_integration
! Программа вычисления определенного интеграла
implicit none
integer :: n, i              ! число прямоугольников, индекс цикла
real(8) :: x, h, s                    ! переменные для вычислений
  write(*,*) 'Введите число прямоугольников: '; read(*,*) n
  s = 0.; h = 1._8/real(n,8)          ! величина шага по оси абсцисс
! Основной цикл вычисления интеграла с вызовом функции
```

```
  do i = 1, n
    x = (i - 0.5) * h
    s = s + f(x)
  end do
  write(*,'(a,2(f13.10,1x))') 'Рассчитанное и точное значения &
           интеграла:', s * h, 4.*atan(1._8)       ! π = 4·arctg(1)
contains
! Подынтегральная функция: f(x) = 4/(1+x2)
  real(8) function f(x)
  implicit none
  real(8), intent(in) :: x
    f = 4. / (1. + x * x)
  end function f
end program numerical_integration
```

9.5. Сортировка списка методом пузырька

Программа *сортировки методом пузырька* списка (массива) из псевдослучайных чисел с выделением процесса сортировки в подпрограмму. Алгоритм основан на последовательном сравнении и обмене соседних элементов списка, если предшествующий оказывается больше последующего. В результате элементы с бóльшими значениями оказываются в конце списка, а элементы с меньшими значениями постепенно перемещаются в направлении начала («всплывают»):

```
program bubble_sort_array
! Программа сортировки списка методом пузырька
implicit none
integer :: n, i                    ! размер списка, индекс цикла
integer, allocatable, dimension(:) :: list              ! список
real :: r                                  ! псевдослучайное число
  write(*,*) 'Введите размер списка: '; read(*,*) n
  allocate(list(n))                ! выделение памяти для списка
  do i = 1, n
    call random_number( r )
   list(i) = int(n*r + 1) ! случайные числа из диапазона [1, n]
  end do
  write(*,'(*(i6))') list                ! вывод исходного списка
  call bubble_sort( list )       ! вызов подпрограммы сортировки
  write(*,'(*(i6))') list        ! вывод отсортированного списка
contains
! Подпрограмма сортировки
  subroutine bubble_sort( x )
```

```
  implicit none
  integer, intent(inout) :: x(:)                          ! список
  integer :: num, j, k          ! размер списка и индексы циклов
  real :: temp
  logical :: sorted = .false.          ! флаг проверки сортировки
    k = 0; num = size(x)                              ! размер списка
    sorting: do while (.not. sorted)                  ! основной цикл
       sorted = .true.
       k = k + 1                                          ! новый шаг
       swapping: do j = 1, num - k                    ! цикл обменов
          if (x(j) > x(j+1)) then
            temp = x(j); x(j) = x(j+1); x(j+1) = temp
           sorted = .false.             ! перестановка произведена
          end if
       end do swapping
    end do sorting
  end subroutine bubble_sort
end program bubble_sort_array
```

9.6. Поиск максимального элемента вектора

Программа поиска максимального значения среди элементов вектора из псевдослучайных чисел с замером времени выполнения вычислительных фрагментов кода (если все элементы неотрицательные, результат представляет собой чебышевскую норму вектора):

```
program max_vector
! Программа поиска максимального элемента вектора
implicit none
integer :: n, i      ! число элементов в векторе, индекс цикла
real(8) :: r1, r2                          ! результаты вычислений
real :: t1, t2                  ! переменные для замера времени
real(8), allocatable, dimension(:) :: x  ! объявление вектора
  write(*,*) 'Введите размер вектора: '; read(*,*) n
  allocate(x(n))                   ! выделение памяти для вектора
 call random_number(x) ! инициализация псевдослучайными числами
! Способ с использованием арифметического цикла
  call cpu_time(t1)                                ! время до цикла
  r1 = x(1)
  do i = 2, n
    if (x(i) > r1) r1 = x(i)
  end do
  call cpu_time(t2)                             ! время после цикла
  write(*,'(a,1x,f12.8)') 'Максимальный элемент вектора:', r1
```

```
  write(*,'(a,1x,f6.3,1x,a)') 'Время расчета: ', t2 - t1, 'c'
! Способ с использованием встроенной функции
  call cpu_time(t1)                                ! время до цикла
  r2 = maxval(x)
  call cpu_time(t2)                              ! время после цикла
  write(*,'(a,1x,f12.8)') 'Максимальный элемент вектора:', r2
  write(*,'(a,1x,f6.3,1x,a)') 'Время расчета: ', t2 - t1, 'c'
  deallocate(x)                                  ! освобождение памяти
end program max_vector
```

9.7. Вычисление скалярного произведения векторов

Программа вычисления скалярного произведения векторов по формуле:

$$r = x \cdot y = \sum_{i=1}^{n} x_i y_i.$$

```
program scalar_product
! Программа вычисления скалярного произведения векторов
implicit none
integer :: n, i                  ! размер векторов, индекс цикла
real(8) :: r1, r2       ! результаты вычислений
real(8), allocatable, dimension(:) :: x, y ! объявление векторов
  write(*,*) 'Введите размер векторов: '; read(*,*) n
  allocate(x(n), y(n))             ! выделение памяти для векторов
  call random_number(x); call random_number(y)   ! инициализация
! Способ с использованием арифметического цикла
  r1 = 0.
  do i = 1, n
    r1 = r1 + x(i) * y(i)
  end do
! Способ с использованием встроенной функции
  r2 = dot_product(x, y)
  write(*,'(a, 2(es23.15))') 'Скалярное произведение:', r1, r2
  deallocate(x, y)                               ! освобождение памяти
end program scalar_product
```

9.8. Умножение матрицы на вектор

Программа умножения квадратной матрицы на вектор двумя способами с выводом результирующего вектора в файл:

$$y_i = \sum_{j=1}^{n} A_{ij} x_j, i = 1, \ldots, n.$$

```
program matrix_vector_mult
! Программа умножения квадратной матрицы на вектор
implicit none
integer :: n, i, j, ierror  ! размер массивов, индексы цикла
real(8), allocatable, dimension(:) :: x, y ! объявление векторов
real(8), allocatable, dimension(:,:) :: A  ! объявление матрицы
  write(*,*) 'Введите размер массивов: '; read(*,*) n
  allocate(A(n,n), x(n), y(n))                ! выделение памяти
  call random_number(A); call random_number(x); y = 0.
! Способ с использованием арифметического цикла
  do j = 1, n
    do i = 1, n
      y(i) = y(i) + A(i,j) * x(j)
    end do
  end do
! Способ с использованием встроенной функции
  y = matmul(A, x)
! Печать результирующего вектора в файл
  open(9, file='output.dat', status='replace', iostat=ierror)
  write(9,'(i5)') n                              ! размер вектора
  write(9,'(f12.8)') y
  close(9)
  deallocate(A, x, y)                      ! освобождение памяти
end program matrix_vector_mult
```

Назначенный порядок вложенности в составном арифметическом цикле обусловливает выполнение вычислений в оптимальном порядке, который соответствует расположению элементов матрицы в оперативной памяти (по столбцам, см. п. 7.4). Данные забираются из ОЗУ в быструю кэш-память порциями размещенных подряд элементов, далее обрабатываются в цикле, пока не закончится очередная порция, после чего будет запрошена следующая и т. д. В рассматриваемом варианте кэш-память используется эффективно, все загруженные в нее элементы матрицы обрабатываются подряд, без пропусков, что повышает общую производительность программы.

Проиллюстрировать эту особенность и оценить ее значимость можно посредством окружения цикла вызовами встроенной функции `cpu_time` и замера времени работы вычислительного фрагмента для двух разных порядков вложенности (замеры произвести с разными уровнями оптимизации).

9.9. Транспонирование матрицы

Программа транспонирования квадратной матрицы с тремя способами выполнения операции:

```
program matrix_transpose
! Программа транспонирования квадратной матрицы
implicit none
integer :: n, i, j                ! размер матрицы, индексы цикла
real(8), allocatable, dimension(:,:) :: A, B ! объявление матриц
  write(*,*) 'Введите размер матрицы: '; read(*,*) n
  allocate(A(n,n), B(n,n)); call random_number(A)
! Способ с использованием арифметического цикла
  do j = 1, n
    do i = 1, n
        B(i,j) = A(j,i)
    end do
  end do
! Способ с использованием оператора маскирования присваивания
  forall (j = 1:n, i = 1:n) B(i,j) = A(j,i)
! Способ с использованием встроенной функции
  B = transpose(A)
  deallocate(A, B)                          ! освобождение памяти
end program matrix_transpose
```

9.10. Перемножение матриц

Программа перемножения двух квадратных матриц с инициализацией разными методами, четырьмя способами выполнения операции и выводом результирующей матрицы в файл (рекомендуется добавить замеры времени):

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}, i,j=1,...,n.$$

```
program matrix_mult
! Программа перемножения квадратных матриц
implicit none
character(len=20) :: fmt
integer :: n, i, j, k, ierror     ! размер матриц, индексы цикла
real(8), allocatable, dimension(:,:) :: A, B, C     ! объявление
  write(*,*) 'Введите размер матриц: '; read(*,*) n
  allocate(A(n,n), B(n,n), C(n,n))           ! выделение памяти
```

```
  call random_number(A); call random_number(B); C = 0.
! Вариант с использованием арифметического цикла
  do j = 1, n
    do i = 1, n
      do k = 1, n
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
      end do
    end do
  end do
! Вариант с использованием векторной формы
  C = 0.
  do j = 1, n
    do k = 1, n
      C(1:n,j) = C(1:n,j) + A(1:n,k) * B(k,j)
    end do
  end do
! Вариант с использованием векторной формы и встроенной функции
  do j = 1, n
    do i = 1, n
      C(i,j) = dot_product(A(i,1:n), B(1:n,j))
    end do
  end do
! Вариант с использованием встроенной функции matmul
  C = matmul(A, B)
! Печать результирующей матрицы в файл
  open(9, file='output.dat', status='replace', iostat=ierror)
  write(9,'(i4,i4)') n, n                    ! размер матрицы
  write(fmt, *) n
  do i = 1, n           ! цикл печати матрицы в двумерной форме
    write(9, '('//fmt//'(f12.8))') ( C(i, 1:n) )
  end do
  close(9)
  deallocate(A, B, C)
end program matrix_mult
```

9.11. Решение СЛАУ с нижней треугольной матрицей

Программа решения СЛАУ `Ax = b` с нижней треугольной матрицей с вызовом подпрограммы и использованием сечений массивов:

```
program slae_left_triangle
! Программа решения СЛАУ с нижней треугольной матрицей
! Элементы на главной диагонали и ниже для A и решения x равны 1
```

```
implicit none
integer :: n, i, j                ! размеры массивов, индексы цикла
real(8), allocatable, dimension(:) :: b, x   ! объявление векторов
real(8), allocatable, dimension(:,:) :: A    ! объявление матрицы
  write(*,*) 'Введите размеры массивов: '; read(*,*) n
  allocate(A(n,n), b(n), x(n))                  ! выделение памяти
  A = 0.; b = [ (real(i), i = 1,n) ]               ! инициализация
  forall (i = 1:n, j = 1:n, i >= j) A(i, j) = 1.
  call tr_solve(A, b, x)          ! вызов подпрограммы решения СЛАУ
  deallocate(A, b, x)                          ! освобождение памяти
contains
! Подпрограмма решения СЛАУ
  subroutine tr_solve(A, b, x)
  implicit none
  integer :: i, n
  real(8), dimension(:,:), intent(in)  :: A
  real(8), dimension(:), intent(in)    :: b
  real(8), dimension(:), intent(inout) :: x
  real(8), dimension(:), allocatable   :: bb
  allocate(bb, source = b); n = size(b)                          !
! Основной вычислительный цикл
    do i = 1, n
      x(i) = bb(i) / A(i, i)
      bb(i + 1:n) = bb(i + 1:n) - A(i + 1:n, i) * x(i)
    end do
  end subroutine tr_solve
end program slae_left_triangle
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Основная литература

Бартеньев О. В. Современный Фортран / О. В. Бартеньев. – 4-е изд., доп. и перераб. – М. : Диалог-МИФИ, 2005. – 445 с.

Горелик А. М. Программирование на современном Фортране / А. М. Горелик. – М. : Финансы и статистика, 2006. – 352 с.

Меткалф М. Описание языка программирования Фортран 90 : [пер. с англ.] / М. Меткалф, Дж. Рид. – М. : Мир, 1995. – 302 с.

Немнюгин С. А. Современный Фортран : самоучитель / С. А. Немнюгин, О. Л. Стесик. – СПб. : БХВ-Петербург, 2004. – 496 с.

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

Рыжиков Ю. И. Современный Фортран / Ю. И. Рыжиков. – СПб. : КОРОНА Принт, 2009. – 290 с.

Шелест В. Д. Программирование: Структурный подход. Алгоритмы / В. Д. Шелест. – СПб. : БХВ-Петербург, 2002. – 584 с.

Adams J. C. Fortran 90 Handbook: Complete ANSI/ISO Reference (Computing That Works) / J. C. Adams, W. S. Brainerd, J. T. Martin. – New York : Mcgraw-Hill, 1992. – 740 p.

The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures / J. C. Adams [et al.]. – Luxsembourg : Springer, 2009. – 712 p.

Brainerd W. S. Guide to Fortran 2003 Programming / W. S. Brainerd. – Springer, 2009. – 357 p.

Chapman S. Fortran for Scientists and Engineers / S. Chapman. – 4th Edition. – New York : Mcgraw-Hill, 2018. – 1049 p.

Дополнительная литература

Абрамов А. Г. Исследовательские проекты в области механики сплошных сред. Пакеты прикладных программ в научных исследованиях : учеб. пособие / А. Г. Абрамов, М. А. Засимова, Н. Г. Иванов. – СПб. : ПОЛИТЕХ-ПРЕСС, 2020. – 229 с.

Арьен М. Современный Fortran на практике : [пер. с англ.] / М. Арьен. – М. : ДМК Пресс, 2015. – 318 с.

Бартеньев О. В. Фортран для студентов / О. В. Бартеньев. – М. : Диалог-МИФИ, 1999. – 397 с.

Брауде Э. Дж. Технология разработки программного обеспечения : [пер. с англ.] / Э. Дж. Брауде. – СПб. : Питер, 2004. – 654 с.

Вирт Н. Алгоритмы + структуры данных = программы : [пер. с англ.] / Н. Вирт. – М. : Мир, 1985. – 406 с.

Воеводин В. В. Матрицы и вычисления / В. В. Воеводин, Ю. А. Кузнецов. – М. : Наука, 1984. – 320 с.

Гарбарук А. В. Механика жидкости и газа. Задачи и решения : учеб. пособие / А. В. Гарбарук. – СПб. : АРКУШ, 2003. – 31 с.

Демидович Б. П. Краткий курс высшей математики : учеб. пособие для вузов / Б. П. Демидович, В. А. Кудрявцев. – М. : Астрель, АСТ, 2007. – 654 с.

Житомирский М. С. Начала вычислительной математики. Введение в числительный эксперимент / М. С. Житомирский, В. Д. Шелест. – СПб. : СПбГТУ, 2001. – 200 с.

Звягин В. Ф. Практикум по современному Фортрану в курсе информатики : учеб. пособие / В. Ф. Звягин, Н. А. Янышина, В. Н. Голыничев. – СПб. : ИТМО, 2010. – 134 с.

Информатика. Краткий курс : учеб. пособие / А. А. Иванков [и др.]. – СПб. : Изд-во Политехн. ун-та, 2007. – 63 с.

Лойцянский Л. Г. Механика жидкости и газа / Л. Г. Лойцянский. – 7-е изд., испр. – М. : Дрофа, 2003. – 840 с.

Мирошниченко Е. А. Технологии программирования : учеб. пособие / Е. А. Мирошниченко. – Томск : Изд-во Томск. политехн. ун-та, 2008. – 128 с.

Немнюгин С. Эффективная работа: UNIX : учеб. рук-во / С. Немнюгин, М. Чаунин, А. Комолкин. – СПб. : Питер, 2001. – 682 с.

Орлов С. А. Технологии разработки программного обеспечения: современный курс по программной инженерии : учебник для вузов / С. А. Орлов, Б. Я. Цилькер. – 4-е изд. – СПб. : Питер, 2012. – 608 с.

Самарский А. А. Численные методы : учеб. пособие для вузов / А. А. Самарский, А. В. Гулин. – М. : Наука, 1989. – 430 с.

Таненбаум Э. Архитектура компьютера : [пер. с англ.] / Э. Таненбаум, Т. Остин. – 6-е изд. – СПб. : Питер, 2019. – 816 с.

Bose S. K. Numerical Methods of Mathematics Implemented in Fortran / S. K. Bose. – Luxsembourg : Springer International Publishing, 2019. – 467 p.

Hahn B. D. Fortran 90 for Scientists and Engineers / B. D. Hahn. – London : Arnold, 1997. – 351 p.

Metcalf M. Fortran 90/95 Explained / M. Metcalf, J. Reid. – 5th edition. – Oxford : Oxford University Press, 1999. – 360 p.

Metcalf M. Fortran 95/2003 Explained / M. Metcalf, J. Reid, M. Cohen. – Oxford : Oxford University Press, 2004. – 440 p.

ПРИЛОЖЕНИЯ

Приложение А

НЕКОТОРЫЕ ЗАМЕЧАНИЯ К РАЗРАБОТКЕ ПРОГРАММ

1. Используйте горизонтальные и вертикальные отступы и другие элементы для облегчения восприятия кода.

2. Пишите комментарии, отражающие смысл выполняемых операций, а не дублирующие описание операторов.

3. Не пишите длинные строки кода (делайте переносы), в том числе и строки с несколькими операторами.

4. Всегда используйте оператор **`implicit none`**, явно описывая все объекты данных и их типы.

5. Давайте переменным, структурам и процедурам осмысленные имена. Избегайте близких по написанию имен. Математическим, физическим и пр. константам давайте содержательные имена, значения которых можно понять с первого взгляда.

6. Создавайте словарь данных, явно объявляющий и описывающий все переменные в программе, обязательно указывайте физические единицы измерения переменных (при наличии).

7. Всегда выводите физические единицы, связанные с любыми переменными и константами, – это важно для верной интерпретации результатов работы программы.

8. Давайте имена вложенным конструкциям из операторов **`if`**, **`do`** и пр.

9. Не используйте оператор безусловного перехода **`go to`**.

10. В сложных выражениях используйте дополнительные круглые скобки, явно определяя приоритет операций и улучшая читаемость кода.

11. Для наглядности ставьте пробелы в формулах и выражениях.

12. Инициализируйте переменные в программе перед их использованием с помощью операторов присваивания, оператора ввода **`read`** или при объявлении типа.

13. Используйте постоянное количество значащих цифр в константах в разных частях программы.

14. Указывайте константы с такой точностью, которую поддерживает компьютер.

15. Избегайте записи, которая может приводить к потерям точности.

16. Учитывайте (при необходимости) трудоемкость базовых операций (присваивание, сложение/вычитание, умножение/деление, возведение в степень) и то, что для целых чисел они выполняются существенно быстрее, чем для вещественных.

17. Проверяйте результат вычислений сначала без оптимизации на уровне компилятора, а затем с оптимизацией, поскольку она может повлиять на точность результата.

18. Ставьте «эхо-печать» входных и выходных данных на экран или в файл для контроля. Предпочтительнее вводить данные из файла, если их объем существенен.

19. Выводите на экран приглашение к вводу параметров, объясняющее, какие данные и в каком порядке требуется ввести.

20. Продумывайте и программируйте разветвления с особой тщательностью. Все возможности должны быть учтены, лишние проверки и взаимоисключающие или повторные условия удалены.

21. Арифметические циклы заменяйте с использованием средств упрощения кода в Фортране (если это возможно), при невозможности это сделать следует особо тщательно решать проблему быстродействия, учитывая в том числе особенности хранения многомерных массивов.

22. Избегайте использования чисел в качестве границ циклов; правильнее задействовать переменные или именованные константы.

23. Исключайте зацикливание итеративных циклов, являющееся следствием ошибки кодирования или, например, отсутствия сходимости итерационного процесса к точному решению задачи.

24. Выделяйте в программе подчиненные алгоритмы.

25. Реализуйте (по возможности) короткие и максимально легкочитаемые процедуры.

26. Следите за соответствием формальных и фактических параметров процедур и их свойств.

27. Опечадтка (опечатка) – это тоже ошибка (будьте внимательны, поскольку их бывает сложно заметить).

Приложение Б

ПОЛЕЗНЫЕ КЛЮЧИ КОМПИЛЯТОРА GFORTRAN

В табл. Б.1 приведены наименование и краткое описание функционала некоторых полезных и востребованных на практике ключей (опций) компилятора gfortran (полная информация об опциях доступна на сайте *https://gcc.gnu.org/onlinedocs/gfortran/Invoking-GNU-Fortran.html*).

Таблица Б.1

Наименование и краткое описание функционала некоторых полезных и востребованных на практике ключей

Ключ	Описание
`-fcheck`	Включение проверок времени выполнения (указывается с аргументом)
`-ffixed-form`	Указание, что текст программы использует фиксированную форму записи кода
`-ffixed-line-length-none`	Снятие ограничения на количество символов в строке
`-ffree-form`	Указание, что текст программы использует свободную форму записи кода (для версий Фортран 90 и выше)
`-fimplicit-none`	Запрет неявной типизации (эквивалентно указанию в программе оператора **`implicit none`**)
`-fsyntax-only`	Проверка кода на наличие синтаксических ошибок без выполнения компиляции
`--help`	Получение краткой справки о доступных ключах компилятора
`-g`	Создание отладочной информации для работы с отладчиком
`-I`	Добавление каталога в список поиска заголовочных файлов в процессе сборки
`-L`	Добавление каталога в список поиска библиотек
`-o <file>`	Сохранить результат работы в файл с заданным именем
`-O`	Указание опции оптимизации (`-O0/-O1/-O2/-O3` и др.)
`-std`	Указание стандарта языка, которому должна соответствовать программа (`f95, f2003, f2008, f2018, gnu, legacy`)
`-v`	Вывод информации о программах, вызванных компилятором
`--version`	Вывод версии компилятора
`-Wall`	Вывод сообщений обо всех предупреждениях или ошибках, возникающих во время компиляции программы
`-Wextra`	Включение вывода предупреждений об использовании языковых функций, которые могут быть проблематичными

Таблица В.1

Встроенные операции языка Фортран

Знак оператора	Тип	Бинарная/ унарная	Операция	Поддерживаемые типы
`=`	Присваивание	Бинарная	Присваивание	`integer, real, complex, logical, character`
`+`	Арифметическая	Бинарная	Сложение	`integer, real, complex`
`-`	Арифметическая	Бинарная	Вычитание	`integer, real, complex`
`+x`	Арифметическая	Унарная	Унарный плюс	`integer, real, complex`
`-x`	Арифметическая	Унарная	Унарный минус (смена знака)	`integer, real, complex`
`*`	Арифметическая	Бинарная	Умножение	`integer, real, complex`
`/`	Арифметическая	Бинарная	Деление	`integer, real, complex`
`**`	Арифметическая	Бинарная	Возведение в степень	`integer, real, complex`
`==`	Сравнение	Бинарная	Равно	`integer, real, complex, character`
`/=`	Сравнение	Бинарная	Не равно	`integer, real, complex, character`
`>`	Сравнение	Бинарная	Больше	`integer, real, complex, character`
`>=`	Сравнение	Бинарная	Больше или равно	`integer, real, complex, character`
`<`	Сравнение	Бинарная	Меньше	`integer, real, complex, character`

Окончание табл. В.1

Знак оператора	Тип	Бинарная/ унарная	Операция	Поддерживаемые типы
<=	Сравнение	Бинарная	Меньше или равно	**integer, real, complex, character**
.eqv.	Логическая	Бинарная	Эквивалентны	**logical**
.neqv.	Логическая	Бинарная	Неэквивалентны	**logical**
.and.	Логическая	Бинарная	Логическое И	**logical**
.or.	Логическая	Бинарная	Логическое ИЛИ	**logical**
.not.	Логическая	Унарная	Логическое НЕ	**logical**
//	Символьная	Бинарная	Конкатенация	**character**

Приложение Г

Таблица Г.1

Некоторые встроенные функции языка Фортран

Функция	Выполняемая операция
Числовые функции	
abs(a)	Абсолютное значение (модуль) аргумента
aimag(z)	Мнимая часть заданного комплексного числа
aint(a [, kind])	Усечение аргумента до целого числа
anint(a [, kind])	Округление аргумента до ближайшего целого числа
ceiling(x)	Округление аргумента в большую сторону
dim(x, y)	Возвращает разницу x-y, если результат положительный, в противном случае – 0
floor(x)	Округление аргумента в меньшую сторону
int(a), int2(a), int8(a)	Преобразование аргумента к типу **integer** с соответствующим параметром разновидности
max(a1, a2, …), min(a1, a2, …)	Максимальное (минимальное) значение для списка аргументов
mod(a, p)	Остаток от деления a на p
nint(a)	Преобразование аргумента к типу **integer** (усечение до ближайшего целого)

Продолжение табл. Г.1

Функция	Выполняемая операция
real(a [, kind])	Преобразование аргумента к типу **real** с соответствующим параметром разновидности
sign(a, b)	Возвращает значение a со знаком b
Математические функции	
asin(x), acos(x), atan(x)	Обратные тригонометрические функции (аргумент задан в радианах)
asind(x), acosd(x), atand(x)	Обратные тригонометрические функции (аргумент задан в градусах)
asinh(x), acosh(x), atanh(x)	Обратные гиперболические тригонометрические функции
exp(x)	Экспоненциальная функция
log(x)	Натуральный логарифм
log10(x)	Десятичный логарифм
sin(x), cos(x), tan(x), cotan(x)	Тригонометрические функции (аргумент задан в радианах)
sind(x), cosd(x), tand(x)	Тригонометрические функции (аргумент задан в градусах)
sinh(x), cosh(x), tanh(x)	Гиперболические тригонометрические функции
sqrt(x)	Функция квадратного корня
Логические функции	
iand, and	Побитовое логическое И
ieor, xor	Побитовое логическое исключающее ИЛИ
ior, or	Побитовое логическое ИЛИ
logical(l, [,kind])	Преобразование в логический тип данных с заданным параметром разновидности
not	Логическое отрицание
Функции времени	
cpu_time(t)	Затраченное процессором время в секундах
ctime(t, r)	Преобразует заданное время t в строковый вид r
date_and_time([d] [, t][, z][, v])	Информация о текущих дате и времени (выдается поэлементная информация об аргументах функции)
itime(t,r)	Информация о текущих дате и времени (запись результата r в виде строки)

Продолжение табл. Г.1

Функция	Выполняемая операция
system_clock ([c] [, cr] [, cm])	Текущее системное время; c – текущее время; cr – число отсчетов в секунду; cm – максимальное значение, которое может принимать c
time(),time8()	Текущее системное время (вывод целого числа)
Функции случайных чисел	
irand(flag)	Целое псевдослучайное число
rand(flag)	Вещественное псевдослучайное число
random_init(r, i_d)	Инициализация генератора случайных чисел
random_number(r)	Генератор псевдослучайных чисел или массивов, значения которых находятся в диапазоне [0…1]
random_seed ([s] [, p] [, g])	Перезапускает/запрашивает состояние генератора псевдослучайных чисел, используемого random_number
Функции работы с массивами	
all(m [, dim])	Возвращает .true., если все элементы логического массива m вдоль заданного (необязательного) измерения dim истинны; в противном случае .false.
allocated(a)	Возвращает значение .true., если размещаемый скаляр или массив a (имеющий атрибут allocatable) в данный момент размещен, .false. – в противном случае
any(m [, dim])	Возвращает .true., если хотя бы один элемент логического массива m вдоль заданного (необязательного) измерения dim истинен; в противном случае – .false.
count(m [, dim] [, kind])	Возвращает число элементов логического массива m, имеющих значение .true., вдоль заданного (необязательного) измерения dim
dot_product(a, b)	Вычисление скалярного произведения векторов a и b
matmul(a, b)	Умножение матриц a и b по правилам линейной алгебры
maxloc(a [, dim] [, m] [, kind]) minloc(a [, dim] [, m] [, kind])	Размер массива-результата равен рангу массива a, значения элементов равны индексам максимального (минимального) элемента массива a по заданному измерению dim, удовлетворяют заданным условиям m

Продолжение табл. Г.1

Функция	Выполняемая операция
`maxval(a [,d] [,m])` `minval(a [,d] [,m])`	Вычисление максимального (минимального) значения массива `a`
`product (a [, dim] [, m])`	Вычисление произведения всех элементов массива `a`, удовлетворяющих критерию отбора `m`, вдоль измерения `d` (необязательны)
`reshape(s, sh [, p] [, o])`	Меняет форму массива при объявлении типов или в коде программы, сохраняя количество элементов
`shape(a [, kind])`	Возвращает одномерный массив целого типа, равный рангу `a` и содержащий форму массива или скаляра `a`
`size(a [, dim])`	Возвращает целое число, равное размеру массива `a`, или при наличии второго параметра – число элементов вдоль заданного измерения `dim`
`sum(a [, dim] [, m])`	Вычисление суммы всех элементов массива `a`, удовлетворяющих необязательному критерию отбора `m`, вдоль необязательного измерения `dim`
`transpose(a)`	Транспонирование матрицы `a`
Функции работы с символьными строками	
`achar(i [, kind])`	Возвращает символ типа **`character`**(1), код которого в таблице ASCII-кодов символов равен `i`
`adjustl(s)`	Выполняет левое выравнивание символьной строки – удаляет все ведущие пробелы и вставляет их в конец
`adjustr(s)`	Выравнивает строку по правой границе за счет удаления всех хвостовых пробелов и их последующей вставки в начало строки
`char(i [, kind])`	Возвращает символ, представленный целым числом `i`
`iachar(c [, kind])`	Возвращает значение целого типа, равное ASCII-коду символа `c`
`ichar(c [, kind])`	Возвращает значение целого типа, равное коду символа из поддерживаемой нативной таблицы символов
`index(s, subs [, back] [, kind])`	Возвращает номер позиции, с которой начинается первое вхождение строки `subs` в строку `s`; `back` задает направление поиска
`len(s)`	Возвращает длину строки

Окончание табл. Г.1

Функция	Выполняемая операция
`len_trim(s)`	Возвращает длину строки без хвостовых пробелов
`repeat(s, ncopies)`	Повторяет строку `s` заданное число раз `ncopies`
`scan(s, set [, back [, kind]])`	Сканирует строку `s` на наличие любого из символов в наборе `set`, `back` задает направление поиска
`trim(s)`	Удаляет хвостовые пробелы из строки
Функции работы с файлами и каталогами	
`access(name, mode)`	Проверка режима доступа к файлу с именем `name`
`chdir(name [, status])`	Смена рабочего каталога на заданный путь `name`, в аргументе `status` возвращается код ошибки
`chmod(name, mode [, status])`	Смена прав доступа к файлу с именем `name` на указанные в аргументе `mode`
`fnum(u)`	Возвращает номер файлового дескриптора
`rename(path1, path2 [, status])`	Переименование файла из имени, указанного в пути `path1`, в имя, указанное в пути `path2`
`unlink(path [, status])`	Удаление файла, указанного в пути `path`, из файловой системы

Более подробная информация о встроенных функциях языка Фортран, их аргументах и возвращаемых значениях доступна на сайте *https://gcc.gnu.org/onlinedocs/gfortran/Intrinsic-Procedures.html*.

Приложение Д

ОФОРМЛЕНИЕ ОТЧЕТОВ ПО ЛАБОРАТОРНЫМ РАБОТАМ

По результатам выполнения лабораторных работ студентам следует подготовить индивидуальный отчет, структура которого должна соответствовать приведенным ниже требованиям.

Первая страница отчета должна представлять собой титульный лист, оформленный в соответствии с принятыми требованиями и включать:

– наименование университета, института, учебного подразделения;
– наименование учебной дисциплины;
– название лабораторной работы;
– ФИО, должность, ученая степень и ученое звание преподавателя;

– ФИО и номер группы студента, направление подготовки, включая шифр;

– город и текущий год.

Рекомендуемый набор разделов отчета:

– титульный лист;

– текст варианта индивидуального задания;

– описание реализованного алгоритма, включая математические формулы, физические величины, единицы измерений и др.;

– графические блок-схемы алгоритмов;

– информация об использованных инструментальных средствах разработки программ (компиляторы, отладчики, IDE и др.), примеры (скриншоты) компиляции и запуска программ;

– исходные коды разработанных программ с содержательными комментариями;

– результаты проверки правильности выполнения программ на тестовых объектах малой размерности (короткие циклы, векторы, матрицы, сравнение с эталонными решениями при их наличии);

– результаты работы программ на реальных объектах (вычислительные эксперименты, время работы для разных реализаций и условий, включая размеры операндов, точность вычислений, опции оптимизации);

– общие выводы по работе.

Отчет может быть предоставлен для проверки и обсуждения как в бумажном, так и в электронном виде (последний вариант предпочтителен).

Абрамов Алексей Геннадьевич
Засимова Марина Александровна
Фролов Максим Евгеньевич

ЦИФРОВЫЕ ТЕХНОЛОГИИ В ПРОФЕССИОНАЛЬНОЙ ДЕЯТЕЛЬНОСТИ

ЯЗЫК ПРОГРАММИРОВАНИЯ ФОРТРАН ДЛЯ НАУЧНЫХ И ИНЖЕНЕРНЫХ ВЫЧИСЛЕНИЙ

Ч а с т ь 1

Учебное пособие

Редактор *Л. В. Ларионова*
Корректор *Н. Б. Цветкова*
Компьютерная верстка *Е. Н. Никулкиной*
Дизайн обложки *Е. В. Гладышевой*

Санитарно-эпидемиологическое заключение
№ 78.01.07.953.П.001342.01.07 от 24.01.2007 г.

Налоговая льгота – Общероссийский классификатор продукции
ОК 005-93, т. 2; 95 3005 – учебная литература

Подписано в печать 14.06.2024. Формат 60×84/16. Печать цифровая.
Усл. печ. л. 9,75. Тираж 50. Заказ 3954.

Отпечатано в Издательско-полиграфическом центре
Политехнического университета.
195251, Санкт-Петербург, Политехническая ул., 29.
Тел.: (812) 552-77-17; 550-40-14.