ВЫСШЕЕ ОБРАЗОВАНИЕ

И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

УМО ВО
РЕКОМЕНДУЕТ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ

Юрайт
издательство
biblio-online.ru

И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

УЧЕБНОЕ ПОСОБИЕ ДЛЯ ВУЗОВ

Рекомендовано Учебно-методическим отделом высшего образования в качестве учебного пособия для студентов высших учебных заведений, обучающихся по инженерно-техническим направлениям

Книга доступна на образовательной платформе «Юрайт» urait.ru, а также в мобильном приложении «Юрайт.Библиотека»

Москва • Юрайт • 2022

УДК 004.42(075.8)
ББК 32.973-018я73
Г56

Авторы:

Гниденко Ирина Геннадиевна — кандидат экономических наук, доцент кафедры вычислительных систем и программирования факультета информатики и прикладной математики Санкт-Петербургского государственного экономического университета;

Павлов Федор Федорович — профессор, кандидат технических наук, профессор кафедры вычислительных систем и программирования факультета информатики и прикладной математики Санкт-Петербургского государственного экономического университета;

Федоров Дмитрий Юрьевич — старший преподаватель кафедры вычислительных систем и программирования факультета информатики и прикладной математики Санкт-Петербургского государственного экономического университета.

Рецензенты:

Чудаков О. Е. — доктор технических наук, профессор кафедры специальных информационных технологий Санкт-Петербургского университета МВД России;

Трофимов В. В. — доктор технических наук, профессор, заведующий кафедрой информатики Санкт-Петербургского государственного экономического университета, заслуженный деятель науки Российской Федерации.

Гниденко, И. Г.

Г56 Технологии и методы программирования : учебное пособие для вузов / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — Москва : Издательство Юрайт, 2022. — 235 с. — (Высшее образование). — Текст : непосредственный.

ISBN 978-5-534-02816-4

В учебном пособии рассматриваются теоретические основы современных технологий и методов программирования и практические вопросы создания программ на языках высокого уровня, описываются эволюция языков программирования, жизненный цикл и организация разработки программного обеспечения, основы программирования на языках C и Python.

Содержание пособия соответствует актуальным требованиям федерального государственного образовательного стандарта высшего образования.

Для студентов высших учебных заведений, обучающихся по инженерно-техническим направлениям.

УДК 004.42(075.8)
ББК 32.973-018я73

ISBN 978-5-534-02816-4

Оглавление

Авторский коллектив

Гниденко Ирина Геннадиевна — кандидат экономических наук, доцент кафедры вычислительных систем и программирования факультета информатики и прикладной математики Института экономики СПбГЭУ (г. Санкт-Петербург) — гл. 1—3 (в соавторстве с Ф. Ф. Павловым); 5, 6;

Павлов Федор Федорович — профессор, кандидат технических наук, профессор кафедры вычислительных систем и программирования факультета информатики и прикладной математики Института экономики СПбГЭУ (г. Санкт-Петербург) — гл. 1—3 (в соавторстве с И. Г. Гниденко);

Федоров Дмитрий Юрьевич — старший преподаватель кафедры вычислительных систем и программирования факультета информатики и прикладной математики Института экономики СПбГЭУ (г. Санкт-Петербург) — гл. 4, 7.

Предисловие

Современное общество является информационным, в нем все большее число людей занято получением, переработкой и использованием информации с применением компьютерных технологий. Компьютеры используются практически во всех областях человеческой деятельности: в профессионльной, учебной, досуговой сферах.

Дальнейшее развитие информационного общества требует разработки большого количества качественных программных продуктов, обеспечивающих удовлетворение растущих потребностей людей.

В этих условиях весьма актуальным становится овладение современными технологиями программирования. Исследования в области управления персоналом показывают, что профессия программиста будет одной из самых востребованных в XXI в.

Целью и задачами учебного пособия «Технологии и методы программирования» является систематизация и упорядочение сведений о технологиях разработки современных программных продуктов. Рассмотрение теоретических основ программирования сопровождается большим количеством примеров, иллюстрирующих приемы создания программ, а также заданиями для самостоятельного выполнения, позволяющими сформировать у обучающихся практические навыки программирования.

Учебное пособие состоит из двух частей.

В первой части, «Методы и этапы технологии программирования», изучаются этапы эволюции технологии программирования, основные этапы и модели жизненного цикла программы, постановка задачи и определение спецификаций, проектирование и реализация программы, анализируются типы пользовательских интерфейсов.

Во второй части, «Основы программирования на языке высокого уровня», рассматриваются основные алгоритмические конструкции и их реализация на языках высокого уровня Python и C. Рассматривается пример проектирования и разработки программного приложения на языке C, а также возможности интеграции языков C и Python.

Приведенные примеры на языке C реализованы в среде разработки Microsoft Visual Studio 2013 (дополнительное тестирование проводилось в компиляторе gcc version 4.9.2 Debian 4.9.2-10). Примеры на языке Python выполнялись в интерпретаторе Python 3.6.

После каждой главы приводится перечень контрольных вопросов, позволяющих оценить степень овладения учебным материалом. Во второй части, кроме контрольных вопросов, приводится список заданий по программированию для самостоятельного выполнения.

Пособие предназначено для студентов, обучающихся по программам бакалавриата и магистратуры, а также для всех, интересующихся современными технологиями программирования.

В результате изучения материалов пособия обучающиеся должны:

знать

- методы и этапы технологии программирования;
- особенности структурного программирования;
- особенности объектно-ориентированного программирования;

уметь

- выбирать необходимые методы программирования для решения поставленных задач;
- разрабатывать программы методом структурного программирования;
- разрабатывать программы методом объектно-ориентированного программирования;

владеть

- навыками разработки алгоритмов решения задач;
- навыками документирования программ;
- современными технологиями разработки программных приложений.

Часть 1

МЕТОДЫ И ЭТАПЫ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

В ч. 1 изучаются эволюция технологии программирования, жизненный цикл и этапы технологии программирования, среды разработки программ, пользовательский интерфейс.

В результате изучения раздела обучающиеся должны:

знать

• современные представления о методах и технологиях программирования;

• стандарты в области разработки и реализации программного обеспечения;

уметь

• грамотно ориентироваться в существующих технологиях программирования;

• применять теоретические знания в области жизненного цикла к организации и разработке программного обеспечения;

владеть

• навыками оценки качества разработанных алгоритмов и программ;

• навыками выбора основных компонентов интерфейса для реализации диалога с пользователем.

Глава 1
ЭВОЛЮЦИЯ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

В результате изучения гл. 1 обучающиеся должны:

знать

• историю, логику и тенденции развития технологий программирования;

• современные представления о методах и технологиях программирования;

уметь

• идентифицировать и классифицировать методы программирования;

• грамотно ориентироваться в существующих технологиях программирования;

владеть

• навыками работы с учебной и научной литературой о методах и технологиях программирования;

• спецификой применения различных методов программирования для решения конкретных задач.

1.1. Неструктурированное программирование

- **Технологией программирования** называется совокупность методов и средств, используемых в процессе разработки программного обеспечения (ПО).

Как любая технология, технология программирования представляет собой последовательность технологических операций (этапов программирования) с описанием операций (исходные данные и результаты) и условиями их выполнения.

До середины 1960-х гг. преимущественно использовалась *неструктурированная*, «стихийная» технология программирования. Структура первых простейших программ состояла собственно из программы, написанной на машинном языке (в двоичных или шестнадцатеричных кодах) и обрабатываемых ею данных.

Появление *машинно-ориентированных языков* (ассемблеров) позволило программистам вместо кодов использовать мнемонические обозначения кодов операций и символические имена данных. Программы стали «читаемыми».

Появление *языков программирования высокого уровня* (FORTRAN, ALGOL) позволило снизить уровень детализации операций. Большим достижением этих языков стала возможность использования подпрограмм. Были созданы большие библиотеки различных подпрограмм. Теперь структура программы состояла из основной программы, области глобальных данных и набора подпрограмм. Недостаток такой структуры — возрастание вероятности искажения части глобальных данных какой-либо подпрограммой при увеличении количества подпрограмм. Для сокращения таких ошибок было предложено размещать в подпрограммах локальные данные. Появилась возможность совместной разработки ПО несколькими программистами.

В 1960-х гг. при разработке сложного ПО (например, операционных систем) стали срываться сроки разработки программ. Разразился «кризис программирования». Причина — несовершенство неструктурированного, «стихийного» программирования. Использовался метод программирования «снизу — вверх», т.е. сначала разрабатывались простые подпрограммы, а затем строилась сложная программа путем их сборки. При сборке программы появлялось большое количество ошибок согласования, а при их исправлении появлялись новые ошибки. Процесс тестирования и отладки занимал 80% времени разработки ПО.

Стоимость аппаратных средств снижалась, а стоимость разработки ПО все время росла из-за того, что создавались все более мощные и сложные прикладные программы при отставании технологии их разработки.

Было разработано много языков, но лишь некоторые из них получили тогда широкое применение (FORTRAN, ALGOL, COBOL). Продолжающийся рост стоимости больших программных продуктов в 1960-х гг. и их ненадежность привели к большим исследовательским работам в области создания технологий программирования.

1.2. Процедурное и модульное программирование

В результате исследовательских работ 1960—1970-х гг. была разработана технология *процедурного* (или *структурного*, *модульного*) программирования, внесшая ясность в написание программ, простоту тестирования и отладки, легкость модификации. По срав-

нению со стихийным программированием технология процедурного программирования — это дисциплинированный подход к написанию программ. Процедурное программирование основано на модели построения программы как иерархии процедур, что и дало название данной технологии.

Для изучения процедурного программирования в 1971 г. Н. Виртом был создан язык программирования Pascal, нашедший большое применение в университетах. На протяжении 1970-х гг. создавался язык С на базе концепции предшествующих двух языков — BCPL и B, разработанных для написания компиляторов и операционных систем. Язык С получил широкую популярность в результате его использования в разработке операционной системы UNIX. В конце 1970-х гг. был создан «классический» язык С Б. Кернигана и Д. Ритчи. На этом языке были написаны фактически все новые операционные системы и системные программные продукты.

Технология процедурного программирования основана на использовании следующих методов (приемов) программирования:

1) метод *декомпозиции* (нисходящего проектирования), т.е. разделение программы на процедуры простейшей структуры и представление программы в виде иерархии процедур;

2) метод *модульной организации*, т.е. группировка процедур и обрабатываемых ими данных в модули, которые программируются и компилируются отдельно. Преимущества данного метода заключаются в параллельной работе программистов, удобстве программирования, возможности создания библиотек;

3) метод *структурного программирования процедур*, который заключается в следующем:

• разделение процедур на вложенные блоки, что позволяет локализовать переменные и операторы их обработки, структурировать процедуру;

• использование операторов ветвления и циклов, осуществляющих передачу управления только сверху — вниз, что приводит к ясности алгоритма, к облегчению программирования и сопровождения программ; операторы безусловной передачи управления goto использовать не рекомендуется;

• форматирование текста процедуры, т.е. использование отступов для отображения вложенности блоков, применение идентификаторов, несущих смысл, и использование комментариев, что приводит к повышению читаемости программ и к облегчению их сопровождения.

Наиболее известными процедурными языками программирования являются PL1, ALGOL-68, Pascal, C, C++.

1.3. Объектно-ориентированное программирование

Применение информационных систем в экономике и управлении привело к появлению больших по объему программ. Область применения программного обеспечения постоянно расширялась, процессы управления, подлежащие автоматизации, усложнялись. Структурный подход к программированию не позволял адекватно моделировать сложную предметную область. Проблема сложности программного обеспечения решалась путем дробления программы на отдельные процедуры и уменьшения их размера для удобства работы и повышения читабельности программы. При таком подходе трудно описать реальные объекты предметной области во всем их многообразии, их поведение и взаимосвязи между ними.

Все вышесказанное привело к появлению нового, **объектно-ориентированного**, подхода к программированию (ООП).

В начале 1980-х гг. Б. Страуструп разработал язык C++, ставший первым промышленно используемым языком, использующим объектно-ориентированный подхода к программированию. Язык C++ был построен на базе двух языков — C и Simula 67, языка программного моделирования, разработанного в Европе. К этому моменту имелись и другие объектно-ориентированные языки, наиболее известным из которых был Smalltalk, являющийся чистым объектно-ориентированным языком. Однако ни один из них не нашел такого широкого применения, как C++.

В 1991 г. нидерландским программистом Гвидо ван Россумом был разработан язык программирования Python, включающий в себя как процедурные, так и объектно-ориентированные возможности.

В 1995 г. фирмой Sun Microsystems был разработан на основе языков C и C++ новый язык Java, используемый для создания интерактивных Web-страниц и в разработке приложений на базе Internet и Intranet, а также для реализации ПО устройств, взаимодействующих по сети. Объектный подход к программированию использован в новых версиях языков программирования Pascal, C++, Modula, Java.

Ключевым понятием ООП является понятие объекта.

Объекты являются программными компонентами, моделирующими элементы реального мира. Каждый объект характеризуется своим состоянием и поведением. Состояние объекта определяется совокупностью его свойств (атрибутов) и их текущими значениями. Поведение определяет взаимодействие объекта с другими объектами: то, как он воздействует на другие объекты, и как другие объекты воздействуют на него. Поведение объекта обычно приво-

дит к изменению состояния его и других объектов: изменяются значения их свойств. Действия, которые могут выполняться объектом, называются методами.

Однотипные объекты объединяются в классы. **Класс** — это совокупность объетов, имеющих одинаковые свойства и методы.

Основыми принципами ООП являются:

Абстракция данных позволяет выделить существенные признаки объекта, отличающие его от других объектов. Абстракция должна охватывать поведение объекта, отделяя существенные особенности поведения с точки зрения решаемой задачи от несущественных.

Инкапсуляция означает, что каждый объект полностью описывается совокупностью своих свойств и методов. Инкапсуляция позволяет скрыть внутреннюю организацию объекта, не влияющую на его внешнее поведение. Таким образом, инкапсуляция позволяет в максимальной степени изолировать объект от внешнего окружения. Основной единицей инкапсуляции в ООП является класс. Класс описывает данные, определяющие состояние объекта, и функции, определяющие поведение объекта. Инкапсуляция обеспечивает сокрытие элементов-данных и элементов-функций (в классе с управлением доступом к ним).

Инкапсуляция упрощает создание и сопровождение больших программ, так как инкапсулированные в объекте функции обмениваются с программой сравнительно небольшими объемами данных. В результате замена или модификация данных и методов, инкапсулированных в объект, как правило, не влечет за собой существенной модификации всей программы в целом. Кроме того, инкапсуляция дает возможность определения правил доступа к элементам объекта класса. Атрибуты доступа *private*, *protected*, *public* объявляют элементы класса соответственно закрытыми, защищенными и открытыми для доступа.

Наследование позволяет создавать новые классы на основе существующих. При этом производный класс наследует данные и функции базового класса. Кроме того, производный класс может добавлять новые данные, а также дополнять или модифицировать функции базового класса. Принцип наследования придает ООП значительную гибкость. При работе с объектами обычно подбирается объект, наиболее близкий по своим свойствам для решения конкретной задачи, и на его основе создаются потомки, обладающие дополнительными свойствами и методами.

Полиморфизм означает способность объектов (экземпляров) классов, связанных наследованием, реагировать различным образом на одно и то же сообщение (вызов функции класса).

Различают несколько видов полиморфизма:

• общий полиморфизм (перегрузка операций, преобразование типов, перегрузка функций);

• чистый полиморфизм (виртуальные функции, абстрактные классы);

• параметрический полиморфизм.

Перегрузка операций — это переопределение действий операций применительно к объектам конкретных классов. Преобразование типов данных при совместном их использовании тоже является одним из методов полиморфизма.

Перегрузка функции — это использование одинакового имени для функций, выполняющих похожие действия, но с разными типами данных, и объявленных в одной области действия.

Виртуальная функция — это элемент-функция базового класса в иерархии наследования, переопределенная в производных классах и вызываемая в зависимости от класса объекта, адресуемого через указатель или ссылку на базовый класс.

Параметрический полиморфизм — это механизм использования обобщенного определения функции или класса (шаблона) для автоматической генерации новых функций или классов для различных типов данных.

Преимущества полиморфизма проявляются в следующем:

• облегчение программирования сложных систем за счет возможности называть похожие (различающиеся только типами своих параметров) действия одним именем;

• обеспечение виртуальными функциями чистого полиморфизма, т.е. возможности использовать один и тот же оператор для вызова множества функций. При этом конкретная функция определяется по типу вызываемого объекта;

• механизм виртуальных функций — это возможность написания простых функций общего назначения для иерархии классов;

• обеспечение компактности программ и расширяемости иерархии классов за счет использования виртуальных функций;

• автоматическая генерация по обобщенному шаблону новых функций или классов для различных типов данных, реализуемая механизмом параметрического полиморфизма.

Итак, объектно-ориентированный подход к программированию имеет ряд существенных преимуществ перед структурным подходом:

• возможность более адекватного моделирования предметной области и, соответственно, программирования в понятиях, максимально приближенных к предметной области;

• многократное использование написанного кода;

• сокращение времени разработки и отладки программ.

Однако следует отметить и ряд недостатков объектно-ориентированного подхода:

Значительная глубина абстракции может привести к созданию «многослойных» приложений, где выполнение объектом требуемого действия сводится к множеству обращений к объектам более низкого уровня, что сказывается на производительности системы.

Инкапсуляция снижает скорость доступа к данным. Запрет на прямой доступ к полям класса извне приводит к необходимости создания и использования методов доступа, что ведет к дополнительным расходам.

Код, относящийся к классам-потомкам может находиться не только в этих классах, но и в их классах-предках. Это приводит к снижению скорости трансляции и выполнения программы.

Обеспечение полиморфизма приводит к необходимости связывать методы, вызываемые программой не на этапе компиляции, а в процессе исполнения программы, на что тратится дополнительное время.

1.4. Декларативное программирование

Рассмотренные выше подходы к программированию реализуются в рамках так называемого императивного стиля программирования. Этот стиль предполагает, что программа представляет собой последовательность команд (инструкций), выполняя которые компьютер решает поставленную задачу. Другими словами, при этом подходе программа должна объяснить компьютеру, как решать задачу. Поэтому при императивном программировании сначала создается модель решения, которая затем переводится на императивный язык, причем сам процесс перевода, вообще говоря, не имеет отношения к решению исходной задачи. Соответственно, процесс решения задачи часто разбивается на два этапа, реализуемых разными специалистами:

- постановка — разработка модели решения задачи;
- кодирование — запись модели на императивном языке программирования.

Помимо императивного, широкое распространение получил **декларативный** стиль программирования. В рамках данного стиля программа представляет собой совокупность утверждений, описывающих фрагмент предметной области или сложившуюся ситуацию. Таким образом, описывается результат, а не методы его достижения.

Программируя в декларативном стиле, программист должен описать, *что* нужно решать. В основе декларативных языков лежит формализованная человеческая логика. Человек описывает решае-

мую задачу, а поиском решения занимается система программирования.

Декларативное программирование может применяться во многих областях человеческой деятельности, к которым относятся:

- создание систем искусственного интеллекта;
- автоматическое доказательство теорем;
- разработка экспертных систем и оболочек экспертных систем;
- создание систем поддержки принятия решений;
- разработка систем обработки естественного языка;
- построение планов действий роботов.

Применение декларативного стиля в этих областях позволяет достичь значительно большей скорости разработки приложений, уменьшить размер исходного кода, создать более понятные, по сравнению с императивными языками, программы. Этот подход существенно проще и прозрачнее формализуется математическими средствами, поэтому программы легче тестировать и верифицировать.

К декларативному стилю в первую очередь относятся функциональный и логический подходы к программированию.

Функциональное программирование сформировалось в результате математической направленности при исследовании в области искусственного интеллекта и освоении новых направлений в информатике. Единственной управляющей конструкцией является вызов функции. В функциональном языке существует некоторое множество базовых функций, на основе которых строятся все другие функции, как композиции базовых. В настоящее время существует множество функциональных языков программирования, ориентированных на разные классы задач и виды технических средств, например, Lisp, Haskell и др.

Логическое программирование возникло как упрощение функционального программирования для математиков и лингвистов, решающих задачи символьной обработки.

Задача описывается как совокупность объектов, за основу описания берутся отношения между объектами. Логическая программа представляет собой набор отношений, которые называются фактами, и правил, на основании которых могут быть получены новые отношения.

Программа не задает никакого процесса вычислений, а представляет собой своего рода базу данных о предметной области задачи. Ее применение инициализируется запросом.

Поиск ответа на запрос заключается в попытке логического вывода запроса на основании фактов и правил, имеющихся в базе. К логичесим языкам относятся Prolog, с множеством диалектов, Datalog, Mercury, и др.

1.5. Компонентные технологии

Опыт использования технологии объектно-ориентированного программирования выявил проблемы совместимости при компоновке объектов компиляции, создаваемых разными компиляторами языка. Отсутствовали стандарты компоновки двоичных результатов компиляции объектов в единое целое даже в пределах одного языка программирования с разными компиляторами. Попытка изменения одного объекта приводила к необходимости перекомпиляции всего кода проекта. В результате разработка ПО была ограничена использованием только одного языка программирования и только одного компилятора.

Изменения в требованиях к информационному сервису, а также появление новых представлений о требуемых данных и функциях привели к усложнению ПО. Это потребовало создания автоматизированных технологий разработки и сопровождения ПО.

Потребность в разработке технологий, устраняющих указанные недостатки и убыстряющих написание программ, привела к появлению компонентных технологий программирования и CASE-технологий.

При компонентном подходе модель построения программы представляет собой совокупность отдельных двоичных объектов-компонентов — физически отдельно существующих частей программы, взаимодействующих между собой через стандартные двоичные интерфейсы. Компонентное программирование подразумевает наличие не только самих компонентов, но и некоторой визуальной среды разработки, позволяющей в диалоге строить программу из этих компонентов.

Объекты-компоненты можно собрать в dll-библиотеки или исполняемые файлы, использовать в любом языке программирования, применять в одном или в разных процессах, на одном или на разных компьютерах.

Технологии программирования, использующие компонентный подход, разработаны на базе технологии COM (*Component Object Model* — компонентная модель объектов) и на базе технологии создания распределенных объектов CORBA (*Common Object Request Broker Architecture* — общая архитектура с посредником обработки запросов объектов). Обе технологии основаны на общих принципах.

Объекты (компоненты) COM имеют следующие особенности:

• являются объектами (экземплярами) классов COM, содержат поля и наборы виртуальных функций, называемых интерфейсами, могут иметь несколько интерфейсов, обеспечивающих доступ к его полям и функциям;

• представлены в двоичном виде, в виде динамически компонуемых библиотек (*dynamic linking libraries, dll*);

• независимы от языков программирования.

Технология COM — это модель взаимодействия типа «клиент — сервер». Клиент — это программа или объект, использующий другой объект. Клиент подсоединяется к объекту через интерфейс. Сервер — это местоположение объектов COM, подключаемых к приложению-клиенту.

Серверы COM разделяются на три группы:

• внутренние серверы — реализованы на базе динамических библиотек и исполняются в том же процессе, что и клиент;

• локальные серверы — реализованы на базе EXE-программ, исполняются в другом процессе по сравнению с клиентом, но на одном компьютере с клиентом;

• удаленные серверы — реализованы на базе EXE-программ и исполняются на удаленном компьютере.

Технология COM фирмы Microsoft и ее распределенная версия DCOM явились основой для разработки компонентных технологий программирования.

Технология ActiveX этой фирмы нашла широкое применение за счет следующих преимуществ:

• быстрая разработка программ из компонентов;

• использование знакомых современных средств разработки: Visual Basic, Visual C++, Borland Delphi, Borland C++;

• применение визуального программирования для создания использования компонентов — элементов управления ActiveX;

• соответствие стандартам Internet и COM.

В языке Java в качестве базовой компонентной модели принята технология JavaBeans. Основой среды JavaBeans является компонентная объектная модель, представляющая собой совокупность архитектуры и прикладных программных интерфейсов. Компонент Java Bean («кофейное зерно») можно определить как многократно используемый программный объект, допускающий обработку в графическом инструментальном окружении и сохранение в долговременной памяти.

Технология JavaBeans поддерживает взаимодействие с похожими компонентными структурами. Например, Windows-программа при наличии соответствующего моста может использовать компонент JavaBeans так, будто он является компонентом COM или ActiveX.

Для создания сложных компонентно-распределенных систем используется технология Enterprise JavaBeans (EJB). Основное назначение технологии EJB — создание инфраструктуры для компонентов, позволяющей легко размещать их на серверах. Технология

освобождает программистов от необходимости детальной реализации сервисов, обеспечивающих хранение, передачу и безопасность данных. Реализация этих процессов может быть осуществлена EJB-системой, при этом у программиста имеется возможность самому контролировать и описывать данные процессы.

Технология CORBA использует принципы, аналогичные технологии COM. В ней также применяется модель взаимодействия «клиент — сервер», но организация взаимодействия производится с помощью специального посредника. Таким посредником выступает брокер запросов к объектам (*ORB, Object Request Broker*). Наиболее широко используется VisiBroker фирмы Borland. Технологию можно применять для разработки распределенного ПО в разнородной вычислительной среде.

CASE-технологии (*Computer-Aided Software/System Engineering*) — разработка программного обеспечения/программных систем с использованием компьютерной поддержки. CASE-средства представляют собой инструменты разработки, автоматизирующие процессы создания и сопровождения ПО, включая этапы анализа и проектирования ПО, генерацию кода, тестирования, документирования и другие процессы.

Большинство CASE-технологий основано на методологии процедурного и объектно-ориентированного проектирования. В настоящий момент на рынке программного обеспечения насчитывается более 300 различных CASE-средств. Наиболее известными являются CA ERwin Process Modeler (ранее BPwin), CA ERwin Data Modeler (ранее ERwin), Rational Rose, ARIS.

1.6. Перспективы развития технологий программирования

Развитие информационных технологий сопровождается появлением новых и совершенствованием существующих подходов к программированию. Все разрабатываемые технологии создания программ должны поддерживаться языками программирования. Увеличение сложности решаемых задач требует создания новых, более мощных, ориентированных на проблемную область, языков программирования. Кроме того, языки программирования должны обеспечивать продление жизненного цикла программ.

По мнению многих исследователей, развитие языков программирования в ближайшее время будет двигаться в направлении все большей абстракции, изменения уровня детализации и наибольшего упрощения. Это приведет к повышению надежности процесса создания программ и уменьшению количества допускаемых разработчиками ошибок.

Рассмотрим еще некоторые направления развития языков программирования.

В последнее время в связи развитием Интернет-технологий широкое распространение получили языки сценариев или скрипты. Первоначально эти языки использовались в качестве внутренних управляющих языков в различных сложных системах. В настоящее время многие из них уже вышли за пределы своего изначального применения и используются во многих областях.

Наиболее распространенными скриптовыми языками являются:

• JavaScript — создан в компании Netscape Communications в качестве языка для описания сложного поведения веб-страниц;

• VBScript — создан в корпорации Microsoft во многом в качестве альтернативы JavaScript;

• Perl создавался в помощь системному администратору операционной системы Unix для обработки различного рода текстов и выделения нужной информации. Развился до мощного средства работы с текстами.

Характерные особенности скриптовых языков:

• инерпретируемость (компиляция невозможна или крайне нежелательна);

• простота синтаксиса;

• легкая расширяемость.

В настоящее время получили развитие программно-аппаратные комплексы, позволяющие организовать параллельное выполнение различных частей одного и того же вычислительного процесса. Для организации подобных вычислений требуется поддержка со стороны языков программирования. Некоторые языки общего назначения содержат элементы поддержки параллелизма, однако для программирования истинно параллельных систем требуются специальные средства.

Язык Occam был создан в 1982 г. и предназначен для программирования многопроцессорных систем распределенной обработки данных. Он описывает взаимодействие параллельных процессов в виде каналов — способов передачи информации от одного процесса к другому.

В 1985 г. была предложена модель параллельных вычислений Linda. Основной ее задачей является организация взаимодействия между параллельно выполняющимися процессами. Это достигается за счет использования глобальной области данных. Один процесс может поместить туда некоторую совокупность данных, а другой — ожидать появления в области необходимой порции данных. Следует отметить, что Linda — это модель параллельных вычислений, она может быть добавлена в любой язык программирования.

В настоящее время дальнейшее развитие получает парадигма объектно-ориентированного программирования. В частности, в рамках этой парадигмы можно выделить *аспектно-ориентированное* (АОП) и *субъектно-ориентированное* (СОП) программирование.

В современном ПО, как правило, можно выделить определенные части, или *аспекты*, отвечающие за ту или иную функциональность, реализация которой рассредоточена по коду программы, но состоит из схожих кусков кода. АОП предполагает наличие языковых средств, позволяющих выделять сквозную функциональность в отдельные модули. Это позволяет упрощать работу (отладку, модифицирование, документирование и т.д.) с компонентами программной системы и снижать сложность системы в целом.

СОП — это метод построения сложных систем как композиции субъектов. Субъект — приложение, способное самостоятельно реализовывать задачу, имеющую несколько путей решения. В отличие от объекта, субъект может сам выбирать этот путь, т.е. способен корректировать последовательности своих действий для достижения поставленной цели. Стратегия управления такими приложениями должна основываться не на конкретных командах операционной системы, а на инструкциях. При управлении объектами используются их отдельные методы, субъекту же указывается номер инструкции, на основании которой он функционирует, и он самостоятельно будет управлять своими методами, чтобы достичь результата.

Контрольные вопросы

1. Какие этапы эволюции прошли технологии программирования?
2. Какие языки и методы программирования вы знаете?
3. Какие языки программирования называются языками высокого уровня?
4. Какая модель построения программ лежит в основе технологии процедурного программирования?
5. Каковы основные методы процедурного программирования?
6. На чем основывается концепция объектно-ориентированного программирования?
7. Каковы основные принципы объектно-ориентированного программирования?
8. Что такое компонентные технологии и CASE-технологии?
9. В чем преимущества и недостатки языков сценария?
10. Какова область применения языков параллельных вычислений?

Глава 2
ОСНОВНЫЕ ЭТАПЫ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

В результате изучения гл. 2 обучающиеся должны:

знать

• стандарты в области разработки и реализации программного обеспечения;

• представления о жизненном цикле программы;

• методологические принципы разработки и реализации алгоритмов и программ;

уметь

• ориентироваться в современных моделях жизненного цикла программного обеспечения;

• применять теоретические знания в области жизненного цикла к организации и разработке программного обеспечения;

владеть

• навыками выбора модели разработки программного обеспечения;

• навыками оценки качества разработанных алгоритмов и программ.

2.1. Алгоритмы и программы

Термин «алгоритм» происходит от имени узбекского математика Аль-Хорезми (825 г.), который ввел правила выполнения четырех арифметических операций в десятичной системе счисления. С 1747 г. использовался термин «алгорисмус», к 1950 г. появился термин «алгорифм» (смысл — алгорифм Евклида). Большой вклад в теорию алгоритмов в связи с эволюцией ЭВМ внесли математики Д. Гильберт, К. Гедель, А. Черч, С. Клини, А. Тьюринг, А. Марков.

Перечислим основные *свойства алгоритмов*:

• определенность (детерминированность) — точность и понятность, обеспечивающие однозначность результата;

• результативность — получение искомого результата через конечное число этапов (шагов);

• дискретность — расчлененность алгоритма на отдельные этапы (шаги);

• массовость — пригодность алгоритма для решения всех задач данного типа.

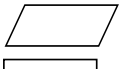
Основными требованиями к алгоритму (как и к программе) являются компактность и минимальное время реализации.

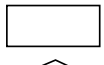
Существуют следующие способы записи алгоритмов:

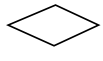
• псевдокод (формульно-словесный) — содержание последовательных этапов вычислений задается с нумерацией в произвольной форме на естественном языке; это искусственный, неформальный язык, но не язык программирования;

• блок-схемный — изображение структуры алгоритма в виде графического представления этапов процесса обработки данных с помощью специальных геометрических символов (блоков). Основными блоками, используемыми для записи алгоритма, являются:

—  — начало/конец алгоритма,

—  — ввод/вывод данных,

—  — процесс,

—  — блок выбора решения;

• алгоритмические языки — непосредственная запись алгоритма решения задачи с помощью операторов языка.

Рассмотрим различные способы записи алгоритма для решения следующей задачи: ввести два числа, a и b. Переменной m присвоить значение наибольшего из этих чисел. Значение переменной m вывести. Алгоритм можно записать с помощью псевдокода или блок-схемы.

Запись алгоритма псевдокодом может выглядеть так:

1) ввести два числа: a и b;

2) если a больше b, переменной m присвоить значение a, иначе b;

3) вывести значение m.

Блок-схема, содержащая те же этапы, показана на рис. 2.1.

Программа — это упорядоченная последовательность команд компьютера, необходимая для решения конкретной задачи.

Программы не имеют заранее строго регламентированного набора качественных характеристик. Перечислим основные *характеристики программ*:

• алгоритмическая сложность (логика алгоритмов обработки информации);

• глубина проработки, полнота и системность реализованных функций программы;

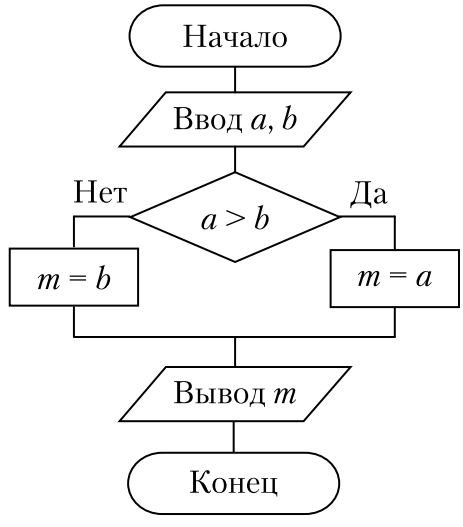


Рис. 2.1. **Блок-схема алгоритма**

• требования к операционной системе и техническим средствам со стороны программы;

• удобство освоения и эксплуатации программы.

Критериями качества программы являются:

• мобильность — независимость программы от программных и технических средств обработки данных;

• надежность — устойчивость в работе программы, точность выполнения функций обработки;

• эффективность — оценка расходов вычислительных ресурсов (объемов памяти для эксплуатации программы и т.п.);

• дружественность — обеспечение дружественного интерфейса для работы пользователя;

• модифицируемость — способность к внесению изменений и расширений функций обработки;

• коммуникативность — возможность интеграции с другими программами.

На качество программы влияют следующие факторы:

• маркетинг рынка и спецификация требований к программе — определение состава функций обработки данных программы, выбор пользовательского интерфейса, требования к комплексу технических и программных средств;

• проектирование структурной схемы программы — алгоритмизация процессов обработки данных, разработка структуры программы и базы данных, выбор метода и средств создания программы — технологии программирования;

• программирование (алгоритмизация, тестирование и отладка) — техническая реализация проекта программы, выполняемая

с помощью выбранного инструментария технологии программирования (алгоритмические языки и системы программирования);

• эксплуатация и сопровождение программы — важный этап, связанный с устранением обнаруженных ошибок;

• распространение программы и завершение жизненного цикла — этап, связанный с постоянной программой маркетинговых мероприятий и заканчивающийся либо отсутствием спроса, либо изменением технической политики разработчика.

2.2. Жизненный цикл программы

Жизненный цикл программного обеспечения (ЖЦ ПО) — это совокупность взаимосвязанных процессов от момента появления идеи создания программного обеспечения до окончания его эксплуатации.

Описание структуры ЖЦ ПО и состав его процессов регламентируется международным стандартом ISO/IEC 12207: 1995 «Information Technologe — Software Life Cycle Processes» («Информационные технологии — Процессы жизненного цикла программного обеспечения»). Отечественный вариант стандарта называется ГОСТ Р ИСО/МЭК 12207—2010 «Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств».

Согласно этому стандарту, архитектура ЖЦ ПО состоит из трех компонентов:

1) *Основные процессы*: Приобретение, Поставка, Разработка, Эксплуатация, Сопровождение;

2) *Поддержка*: Документирование, Конфигурационное управление, Обеспечение качества, Верификация, Валидация (проверка правильности), Управление проектом, Ревизия отчетов, Решение задач (устранение дефектов);

3) *Организация*: Управление, Инфраструктура, Совершенствование, Обучение.

Стандарт состоит из семи разделов и четырех приложений. Разделы 1—4 являются вводными. Разделы 5—7 состоят из подразделов, в которых раскрывается содержание работ и даются комментарии к ним. Общее число работ и комментариев превышает 220.

Процесс **Разработка программного обеспечения** (разд. 5.3) содержит 13 подразделов:

• *подготовка процесса* — выбор модели жизненного цикла, стандартов, составление плана работ;

• *анализ требований к системе* — определение функциональных, эксплуатационных, пользовательских требований;

• *проектирование архитектуры системы* — определение состава оборудования, ПО и операций, выполняемых персоналом;

• *анализ требований к программному обеспечению* — определение функциональных возможностей, в том числе среды функционирования компонентов, внешних интерфейсов, требований к данным, установке, приемке, документации, эксплуатации и сопровождению;

• *проектирование архитектуры программного обеспечения* — определение структуры программного обеспечения, документирование интерфейсов его компонентов;

• *детальное проектирование программного обеспечения* — описание компонентов ПО и интерфейсов между ними, разработка требований к тестам;

• *кодирование и тестирование компонентов программного обеспечения* — разработка и тестирование компонентов;

• *интеграция компонентов программного обеспечения* — сборка программных компонентов в соответствии с планом интеграции, тестирование ПО на соответствие требованиям, соответствующим своим спецификациям;

• *квалификационное тестирование программного обеспечения* — тестирование в присутствии заказчика, проверка документации;

• *интеграция системы* — сборка всех компонентов системы, ПО и оборудования;

• *квалификационное (аттестационное) тестирование системы* — тестирование системы на соответствие требованиям, проверка оформления и полноты документации;

• *инсталляция программного обеспечения* — установка программного обеспечения на оборудовании заказчика и проверка его работоспособности;

• *приемка программного обеспечения* — оценка результатов квалификационного тестирования и передача программного обеспечения заказчику.

Можно выделить следующие укрупненные этапы разработки ПО с учетом соответствующих *стадий разработки* по ГОСТ 19.102—77 «Стадии разработки»:

• постановка задачи (стадия «Техническое задание»);

• определение спецификаций (стадия «Эскизный проект»);

• проектирование (стадия «Технический проект»);

• реализация (стадия «Рабочий проект»).

По стандарту возможно также наличие отдельного этапа сопровождения, заключающегося в сопровождении и модификации программного продукта.

Деление на этапы является условным и может изменяться при необходимости. Например, тестирование и отладка могут быть ча-

стью этапа реализации (или программирования), а могут быть отдельным этапом.

Постановка задачи — это процесс формулировки назначения ПО и основных требований к нему.

Спецификациями называют полное и точное описание функций и ограничений разрабатываемого ПО.

Проектирование — это процесс разработки структурной схемы ПО с проектированием компонентов и их взаимосвязей.

Реализация — это процесс программирования компонентов ПО на выбранном языке программирования, их тестирования и отладки.

Разработка программ является одной из самых длительных стадий создания ПО. Поэтому оценка сложности и трудоемкости этого этапа оказывает основное влияние на планирование всего комплекса работ в целом.

К оценке длительности и трудоемкости разработки ПО предъявляются следующие требования:

- использование предыдущего опыта оценки аналогичных программных модулей;
- оценка всего комплекса работ, включая новую разработку, внесение изменений в существующие модули, руководство процессом разработки и тестирование результатов;
- учет квалификации сотрудников, выполняющих разработку;
- оценка точности полученных результатов.

На практике применяются следующие основные методы оценки трудоемкости разработки ПО:

- методика COCOMO;
- оценка трудозатрат по функциональным точкам;
- экспертная оценка.

Методика COCOMO (Constructive Cost Model) базируется на оценке трудоемкости в зависимости от количества строчек программного кода. Эта зависимость выражается формулой

$$T = \alpha \cdot KLOC^{\beta} \cdot EAF, \qquad (2.1)$$

где T — трудозатраты (чел/мес); $KLOC$ — количество строчек кода (без учета комментариев); EAF — фактор корректировки трудозатрат в зависимости от факторов среды; α и β — константы, определенные в процессе анализа проекта.

Данный метод позволяет приблизительно оценить трудоемкость всего проекта в целом на начальной стадии разработки.

Метод оценки трудозатрат по функциональным точкам рассчитывает количество строчек программного кода, приходящихся на реализацию одной функциональной задачи проекта в зависимости от используемого языка программирования.

В рамках этого подхода производится приблизительная оценка количества функций, входящих в реализацию каждого функционального требования. Полученное в результате количество функциональных точек переводится в условные строки программного кода.

Данный метод дает возможность получить приблизительные усредненные результаты, не учитывающие квалификацию персонала, занимающегося разработкой.

Метод экспертной оценки учитывает мнения экспертов при оценке затрат на реализацию каждого функционального требования. Каждый эксперт дает три оценки — наилучшую, наихудшую и наиболее вероятную. Общая оценка определяется по формуле

$$T = \sum_i \frac{L_i + 4 \cdot M_i + S_i}{6}, \qquad (2.2)$$

где L_i — наилучшая оценка; S_i — наихудшая оценка; M_i — наиболее вероятная оценка.

Данный метод отличается достаточной субъективностью. Мнения экспертов не всегда являются достаточно обоснованными при новых разработках. Кроме того, при корректировке проектных решений требуется повторное привлечение экспертов.

Разработка программных средств, как правило, ведется коллективом специалистов. От правильной организации этого коллектива во многом зависит успешность всего проекта.

Можно выделить следующие виды разработки программных средств:

- авторская разработка;
- коллективная разработка.

При *авторской разработке* весь жизненный цикл разработки поддерживается одним человеком — автором. Такой метод применим в области создания наукоемких приложений, для реализации которых требуется детальное изучение предметной области, а круг потребителей приложения достаточно узок. Однако для современных массовых разработок такой метод неприменим из-за их объема и сложности, а также необходимости их длительной поддержки и сопровождения.

При *коллективной разработке* производится разделение труда между работниками, занятыми разработкой. При этом могут использоваться следующие модели:

Иерархическая модель характеризуется наличием руководящего звена (главного разработчика), который осуществляет разделение большой разработки на более мелкие части, передаваемые по иерархической лестнице руководящим звеньям более низкого уровня

вплоть до конкретного специалиста. Сборка готового программного продукта ведется в обратном направлении от более низкого к более высокому звену. При этом процесс сборки достаточно затруднителен и не всегда успешен.

В *матричной модели* коллектив разработчиков состоит из равноправных специалистов, занимающихся приблизительно одинаковыми задачами в рамках единого проекта. Разделение обязанностей внутри коллектива происходит в соответствии с особенностями проекта. Координация действий внутри такого коллектива затруднена.

В *модели команды главного программиста* один специалист занимается непосредственно разработкой. Он принимает все проектные решения, осуществляет написание и отладку кода, а также подготовку проектной документации. Все остальные члены коллектива выполняют поручения главного программиста и оказывают ему всестороннюю помощь и поддержку. Такая модель применима для небольших проектов, когда трудоемкость разработки относительно невелика.

Ядерная модель предполагает наличие одного главного разработчика, задачей которого является создание прототипа системы. Затем на основе прототипа коллектив разработчиков создает готовый программный продукт.

Процесс обеспечения качества предполагает обеспечение гарантий того, что программные продукты и процессы соответствуют установленным требованиям и утвержденным планам.

В целях обеспечения объективности и беспристрастности контроль качества должен производиться субъектами, непосредственно не связанными с разработкой программного продукта или выполнением процесса в проекте.

При обеспечении качества могут использоваться результаты других вспомогательных процессов, таких как верификация, аттестация, совместные анализы, аудит и решение проблем.

Процесс обеспечения качества состоит из следующих работ:

- подготовка процесса;
- обеспечение качества продукта;
- обеспечение процесса;
- обеспечение систем качества.

Подготовка процесса предполагает адаптацию процесса обеспечения качества к условиям конкретного проекта. В рамках данной работы должен быть разработан план выполнения задач процесса обеспечения качества, содержащий стандарты качества, методологии, процедуры и средства для выполнения работ по обеспечению качества. Лица, отвечающие за соблюдение соответствия условиям

договора, должны быть организационно независимы и иметь ресурсы и полномочия для работы.

Обеспечение качества продукта означает, что все программные продукты должны быть изготовлены по условиям договора, полностью соответствовать требованиям, указанным в договоре, и удовлетворять заказчика.

Обеспечение процесса подразумевает, что все характеристики программного продукта и процессов соответствуют установленным стандартам и процедурам, а персонал, участвующий в реализации проекта, обладает достаточным опытом и знаниями и способен к обучению.

Обеспечение систем качества означает проведение дополнительных работ по управлению качеством в соответствии с разделами стандарта, указанными в договоре.

Рассмотрим эволюцию моделей жизненного цикла программного обеспечения.

Можно выделить три модели жизненного цикла программного обеспечения:

- каскадная модель;
- модель с промежуточным контролем;
- спиральная модель.

Каскадная модель предполагает переход на следующий этап после завершения всех операций предыдущего этапа. Достоинства модели заключаются в следующем:

- получение после этапа законченного набора проектной документации без возврата на предыдущие этапы;
- простота планирования процесса разработки программного обеспечения.

Однако данная модель применима только к созданию систем, для которых точно и полно сформулированы все требования. Такие разработки встречаются редко. Необходимость возвратов на предыдущие этапы бывает обусловлена следующими причинами:

- неточные спецификации, уточнение которых требует пересмотра предыдущих решений;
- изменения требований заказчика;
- моральное устаревание технических и программных средств.

Модель с промежуточным контролем поддерживает итерационный характер процесса разработки, т.е. возврат на предыдущие этапы. Опасность использования модели состоит в том, что разработка программного обеспечения может затянуться.

Спиральная модель основана на том, что программное обеспечение создается не сразу, а итерационно с использованием метода прототипирования. Прототипом называется программный про-

дукт, реализующий внешние интерфейсы и отдельные функции. Например, на первой итерации создается и поставляется пользователю первая версия программного продукта с реализацией внешних интерфейсов и главных функций, на следующих итерациях — следующие версии программного продукта с реализацией дополнительных функций.

Достоинства спиральной модели заключаются в следующем:

- программный продукт может поставляться пользователю с первой версии;
- сокращение времени появления первых версий программного продукта;
- быстрое продвижение следующих версий продукта на рынке;
- ускорение уточнений спецификаций за счет появления практики использования продукта;
- уменьшение вероятности морального устаревания системы за время разработки.

Недостатки спиральной модели состоят в трудности управления временем разработки и, главное, в дороговизне такой организации работ, предполагающей постоянное использование всех категорий специалистов в одном проекте.

2.3. Постановка задачи и спецификация программы

Постановка задачи — это процесс формулировки назначения программного обеспечения и основных требований к нему. Описываются *функциональные требования*, определяющие функции, которые должно выполнять программное обеспечение, и *эксплуатационные требования*, определяющие характеристики его функционирования. Этап постановки задачи заканчивается разработкой *технического задания* с принятием основных проектных решений.

Техническое задание в соответствии со стандартом ГОСТ 19.201—78 «Техническое задание. Требования к содержанию и оформлению» имеет следующие основные разделы:

- введение: наименование и краткая характеристика программного обеспечения;
- основание для разработки;
- назначение разработки: описание функционального и эксплуатационного назначения, спецификации функций;
- требования к программному изделию: к функциональным характеристикам, к надежности, к техническим средствам;
- требования к программной документации;
- технологические требования.

Технологические требования определяют выбор следующих принципиальных решений, влияющих на процесс проектирования программного обеспечения:

- архитектура программного обеспечения;
- пользовательский интерфейс;
- метод программирования;
- язык программирования;
- среда программирования.

Архитектурой программного обеспечения называют описание создаваемого программного обеспечения на уровне его компонентов и связей между ними.

Архитектура программной системы во многом зависит от предметной области, для которой разрабатывается система. Поэтому часто архитектуры систем, разрабатываемых для одной и той же предметной области, имеют много общего. Следовательно, при проектировании архитектуры новой системы можно воспользоваться решениями, удачно примененными в ранее разработанных системах.

Развитие данного подхода привело к появлению *архитектурных шаблонов* (паттернов), на основе которых может создаваться архитектура конкретной системы. Архитектурный паттерн представляет собой описание типовой организации программной системы, опробованной в различных условиях и продемонстрировавшей свою эффективность.

Распространенными являются следующие архитектурные паттерны:

- модель-представление-контроллер (*Model-View-Controller*, *MVC*) применяется в системах, ориентированных на обслуживание клиентов;
- паттерн хранилища данных применяется в системах, функционирование которых связано с обработкой большого объема данных (информационные системы, системы управления); архитектура таких систем должна содержат компоненты, генерирующие данные, и компоненты, их обрабатывающие;
- паттерн «клиент-сервер» предусматривают распределение заданий между поставщиками и заказчиками услуг. Поставщики (серверы) предоставляют заказчикам (клиентам) определенный набор услуг (сервисов), доступ к которым осуществляется с помощью удаленного вызова соответствующих процедур;
- паттерны потоков данных предполагают построение архитектуры системы из функциональных модулей, которые получают входные данные и преобразуют их в выходные. Преобразования могут осуществляться как последовательно, так и параллельно;

• многоуровневая система представляет собой иерархическую систему, состоящую из нескольких уровней, каждый из которых выполняет определенные функции. Каждый уровень предоставляет услуги вышестоящему уровню и использует услуги нижестоящего уровня.

При процедурном программировании модель построения программы — иерархия множества подпрограмм, при объектно-ориентированном программировании — иерархия множества классов, совокупность обменивающихся сообщениями объектов классов. При этом простом виде архитектуры программа — это совокупность отдельно компилируемых программных единиц, используемая при решении задач.

Пакеты программ — это совокупность программ, решающих задачи некоторой прикладной области. Например, пакет математических программ, пакет графических программ.

Программные комплексы — это совокупность взаимосвязанных программ, совместно решающих небольшой класс задач некоторой прикладной области. Для решения задачи могут использоваться несколько программ комплекса, управляемые интерфейсом с пользователем, причем исходные данные и результаты желательно хранить в пределах одного проекта.

Программные системы — это совокупность подсистем программ, совместно решающих большой класс сложных задач некоторой прикладной области. Отличие от программных комплексов — взаимодействие программ системы через общие данные и наличие развитых пользовательских интерфейсов.

Пользовательский интерфейс представляет собой совокупность программных и аппаратных средств, обеспечивающих диалог пользователя и компьютера. Различают следующие типы пользовательских интерфейсов:

• примитивные интерфейсы;
• интерфейсы-меню;
• интерфейсы со свободной навигацией;
• интерфейсы прямого манипулирования.

К технологическим требованиям к программному обеспечению относятся:

• выбор метода программирования: процедурный или объектно-ориентированный;
• выбор языка программирования: C++, Java, Python и др.;
• выбор среды программирования: Visual Studio фирмы Microsoft, Embarcadero RAD Studio (включающая Delphi и C++ Builder), Eclipse и др.

Спецификациями называют полное и точное описание функций и ограничений разрабатываемого программного обеспечения. Существуют функциональные спецификации, описывающие функции разрабатываемого программного обеспечения, и эксплуатационные спецификации, описывающие требования к техническим средствам. Требование полноты означает строгое соответствие техническому заданию, а требование точности — однозначное толкование спецификаций разработчиком и заказчиком.

Спецификации содержат:

- декомпозицию и содержательную постановку задач;
- эксплуатационные ограничения;
- математические методы решения;
- модели программного обеспечения.

Модели программного обеспечения зависят от технологии программирования (процедурная или объектно-ориентированная).

При процедурном программировании используются следующие модели разрабатываемого программного обеспечения:

- функциональные диаграммы, отражающие взаимосвязи функций разрабатываемого программного обеспечения и ориентированные на иерархию функций, декомпозицию функций;
- диаграммы потоков данных, описывающие взаимодействие источников и потребителей информации и позволяющие специфицировать как функции, так и данные;
- диаграммы структур данных, описывающие базы данных разрабатываемого программного обеспечения;
- диаграммы переходов состояний, описывающие поведение разрабатываемого программного обеспечения при получении управляющих данных извне (например, команды пользователя).

При объектно-ориентированном программировании модели разрабатываемого программного обеспечения основаны на предметах реального мира. На этапе определения спецификаций требуется разработать модель предметной области программного обеспечения на базе иерархии классов (типов, определенных пользователем), объектной декомпозиции. Разрабатываемое программное обеспечение представляется в виде совокупности объектов (экземпляров классов), в результате взаимодействия которых через передачу сообщений происходит выполнение требуемых функций.

В настоящее время стандартным средством описания разрабатываемого программного обеспечения с использованием объектно-ориентированного подхода является фактически графический язык UML (*Unified Modeling Language,* универсальный язык моделирования), разработанный авторами Г. Буч, Д. Рамбо и И. Якобсоном в 1995 г.

Язык UML описывает модель сложной системы в виде специальных графических конструкций (диаграмм).

Диаграмма представляет собой связный граф, вершины которого представляются в виде геометрических фигур, связанных между собой ребрами (дугами).

Все диаграммы используют единую графическую нотацию.

В диаграммах используется три типа графических элементов, имеющих различное назначение:

• геометрические фигуры обозначают вершины графа. Их форма строго соответствует определенным элементам языка (класс, вариант использования, состояние, деятельность). Каждой вершине присваивается уникальное имя;

• различные линии (ребра), соединяющие геометрические фигуры (вершины графа) обозначают связи и взаимодействия элементов;

• специальные графические символы, располагаются около других элементов диаграммы и содержат дополнительную информацию. Дополнительный текст может размещаться также внутри контуров геометрических фигур.

Нотация языка UML определяет использование следующих основных видов диаграмм:

• вариантов использования (прецедентов) — описывают основные функции системы;

• классов — описывают классы, их характеристики (поля и операции) и связи между ними;

• кооперации — показывают взаимодействие объектов в процессе реализации варианта использования;

• пакетов — показывают связи наборов классов, объединенных в пакеты;

• состояний — демонстрируют состояния объекта и переходы из одного состояния в другое;

• деятельности — представляют собой схему потоков управления для решения некоторой задачи;

• компонентов — описывают компоненты программного обеспечения и их связи между собой;

• развертывания — позволяют связывать программные и аппаратные компоненты системы.

Каждая из этих диаграмм детализирует и конкретизирует различные представления о модели системы в терминах языка. Совокупность этих диаграмм содержит всю необходимую информацию для реализации проекта сложной программной системы. При создании моделей следует придерживаться следующих основных рекомендаций.

Любая модель должна содержать только те элементы, которые определены в нотации языка UML. Каждый элемент может быть использован только в соответствии с назначением и по правилам, определенным в языке.

Каждая диаграмма должна полностью описывать рассматриваемый фрагмент предметной области, на ней должны быть представлены все значимые элементы. Отсутствие каких-либо элементов может привести к серьезным ошибкам в моделировании.

Все элементы диаграммы должны принадлежать к одному уровню представления. При моделировании сложных систем часть используют иерархический подход, при котором диаграммы нижнего уровня уточняют и детализируют элементы диаграмм высших уровней.

Желательно информацию обо всех элементах диаграммы представлять в явном виде, не используя значения, заданные по умолчанию. Это ускорит и упростит процесс реализации модели.

Каждая модель конкретной программной системы может содержать все или только некоторые типы диаграмм.

2.4. Проектирование и реализация программы

Проектирование — это процесс разработки структурной схемы программного обеспечения с проектированием компонентов и их взаимосвязей.

Методологической основой проектирования программного обеспечения является системный подход.

Системный подход — это методология специального научного познания, в основе которого лежит исследование объектов как систем. При этом исследование объектов нацелено:

- на раскрытие целостности объекта и обеспечивающих его механизмов;
- выявление многообразных типов связей сложного объекта;
- сведение этих связей в единую теоретическую картину.

Системный подход реализует представление сложного объекта в виде иерархической системы взаимосвязанных моделей, позволяющих фиксировать целостные свойства объекта, его структуру и динамику.

Методология структурного анализа и проектирования ПО определяет руководящие указания для оценки и выбора проекта разрабатываемого ПО, шаги работы, которые должны быть выполнены, их последовательность, правила распределения и назначения операций и методов.

Структурный анализ — это метод исследования системы, базирующейся на декомпозиции системы, начиная с самого общего об-

зора с постепенной детализацией на каждом следующем уровне. В результате образуется иерархическая структура со множеством уровней, на каждом из которых детализируются элементы предыдущего уровня.

Методы структурного анализа основываются на соблюдении следующих правил:

• разбиение системы на уровни абстракции с ограничением числа элементов на уровне;

• включение на каждом уровне только существенных для этого уровня деталей;

• использование строгих формальных правил записи и условных обозначений — нотаций;

• последовательное приближение к конечному результату.

Основными средствами структурного анализа являются:

• DFD (*Data Flow Diagrams*) — диаграммы потоков данных в нотациях Гейна — Сарсона, Йордона — Де Марко и др., обеспечивающие требования анализа и функционального проектирования информационных систем;

• STD (*State Transition Diagrams*) — диаграммы перехода состояний, основанные на расширениях Хартли и Уорда — Меллора для проектирования систем реального времени;

• ERD (*Entity-Relationship Diagrams*) — диаграммы «сущность-связь» в нотациях Чена и Баркера; структурные карты Джексона и (или) Константайна для проектирования межмодульных взаимодействий и внутренней структуры объектов;

• FDD (*Functional Decomposition Diagrams*) — диаграммы функциональной декомпозиции;

• SADT (*Structured Analysis and Design Technique*) — технология структурного анализа и проектирования;

• семейство IDEF (*Integration Definition for Function Modeling*).

Тип модели программного обеспечения определяется выбранной технологией — методом программирования (процедурный, объектно-ориентированный, компонентный).

При *процедурном подходе* модель построения программного обеспечения — иерархия функций, т.е. декомпозиция модели программы по функциональному принципу.

Проектирование заключается в декомпозиции модели программного обеспечения методом пошаговой детализации. В результате такой декомпозиции происходит постепенное разделение процедур на более мелкие функции, каждая из которых реализует некоторую часть общего процесса. В результате получается структурная схема модели программы, представляющая собой многоуровневую, иерархическую схему взаимодействия подпрограмм по управлению.

Метод пошаговой детализации применяет нисходящую пошаговую разработку алгоритма, когда на каждом шаге происходит разложение функции на подфункции.

При *объектно-ориентированном подходе* модель построения программы — это иерархия классов, объектная декомпозиция. В результате декомпозиции получается структурная схема программы, представляющая собой многоуровневую, иерархическую схему взаимодействия экземпляров (объектов) классов. Следующий этап проектирования программного обеспечения заключается в разработке классов и их интерфейсов с описанием элементов-данных и элементов-функций каждого класса. Для проектирования пользовательских интерфейсов используются сложные интерфейсы: меню с иерархической структурой команд, свободная навигация, не привязанная к уровням иерархии (используется в Windows-приложениях).

Этапы проектирования программного обеспечения при объектно-ориентированном подходе принципиально отличаются от процедурного подхода, так как проектирование программы ведется в терминах (понятиях) прикладной области и отражает ее иерархию. *Реализация* — это процесс создания кода компонентов программного обеспечения на выбранном языке программирования, его тестирования и отладки.

При процедурном подходе реализация заключается в программировании функций и файлов (модулей) с использованием методов структурного программирования функций и программирования «сверху-вниз» (*top-down*).

Этап программирования задачи выполняется в следующей последовательности:

- программирование функций верхнего уровня схемы;
- программирование функций нижнего уровня схемы и т.д.

При программировании и отладке функций верхнего уровня функции нижнего уровня, текста которых еще нет, имитируются «заглушками», т.е. выводом сообщений о вызове этих функций.

При объектно-ориентированном подходе этап реализации заключается в программировании элементов-функций целых взаимосвязанных классов, начиная с базовых классов.

Тестирование и отладка являются завершающими этапами процесса реализации программного обеспечения. Тестирование позволяет обнаружить ошибки, а отладка — их исправить.

В соответствии с этапами обработки программы (компиляция, компоновка, выполнение) различают следующие группы ошибок:

- синтаксические ошибки, обнаруживаемые компилятором при синтаксическом и семантическом анализе программы;

• ошибки компоновки, фиксируемые компоновщиком (редактором связей) при объединении модулей программы;

• ошибки выполнения, обнаруживаемые операционной системой или пользователем при выполнении программы.

Из всех групп ошибок самыми сложными для тестирования и отладки являются ошибки выполнения программы, а среди них — логические ошибки, имеющие непредсказуемые причины. Так, причинами могут быть ошибки при проектировании программы, разработке алгоритмов, определении структуры данных.

Тестирование — это процесс выполнения программы на тестовых наборах с целью обнаружения ошибок, допущенных при реализации программы. Согласно рекомендациям Microsoft, различают следующие стадии тестирования:

• модульное тестирование, проверяющее небольшие отдельные части программы (циклы, блоки, подпрограммы);

• компоновочное тестирование, проверяющее следующий уровень — программные файлы (модули), объединение, взаимодействие отдельных частей программы;

• системное тестирование, проверяющее полную версию программы, взаимодействие с операционной системой;

• стресс-тестирование, изучающее работу программы при ограниченных системных ресурсах;

• бета-тестирование, позволяющее узнать мнение специалистов-пользователей;

• приемно-сдаточное тестирование — приемка пользователями в реальных условиях. Тестовый набор должен содержать для каждого теста: описание тестируемого элемента, цель и инструкцию проведения теста, исходные данные и ожидаемые результаты, описание среды тестирования и др.

Существуют и другие виды тестирования, направленные на проверку различных аспектов корректности кода программы и его соответствия техническому заданию. Отдельной разновидностью можно считать методологию разработки через тестирование (TDD, *Test-Driven Development*), согласно которой тесты для различных функций программы разрабатываются до создания кода этих функций. Такая методика позволяет сократить объем программного кода и повысить его надежность, но увеличивает затраты на разработку.

Отладка — это процесс поиска и исправления ошибок, обнаруженных при тестировании программы. Имеются методы отладки программного обеспечения, основанные на анализе текста программы и результатов тестирования без дополнительной информации. Например, метод индукции включает следующие процессы отладки: выявление симптомов ошибки, изучение фрагмента програм-

мы, выдвижение гипотезы об ошибке, проверка гипотезы и, при необходимости, выдвижение новой гипотезы, нахождение ошибки.

Имеются также методы отладки, позволяющие получать дополнительную информацию об ошибке и облегчающие процесс поиска и исправления ошибки. К ним относятся метод отладочного вывода и интегрированные средства отладки. Метод отладочного вывода заключается в добавлении в программу дополнительного отладочного вывода в узловых точках. Но, безусловно, наиболее эффективными методами являются интегрированные средства отладки, имеющиеся в современных средах программирования. Например, отладчик Visual C++ встроен в среду Visual Studio, имеет свои меню и панели инструментов, которые позволяют выполнять следующие действия:

- установка различных точек прерывания, связанных с кодом, с данными, с сообщением, условных точек прерывания;
- выполнение программы до точки прерывания;
- просмотр в шести окнах отладчика информации о текущем состоянии программы при остановке программы в точке прерывания и при пошаговом выполнении;
- пошаговое выполнение программы с заходом в функции и без захода;
- выполнение программы до строки с курсором.

Обеспечение устойчивости программной системы достигается с помощью *защитного программирования*. В рамках этого подхода в текст модуля включают проверки корректности входных и выходных данных в соответствии со спецификацией этого модуля. В случае отрицательного результата проверки возбуждается соответствующая исключительная ситуация. Для этого в модуль включаются обработчики соответствующих исключительных ситуаций. Эти обработчики, помимо выдачи необходимой диагностической информации, могут принять меры либо по исключению ошибки в данных, либо по ослаблению влияния ошибки.

Применение защитного программирования модулей приводит к снижению эффективности программного обеспечения как по времени, так и по памяти. Поэтому необходимо разумно регулировать степень применения защитного программирования в зависимости от требований к надежности и эффективности программного обеспечения.

2.5. Документирование программ

Процесс разработки программного обеспечения сопровождается созданием большого количества документов. Эти документы сопровождают процессы:

• управления разработкой программного обеспечения;

• передачи информации между разработчиками программного обеспечения;

• передачи пользователям информации, необходимой для эксплуатации программного обеспечения.

Вся документация делится на две основные группы:

• документы управления разработкой программного обеспечения;

• документы, входящие в состав программного обеспечения.

К первой группе относятся документы, связанные с разработкой и сопровождением программного обеспечения, которые обеспечивают связи внутри коллектива разработчиков и между коллективом разработчиков и лицами, управляющими процессом разработки. Сюда включаются различные стандарты, планы, отчеты, а также рабочие документы и переписка между разработчиками и управляющими разработкой.

Ко второй группе относятся документы, включаемые в состав программного обеспечения. Эти документы в свою очередь делятся на две группы:

• документация по сопровождению;

• пользовательская документация.

В состав документации по сопровождению включаются документы, описывающие структуру и состав ПО: архитектуру программной системы, спецификации и тексты программных модулей, средства тестирования, а также руководство по сопровождению программной системы с указанием аппаратно- и программно-зависимых ее частей.

К пользовательской документации относятся документы, описывающие процессы установки и настройки ПО, а также инструкции системным администраторам и конечным пользователям по эксплуатации программных систем.

Качество программной документации напрямую влияет на успех эксплуатации ПО в целом. В связи с этим для обеспечения качества программной документации разработан ряд стандартов, в которых предписывается порядок разработки этой документации, формулируются требования к каждому виду документов и определяются их структура и содержание.

Контрольные вопросы

1. Каковы основные этапы решения задач на ЭВМ?

2. Что такое жизненный цикл программного обеспечения?

3. Какие модели жизненного цикла программного обеспечения вы знаете?

4. Что называется архитектурой программного обеспечения?
5. Каковы основные типы пользовательских интерфейсов?
6. Что такое спецификации, какие сведения они содержат?
7. Какие диаграммы включает язык UML?
8. Что представляет собой структурный анализ?
9. Какие правила лежат в основе структурного анализа?
10. Какая модель построения программы используется при объектно-ориентированном подходе?
11. В чем заключается этап реализации программного обеспечения?
12. Какие методы оценки трудоемкости разработки программного обеспечения вы знаете?
13. Какие способы записи алгоритма вы знаете?
14. Какие виды организации коллектива разработчиков программного обеспечения вы знаете?
15. Какие работы выполняются в процессе обеспечения качества программного продукта?
16. Какие основные группы ошибок в программных продуктах вы знаете?
17. Какие стадии тестирования ПО вы знаете?
18. Каковы основные методы отладки?
19. Что подразумевается под защитным программированием?
20. Каковы основные группы документации программного обеспечения?
21. Какими свойствами обладают алгоритмы?
22. Какие существуют формы записи алгоритма?
23. По каким критериям оценивается качество программы?
24. Какие факторы влияют на качество программ?
25. Что представляет собой инструментарий технологии программирования?

Глава 3
ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС

В результате изучения гл. 3 обучающиеся должны:

знать

• понятие и виды пользовательских интерфейсов;

• основные компоненты пользовательских интерфейсов;

уметь

• ориентироваться в существующих видах пользовательских интерфейсов;

• выбирать наиболее подходящие типы пользовательских интерфейсов для решения конкретных задач;

владеть

• спецификой реализации различных типов диалогов;

• навыками выбора основных компонентов интерфейса для реализации диалога с пользователем.

3.1. Типы пользовательских интерфейсов

Пользовательский интерфейс представляет собой совокупность программных и аппаратных средств, обеспечивающих диалог пользователя и компьютера. *Диалог* — это процесс обмена информацией между пользователем и программой, осуществляемый в реальном масштабе времени и служащий для решения конкретной задачи. Различают следующие типы пользовательских интерфейсов:

• примитивные интерфейсы;

• интерфейсы-меню;

• интерфейсы со свободной навигацией;

• интерфейсы прямого манипулирования.

Примитивный интерфейс реализует единственный сценарий работы программного обеспечения: например, ввод данных — обработка — вывод результатов. Подобный интерфейс применяется крайне редко, например, когда программа имеет только одну функцию.

Интерфейс-меню реализует множество сценариев работы с иерархической структурой команд, выбираемых пользователем. Иерархическая организация меню с графом типа «дерева» задает строго ограниченную навигацию: вверх-вниз, вправо-влево по ветвям графа.

Интерфейс со свободной навигацией реализует множество сценариев работы, не привязанных к уровням иерархии команд. Такой интерфейс предполагает определение множества команд на конкретном шаге работы. Такие интерфейсы называют графическими пользовательскими интерфейсами (*GUI, Graphic User Interface*) или интерфейсами WYSIWYG (*What You See Is What You Get,* что видишь, то и получишь). Преимущества интерфейсов со свободной навигацией по сравнению с интерфейсами-меню заключаются в возможности доступа к любым командам и в выборе команд, имеющих смысл в данный момент. Диалоговые окна программы содержат меню различных видов (кнопочное, ниспадающее, контекстное) и элементы управления вводом/выводом, реализующие диалог с пользователем.

Интерфейс прямого манипулирования реализует множество сценариев работы посредством выбора и перемещения пиктограмм, определяющих объекты предметной области. Этот тип интерфейса реализован в интерфейсе операционной системы Windows, является альтернативой интерфейсу со свободной навигацией.

Тип интерфейса зависит от технологических требований программного обеспечения:

- выбор метода программирования: процедурный или объектно-ориентированный;
- выбор языка программирования: C++, Java, Python и др.;
- выбор среды программирования: Visual Studio фирмы Microsoft, Embarcadero RAD Studio и др.

Использование интерфейса со свободной навигацией или интерфейса прямого манипулирования требует выбора объектно-ориентированного подхода к программированию и выбора современной среды визуального программирования, например, Visual Studio, RAD Studio или Eclipse.

В последнее время большое распространение получили веб-интерфейсы. Веб-интерфейс — это среда взаимодействия пользователя и программы, запущенной на удаленном сервере. Чаще всего такие интерфейсы применяются для работы с различными онлайн-сервисами: начиная с электронной почты и заканчивая системами веб-аналитики. Взаимодействие с сервисом при этом происходит через специальную графическую оболочку, состоящую из кнопок, окон, полей ввода и других элементов.

3.2. Классификация диалогов и их реализация

Как уже отмечалось выше, диалог — это процесс обмена информацией между пользователем и программой.

Различают два типа диалога: управляемые программой и управляемые пользователем.

Диалог, управляемый программой, предусматривает наличие жесткого сценария диалога, заложенного в программу. Диалог, управляемый пользователем, предусматривает, что сценарий диалога зависит от пользователя.

Различают три формы диалога:

- фразовую;
- директивную;
- табличную.

Фразовая форма предполагает общение с пользователем на естественном языке. Реализация такой формы крайне сложна. Поэтому чаще всего используют диалоги, предполагающие односложные ответы.

Пример

Вопрос программы: *Введите возраст*
Ответ пользователя: *40*

Недостатки фразовой формы — это отсутствие гарантий правильности ответа и значительная ресурсозатратность, связанная со сложностью реализации.

Директивная форма предполагает использование команд (директив) специально разработанного формального языка. Команду можно вводить:

- в виде строки текста в командной строке (например, команды MS-DOS);
- нажатием некоторой комбинации клавиш (например, акселераторы);
- посредством манипулирования мышью (например, перетаскивание пиктограмм).

Основными достоинствами директивной формы являются:

- небольшой объем вводимой информации;
- гибкость;
- использование ограниченного набора команд, обеспечивающее предсказуемый результат;
- ориентация на диалог, управляемый пользователем;
- использование минимальной области экрана или отсутствие экранного вывода.

К недостаткам директивной формы относятся:

• практическое отсутствие подсказок на экране, что требует запоминания;

• отсутствие обратной связи.

Директивная форма удобна для пользователя-профессионала.

Табличная форма предполагает, что пользователь выбирает ответ из списка, предложенного программой. Достоинствами табличной формы являются:

• наличие подсказки (направленность не на запоминание, а на узнавание);

• сокращение количества ошибок ввода: пользователь не вводит информацию, а указывает на нее;

• сокращение времени обучения пользователя.

К недостаткам табличной формы относятся:

• необходимость наличия навыков навигации по экрану;

• использование большой площади экрана для изображения визуальных компонентов;

• интенсивное использование ресурсов компьютера, связанное с необходимостью постоянного обновления информации на экране.

3.3. Основные компоненты интерфейсов

Графические пользовательские интерфейсы поддерживаются практически всеми современными операционными системами общего назначения: Windows, Linux, macOS и др. В рамках указанных операционных систем для таких интерфейсов разработаны наборы стандартных компонентов взаимодействия с пользователем.

Пользовательские интерфейсы современных программ строятся по технологии WIMP: Windows (окна), Icons (пиктограммы), Mouse (мышь), Pop-up (всплывающие или выпадающие меню).

Таким образом, основными элементами графических интерфейсов являются окна, пиктограммы, компоненты ввода-вывода, мышь, которую используют в качестве указывающего устройства и устройства прямого манипулирования объектами на экране.

Окно — это ограниченная рамкой область экрана, которая может менять размеры и местоположение в пределах экрана. Все окна можно разделить на пять категорий:

• основные окна (окна приложения);

• дочерние окна;

• окна диалога;

• информационные окна;

• окна меню.

Окно приложения обычно содержит: рамку, строку заголовка с кнопкой системного меню и кнопками выбора представления окна и выхода, строку меню, пиктографическое меню (панель инструментов), горизонтальную и вертикальную полосы прокрутки, строку состояния.

Дочернее окно используют в многодокументных программных интерфейсах (MDI). В отличие от окна приложения, дочернее окно не содержит меню.

Диалоговое окно используют для просмотра и задания различных режимов работы и необходимых параметров. Оно может содержать компоненты, обеспечивающие пользователю возможность ввода-вывода информации.

Информационные окна бывают двух типов: окна сообщений и окна помощи.

Окна меню можно использовать как открывающиеся панели иерархического меню или как отдельные контекстные меню.

Пиктограмма — это небольшое окно с графическим изображением. Различают следующие пиктограммы:

- программные пиктограммы (свернутое в пиктограмму окно приложения);
- пиктограммы дочерних окон;
- пиктограммы панели инструментов (дублируют команды меню для быстрого вызова);
- пиктограммы объектов (используются для прямого манипулирования объектами).

Прямое манипулирование объектами — это возможность замены команды воздействия на некоторый объект физическим действием в интерфейсе, осуществляемым с помощью мыши. При этом любая область экрана рассматривается как адресат, который может быть активизирован при подведении курсора и нажатии клавиши мыши.

По реакции на воздействие различают следующие типы адресатов:

- указание и выбор (развертывание пиктограммы, определение активного окна);
- буксировка и «резиновая нить» (перенос объекта и его границ);
- экранные кнопки.

Для реализации диалогов, управляемых пользователем, применяют меню различных видов.

Для реализации диалогов, управляемых системой, обычно используют диалоговые окна.

Сделано множество попыток создания социализированного пользовательского (интеллектуального) интерфейса. В основе такого интерфейса лежит идея создания персонифицированного интерфейса.

К интеллектуальным элементам пользовательских интерфейсов относятся Мастер, Советчик, Агент.

Программу-мастер используют для выполнения общераспространенных, но редко выполняемых отдельным пользователем задач. Выполнение подобных действий требует от пользователя принятия сложных взаимосвязанных решений, последовательность которых диктует программа-мастер. Интеллектуальные Мастера способны на каждом шаге демонстрировать в окне просмотра результаты ответов пользователя на предыдущие вопросы, помогая последнему сориентироваться в ситуации.

Мастер реализует последовательный или древовидный сценарий диалога. Его целесообразно использовать для решения хорошо структурированных, последовательных задач.

Советчики представляют собой форму подсказки. Их можно вызвать с помощью меню справки, командной строки окна или из всплывающего меню. Советчики помогают пользователям в выполнении конкретных задач.

Программные агенты используются для выполнения рутинной работы. Различают:

- программы-агенты, настраиваемые на выполнение указанных задач;
- программы-агенты, способные обучаться, фиксируя действия пользователя.

Контрольные вопросы

1. Каковы основные типы пользовательских интерфейсов?
2. В чем преимущество интерфейса со свободной навигацией по сравнению с интерфейс-меню?
3. Какие интерфейсы называются графическими?
4. Какие интерфейсы используются при объектно-ориентированном подходе к программированию?
5. Что такое диалог?
6. Какие типы диалога вы знаете?
7. Какие формы диалога вы знаете?
8. Каковы основные компоненты графических пользовательских интерфейсов?
9. Какие виды пиктограмм вы знаете?
10. Какие элементы пользовательских интерфейсов относятся к интеллектуальным?

Часть 2

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

В ч. 2 рассматриваются основы программирования на языках программирования высокого уровня Python и C.

В результате изучения раздела обучающиеся должны:

знать

• структуру программы;

• основные типы данных, их особенности и их особенности их обработки в языках программирования;

уметь

• реализовывать стандартные алгоритмические структуры для решения задач;

• выполнять стандартные операции над данными различного типа;

владеть

• основными принципами объектно-ориентированного и структурного подхода при создании программ;

• навыками написания программ на языках высокого уровня.

Глава 4
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ PYTHON

В результате изучения гл. 4 обучающиеся должны:

знать

- структуру программы;
- основные типы данных, их особенности;
- стандартные модули языка;

уметь

- выполнять стандартные операции над данными различного типа;
- применять стандартные алгоритмические структуры для обработки данных;

владеть

- основными принципами объектно-ориентированного подхода при создании программ;
- спецификой работы с переменными различных типов данных.

4.1. Знакомство с языком программирования Python

Python — высокоуровневый язык программирования общего назначения с динамической типизацией, автоматическим управлением памятью, поддержкой многопоточных вычислений и удобными структурами данных. На рис. 4.1 перечислены области применения языка программирования Python.

Прежде чем переходить к выполнению программ на языке Python, рассмотрим, как запускаются программы на компьютере (рис. 4.2). Выполнение программ осуществляется операционной системой (Microsoft Windows, GNU/Linux и пр.). В задачи операционной системы входит выделение ресурсов (оперативной памяти и пр.) для программы, запрет или разрешение на доступ к устройствам ввода/вывода и т.д.

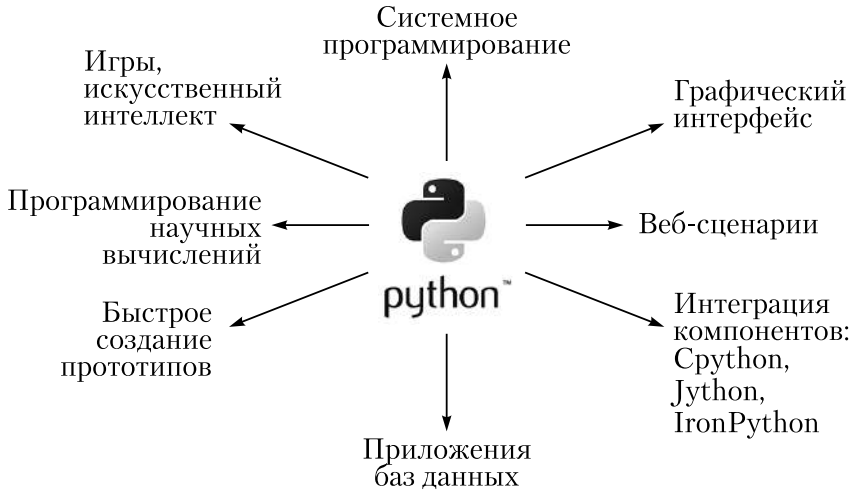


Рис. 4.1. **Области применения языка программирования Python**

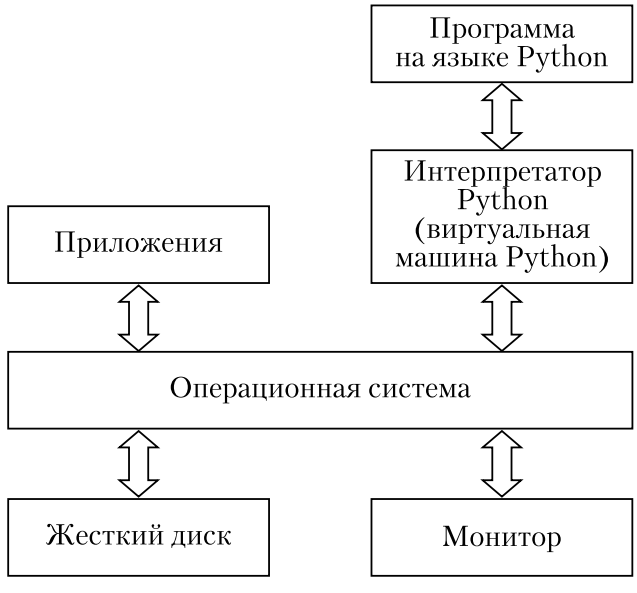


Рис. 4.2. **Схема запуска программ**

Для запуска программ, написанных на языке программирования Python, необходима программа-интерпретатор[1] (*виртуальная машина*) Python. Данная программа скрывает от Python-программиста все особенности операционной системы, поэтому, написав программу на Python в системе Windows, ее можно запустить, например, в GNU/Linux и получить схожий результат.

Скачать и установить интерпретатор Python[2] можно совершенно бесплатно с официального сайта http://python.org. Для работы

[1] Python является интерпретируемым языком программирования (команды выполняются шаг за шагом), в отличие от компилируемых, где текст программы переводится в эквивалентный машинный код.

[2] Процесс установки зависит от операционной системы.

нам понадобится интерпретатор Python версии 3.6 или выше. В процессе установки рекомендуется указать путь **C:/Python36-32**.

После установки программы-интерпретатора запустите интерактивную графическую среду IDLE[1] и дождитесь появления приглашения для ввода команд:

```
Type "copyright", "credits" or "license()" for more
information.
>>>
```

4.2. Интеллектуальный калькулятор

В самом начале обучения Python можно рассматривать как интерактивный интеллектуальный калькулятор. В интерактивном режиме IDLE найдем значения математических выражений. После завершения набора выражения нажмите клавишу <*Enter*> для завершения ввода и последующего выполнения указанных выражений:

```
>>> 3.0 + 6
9.0
>>> 4 + 9
13
>>> 1 - 5
-4
>>> _ + 6
2
```

Нижним подчеркиванием в предыдущем примере обозначается последний полученный результат.

Если совершить ошибку при вводе команды, то интерпретатор Python сообщит об этом:

```
>>> a
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

В математических выражениях в качестве операндов могут использоваться как *целые числа*[2] (1, 4, –5), так и *вещественные*[3] (в про-

---

[1] Выбор среды IDLE обусловлен тем, что она входит в стандартную поставку интерпретатора Python и является свободно распространяемой.

[2] А также комплексные числа и логические значения (**True**, **False**).

[3] Объяснение правил хранения вещественных чисел в компьютере выходит за рамки учебника, сравните в следующем примере результаты вычислений:

```
>>> 2/3 + 1
1.6666666666666665
>>> 5/3
1.6666666666666667
>>>
```

граммировании их еще называют *числами с плавающей точкой*): 4.111, –9.3. Математические операции, доступные над числами в Python[1], представлены в табл. 4.1.

Таблица 4.1

Математические операторы в Python

Оператор	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление (в результате вещественное число)
//	Деление с округлением вниз
**	Возведение в степень
%	Остаток от деления

```
>>> 5/3
1.6666666666666667
>>> 5 // 3
1
>>> 5 % 3
2
>>> 5 ** 67
677626357803440271254658000543713569641113281 25
```

Если один из операндов является вещественным числом, то в результате вычислений получится вещественное число.

При вычислении математических выражений Python придерживается приоритета операций (следует математическим соглашениям):

```
>>> -2 ** 4
-16
>>> -(2**4)
-16
>>> (-2) ** 4
16
```

В случае сомнений в порядке применения математических операторов и для упрощения чтения выражений будет полезным обозначить приоритет в виде круглых скобок.

Выражаясь в терминах программирования, только что мы познакомились с *числовым типом данных* (целочисленным типом **int** и вещественным типом **float**), т.е. множеством числовых значений и множеством математических операций, которые можно выполнять над данными значениям.

[1] Любопытно, что в Python выражение **(b * (a // b) + a % b)** эквивалентно **a**.

Язык программирования Python предоставляет большой выбор встроенных типов данных, о которых речь пойдет дальше.

4.3. Переменные

Рассмотрим выражение **y = x + 3 * 6**, где **y** и **x** являются *переменными*, которые могут содержать некоторые значения. На языке Python вычислить значение **y** при **x** равном 1 можно следующим образом:

```
>>> x = 1
>>> y = x + 3 * 6
>>> y
19
```

В выражении нельзя использовать переменную, если ранее ей не было присвоено значение с помощью *инструкции присваивания*. Для Python такие переменные не определены, и их использование приведет к ошибке.

Содержимое переменной **y** можно вывести на экран, если в интерактивном режиме ввести ее имя.

Имена переменным задает программист, но есть несколько ограничений, связанных с их наименованием. Имена переменных нельзя начинать с цифры и в качестве имен переменных нельзя использовать ключевые слова, которые для Python имеют определенный смысл (эти слова подсвечиваются в IDLE оранжевым цветом, табл. 4.2).

Таблица 4.2

Ключевые слова Python

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None	—	—

Далее мы часто будем обращаться к формуле перевода из шкалы в градусах по Цельсию (T_C) в шкалу в градусах по Фаренгейту (T_F):

```
TF = 9/5 * TC + 32
```

Определим значение T_F при T_C, равном 26. Создадим переменную с именем **cel**, содержащую значение целочисленного типа 26:

```
>>> cel = 26
>>> cel
26
>>> 9/5 * cel + 32
78.80000000000001
```

В момент выполнения инструкции присваивания **cel** = 26 в памяти компьютера создается *объект* (рис. 4.3), расположенный по некоторому *адресу* (условно обозначим его как *id*1), имеющий значение 26 целочисленного типа **int**. Затем создается переменная с именем **cel**, которой *присваивается адрес* объекта *id*1.

Таким образом, переменные в Python содержат адреса объектов. Иначе можно сказать, что *переменные ссылаются на объекты*. Тип переменной определяется типом объекта, на который она ссылается. В дальнейшем для упрощения будем говорить, что переменная хранит значение.

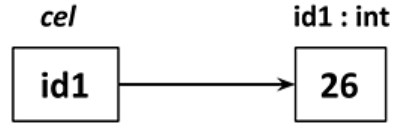


Рис. 4.3. **Модель памяти для выражения cel = 26**[1]

Вычисление следующего выражения приведет к присваиванию переменной **cel** значения 72, т.е. сначала вычисляется правая часть, затем результат присваивается левой части:

```
>>> cel = 26 + 46
>>> cel
72
```

Рассмотрим более сложный пример:

```
>>> diff = 20
>>> double = 2 * diff
>>> double
40
```

Во втором выражении в первую очередь произойдет вычисление правой части, где на место переменной **diff** подставится значение 20, и результат вычисления присвоится переменной **double**. По окончании вычислений память будет иметь вид, представленный на рис. 4.4.

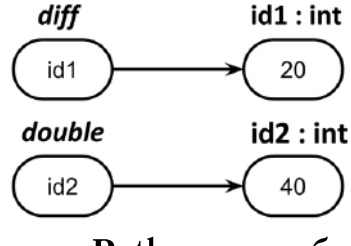


Рис. 4.4. **Схема памяти Python при работе с переменными**

[1] Рисунки к гл. 4 даны по изданию: *Федоров Д. Ю.* Основы программирования на примере языка Python : учеб. пособие. СПб., 2016.

Далее присвоим переменной **diff** значение 5 и выведем на экран содержимое переменных **double** и **diff**:

```
>>> diff = 5
>>> double
40
>>> diff
5
```

В момент присваивания переменной **diff** значения 5 в памяти (рис. 4.5) создастся объект по адресу *id*3, содержащий целочисленное значение 5. После этого изменится содержимое переменной **diff**, вместо адреса *id*1 туда запишется адрес *id*3. Также Python увидит, что на объект по адресу *id*1 больше никто не ссылается и поэтому удалит его из памяти (произведет автоматическую *сборку мусора*).

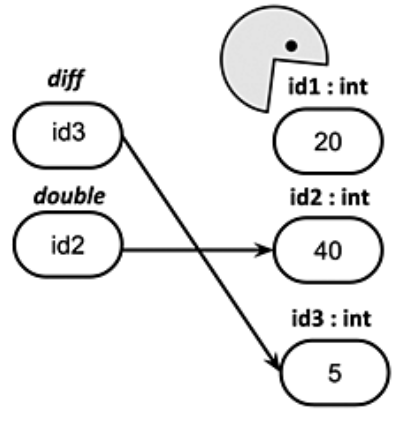


Рис. 4.5. **Схема памяти Python при работе с переменными**

Внимательный читатель заметил, что Python не изменяет существующие числовые объекты, а создает новые, т.е. объекты числового типа данных в Python являются *неизменяемыми*.

У начинающих программистов часто возникает недоумение при виде следующих выражений:

```
>>> num = 20
>>> num = num * 3
>>> num
60
```

Если вспомнить, что сначала вычисляется правая часть выражения, то все станет на свои места.

4.4. Функции

Функцией в программировании называется последовательность инструкций, которая выполняет вычисления. С чем можно сравнить функцию? Напрашивается аналогия с «черным ящиком», когда мы

знаем, что поступает на вход и что при этом получается на выходе, а внутренности «черного ящика» от нас скрыты. В качестве примера можно привести банкомат. На вход банкомата поступает пластиковая карточка (пин-код, денежная сумма), на выходе мы ожидаем получить запрашиваемую сумму. Нас не очень сильно интересует принцип работы банкомата до тех пор, пока он работает без сбоев.

Рассмотрим встроенную функцию с именем **abs**, принимающую на вход один аргумент — объект числового типа, и возвращающую абсолютное значение для этого объекта (рис. 4.6).

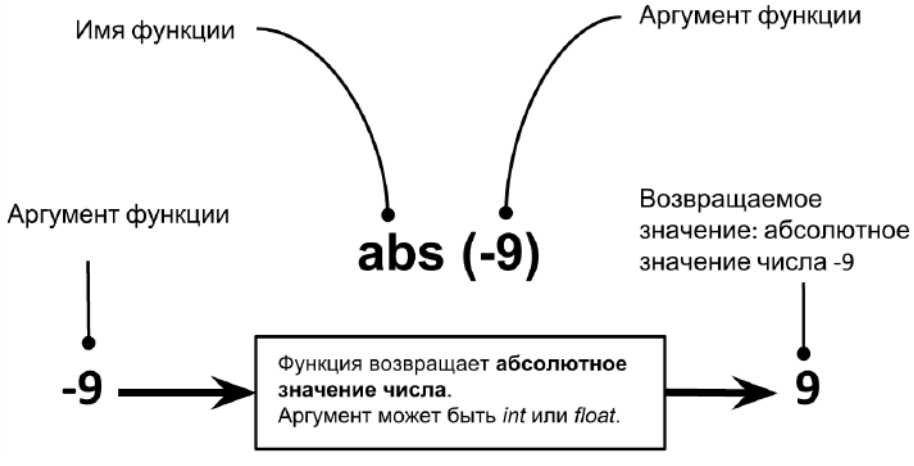


Рис. 4.6. **Функция как «черный ящик»**

Пример вызова функции **abs** с *аргументом* –9 имеет вид:

```
>>> abs(-9)
9
>>> d = 1
>>> n = 3
>>> abs(d - n)
2
>>> abs(-9) + abs(5.6)
14.6
```

Результат вызова функции можно присвоить переменной, использовать его в качестве операндов математических выражений, что позволяет формировать более сложные выражения.

Рассмотрим примеры нескольких встроенных математических функций.

Функция **pow(x, y)** возвращает значение **x** в степени **y**. Эквивалентно записи **x**y**, с которой мы уже встречались.

```
>>> pow(4, 5)
1024
```

Функция **pow** может принимать третий аргумент, тогда вызов функции с аргументами **pow(x, y, z)** эквивалентен вычислению выражения **(x ** y) % z**:

```
>>> pow(4, 5, 3)
1
```

Функция **round(number)** возвращает число с плавающей точкой, округленное до 0 цифр после запятой (по умолчанию). Функция может быть вызвана с двумя аргументами: **round(number [, ndigits])**, где **ndigits** — число знаков после запятой:

```
>>> round(4.56666)
5
>>> round(4.56666, 3)
4.567
```

Помимо составления сложных математических выражений Python позволяет передавать результат вызова функции в качестве аргументов других функций без использования дополнительных переменных.

На рис. 4.7 представлен пример вызова функций и порядок их вычисления. В этом примере на месте числовых объектов –2, 4.3 могут находиться более сложные выражения, поэтому они также нуждаются в вычислении.

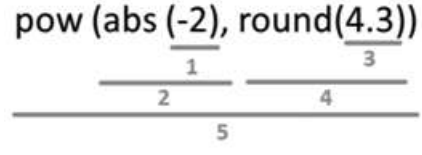


Рис. 4.7. **Порядок вычисления составного выражения**

На практике часто при написании программ требуется преобразовывать типы объектов.

Функция **int** принимает любое значение и преобразует его в целое число, если это возможно (возвращает 0, если аргументы не переданы):

```
>>> int()
0
```

Функция **int** может преобразовать число с плавающей точкой в целое, но это не округление, а отсечение дробной части:

```
>>> int(5.6)
5
```

Функция **float** возвращает число с плавающей точкой, построенное из числа или строки[1] (возвращает 0.0, если аргументы не переданы):

```
>>> float(5)
5.0
>>> float()
0.0
```

[1] О строках речь пойдет ниже.

Описание функций содержится в документации, которая может быть вызвана с помощью функции **help** (на вход подается имя функции):

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Вернемся к формуле перевода градусов по шкале Фаренгейта (T_F) в градусы по шкале Цельсия (T_C):

```
TC = 5/9 * (TF - 32)
```

Произведем несколько вычислений, где переменная **deg_f** будет содержать значение в градусах по Фаренгейту:

```
>>> deg_f = 80
>>> deg_f
80
>>> 5/9 * (deg_f - 32)
26.666
>>> deg_f = 70
>>> 5/9 * (deg_f - 32)
```

Заметим, что каждый раз для перевода приходится набирать одно и то же выражение. Упростим вычисления, создав собственную функцию, переводящую градусы по шкале Фаренгейта в градусы по шкале Цельсия.

В первую очередь необходимо придумать имя функции (рис. 4.8), к примеру, назовем функцию **convert_co_cels**. Постарайтесь, чтобы имя было осмысленным (**lena123** — плохой пример) и отражало смысл функции, вспомните о правилах наименования переменных. Помимо этого, нежелательно, чтобы имя вашей функции совпадало с именами встроенных функций Python (встроенные функции в IDLE подсвечиваются фиолетовым цветом).

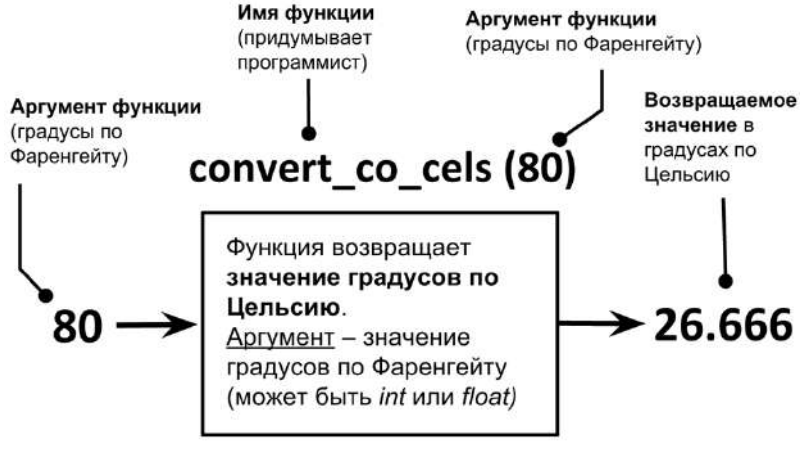


Рис. 4.8. **Создание собственной функции**

Представим, что функция с именем **convert_co_cels** создана, тогда ее вызов для значения (аргумента) 80 будет иметь вид: **convert_co_cels(80)**.

Перейдем непосредственно к созданию функции (рис. 4.9). Ключевое слово **def** означает, что далее следует определение функции. После **def** указывается имя функции **convert_co_cels**, затем в скобках указывается *параметр* (или параметры), которому будет присваиваться значение при вызове функции. Параметры функции — обычные переменные, которыми функция пользуется для внутренних вычислений. Переменные, объявленные внутри функции, называются *локальными* и не видны вне функции. После символа «:» начинается *тело функции* — блок команд, относящийся к функции. Тело функции может содержать любое количество инструкций. В интерактивном режиме Python самостоятельно расставит отступы[1] от края редактора, тем самым обозначив, где начинается тело функции. Выражение, стоящее после инструкции **return**, будет возвращаться в качестве результата вызова функции.

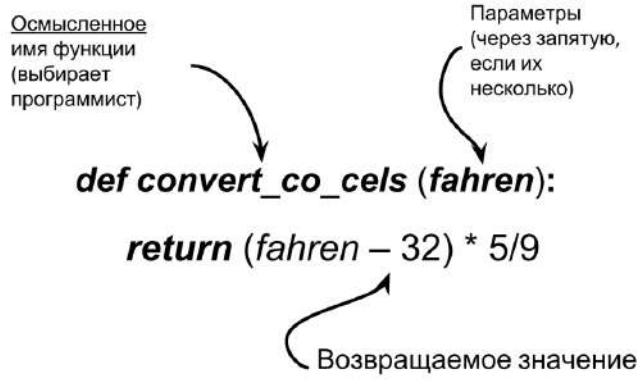


Рис. 4.9. **Схема создания функции в Python**

В интерактивном режиме создание функции имеет следующий вид (для завершения ввода функции необходимо два раза нажать клавишу <*Enter*>, дождавшись приглашения для ввода команд >>>):

```
>>> def convert_co_cels(fahren):
    return (fahren-32) * 5/9

>>> convert_co_cels(451)
232.77777777777777
>>> convert_co_cels(300)
148.88888888888889
```

После создания функции ее можно вызвать, подставив в скобках аргументы, т.е. задав конкретные значения.

---

[1] Отступы играют важную роль в Python, отделяя блок команд тела функции, цикла и пр.

4.5. Программы в отдельном файле

Внимательный читатель заметил, что в интерактивном режиме нельзя вернуться и внести изменения в выражение, которое уже было выполнено, поэтому приходится набирать его повторно.

Многострочные выражения, где легко допустить ошибку при наборе, удобно помещать в отдельные текстовые файлы с расширением **.py**.

В меню IDLE выберите *File → New File*. Появится окно текстового редактора, в котором можно набирать команды на языке программирования Python. Наберем в редакторе следующий код:

```
# это комментарии, и они игнорируются Python
# firstprog.py
a=5
print(a)
print(a+5)
```

В меню редактора выберем *Save As* и сохраним файл в директорию **C:/Python36-32/**, указав произвольное имя, например **firstprog.py**. В ранних версиях IDLE приходилось вручную прописывать расширение файла.

Для выполнения программы в меню редактора IDLE выберем *Run → Run Module* (или нажмем клавишу <*F5*>). Результат работы программы отобразится в интерактивном режиме:

```
========RESTART: C:/Python36-32/firstprog.py ========
5
10
```

Функция **print** в примере отображает содержимое переменных, переданных ей в качестве аргументов. Вспомним, что интерактивный режим позволял нам обходиться без вызова этой функции.

Разберемся теперь, как создавать и вызывать функции, находящиеся в отдельном файле. Создадим файл **myfunc.py**, содержащий следующий код (тело функции в Python принято отделять либо четырьмя пробелами, либо одной табуляцией — придерживайтесь в рамках файла одного из этих правил):

```
# myfunc.py
def f(x):
    x = 2 * x
    return x
```

Выполним программу с помощью нажатия на клавишу <*F5*>. Увидим, что в интерактивном режиме программа выполнилась, но ничего не вывела на экран:

```
>>>
```

Все правильно, ведь мы не вызвали функцию! После запуска программы в интерактивном режиме вызовем функцию **f** с различными аргументами:

```
>>> f(4)
8
>>> f(56)
112
```

Работает! Теперь вызовем функцию **f** в файле, но не забудем про **print**. Далее представлена обновленная версия программы:

```
# myfunc2.py
def f(x):
    x = 2 * x
    return x

print(f(4))
print(f(56))
```

Выполним программу и увидим, что в интерактивном режиме отобразился результат:

```
8
112
```

4.6. Область видимости переменных

Ранее мы отметили, что переменная является *локальной* (видна только внутри функции), если значение ей присваивается внутри функции, в ином случае — переменная *глобальная*, т.е. видна (к ней можно обратиться) во всей программе, в том числе и внутри функции.

В отдельный файл с именем **var_prog.py** поместим следующий код:

```
# var_prog.py
a = 3 # глобальная переменная
print('глобальная переменная a = ', a)
y = 8 # глобальная переменная
print('глобальная переменная y = ', y)

def func():
    print('func: глобальная переменная a = ', a)
    y = 5 # локальная переменная
    print('func: локальная переменная y = ', y)

func() # вызываем функцию func
print('??? y = ', y) # отобразится глобальная переменная
```

Обратим внимание, что у функции **print** может быть несколько аргументов, заданных через запятую. В одиночные апострофы по-

мещается строка[1]. После выполнения программы получим следующий результат:

```
глобальная переменная a = 3
глобальная переменная y = 8
func: глобальная переменная a = 3
func: локальная переменная y = 5
??? y = 8
```

Внутри функции мы смогли обратиться к глобальной переменной **a** и вывести ее значение на экран. Затем внутри функции создается локальная переменная **y**, причем ее имя совпадает с именем глобальной переменной — в этом случае при обращении к **y** выводится содержимое локальной переменной, а глобальная переменная остается неизменной.

Как быть, если мы хотим изменить содержимое глобальной переменной внутри функции? В следующем примере демонстрируется подобное изменение с использованием ключевого слова **global**:

```
# var_prog2.py
x = 50        # глобальная переменная
def func():
    global x # указываем, что x глобальная переменная
    print('x равно', x)
    x = 2     # изменяем глобальную переменную
print('Заменяем глобальное значение x на', x)

func()
print('Значение x составляет', x)
```

Результат работы программы **var_prog2.py**:

```
x равно 50
Заменяем глобальное значение x на 2
Значение x составляет 2
```

4.7. Применение функций

На практике часто функции используются для сокращения исходного кода программы, например, если объявить функцию вида:

```
def func(x):
    c = 7
    return x + 8 + c
```

то код на рис. 4.10 может быть заменен тремя вызовами функции с различными аргументами.

Бывают случаи, когда функция ничего не принимает на вход и ничего не возвращает[2] (явно не используется инструкция **return**).

---

[1] Можно было бы использовать двойные кавычки. О строках речь пойдет ниже.

[2] На самом деле, если не указать инструкцию return, то Python все равно вернет объект None.

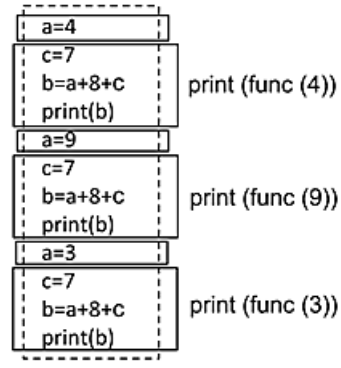


Рис. 4.10. **Пример многократного вызова функции**

Пример подобной функции:

```
def print_hello():
    print('Привет')
    print('Hello')
    print('Hi')
```

Видим, что внутри функции **print_hello** происходит вызов **print**, поэтому в момент вызова **print_hello** еще раз вызывать **print** не требуется. Пример на рис. 4.11 демонстрирует, что множество повторяющихся вызовов **print** можно заменить несколькими вызовами функции **print_hello**.

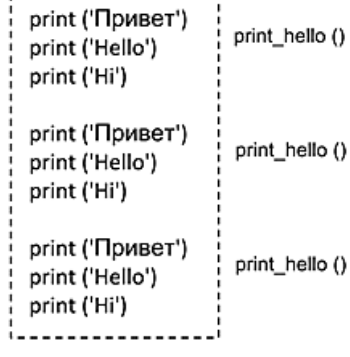


Рис. 4.11. **Пример использования вызова функций**

Имена функций в Python являются обычными переменными, содержащими адрес объекта[1] типа функция[2], поэтому этот адрес можно присвоить другой переменной и вызвать функцию уже с другим именем:

```
# call_func.py
def summa(x, y):
    return x + y
f = summa
v = f(10, 3) # вызываем функцию с другим именем
```

[1] В Python все является объектами.

[2] Да-да, это еще один тип данных в Python.

Параметры функции могут принимать значения по умолчанию:

```
# func_var.py
def summa(x, y = 2):
    return x + y
a = summa(3) # вместо y подставляется значение
             # по умолчанию
b = summa(10, 40) # теперь значение второго параметра
                  # равно 40
```

В связи с тем, что имя функции является обычной переменной, можно передать ее в качестве аргумента при вызове функции:

```
# call_summa.py
def summa(x, y):
    return x + y
def func(f, a, b):
    return f(a, b)
v = func(summa, 10, 3) # передаем summa в качестве
                       # аргумента
```

Этот пример показывает, как из функции **func** можно вызвать функцию **summa**.

Помимо этого, в момент вызова функции можно присваивать значения конкретным параметрам (использовать *ключевые аргументы*):

```
# key_arg.py
def func(a, b=5, c=10):
    print('a равно', a, ', b равно', b, ', a c равно', c)

func(3, 7) # a=3, b=7, c=10
func(25, c=24) # a=25, b=5, c=24
func(c=50, a=100) # a=100, b=5, c=50
```

Ошибкой будет являться вызов функции, при котором не задан аргумент **a**, так как для него не указано значение по умолчанию.

4.8. Строки и операции над строками

Для работы с текстом в Python предусмотрен специальный строковый тип данных **str**. Строковые объекты создаются, если текст поместить в одиночные апострофы или двойные кавычки:

```
>>> 'hello'
'hello'
>>> "Hello"
'Hello'
```

Без кавычек Python расценит текст как имя переменной и попытается вывести на экран ее содержимое, если такая переменная существует:

```
>>> hello
Traceback (most recent call last):
```

```
  File "<pyshell#2>", line 1, in <module>
    hello
NameError: name 'hello' is not defined
```

Можно создать пустую строку:

```
>>> ''
''
```

Для работы со строками в Python предусмотрено большое число встроенных функций. Рассмотрим, например, функцию **len**, которая определяет длину строки, переданной ей в качестве аргумента:

```
>>> help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Пример вызова функции **len** для строкового аргумента:

```
>>> len('Привет!')
7
```

4.9. Операции над строками

С помощью операции *конкатенации* (оператор «+» для строк) Python позволяет объединить несколько строк в одну (также допускается расположить строки последовательно без каких-либо операторов):

```
>>> 'Привет, ' + 'земляне!'
'Привет, земляне!'
>>> 'Привет, ' 'земляне!'
'Привет, земляне!'
```

Здесь начинаются удивительные вещи! Помните, мы говорили, что операции зависят от типа данных? Над объектами определенного типа можно производить только определенные операции: числа — складывать, умножать и т.д., т.е. производить над ними арифметические операции. Так вот, для строковых объектов операция сложения объединяет строки, а для числовых — складывает. Что произойдет, если применить оператор сложения одновременно к числу и строке?

```
>>> 'Марс' + 5
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    'Марс' + 5
TypeError: must be str, not int
```

Python не разобрался, что мы от него хотим: сложить числа или объединить строки. К примеру, чтобы объединить строки, можно с помощью функции **str** преобразовать число 5 в строку '5' и выполнить объединение:

```
>>> 'Марс' + str(5)
'Марс5'
```

Можно произвести обратное преобразование типов (из строки в число):

```
>>> int("-5")
-5
```

Попросим Python повторить (размножить) строку 10 раз:

```
>>> "СПАМ" * 10
'СПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМ'
```

Отметим, что оператор умножения для строковых объектов приобрел новый смысл.

Строки, по аналогии с числами, можно присваивать[1] переменным:

```
>>> s = "Я изучаю программирование"
>>> s
'Я изучаю программирование'
>>> s*4
'Я изучаю программированиеЯ изучаю программированиеЯ
изучаю программированиеЯ изучаю программирование'
>>> s + " на языке Python"
'Я изучаю программирование на языке Python'
```

Поместить разные виды кавычек в строку можно несколькими способами:

```
>>> "Hello's"
"Hello's"
>>> 'Hello\'s'
"Hello's"
```

Первый способ — заключить строку в кавычки разных типов, чтобы указать Python, где заканчивается строка. Второй — использовать специальные символы (*управляющие escape-последовательности*), которые записываются как два символа, но Python видит их как один:

```
>>> len("\'")
1
```

Полезно помнить часто встречающиеся управляющие последовательности:

```
\n - переход на новую строку
\t - знак табуляции
\\ - наклонная черта влево
\' - символ одиночной кавычки
\" - символ двойной кавычки
```

[1] Напоминаем, что в переменной хранится адрес объекта (в данном случае строкового объекта).

При попытке разбить длинную строку с помощью <*Enter*> возникает ошибка:

```
>>> 'Это длинная
SyntaxError: EOL while scanning string literal
>>> строка
Traceback (most recent call last):
File "<pyshell#20>", line 1, in <module>
строка
NameError: name 'строка' is not defined
```

Для создания многострочной строки ее необходимо заключить с обеих сторон в три одиночных апострофа:

```
>>> '''Это длинная
строка'''
'Это длинная\nстрока'
```

Отметим, что при выводе на экран многострочной строки перенос строки отобразился в виде специального символа '**\n**'.

4.10. Дополнительные возможности функции print

Вернемся к уже встречавшейся нам функции **print**. Передадим на вход этой функции строку со специальным символом:

```
>>> print('Это длинная\nстрока')
Это длинная
строка
```

Функция **print** смогла распознать специальный символ и отобразить на экране перевод строки. Рассмотрим еще несколько примеров:

```
>>> print(1, 3, 5)
1 3 5
>>> print(1, '2', 'снова строка', 56)
1 2 снова строка 56
```

Таким образом, функция **print** позволяет выводить объекты разных типов.

На самом деле у этой функции есть несколько «скрытых» аргументов, которые задаются по умолчанию в момент вызова (рис. 4.12).

Рассмотрим пример:

```
>>> print(1, 6, 7, 8, 9)
1 6 7 8 9
```

По умолчанию функция **print** для вывода набора объектов использует в качестве разделителя пробел. Для задания другого разделителя можно изменить значение параметра **sep=':'** в момент вызова функции:

```
>>> print(1, 6, 7, 8, 9, sep=':')
1:6:7:8:9
```

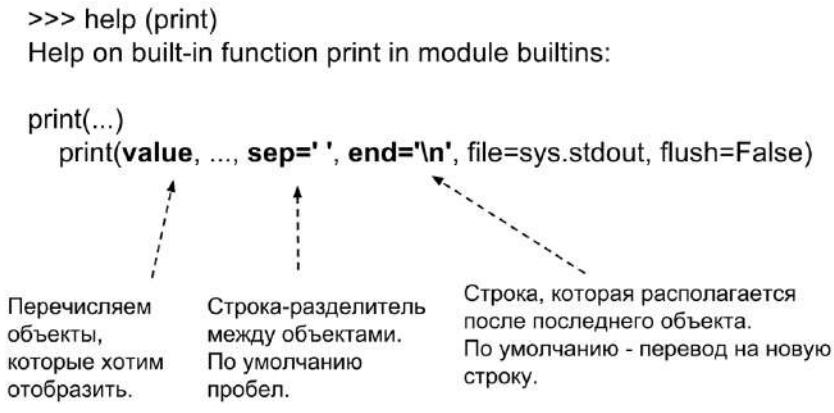


Рис. 4.12. **«Скрытые» параметры функции print**

4.11. Ввод значений с клавиатуры

Усложним программу и попросим пользователя ввести что-нибудь с клавиатуры. В Python для этого предусмотрена специальная функция **input**:

```
>>> s = input()
Земляне, мы прилетели с миром!
>>> s
'Земляне, мы прилетели с миром!'
>>> type(s)
<class 'str'>
```

В примере мы вызвали функцию **input** и результат ее работы присвоили переменной **s**. После этого пользователь ввел значение с клавиатуры и нажал <*Enter*>, т.е. указал на завершение ввода. Содержимое переменной **s** вывели на экран — отобразилась фраза, которую ввел пользователь. Затем вызвали функцию **type**, которая позволяет определить тип объекта, ссылка на который содержится в переменной **s**.

Внимание: функция **input** возвращает строковый объект!

Работа с функцией **input** может привести к неожиданным ошибкам:

```
>>> s = input("Введите число: ")
Введите число: 555
>>> s + 5
Traceback (most recent call last):
File "<pyshell#33>", line 1, in <module>
s + 5
TypeError: must be str, not int
```

В этом примере используется входной аргумент функции **input**, который выводит на экран строку-приглашение перед пользовательским вводом: *"Введите число: "*. После ввода значения с клави-

атуры пытаемся сложить его с числом 5 и вместо ожидаемого результата получаем сообщение об ошибке. Исправить ошибку позволяет преобразование типов (вызов функции **int**):

```
>>> s = int(input("Введите число: "))
Введите число: 555
>>> s + 5
560
```

Каждый символ строки имеет собственный порядковый номер (*индекс*). Нумерация символов начинается с нуля. Python позволяет обратиться к заданному символу строки с помощью оператора скобок следующим образом:

```
>>> s = 'Я люблю писать программы!'
>>> s[0]
'Я'
>>> s[-1]
'!'
```

В квадратных скобках указывается индекс символа. Если нулевой индекс — первая буква строки, что тогда означает индекс –1? Очевидно, что последний символ. Для перевода отрицательных индексов в положительные используется следующая формула: длина строки + отрицательный индекс. Например, для –1: **len(s)**–1, т.е. 24.

```
>>> len(s)-1
24
>>> s[24]
'!'
```

Попробуем изменить первый символ строки (с нулевым индексом):

```
>>> s = 'Я люблю писать программы!'
>>> s[0] = 'J'
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    s[0] = 'J'
TypeError: 'str' object does not support item assignment
```

Попытка изменить символ в строке **s** привела к ошибке. Дело в том, что в Python строки, как и числа, являются неизменяемыми объектами.

Работа со строковыми объектами не отличается от работы с числовыми объектами (рис. 4.13).

В момент изменения значения переменной **s** (рис. 4.14) будет создан новый строковый объект по адресу *id*2, и этот адрес будет записан в переменную **s**.

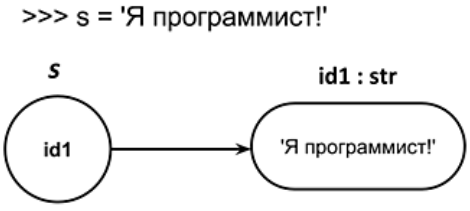


Рис. 4.13. **Схема присвоения переменной строкового объекта**

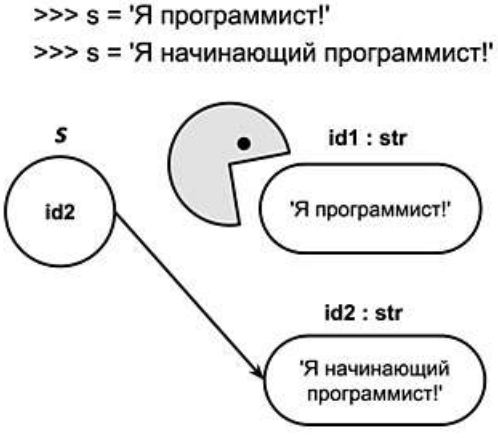


Рис. 4.14. **Схема присвоения переменной нового строкового объекта**

Прежде чем мы поймем, как можно изменить строку, познакомимся со *срезами*:

```
>>> s = 'Питоны водятся в Африке'
>>> s[1:3]
'ит'
```

s[1:3] — срез (часть) строки **s**, начиная с индекса 1, заканчивая индексом 3 (не включительно). Это легко понять, если индексы представить в виде смещений (рис. 4.15).

Рис. 4.15. **Индексы символов в строке как смещения**

Со срезами можно производить различные манипуляции:

```
>>> s[:3] # с 0 индекса по третий не включительно
'Пит'
>>> s[:] # вся строка
'Питоны водятся в Африке'
>>> s[::2] # третий аргумент задает шаг (по умолчанию один)
'Птн ояс фие'
>>> s[::-1] # «обратный» шаг
'екирфА в ястядов ынотиП'
>>> s[:-1] # вспомним, как мы находили отрицательный
           # индекс
```

```
'Питоны водятся в Африк'
>>> s[-1:] # снова отрицательный индекс
'е'
```

Теперь, используя срезы, изменим первый символ в строке, создав новую строку:

```
>>> s = 'Я люблю писать программы!'
>>> 'J' + s[1:] # формируется новая строка
'J люблю писать программы!'
```

4.12. Логические выражения

В Python для сравнения чисел предусмотрено несколько операторов сравнения:

- > — больше;
- < — меньше;
- >= — больше или равно;
- <= — меньше или равно;
- == — равно;
- != — не равно.

В интерактивном режиме произведем сравнение двух чисел:

```
>>> 6 > 5
True
>>> 7 < 1
False
>>> 7 == 7 # не путайте == и = (присвоить)
True
>>> 7 != 7
False
```

Python возвращает **True**[1] (истину[2]), когда сравнение верное, и **False** (ложь) — в ином случае. **True** и **False** относятся к *логическому* (*булевому*) типу данных **bool**:

```
>>> type(True)
<class 'bool'>
```

Рассмотрим более сложные логические выражения.

Логическим высказыванием (*предикатом*)[3] будем называть любое повествовательное предложение, в отношении которого можно однозначно сказать, истинно оно или ложно.

Например, высказывание «6 — четное число». Истинно или ложно? Очевидно, что истинно. А высказывание «6 больше 19»? Высказывание ложно.

---

[1] Обязательно с большой буквы.

[2] **True** интерпретируется Python как число **1**, а **False** как число **0**.

[3] Информатика. Теория (с задачами и решениями). Интернет-версия издания: *Шауцукова Л. З.* Информатика 10—11. М. : Просвещение, 2000 (режим доступа: http://book.kbsu.ru/theory/chapter5/1_5_1.html).

Является ли высказыванием фраза «у него голубые глаза»? Однозначности в этой фразе нет, поэтому она не является высказыванием.

Высказывания можно комбинировать. Высказывания «Петров — врач», «Петров — шахматист» можно объединять с помощью связок И (and), ИЛИ (or).

«Петров — врач И шахматист». Это высказывание истинно, если *оба* высказывания «Петров — врач» И «Петров — шахматист» являются истинными.

«Петров — врач ИЛИ шахматист». Это высказывание истинно, если истинным является *одно* из высказываний «Петров — врач» или «Петров — шахматист».

Рассмотрим пример комбинаций из высказываний на языке Python:

```
>>> 2 > 4
False
>>> 45 > 3
True
>>> 2 > 4 and 45 > 3 # комбинация False and (и) True
                     # вернет False
False
>>> 2 > 4 or 45 > 3  # комбинация False or (или) True
                     # вернет True
True
```

Все, что сказано выше о комбинации логических высказываний, можно объединить и представить в виде таблицы истинности, где 0 — **False**, а 1 — **True** (табл. 4.3).

Таблица 4.3

Истинность комбинаций логических высказываний

X Y	and	or
0 0	0	0
0 1	0	1
1 0	0	1
1 1	1	1

Для Python истинным или ложным может быть не только логическое высказывание, но и объект.

Внимание: в Python любое число, не равное нулю, или непустой объект интерпретируются как истина. Числа, равные нулю, пустые объекты и специальный объект **None**[1] интерпретируются как ложь.

[1] Имеет специальный тип **NoneType**.

Рассмотрим пример:

```
>>> '' and 2 # False and True
''
>>> '' or 2 # False or True
2
```

В примере логический оператор **and** (и) применяется к двум операндам: пустому строковому объекту (ложный) и ненулевому числовому объекту (истинный). В итоге Python вернул нам пустой строковый объект. Затем аналогично применяется логический оператор **or** (или), в результате чего мы получили числовой объект. Разберемся, откуда взялся такой результат.

Рассмотрим более подробно три логических оператора: **and**, **or**, **not**.

Отрицание **not** (не) — наиболее простой из них, поэтому начнем с него:

```
>>> y = 6 > 8
>>> y
False
>>> not y
True
>>> not None
True
>>> not 2
False
```

Результатом применения логического оператора **not** будет отрицание операнда, т.е. если операнд истинный, то результатом применения оператора **not** станет ложь, и наоборот.

В результате применения логического оператора **and** (и) получим **True** (истину) или **False** (ложь)[1], если его операндами являются логические высказывания:

```
>>> 2 > 4 and 45 > 3 # комбинация False and (и) True
                     # вернет False
False
```

Если операндами оператора **and** являются объекты, то в результате Python вернет объект:

```
>>> '' and 2 # False and True
''
```

Для вычисления результата применения оператора **and** Python вычисляет операнды слева направо и возвращает первый объект, имеющий ложное значение.

Посмотрим на столбец **and** таблицы истинности (см. табл. 4.3) и проследим закономерность. Если среди операндов **X**, **Y** есть лож-

[1] Исходя из таблицы истинности.

ный, то результатом является ложное значение, но вместо ложного значения для операндов-объектов Python возвращает первый ложный операнд, встретившийся в выражении, и дальше вычисления не производит. Это называется *вычислением по короткой схеме*:

```
>>> 0 and 3 # вернет первый ложный объект-операнд
0
>>> 5 and 4 # вернет крайний правый объект-операнд
4
```

Если Python не удается найти ложный объект-операнд, то он возвращает крайний правый операнд.

Логический оператор **or** действует похожим образом, но для объектов-операндов Python возвращает первый объект, имеющий истинное значение. Python прекратит дальнейшие вычисления, как только будет найден первый объект, имеющий истинное значение:

```
>>> 2 or 3 # вернет первый истинный объект-операнд
2
>>> None or 5 # вернет второй объект-операнд,
              # так как первый ложный
5
>>> None or 0 # вернет оставшийся объект-операнд
0
```

Таким образом, конечный результат становится известен еще до вычисления остальной части выражения!

Логические выражения можно комбинировать следующим образом:

```
>>> 1 + 3 > 7 # приоритет + выше, чем >
False
>>> (1 + 3) > 7 # скобки способствуют наглядности
False
>>> 1 + (3 > 7)
1
```

В Python есть возможность проверить принадлежность числа интервалу:

```
>>> x = 0
>>> -5 < x < 10 # эквивалентно: x> -5 and x< 10
True
```

Теперь вы без труда сможете разобраться в работе следующего кода:

```
>>> x = 5 < 10 # True
>>> y = 2 > 3 # False
>>> x or y
True
>>> (x or y) + 6
7
```

Решим небольшую задачку. Как вычислить **1/x**, чтобы не возникало ошибки деления на нуль? Для этого достаточно воспользоваться логическим оператором. Каким? Прямой путь приводит к ошибке:

```
>>> x = 0
>>> 1 / x
Traceback (most recent call last):
  File "<pyshell#88>", line 1, in <module>
    1 / x
ZeroDivisionError: division by zero
```

Решением будет следующий код:

```
>>> x = 1
>>> x and 1/x
1.0
>>> x = 0
>>> x and 1/x
0
```

Строки в Python можно сравнивать аналогично числам. Символы представлены в компьютере в виде чисел. Есть специальная таблица, которая ставит в соответствие каждому символу некоторое число[1]. Определить, какое число соответствует символу, можно с помощью функции **ord**:

```
>>> ord('L')
76
>>> ord('Ф')
1060
>>> ord('A')
65
```

Таким образом, сравнение символов сводится к сравнению чисел, которые им соответствуют:

```
>>> 'A' > 'L'
False
```

Сравнение строк в Python происходит посимвольно:

```
>>> 'Aa' > 'Ll'
False
```

Следующий полезный логический оператор, с которым познакомимся, — **in**. Он принимает две строки и возвращает **True**, если первая строка является подстрокой во второй строке:

```
>>> 'a' in 'abc'
True
>>> 'A' in 'abc' # большой буквы A нет
False
```

[1] Подробнее о Unicode см.: https://docs.python.org/3/howto/unicode.html.

```
>>> "" in 'abc' # пустая строка есть в любой строке
True
>>> '' in ''
True
```

Освоив логические операторы, перейдем к их применению на практике.

4.13. Условная инструкция if

Наиболее часто логические выражения используются внутри условной инструкции **if** (рис. 4.16).

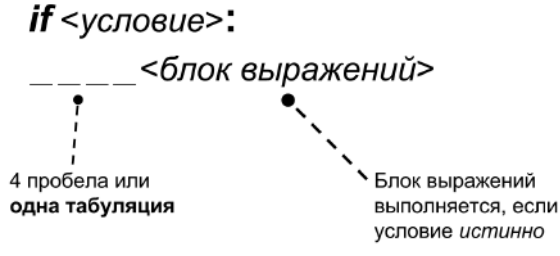


Рис. 4.16. **Схема использования условной инструкции if**

Блок выражений выполняется только в том случае, если выражение, которое находится в условии, является истинным. Для примера обратимся к таблице водородных показателей[1] для различных веществ (табл. 4.4).

Таблица 4.4

Значения pH для различных веществ

Вещество	pH
Яблочный сок	3,0
Кофе	5,0
Шампунь	5,5
Чай	5,5
Питьевая вода	6,5—8,5
Кровь	7,36—7,44
Морская вода	8,0
Мыло (жировое) для рук	9,0—10,0

Произведем проверку:

```
>>> pH = 5.0
>>> if pH == 5.0:
```

[1] См.: Wikipedia, Водородный показатель, http://ru.wikipedia.org/wiki/Водородный_показатель.

```
  print(pH, "Кофе")

5.0 Кофе
```

В примере переменной **pH** присваивается вещественное значение 5.0. Затем значение переменной сравнивается с водородным показателем для кофе (5.0), и, если они совпадают, вызывается функция **print**.

Можно производить несколько проверок подряд:

```
>>> pH = 5.0
>>> if pH == 5.0:
  print(pH, "Кофе")

5.0 Кофе
>>> if pH == 8.0:
  print(pH, "Вода")
```

Ход выполнения данной программы показан на рис. 4.17.

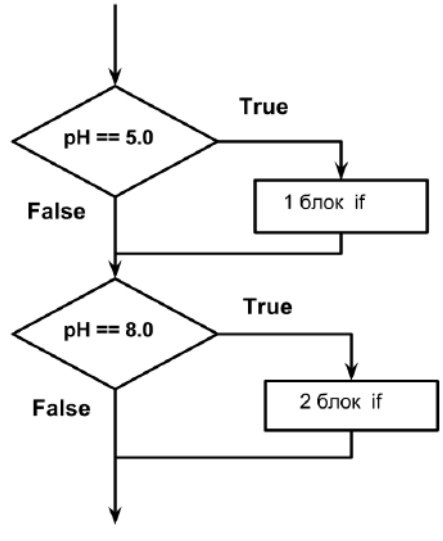


Рис. 4.17. **Ход выполнения программы, содержащей условные инструкции**

На практике встречаются задачи, где выполнять все проверки не имеет смысла. В следующем примере используется инструкция **elif** (сокращение от **else if**):

```
# elif.py
pH = 3.0
if pH == 8.0:
    print(pH, "Вода")
elif 7.36 < pH < 7.44:
    print(pH, "Кровь")
```

Ход выполнения программы показан на рис. 4.18.

Если **pH == 8.0** является ложным, то проверяется **7.36<pH<7.44**, в случае его истинности выполняется блок выражений **elif**. Не су-

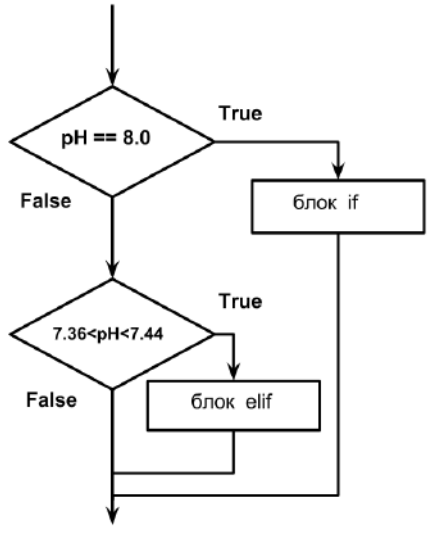


Рис. 4.18. **Ход выполнения программы, содержащей условную инструкцию elif**

ществует ограничений на количество инструкций **elif**. Общий синтаксис представлен ниже.

```
if <условие>:
     <блок выражений>
elif <условие>:
     <блок выражений>
else:
     <блок выражений>
```

Блок выражений, относящийся к **else**, выполняется, когда все вышестоящие условия вернули **False**.

Рассмотрим первую «большую» программу:

```
# big_prog.py
# строку преобразовали к вещественному типу
pH = float(input("Введите pH: "))
if pH == 7.0:
     print(pH, "Вода")
elif 7.36 < pH < 7.44:
     print(pH, "Кровь")
else:
     print("Что это?!")
```

В следующем листинге представлен более сложный пример программы, использующей условные инструкции[1]:

```
# big_prog2.py
value = input("Введите pH: ")
if len(value) > 0: # проверяем, что пользователь хоть
                   # что-нибудь ввел
```

[1] При наборе исходного текста следите за отступами — в Python это чрезвычайно важно.

```
    # переводим в вещественное число ввод пользователя:
    pH = float(value)
    if pH == 7.0: # вложенный if
        print(pH, "Вода")
    elif 7.36 < pH < 7.44:
        print(pH, "Кровь")
    else:
        print("Что это?!")
else:
    print("Введите значение pH!")
```

4.14. Строки документации

В тройные двойные кавычки (""") в теле функции с обеих сторон помещается информация, которую выводит на экран функция справки **help**.

Рассмотрим пример документирования собственных функций. Для этого напишем функцию, которая ничего не будет делать (в теле функции для этого указывается инструкция **pass**). Исходный текст функции представлен в следующем листинге:

```
# mydoc.py
def my_function():
    """ Функция, которая ничего не делает.
    """
    pass
help(my_function) # отображение описания функции
                  # my_function
```

Результат запуска программы:

```
Help on function my_function in module __main__:

my_function()
    Функция, которая ничего не делает.
```

4.15. Модули

Предположим, что мы написали несколько полезных функций, которые часто используем в своих программах. Чтобы к ним быстро обращаться, удобно эти функции поместить в отдельный файл и загружать их оттуда. В Python такие файлы с набором функций называются *модулями*. Чтобы воспользоваться функциями, которые находятся в отдельном модуле, его необходимо *импортировать* с помощью инструкции **import**:

```
>>> import math
```

В приведенном ниже примере в память загружается стандартный модуль math, который содержит набор математических функций. Теперь мы можем обращаться к функциям, находящимся вну-

три этого модуля, следующим образом (например, для нахождения квадратного корня из 9):

```
>>> math.sqrt(9)
3.0
```

При обращении к функции из модуля указывается имя модуля и, через точку, имя функции с аргументами. Узнать о функциях, которые содержит модуль, позволяет функция **help**:

```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available. It provides access
to the mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
...
```

Если необходимо посмотреть описание отдельной функции модуля, то вызываем **help** для нее:

```
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

В момент импортирования модуля **math** создается переменная с именем **math**:

```
>>> type(math)
<class 'module'>
```

Функция **type** показала, что тип данных переменной **math** — модуль (**module**). Переменная **math** содержит ссылку (адрес) модульного объекта. В этом объекте содержатся ссылки на функции. Схема работы с объектом типа модуль представлена на рис. 4.19.

В момент вызова функции **sqrt** Python находит переменную **math** (модуль должен быть предварительно импортирован), просматривает модульный объект, находит функцию **sqrt** внутри этого модуля и затем вызывает ее.

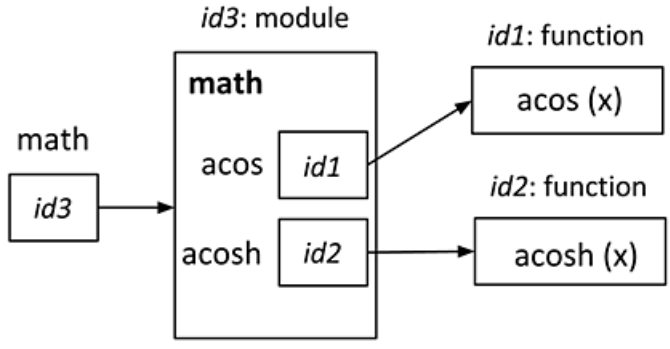


Рис. 4.19. **Схема, представляющая объект типа модуль**

В Python можно импортировать отдельную функцию из модуля:

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

В этом случае переменная **math** создана не будет, а в память загрузится только функция **sqrt**. Теперь вызов функции можно производить, не обращаясь к имени модуля math, но надо быть крайне внимательным. Приведем пример:

```
>>> def sqrt(x):
    return x * x

>>> sqrt(5)
25
>>> from math import sqrt
>>> sqrt(9)
3.0
```

Объявили собственную функцию с именем **sqrt**, затем вызвали ее и убедились, что она работает. После этого импортировали функцию **sqrt** из модуля **math**. Затем снова вызвали **sqrt** и видим, что это не наша функция! Ее подменили!

Следующий пример:

```
>>> def sqrt(x):
  return x * x

>>> sqrt(6)
36
>>> import math
>>> math.sqrt(9)
3.0
>>> sqrt(7)
49
```

Снова объявляем собственную функцию с именем **sqrt** и вызываем ее. Затем импортируем модуль **math** и из него вызываем стандартную функцию **sqrt**. Видим, что теперь вызываются обе функции.

Когда мы только начинали знакомиться с функциями, то использовали функцию **abs** для нахождения модуля числа. На самом деле эта функция тоже находится в модуле, который Python загружает в память в момент начала своей работы. Этот модуль называется **__builtins__** (два нижних подчеркивания до и после имени модуля). Если вызывать справку для данного модуля, то обнаружим огромное количество функций и переменных:

```
>>> help(__builtins__)
Help on built-in module builtins:

NAME
    builtins - Built-in functions, exceptions, and
    other objects.

DESCRIPTION
    Noteworthy: None is the 'nil' object; Ellipsis
    represents '...' in slices.
...
```

Функция **dir** возвращает перечень имен всех функций и переменных, содержащихся в модуле:

```
>>> dir(__builtins__)
...
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
'bytes', 'callable', 'chr', 'classmethod', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict',
'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'map', 'max',
'memoryview', 'min', 'next', 'object', 'oct', 'open',
'ord', 'pow', 'print', 'property', 'quit', 'range',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'vars', 'zip']
```

Часть из этих функций вы знаете, с другими мы познакомимся в следующих главах.

4.16. Создание собственных модулей

Рассмотрим процесс создания собственных модулей. Для этого создадим файл с именем **mm.py** (для модулей *обязательно* указывается расширение **.py**), содержащий следующий исходный код (содержимое будущего модуля):

```
# mm.py
def f():
    return 4
```

Затем сообщим Python, где искать созданный модуль. Выясним через обращение к переменной **path** модуля **sys**, где Python по умолчанию хранит модули (у вас список каталогов может отличаться):

```
>>> import sys
>>> sys.path
['', 'C:\\Python36-32\\Lib\\idlelib', 'C:\\Python36-
32\\python36.zip', 'C:\\Python36-32\\DLLs',
'C:\\Python36-32\\lib', 'C:\\Python36-32',
'C:\\Python36-32\\lib\\site-packages']
```

Поместим созданный модуль в один из перечисленных каталогов, например в **'C:\\Python36-32'**. Если все сделано правильно, то импортируем модуль, указав только его имя без расширения:

```
>>> import mm
>>> mm.f() # через точку из модуля mm вызываем функцию f
4
```

Создадим следующий модуль (по аналогии с предыдущим), укажем для него имя **mtest.py**:

```
# mtest.py
print('test')
```

Импортируем созданный модуль несколько раз подряд:

```
>>> import mtest
test
>>> import mtest
```

Заметим, что, во-первых, импорт модуля приводит к выполнению содержащихся в нем инструкций, во-вторых, модуль повторно не импортируется. Дело в том, что импорт модулей — ресурсоемкий процесс, поэтому лишний раз Python его не производит. В случае внесения изменений в модуль необходимо воспользоваться функцией **reload** из модуля **imp**:

```
>>> import imp
>>> imp.reload(mtest)
test
<module 'mtest' from 'C:\\Python36-32\\mtest.py'>
```

Этот код указывает Python, что модуль требует повторной загрузки. После вызова функции **reload** с указанием в качестве аргумента имени модуля обновленный модуль загрузится повторно.

Продолжим эксперименты. Создадим еще один модуль с именем **mypr.py**:

```
# mypr.py
def func(x):
    return x ** 2 + 7

x = int(input("Введите значение: "))
print(func(x))
```

Импорт модуля приводит к выполнению инструкций, содержащихся внутри модуля:

```
>>> import mypr
Введите значение: 111
12328
```

Каким образом поступить, если необходимо импортировать функцию **func** из модуля для использования ее в другой программе? Чтобы отделить исполнение модуля (Run → Run Module) от его импортирования (**import mypr**), существует специальная переменная **__name__** (имена специальных функций и переменных в Python начинаются и заканчиваются двумя нижними подчеркиваниями).

При выполнении модуля (нажатии *<F5>*) переменная **__name__** будет содержать строку "**__main__**", а в случае импорта — имя модуля (рис. 4.20).

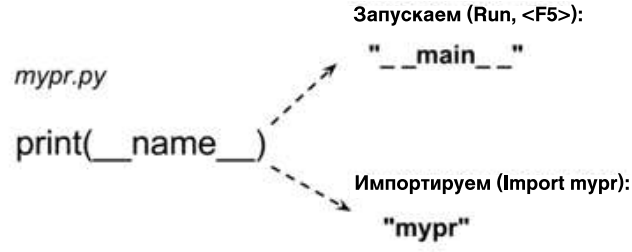


Рис. 4.20. **Разделение импорта и выполнения модуля**

Рассмотрим использование этого приема на практике, для этого создадим модуль с именем **mmtest.py**:

```
# mmtest.py
def func(x):
    return x ** 2 + 7

if __name__ == "__main__":
    x = int(input("Введите значение: "))
    print(func(x))
```

При запуске (Run → Run Module) данного модуля выполнятся все инструкции, содержащиеся в нем, так как условие **__name__ == "__main__"** вернет значение **True**. Импорт модуля (import mmtest) приведет к загрузке в память функции **func**.

4.17. Автоматизированное тестирование функций

В разработке программного обеспечения существует подход (разработка через тестирование, *TDD, Test-Driven Development*), при котором сначала разрабатываются тесты, т.е. сперва описывают, как программа (функция) должна работать, а после этого пишут саму программу (функцию). Это позволяет впоследствии проверить правильность ее написания.

Представим, что мы уже создали функцию **func_m**, которая вычисляет среднее арифметическое, округляя его до трех знаков после запятой. Представим, как бы мы вызвали такую функцию, т.е. напишем тесты:

```
>>> func_m(20, 30, 70)
40.0
>>> func_m(1, 5, 8)
4.667
```

Теперь реализуем функцию **func_m** и в ее описание добавим тесты, проверяющие правильность работы функции. Затем импортируем модуль **doctest** и вызовем функцию **testmod**, которая запустит текущий модуль и проверит, совпадают ли результаты вызовов функций в описании с тем, что получается в реальности. Если результат совпадает, то на экране ничего не появится, в ином случае отобразятся ошибки.

В отдельном файле с именем **test_func.py** создайте и выполните (Run Run Module) следующий код:

```
# test_func.py
def func_m(v1, v2, v3):
    """Вычисляет среднее арифметическое трех чисел.

    >>> func_m(20, 30, 70)
    40.0

    >>> func_m(1, 5, 8)
    4.667

    """
    return round((v1 + v2 + v3) / 3, 3)

import doctest
# автоматически проверяет тесты в документации
doctest.testmod()
```

В результате выполнения программа ничего не выведет на экран, так как автоматизированное тестирование не выявило ошибок. Теперь исправим тестовые вызовы в программе (**test_func2.py**):

```
# test_func2.py
def func_m(v1, v2, v3):
    """Вычисляет среднее арифметическое трех чисел.

    >>> func_m(20, 30, 70)
    60.0

    >>> func_m(1, 5, 8)
    6.667

    """
    return round((v1 + v2 + v3) / 3, 3)
```

```
import doctest
# автоматически проверяет тесты в документации
doctest.testmod()
```

В результате выполнения программы увидим сообщения о возникших ошибках в процессе проверки результатов:

```
**********************************************************
File "C:/Python36-32/test_func2.py", line 4,
in __main__.func_m
Failed example:
     func_m(20, 30, 70)
Expected:
     60.0
Got:
     40.0
**********************************************************
File “C:/Python36-32/test_func2”, line 7,
in __main__.func_m
Failed example:
     func_m(1, 5, 8)
Expected:
     6.667
Got:
     4.667
**********************************************************
1 items had failures:
     2 of 2 in __main__.func_m
***TestFailed*** 2 failures.
>>>
```

4.18. Строковые методы

Вызовем функцию **type** и передадим ей целочисленный аргумент:

```
>>> type(0)
<class 'int'>
```

Функция сообщила, что целочисленный объект 0 относится к классу '**int**', т.е. тип данных в Python является *классом*. Для упрощения представим класс как аналог модуля, т.е. набор функций и переменных, содержащихся внутри класса. Функции, которые находятся внутри класса, называются *методами*. Их главное отличие от вызова функций из модуля заключается в том, что в качестве первого аргумента метод принимает, например, строковый объект, если это метод строкового класса.

Рассмотрим пример вызова строкового метода:

```
>>> str.capitalize('hello')
'Hello'
```

По аналогии с вызовом функции из модуля указывается имя класса — **str**, затем через точку — имя строкового метода **capitalize**, который принимает один строковый аргумент (рис. 4.21).

Рис. 4.21. **Полная форма для строковых методов**

Метод — это обычная функция, расположенная внутри класса. Вызовем строковый метод **center**:

```
>>> str.center('hello', 20)
'  hello  '
```

Этот метод принимает два аргумента — строку и число (рис. 4.22).

Рис. 4.22. **Полная форма для строковых методов**

Форма вызова метода через обращение к его классу с помощью точки называется *полной формой*, но чаще используют *сокращенную форму вызова метода*:

```
>>> 'hello'.capitalize()
'Hello'
```

Первый аргумент метода поместили на место имени класса (рис. 4.23).

Рис. 4.23. **Сокращенная форма для вызова методов**

Важно отметить, что Python, перед тем как вызвать метод, всегда преобразует сокращенную форму в аналогичную ей полную форму.

Получение справки для метода производится путем вызова функции **help** (вместо имени модуля указывается имя класса):

```
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
     S.capitalize() -> str

     Return a capitalized version of S, i.e. make the
     first character have upper case and the rest lower
     case.
```

Вынесенный из метода первый строковый аргумент может быть выражением, возвращающим строку:

```
>>> ('TTA' + 'G' * 3).count('T')
2
```

Несложно догадаться, что делает метод **count**.

Следующий полезный метод — **format**[1]:

```
>>> '{0} и {1}'.format('труд', 'май')
'труд и май'
```

Вместо {0} и {1} подставятся аргументы метода **format**.

Поменяем их местами:

```
>>> '{1} и {0}'.format('труд', 'май')
'май и труд'
```

Python содержит строковые методы, которые возвращают истину (True) или ложь (False).

Строковый метод **startswith** проверяет, начинается ли строка с символа, переданного в качестве аргумента методу:

```
>>> 'spec'.startswith('a')
False
```

При работе с текстом на практике часто используют строковый метод **strip**, который возвращает строку, очищенную от символа переноса строки (\n) и пробелов:

```
>>> s = '                    \n ssssss \n'
>>> s.strip()
'ssssss'
```

Строковый метод **swapcase** возвращает строку с противоположными регистрами символов:

```
>>> 'Hello'.swapcase()
'hELLO'
```

Вызовы методов в Python можно производить в одну строку:

```
>>> 'ПРИВЕТ'.swapcase().endswith('т')
True
```

[1] Подробнее см. https://docs.python.org/3.1/library/string.html#format-examples.

В первую очередь вызывается строковый метод **swapcase** для строки **'ПРИВЕТ'**, затем для результирующей строки вызывается метод **endswith** с аргументом **'т'** (рис. 4.24).

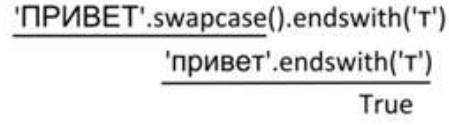


Рис. 4.24. **Порядок вызова методов**

Запустите каждый из перечисленных ниже строковых методов в интерактивном режиме на примере различных строк и разберитесь в принципе их работы.

Предположим, что переменная **s** содержит некоторую строку, тогда к ней применимы следующие методы[1]:

- **s.upper** — возвращает строку в верхнем регистре;
- **s.lower** — возвращает строку в нижнем регистре;
- **s.title** — возвращает строку, первый символ которой в верхнем регистре;
- **s.find('вет', 2, 3)** — возвращает позицию подстроки в интервале либо –1;
- **s.count('e', 1, 5)** — возвращает количество подстрок в интервале либо –1;
- **s.isalpha** — проверяет, состоит ли строка только из букв;
- **s.isdigit** — проверяет, состоит ли строка только из чисел;
- **s.isupper** — проверяет, написаны ли все символы в верхнем регистре;
- **s.islower** — проверяет, написаны ли все символы в нижнем регистре;
- **s.istitle** — проверяет, начинается ли строка с большой буквы;
- **s.isspace** — проверяет, состоит ли строка только из пробелов.

Выполним объединение двух строк с помощью оператора **+**:

```
>>> 'TT' + 'rr'
'TTrr'
```

При выполнении этого кода Python вызывает специальный строковый метод **__add__** и передает ему в качестве первого аргумента строку **'rr'**:

```
>>> 'TT'.__add__('rr')
'TTrr'
```

Напомним, что этот вызов затем преобразуется в полную форму:

```
>>> str.__add__("TT", 'rr')
'TTrr'
```

[1] https://docs.python.org/3/library/stdtypes.html#string-methods.

Забегая вперед, скажем, что за каждой из операций над типами данных скрывается свой специальный метод.

4.19. Списки

4.19.1. Создание списка

Предположим, что необходимо обработать информацию о курсах валют[1] (рис. 4.25).

Дата	Доллар США USD	ЕВРО EUR
30.05.2015	52.9716	58.0145
29.05.2015	52.2907	57.1433
28.05.2015	51.0178	55.6757
27.05.2015	50.3223	54.8412
26.05.2015	49.8613	54.7477
23.05.2015	49.7901	55.5508
22.05.2015	49.9204	55.5714

Рис. 4.25. **Информация о курсе валют**

Курс валюты на каждый день можно поместить в отдельную переменную:

```
>>> day1 = 56.8060
>>> day2 = 57.1578
```

Схематично подобная работа с переменными представлена на рис. 4.26.

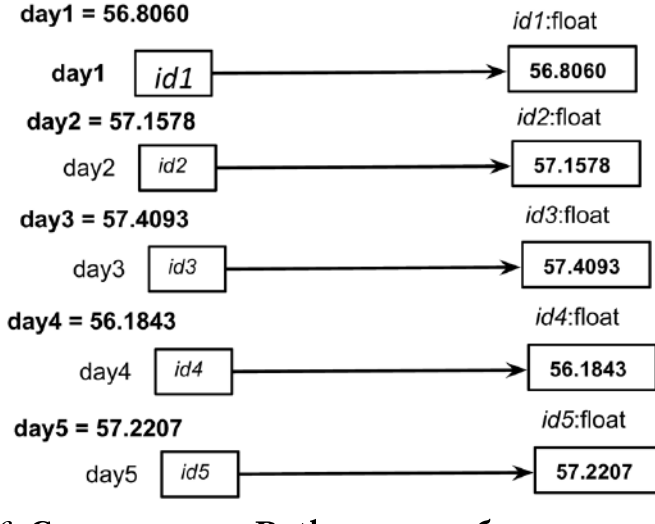


Рис. 4.26. **Схема памяти Python при работе с переменными**

[1] URL: http://www.sberometer.ru/cbr/.

Как поступить, если необходимо обработать курсы валют за последние два года? В таком случае на помощь приходят *списки*. Их можно рассматривать как аналог массива в других языках программирования, за исключением важной особенности — списки в качестве своих элементов могут содержать объекты различных типов.

Список (*list*) в Python является объектом, поэтому может быть присвоен переменной (напоминаем, что в переменной хранится адрес объекта).

Представим список для задачи с курсом валют:

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
```

Список позволяет хранить разнородные данные, обращаться к которым можно через имя списка (в данном случае переменную **e**). Рассмотрим схематично, как Python работает со списками (рис. 4.27).

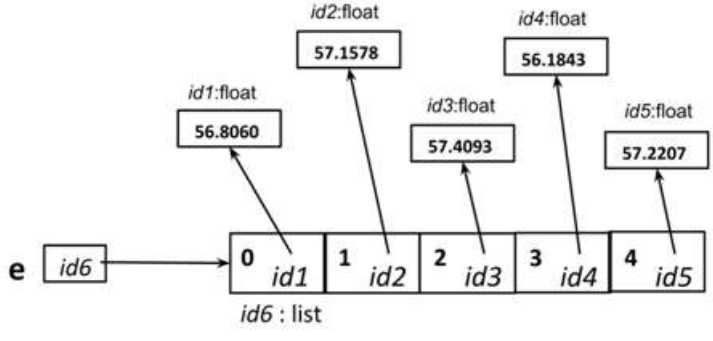


Рис. 4.27. **Схема памяти Python при работе со списком**

Переменная **e** содержит адрес списка (*id*6), каждый элемент списка содержит указатель (адрес) объекта. В общем виде создание списка показано на рис. 4.28.

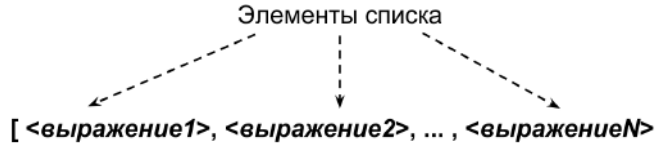


Рис. 4.28. **Общая форма создания списка**

На месте элементов списка могут находиться сложные выражения.

4.19.2. Операции над списками

Обращаться к отдельным элементам списка можно по их *индексу* (позиции), начиная с нуля (по аналогии со строками):

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e[0]
56.806
>>> e[1]
```

```
57.1578
>>> e[-1] # последний элемент
57.2207
```

Обращение по несуществующему индексу приведет к ошибке **IndexError**:

```
>>> e[100]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    e[100]
IndexError: list index out of range
```

До настоящего момента мы рассматривали неизменяемые типы данных (классы). Вспомните, как Python ругался при попытке изменить строку. Проведем эксперимент по изменению списка:

```
>>> h = ['Hi', 27 , -8.1, [1,2]] # создание списка
>>> h[1] = 'hello' # изменение элемента списка в позиции 1
>>> h # результирующий список
['Hi', 'hello', -8.1, [1, 2]]
>>> h[1] # измененный элемент списка
'hello'
```

На рис. 4.29 представлена схема памяти в момент создания списка **h**.

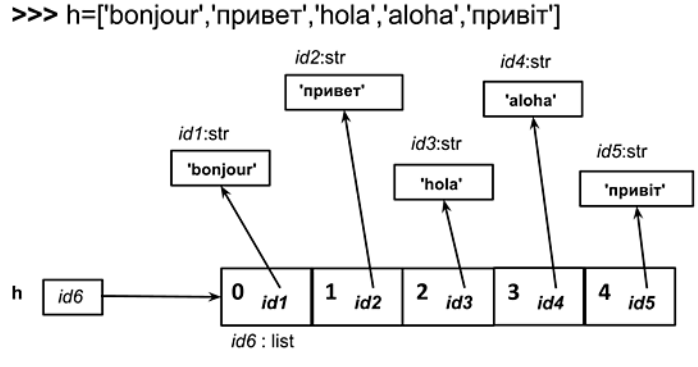


Рис. 4.29. **Схема памяти Python при работе со списком**

Произведем изменение списка:

```
>>> h[1] = 'hello'
>>> h
['bonjour', 'hello', 'hola', 'aloha', 'привіт']
>>> h[1]
'hello'
```

Схема памяти в момент изменения списка показана на рис. 4.30.

Видим, что в памяти создается новый строковый объект **‘hello’**. Затем адрес этого объекта (*id*7) помещается в первую ячейку списка (вместо адреса *id*2). Python обнаружит, что на объект по адресу *id*2 больше нет ссылок, поэтому удалит его из памяти (произведет автоматическую сборку мусора).

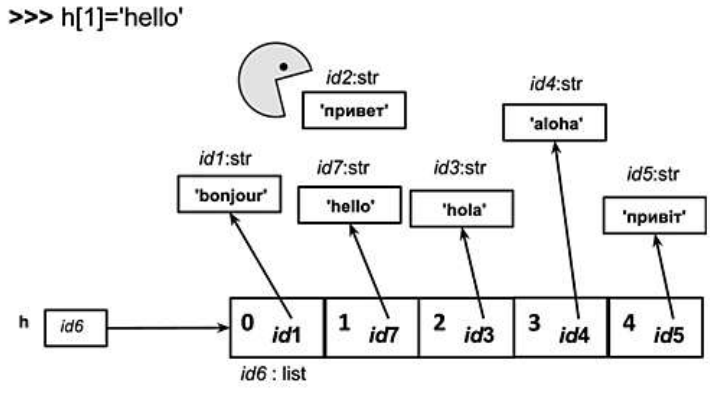


Рис. 4.30. **Схема памяти Python в момент изменения списка**

Список, наверное, наиболее часто встречающийся тип данных, с которым приходится сталкиваться при написании программ. Это связано со встроенными в Python функциями, которые позволяют легко и быстро обрабатывать списки:

- **len(**L**)** — возвращает число элементов в списке L;
- **max(**L**)** — возвращает максимальное значение в списке L;
- **min(**L**)** — возвращает минимальное значение в списке L;
- **sum(**L**)** — возвращает сумму значений в списке L;
- **sorted(**L**)** — возвращает копию списка L, в котором элементы упорядочены по возрастанию. Не изменяет список L.

Примеры вызовов функций для работы со списками:

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
>>> len(e)
5
>>> max(e)
57.4093
>>> min(e)
56.1843
>>> sum(e)
284.7781
>>> sorted(e)
[56.1843, 56.806, 57.1578, 57.2207, 57.4093]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
```

Оператор «+», примененный к спискам, служит для их объединения (вспомните строки):

```
>>> original = ['H','B']
>>> final = original + ['T']
>>> final
['H', 'B', 'T']
```

Операция повторения (вновь возникает аналогия со строками):

```
>>> final = final * 5
>>> final
['H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B',
'T', 'H', 'B', 'T']
```

Инструкция **del** позволяет удалять из списка элементы по указанному индексу (это инструкция, поэтому скобки не требуются):

```
>>> del final[0]
>>> final
['B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T',
'H', 'B', 'T']
```

Рассмотрим интересный пример. Создадим функцию **f**, объединяющую два списка:

```
>>> def f(x, y):
  return x + y

>>> f([1, 2, 3], [4, 5, 6])
[1, 2, 3, 4, 5, 6]
```

Теперь передадим в качестве аргументов функции **f** две строки:

```
>>> f("123", "456")
'123456'
```

Далее передадим в качестве аргументов функции **f** два числа:

```
>>> f(1, 2)
3
```

В итоге небольшая функция умеет объединять или складывать переданные ей объекты в зависимости от их класса (типа данных).

Оператор **in** по аналогии со строками применим к списку:

```
>>> h = ['bonjour', 7, 'hola', -1.0, 'привіт']
>>> if 7 in h:
  print('Значение есть в списке')

Значение есть в списке
```

Аналогично строкам, для списка существует оператор извлечения среза:

```
>>> h = ['bonjour', 7, 'hola', -1.0, 'привіт']
>>> h
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> g = h[1:2]
>>> g
[7]
```

Схема памяти применительно к срезам представлена на рис. 4.31.

Переменной **g** присваивается адрес нового списка (*id*7), содержащего указатель на числовой объект, выбранный с помощью среза.

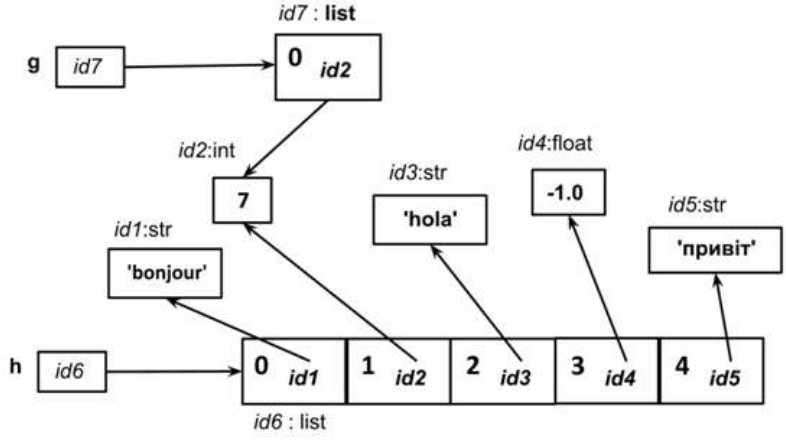


Рис. 4.31. **Схема памяти Python при работе со срезами**

Вернемся к инструкции **del** и удалим с помощью среза подсписок:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]    # удаление подсписка
>>>a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

4.19.3. Псевдонимы и копирование списков

Рассмотрим особенность, связанную с изменяемостью спискового типа данных, для этого выполним следующие инструкции:

```
>>> h
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> p = h    # содержат указатель на один и тот же
             # список
>>> p
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> p[0] = 1      # модифицируем одну из переменных
>>> h          # изменилась другая переменная!
[1, 7, 'hola', -1.0, 'привіт']
>>> p
[1, 7, 'hola', -1.0, 'привіт']
```

На схеме, представленной на рис. 4.32, видно, что переменные **p** и **h** указывают на один и тот же список, т.е. являются псевдонимами.

Возникает вопрос, как проверить, ссылаются ли переменные на один и тот же список:

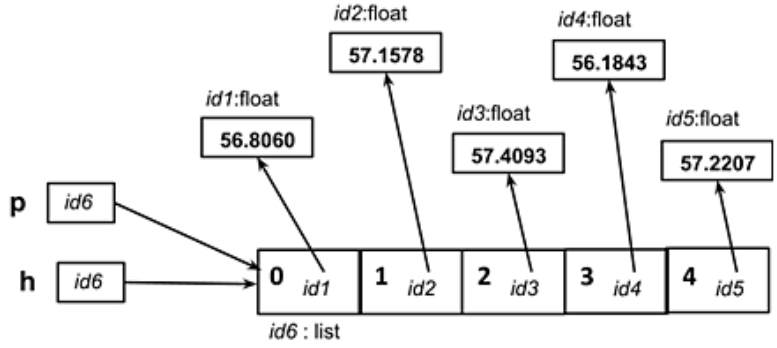


Рис. 4.32. **Схема памяти Python при работе с псевдонимами**

```
>>> x = y = [1, 2] # создали псевдонимы
>>> x is y # ссылаются ли переменные на один и тот же
           # объект
True
>>> x = [1, 2]
>>> y = [1, 2]
>>> x is y
False
```

К спискам применимы два вида копирования. Первый вид – *поверхностное копирование*, при котором создается новый объект, но он будет заполнен ссылками на элементы, которые содержались в оригинале:

```
>>> a = [4, 3, [2, 1]]
>>> b = a[:]
>>> b is a
False
>>> b[2][0] = -100
>>> a
[4, 3, [-100, 1]] # список a тоже изменился
>>>
```

Второй вид копирования — *глубокое копирование*. При глубоком копировании создается новый объект и рекурсивно создаются копии всех объектов, содержащихся в оригинале:

```
>>> import copy
>>> a = [4, 3, [2, 1]]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
>>> a
[4, 3, [2, 1]] # список a не изменился
>>>
```

С одной стороны, список предоставляет возможность модификации, с другой — появляется опасность повлиять на псевдоним списка.

4.19.4. Методы списка

Работа с методами списка похожа на работу со строковыми методами[1]. Далее приведены наиболее популярные методы списка[2]:

```
>>> colors = ['red', 'orange', 'green']
>>> colors.extend(['black','blue']) # расширяет
                                    # список списком
>>> colors
['red', 'orange', 'green', 'black', 'blue']

>>> colors.append('purple') # добавляет элемент в список
>>> colors
['red', 'orange', 'green', 'black', 'blue', 'purple']

>>> colors.insert(2,'yellow')  # добавляет элемент
                               # в указанную позицию
>>> colors
['red', 'orange', 'yellow', 'green', 'black', 'blue',
'purple']

>>> colors.remove('black')  # удаляет элемент из списка
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']

>>> colors.count('red') # количество повторений
                        # аргумента метода
1

>>> colors.index('green') # возвращает позицию в списке
3

>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']

>>> colors.pop() # удаляет и возвращает последний
                 # элемент списка
'purple'

>>> colors
['red', 'orange', 'yellow', 'green', 'blue']

>>> colors.reverse() # список в обратном порядке
>>> colors
['blue', 'green', 'yellow', 'orange', 'red']

>>> colors.sort() # сортирует список (вспомните
                  # о сравнении строк)
```

---

[1] Только вместо класса **str** используется **list**, а в качестве первого аргумента в полной форме метод принимает список.

[2] Подробнее см. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists.

```
>>> colors
['blue', 'green', 'orange', 'red', 'yellow']

>>> colors.clear() # или del color[:], очищает список,
                   # с версии Python 3.3
>>> colors
[]
```

4.19.5. Преобразование типов

На практике довольно часто возникает необходимость в изменении строк, но напрямую мы этого сделать не можем (строки относятся к неизменяемому типу данных). Тогда на помощь приходят списки. Решением может стать преобразование строки в список, изменение списка и обратное преобразование списка в строку:

```
>>> s = 'Строка для изменения'
>>> list(s) # функция list пытается преобразовать
            # аргумент в список
['С', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ',
'и', 'з', 'м', 'е', 'н', 'е', 'н', 'и', 'я']
>>> lst = list(s)
>>> lst[0] = 'М'  # изменяем список, полученный из стро-
ки
>>> lst
['М', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ',
'и', 'з', 'м', 'е', 'н', 'е', 'н', 'и', 'я']
>>> s = ''.join(lst)  # преобразуем список в строку
>>> s
'Мтрока для изменения'
```

Отдельно остановимся на использовании *строкового* метода **join**:

```
>> A = ['red', 'green', 'blue']
>>> ' '.join(A)
'red green blue'
>>> ''.join(A)
'redgreenblue'
>>> '***'.join(A)
'red***green***blue'
```

Строковый метод **join** принимает в качестве аргумента список, который необходимо преобразовать в строку, а в качестве строкового объекта указывается соединитель элементов списка.

Похожим образом можно преобразовать число в список (на промежуточном этапе используется строка) и затем изменить полученный список:

```
>>> n = 73485384753846538465
>>> list(str(n)) # число преобразуем в строку, затем
                 # строку в список
['7', '3', '4', '8', '5', '3', '8', '4', '7', '5', '3',
'8', '4', '6', '5', '3', '8', '4', '6', '5']
```

Если строка содержит разделитель, то ее можно преобразовать в список с помощью строкового метода **split**, который по умолчанию в качестве разделителя использует пробел:

```
>>> s = 'd a dd dd gg rr tt yy rr ee'.split()
>>> s
['d', 'a', 'dd', 'dd', 'gg', 'rr', 'tt', 'yy', 'rr', 'ee']
```

Вызов метода **split** с разделителем ":":

```
>>> s = 'd:a:dd:dd:gg:rr:tt:yy:rr:ee'.split(":")
>>> s
['d', 'a', 'dd', 'dd', 'gg', 'rr', 'tt', 'yy', 'rr', 'ee']
```

4.19.6. Вложенные списки

Ранее уже упоминалось, что в качестве элементов списка могут находиться объекты любого типа (класса), например списки:

```
>>> lst = [['A', 1], ['B',2], ['C',3]]
>>> lst
[['A', 1], ['B', 2], ['C', 3]]
>>> lst[0]
['A', 1]
```

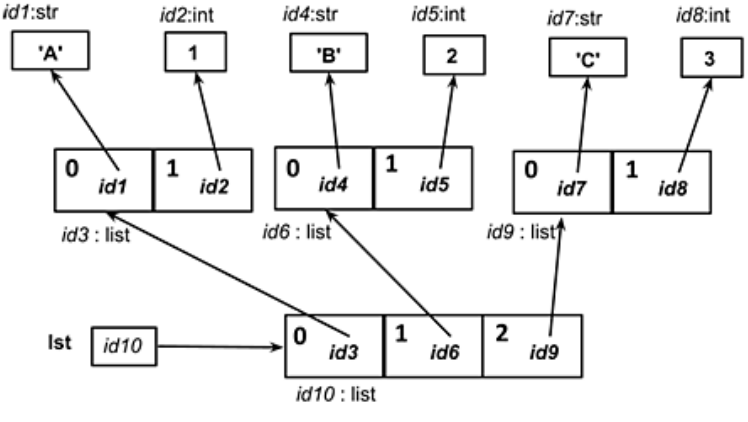


Рис. 4.33. **Схема памяти при работе Python с вложенными списками**

Подобные структуры данных используются для хранения матриц. Обращение к вложенному списку происходит через указание двух индексов (индекс внешнего списка и индекс вложенного списка):

```
>>> lst[0][1]
1
```

Схематично вложенные списки показаны на рис. 4.33.

4.20. Итерации

Язык программирования Python позволяет в кратчайшие сроки создавать прототипы[1] реальных программ благодаря тому, что в не-

[1] Быстрая, черновая реализация будущей программы.

го заложены конструкции для решения типовых задач, с которыми часто приходится сталкиваться программисту.

Вспомните, как мы решали задачу подсчета суммы элементов списка через вызов единственной функции **sum([1, 4, 5, 6, 7.0, 3, 2.0])**.

Рассмотрим еще несколько подобных приемов, которые значительно упрощают жизнь разработчика на языке Python.

4.20.1. Инструкция for

К примеру, мы хотим вывести на экран каждый из элементов списка **num**:

```
>>> num = [0.8, 7.0, 6.8, -6]
>>> num
[0.8, 7.0, 6.8, -6]
>>> print(num[0],'- number')
0.8 - number
>>> print(num[1],'- number')
7.0 - number
```

Теперь представим, что в списке пятьсот элементов... Для подобных случаев в Python существуют *циклы*. Перепишем этот пример с использованием инструкции **for**:

```
>>>num = [0.8, 7.0, 6.8, -6]
>>>for i in num:
print(i, '- number')

0.8 - number
7.0 - number
6.8 - number
-6 - number
```

Цикл **for** позволяет обратиться к каждому из элементов указанного списка. Имя переменной, в которую на каждом шаге будет помещаться элемент списка, выбирает программист. В нашем примере это переменная с именем **i**. На первом шаге переменной **i** присваивается первый элемент списка **num**, равный 0.8. Затем программа переходит в тело цикла **for**, отделенное отступами (четыре пробела или одна табуляция). В теле цикла содержится вызов функции **print**, которой передается на вход переменная **i**. На следующем шаге (итерации цикла) переменной **i** будет присвоен второй элемент списка, равный 7.0. Произойдет вызов функции **print** для отображения текущего содержимого переменной **i** на экране и т.д. Тело цикла выполняется до тех пор, пока не будет достигнут конец списка.

В общем виде инструкция **for** для обращения к каждому из элементов указанного списка выглядит так:

```
for <переменная> in <список> :
    <тело цикла>
```

Например:

```
>>> for i in [1, 2, 'hi']:
    print(i)

1
2
hi
```

Инструкция **for** схожим образом работает для строк:

```
>>> for i in 'hello':
    print(i)

h
e
l
l
o
```

По аналогии со списком для строки происходит обращение к каждому из символов, пока не будет достигнут конец строки. В общем виде запись инструкции цикла **for** для заданной строки выглядит так:

```
for <переменная> in <строка> :
    <тело цикла>
```

Инструкция **for** позволяет производить операции над отдельными элементами списка или строки (в общем случае *последовательности*):

```
>>> num = [0.8, 7.0, 6.8, -6]
>>> for i in num:
  if i == 7.0:
    print (i, '- число 7.0')

7.0 - число 7.0
```

В примере условная инструкция **if** позволила вывести на экран только указанное значение из списка, выполнив в теле цикла сравнение с каждым элементом списка.

Похожим образом с помощью инструкции цикла производится поиск символа в строке:

```
>>> country = "Russia"
>>> for ch in country:
  if ch.isupper():
    print (ch)

R
```

4.20.2. Функция range

Достаточно часто на практике при разработке программ необходимо получить *диапазон* целых чисел (рис. 4.34).

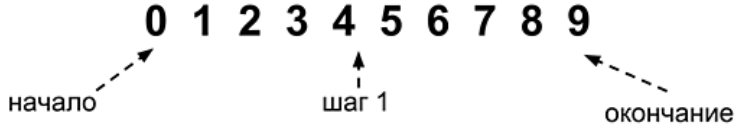


Рис. 4.34. **Диапазон целых чисел**

Для решения этой задачи в Python предусмотрена функция **range**[1]. В качестве аргументов функция принимает начальное значение диапазона (по умолчанию 0), конечное значение (*не включительно*) и шаг (по умолчанию 1). Если вызвать функцию **range** в интерактивном режиме, то диапазона чисел мы не увидим[2]:

```
>>> range(0,10,1)
range(0, 10)
>>> range(10)
range(0, 10)
```

Для создания диапазона (неизменяемой последовательности) необходимо использовать, например, инструкцию **for**:

```
>>> for i in range(0, 10, 1):
    print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
>>> for i in range(10):
    print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
>>> for i in range(2, 20, 2):
    print(i, end=' ')

2 4 6 8 10 12 14 16 18
```

Таким образом, в переменную **i** на каждом шаге цикла (итерации) будет записываться значение из диапазона, который генерируется функцией **range**.

При желании можно получить диапазон в обратном порядке следования (обратите внимание на аргументы функции **range**):

```
>>> for i in range(20, 2, -2):
    print(i, end=' ')

20 18 16 14 12 10 8 6 4
```

---

[1] Подробнее см. https://docs.python.org/3.6/library/stdtypes.html#range.

[2] Функция **range** вернула объект типа **range**. Интересно, что **range** для экономии места хранит в памяти только начало, окончание и шаг диапазона.

Теперь с помощью диапазона найдем сумму чисел на интервале от 1 до 100:

```
>>> total = 0
>>> for i in range(1, 101):
    total = total + i

>>> total
5050
```

Переменной **i** на каждой итерации цикла последовательно присваиваются значения из диапазона от 1 до 100 (напоминаем, что крайнее правое значение не включается). На первой итерации переменная **total** содержит значение 0 (инициализировали **total** перед входом в цикл). В теле цикла сначала вычисляется правая часть инструкции присваивания, т.е. **total + i**. Переменная **i** на первом шаге содержит значение 1 (первое значение из диапазона), таким образом, правая часть инструкции присваивания будет равна значению 1. Это значение помещается в левую часть, т.е. присваивается переменной **total**. На втором шаге **total** уже будет содержать значение 1, **i** — 2, т.е. правая часть инструкции присваивания станет равна 3, далее это значение помещается в переменную **total** и т.д., до конца диапазона, т.е. до значения 100. По окончании цикла в **total** будет содержаться искомая сумма чисел.

В Python возможно более элегантное решение данной задачи в функциональном стиле:

```
>>> sum(list(range(1, 101)))
5050
```

Это решение требует небольших пояснений. Диапазоны можно использовать при создании списков:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
```

Вызов функции **sum** для списка в качестве аргумента приводит к вычислению суммы элементов списка.

Диапазон, создаваемый функцией **range**, на практике часто используется для задания *индексов*. Следующий пример демонстрирует изменение списка путем умножения каждого из его элементов на 2:

```
# range_two.py
lst = [4, 10, 5, -1.9]
print(lst)
for i in range(len(lst)):
    lst[i] = lst[i] * 2
print(lst)
```

В качестве аргумента функции **range** в примере задается длина списка. В этом случае создаваемый диапазон будет от 0 до **len(lst) - 1**. Python не включает крайний правый элемент диапазона, так как длина списка всегда на 1 больше, чем индекс последнего его элемента (напоминаем, индексация списка начинается с нуля).

В результате выполнения программы:

```
[4, 10, 5, -1.9]
[8, 20, 10, -3.8]
```

4.20.3. Создание списка

Первым рассмотрим пример создания списка с помощью метода **append**:

```
>>> a = []
>>> for i in range(1,15):
        a.append(i)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Инструкция **for** позволяет последовательно из диапазона от 1 до 14 выбрать числа и с помощью метода **append**, находящегося в теле цикла, добавить их к списку **a**.

Следующий пример — создание списка из диапазона с помощью функции **list**:

```
>>> a = list(range(1, 15))
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Python поддерживает отдельные возможности функционального стиля программирования, который строится исключительно на вызове функций и позволяет компактно записывать команды. Одна из таких возможностей — «списковое включение» (*list comprehension*, иначе «списочное встраивание»). Рассмотрим его применительно к созданию списка:

```
>>> a = [i for i in range(1,15)]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Правила работы со списковым включением представлены на рис. 4.35.

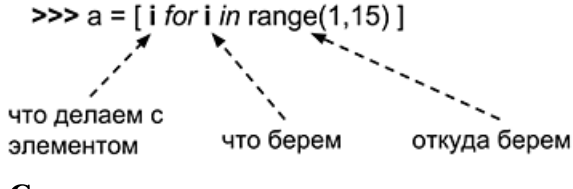


Рис. 4.35. **Схема использования спискового включения**

В следующем примере выбираем из диапазона все числа от 1 до 14, возводим их в квадрат и «на лету» формируем из них новый список:

```
>>> a = [i**2 for i in range(1, 15)]
>>> a
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

Списковое включение позволяет задавать условие для выбора значения из диапазона (в следующем примере исключили значение 4):

```
>>> a = [i**2 for i in range(1, 15) if i != 4]
>>> a
[1, 4, 9, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

Вместо диапазонов, генерируемых функцией **range**, можно указывать существующий список:

```
>>> a = [2, -2, 4, -4, 7, 5]
>>> b = [i**2 for i in a]
>>> b
[4, 4, 16, 16, 49, 25]
```

В примере последовательно выбираются значения из списка **a**, каждый из его элементов возводится в квадрат и «на лету» добавляется в новый список.

По аналогии со списками можно последовательно перебирать символы из строки и формировать из них список (в следующем примере исключили символ **i**):

```
>>> c = [c*3 for c in 'list' if c != 'i']
>>> c
['lll', 'sss', 'ttt']
```

Функция **map**[1] позволяет, используя функциональный стиль программирования, создавать новый список на основе существующего:

```
>>> def f(x):
    return x + 5

>>> list(map(f, [1, 3, 4]))
[6, 8, 9]
```

В примере функция **map** принимает в качестве аргументов имя функции **f** и список (или строку). В процессе вызова функции **map** каждый элемент указанного списка (или строки) подается на вход функции **f**, и результат вызова функции **f** для каждого из элементов указанного списка «на лету» добавляется как элемент нового списка. Функция **map** возвращает объект типа **map**, поэтому получить итоговый список можно с помощью инструкции **for** либо функции **list**.

[1] Подробнее см.: https://docs.python.org/3.6/library/functions.html#map.

Пример вызова функции **map** для строки:

```
>>> def f(s):
    return s * 2

>>> list(map(f, “hello”))
['hh', 'ee', 'll', 'll', 'oo']
```

Функциональные возможности Python позволяют определять небольшие однострочные функции на месте вызова функции (исторически такие функции получили название λ-функций):

```
>>> list(map(lambda s: s*2, "hello"))
['hh', 'ee', 'll', 'll', 'oo']
```

Далее рассмотрим генерацию списка, состоящего из случайных целых чисел:

```
>>> from random import randint
>>> A = [randint(1, 9) for i in range(5)]
>>> A
[2, 1, 1, 7, 8]
```

В данном примере функция **range** выступает как счетчик количества элементов создаваемого списка. Обратите внимание, что при формировании нового списка переменная **i** не используется. В результате пять раз будет произведен вызов функции **randint**[1], которая сгенерирует целое псевдослучайное число из интервала от 1 до 9, и уже это число добавится в новый список.

Сформируем список, значения для которого будем задавать с клавиатуры:

```
# input_list.py
a = [] # объявляем пустой список
n = int(input()) # указываем количество элементов списка
for i in range(n): # функция range контролирует длину
                   # списка
    new_element = int(input()) # указываем очередной
                               # элемент списка
    a.append(new_element) # добавляем элемент в список
    # последние две строки можно было заменить одной:
    # a.append(int(input()))
print(a)
```

Результат выполнения программы:

```
3
4
2
1
[4, 2, 1]
```

[1] Подробнее см.: https://docs.python.org/3.6/library/random.html.

Перепишем программу в одну строку, используя возможности спискового включения:

```
>>> A = [int(input()) for i in range(int(input()))]
3
4
2
1
>>> A
[4, 2, 1]
```

4.20.4. Инструкция while

Инструкция **for** используется, если заранее известно, сколько итераций необходимо выполнить (указывается через аргумент функции **range** или пока не закончится список/строка). Если заранее количество итераций цикла неизвестно, то применяется инструкция цикла **while**. Тело цикла **while** выполняется до тех пор, пока выражение является истинным или не произошел выход из цикла с помощью инструкции **break**.

```
while <выражение>:<тело цикла>
```

Далее приведен пример программы подсчета числа кроликов с использованием инструкции цикла **while**:

```
# while_rabbits.py
rabbits = 3
while rabbits > 0:
    print(rabbits)
    rabbits = rabbits - 1
```

Инструкция **while** выполняется до тех пор, пока число кроликов в условии положительное (**rabbits > 0**). На каждой итерации цикла переменная **rabbits** уменьшается на 1, чтобы не получился бесконечный цикл, при котором условие всегда будет оставаться истинным. В начале работы программы инициализируется переменная **rabbits** (переменной присваивается начальное значение 3), затем происходит переход в тело цикла **while**, так как условие **rabbits** > 0 является истинным (вернет значение **True**). Далее в теле цикла вызывается функция **print**, которая отобразит на экране текущее значение переменной **rabbits**. После этого переменная **rabbits** уменьшится на 1 и снова произойдет проверка условия 2 > 0 (вернет **True**), перейдем в тело цикла и т.д., пока не дойдем до условия 0 > 0. В этом случае вернется логическое значение **False** и произойдет выход из инструкции **while**. В результате выполнения программы:

```
3
2
1
```

Бесконечный цикл, организованный с помощью инструкции **while**, позволяет производить обработку введенных с клавиатуры строк:

```
# while_input.py
text = ""
while True:
    text = input("Введите число или стоп для выхода: ")
    if text == "стоп":
        print("Выход из программы! До встречи!")
        break    # инструкция выхода из цикла
    elif text == '1':
        print("Число 1")
    else:
        print("Что это?!")
```

Результат выполнения программы имеет следующий вид:

```
Введите число или стоп для выхода: 4
Что это?!
Введите число или стоп для выхода: 1
Число 1
Введите число или стоп для выхода: стоп
Выход из программы! До встречи!
```

Тело цикла **while** выполняется бесконечное число раз, так как **True** всегда является истиной. Выход из цикла осуществляется по инструкции **break**, если пользователь введет с клавиатуры строку «стоп».

Следующий исходный текст демонстрирует пример подсчета суммы чисел, встречающихся в строке:

```
# total_num.py
s = 'aa3aBbb6ccc'
total = 0
for i in range(len(s)):
    if s[i].isalpha(): # посимвольно проверяем наличие
                       # буквы
        continue # инструкция перехода к следующему
                 # шагу цикла
    total = total + int(s[i]) # накапливаем сумму,
                              # если встретилась цифра

print("сумма чисел:", total)
```

В примере используется инструкция **continue**. Выполнение данной инструкции приводит к переходу на следующий шаг цикла, т.е. все команды, которые находятся после **continue**, будут проигнорированы. Далее представлен результат выполнения программы:

```
сумма чисел: 9
```

4.20.5. Вложенные циклы

Python позволяет вкладывать инструкции циклов друг в друга так, как показано в следующем примере:

```
# for_outer_inner.py
outer = [1, 2, 3, 4]
inner = [5, 6, 7, 8]
for i in outer: # внешний цикл
    for j in inner: # вложенный (внутренний) цикл
        print('i=', i, 'j=', j)
```

В примере внешний цикл **for** перебирает все элементы списка **outer** (на первой итерации внешнего цикла фиксируется **i** = 1), затем управление передается вложенному циклу **for**, который проходит по всем элементам списка **inner** (изменяется переменная **j**). После того, как закончатся элементы в списке **inner**, управление вновь возвращается внешнему циклу (фиксируется следующее значение **i** = 2), после этого вложенный цикл проходит по всем элементам списка **inner**. Так повторяется до тех пор, пока не закончатся элементы в списке **outer** (рис. 4.36).

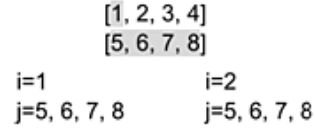


Рис. 4.36. **Пример выполнения вложенных циклов**

Результат выполнения программы:

```
i= 1 j= 5
i= 1 j= 6
i= 1 j= 7
i= 1 j= 8
i= 2 j= 5
i= 2 j= 6
i= 2 j= 7
i= 2 j= 8
i= 3 j= 5
i= 3 j= 6
i= 3 j= 7
i= 3 j= 8
i= 4 j= 5
i= 4 j= 6
i= 4 j= 7
i= 4 j= 8
```

Рассмотрим пример работы с вложенными списками (см. п. 4.19.6):

```
## for_lst.py
lst = [[1, 2, 3],
       [4, 5, 6]]
for i in lst:
    print(i)
```

Результат выполнения программы:

```
[1, 2, 3]
[4, 5, 6]
```

В примере с помощью инструкции **for** перебираются все элементы списка, которые также являются списками. Чтобы добраться до элементов вложенных списков, необходимо воспользоваться вложенной инструкцией **for**:

```
# for_lst2.py
lst = [[1, 2, 3],
       [4, 5, 6]]

for i in lst:     # внешний цикл
    print()
    for j in i:  # вложенный цикл
        print(j, end = "")
```

Результат выполнения программы:

```
123
456
```

4.21. Множества

Множество **(set)** в языке Python — неупорядоченная коллекция неизменяемых уникальных элементов:

```
>>> v = {'A', 'C', 4, '5', 'B'}
>>> v
{'C', 'B', '5', 4, 'A'}
```

Отметим, что порядок элементов созданного множества отличается от порядка элементов множества, отображенного на экране, так как множество — это *неупорядоченная коллекция*. На рис. 4.37 множество представлено схематично.

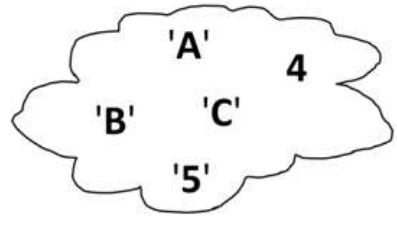


Рис. 4.37. **Множество в языке Python**

Множества в Python обладают интересными свойствами:

```
>>> v = {'A', 'C', 4, '5', 'B', 4}
>>> v
{'C', 'B', '5', 4, 'A'}
```

Видим, что повторяющиеся элементы, которые мы добавили при создании множества, были удалены (выполняется свойство

уникальности элементов множества). Перейдем к рассмотрению способов создания множества:

```
>>> set([3, 6, 3, 5])
{3, 5, 6}
```

В примере множество было создано на основе списка. Обратите внимание, что в момент создания множества из списка будут удалены повторяющиеся элементы. Это легкий способ очистить список от повторов:

```
>>> list(set([3, 6, 3, 5]))
[3, 5, 6]
```

Функция **range** позволяет создавать множества из диапазона:

```
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Рассмотрим некоторые операции над множествами[1]:

```
>>> s1 = set(range(5))
>>> s2 = set(range(2))
>>> s1
{0, 1, 2, 3, 4}
>>> s2
{0, 1}
>>> s1.add('5') # добавление элемента
>>> s1
{0, 1, 2, 3, 4, '5'}
```

У множеств в языке Python много общего с математическими множествами (рис. 4.38):

```
>>> s1.intersection(s2) # пересечение множеств (s1 & s2)
{0, 1}
>>> s1.union(s2)        # объединение множеств (s1 | s2)
{0, 1, 2, 3, 4, '5'}
```

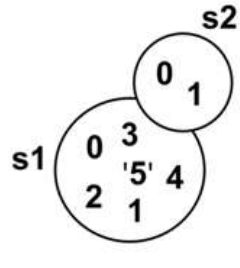


Рис. 4.38. **Операции над множествами**

4.22. Кортежи

Кортеж (**tuple**) в Python схож по своим свойствам со списком за исключением изменяемости. Кортежи используются, когда необходимо быть уверенным, что элементы структуры данных не будут

[1] Подробнее см.: https://docs.python.org/3/tutorial/datastructures.html#sets.

изменены в процессе работы программы. Вспомните проблему с псевдонимами (см. п. 4.19.3).

Рассмотрим некоторые популярные операции над кортежами[1]:

```
>>> ()      #  создание пустого кортежа
()
>>> (4)     # это не кортеж, а целочисленный объект!
4
>>> (4,)    # а вот это - кортеж, состоящий из одного
            # элемента!
(4,)
>>> b = ('1', 2, '4')    # создаем кортеж
>>> b
('1', 2, '4')
>>> len(b)  # определяем длину кортежа
3
>>> t = tuple(range(10)) # создание кортежа с помощью
                         # функции range
>>> t + b    # слияние кортежей
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, '1', 2, '4')
>>> r = tuple([1, 5, 6, 7, 8, '1']) # кортеж из списка
>>> r
(1, 5, 6, 7, 8, '1')
```

С помощью кортежей можно присваивать значения одновременно двум переменным:

```
>>> (x, y) = (10, 5)
>>> x
10
>>> y
5
>>> x, y = 1, 3 # убрали круглые скобки
>>> x
1
>>> y
3
```

Поменять местами содержимое двух переменных можно следующим образом:

```
>>> x, y = y, x
>>> x
3
>>> y
1
```

Кортеж изменить нельзя, но можно изменить, например, список, входящий в кортеж:

```
>>> t = (1, [1,3], '3')
>>> t[1]
```

[1] Подробнее см.: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences.

```
[1, 3]
>>> t[1][0] = '1'
>>> t
(1, ['1', 3], '3')
```

4.23. Словари

Словарь (**dict**) в Python — неупорядоченная изменяемая коллекция или, проще говоря, «список» с произвольными ключами, неизменяемого типа.

Рассмотрим пример создания словаря, который каждому слову на английском языке (*множество ключей словаря*) ставит в соответствие слово на испанском языке (*множество значений словаря*). Схематично такой словарь представлен на рис. 4.39.

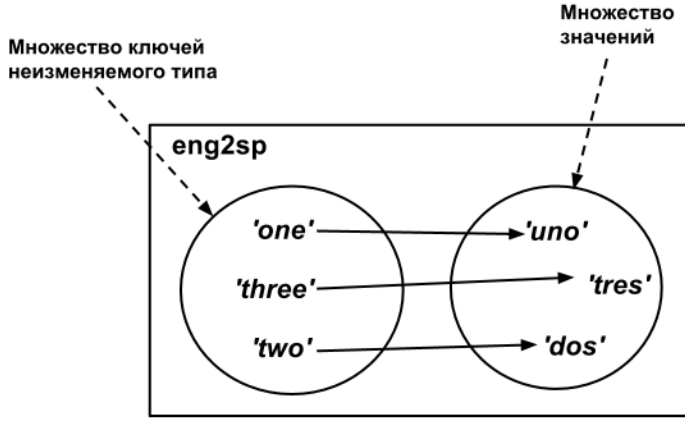


Рис. 4.39. **Схема соответствия множеству ключей множества значений словаря**

Создадим словарь-переводчик с английского языка на испанский язык:

```
>>> eng2sp = dict()      # создаем пустой словарь
>>> eng2sp
{}
>>> eng2sp['one']='uno' # добавляем 'uno' для элемента
                        # с индексом 'one'
>>> eng2sp
{'one': 'uno'}
>>> eng2sp['one']
'uno'
>>> eng2sp['two'] = 'dos'
>>> eng2sp['three'] = 'tres'
>>> eng2sp
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
```

В качестве индексов словаря в примере используются неизменяемые строки, но можно было воспользоваться кортежами, так как они тоже неизменяемые:

```
>>> e = {}
>>> e
{}
>>> e[(4,'6')] = '1'
>>> e
{(4, '6'): '1'}
```

Результирующий словарь **eng2sp** отобразился в «перемешанном» виде, так как, по аналогии с множествами, словари являются *неупорядоченной коллекцией*. Фактически, словарь — это отображение двух множеств: множества ключей и множества значений.

К словарям применим оператор **in**:

```
>>> eng2sp
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
>>> 'one' ineng2sp   # поиск по множеству КЛЮЧЕЙ
True
```

На практике часто словари используются, если требуется найти частоту встречаемости элементов последовательности (списке, строке, кортеже[1]).

Напишем функцию, которая возвращает словарь, содержащий частоту встречаемости элементов переданной *последовательности*:

```
def histogram(s):
       d = dict()
       for c in s:
            if c not in d:
                d[c] = 1
            else:
                d[c] = d[c] + 1 # или d[c] += 1
       return d
```

Результат вызова функции **histogram** для списка, строки, кортежа соответственно:

```
>>> histogram([2, 5, 6, 5, 4, 4, 4, 4, 3, 2, 2, 2, 2])
{2: 5, 3: 1, 4: 4, 5: 2, 6: 1}
>>> histogram("ywte3475eryt3478e477477474")
{'4': 6, '8': 1, 'e': 3, '3': 2, '7': 7, '5': 1, 'r': 1,
'y': 2, 'w': 1, 't': 2}
>>> histogram((5, 5, 5, 6, 5, 'r', 5))
{5: 5, 6: 1, 'r': 1}
```

4.24. Обработка исключений в Python

Рассмотрим пример программы, приводящей к ошибке:

```
# zero_error.py
x = int(input())
print(5/x)
```

[1] Все эти типы данных имеют общие свойства, так как относятся к последовательностям. Подробнее см.: https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range.

Выполним программу и убедимся, что перевод буквы в число и деление на нуль приводят к ошибкам:

```
r
Traceback (most recent call last):
  File "C:\Python36-32\zero_error.py", line 1, in <module>
    x = int(input())
ValueError: invalid literal for int() with base 10: 'r'
>>>
0
Traceback (most recent call last):
  File "C:\Python36-32\zero_error.py", line 2, in <module>
    print(5/x)
ZeroDivisionError: division by zero
```

Каким образом произвести проверку, чтобы избежать аварийного завершения программы? Можно, например, путем проверок контролировать ввод с клавиатуры, как это обычно делается в процедурных языках программирования:

```
# if_error.py
x = int(input())
if x == 0:
    print("Error!")
else:
    print(5/x)
```

Теперь программа при делении на нуль не производит аварийного завершения:

```
>>>
0
Error!
```

В Python реализован[1] перехват ошибок, основанный на обработке исключительных ситуаций. Рассмотрим измененную программу:

```
# try_error.py
try:
    x = int(input("Введите число: "))
    print(5/x)
except:
    print("Возникла ошибка деления на нуль")
```

Выполним программу и попытаемся разделить на нуль:

```
Введите число: 0
Возникла ошибка деления на нуль
```

В блок **try** помещается код, в котором может произойти ошибка. В случае возникновения ошибки (исключительной ситуации) управление передается в блок **except**. Повторно выполним програм-

[1] Другие объектно-ориентированные языки тоже поддерживают данный механизм.

му и обнаружим, что при возникновении ошибки перевода буквы в число, снова попадаем в блок **except**:

```
Введите число: к
Возникла ошибка деления на нуль
```

Дело в том, что **except** без указания типа исключительной ситуации перехватывает все виды возникающих ошибок. Как нам отделить ошибку деления на нуль от ошибки преобразования типов? Перейдем в интерактивный режим и выполним несколько операций, приводящих к исключениям[1]:

```
>>> 4/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4/0
ZeroDivisionError: division by zero
>>> int("r")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("r")
ValueError: invalid literal for int() with base 10: 'r'
```

При делении на нуль возникает ошибка **ZeroDivisionError**, при преобразовании типов — **ValueError**. Воспользуемся этим знанием и изменим нашу программу:

```
# try_except.py
try:
    x = int(input("Введите число: "))
    print(5/x)
except ZeroDivisionError:  # указываем тип исключения
    print("Возникла ошибка деления на нуль")
except ValueError:
    print("Возникла ошибка преобразования типов")
```

Запустим программу и убедимся, что теперь срабатывают различные блоки **except** в зависимости от типа возникающих исключений:

```
Введите число: 0
Возникла ошибка деления на нуль
>>>
Введите число: у
Возникла ошибка преобразования типов
```

Рассмотрим следующий пример обработки исключений:

```
# try_finally.py
try:
    x = int(input("Введите число: "))
    print(5/x)
```

[1] Подробнее см.: https://docs.python.org/3/library/exceptions.html#bltin-exceptions.

```
except ZeroDivisionError as z:
    print("Обрабатываем исключение - деление на нуль!")
    print(z) # выводим на экран информацию об ошибке
except ValueErrorasv:
    print("Обрабатываем исключение - преобразование"
        + " типов!")
    print(v)
else:
    print("Выполняется, если не произошло"
        + " исключительных ситуаций!")
finally:
    print("Выполняется всегда и в последнюю очередь!")
```

Выполним программу для различных входных значений:

```
Введите число: 0
Обрабатываем исключение - деление на нуль!
division by zero
Выполняется всегда и в последнюю очередь!
>>>
Введите число: r
Обрабатываем исключение - преобразование типов!
invalid literal for int() with base 10: 'r'
Выполняется всегда и в последнюю очередь!
```

В примере показано, что информацию об исключении можно помещать в переменную (с помощью инструкции **as**) и выводить на экран с помощью функции **print**.

Перехват исключений используется при написании функций. Рассмотрим пример обработки ошибки, когда искомый элемент не обнаружен в списке:

```
# list_find.py
def list_find(lst, target):
    try:
        index = lst.index(target)
    # метод index генерирует исключение ValueError:
    except ValueError:
    ## ValueError: value is not in list
        index = -1
    return index

print(list_find([3, 5, 6, 7], -6))
```

Результат выполнения программы:

```
-1
```

4.25. Работа с файлами

На практике данные для обработки часто поступают из внешних источников — файлов. Существуют различные форматы файлов, наиболее простой и универсальный — текстовый. Он открыва-

ется в любом текстовом редакторе (например, стандартном Блокноте). Расширения у текстовых файлов: .txt, .html, .csv и т.д.

Помимо текстовых есть другие типы файлов (музыкальные, видео, .doc, .ppt и пр.), которые открываются в специальных программах (музыкальных или видеопроигрывателях, MS Word и т.п.).

В этой главе остановимся на текстовых файлах, хотя возможности Python ими не ограничиваются.

Выполните следующие шаги:

1) создайте каталог *file_examples*;

2) с помощью (например, Блокнота) создайте в каталоге *file_examples* текстовый файл *example_text.txt*, содержащий следующий текст:

```
First line of text
Second line of text
Third line of text
```

3) создайте в каталоге *file_examples* файл *file_reader.py*, содержащий исходный текст программы на языке Python:

```
# file_reader.py
file = open('example_text.txt', 'r')
contents = file.read()
print(contents)
file.close()
```

Выполним программу *file_reader.py*:

```
First line of text
Second line of text
Third line of text
```

Описание программы представлено на рис. 4.40.

Рассмотренный подход к работе с файлами[1] Python унаследовал от языка C. По умолчанию, если не указывать режим открытия, используется открытие на «чтение» (можно открыть на «запись» или «добавление»).

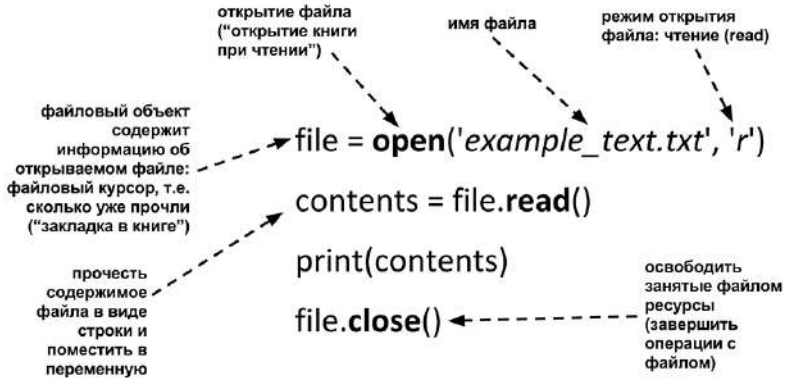


Рис. 4.40. **Описание программы для работы с файлами в Python**

[1] Подробнее см.: https://docs.python.org/3/library/os.html#os-file-dir.

Файлы особенно подвержены ошибкам во время работы с ними: жесткий диск может заполниться, пользователь может удалить используемый файл во время записи, файл могут переместить и т.д. Эти и другие типы ошибок можно перехватить с помощью обработки исключений.

Далее показан пример обработки ошибки открытия несуществующего файла с именем **1example_text.txt**:

```
# file_reader2.py
# Ошибка при открытии файла
try:
    f = open('1example_text.txt') # открытие файла на чте-
ние
except:
    print("Ошибка открытия файла")
else:   # выполняется, если не произошло ошибки
    f.close()
    print('(Очистка: Закрытие файла)')
```

Выполним программу:

```
Ошибка открытия файла
```

В дальнейшем для работы с файлами будем использовать *менеджер контекста* (инструкцию **with**[1]), который не требует ручного освобождения ресурсов, т.е. вызова метода **close**. В следующем примере представлен исходный код программы с использованием менеджера контекста:

```
# file_reader3.py
try:
    # освобождение ресурсов происходит автоматически
    # внутри менеджера контекста:
    with open('1example_text.txt', 'r') as file:
        contents = file.read()
    print(contents)
except:
    print("Ошибка открытия файла")
```

Выполним программу:

```
Ошибка открытия файла
```

Разберемся, каким образом Python определяет, где искать файл для открытия. В момент вызова функции **open** Python ищет указанный файл в текущем *рабочем каталоге* (там, где расположена запускаемая программа). Определить текущий рабочий каталог можно следующим способом:

```
>>> import os
>>> os.getcwd()
'C:\\Python36-32\\file_examples'
```

[1] Менеджер контекста используется не только при работе с файлами.

Если открываемый файл находится в другом каталоге, то необходимо указать путь к нему:

1) абсолютный путь, т.е. начиная с корневого каталога:

```
'C:\\Users\\Dmitriy\\data1.txt'
```

2) относительный путь *'data\\data1.txt'*, т.е. путь относительно текущего рабочего каталога *'C:\\Users\\Dmitriy\\home'* (см. рис. 4.41).

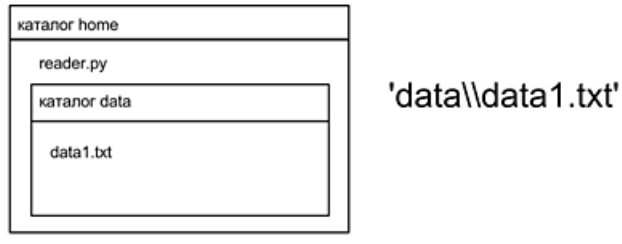


Рис. 4.41. **Схема определения относительного пути**

Далее рассмотрим некоторые способы чтения содержимого файла.

Пример чтения содержимого всего файла, начиная с текущей *позиции курсора* (перемещает курсор в конец файла):

```
# file_reader4.py
with open('example_text.txt', 'r') as file:
    contents = file.read()
print(contents)
```

Результат выполнения программы:

```
First line of text
Second line of text
Third line of text
```

Следующий пример демонстрирует работу с *курсором* (аналог закладки в книге, с которой продолжается чтение):

```
# file_reader5.py
with open('example_text.txt', 'r') as file:
    contents = file.read(10)   # указываем кол-во
                               # символов для чтения
    # курсор перемещается на 11 символ
    rest = file.read()     # читаем с 11 символа
print("10:", contents)
print("остальное:", rest)
```

Результат работы программы:

```
10: First line
остальное: of text
Second line of text
Third line of text
```

Если необходимо получить список, состоящий из строк, то можно воспользоваться методом **readlines** так, как показано в примере:

```
# file_reader6.py
with open('example_text.txt', 'r') as file:
    lines = file.readlines()
print(lines)
```

Результат работы программы:

```
['First line of text\n', 'Second line of text\n', 'Third
line of text']
```

Для выполнения следующего примера создайте файл **plan.txt**, содержащий текст:

```
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

Напишем программу, обрабатывающую содержимое файла **plan.txt**:

```
# file_reader7.py
with open('plan.txt', 'r') as file:
    # Читаем содержимое файла в виде списка строк
    planets = file.readlines()
print(planets)
# Отображаем элементы списка в обратном порядке
for planet in reversed(planets):
    # Возвращаем копию строки, из которой удален символ /n
    print(planet.strip())
```

В результате выполнения программы:

```
['Mercury\n', 'Venus\n', 'Earth\n', 'Mars\n',
'Jupiter\n', 'Saturn\n', 'Uranus\n', 'Neptune']
Neptune
Uranus
Saturn
Jupiter
Mars
Earth
Venus
Mercury
```

Если необходимо выполнить некоторые операции с каждой из строк файла, начиная с текущей позиции файлового курсора до конца файла:

```
# file_reader8.py
with open('plan.txt', 'r') as file:
    for line in file:
        print(line)
        print(len(line.strip()))
```

Результат выполнения программы:

```
Mercury

7
Venus

5
Earth

5
Mars

4
Jupiter

7
Saturn

6
Uranus

6
Neptune
7
```

В следующем примере производится запись строки в файл. Если файла с указанным именем в рабочем каталоге нет, то он будет создан, если файл с таким именем существует, то он будет перезаписан:

```
# file_write.py
with open("top.txt", 'w') as output_file:
    output_file.write("Hello!\n")
    # метод write возвращает число записанных символов
```

Для добавления строки в файл необходимо открыть файл в режиме **'a'** (сокр. от append):

```
# file_append.py
with open("top.txt", 'a') as output_file:
    output_file.write("Hello!\n")
```

4.26. Регулярные выражения

Python поддерживает мощный язык регулярных выражений[1], т.е. шаблонов, по которым можно искать/заменять некоторый текст. Например, регулярное выражение **'[ea]'** означает любой символ из набора в скобках, т.е. регулярное выражение **'r[ea]d'** совпадает с **'red'** и **'radar'**, но не со словом **'read'**. Для работы с регулярными выражениями необходимо импортировать модуль **re**:

[1] Подробнее см.: https://docs.python.org/3/howto/regex.html.

```
>>> import re
>>> re.search("r[ea]d", "rad") # указываем шаблон и текст
<_sre.SRE_Match object; span=(0, 3), match='rad'>
>>> re.search("r[ea]d", "read")
>>> re.search("[1-8]", "3")
<_sre.SRE_Match object; span=(0, 1), match='3'>
>>> re.search("[1-8]", "9")
>>>
```

В случае совпадения текста с шаблоном возвращается объект **match**[1], иначе возвращается **None**.

4.27. Объектно-ориентированное программирование на Python

4.27.1. Основы объектно-ориентированного подхода

Предположим, что существует набор строковых переменных для описания адреса проживания некоторого человека:

```
addr_name = 'Иван Иванов'
addr_line1 = 'Московский пр. 122' # адрес прописки
addr_line2 = ''           # фактический адрес проживания
addr_city = 'Москва'
addr_state = 'Восточный'      # административный округ
addr_zip = '123678'           # индекс адреса прописки
```

Напишем функцию **(addr.py)**, которая выводит на экран всю информацию о человеке:

```
def printAddress(name, line1, line2, city, state,
zip_code):
    print(name)
    if(len(line1) > 0):
        print(line1)
    if(len(line2) > 0):
        print(line2)
    print(city + ", " + state + " " + zip_code)
# Вызов функции, передача аргументов:
printAddress(addr_name, addr_line1, addr_line2,
    addr_city, addr_state, addr_zip)
```

Результат работы получившейся программы **(addr.py)**:

```
Иван Иванов
Московский пр. 122
Москва, Восточный 123678
```

Предположим, изменились сведения о человеке и появился индекс адреса проживания. Создадим новую переменную:

```
# создадим переменную, содержащую индекс адреса проживания
addr_zip2 = "678900"
```

[1] Подробнее см.: https://docs.python.org/3/library/re.html#match-objects.

Функция **printAddress** с учетом новых сведений будет иметь следующий вид (**addr1.py**):

```
def printAddress(
    name, line1, line2, city, state, zip, zip2):
    # добавили параметр zip2
    print(name)
    if(len(line1) > 0):
        print(line1)
    if(len(line2) > 0):
        print(line2)
    # добавили вывод на экран переменной zip2
    print(city + ", " + state + " " + zip + " " + zip2)

# Добавили новый аргумент addr_zip2:
printAddress(addr_name, addr_line1, addr_line2,
    addr_city, addr_state, addr_zip, addr_zip2)
```

Пришлось в нескольких местах изменить исходный текст программы (**addr1.py**), чтобы учесть новое условие. В чем состоит недостаток рассмотренного подхода? Огромное количество переменных! Чем больше сведений о человеке нужно обработать, тем больше переменных мы должны создать. Конечно, можно поместить все в список (элементами списка тогда будут строки), но в Python есть более универсальный подход для работы с наборами разнородных данных.

Объединим все сведения о человеке в единую структуру (*класс*) с именем **Address**:

```
class Address():     # имя класса выбирает программист
    name = ""        # строковое поле класса
    line1 = ""
    line2 = ""
    city = ""
    state = ""
    zip_code = ""
```

Класс в нашей программе задает шаблон для хранения места проживания человека. Превратить шаблон в конкретный адрес можно через создание *объекта* (*экземпляра*)[1] класса **Address**[2]:

```
homeAddress = Address()
```

Теперь заполним поля объекта конкретными значениями (через точку):

```
## заполняем поле name объекта homeAddress:
homeAddress.name = "Иван Иванов"
homeAddress.line1 = "Московский пр. 122"
```

[1] В Python классы являются объектами, но для упрощения будем считать, что это только шаблон для создания объектов.

[2] Вспомните о создании объекта класса int: a = int ().

```
homeAddress.line2 = "Тихая ул. 12"
homeAddress.city = "Москва"
homeAddress.state = "Восточный"
homeAddress.zip_code = "123678"
```

Создадим еще один объект класса **Address**, который содержит информацию о загородном доме того же человека:

```
# переменная содержит адрес объекта класса Address:
vacationHomeAddress = Address()
```

Зададим поля объекта, адрес которого находится в переменной **vacationHomeAddress**:

```
vacationHomeAddress.name = "Иван Иванов"
vacationHomeAddress.line1 = "пр.Мира 12"
vacationHomeAddress.line2 = ""
vacationHomeAddress.city = "Коломна"
vacationHomeAddress.state = "Центральный район"
vacationHomeAddress.zip_code = "12489"
```

Выведем на экран информацию о городе для основного и загородного адресов проживания (через указание имен объектов):

```
print("Основной адрес проживания " + homeAddress.city)
print("Адрес загородного дома "
    + vacationHomeAddress.city)
```

Изменим исходный код функции **printAddress** с учетом полученных знаний об объектах:

```
def printAddress(address): # передаем в функцию объект
    print(address.name)   # выводим на экран поле объекта
    if(len(address.line1) > 0):
        print(address.line1)
    if(len(address.line2) > 0):
        print(address.line2)
    print(address.city + ", " + address.state + "  "
    + address.zip_code)
```

Если объекты **homeAddress** и **vacationHomeAddress** ранее были созданы, то можем вывести информацию о них, передав в качестве аргумента функции **printAddress**:

```
printAddress(homeAddress)
printAddress(vacationHomeAddress)
```

В результате выполнения программы (**addr2.py**) получим:

```
Основной адрес проживания Москва
Адрес загородного дома Коломна
Иван Иванов
Московский пр. 122
Тихая ул. 12
Москва, Восточный 123678
Иван Иванов
пр.Мира 12
Коломна, Центральный район 12489
```

Возможности классов и объектов не ограничиваются лишь объединением переменных под одним именем, т.е. *хранением состояния объекта*. Классы также позволяют задавать функции внутри себя (*методы*) для работы с полями класса, т.е. влиять на *поведение объекта*. Для демонстрации создадим класс **Dog**:

```
# ndog.py
class Dog():
    age = 0       # возраст собаки
    name = ""     # имя собаки
    weight = 0    # вес собаки
    # первым аргументом любого метода всегда является
    # self, т.е. сам объект:
    def bark(self): # функция внутри класса называется
                    # методом
        # self.name - обращение к имени текущего
        # объекта-собаки
        print(self.name, " говорит гав")

# Создаем объект myDog класса Dog:
myDog = Dog()

# Присваиваем значения полям объекта myDog:
myDog.name = "Лайка" # имя собаки
myDog.weight = 20    # вес собаки
myDog.age = 1        # возраст собаки

# Вызываем метод bark объекта myDog,
# т.е. попросим собаку подать голос:
myDog.bark()
# Полная форма для вызова метода myDog.bark()
# будет: Dog.bark(myDog), где myDog - сам объект (self)
```

Результат работы программы:

```
Лайка говорит гав
```

Данный пример демонстрирует *объектно-ориентированный подход* в программировании, когда создаются объекты, приближенные к реальной жизни. Между объектами происходит взаимодействие по средствам вызова методов. Поля объекта (переменные) фиксируют его состояние, а вызов метода приводит к реакции объекта и (или) изменению его состояния (изменению переменных внутри объекта).

В предыдущем примере между созданием объекта **myDog** класса **Dog** и присвоением ему имени (**myDog.name = "Лайка"**) прошло некоторое время. Может произойти так, что программист забудет указать имя, и тогда собака останется безымянной — такого допустить нельзя. Избежать подобной ошибки позволяет специальный метод (*конструктор*), который вызывается автоматически

в момент создания объекта заданного класса. Рассмотрим пример работы конструктора:

```
# dog1.py
class Dog():
     name = ""
     # Конструктор вызывается в момент создания объекта
     # этого класса;
     # специальный метод Python, поэтому два нижних
     # подчеркивания
     def __init__(self):
          print("Родилась новая собака!")

# Создаем собаку (объект myDog класса Dog)
myDog = Dog()
```

Запустим программу:

```
Родилась новая собака!
```

Следующий пример демонстрирует присвоение имени собаке через вызов конструктора класса:

```
# dog2.py
class Dog():
     name = ""
     # Конструктор
     # Вызывается на момент создания объекта этого класса
     def __init__(self, newName):
          self.name = newName

# Создаем собаку и устанавливаем ее имя:
myDog = Dog("Лайка")

# Выводим имя собаки:
print(myDog.name)

# Следующая команда выдаст ошибку, потому что
# конструктору не было передано имя:
# herDog = Dog()
```

Теперь имя собаки присваивается в момент ее создания (рождения). В конструкторе указали **self.name**, так как в момент вызова конструктора вместо **self** подставится конкретный объект, т.е. **myDog**.

Результат выполнения программы:

```
Лайка
```

В предыдущем примере для обращения к имени собаки мы выводили на экран поле **myDog.name**, т.е. залезали во внутренности объекта и доставали оттуда информацию о нем. Звучит жутковато, поэтому для более «гуманной» работы добавим в класс **Dog** два метода **setName** и **getName**:

```
# dog3.py
class Dog():
    name = ""
    # Конструктор вызывается в момент создания объекта
    # этого класса
    def __init__(self, newName):
        self.name = newName
    # Можем в любой момент вызвать метод и изменить
    # имя собаки
    def setName(self, newName):
        self.name = newName
    # Можем в любой момент вызвать метод и узнать имя
    # собаки
    def getName(self):
        return self.name
    # возвращаем текущее имя объекта

# Создаем собаку с начальным именем:
myDog = Dog("Лайка")

# Выводим имя собаки:
print(myDog.getName())

# Устанавливаем новое имя собаки:
myDog.setName("Шарик")

# Проверяем, что имя изменилось:
print(myDog.getName())
```

Выполним программу:

```
Лайка
Шарик
```

4.27.2. Наследование классов

Объектно-ориентированный подход (ООП) в программировании тесно связан с мышлением человека, с устройством его памяти. Чтобы лучше понять особенности ООП, рассмотрим модель хранения и извлечения информации из памяти человека (модель предложена Коллинзом и Квиллианом)[1]. В своем эксперименте ученые использовали семантическую сеть, в которой были представлены знания о канарейке (рис. 4.42).

Например, «канарейка — это желтая птица, которая умеет петь», «птицы имеют перья и крылья, умеют летать» и т.п. Знания в этой сети представлены на различных уровнях: на нижнем уровне располагаются более частные знания, а на верхних — более общие. При таком подходе для понимания высказывания «Канарейка мо-

[1] *Гаврилова Т. А., Кудрявцев Д. В., Муромцев Д. И.* Инженерия знаний. Модели и методы : учебник. СПб. : Лань, 2016.

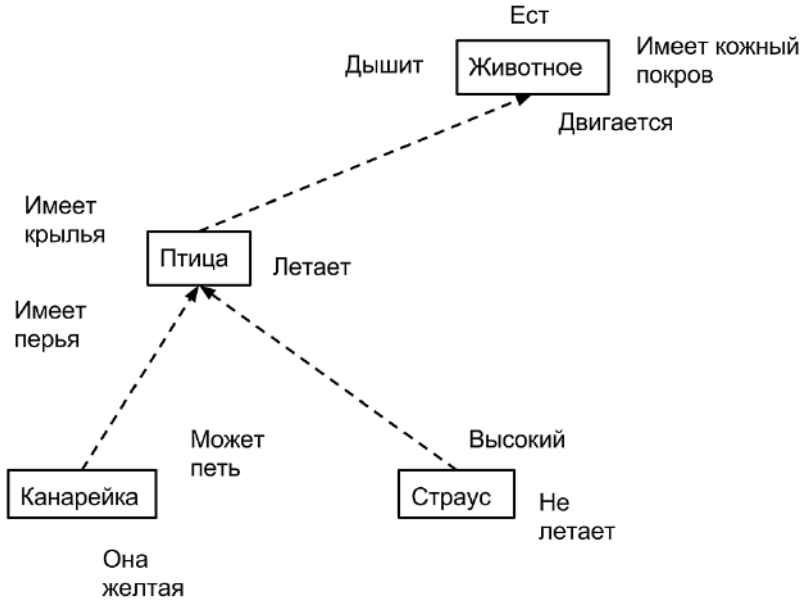


Рис. 4.42. **Модель хранения и извлечения информации из памяти человека (предложена Коллинзом и Квиллианом)**

жет летать» необходимо воспроизвести информацию о том, что канарейка относится к множеству птиц, и у птиц есть общее свойство «летать», которое распространяется (*наследуется*) и на канареек. Лабораторные эксперименты показали, что реакции людей на простые вопросы типа «Канарейка — это птица?», «Канарейка может летать?» или «Канарейка может петь?» различаются по времени. Ответ на вопрос «Может ли канарейка летать?» требует большего времени, чем на вопрос «Может ли канарейка петь». По мнению Коллинза и Квиллиана, это связано с тем, что информация запоминается человеком на наиболее абстрактном уровне. Вместо того чтобы запоминать все свойства каждой птицы, люди запоминают только отличительные особенности, например, желтый цвет и умение петь у канареек, а все остальные свойства переносятся на более абстрактные уровни: канарейка как птица умеет летать и покрыта перьями; птицы, будучи животными, дышат и питаются и т.д. Действительно, ответ на вопрос «Может ли канарейка дышать?» требует большего времени, так как человеку необходимо проследовать по иерархии понятий в своей памяти. С другой стороны, конкретные свойства могут перекрывать более общие, что также требует меньшего времени на обработку информации. Например, вопрос «Может ли страус летать» требует меньшего времени для ответа, чем вопросы «Имеет ли страус крылья?» или «Может ли страус дышать?».

Упомянутое выше свойство наследования нашло свое отражение в объектно-ориентированном программировании.

К примеру, необходимо создать программу, содержащую описание классов Работник (**Employee**) и Клиент (**Customer**). Эти классы имеют общие свойства, присущие всем людям, поэтому создадим *базовый* класс Человек (**Person**) и наследуем от него *дочерние* классы **Employee** и **Customer** (рис. 4.43).

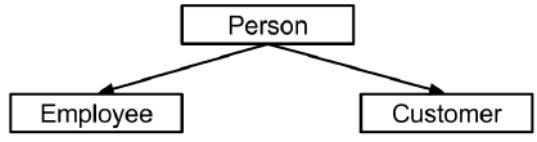


Рис. 4.43. **Иерархия классов**

Код, описывающий иерархию классов, представлен ниже (**person.py**):

```
class Person():
    name = ""       # имя человека
class Employee(Person): # указываем базовый класс Person
    job_title = ""   # наименование должности работника
class Customer(Person): # указываем базовый класс Person
    email = ""        # почта клиента
```

Создадим объекты на основе классов и заполним их поля:

```
person1 = Person()       # создаем объект класса Person
person1.name = "Иван Петров"    # заполняем поле объекта

personEmployee = Employee() # создаем объект класса
                            # Employee
# поле name унаследовано от класса Person:
personEmployee.name = "Петр Иванов"
personEmployee.job_title = "Программист"

personCustomer = Customer()
personCustomer.name = "Петр Петров"
personCustomer.email = "me@me.ru"
```

В объектах классов **Employee** и **Customer** появилось поле **name**, унаследованное от класса **Person**.

Результат работы программы **person.py**:

```
Иван Петров Петр Иванов Программист Петр Петров me@me.ru
```

Далее представлен пример наследования методов:

```
# person1.py
class Person():
    name = ""
    def __init__(self): # конструктор базового класса
        print("Создан человек")

class Employee(Person):
    job_title = ""
```

```
class Customer(Person):
    email = ""

person1 = Person()
personEmployee = Employee()
personCustomer = Customer()
```

Результат выполнения программы:

```
Создан человек
Создан человек
Создан человек
```

Таким образом, при создании объектов вызывается конструктор, унаследованный от базового класса. Если дочерние классы содержат собственные конструкторы, то выполняться будут они. Пример создания собственных конструкторов для дочерних классов:

```
# person2.py
class Person():
    name = ""
    def __init__(self):  # конструктор базового класса
        print("Создан человек")

class Employee(Person):
    job_title = ""
    def __init__(self): # конструктор дочернего класса
        print("Создан работник")

class Customer(Person):
    email = ""
    def __init__(self):  # конструктор дочернего класса
        print("Создан покупатель")

person1 = Person()
personEmployee = Employee()
personCustomer = Customer()
```

Результат работы программы:

```
Создан человек
Создан работник
Создан покупатель
```

Видим, что в момент создания объекта вызывается конструктор, содержащийся в дочернем классе, т.е. конструктор базового класса был *переопределен* в дочерних классах. Порой на практике требуется вызвать конструктор базового класса из конструктора дочернего класса:

```
# person3.py
class Person():
    name = ""
    def __init__(self):
        print("Создан человек")
```

```
class Employee(Person):
    job_title = ""
    def __init__(self):
        Person.__init__(self) # вызываем конструктор
                              # базового класса
        print("Создан работник")

class Customer(Person):
    email = ""
    def __init__(self):
         Person.__init__(self) # вызываем конструктор
                               # базового класса
         print("Создан покупатель")

person1 = Person()
personEmployee = Employee()
personCustomer = Customer()
```

Результат работы программы:

```
Создан человек
Создан человек
Создан работник
Создан человек
Создан покупатель
```

4.28. Разработка приложений с графическим интерфейсом

4.28.1. Основы работы с модулем tkinter

Язык Python позволяет создавать приложения с графическим интерфейсом, для этого применяются различные графические библиотеки[1]. Остановимся на рассмотрении стандартной графической библиотеки **tkinter**[2] (входит в стандартную поставку Python, доступную с официального сайта **python.org**).

Первым делом при работе с **tkinter** необходимо создать главное (*корневое*) окно (рис. 4.44), в котором размещаются остальные графические элементы — *виджеты*. Существует большой набор виджетов[3] на все случаи жизни: для ввода текста, вывода текста, выпадающие меню и т.д. Среди виджетов есть кнопка, при нажатии на которую происходит заданное событие. Некоторые виджеты (*фреймы*) используются для группировки других виджетов внутри себя.

Приведем пример простейшей программы для отображения главного окна:

---

[1] Подробнее см.: https://wiki.python.org/moin/GuiProgramming.

[2] Подробнее см.: https://docs.python.org/3/library/tkinter.html.

[3] Подробнее см.: http://effbot.org/tkinterbook/tkinter-index.htm.

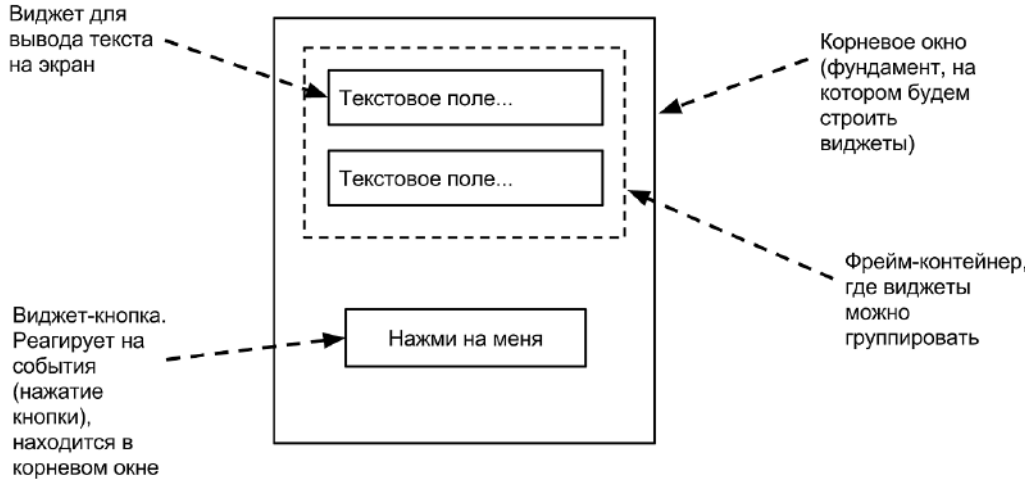


Рис. 4.44. **Схема главного окна в tkinter**

```
# mytk1.py
# Подключаем модуль, содержащий методы для работы
# с графикой
import tkinter
# Создаем главное (корневое) окно,
# в переменную window записываем ссылку на объект
# класса Tk
window = tkinter.Tk()
# Задаем обработчик событий для корневого окна
window.mainloop()
```

Результат выполнения программы показан на рис. 4.45.

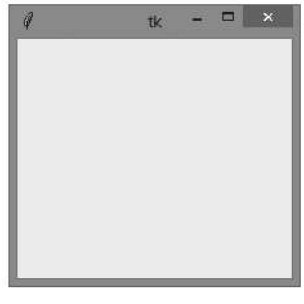


Рис. 4.45. **Главное окно**

С помощью трех строчек кода на языке Python удалось вывести на экран полноценное окно, которое можно свернуть, растянуть или закрыть.

Графические (оконные) приложения отличаются от консольных (без оконных) наличием обработки событий. Для консольных (скорее, интерактивных) приложений, с которыми мы работали ранее, не требовалось определять, какую кнопку мыши и в какой момент времени нажал пользователь программы. В оконных приложениях важны все манипуляции с мышью и клавиатурой, так как от этого зависит, например, какой пункт меню выберет пользователь.

Слева на рис. 4.46 показан алгоритм работы консольной программы, справа — программы с графическим интерфейсом.

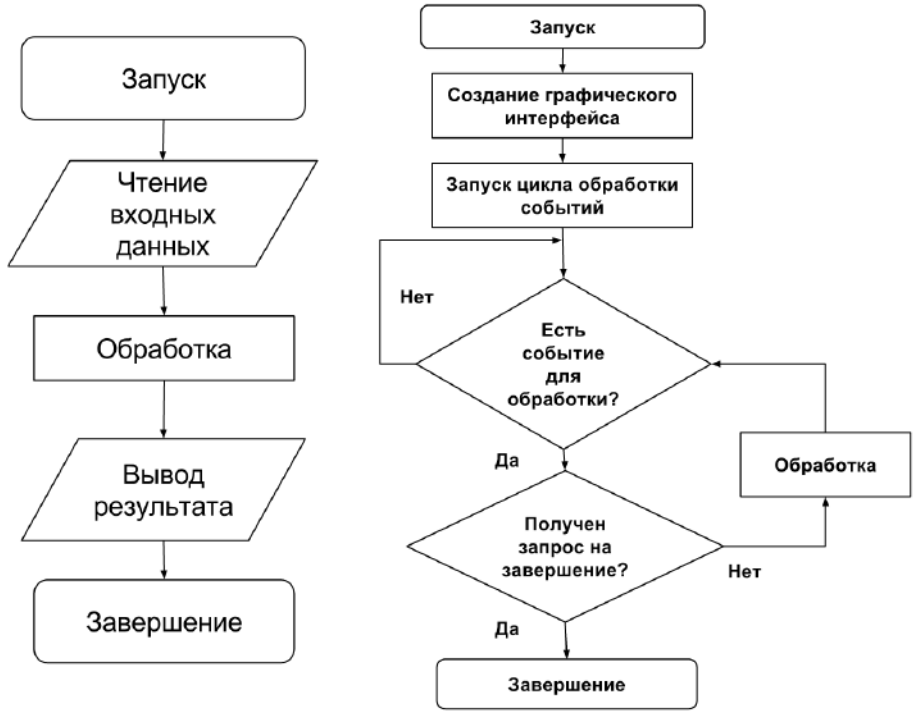


Рис. 4.46. **Схема работы консольной программы и программы с графическим интерфейсом**

Следующий пример демонстрирует создание виджета **Label**:

```
# mytk2.py
import tkinter
window = tkinter.Tk()
# Создаем объект-виджет класса Label в корневом окне window
# text - параметр для задания отображаемого текста
label = tkinter.Label(window, text = "Это текст в окне!")
# Отображаем виджет с помощью менеджера pack
label.pack()
window.mainloop()
```

В результате работы программы отображается графическое окно с текстом внутри (рис. 4.47).

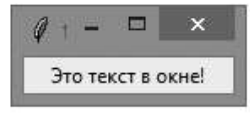


Рис. 4.47. **Текст в окне**

Далее представлен пример размещения виджетов во фрейме:

```
# mytk3.py
import tkinter
window = tkinter.Tk()
# Создаем фрейм в главном окне
```

```
frame = tkinter.Frame(window)
frame.pack()
# Создаем виджеты и помещаем их во фрейме frame
first = tkinter.Label(frame, text='First label')
# Отображаем виджет с помощью менеджера pack
first.pack()
second = tkinter.Label(frame, text='Second label')
second.pack()
third = tkinter.Label(frame, text='Third label')
third.pack()
window.mainloop()
```

Пример выполнения программы показан на рис. 4.48.

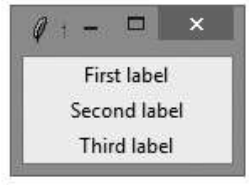


Рис. 4.48. **Размещение виджетов во фрейме**

Можно изменять параметры фрейма в момент создания объекта[1]:

```
# mytk4.py
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
# Можем изменять параметры фрейма:
frame2 = tkinter.Frame(window, borderwidth=4,
    relief=tkinter.GROOVE)
frame2.pack()
# Размещаем виджет в первом фрейме (frame)
first = tkinter.Label(frame, text='First label')
first.pack()
# Размещаем виджеты во втором фрейме (frame2)
second = tkinter.Label(frame2, text='Second label')
second.pack()
third = tkinter.Label(frame2, text='Third label')
third.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 4.49.

Рис. 4.49. **Изменение параметров фрейма при создании объекта**

[1] Подробнее см.: http://effbot.org/tkinterbook/frame.htm.

В следующем примере для отображения в виджете **Label** содержимого переменной используется переменная **data** класса **StringVar** (из модуля **tkinter**). В дальнейшем из примеров станет понятнее, почему в **tkinter** используются переменные собственного класса[1].

```
# mytk5.py
import tkinter
window = tkinter.Tk()
# Создаем объект класса StringVar
# (создаем строковую переменную, с которой умеет
# работать tkinter):
data = tkinter.StringVar()
# Метод set позволяет изменить содержимое переменной:
data.set('Данные в окне')
# связываем текст для виджета Label с переменной data
label = tkinter.Label(window, textvariable = data)
label.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 4.50.

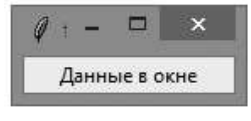


Рис. 4.50. **Отображение переменной в окне**

4.28.2. Шаблон «Модель — Вид — Контроллер» на примере модуля tkinter

Следующий пример показывает, каким образом использовать виджет **(Entry)** для ввода данных:

```
# mytk6.py
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
var = tkinter.StringVar()
# Обновление содержимого переменной в момент ввода текста
label = tkinter.Label(frame, textvariable=var)
label.pack()
# Пробуем набрать текст в появившемся поле для ввода
entry = tkinter.Entry(frame, textvariable=var)
entry.pack()
window.mainloop()
```

Запустим программу и внутри текстового окна попробуем набрать произвольный текст (рис. 4.51).

[1] Tkinter поддерживает работу с переменными классов: BooleanVar, DoubleVar, IntVar, StringVar.

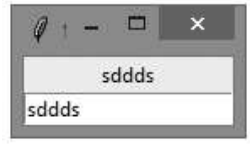


Рис. 4.51. **Виджет для ввода данных**

Видим, что текст, который мы набираем, сразу отображается в окне. Дело в том, что виджеты **Label** и **Entry** используют для вывода и ввода текста одну и ту же переменную **data** класса **StringVar**. Подобная схема работы оконного приложения укладывается в универсальный шаблон (паттерн) MVC. В общем виде под *моделью* (Model) понимают способ хранения данных (например, в переменной какого класса). *Вид* (View) служит для отображения данных. *Контроллер* (Controller) отвечает за обработку данных (рис. 4.52).

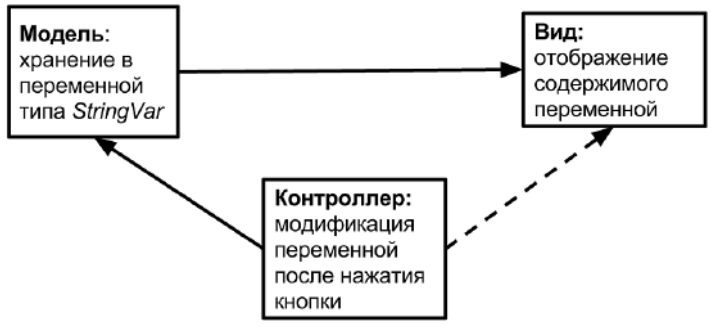


Рис. 4.52. **Схема шаблона MVC**

Особенностью шаблона MVC является то, что в случае изменения контроллером данных (как это было в предыдущем примере с изменением переменной **var**) «посылается сигнал» *виду* с просьбой обновить отображаемое содержимое (перерисовать окно), отсюда возникает обновление текста в режиме реального времени. Следующий пример демонстрирует возможности обработки событий при нажатии на кнопку (виджет **Button**):

```
# mytk7.py
import tkinter

# Контроллер: функция вызывается в момент нажатия
# на кнопку
def click():
    # метод get возвращает текущее значение counter
    # метод set устанавливает новое значение counter
    counter.set(counter.get() + 1)

window = tkinter.Tk()
# Модель: создаем объект класса IntVar
counter = tkinter.IntVar()
# Обнуляем созданный объект с помощью метода set
counter.set(0)
```

```
frame = tkinter.Frame(window)
frame.pack()
# Создаем кнопку и указываем обработчик (функция click)
# при нажатии на нее
button = tkinter.Button(frame, text='Click',
    command=click)
button.pack()
# Вид: в реальном времени обновляется содержимое
# виджета Label
label = tkinter.Label(frame, textvariable=counter)
label.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 4.53.

Рис. 4.53. **Обработка события при нажатии на кнопку**

Далее представлен более сложный пример с двумя кнопками и двумя обработчиками событий (**click_up**, **click_down**):

```
# mytk8.py
import tkinter
window = tkinter.Tk()
# Модель:
counter = tkinter.IntVar()
counter.set(0)

# Два контроллера:
def click_up():
    counter.set(counter.get() + 1)
def click_down():
    counter.set(counter.get() - 1)

# Вид:
frame = tkinter.Frame(window)
frame.pack()
button = tkinter.Button(frame, text='Up',
    command=click_up)
button.pack()
button = tkinter.Button(frame, text='Down',
    command=click_down)
button.pack()
label = tkinter.Label(frame, textvariable=counter)
label.pack()
window.mainloop()
```

Результат работы программы показан на рис. 4.54.

Рис. 4.54. **Пример с двумя кнопками и двумя обработчиками событий**

4.28.3. Изменение параметров по умолчанию при работе с tkinter

Tkinter позволяет изменять параметры виджетов в момент их создания:

```
# mytk9.py
import tkinter
window = tkinter.Tk()
# Создаем кнопку, изменяем шрифт с помощью кортежа
button = tkinter.Button(window, text='Hello',
                        font=('Courier', 14, 'bold italic'))
button.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 4.55.

Рис. 4.55. **Изменение шрифта при создании виджета**

В следующем примере изменяются параметры виджета **Label**:

```
# mytk10.py
import tkinter
window = tkinter.Tk()
# Изменяем фон, цвет текста:
button = tkinter.Label(window, text='Hello', bg='green',
    fg='white')
button.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 4.56.

Рис. 4.56. **Изменение параметров виджета Label**

Менеджер расположения (*геометрии*) **pack** тоже имеет свои параметры. Далее представлен исходный код, демонстрирующий изменение параметров менеджера геометрии (рис. 4.57):

```
# mytk11.py
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text='Name')
# Выравнивание по левому краю
label.pack(side = 'left')
entry = tkinter.Entry(frame)
entry.pack(side='left')
window.mainloop()
```

Рис. 4.57. **Изменение параметров менеджера геометрии**

Особенность следующего примера в том, что введенный текст (через виджет **Entry**) отображается на экране (через виджет **Label**) только в момент нажатия кнопки (виджет **Button**), а не в реальном времени, как это было ранее (рис. 4.58).

```
# mytk12.py
import tkinter
# Вызывается в момент нажатия на кнопку:
def click():
    # Получаем строковое содержимое поля ввода и
    # с помощью config изменяем отображаемый текст
    label.config(text=entry.get())

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
entry = tkinter.Entry(frame)
entry.pack()
label = tkinter.Label(frame)
label.pack()
# Привязываем обработчик нажатия на кнопку
# к функции click
button = tkinter.Button(frame, text='Печать!',
command=click)
button.pack()
window.mainloop()
```

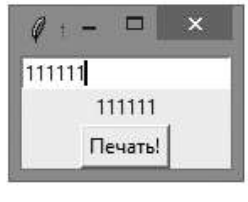


Рис. 4.58. **Вывод текста при нажатии на кнопку**

4.29. Реализация алгоритмов

Предположим, необходимо найти позицию наименьшего элемента в следующем наборе данных: 809, 834, 477, 478, 307, 122, 96, 102, 324, 476.

Первым делом выбираем подходящий для хранения тип данных. Очевидно, что это будет список:

```
>>> counts = [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
>>> counts.index(min(counts))
# решение задачи в одну строку!
6
>>>
```

Усложним задачу и попытаемся найти позицию двух наименьших элементов в неотсортированном списке. Возможные алгоритмы решения:

1) *поиск, удаление, поиск*. Поиск индекса минимального элемента в списке, удаление его, снова поиск минимального, возвращаем удаленный элемент в список;

2) *сортировка, поиск минимальных, определение индексов*;

3) *перебор всего списка*. Сравниваем каждый элемент по порядку, получаем два наименьших значения, обновляем значения, если найдены наименьшие.

Рассмотрим каждый из перечисленных алгоритмов.

1. **Поиск, удаление, поиск**: поиск индекса минимального элемента в списке, удаление его, снова поиск минимального, возвращение удаленного элемента обратно в список.

Начнем:

```
          [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
индекс:    0    1    2    3    4    5   6   7    8    9
```

Первый минимальный: 96.

Индекс элемента 96: 6.

Удаляем из списка найденный минимальный элемент **(96)**, при этом индексы в обновленном списке смещаются:

```
          [809, 834, 477, 478, 307, 122,     102, 324, 476]
индекс:    0    1    2    3    4    5       6    7    8
```

Второй минимальный: 102.

Индекс элемента 102: 6.

Возвращаем удаленный (первый минимальный) элемент обратно в список:

```
          [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
индекс:    0    1    2    3    4    5   6   7    8    9
```

Не забываем о смещении индексов после удаления первого минимального элемента: индекс второго минимального элемента ра-

вен индексу первого минимального элемента, поэтому увеличиваем индекс второго минимального на **1**. Функция, реализующая данный алгоритм:

```
def find_two_smallest(L):
  smallest = min(L)
  min1 = L.index(smallest)
  L.remove(smallest)
  # удаляем первый минимальный элемент
  next_smallest = min(L)
  min2 = L.index(next_smallest)
  L.insert(min1, smallest)
  # возвращаем первый минимальный
  # проверяем индекс второго минимального из-за смещения:
  if min1 <= min2:
    min2 += 1     # min2 = min2 + 1
  return (min1, min2)   # возвращаем кортеж
```

2. **Сортировка, поиск минимальных, определение индексов:** реализация второго алгоритма интуитивно понятна, поэтому приведем только исходный текст функции:

```
def find_two_smallest(L):
    temp_list = sorted(L)
    smallest = temp_list[0]
    next_smallest = temp_list[1]
    min1 = L.index(smallest)
    min2 = L.index(next_smallest)
    return (min1, min2)
```

3. **Перебор всего списка:** сравниваем каждый элемент по порядку, получаем два наименьших значения, обновляем значения, если найдены наименьшие.

Третий алгоритм наиболее сложный из перечисленных выше, поэтому остановимся на нем подробнее.

В отличие от человека, который может охватить взглядом сразу весь список и интуитивно сказать, какой из элементов является минимальным, компьютер не обладает подобным интеллектом. Он просматривает элементы по одному, последовательно перебирая их и сравнивая.

На первом шаге рассматриваются первые два элемента списка.

```
          [809, 834, ...]
индекс:     0     1
```

Сравниваем **809** и **834** и определяем наименьший из них, чтобы задать начальные значения **min1** и **min2**, где будут храниться *индексы* первого минимального и второго минимального элементов соответственно. Затем перебираем элементы, начиная с индекса **2** до окончания списка.

Определили, что **809** — первый минимальный, а **834** — второй минимальный элемент из двух первых встретившихся значений списка. Рассматриваем следующий элемент списка (**477**).

Элемент **477** оказался наименьшим (назовем это «первым случаем»). Поэтому обновляем содержимое переменных **min1** и **min2**, так как нашли новый наименьший элемент.

Рассматриваем следующий элемент списка (**478**). Он оказался между двумя минимальными элементами (назовем это «вторым случаем»).

Снова обновляем минимальные элементы (теперь обновился только **min2**), и т.д. пока не дойдем до конца списка.

Далее представлен исходный текст функции, реализующий предложенный алгоритм:

```
def find_two_smallest(L):
    if L[0] < L[1]:
        min1, min2 = 0, 1
        # устанавливаем начальные значения
    else:
        min1, min2 = 1, 0

    for i in range(2, len (L)):
        if L[i] < L[min1]:  # «первый случай»
            min2 = min1
            min1 = i
        elif L[i] < L[min2]:  # «второй случай»
            min2 = i
    return (min1, min2)
```

Контрольные вопросы и задания

1. Каковы сильные и слабые стороны языка программирования Python?
2. Какие правила наименования переменных в Python существуют? Опишите модель памяти Python при работе с переменными.
3. Опишите процесс создания функций в Python.
4. Какие отличия между выполнением команд в файле от выполнения в интерактивном режиме?
5. Какие существуют операции над строками в языке Python?
6. Какие существуют операторы отношений в Python? Перечислите правила логических операций над объектами.
7. В каких случаях применяется условная инструкция if?
8. Что такое модуль в Python?
9. Опишите процесс создания собственных модулей в Python.
10. Какие существуют строковые методы в Python? В чем отличие функций от методов?
11. Что такое список в Python? Опишите процесс создания списка.
12. Перечислите основные операции над списками в Python.

13. Что такое пседонимы? В чем заключается клонирование списков в Python?

14. Перечислите основные методы списка в Python.

15. Приведите примеры преобразования типов в Python (списки, строки).

16. Опишите возможности применения вложенных списков в Python.

17. Какие циклы существуют в Python?

18. В каких случаях применяется цикл for (на примере списков и строк)?

19. В каких случаях используется функция range в Python?

20. Перечислите способы генерации списка в Python.

21. В каких случаях применяется цикл while в Python?

22. Опишите область применения вложенных циклов в Python (на примере вложенных списков).

23. Что такое множество? Какие операции существуют над множествами в Python?

24. Что такое кортеж? Какие операции над кортежами существуют в Python?

25. Что такое словарь? Какие операции над словарями существуют в Python?

26. Как происходит обработка исключений в Python?

27. Какие особенности объектно-ориентированного программирования существуют в Python? Что такое классы, объекты?

28. Опишите структуру оконного приложения на примере модуля tkinter.

29. Что такое шаблон «Модель-вид-контроллер» (на примере модуля tkinter)?

Задания для самостоятельного выполнения

1. Найдите значение выражения $2 + 56 \cdot 5.0 - 45.5 + 5^5$.

2. Найдите значения для следующих выражений:

```
pow(abs(-5) + abs(-3), round(5.8))
int(round(pow(round(5.777, 2), abs(-2)), 1))
```

3. Создайте собственные функции для вычисления следующих выражений:

а) $x^4 + 4^x$;

б) $y^4 + 4^x$.

4. Создайте в отдельном файле функцию, переводящую градусы по шкале Цельсия T_C в градусы по шкале Фаренгейта T_F по формуле:

$$T_F = 9/5 \cdot T_C + 32.$$

5. Напишите в отдельном файле функцию, вычисляющую среднее арифметическое трех чисел. Задайте значения по умолчанию, в момент вызова используйте ключевые аргументы.

6. Попросите пользователя ввести свое имя и после этого отобразите на экране строку вида: «Привет, <имя>!» Вместо <имя> должно указываться то, что пользователь ввел с клавиатуры.

Пример работы программы:

```
Как тебя зовут?
Вася
Привет, Вася!
```

7. Напишите программу, определяющую сумму и произведение трех чисел (типа int, float), введенных с клавиатуры.

Пример работы программы:

```
Введите первое число: 1
Введите второе число: 4
Введите третье число: 7
Сумма введенных чисел: 12
Произведение введенных чисел: 28
```

8. Напишите собственную программу, определяющую максимальное из двух введенных чисел. Реализовать в виде вызова собственной функции, возвращающей большее из двух переданных ей чисел.

9. Напишите программу, проверяющую целое число на четность. Реализовать в виде вызова собственной функции.

10. Напишите программу, которая по коду города и длительности переговоров вычисляет их стоимость и выводит результат на экран: Екатеринбург — код 343, 15 руб/мин; Омск — код 381, 18 руб/мин; Воронеж — код 473, 13 руб/мин; Ярославль — код 485, 11 руб/мин.

11. Найдите площадь треугольника с помощью формулы Герона. Стороны задаются с клавиатуры. Реализовать вычисление площади в виде функции, на вход которой подаются три числа, на выходе возвращается площадь. Функция находится в отдельном модуле, где происходит разделение между запуском и импортированием.

12. Напишите программу-игру в виде отдельного модуля. Компьютер загадывает случайное число, пользователь пытается его угадать. Программа запрашивает число ОДИН раз. Если число угадано, то выводим на экран «Победа», иначе — «Повторите еще раз»! Для написания программы понадобится функция **randint()** из модуля **random**[1].

13. Напишите функцию, вычисляющую значение: $x^4 + 4^x$. Автоматизируйте процесс тестирования функции с помощью модуля **doctest**.

14. Задана строка s = "У лукоморья 123 дуб зеленый 456":

1) определить, встречается ли в строке буква 'я'. Вывести на экран ее позицию (индекс) в строке;

2) определить, сколько раз в строке встречается буква 'у';

3) определить, состоит ли строка только из букв, ЕСЛИ нет, ТО вывести строку в верхнем регистре;

4) определить длину строки. ЕСЛИ длина строки превышает четыре символа, ТО вывести строку в нижнем регистре;

5) заменить в строке первый символ на 'О'. Результат вывести на экран.

15. Написать в отдельном модуле функцию, которая на вход принимает два аргумента: строку (s) и целочисленное значение (n). ЕСЛИ длина

[1] https://docs.python.org/3/library/random.html.

строки s превышает n символов, ТО функция возвращает строку s в верхнем регистре, ИНАЧЕ возвращается исходная строка s.

16. Дан список L = [3, 6, 7, 4, –5, 4, 3, –1]:

1) определите сумму элементов списка L. ЕСЛИ сумма превышает значение 2, ТО вывести на экран число элементов списка;

2) определите разность между минимальным и максимальным элементами списка. ЕСЛИ абсолютное значение разности больше 10, ТО вывести на экран отсортированный по возрастанию список, ИНАЧЕ вывести на экран фразу 'Разность меньше 10'.

17. Дан список L = [3, 'hello', 7, 4, 'привет', 4, 3, –1]. Определите наличие строки 'привет' в списке. ЕСЛИ такая строка в списке присутствует, ТО вывести ее на экран, повторив 10 раз.

18. Дан список L = [3, 'hello', 7, 4, 'привет', 4, 3, –1]. Определите наличие строки 'привет' в списке. ЕСЛИ такая строка в списке присутствует, ТО удалить ее из списка, ИНАЧЕ добавить строку в список. Подсчитать, сколько раз в списке встречается число 4, ЕСЛИ больше одного раза, ТО очистить список.

19. Напишите программу, которая запрашивает у пользователя две строки и формирует из этих строк список. Если строки состоят только из чисел, то программа добавляет в середину списка сумму введенных чисел, иначе добавляется строка, образованная из слияния двух введенных ранее строк. Итоговая строка выводится на экран.

20. Задан список слов. Необходимо выбрать из него случайное слово. Из выбранного случайного слова случайно выбрать букву и попросить пользователя ее угадать. Пример работы программы:

- задан список слов: ['самовар', 'весна', 'лето'];
- выбираем случайное слово: 'весна';
- выбираем случайную букву: 'с';
- выводим на экран: ве?на.

Пользователь пытается угадать букву.

Подсказка: используйте метод *choice()* модуля *random*.

21. Найдите все значения функции $y(x) = x^2 + 3$ на интервале от 10 до 30 с шагом 2.

22. Дан список L = [–8, 8, 6.0, 5, 'строка', –3.1]. Определить сумму чисел, входящих в список L.

Подсказка: для определения типа объекта можно воспользоваться сравнением вида: type(–8) == int.

23. Напишите программу-игру. Компьютер загадывает случайное число, пользователь пытается его угадать. Пользователь вводит число до тех пор, пока не угадает или не введет слово 'Выход'. Компьютер сравнивает число с введенным и сообщает пользователю, больше оно или меньше загаданного.

24. Дано число, введенное с клавиатуры. Определите сумму квадратов нечетных цифр в числе.

25. Найдите сумму чисел, вводимых с клавиатуры. Количество вводимых чисел заранее неизвестно. Окончание ввода, например, слово 'Стоп'.

26. Дан произвольный текст. Найдите номер первого самого длинного слова в нем.

27. Дан произвольный текст. Напечатайте все имеющиеся в нем цифры, определите их количество, сумму и найдите максимальное.

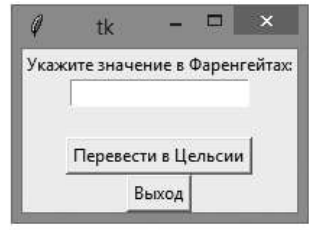
28. Напишите функцию, которая возвращает разность между наибольшим и наименьшим значениями из списка целых случайных чисел.

29. Напишите программу, проверяющую четность числа, вводимого с клавиатуры. Выполните обработку возможных исключений.

30. Создайте класс **StringVar** для работы со строковым типом данных, содержащий методы **set()** и **get()**. Метод **set()** служит для изменения содержимого строки класса **StringVar**, **get()** — для получения содержимого строки класса **StringVar**. Создайте объект типа **StringVar** и протестируйте его методы.

31. Создайте класс точки **Point**, позволяющий работать с координатами (x, y). Добавьте необходимые методы класса.

32. Напишите оконное приложение, позволяющее переводить градусы по шкале Фаренгейта в градусы по шкале Цельсия:



Глава 5
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ С

В результате изучения гл. 5 обучающиеся должны:

знать

- структуру программы;
- основные типы данных, их особенности;
- стандартные библиотеки языка;

уметь

- выполнять стандартные операции над данными различного типа;
- работать с указателями;

владеть

- навыками обработки статических и динамических массивов;
- навыками создания динамических структур.

5.1. Структура программы

Язык программирования С был создан в 1972 г. Деннисом Ритчи. В языке С сочетаются лучшие свойства языка ассемблера и языков высокого уровня. Наличие большого набора управляющих конструкций, возможность обработки сложных структур данных, присущие языкам высокого уровня, дополняются гибкими средствами ввода/вывода и работы с памятью, характерными для языка ассемблера.

Программа на С состоит из нескольких файлов двух типов: файлов заголовков (с расширением «.h») и файлов кодов (с расширением «.cpp»).

Файлы заголовков содержат общие составные части программы, например, директивы препроцессора для подключения заголовочных файлов используемых библиотек языка, объявления типов, определяемых пользователем, определения именных констант, прототипы функций (заголовки функций без тела).

Файлы кодов содержат следующие составные части: определения функций (тексты функций), директивы препроцессора для

подключения файлов заголовков, объявления глобальных переменных. В одном из файлов кодов находится главная функция **main()**, с которой начинается выполнение программы.

Большим преимуществом языка С является его расширяемость, т.е. возможность внесения изменений в язык. Это достигается благодаря тому, что язык С содержит небольшое число операторов и большой набор различных библиотек функций.

Расширения языка можно разбить на несколько групп:

- первая группа размещается в стандартной библиотеке;
- расширения, относящиеся ко второй группе, поддерживаются конкретным компилятором;
- третья группа разрабатывается самими программистами и содержится в дополнительных библиотеках функций С.

Большинство функций, являющихся составной частью языка С (функции ввода-вывода, математические операции, операции работы со строками и т.д.), стандартизированы подкомитетом Американского национального института стандартов (ANSI). В зависимости от компилятора могут существовать различия в перечне доступных функций и в выполняемых ими действиях.

Общая структура программы на языке С представлена на рис. 5.1.

Итак, программа на языке С состоит из одного или нескольких модулей.

Модуль — это самостоятельно компилируемый файл (после компиляции он превращается в объектный модуль с расширением «obj»). В свою очередь модуль состоит из одной или нескольких

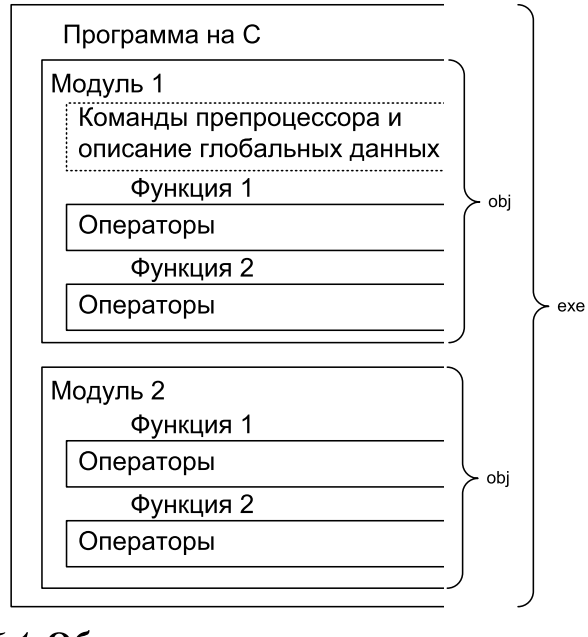


Рис. 5.1. **Общая структура программы на языке С**

функций, а также может включать команды препроцессора и описания глобальных данных, которые будут рассмотрены позже.

Модули одной программы неравноценны: один из них должен содержать функцию **main()**, а остальные — любые другие функции.

Модули объединяются в единый модуль при компоновке программы (выполняемый модуль «exe»).

Функция — это самостоятельный фрагмент исходного текста программы, предназначенный для конкретной задачи. В языке С функции являются основными строительными блоками, из которых составляется программа. Функции позволяют избавиться от повторного программирования в случае частого выполнения одних и тех же действий. Функции состоят из операторов языка С.

Алфавит языка программирования С основывается на множестве символов таблицы кодов ASCII и включает:

- строчные и прописные буквы латинского алфавита;
- цифры от 0 до 9;
- символ подчеркивания «_»;
- набор специальных символов: " { } , | [ ] + - % / \ ; ' : ? <> = ! & # ~ ^ . * ;
- прочие символы.

Составные части программы строятся с использованием базовых элементов языка — *лексем*, к которым относятся ключевые слова, идентификаторы, константы, символы операций и пунктуации.

Лексемы разделяются *разделителями*, к которым относятся:

- пробельные символы (пробел, символы табуляции, символ новой строки, комментарии);
- последовательности специальных символов: ... ; {};
- большинство символов операций, в зависимости от контекста.

Комментарии — это тексты, предназначенные для аннотирования программы. Существуют два способа обозначения комментария:

- с символов // и до конца строки;
- с символов /* и до символов */.

Предложения в С называются *операторами*. Оператор языка С состоит из выражений и может содержать вложенные операторы.

5.2. Константы и переменные

Константы (литералы) в языке С могут быть следующих типов:

- целочисленные;
- вещественные;
- символьные;
- строковые.

Целочисленные константы служат для хранения целочисленных значений. Такие значения в языке С могут представляться в десятичной, восьмеричной и шестнадцатеричной системах счисления. *Десятичные* константы — любые числа без дробной части, не начинающиеся с нуля. *Восьмеричные* целые константы начинаются с нуля. *Шестнадцатеричные* константы начинаются с 0x или с 0X. Непосредственно за константой могут располагаться в произвольном сочетании один или два специальных суффикса: **U** (или **u**), означающий **unsigned** — беззнаковый; и **L** (или **l**), означающий **long**. Например:

- 23 — десятичная константа;
- 023 — восьмеричная константа;
- 0x23 — шестнадцатеричная константа;
- 23U — десятичная константа без знака;
- 0x23L — шестнадцатеричная константа в формате длинного целого.

Вещественная константа используется для задания вещественных значений. Такая константа может быть записана в обычной десятичной или экспоненциальной (научной) форме. В последнем случае число записывается в виде мантиссы и порядка. Мантисса отделяется от порядка символом **E (e)**. Непосредственно за константой могут располагаться один из двух специальных суффиксов: **F (f)**, означающий **float**; и **L** (или **l**), означающий **long**. Например:

- 10.25 — вещественное число 10,25;
- 1025. — вещественное число 1025;
- 1.025E-3 — вещественное число 0,001025;
- 10.25f — вещественное число в формате с одинарной точностью;
- 10.25l — вещественное число высокой точности.

Символьная константа — это последовательность из одного или нескольких символов, заключенных в одинарные кавычки (апострофы). Символьные константы имеют числовые значения, равные кодам ASCII соответствующих символов. Любой символ при этом может быть представлен в нескольких форматах: обычном, восьмеричном и шестнадцатеричном.

Символьные константы могут содержать *управляющие последовательности* для преобразования определенных символов, не имеющих графического аналога. Перечень таких констант приведен в табл. 5.1.

Например:

- 'a' — символ a;
- 'A' — символ A;
- '\127' — символ, имеющий код 127 (u);
- '\x7F' — символ, имеющий код 7F в шестнадцатеричном представлении (u);
- '\\' — символы //.

Таблица 5.1

Символьные константы, не имеющие графического представления

Символ	Шестнадцатеричное представление	Имя	Описание
\0	\x00	null	Пустой символ
\a	\x07	bel	Звуковой сигнал (звонок)
\b	\x08	bs	Возврат на шаг (забой)
\f	\x0C	ff	Перевод страницы
\n	\x0A	lf	Перевод строки
\r	\x0D	cr	Возврат каретки
\t	\x09	ht	Горизонтальная табуляция
\v	\x0B	vt	Вертикальная табуляция
\\	\x5C	\	Обратная косая черта (обратный слэш)
\'	\x27	'	Одинарная кавычка
\"	\x22	"	Двойная кавычка
\?	\x3F	?	Вопросительный знак

Строковая константа — это ноль и более символов, заключенных в двойные кавычки, имеет тип массив символов (**char**). Во внутреннем представлении она содержит в конце добавляемый автоматически символ ноль '**\0**', являющийся индикатором конца константы. Нулевая (пустая строка) записывается в виде "" и хранится в виде одного символа '**\0**'. Константы могут быть записаны в нескольких строках. Последний символ переносимой строки — это \. Этот символ и символ новой строки отбрасываются при выводе данных. Соседние строковые константы, разделенные только пробелами, конкатенируются (объединяются). Например:

- "Строка" — константа «Строка»;
- "Часть1""Часть2" — константа «Часть1Часть2»;
- "Часть1\13\x0AЧасть2" — константа «Часть1
  Часть2».

Переменная — это поименованная область памяти, имеющая определенное значение и тип.

Тип является основной характеристикой переменной. Он определяет, что и как следует делать со значениями переменных: определяет структуру и размеры переменных, диапазон и способы интерпретации их значений, множество допустимых операций.

Типы данных в С делятся на основные (стандартные) и производные.

Стандартные (базовые) типы включают:

- целочисленные;
- вещественные;
- символьные.

Помимо стандартных типов в языке С существуют возможность использовать *производные* типы данных, создаваемые из элементов основных. Эти типы данных будут рассмотрены ниже.

Для базовых типов используются стандартные идентификаторы типов, приведенные в табл. 5.2.

Пара модификаторов **signed** (знаковое) и **unsigned** (беззнаковое) указывает на то, что переменная может принимать как положительное, так и отрицательное значение (**signed**) или только не отрицательное (**unsigned**). В первом случае самый левый бит памяти, выделяемой под число, отводится для хранения знака числа, ос-

Таблица 5.2

Стандартные идентификаторы типов данных

Тип данных		Отводимая память (в байтах)	Диапазон хранимых значений	
Название	Идентификатор		Min	Max
Символ	signed char	1	–128	127
Беззнаковый символ	unsigned char	1	0	255
Короткое целое	signed short	2	–32768	32767
Неотрицательное короткое целое	unsigned short	2	0	65535
Целое	signed int	4	–2147483648	2147483647
Неотрицательное целое	unsigned int	4	0	4294967295
Длинное целое	signed long	4	–2147483648	2147483647
Неотрицательное длинное целое	unsigned long	4	0	4294967295
Вещественное одинарной точности	float	4	–3.4E+38	3.4E+38
Вещественное двойной точности	double	8	–1.7E+308	1.7E+308
Вещественное высокой точности	long double	10	–3.4E+4932	3.4E+4932

тальные — для значения числа; во втором — все биты отводятся под хранение значения числа.

Параметры типов **signed int** и **unsigned int** зависят от конкретной реализации компилятора и целевой платформы: для 32-разрядной платформы они эквивалентны соответствующим параметрам типа **long**.

Тип данных определяет перечень операций, которые могут выполняться над переменной, и размер памяти, выделяемой для работы с этой переменной.

Выделение памяти означает предоставление определенного участка памяти под хранение переменной. При создании переменной выделение памяти производится автоматически.

Наряду с выделением памяти существует процесс *освобождения памяти,* т.е. указание на то, что участок памяти, ранее занимаемый переменной, теперь становится свободным.

Выделение памяти происходит при объявлении переменной. *Объявление переменной* содержит тип данных и имя переменной:

```
<тип данных><имя переменной>
```

Например:

```
int k; //объявлена переменная k целого типа
float p; // объявлена переменная p вещественного типа
```

В соответствии с указанным типом под переменную будет выделен участок памяти определенного размера.

При объявлении переменной может быть выполнена ее инициализация, т.е. присвоение переменной значения. Например, в следующей строке объявлены целые переменные **i** и **j**, переменной **j** присвоено значение 5.

```
int i, j = 5;
```

Язык C допускает *преобразование* (*приведение*) *типов и переименование типов данных.*

Возможны явные и неявные (стандартные) преобразования типов.

Явное преобразование типов выполняется специальной операцией приведения типа:

```
(<тип>) <выражение>; //используется в C и C++
```

Например:

```
int a;       // объявлена целочисленная переменная a
double b;    // объявлена вещественная переменная b
             // двойной точности
b = 10.5;    // переменной b присвоено значение 10,5.
a = (int)b; // переменной a присвоено значение переменной
             // b, приведенной к целому типу. В результате
             // a получит значение 10.
```

Неявные преобразования выполняются компилятором автоматически следующим образом:

• в выражениях операнды преобразуются к одинаковому типу по принципу «простой — в более сложный» (допускающий больший диапазон значений):

```
char → short int → unsigned int → long → unsigned
long → float → double → long double;
```

• в операторах присваивания правая часть преобразуется к типу левой части.

Переименование типа проводится объявлением нового идентификатора типа данных, синонима существующего типа, с целью использования более удобного или сокращенного названия. Синтаксис:

```
typedef <тип><идентификатор>
```

Здесь тип — существующий тип данных, идентификатор — идентификатор-синоним, например:

```
// тип t объявлен как синоним типа unsignd int
typedef unsigned int t;
// при объявлении переменных x и y используется
// длинное название типа
unsigned int x, y;
// при объявлении переменных p и q
// используется короткое название типа
t p, q;
```

Помимо типа переменная характеризуется временем жизни и областью видимости.

Под *временем жизни* переменной понимается время от создания переменной (выделения памяти под нее) до ее уничтожения (освобождения памяти).

В большинстве случаев память под переменную выделяется в момент ее объявления. Уничтожение же может происходить в разные моменты времени в зависимости от области видимости, определяемой местом и способом объявления переменной.

Под *областью видимости* переменной понимается часть программы, в которой данная переменная существует, где можно получать и менять ее значение. Область действия переменной зависит от места и способа ее объявления.

В зависимости от мест объявления переменных, они делятся:

• на *глобальные* (внешние) — переменные, объявленные в начале модуля, вне всех его функций. Областью видимости таких переменных является весь модуль, они доступны в любой функции этого модуля. Заданием специального класса памяти видимость переменной можно распространить и на другие модули;

• *локальные* — переменные, объявленные в начале функции. Область видимости такой переменной ограничена функцией, в которой переменная была объявлена.

Например:

```
int i; //объявлена глобальная переменная i
void main()
{
  int j; // объявлена локальная переменная j
    ...
}
```

Объявление переменной может сопровождаться указанием класса памяти:

```
<класс памяти><тип переменной><имя переменной>
```

Класс памяти определяет порядок размещения переменной в памяти и задается с помощью специальных спецификаторов.

В языке С существует четыре таких спецификатора:

• auto;
• register;
• static;
• extern.

Переменные, объявленные с классом памяти *auto,* размещаются в локальной памяти. При этом размер выделяемой памяти известен заранее, еще на этапе компиляции программы. По завершении части программы (функции), в которой была объявлена переменная, соответствующая область локальной памяти освобождается и размещенная в ней переменная уничтожаются. Переменные, объявленные таким образом, являются локальными. Класс памяти auto используется по умолчанию, поэтому при объявлении переменных указывается редко.

Спецификатор *register* также относится к автоматическому классу памяти. Переменные, объявленные с таким спецификатором, по возможности располагаются не в локальной памяти, а в одном из доступных регистров. В этом случае доступ к переменной осуществляется намного быстрее. Если в момент выполнения кода данной части программы все регистры оказываются занятыми, переменная будет обрабатываться как имеющая класс auto. Область видимости и время жизни переменных класса register полностью аналогичны классу auto.

Спецификатор *static* относится к внутреннему статическому классу памяти. Переменная, объявленная с таким классом памяти, располагается по фиксированному адресу. Поэтому такая переменная существует с момента ее объявления до конца выполнения программы (несмотря на то, что переменная может быть объявлена

локально). В отличие от объекта класса памяти auto, переменная не будет уничтожаться при выходе из функции, где она была объявлена.

Например:

```
void ab()
{
  int a=1; // объявлена целая локальная переменная a,
           // ей присвоено начальное значение, равное 1
  static int b=1; // объявлена статическая переменная b,
                  // ей присвоено начальное значение,
                  // равное 1
  a++;  // значение переменной a увеличивается на 1
  b++;  // значение переменной b увеличивается на 1
}
```

В начале выполнения функции **ab()** значения переменных **a** и **b** равны 1. В ходе выполнения функции значения обеих переменных увеличиваются на 1. По окончании выполнения функции переменная **a** будет уничтожена, а при повторном вызове функции создана вновь со значением 1. Значение же переменной **b** будет увеличиваться на 1 при каждом вызове функции, сама переменная не уничтожается на протяжении всей программы. Значение 1 присваивается переменной **b** только один раз при первом вызове функции **ab()**, когда переменная создается. При последующих вызовах функции присвоение переменной **b** значения 1 игнорируется.

Extern является спецификатором внешнего статического класса памяти. Этот спецификатор позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом модуле. С помощью спецификатора extern можно получить доступ к внешней переменной, объявленной в другом модуле. Спецификатор extern указывает компилятору на то, что в другом модуле такая переменная уже описана. Поэтому выделять под нее память в данном модуле не следует, а следует найти эту переменную в других модулях на стадии компоновки. Если такая переменная найдена не будет, то компоновщик выдаст ошибку.

5.3. Операции над данными

Выражением называется последовательность операций, операндов и знаков препинания, задающих определенное вычисление. Вычисление выражений выполняется по определенным правилам преобразования и приоритета, которые зависят от используемых в выражениях операций, наличия круглых скобок и типов данных операндов.

Операндами называются данные (переменные, константы), над которыми выполняются операции.

В языке С имеется достаточно большой набор операций. Для удобства рассмотрения все операции можно классифицировать, например, по количеству используемых операндов. По этому признаку все операции делятся:

- на унарные (с одним операндом);
- бинарные (с двумя операндами);
- операция с тремя операндами.

Унарные операции в языке С могут использоваться в префиксной и постфиксной формах. Общий синтаксис операций:

```
<операция><операнд> // префиксная форма;
<операнд><операция> // постфиксная форма.
```

Выбор определенной формы записи зависит от конкретной операции.

К унарным операциям относятся следующие:

- операции изменения знака операнда;
- операция побитового отрицания;
- операция логического отрицания;
- операция определения размера;
- инкремент и декремент;
- операция доступа;
- адресные операции.

Операции изменения знака операнда унарный плюс (+) и унарный минус (-) могут выполняться над любыми выражениями арифметического типа. Унарный минус эквивалентен умножению значения операнда на –1. Унарный плюс эквивалентен умножению значения операнда на +1. Эта операция фактически ничего не делает, но может быть использована для явного указания того, что используемая в качестве операнда константа является положительным значением.

Операция побитового отрицания (~) выполняется над операндом целочисленного типа и заключается в побитовом инвертировании двоичного кода операнда (все единицы в этом коде заменяются нулями, а нули — единицами). Результатом операции является число, имеющие таким образом сформированный двоичный код.

Операция логического отрицания (**!**) выполняется над операндом арифметического типа. Любое значение операнда, отличное от нуля, в языке С считается истинным (числовой эквивалент этого значения — 1), значение, равное нулю, — ложным. Операция меняет логическое значение операнда: если он был равен нулю, то возвращается значение 1, если операнд был не равен нулю, то возвращается значение 0.

Операция определения размера (**sizeof**) может выполняться над выражением любого типа или над типом данных. В первом случае

результатом выполнения операции является размер значения выражения в байтах, во втором — размер указанного типа данных в байтах. Результат операции имеет тип **size_t**, относящийся к целочисленному беззнаковому типу. Синтаксис операции:

```
sizeof <выражение>;
sizeof (<тип>).
```

Например:

```
int i, j, k;
j = sizeof(i);   // значение j равно 4;
k = sizeof(int); // значение k равно 4.
```

К операциям *увеличения* и *уменьшения значения* относятся:

- инкремент **(++)**;
- декремент **(--)**.

Инкремент увеличивает значение операнда на 1, декремент — уменьшает значение операнда на 1.

Эти операции могут использоваться в префиксной и постфиксной формах. В префиксной форме в вычислении значения выражения используется уже измененное значение операнда (в результате значение выражения совпадает со значением операнда), в постфиксной форме сначала производится вычисление выражения, а потом изменение значения операнда (в результате значение выражения оказывается на единицу больше или меньше значения операнда). Например:

```
int i,j;
i=1;
j=++i; // использована префиксная форма. Значение i
       // сначала увеличено на 1 (i=2), а потом
       // присвоено j
       // (j=2)
i=1;
j=i++; // использована постфиксная форма. Значение i
       // сначала присвоено j (j=1), а затем i
       // увеличено на 1 (i=2)
```

К адресным операциям относятся:

- операция получения адреса операнда **(&)**;
- операция обращения по адресу **(*)**.

Эти операции выполняются над данными, имеющими тип указателя, и будут рассмотрены ниже.

Бинарные операции выполняются над двумя операндами. Общий синтаксис этих операций:

```
<операнд 1><операция><операнд 2>
```

К бинарным операциям относятся следующие:

- арифметические операции;
- операции сравнения;

• побитовые операции;
• логические операции;
• операция присваивания;
• прочие бинарные операции.

Арифметические операции выполняются над операндами числового типа. Результатом также является значение числового типа. К разряду арифметических относятся операции:

• сложения (**+**) и вычитания (**-**). Операнды в этих операциях, кроме числового типа, могут иметь и тип указателя, который будет рассмотрен позднее;
• умножения (*****) и деления (**/**);
• получения остатка от деления (деление по модулю **%**). Операндами для этой операции являются целочисленные значения. Если изначально операнды не являются целыми числами, то они будут приведены к целому типу. Результат операции будет положительным, если оба операнда неотрицательны, в противном случае знак результата зависит от конкретной реализации компьютера.

Операции сравнения выполняются над операндами числового типа. Результатом операции является одно из двух целочисленных значений: 0 соответствует *лжи* (FALSE), 1 соответствует *истине* (TRUE). К операциям сравнения относятся:

• **<** — меньше;
• **<=** — меньше или равно;
• **>** — больше;
• **>=** — больше или равно;
• **==** — равно;
• **!=** — не равно.

Например:

```
int i,j;          // объявлены целочисленные переменные
                  // i и j
char res1, res2;  // объявлены символьные переменные
                  // res1 и res2
i = 5;            // переменной i присвоено значение 5
j = 8;            // переменной j присвоено значение 8
res1 = i == j;    // переменной res1 присвоено значение
                  // результата операции i == j,
                  // в результате эта переменная получит
                  // значение 0 (FALSE)
res2 = i < j;     // переменной res2 присвоено значение
                  // результата операции i < j,
                  // в результате эта переменная получит
                  // значение 1 (TRUE)
```

Побитовые операции выполняются над целочисленными операндами. Результатом также является значение целого типа. К побитовым операциям относятся:

• операции побитового левого (<<) и правого (>>) сдвига. Операция сдвигает двоичный код левого операнда влево или вправо на количество разрядов, заданное значением правого операнда. Соответствующее количество разрядов слева или справа теряется, а вновь добавленные разряды заполняются нулями. Например:

```
unsigned char i, j, k;
i = 163;     // двоичное представление 163 равно 10100011
j = i >> 2;  // сдвиг вправо дает значение 00101000,
             // равное 40
k= i << 5;   // сдвиг влево дает значение 01100000,
             // равное 96
```

• поразрядные побитовые операции: конъюнкция (&), дизъюнкция (|), исключающая дизъюнкция (^). Двоичный код результата формируется путем выполнения соответствующих побитовых операций над двоичными кодами операндов. Результаты побитовых операций приведены в табл. 5.3. Например:

```
unsigned char i, j, k, m, n;
i=101;       // двоичное представление 101 равно 01100101
j=98;        // двоичное представление 98 равно 01100010
k=i&j;       // побитовая конъюнкция дает значение
             // 01100000, равное 96
m = i | j;   // побитовая дизъюнкция дает значение
             // 01100111, равное 103
n = i ^ j;   // побитовая исключающая дизъюнкция дает
             // значение 00000111, равное 7
```

Таблица 5.3

Результаты побитовых операций

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Логические операции объединяют несколько операндов, имеющих значение ЛОЖЬ (нулевое значение) или ИСТИНА (любое ненулевое значение). Часто эти операции используются для создания сложных условий, объединяющих несколько простых операций сравнения. К логическим операциям относятся логическое И (&&) и логическое ИЛИ (||). Результаты логических операций приведены в табл. 5.4. В этой таблице под значением X подразумевается любое ненулевое значение.

В результате *операции присваивания* левому операнду присваивается значение правого операнда, в качестве которого может вы-

Таблица 5.4

Результаты логических операций

a	b	a && b	a ‖ b
0	0	0	0
0	X	0	1
X	0	0	1
X	X	1	1

ступать выражение любого типа. Тип присваиваемого значения преобразуется к типу переменной, получающей это значение.

Синтаксис операции:

```
<операнд1> = <операнд2>
```

Результатом операции является значение правого операнда. Это позволяет записывать несколько операторов присваивания в цепочку:

$A_n = A_{n-1} = \ldots = A_2 = A_1$

При выполнении такой операции операнды группируются справа налево:

$A_n = (A_{n-1} = (\ldots = (A_2 = A_1)))$

В результате всем операндам будет присвоено значение A_1.

В языке С может применяться специальная форма операции присваивания. Ее синтаксис:

```
<операнд1><операция> = <операнд2>
```

Операция выполняется над значениями операнда1 и операнда2, полученное значение присваивается операнду1. Например:

- операция i = i + 5 эквивалентна операции i+=5
- операция i = i & j эквивалентна операции i&= j

К *прочим бинарным* операциям в языке С можно отнести операции:

- последовательного выполнения «**,**». Эта операция группирует выражения слева направо. Разделенные операцией выражения вычисляются последовательно слева направо, в качестве результата сохраняются тип и значение самого правого выражения. Операция обычно используется для выполнения нескольких выражений в ситуациях, где по синтаксису может быть лишь одно;
- приведения типа «**( )**». Операция обеспечивает приведение типа значения выражения, представляемого вторым операндом, к типу, спецификация которого задается первым операндом. Например:

```
(float) i; // переменная i приводится к типу
           // вещественного числа одинарной точности
```

• индексации «([])» используется при работе с массивами и будет рассмотрена ниже;

• вызова функции «()». Эта операция будет рассмотрена ниже;

• обращения к элементам объекта сложного типа. Эти операции будут рассмотрены ниже.

В языке С имеется единственная операция, работающая с тремя операндами — это условная операция «**?**». Ее синтаксис:

```
<операнд1> ? <операнд2> : <операнд3>
```

При выполнении операции сначала вычисляется значение операнда1. Если это значение является истинным (отличным от 0), то результатом операции является значение операнда2, если ложным (равным 0), то операнда3.

Например, следующая операция будет вычислять модуль числа X:

```
X>=0 ? X: -X;
```

Если число X неотрицательно, то результатом операции будет значение числа X, а иначе — значение числа X с противоположным знаком.

Приоритеты выполнения операций в языке С приведены в табл. 5.5. Порядок выполнения всех перечисленных в ней операций — слева направо.

Таблица 5.5

Приоритеты выполнения операций

Приоритет	Операция	Обозначение
1	Префиксный инкремент	++
	Префиксный декремент	--
	Вызов функции	()
	Индексация	[]
	Указатель структуры	→
	Член структуры	.
2	Логическое отрицание	!
	Побитовое отрицание	~
	Унарный минус	-
	Унарный плюс	+
	Приведение типа	(type)
	Обращение по адресу	*
	Получение адреса	&
	Определение размера	sizeof

Окончание табл. 5.5

Приоритет	Операция	Обозначение
3	Умножение	*
	Деление	/
	Остаток от деления	%
4	Сложение	+
	Вычитание	-
5	Поразрядный сдвиг влево	<<
	Поразрядный сдвиг вправо	>>
6	Меньше	<
	Меньше или равно	<=
	Больше	>
	Больше или равно	>=
7	Равно	==
	Не равно	!=
8	Поразрядная конъюнкция	&
9	Поразрядная исключающая дизъюнкция	^
10	Поразрядная дизъюнкция	|
11	Логическое И	&&
12	Логическое ИЛИ	||
13	Условная операция	? :
14	Присваивание	=
	Составное сложение	+=
	Составное вычитание	-=
	Составное умножение	*=
	Составное деление	/=
	Составной остаток от деления	%=
	Составной поразрядный сдвиг влево	<<=
	Составной поразрядный сдвиг вправо	>>=
	Составная поразрядная конъюнкция	&=
	Составная поразрядная исключающая дизъюнкция	^=
	Составная поразрядная дизъюнкция	|=
15	Операция «запятая»	,
	Постфиксный инкремент	++
	Постфиксный декремент	--

Кроме обычных операций, в состав языка С входят также базовые функции. Стандартная библиотека С определяет набор функций, общий для всех компиляторов ISO/ANSI C. Он содержит широкий диапазон подпрограмм, прототипы которых объединены по назначению в файлы заголовков.

5.4. Основные алгоритмические структуры

Обычно операторы программы выполняются последовательно один за другим. Однако часто требуется передать управление оператору, не следующему по порядку. Для реализации такой передачи используется *оператор безусловного перехода goto*. Синтаксис оператора:

```
goto <метка>;
```

Метка в программе записывается в начале строки и отделяется от операторов двоеточием. Например:

```
i=5;
goto m1;
j=10; // Никогда не выполняется
m1: i+=3;
```

Многократное использование в программе операторов goto приводит к большим трудностям в сопровождении программ, поэтому, в соответствии с принципами структурного программирования, применение оператора **goto** считается нежелательным. Тем не менее в исключительных случаях (в программах со множественным вложением условных операторов) его применение может значительно сократить размер кода и улучшить его читаемость.

В структурном программировании выделяют три вида алгоритмических (управляющих) структур:

- структура следования;
- структура ветвления;
- структура повторения.

Структура следования, представленная на рис. 5.2, встроена в язык С и означает режим последовательного выполнения операторов программы.

Структура ветвления (выбора) служит для программирования разветвляющихся вычислительных процессов и реализована в виде трех структур:

- структура с единственным выбором (рис. 5.3) выполняет некоторое действие, если условие истинно, и игнорирует его, если условие ложно;
- структура с двойным выбором (рис. 5.4) выполняет одно действие, если условие истинно, и второе действие, если условие ложно;

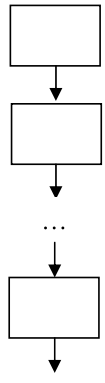


Рис. 5.2. **Структура следования**

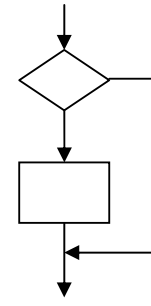


Рис. 5.3. **Структура с единственным выбором**

• структура с множественным выбором (рис. 5.5) выполняет выбор действия среди множества действий.

Структура с единственным выбором реализуется с помощью условного оператора *if*. Синтаксис:

```
if(<выражение>) <оператор>
```

При выполнении условного оператора сначала вычисляется значение выражения, стоящего после **if**. Если значение выражения истинно (отлично от 0), то выполняется указанный оператор, в противном случае условный оператор игнорируется и управление передается следующему за ним оператору. Выражение *должно* принимать целое значение или будет преобразовано к целому значению.

Примечание: в конструкциях языка С под оператором может пониматься и блок операторов — последовательность действий, заключенная в фигурные скобки. Синтаксис:

```
{
  Объявления и определения данных
  Операторы
}
```

Такая конструкция позволяет указать выполнение нескольких операторов там, где по синтаксису может быть только один. Объявления и определения данных, заданные в блоке, действуют только в этом блоке.

С помощью блока операторов в условном операторе **if** можно задать выполнение нескольких операторов при истинном значении выражения.

Пример 5.4.1

Ввести число X и вычислить Y, равное

$$Y = \sqrt{X - 10}.$$

При X, меньшем -10, заменить число X на X^2.

```
#include <stdio.h>
#include <math.h>
int main()
{
  float X, Y;
  scanf("%f", &X); // ввод числа X
  if (X < 10)
    X = X * X;
  Y = sqrt(X - 10);
  // Y присваивается значение функции квадратного корня
  // из X-10
  printf("%f", Y); // вывод значения Y
  return 0;
}
```

Пример 5.4.2

Ввести числа X и Y и вычислить Z, равное

$$Z = \frac{X + Y}{X - Y - 1}.$$

При X, равном $(Y + 1)$, число X увеличить в два раза, а число Y уменьшить в два раза.

```
#include <stdio.h>
int main()
{
  float X, Y, Z;
  scanf("%f %f", &X, &Y);  //ввод чисел X и Y
  if (X == (Y + 1)) // при X=Y+1 выполняется блок
                    // операторов
  {
    X = X * 2; // X увеличивается в 2 раза
    Y = Y / 2; // Y уменьшается в 2 раза
  }
  Z = (X + Y) / (X - Y - 1); // вычисление значения Z
  printf("%f", Z);  // вывод числа Z
  return 0;
}
```

Структура с двойным выбором реализуется с помощью условного оператора **if — else** (см. рис. 5.4). Синтаксис:

```
if(<выражение>) <оператор1>; else<оператор 2>;
```

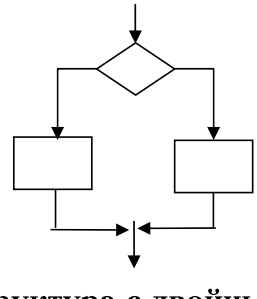


Рис 5.4. **Структура с двойным выбором**

При выполнении этого оператора сначала вычисляется значение выражения, стоящего после **if**. Если значение выражения истинно (отлично от 0), то выполняется оператор1, в противном случае выполняется оператор2.

Пример 5.4.3

Ввести числа *X* и *Y*, числу Z присвоить значение, равное половине наибольшего из введенных чисел.

```
#include <stdio.h>
int main()
{
  float X, Y, Z;
  scanf("%f %f", &X, &Y);  // ввод чисел X и Y
  if (X > Y)  // если X > Y, то X - наибольшеее
    Z = X / 2;
  else
    Z = Y / 2; // иначе Y наибольшее
  printf("%f", Z);
  return 0;
}
```

Пример 5.4.4

Ввести числа *X* и *Y*, не равные друг другу. Наибольшее число уменьшить в два раза, а наименьшее — увеличить в два раза.

```
#include <stdio.h>
int main()
{
  float X, Y;
  scanf("%f %f", &X, &Y); // ввод чисел X и Y
    if (X>Y)  // X - наибольшее, выполняется блок
              // операторов
  {
    X = X / 2;
    Y = Y * 2;
  }
  else  // Y - наибольшее, выполняется блок операторов
  {
    Y = Y / 2;
    X = X * 2;
  }
  printf("%f %f", X, Y);
  return 0;
}
```

Оператор **if** может быть вложенным. Это означает, что данный оператор может быть включен в конструкцию **if** или **else** другого условного оператора. При этом компилятор связывает каждое ключевое слово **else** с наиболее близким **if**, для которого нет **else**.

Пример 5.4.5

Ввести три числа *a*, *b*, *c*, не равные друг другу. Найти сумму максимального и минимального из этих чисел.

```
#include <stdio.h>
int main()
{
  float a, b, c, s;  // s - сумма максимального
                     // и минимального числа
  scanf("%f %f %f", &a, &b, &c); // ввод исходных чисел
  if (a>b)
    if (a>c)    // максимальное число a
      if (b>c) // минимальное число c
        s = a + c;
      else      // минимальное число b
        s = a + b;
      else      // максимальное число c, минимальное - b
      s = c + b;
  else
    if (b>c)    // максимальное число b
      if (a>c) // минимальное число c
        s = b + c;
      else      // минимальное число a
        s = b + a;
      else      // максимальное число c, минимальное - a
        s = c + a;
  printf("%f", s);
  return 0;
}
```

При составлении выражения в операторе **if** можно использовать логические операции И и ИЛИ, позволяющие объединить несколько простых выражений в одно составное. Программа, реализующая предыдущий пример, с помощью этих операций может быть записана в следующем виде:

```
#include <stdio.h>
int main()
{
  float a, b, c,s;
  scanf("%f %f %f", &a, &b, &c);
  if ((a > b) && (b > c))
    s = a + c;
  if ((a > c) && (c > b))
    s = a + b;
  if ((b > a) && (a > c))
    s = b + c;
  if ((b > c) && (c > a))
    s = b + a;
  if ((c > a) && (a > b) )
    s = c + b;
```

```
  if ((c > b) && (b > a))
    s = c + a;
  printf("%f", s);
  return 0;
}
```

При использовании логических операций И и ИЛИ следует иметь в виду, что выражение может не вычисляться до конца. Действительно:

• если значение левого операнда операции && равно FALSE (нулю), то значение правого операнда не вычисляется, так как результат всей операции от него не зависит;

• если значение левого операнда операции || равно TRUE (не равно нулю), то значение правого операнда также не вычисляется.

Рассмотрим еще один пример использования вложенного оператора **if**.

Пример 5.4.6

Ввести коэффициенты квадратного уравнения *a*, *b*, *c*. Найти корни уравнения.

```
#include <stdio.h>
#include <math.h>
int main()
{
  int a, b, c, d;
  float x1, x2;
  scanf("%d %d %d",&a, &b, &c);  // ввод коэффициентов
                                 // уравнения
  d=b*b-4*a*c;  // вычисление дискриминанта
  if (d<0) // если дискриминант меньше 0, корней нет
    printf("net korney\n");
  else
    if(d==0)  // если дискриминант равен 0, у уравнения
              // один корень
      printf("1 korn:  %d\n", -b/(2*a));
  else// в противном случае у уравнения два корня
    printf("2 korna: %f %f\n", (-b+sqrt(d))/(2*a),
    (-b-sqrt(d))/(2*a));
}
```

Структура с множественным выбором (см. рис. 5.5) реализуется с помощью оператора выбора *switch-case*. Синтаксис:

```
switch (<выражение>)
{
case<метка1>:
  [<Операторы>;]
case<метка2>:
  [<Операторы>]
...
```

```
[default:< операторы>;]
}
```

Примечание: наличие квадратных скобок в синтаксисе указывает на то, что заключенный в них элемент конструкции не является обязательным, при написании конкретного оператора может быть опущен.

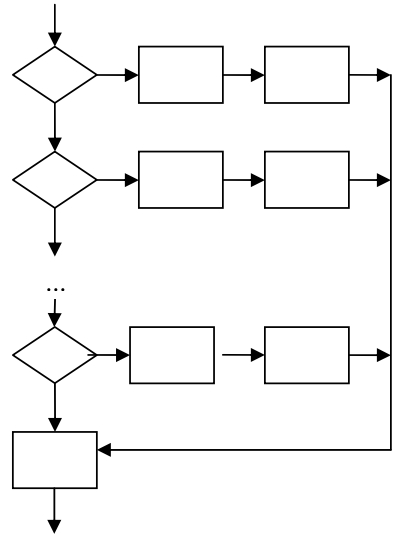


Рис 5.5. **Структура с множественным выбором**

Оператор **switch** сравнивает значение выражения, стоящего после **switch**, со всеми значениями (метками), перечисленными с помощью ключевых слов **case**. Как только значение выражения совпадает с какой-либо меткой, управление передается оператору, следующему за этой меткой. Если значение выражения не совпадает ни с одной меткой, то управление передается оператору с меткой **default** (если такая метка есть) или оператору, следующему за оператором **switch** (если метка **default** отсутствует).

Как выражение, так и метки должны иметь значения целого типа (включая тип char).

В языке C в операторе **switch-case** после выполнения оператора (группы операторов), следующих за соответствующей меткой, выполнение оператора **switch** не прекращается, а продолжается до самого конца, выполняя все действия, указанные для всех нижестоящих меток. Это позволяет записывать несколько последовательных меток для одной и той же группы операторов. В этом случае группа операторов будет выполняться, если значение выражения равно хотя бы одной из меток.

Если в программе не требуется выполнять действия, указанные для последующих меток, прервать выполнение оператора **switch** можно с помощью оператора **break**. Оператор **break** прерывает выполнение оператора **switch** и передает управление следующему после **switch** оператору.

Пример 5.4.7

Ввести номер месяца и вывести номер квартала, к которому относится введенный месяц.

```
#include <stdio.h>
int main()
{
  int m;
  scanf("%d", &m);
  switch (m)
  {
    case 1:
    case 2:
    case 3:  // номер месяца равен 1, 2 или 3
      printf("1 kvartal");
      break; // выводится сообщение "1 квартал",
             //выполнение опратора switch прерывается
    case 4:
    case 5:
    case 6:
      printf("2 kvartal");
      break;
    case 7:
    case 8:
    case 9:
      printf("3 kvartal");
      break;
    case 10:
    case 11:
    case 12:
    printf("4 kvartal");
    break;
    default: // введен неверный номер месяца,
             // вывод сообщения об ошибке
      printf("error");
      break;
  }
}
```

Структура повторения служит для программирования циклических вычислительных процессов и реализована в языке С в виде трех типов циклических структур:

- цикл с заранее известным числом повторений (**for**);
- цикл с предусловием (**while**);
- цикл с постусловием (**do while**).

В процессе реализации цикла с заранее известным числом повторений **for** (рис. 5.6) выполняются три операции:

- инициализация счетчика (задается переменная цикла с ее начальным значением);

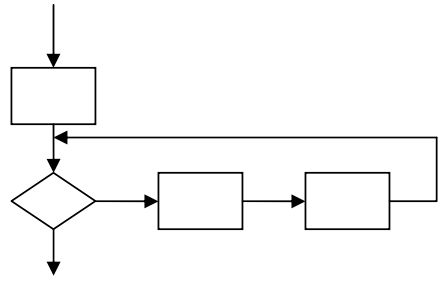


Рис. 5.6. **Структура цикла с заранее известным числом повторений**

• проверка условия повторения (значение счетчика сравнивается с конечным значением);

• приращение счетчика (изменение значения счетчика при каждом прохождении тела цикла).

Тело цикла повторяется, пока условие повторения цикла истинно. В качестве тела цикла может выступать один оператор или блок операторов.

Синтаксис цикла for:

```
for ([<выражение1>];[<выражение2>];[<выражение3>]) оператор;
```

Выражение1 задает инициализацию счетчика и выполняется один раз в начале цикла. Это выражение может быть пустым.

Выражение2 проверяет условие продолжения цикла. Вычисление этого выражения производится в начале каждой итерации и, если его значение равно TRUE, итерация выполняется, иначе цикл заканчивается. Если это выражение равно FALSE в самом начале цикла, тело цикла не будет выполнено ни разу. Выражение может быть пустым, тогда его значение равно TRUE. В этом случае цикл может быть бесконечным. Например:

```
#include <stdio.h>
int main()
{
  int i, j;
  for (i=1 ; ; i++)
  /*
  условие выполнения цикла является пустым,
  следовательно, оно всегда выполняется
  */
  j = i;
  return 0;
}
```

Выражение3 модифицирует счетчик и выполняется в конце каждой итерации цикла. Выражение может быть пустым.

В языке С переменная-счетчик может быть изменена в теле цикла. Это может оказывать влияние на длительность цикла. Например:

```
#include <stdio.h>
int main()
{
  int i, j;
  for (i = 0; i< 7; i++)
  {
    /* в каждой итерации переменная цикла увеличивается
    на 2, в конце каждой итерации — еще на 1. Таким
    образом, цикл выполнится 3 раза */
    i += 2;
    j = i;      // j примет значения 2, 5 и 8
    printf("%d ", j);
  }
  return 0;
}
```

В качестве *выражения1* и *выражения3* могут использоваться сложные выражения, созданные с использованием операции «,».

Пример 5.4.8

С клавиатуры ввести *n* чисел. Количество чисел *n* вводится вначале. Найти и вывести сумму введенных чисел.

```
#include<stdio.h>
int main()
{
  int s, i, x, n;
  s = 0; // s — сумма введенных чисел, изначально равна 0
  scanf("%d", &n); // ввод количества чисел
  for (i = 0; i<n; i++) // i — счетчик количества чисел,
                        // меняется от 0 до n-1
  {
    scanf("%d", &x);    // ввод очередного числа
    s+=x ;  // добавление введенного числа к сумме
  }
  printf("%d", s);      // вывод накопленной суммы чисел
  return 0;
}
```

Пример 5.4.9

С клавиатуры ввести *n* чисел. Найти и вывести максимальное из введенных чисел.

```
#include <stdio.h>
int main()
{
  int n, i, x;
  int max;
  scanf("%d", &n); // ввод количества чисел
  scanf("%d", &x); // ввод первого числа
  max = x;     // первое число принимается за максимум
  for (i=1; i<n; i++)
  {
```

```
    scanf("%d", &x); // ввод очередного числа
    if (x>max) // если очередное число больше максимума,
               // максимум меняется
      max = x;
  }
  printf("%d", max);
  return 0;
}
```

Пример 5.4.10

Два поезда двигаются в одном направлении из пунктов, расстояние между которыми s километров. Скорость первого поезда — v1 км/ч, скорость второго поезда — v2 км/ч, v2 > v1. Через сколько часов второй поезд догонит первый?

```
#include <stdio.h>
int main()
{
    int i, j, t = 0;
    int v1, v2, s;
    scanf("%d %d %d", &s, &v1, &v2);
/* Инициализация счетчиков включает две переменные,
начальная разность между значениями которых равна s.
При каждом выполнении счетчики увеличиваются на v1 и v2.
Соответственно, цикл выполняется до тех пор, пока
значение первого счетчика больше значения второго. */
for (i = 0, j = -s; i>j; i += v1, j += v2)
  t++;

  printf("%d", t);
  return 0;
}
```

Цикл с предусловием *while* (рис. 5.7) выполняется, пока указанное в нем условие является истинным. Синтаксис:

```
while (<условие>) <оператор>;
```

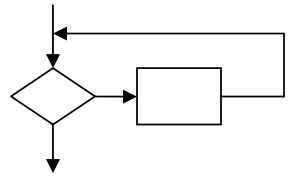


Рис. 5.7. **Структура цикла с предусловием**

Условие проверяется в начале цикла. Если оно равно TRUE, то выполняется идущий за ним оператор (блок операторов), после чего условие проверяется снова. Если условие ложно, цикл **while** заканчивается. Если условие ложно в самом начале, то цикл не выполнится ни одного раза.

При построении этого цикла в его тело нужно включить операторы, изменяющие значение проверяемого условия так, чтобы в конце концов оно стало ложным. В противном случае выполнение цикла никогда не завершится.

Пример 5.4.11

С клавиатуры вводится ряд чисел. Признаком окончания ввода является ввод числа, равного нулю. Найти и вывести среднее арифметическое введенных чисел.

```
#include<stdio.h>
int main()
{
  int x,i;
  float sr;
  sr=0;  // сумма чисел
  i=0;   // количество чисел
  scanf("%d", &x);  // ввод очередного числа
  while (x!=0) // пока число не равно нулю
  {
        sr+=x;  // увеличение суммы чисел
        i++;    // увеличение количества чисел
        scanf("%d", &x);  // ввод нового числа
  }
  if (i!=0)    // если количество чисел не равно 0
        printf("%f",sr/i);  // вывод среднего
                            // арифметического
  else
        printf("no numbers"); // вывод сообщения
                              // об отсутствии чисел
  return 0;
}
```

Пример 5.4.12

С клавиатуры ввести целое число. Найти количество цифр этого числа.

```
#include <stdio.h>
int main()
{
  int n, m = 0; // m - количество цифр числа,
  изначально равна 0
  scanf("%d", &n); // ввод исходного числа
  while (n > 0) // пока число больше 0
  {
    m++; // количество цифр в числе увеличивается на 1
    n /= 10;  // новое число получается путем деления
              // текущего числа на 10
  }
  printf("%d", m);
  return 0;
}
```

Цикл с постусловием *do while* (рис. 5.8) повторяет действие, пока условие остается истинным, причем при первой итерации условие не проверяется. Таким образом, этот цикл обязательно выполнится хотя бы один раз. Синтаксис:

```
do
  <оператор>;
while (<условие>);
```

В отличие от цикла **while**, проверка условия выполнения цикла здесь производится в конце, а не в начале каждой итерации. В остальном работа циклов идентична.

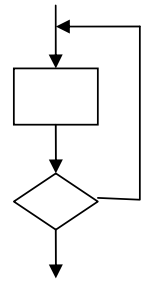


Рис. 5.8. **Структура цикла с постусловием**

Пример 5.4.13

Решение задачи 5.4.11 с помощью оператора **do while**:

```
#include <stdio.h>
int main()
{
  int i = 0;
  float x, s = 0;
  do
  {
    scanf("%d", &x);
    s += x;
    i++;
  }
  while (x != 0);
    // количество чисел на 1 меньше i, так как при вводе 0
    // количество чисел все равно увеличивается
    printf("%f ", s/(i - 1));
    return 0;
}
```

Все циклические конструкции могут быть вложенными. Тело цикла среди прочих операторов может включать и операторы цикла.

Пример 5.4.14

С клавиатуры ввести *n* целых чисел. Найти и вывести число, имеющее максимальное количество целочисленных делителей. Если таких чисел несколько, вывести первое из них.

```
#include <stdio.h>
int main()
{
// n — количество чисел, x — текущее число,
// y — число с максимальным количеством делителей
  int n, x, y;
// k — количество делителей каждого числа,
// max — максимальное количество делителей

  int k, max;
  int i, j; // i и j - счетчики циклов
  scanf("%d", &n);
  max = 0;
  for (i = 0; i<n; i++)
  // внешний цикл перебирает все числа
  {
    scanf("%d", &x);
    k = 1;
    // внутренний цикл считает количество делителей
    // каждого числа
    for (j = 1; j < x / 2 + 1; j++)
      if (x%j == 0)
        k++;
      if (k > max)
      {
        max = k;
        y = x;
      }
  }
  printf("%d", y);
  return 0;
}
```

Пример 5.4.15

С клавиатуры ввести *n* целых чисел. Найти и вывести число с минимальной суммой цифр. Если таких чисел несколько, вывести последнее из них.

```
#include <stdio.h>
int main()
{
  int n,x,y,z;  // n - количество чисел, x - текущее
                // число, y - число с минимальной
                // суммой цифр, z - буферная переменная
  int s,m;      // s - сумма цифр числа,
                // m - минимальная сумма
  int i;        // i - счетчик цикла
  m = 10000000; // изначально минимуму присваивается
                // заведомо невозможно большое значение
   scanf ("%d", &n);
   for (i = 1; i <= n; i++) // внешний цикл перебирает
                            // все числа
```

```
  {
    scanf("%d", &x);
    z = x;
    s = 0;
    while (x != 0)  // внутренний цикл вычисляет сумму
                    // цифр каждого числа
    {
      s += x % 10;
      x /=  10;
    }
    if (s <= m) { m = s; y = z; }
  }
  printf("%d", y);
}
```

В языке С имеются операторы, позволяющие управлять выполнением итераций внутри тела цикла. К ним относятся следующие операторы:

• *break* полностью прерывает выполнение цикла и передает управление следующему за циклом оператору; кроме того, оператор **break** используется в теле оператора **switch-case** для разделения выполняемых блоков, о чем говорилось выше;

• *continue* прерывает выполнение текущей итерации цикла и передает управление на начало следующей итерации.

Пример 5.4.16

Подсчитать средний балл студентов, сдавших экзамен. При вводе оценки, меньшей 2 или большей 5, считать, что при подведении итогов экзамена допущена ошибка, и прервать обработку. Оценки, равные 2, игнорировать.

```
#include <stdio.h>
int main()
{
  int n; // n - количество студентов,
         // ball - текущая оценка, s - сумма оценок,
         // k - количество студентов, сдавших экзамен
  int ball, s=0, k=0;
  int i, j=0; // i - счетчик цикла, j - признак
              // прерывания цикла
  scanf ("%d", &n);

  for (i = 1; i <= n; i++)
   {
    scanf("%d", &ball);
    // если оценка больше 5 или меньше 2, выдается
    // сообщение об ошибке, и цикл прерывается
    if ((ball > 5) || (ball < 2))
    {
      printf("input error");
```

```
        j = 1;
        break;
      }

      // оценка, равная 2, игнорируется, происходит
      // переход к обработке следующей оценки
      if (ball == 2)
        continue;
        s += ball;
        k++;
    }

    if (j==0)
      // если цикл не прерывался, выводится результат
      printf("%f",(float)s/k);
    return 0;
}
```

Оператор **return** осуществляет передачу управления из внутренней функции во внешнюю. Более подробно работа этого оператора будет рассмотрена ниже.

Часть действий по обработке данных в программе может быть выполнена на этапе предварительной обработки перед стадией компиляции. Для этого служат директивы препроцессора (прекомпилятора).

Директива препроцессора — это строка, начинающаяся с символа **#** и заканчивающаяся символом конца строки. Обычно директивы препроцессора выполняют подстановку, замену или удаление участка текста программы перед ее компиляцией.

Директива *#include* производит подстановку содержимого указанного файла в то место модуля, где эта директива записана. Директива может быть записана в двух формах:

- **#include** <*файл.h*> — подключение заголовочного файла; указанный файл берется из каталога, в котором хранятся заголовочные файлы стандартных библиотек;
- **#include** *"файл.h"* — подключение заголовка; указанный файл берется из текущего каталога, а если он там отсутствует, то из каталога заголовочных файлов стандартных библиотек.

При задании имени файла можно указывать относительный и абсолютный путь к нему.

Директива *#define* определяет макрос. Она позволяет связать указанный идентификатор (замещаемую часть) с последовательностью лексем (замещающей частью). В результате все вхождения данного идентификатора, встречающиеся после этой директивы, будут заменены на указанные лексемы. Директива может быть записана в двух формах: простого макроса и макроса с параметрами.

Синтаксис простого макроса:

```
#define <идентификатор>< последовательность_лексем>
```

Идентификатор макроса в тексте программы заменяется последовательностью лексем. Например:

```
#define max_kart 100 //заменяет max_kart на 100
```

Синтаксис макроса с параметрами:

```
#define <идентификатор>(<формальные параметры>)
<последовательность_лексем>
```

В программе такой макрос вызывается с помощью оператора:

```
<идентификатор>(<фактические параметры>);
```

При вызове макроса происходят две замены. Сначала идентификатор макроса в исходном тексте программы заменяется последовательностью лексем, а затем формальные параметры в теле макроса заменяются значениями фактических параметров оператора вызова макроса.

С помощью такой формы макроса можно записывать простейшие функции, которые не будут компилироваться отдельно, а будут подставлены непосредственно в текст программы на месте их вызова.

Например:

```
#include <stdio.h>
#define max(x, y) (x > y ? x : y)
int main()
{
  int x, y;
  scanf("%d", &x);
  scanf("%d", &y);
  printf("%d", max(x, y));
  return 0;
}
```

В этом примере макрос **max** сравнивает значения двух переменных **x** и **y** и присваивает идентификатору макроса **max** значение наибольшей из них.

В директиве **#define** замещающая часть может отсутствовать. Например:

```
#define VALUE1
```

Такая форма обычно используется в связке с другими директивами препроцессора.

Директива *#undef* отменяет определение макроса. Например:

```
#undef max_kart // разрывает связь между max_kart и 100
```

Все вхождения идентификатора **max_kart**, встречающиеся после этой директивы, на 100 заменяться не будут. Более того, после директивы **#undef** идентификатор **max_kart** с помощью новой ди-

ректива **#define** может быть связан с другим значением, и все его дальнейшие вхождения будут заменяться новым значением.

Директивы условной компиляции вызывают обработку частей программы препроцессором в зависимости от условий. К этим директивам относятся *#if, #ifdef, #ifndef, #else, #elif, #endif*.

Директива **#ifdef** обрабатывает часть программы, если идентификатор, указанный в директиве, определен ранее в программе с помощью директивы **#define**. В противном случае эта часть программы будет удалена. Действие директивы **#ifdef** распространяется на часть программы до директивы **#else**, а при ее отсутствии до директивы **#endif**.

Общий синтаксис директивы:

```
#if (<идентификатор1>)  // если идентификатор1 объявлен,
{...;}                  // то сохраняется эта часть
                        // программы
[#elif (<идентификатор2>) // если идентификатор1
                          // не объявлен,
                          // а идентификатор2 объявлен,
{...;}]                   // то выполняется эта часть
                          // программы
[#else {...;}]    // если ни один из идентификаторов
                  // не объявлен, то выполняется
                  // эта часть программы
#endif
```

Например:

```
#define INTEGERS
...
#ifdef INTEGERS // идентификатор INTEGERS объявлен,
  int i, j;     // поэтому переменные i и j имеют тип int
#else           // если бы идентификатор INTEGERS
                // не был объявлен,
  double i, j;  // то переменные i и j имели бы
                // тип double
#endif
...
```

Действия, выполняемые директивой **#ifndef**, полностью противоположны действиям директиве **#ifdef**. Эта директива сохраняет последующий код, если идентификатор, следующий за ней, не был объявлен, и удаляет, если был объявлен. Синтаксис и применение этой директивы полностью аналогичны директиве **#ifdef**.

5.5. Указатели

Указатель является отдельным типом данных в языке С. Указатель — это переменная, содержащая адрес другой переменной. Чтобы получить прямой доступ к области памяти, где хранится инфор-

мация, необходимо иметь некоторый тип данных, способный хранить адрес памяти и производить прямые операции с памятью. Этот тип данных и есть указатель — тип, напрямую указывающий на некоторую область памяти.

Как и любая другая переменная, указатель в программе должен быть объявлен. Для объявления указателя используется символ «*». Синтаксис объявления указателя:

```
[<класс_памяти><тип_данных> * <имя>;
```

Например:

```
int * i ; // i - указатель на тип int
float * f // f - указатель на тип float
```

Размер области памяти, на которую ссылается указатель, зависит от типа данных, размер же области памяти, занимаемой самим указателем, фиксированный (4 байта).

При объявлении указателей (так же, как и других типов данных) может быть указан модификатор *const*. Использование ключевого слова **const** определяет, что указатель и (или) указываемая переменная не должны изменяться.

Чтобы запретить изменение указываемой переменной, модификатор **const** размещается перед типом указателя:

```
int tabn1, tabn2;
const int *p = &tabn1; // p - указатель на константу
                       // типа int
*p = 1000; // Присвоение объекту tabn1, на который
           // указывает указатель на константу p,
           // недопустимо
p = &tabn2; // Присвоение значения указателю допустимо
```

Чтобы запретить изменение самого указателя, модификатор **const** следует разместить после типа указателя:

```
int * const p=&tabn1;  // Указатель-константа на int
// Присвоение объекту tabn1, на который указывает
// указатель на константу p, допустимо
*p=1000;
// Присвоение нового значения указателю на константу
// недопустимо
p=&tabn2;
```

Указатели с модификатором **const** называются *константными указателями*.

Если необходимо, чтобы указываемая переменная не изменялась, нужно записать **const** перед типом указателя.

Над указателями можно выполнять следующие операции:

- получение адреса переменной (&);
- операция косвенной адресации (*);
- операция присваивания (=);

• арифметические операции (+, -, ++, --);
• операция индексации ([ ]);
• операции сравнения.

Операция получения адреса «&» возвращает адрес памяти, в которой хранится значение переменной. Операция может выполняться над переменной любого типа. Например:

```
int i;
int* p;
p=&i; // в переменной p записан адрес участка памяти,
      // в котором хранится переменная i
```

Операция получения адреса может применяться и к указателям. Например:

```
int * p1;
int **p2;
p2 = &p1;
```

Операция косвенной адресации (*разадресации*) «*» возвращает переменную, на которую ссылается указатель. Операция может выполняться только над переменными типа «указатель». Тип переменной, возвращаемой операцией «**», тождественен типу, на который указывает указатель.

В сочетании с *операцией присваивания* операция позволяет записывать заданное значение в память или читать значение, хранящееся в указанном участке памяти. Например:

```
int i, j;
int * p;
p = &i; // в переменную p записан адрес переменной i
*p = 1; // по адресу, хранимому в переменной p,
        // записано 1
        // в результате переменной i присвоено значение 1
j = *p; // в переменную j записано значение ячейки
        // памяти, адрес которой хранится в переменной
        // p (т.е. 1)
```

Арифметические операции позволяют изменить (увеличить или уменьшить) адрес участка памяти, хранящийся в указателе. При этом следует помнить, что разные типы данных занимают в памяти разные по размеру участки. Например, размер участка памяти, отводимой под хранение переменной типа **char**, равен 1 байту, а переменной типа **int** — 4 байтам. Соответственно, арифметические операции над указателями перемещают указатель на заданное количество адресуемых участков. Изменение значения указателя можно вычислить следующим образом: при изменении указателя **p = p + i**, где **p** — указатель на некоторый тип; **i** – целое число, физический адрес, хранимый в указателе, изменится на **i*s**, где **s** — размер типа, на который указывает указатель **p**.

Аналогично для указателей работают операции вычитания, инкремента и декремента. Например:

```
double *p;
p++; // значение указателя увеличится на 8 байт
```

Операция индексации [] совмещает действия арифметических операций и операции косвенной адресации и позволяет читать или записывать значения в ячейку памяти, смещенную на заданное количество позиций.

Например

```
i = p[5];
i = *(p + 5); // выражения эквивалентны
```

Операции сравнения для указателей выполняются аналогично операциям сравнения для других типов данных.

Например:

```
#include <stdio.h>
int main()
{
  int i=2, j=7, *p1, *p2, *p3;
  p1 = &i;
  p2 = &i;
  p3 = &j;

  if (p1 == p2)
    printf("YES ");
  else
    printf("NO \n"); // YES, так как p1 и p2 содержат
                     // адрес одной и той же переменной

  if (*p1 > *p3)
    printf("YES ");
  else
    printf("NO \n"); // NO, так как значение, записанное
                     // по адресу в p1, меньше значения,
                     // записанного по адресу в p2

    return 0;
}
```

5.6. Обработка массивов

Массив — это непрерывная по расположению в памяти поименованная совокупность данных, состоящая из фиксированного числа элементов одинакового типа.

Как и другие переменные, массив в программе должен быть объявлен. Синтаксис объявления массива:

```
[<класс_памяти>]<тип_данных><имя_массива>[<размер>]
```

где размер — количество элементов массива. Количество элементов указывается в квадратных скобках.

Массив может иметь любой класс памяти, кроме **register**.

Массивы различаются размерностью. Говорят о одномерном массиве (ряде данных), двумерном массиве (таблице) и так далее — N-мерном массиве. В языке C/C++ предусмотрено понятие только одномерного массива. Многомерные массивы представляются как массив, состоящий из элементов типа массив.

```
// объявление одномерного массива из 10 целых чисел
int a[10];
// объявление двумерного массива вещественных
// чисел, состоящего из 3 строк и 4 столбцов
float b[3][4];
```

Для обращения к конкретному элементу массива используется операция индексации «[]». Индекс — это порядковый номер элемента в массиве в соответствующем измерении. Индексация элементов начинается с 0.

Например, при объявлении массива **int a[4]** этот массив будет состоять из четырех элементов: **a[0]**, **a[1]**, **a[2]**, **a[3]**.

При обращении к элементу двумерного массива задаются два индекса: первый указывает номер строки, а второй — номер столбца, на пересечении которых находится элемент.

Например, **b[1][3]** — это элемент, находящийся на пересечении первой строки и третьего столбца двумерного массива **b** (номера строк и столбцов индексируются с 0).

Индекс, указываемый в операции индексации, может быть целой константой, целой переменной или любым выражением целого типа.

В памяти массив хранится в виде непрерывного блока данных размером $(x_0 \cdot x_1 \cdot x_2 \cdot \ldots \cdot x_{n-1}) \cdot S$ байт, где n — количество измерений массива; x_1 — x_{n-1} — количество элементов массива в каждом измерении; S — размер типа данных элементов массива. Например, массив **int x[3][4]** будет занимать в памяти $3 \cdot 4 \cdot 4 = 48$ байт.

Так как память линейна, то массив любой размерности в памяти хранится как одномерный массив, состоящий из $x_0 \cdot x_1 \cdot x_2 \cdot \ldots \cdot x_{n-1}$ элементов. Номер элемента в этом массиве можно вычислить по формуле

$$number = \sum_{k=0}^{N-2} \left(i_k \cdot \sum_{j=k+1}^{N-1} x_j \right) + x_{N-1},$$

где i_k — индекс элемента в k-м измерении. Например, в массиве **x[2][4][3]** элемент **x[1][3][2]** будет иметь номер

$$1 \cdot 4 \cdot 3 + 3 \cdot 3 + 2 = 23.$$

При объявлении массива можно одновременно выполнить его инициализацию набором значений, заключенных в фигурные скобки. Количество инициализирующих значений не обязательно должно совпадать с количеством элементов массива. Если значений меньше, то оставшиеся значения элементов массива не определены. Например:

```
// элементы массива последовательно принимают
// перечисленные значения
int a[5]={5, 10, 6, 7, 8};
// перечисленные значения присваиваются первым трем
// элементам массива, последние два элемента
// не определены
int x[5]={4, 2, 12}
```

При инициализации можно объявлять массив без явного указания его размера. Размер массива будет установлен автоматически по количеству инициализирующих значений.

При инициализации многомерных массивов разбиение элементов по размерностям происходит в соответствии с объявлением массива:

```
int a[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

В этом массиве первые пять элементов относятся к нулевой строке массива, следующие пять — к первой, в последней строке определено значение одного элемента.

Можно объявить и инициализировать массив следующим образом:

```
int a[3][5]={{1,2,3},{4,5,6,7,8},{9,10,11}}
```

Здесь группы значений, перечисленных в круглых скобках, инициализируют каждую строку двумерного массива.

При объявлении и инициализации многомерных массивов разрешается не указывать размер только самого левого измерения.

Элемент массива в программе можно использовать как любую переменную: их можно вводить, присваивать им значения, выполнять над ними любые операции.

Пример 5.6.1

Ввести одномерный массив, состоящий из 10 элементов. Вычислить среднее арифметическое массива. Все элементы массива, меньшие среднего арифметического, заменить нулем. Вывести полученный массив и количество замен.

```
#include <stdio.h>
#define N 10
int main()
{
  // объявление массива из 10 элементов
```

```
    int x[N];
    // k — количество элементов, меньших среднего
    // арифметического
    int i, k=0;
    float sr=0;
    // sr — среднее арифметическое элементов массива
    for (i = 0; i < N; i++)
    {
      scanf("%d", &x[i]);  // ввод и суммирование
                           // очередного элемента
      sr += x[i];
    }
    sr /= N;  // вычисление среднего арифметического
    for (i = 0; i < N; i++)
      // сравнение очередного элемента со средним
      // арифметическим
      if (x[i] < sr)
      {
        // обнуление элементов, меньших среднего
        // арифметического
        x[i] = 0;
        k++;  // увеличение количества замен
      }
    for (i = 0; i < N; i++)
      printf("%d ", x[i]);  // вывод обновленного массива
      printf("%d %f", k, sr); // вывод количества замен
      return 0;
}
```

Пример 5.6.2

Ввести двумерный массив размером 3×4. Поменять местами первый максимальный и первый минимальный элементы массива. Полученный массив вывести.

```
#include <stdio.h>
int main()
{
  int x[3][4];
  int max, imx, jmx, min, imn, jmn, y;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
      scanf("%d", &x[i][j]); // ввод элементов массива
  max =x[0][0];
  imx = 0;
  jmx = 0;
  /* за максимум принимается первый элемент массива.
  Запоминается номер строки и столбца, в которых
  расположен максимум */
  min = x[0][0];
  imn = 0;
```

```
  jmn = 0;
  /* за минимум принимается первый элемент массива.
  Запоминается номер строки и столбца, в которых
  расположен минимум */

  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
      // если очередной элемент больше максимума
      if (x[i][j] > max)
      {
        // меняется максимум и его расположение
        max = x[i][j]; imx = i; jmx = j;
      }
      else
      // если очередной элемент меньше минимума
      if (x[i][j] < min)
      {
        // меняется минимум и его расположение
        min = x[i][j]; imn = i; jmn = j;
      }
      y = x[imx][jmx];
      // максимальный и минимальный элемент меняются
      // местами
      x[imx][jmx] = x[imn][jmn];
      x[imn][jmn] = y;
    for (int i = 0; i< 3; i++)
    //вывод обновленного массива
    {
      for (int j = 0; j < 4; j++)
        printf("%d ", x[i][j]);
        printf("\n");
    }
  return 0;
}
```

Имя массива — это адрес памяти, начиная с которого расположен массив, т.е. адрес первого элемента массива. Этот адрес не может быть изменен, поэтому в большинстве случаев можно считать, что массив является константным указателем на область хранения его данных. Например:

```
int x[10];   // x является указателем на первый элемент
             // массива
int j;
int * p;
p=x;         // эти два оператора
p =&x[0];    // приведут к одинаковому результату
j = x[3];    // эти два оператора
j = * (p+3); // приведут к одинаковому результату
```

Пример 5.6.3

Дан одномерный массив, состоящий из пяти элементов. Все положительные элементы массива увеличить на 3, остальные уменьшить на 3. Вывести обновленный массив.

```
#include <stdio.h>
int main()
{
// объявляем и инициализируем массив из пяти элементов
  int x[] = {5, -10, 15, -20, 25};
  int i, *p, *p1;
  p1 = x;    // указатель p1 установлен на начало
             // массива x
  for (p = x; p < p1 + 5; p++) // указатель p движется
                               // по массиву x
  {
    if (*p>0) // если значение, записанное по
              // указателю, больше 0
      *p = *p + 5; // оно увеличивается на 5
    else
      *p = *p - 5; // иначе уменьшается на 5
  }
  for (p = x; p < p1+5; p++)
    printf("%d\n", *p);
  return 0;
}
```

Различают статическое и динамическое выделение памяти.

При *статическом выделении* объем необходимой памяти известен на этапе компиляции. Операции выделения и освобождения памяти при этом выполняются автоматически: память выделяется при объявлении переменной (массива переменных) и освобождается при завершении времени жизни переменной.

Динамическое выделение памяти происходит во время работы программы. В этом случае операции выделения и освобождения памяти должны быть прописаны в программе. Динамическое выделение обычно используется в том случае, когда объем необходимой памяти заранее неизвестен. Доступ к динамически выделенной памяти производится с помощью указателей.

Для управления динамическим выделением памяти используются функции, прототипы которых прописаны в файле **alloc.h** и **stdlib.h**: *calloc, malloc, realloc, free*.

Функция **calloc** динамически выделяет блок памяти и инициализирует его нулями. Синтаксис вызова функции:

```
calloc(size_t n, size_t s),
```

где **n** — количество элементов, под хранение которых выделяется память; **s** — размер памяти в байтах для хранения одного элемента.

Функция возвращает указатель на выделенный блок в случае успешного выделения памяти или NULL, если память выделить не удалось. Например:

```
int *p;
p = calloc(5, sizeof(int));
```

выделяется память размером $5 \cdot 4 = 20$ байтов, в случае успешного выделения в **p** записывается адрес начала выделенного блока.

Функция **malloc** динамически выделяет блок памяти без инициализации его содержимого. Синтаксис вызова функции:

```
malloc(size_t s)
```

где **s** — размер выделяемого блока в байтах.

Так же, как и функция **calloc**, **malloc** возвращает указатель на начало выделенного блока памяти или NULL. Например:

```
int *p;
p = malloc(5*sizeof(int));
```

Функция **realloc** производит перераспределение уже выделенной памяти, увеличивая или уменьшая ее размер. Синтаксис вызова функции:

```
realloc(void *bl, size_t s)
```

где **bl** — указатель на блок предварительно выделенной динамической памяти; **s** — новый размер блока **bl**.

Если функция **realloc** вызвана для увеличения размера предварительно выделенной памяти, то по возможности к выделенной памяти добавляется непрерывной участок требуемого размера (весь блок должен занимать непрерывный участок памяти). Если такой возможности нет, то функция производит выделение нового блока памяти нужного размера, копирование в него данных старого блока и освобождение старого блока. При уменьшении размера требуемой памяти ненужный участок памяти освобождается. Функция **realloc** возвращает указатель на новый выделенный блок памяти или NULL, если память не была выделена или ее размер равен нулю.

Функция **free** освобождает блок памяти, выделенный функциями **calloc**, **malloc** или **realloc**. Синтаксис вызова функции:

```
free(void* bl);
```

где **bl** — указатель на освобождаемый блок памяти.

Пример 5.6.4

Ввести динамический массив, состоящий из *n* элементов. Подсчитать количество элементов массива, меньших среднего арифметического.

```
#include <stdio.h>
#include <stdlib.h>
int main()
```

```
{
  int n;
  scanf("%d", &n);  // ввод количества элементов
  int i, k = 0;
  float sr = 0;
  // динамическое выделение памяти
  int *x = (int *) calloc(n, sizeof(int));
  if (x)  // если память выделилась успешно
  {
    for (i = 0; i < n; i++)
    {
      scanf("%d", &x[i]); // ввод очередного элемента
      sr += x[i]; // добавление элемента к сумме
    }
    sr /= n;  // вычисление среднего арифметического
    for (i = 0; i < n; i++)
      if (x[i] < sr)
      {
        k++; // подсчет количества элементов, меньших
             // среднего арифметического
      }
      printf("% f %d", sr, k);  // вывод результатов
      free(x);
  }
  else
    printf("memory error\n"); // вывод сообщения об
                              // ошибке выделения памяти
    return 0;
}
```

Пример 5.6.5

Ввести динамический массив, состоящий из *n* элементов. Вставить в массив после первого максимального элемента нули, количество которых равно разности максимального и минимального элементов.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i, n, m, k, r;
  scanf("%d", &n); // ввод количества элементов массива
  int *x = (int *)malloc(n*sizeof(int));
  // динамическое выделение памяти для хранения массива
  if (x)
  {
    for (i = 0; i<n; i++)
      scanf("%d", &x[i]); // ввод очередного элемента
                          // массива
      m = x[0];
      k = 0;
    for (i = 1; i<n; i++) // нахождение максимального
                          // элемента
```

```
        if (x[i]>m)          // и его места в массиве
        {
          m = x[i];
          k = i;
        }
        r = x[0];
        for (i = 1; i < n; i++) // нахождение
                                // минимального элемента
        if (x[i] < r)
          r = x[i];
          // переопределение выделенной для хранения
          // массива памяти
          x = (int *) realloc(x, (n+m-r)*sizeof(int));

        if (x) // если память выделена успешно
        {
          for (i = 0 ; i <n-k; i++)
          // элементы массива, расположенные после
          // максимального, сдвигаются вправо
          for (i = k+1; i <k+ m-r+1; i++)
            x[n+m-r-i] = x[n-i];
          // добавление в массив нулей
            x[i] = 0;
          // вывод обновленного массива
          for (i = 0; i < n+m-r ; i++)
            printf("%d ", x[i]);
            free(x);  // освобождение памяти
        }
        else
          printf ("memory error\n");
    }
    else
      printf("memory error\n");
      return 0;
}
```

Динамическое выделение памяти для многомерных массивов выполняется с помощью выделения нескольких областей памяти. В первой области, которая непосредственно задается указателем, хранятся указатели на указатели или области памяти с данными более низкого уровня.

Пример 5.6.6

Ввести динамический двумерный массив, состоящий из *n* строк и *m* столбцов. Найти среднее арифметическое введенного массива.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n, m, s=0;
  int i, j;
```

```
  int **x;  // x - указатель на указатель
  scanf ("%d %d", &n, &m);
  // выделение памяти под хранение указателей
  x = (int **)calloc(n, sizeof(int));
  for (i = 0; i < n; i++)
  {
    // выделение памяти под хранение каждой строки
    // массива
    x[i] = (int *)calloc(m, sizeof(int));
    for (j = 0; j < m; j++)
    {
      scanf("%d", &x[i][j]);
      s += x[i][j];
    }
  }
  printf("%f", (float)s/n*m);
  free(x);
  return 0;
}
```

Такой метод выделения памяти является очень трудоемким, особенно с увеличением размерности массива. Поэтому при динамическом выделении памяти целесообразно многомерные массивы хранить, как одномерные, пересчитывая их индексы.

Пример 5.6.7

Листинг из примера 5.6.6 с представлением двумерного массива как одномерного выглядит так:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n, m, s=0;
  int i, j;
  int *x;
  scanf ("%d %d", &n, &m);
  // выделение памяти под хранение всего массива
  x = (int *)calloc(n*m, sizeof(int));
    for (i = 0; i < n; i++)
    {
      for (j = 0; j < m; j++)
      {
        scanf ("%d", &x[i*n+j]); // использование
                                 // пересчитанных
        s += x[i*n+j];           // индексов массива
      }
    }
    printf("%f", (float)s/n*m);
    free(x);
    return 0;
}
```

5.7. Функции

Функция — это поименованный блок программы, состоящий из последовательности операторов. Функция вызывается на выполнение другими функциями через набор параметров.

Функция состоит из заголовка и тела (текста) функции. *Заголовок функции* задает имя функции, типы аргументов и тип возвращаемого значения. *Тело функции* включает объявления локальных переменных, определения типов, операторы.

Вызову функции в программе должно предшествовать либо объявление функции (прототип функции), либо определение функции (текст функции).

Прототип функции — это заголовок функции без тела функции, заканчивающийся разделителем (;). Прототип функции обычно помещают в заголовочный файл, который подключают к соответствующему файлу кодов с помощью директивы компилятора #include.

Синтаксис прототипа функции:

```
<тип_возвращаемого_значения><имя_функции>
(<тип><формальный_параметр1>
[,<тип><формальный_параметр2>,…]);
```

Например:

```
// функция poisk принимает 2 значения типа массив char
// и int,
// возвращает значение типа int
int poisk (char tabn[], int nomer);
// Параметры можно задавать без имен переменных
int poisk (char*, int);
// Функция init не принимает и не возвращает значений
void init();
// функция sr принимает значение типа int, возвращает
// float
float sr (int n);
```

Прототип функции сообщает компилятору тип возвращаемых данных, количество параметров и их тип, порядок их следования. Компилятор использует прототип функции для проверки правильности вызовов функции.

Определение функции — это заголовок и тело функции.

Синтаксис определения функции:

```
<тип_возвращаемого_значения><имя_функции>
(<тип><формальный_параметр1>
[,<тип><формальный_параметр2>,...]);
{
... //объявление переменных и операторы
  [return <значение >;]
}
```

где значение — это возвращаемое значение указанного в заголовке функции типа.

Оператор *return* осуществляет передачу управления из текущей функции в вызывающую. Существует две формы оператора **return**:

```
return;
return (выражение);
```

Первая форма обеспечивает простую передачу управления из текущей функции в вызывающую. Применяется для выхода из функции типа **void**.

Вторая форма обеспечивает передачу управления и возвращение значения выражения в вызывающую функцию. Применяется в функциях типа, отличного от **void**. Тип возвращаемого значения должен совпадать с типом функции.

Оператор вызова функции служит для вызова вызываемой функции в теле вызывающей функции. Он может быть представлен двумя способами:

• как простой оператор без возврата значений через оператор **return**;

• как операнд в выражении при возврате значений через оператор **return**.

Вызов функции осуществляется либо прямо по имени функции, либо через указатель на функцию.

Синтаксис вызова функции по имени:

```
<имя функции>(<фактический параметр_1>[, …])),
```

Например:

```
int main() //вызывающая функция
{
  floats;
  int kol;
  ...
  s = sr(kol); // вызов функции sr с фактическим
               // параметром kol
  ...
}
```

Оператор вызова функции передает управление вызываемой функции, копии фактических параметров присваиваются формальным параметрам вызываемой функции. Значения фактических входных параметров перед вызовом функции должны быть определены. Оператор **return** передает управление обратно в вызывающую функцию. Необходимо заметить, что формальные и фактические параметры изолированы друг от друга.

Передача параметров может выполняться двумя способами:

• передача значений переменных — вызываемой функции передаются копии значений переменных вызывающей функции;

• передача значений указателей — вызываемой функции передаются копии значений указателей — адреса переменных вызывающей функции.

Пример 5.7.1

Даны два числа. Вывести наибольшее из них. Для нахождения наибольшего числа использовать функцию.

1. Передача параметров по значениям переменных:

```
#include <stdio.h>
// вызываемая функция, x и y — формальные параметры
int max(int x, int y)
{
  return (x > y ? x : y);
}
int main()
{
  int a,b,c;
  scanf("%d %d", &a, &b);  // ввод исходных данных
  c = max(a, b);     // вызов функции, a и b -
                     // фактические параметры
  printf("%d", c);
  return 0;
}
```

Этот способ передачи параметров, как правило, используется для небольших по объему данных: данных базовых типов.

2. Передача значений указателей:

```
#include <stdio.h>
// вызываемая функция, x и y — формальные
// параметры (используются указатели на них)
int max(int *x, int *y)
{
  return (*x > *y ? *x : *y);
}
int main()
{
  int a,b,c;
  scanf("%d %d", &a, &b);
  // вызов функции, a и b — фактические параметры
  // (используется операция адресации)
  c = max(&a, &b);
  printf("%d", c);
  return 0;
}
```

Этот способ целесообразно использовать при передаче данных большого объема: массивов, структур.

Передача значений указателей используется и при необходимости изменения значений переменных вызывающей функции.

При передаче параметров по значениям переменных значения фактических параметров копируются в формальные параметры, поэтому при выходе из вызываемой функции никаких изменений значений фактических параметров в вызывающей функции не происходит. При этом вызываемая функция может вернуть вызывающей с помощью оператора **return** только одно значение. Решить данную проблему позволяет передача значений указателей.

Пример 5.7.2

Ввести два числа *x* и *y*. Поменять значения переменных *x* и *y* так, чтобы числа располагались в порядке возрастания. Для обмена переменных использовать функцию.

1. Передача параметров по значению переменных:

```
#include <stdio.h>
void change(float x, float y)
{
  float z;
  if (y < x) // если вторая переменная меньше первой
  {
    z = x;  // переменные меняются местами
    x = y;
    y = z;
  }
}

int main()
{
  float a,b;
  scanf ("%f %f", &a, &b);
  change (a, b); // в вызывающей функции переменные
                 // местами не поменялись
  printf("%f %f", a, b);
  return 0;
}
```

2. Передача значений указателей:

```
#include <stdio.h>
// формальные параметры — значения, записанные
// по адресам y и x
void change(float *x, float *y)
{
  float z;
  if (*y < *x)
  {
    z = *x;
    *x = *y;
    *y = z;
  }
}
```

```
int main()
{
  float a,b;
  scanf ("%f %f", &a, &b);
  // фактические параметры — адреса переменных a и b
  change (&a, &b);
  printf("%f %f", a, b);
  return 0;
}
```

В результате вызываемая функция переставит значения, записанные в тех участках памяти, адреса которых были переданы из вызывающей функции.

Обычно одни функции вызывают другие функции согласно иерархической структуре программы. Однако любая функция может вызвать на выполнение и саму себя. Такой вызов функции называется рекурсивным.

Рекурсивная функция — это функция, которая вызывает на выполнение сама себя.

При каждом рекурсивном вызове функции для формальных параметров выделяется новая область памяти, так что их значения из предыдущих вызовов не теряются, но в каждый момент времени доступны только значения текущего вызова.

Пример 5.7.3

С клавиатуры ввести массив из *n* чисел. Найти максимальное из них.

Нахождение максимального числа можно рассматривать как рекурсивный алгоритм, в котором каждое очередное число сравнивается с максимальным из предыдущих.

```
#include <stdio.h>
#include <stdlib.h>
// вызываемая функция находит максимальное
// из двух чисел
int max(int x, int m)
{
  return x > m ? x : m;
}

int main()
{
  int n;
  int i, m;
  scanf("%d", &n);
  int *x = (int *)malloc(n*sizeof(int));
  for (i = 0; i < n; i++)
    scanf("%d", &x[i]);
  m = x[0];  // за максимум принимается нулевой элемент
             // массива
```

```
    for (i = 1; i < n; i++)
      m = max(x[i], m); // функция max вызывается
                        // рекурсивно
    // очередной элемент массива сравнивается с текущим
    // максимумом
    printf("%d ", m);
    free(x); // освободить память
    return 0;
}
```

Допускается любое количество рекурсивных вызовов функции. Их число ограничивается только размером области памяти, в которой сохраняется информация о каждом вызове. При слишком большом числе рекурсивных вызовов может произойти переполнение памяти.

5.8. Функции ввода-вывода данных

В языке С нет встроенных средств ввода-вывода, и вместо них используются библиотеки ввода-вывода **stdio.h** и **conio.h**.

Рассмотрим наиболее часто используемые функции этих библиотек.

Функция *printf* осуществляет форматированный вывод данных на экран. Синтаксис функции:

```
int printf (const char* <формат>,
  [<аргумент1>, <аргумент2>, ...]);
```

где формат — символьная строка, задающая формат печати данных; аргумент1, аргумент2, ... — список печатаемых данных.

В строке формата указываются:

• символы, непосредственно выводимые на печать. К их числу относятся и специальные символы (например, \n — перевод строки, \t — табуляция);

• спецификаторы форматирования, задающие формат печати данных. Спецификаторы форматирования начинаются со знака %. Каждому данному из списка аргументов должен соответствовать свой спецификатор форматирования.

Синтаксис спецификатора форматирования:

```
%[флаги] [ширина][.точность]тип
```

Флаги — это необязательная последовательность символов, являющаяся модификатором форматов (табл. 5.6).

Ширина — необязательный параметр, задающий минимальную ширину поля. Если длина выводимых на экран данных меньше этого значения, они дополняются пробелами или нулями до заданного значения ширины. Если длина данных больше заданного зна-

Таблица 5.6

Флаги форматирования

Флаг	Форматирование при наличии флага	Форматирование при отсутствии флага
–	Выравнивание по левой границе поля заданной ширины	Выравнивание по правой границе
+	Добавляется знак «+» для положительных чисел (для знаковых форматов)	Печатается только знак «-» для отрицательных чисел
0	Добавляются нули до достижения минимальной заданной длины. В случае целых форматов или наличия флага «-» данный флаг игнорируется	Нули не добавляются
" " (пробел)	Добавляется пробел перед печатью положительного значения знакового типа. При наличии флага "+" данный флаг игнорируется	Пробел не добавляется
#	Добавляется префикс системы исчисления для восьмеричных и шестнадцатеричных форматов	Префикс системы счисления не добавляется

чения, они выводятся целиком. Используются две формы задания ширины:

• n — ширина поля составляет n символов, дополнение до заданной ширины производится пробелами;

• 0n — ширина поля составляет n символов, дополнение до заданной ширины производится нулями.

Точность — необязательный параметр, определяющий точность печатаемого значения. Действия, выполняемые этим параметром, зависят от типа выводимых данных (табл. 5.7).

Тип — это символ, задающий формат вывода данных. Типы представлены в табл. 5.8.

Таблица 5.7

Точность при форматированном выводе

Точность	Выполняемые действия
c, C	Нет действий
d, i, u, o, x, X	Если длина числа меньше точности, то слева оно дополняется нулями, если больше — печатается полностью
e, E, f	Задает количество знаков после десятичной точки, число округляется до заданного количества знаков
g, G	Задает количество печатаемых значащих разрядов
s, S	Задает количество печатаемых символов, начиная с начала строки, все остальные символы не печатаются

Таблица 5.8

Типы при форматированном выводе

Тип	Аргумент	Формат вывода
d, i	Целое число	Десятичное целое число со знаком
c	Символ	Символ с заданным кодом
s	Строка символов	Строка символов (соответствующий параметр указывает на начало строки)
e	Число с плавающей точкой	Число с плавающей точкой, экспоненциальная запись
f	Число с плавающей точкой	Число с плавающей точкой, десятичная запись
g	Число с плавающей точкой	Формат f или e. Используется наиболее компактный формат для данного выражения
u	Целое число	Десятичное целое число без знака
o	Целое число	Восьмеричное целое число без знака
x X	Целое число	Шестнадцатеричное целое число без знака

По умолчанию числа с плавающей точкой печатаются с точностью до шести знаков после десятичной точки.

Например, после выполнения программы:

```
#include <stdio.h>
int main()
{
  int i = 123;
  float x = 123.45;
  char ch = 'a';
  char fkart[] = "kart.dat";
  printf("%5d %-5d %10.3f %3c %10s %10.4s", i, i, x,
  ch, fkart, fkart);
}
```

на печать будет выведена следующая строка:

```
123 123 123.450 a kart.dat kart
```

Функция *scanf* выполняет ввод значений клавиатуры. Синтаксис функции:

```
int scanf (const char* <формат>,
  [<аргумент1>, <аргумент2>, …]);
```

Формат — символьная строка, задающая формат ввода данных. Спецификаторы формата в основном совпадают со спецификаторами функции **printf**, за исключением того, что в **scanf** отсутствует спецификатор **%g**, а для ввода целых чисел типа **short** применяется спецификатор **%h**.

Введенные с клавиатуры значения записываются в аргументы, поэтому в списке аргументов задаются указатели на переменные

(адреса участков памяти), в которые следует записать введенные значения, например:

```
int i;
float x;
scanf("%d %f", &i, &x);
```

Функции *sprintf* и *sscanf* работают аналогично функциям **printf** и **scanf**, но ввод/вывод осуществляется не с клавиатуры и на экран, а из строки и в строку символов. Подробно эти функции будут рассмотрены позже.

Точно так же, функции *fprintf* и *fscanf* используются для ввода/вывода данных из файла и в файл. Эти функции также будут рассмотрены позже.

Кроме функции форматированного ввода/вывода, в языке С используются функции неформатированного ввода/вывода. К ним относятся:

- *gets* — ввод строки с клавиатуры (применение данной функции не рекомендуется, так как может приводить к переполнению буфера в стеке);
- *getch*, *getchar* — ввод одного символа с клавиатуры. Функция возвращает ASCII код введенного символа;
- *puts* — вывод строки на экран;
- *putch*, *putchar* — вывод символа на экран;
- *fgets* — ввод строки из файла;
- *getc*, *fgetc* — ввод одного символа из файла;
- *fputs* — вывод строки в файл;
- *putc*, *fputc* — вывод символа в файл.

5.9. Обработка строк

В языке С нет специального типа данных, предназначенного для хранения строк. Строковые данные представляются в виде массива символов, оканчивающегося символом с кодом 0 (терминирующим нулем).

Например, слово Program будет храниться в памяти в виде:

'P'	'r'	'o'	'g'	'r'	'a'	'm'	0

Для хранения строковых данных используются статические и динамические массивы.

Для работы со строками в языке С используется набор функций, прототипы которых описаны в файле **string.h**.

Функция **sprintf** записывает форматированные данные в массив символов. Синтаксис функции:

```
int sprintf (char *<буфер>, const char *<формат>
  [, <аргумент1>, ... ]);
```

Форматированная строка выводится в область памяти, которая задается указателем <буфер>. Параметры, формат и аргументы соответствуют аналогичным параметрам функции **printf**.

Функция **sscanf** осуществляет чтение данных из строки по заданному формату. Синтаксис функции:

```
int sscanf(const char *<буфер>, const char *<формат>
  [, <аргумент1>, ... ]);
```

Исходные данные считываются из области памяти, которая задается указателем <буфер>. Параметры формат и аргументы соответствуют аналогичным параметрам функции **scanf**. Аргументы, в которых считываются данные, задаются указателями на соответствующие переменные.

Функция *stcmp* выполняет сравнение строк путем последовательного сравнения символов, расположенных в строках в одинаковых позициях. Синтаксис функции:

```
int strcmp(const char *<строка1>, const char *<строка2> );
```

Равными считаются строки, имеющие одинаковую длину и состоящие из одинаковых символов, расположенных в одинаковом порядке. Меньшей считается строка, у которой меньше код первого отличающегося символа. При совпадении символов, но разной длине, меньшей считается та строка, которая короче.

Функция возвращает значение:

- 0, если строки равны;
- –1, если первая строка меньше второй;
- 1, если первая строка больше второй.

Сравнение строк производится с учетом регистра. Для сравнения без учета регистра используется функция *stricmp*, имеющая такой же синтаксис и работающая аналогично.

Функция *strcat* выполняет объединение двух строк. Синтаксис функции:

```
char *strcat( char *<строка1>, const char *<строка2> );
```

Сформированная в результате работы функции строка содержит строку1, в конец которой добавлена строка2. Результат записывается в строку1.

Функция *strcpy* копирует одну строку в другую. Синтаксис функции:

```
char *strcpy( char *<строка1>, const char *<строка2> );
```

Строка2 копируется в область памяти, заданную указателем на строку1.

Функция *strlen* вычисляет длину строки в символах. Терминирующий нуль в число символов не включается. Синтаксис функции:

```
size_t strlen( const char *<строка1> );
```

Функция возвращает количество символов в строке1.

Функция *strstr* ищет одну строку в другой. Синтаксис функции:

```
char *strstr( constchar *<строка1>,
  constchar *<строка2> );
```

Функция ищет строку2 в строке1 и возвращает указатель на ее первое вхождение (ячейку памяти с первым символом подстроки) или NULL в случае отсутствия строки2 в строке1. Если строка2 является пустой, то функция возвращает указатель на строку1.

Для поиска в строке используются также функции, синтаксис которых аналогичен *strstr*:

- *strspn* — поиск первого вхождения любой подстроки строки2 в строке1;
- *strcspn* — поиск в строке1 первого символа, содержащегося в строке2 (возвращает индекс символа);
- *strspnp* — поиск в строке1 первого символа, не содержащегося в строке2;
- *strchr* — поиск в строке1 первого вхождения символа, заданного вторым аргументом функции;
- *strpbrk* — поиск в строке1 первого символа, содержащийся в строке2 (возвращает указатель на ячейку памяти, хранящую этот символ);
- *strrchr* — поиск в строке1 последнего вхождения символа, заданного вторым аргументом функции.

Функция *strtok* разбивает строку на подстроки. Синтаксис функции:

```
char *strtok( char *<строка1>, const char *<строка2>);
```

В качестве разделителей подстрок используются символы, входящие в строку2. Если указатель на строку1 не равен NULL, то разбиение производится с начала строки1, и функция возвращает указатель на первую найденную подстроку. Если указатель на строку1 равен NULL, то процесс разбиения продолжается, функция возвращает указатель на очередную выделенную подстроку или NULL, если больше подстрок нет.

Многие из перечисленных функций работы со строками имеют модификации, отличающиеся наличием буквы n в имени функции. Эти функции имеют дополнительный параметр, задающий количество символов от начала строки. В результате функция работает не со всей строкой, а только с ее первыми n символами:

- *strncmp* — сравнение первых n символов двух строк с учетом регистра;
- *strnicmp* — сравнение первых n символов двух строк без учета регистра;

• *strncat* — объединение первых n символов двух строк;

• *strncpy* — копирование n первых символов исходной строки в результирующую строку.

Перечислим еще несколько полезных функций работы со строками:

• *strlwr* — смена регистра всех символов строки на нижний (строчные буквы);

• *strupr* — смена регистра всех символов строки на верхний (заглавные буквы);

• *strset* — заполнение строки символом с заданным кодом (все символы данной строки будут заменены на заданный символ);

• *strnset* — заполнение первых *n* символов строки символом с заданным кодом;

• *strrev* — реверсирование строки (запись символов строки в обратном порядке).

Пример 5.9.1

Ввести предложение, в котором все слова разделены ровно одним пробелом. Подсчитать количество слов в предложении.

Количество слов в предложении будет на одно больше количества пробелов, так как последнее слово пробелом не заканчивается.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char s1[80], *p;   // объявление массива символов
                     // и указателя
  int  n, k;
  fgets (s1, sizeof(s1), stdin);  // ввод предложения
  n = strlen(s1);   // n - количество символов
                    // в предложении
  k = 0;
  p = s1;   // указатель установлен на начало
            // предложения
  for (p = s1; p < s1+n; p++)
  // указатель перемещается от первого до последнего
  // символа предложения
  if (*p == ' ') k++;  // подсчет количества пробелов
    printf("%d\n", k+1);
    return 0;
}
```

Пример 5.9.2

Ввести предложение, ввести два символа. Заменить в предложении все первые символы вторыми.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[80], *p;  // s1 - исходное предложение
    char sm1, sm2; // sm1 - заменяемый символ,
                   // s2 - заменяющий символ
    int n, k;
    fgets(s1,sizeof(s1),stdin);
    scanf("%c %c", &sm1, &sm2);
    n = strlen(s1);
    k = 0;
    for (p = s1; p < s1 + n; p++) // проход по всем
                                  // символам
      if (*p == sm1)  // запись заменяемого символа
        *p = sm2;     // вместо замещаемого
    printf("%s", s1);
    return 0;
}
```

Пример 5.9.3

Ввести предложение, в котором слова разделены пробелами. Вывести самое длинное слово.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char s1[80], s2[80]; // s1 - исходное предложение,
                       // s2 - искомое слово
  fgets(s1,sizeof(s1),stdin);
  char *pStr = s1, *pWord = NULL;
  int maxLength = 0;
  while (pWord = strtok(pStr, " "))
  // выделение слов, разделенных пробелами
  {
    int length = strlen(pWord);
    if (length>maxLength)  // если длина очередного
                           // слова больше максимума
    {
      strcpy(s2, pWord);  // слово копируется в s2
      maxLength = length; // меняется максимальная
                          // длина слова
    }
    // указатель устанавливается на NULL для продолжения
    // разбиения предложения на слова
    pStr = NULL;
  }
    printf("%s %d", s2, maxLength); // вывод результатов
    return 0;
}
```

5.10. Работа с файлами

Библиотека языка С поддерживает потоковый ввод/вывод.

Поток — это абстрактное понятие, относящееся к любому переносу данных от *источника* (или *поставщика*) к *приемнику* (или *потребителю*) данных.

На уровне потокового ввода/вывода обмен данными производится побайтно. Такой способ обмена в языке С используется при работе с файлами. Функции библиотеки ввода/вывода языка С, поддерживающие обмен данными с файлами на уровне потока, позволяют обрабатывать данные различных размеров и форматов.

Прототипы функций ввода/вывода содержатся в файле **stdio.h**.

Процесс работы с файлами включает в себя три основные стадии:

- открытие файла;
- считывание данных из файла или запись в файл;
- закрытие файла.

Для открытия файла используется функция *fopen*. Ее синтаксис:

```
FILE *fopen(const char *<имя_файла>,
  const char *<режим>);
```

При задании режима открытия файлов указываются:

- режим чтения/записи (табл. 5.9);
- режим трансляции символов (табл. 5.10).

Функция *fopen* возвращает указатель на специальную структуру FILE, связанную с открытым потоком, или NULL (в случае неудачного завершения операции открытия).

После окончания работы с файлом он обязательно должен быть закрыт. Отсутствие операции закрытия может привести к тому, что

Таблица 5.9

Режимы чтения/записи файла

Режим	Действие
r	Открытие файла для чтения. При отсутствии файла функция выдает ошибку (NULL)
w	Открытие файла для записи. Если файл существует, его старое содержимое будет удалено
a	Открытие файла для добавления. Данные добавляются в конец существующего файла
r+	Открытие файла для чтения и записи (файл должен существовать)
w+	Открытие пустого файла для чтения и записи. Если файл существует, то старое содержимое будет удалено
a+	Открытие файла для чтения и добавления в существующий файл

Таблица 5.10

Режимы трансляции символов

Режим	Действие
пусто	Текстовый режим. Некоторые сочетания байтов транслируются в один байт вместо нескольких, записанных в файле
t	Модифицированный текстовый режим. Дополнительная трансляция некоторых сочетаний байтов в один
b	Бинарный режим. Никакой трансляции не производится. Все байты считываются точно так, как записаны в файле

некоторые данные не будут записаны в файл. Для закрытия используется функция *fclose*. Ее синтаксис:

```
int fclose( FILE *<указатель> );
```

Ее аргументом является указатель на идентификатор потока, полученный от функции **fopen**.

Функция возвращает значение NULL в случае успешного завершения операции закрытия или код возникшей ошибки.

Организация типовой работы с файлами выглядит следующим образом:

```
FILE* f    // Открытие файла для чтения
// если функция открытия файла finput завершилась
// неудачно
if( (f = fopen( "finput", "r" )) == NULL )
  printf("ошибка открытия файла");
else
  {
  // обработка файла
  // закрытие файла выполняется только в случае его
  // успешного открытия
  if(fclose(f))  // если возникла ошибка закрытия файла
    printf( "ошибка закрытия файла");
}
```

При работе с файлами ввод/вывод данных осуществляется в место в файле, определяемое позицией курсора. *Курсор* — это абстрактный указатель, показывающий текущую позицию в открытом файле. Текущей позицией является порядковый номер от начала файла символа, с которого начнется чтение/запись при следующем обращении к файлу. При чтении из файла курсор сдвигается на количество прочитанных символов вперед и указывает на символ, с которого начнется чтение при следующем обращении к файлу. То же самое происходит и при записи. При открытии файла на чтение («r») или запись («w») курсор устанавливается на начало файла, при открытии файла на добавление («a») — на конец файла. Курсор может быть установлен вручную, но при этом его

позиция не может быть раньше начала файла и дальше текущей длины файла.

Функция *fgets* читает из заданного файла последовательность символов и записывает ее в указанную строку. Число символов (*n*) задает максимальное количество считываемых из файла символов. Чтение будет производиться до тех пор, пока не будет считано *n* – 1 символов или не встретится символ новой строки (этот символ также будет перенесен в результирующую строку). При успешном завершении функция вернет указатель на введенную строку, в случае ошибки — NULL.

Функция *fputs* записывает указанную строку в файл и возвращает неотрицательное значение (чаще всего 0) при успешном завершении или EOF при наличии ошибки.

Функции *fscanf* и *fprintf* выполняют ввод/вывод данных в файл в форматированном виде. Их синтаксис:

```
int fscanf(FILE* <указатель>, const char* <формат>,
  [* <аргумент1>, * <аргумент2>, ...]);
int fprintf (FILE* <указатель>, const char* <формат>,
  [<аргумент1>, <аргумент2>, ...]);
```

Эти функции работают аналогично функциям **scanf** и **printf**, при этом ввод/вывод данных осуществляется в файл, идентификатор которого задается указателем.

Функции *fread* и *fwrite* осуществляют блочный ввод/вывод данных. Обмен данными производится блоками фиксированной длины. Синтаксис функций:

```
size_t fread( void *<буфер>, size_t <размер_блока>,
size_t <количество_блоков>, FILE *<указатель>);
size_t fwrite( const void *<буфер>, size_t <размер_блока>,
size_t <количество_блоков>, FILE *<указатель>);
```

Функция **fread** читает данные из файла и записывает их в область данных, на которую указывает буфер. Число прочитанных байт равно произведению количества блоков и размера блока. Функция **fwrite** записывает из буфера в файл заданное количество блоков заданной длины. Обе функции возвращают количество полностью прочитанных/записанных блоков.

Функция *fseek* осуществляет установку курсора на определенное место в файле. Ее синтаксис:

```
int fseek( FILE *<указатель>, long <смещение>,
  int <позиция>);
```

Позиция здесь указывает, от какого места в файле следует отсчитывать смещение:

- SEEK_CUR — от текущей позиции курсора;
- SEEK_END — от конца файла;
- SEEK_SET — от начала файла.

Например:

```
// установить курсор на вторую позицию от начала файла
fseek (f, 2, SEEK_SET);
// установить курсор на конец файла
fseek (f, 0, SEEK_END);
// установить курсор на 1 позицию назад от текущей
fseek (f, -1, SEEK_CUR);
```

Функция *ftell* возвращает текущую позицию курсора в файле. Ее синтаксис:

```
long ftell( FILE *<указатель>);
```

Функция *feof* позволяет определить, достигнут ли конец файла. Ее синтаксис:

```
int feof( FILE *<указатель>);
```

При достижении конца файла функция возвращает ненулевое значение.

Пример 5.10.1

Прочитать входной файл и вывести в выходной файл все символы, имеющий четный номер.

```
#include <stdio.h>
int main()
{
  // объявление 2-х файловых переменных
  FILE *f_in, *f_out;
  char ch;
  int count = 0;    // счетчик элементов
  f_in = fopen("input.txt", "r"); // открытие файла
                                  // для чтения

  if (f_in != NULL)    // если файл открылся
  {
    f_out = fopen("output.txt", "w"); // открытие файла
                                      // для записи
    while (!feof(f_in))
    {
      ch = fgetc(f_in);
      // в выходной файл выводятся символы с четными
      // номерами
      if (++count % 2 == 0)
        putc(ch, f_out);
    }
    fclose(f_in);     //  закрытие файлов
    fclose(f_out);
  }

  else
    printf("file not opened  \n");
    return 0;
}
```

Пример 5.10.2

Прочитать строки, записанные во входном файле, и записать их в выходной файл.

```
#include <stdio.h>
int main()
{
  FILE *f_in, *f_out;
  char buffer[256];      // длина строки - 255
  f_in = fopen ("proba.txt", "r"); // открытие входного
                                   // файла
  f_out = fopen ("rez.txt", "w");  // открытие выходного
                                   // файла

  if (f_in != NULL)
  {
    while (!feof(f_in))
    {
      fgets(buffer, sizeof(buffer), f_in); // чтение
                                           // строки
      fputs(buffer, f_out);      // запись строки
    }
  }

  else
    printf("File not opened");

  fclose(f_in);  // закрытие файлов
  fclose(f_out);
  return 0;
}
```

5.11. Типы данных, определяемые пользователем

К типам данных, определяемым пользователем, относятся перечисления, структуры и объединения.

Перечисление — это простой целочисленный тип данных, представляющий собой упорядоченную последовательность идентификаторов (перечисляемых констант), принимаемых переменной. Каждому идентификатору назначается целое значение.

Объявление перечисления производится следующим образом:

```
enum <имя_перечисления>
{
  <константа1>,
  <константа2>
}<имя_создаваемой_переменной>;
```

Создаваемая переменная будет иметь тип **имя_перечисления** и сможет принимать только те значения, которые указаны в списке констант.

Имя перечисления может быть опущено, в этом случае повторное использование данного перечисления будет невозможно. Имя переменной также может быть опущено. В этом случае в момент объявления перечисления переменная создаваться не будет, переменная такого типа может быть создана позднее с помощью определения:

```
enum <имя_перечисления><имя_переменной>;
```

Каждой константе из списка присваивается уникальное целочисленное значение. По умолчанию каждая константа получает значение предыдущей константы, увеличенное на 1, при этом первая константа по умолчанию принимает значение 0. Однако константам могут присваиваться значения и в явном виде с помощью операции присваивания. Например:

```
enum selector
{
  vyhod,  // vyhod=0,
  sozd,   // sozd=1
  prosm,  // prosm=2
  ud,     // ud=3
  zam     // zam=4
} f;      // создана переменная f типа selector
enum dni
{
  pn,       // pn=0
  vt,       // vt=1
  sr = 5,   // sr=5
  cht,      // cht=6
  pt = 12,  // pt=12
  sb,       // sb=13
  vs        // vs=14
};  // переменная типа dni в момент объявления
    // перечисления не создается
enum dni d; // объявление переменной d типа dni
```

Над переменной перечисляемого типа возможны операции:

• *операция присваивания*, например

```
sel1 = sozd; d = vs;
```

• *операции отношения*, например

```
if (sel1 == sozd) ...;
```

• любые операции, выполняемые над типом **int**.

Использование перечислений дает возможность применять вместо числовых значений смысловые идентификаторы, в результате чего облегчается программирование и повышается читабельность программы.

Структура — это поименованная совокупность данных, состоящая из фиксированного числа компонентов разных типов. Компоненты структуры называются полями.

Синтаксис объявления структуры:

```
struct <имя_структуры>
{
  <тип><элемент1>;
  <тип><элемент2>;
     ...
}<имя_создаваемой_переменной>;
```

Например:

```
struct tip_rab  // объявление типа структура tip_rab
{
int tabn;        // элементы структуры
  char fio[20];
  float zarp;
}rab1;      // создается переменная rab1 типа
            // struct tip_rab
```

или:

```
struct tip_rab rab2, brigada[10]; // объявление
// переменных типа структура tip_rab
```

Для доступа к элементам структуры используются операторы:

- *прямого* выбора поля структуры («.»);
- *косвенного* выбора поля структуры («->»).

Операция «.» обеспечивает доступ к элементу структуры по имени переменной, реализующей структуру, например

```
rab1.tabn=120; // элементу tabn структуры tip_rab
               // присвоено значение 120
```

Операция «->» используется для доступа к элементу структуры при заданном указателе на структуру, например

```
// объявление указателя на структуру tip_rab
struct tip_rab * p1;
// получение значения элемента tabn структуры tip_rab
i = p->item1;
```

То же самое действие можно выполнить с использованием операции «.» и операции косвенной адресации «*»:

```
i = (*p).tabn;
```

Определение адреса поля структуры выполняется с помощью операции адресации: **&prab -> tabn**. Здесь операция **->** имеет более высокий приоритет, чем операция &.

Над целыми структурами, имеющими один и тот же идентификатор типа, возможна операция присваивания, например:

```
rab1 = rab2;
```

Тип «структура» является основным типом данных в экономических и управленческих задачах.

Объединение — это поименованная совокупность данных, состоящая из фиксированного числа компонентов разных типов, но активным может быть только один компонент. Многие синтаксические и функциональные свойства объединений совпадают со структурами. Например:

```
union tip_rab    // объявление типа объединение
{
  int tabn;      // размер 2 байта
  char fio[20]; // размер 20 байт
  float zarp;    // размер 4 байта
};
union tip_rab rab1,rab2;  // объявление переменных
                          // типа tip_rab
```

Размер объединения равен размеру максимального компонента объединения, в приведенном примере он равен 20 байтам. Переменная **rab1** может одновременно хранить в 20 байтах либо значение **int**, либо массив **char**, либо значение **float**. Операция **sizeof (rab1)** возвращает значение 20, но в том случае, когда **rab1** содержит объект типа **int**, 18 байтов остаются неиспользованными (туда помещаются символы-заполнители), аналогично, когда **rab1** содержит объект типа **float**, неиспользованными остаются 16 байтов.

Доступ к полям объединения выполняется так же, как и к полям структуры. Объединение может быть инициализировано только одним значением.

Пример 5.11.1

Ввести сведения о 15 студентах, включающие код студента, номер группы, код дисциплины и полученную оценку. Рассчитать средний балл, полученный студентами указанной группы по указанной дисциплине.

```
#include <stdio.h>
int main()
{
  struct student {  // структура student включает поля:
    int num;        // код студента
    int grup;       // номер группы
    int exam;       // код дисциплины
    int ball;       // оценка
  };
  int gr, ex;
  int kol = 0;      // kol - количество студентов
  float sr = 0;     // sr - средний балл
  struct student st[15];
  int i;
  for (i = 0; i< 15; i++) // ввод сведений о студентах
  {
    printf("vvedite nomer");
    scanf("%d", &st[i].num);
```

```
      printf("vvedite gruppu");
      scanf("%d", &st[i].grup);
      printf("vvedite examen");
      scanf("%d", &st[i].exam);
      printf("vvedite ocenku");
      scanf("%d", &st[i].ball);
    }
    printf("vvedite kod examena"); // ввод кода дисциплины
    scanf("%d", &ex);
    printf("vvedite nomer gruppy"); // ввод номера группы
    scanf("%d", &gr);
    for (i = 0; i < 10; i++)
    if ((st[i].grup == gr) && (st[i].exam == ex))
    {
      sr += st[i].ball;  // вычисление суммы баллов
      kol++;             // и количества студентов
    }
    if (kol != 0) // если количество студентов не равно 0
      printf("%f", sr / kol);  // вывод среднего балла
    else
      printf("net dannyh"); // вывод сообщения
                            // об отсутствии данных
    return 0;
  }
```

5.12. Расширения языка C++

В данном параграфе рассматриваются некоторые базовые отличия языка C++ от языка C.

В языке C++ появилась возможность использования ссылок. *Ссылка* — это псевдоним (альтернативное имя) для объекта. Использование ссылок представляет собой альтернативу использованию указателей.

Синтаксис описания ссылки:

```
<тип>&<имя_ссылки> = <значение>;
```

Например:

```
int tabn = 1000;
int &r = tabn;
```

Теперь **r** является псевдонимом **tabn**.

При объявлении ссылки обязательно должна быть выполнена ее инициализация, причем инициализирована ссылка может быть только один раз. После инициализации значении ссылки изменить нельзя, ее нельзя связать с другим объектом.

Ссылки часто используют для передачи параметров функции. Передача параметров по ссылке аналогична передаче параметров по указателю. Однако использование ссылок более удобно, так как

ссылка пишется как обычная переменная и не требует операций адресации (&) и разадресации (*).

В языке C++ появились новые средства динамического распределения памяти. К ним относятся операторы **new** и **delete**.

Оператор *new* выделяет память требуемого объема в соответствии с переданным типом данных и возвращает указатель на выделенную память. В качестве типа может выступать любой базовый либо пользовательский тип. Оператор **new** можно использовать для выделения памяти под массивы.

Оператор *delete* освобождает выделенную память. Специальная форма оператора **delete[]** для массивов объектов выполняет не только освобождение всей выделенной памяти (аналогично **delete**), но и уничтожение каждого объекта в массиве.

В языке C++ появилось понятие класса.

Класс — это совокупность данных различных типов и функций для их обработки с атрибутами доступа к ним. Классы являются дальнейшим развитием структур и позволяют не только хранить данные в структурированном виде, но и разместить вместе с ними функции, оперирующие с этими данными. Это позволяет разграничить доступ к данным и более понятно описывать методы работы с данными. Функции, включенные в состав класса, называются *методами*.

Элементами класса служат элементы-данные и элементы-функции.

Элементы-данные — это совокупность взаимосвязанных данных различных типов, объявленная в определении класса.

Элементы-функции — это функции, объявленные в определении класса и обрабатывающие элементы-данные класса.

Доступ к переменным и методам класса определяется их областью видимости. Область видимости принадлежащих классу данных и методов задается с помощью ключевых слов:

- private (собственный);
- protected (защищенный);
- public (общедоступный).

Собственные (*private*) данные и методы доступны только внутри данного класса.

Защищенные (*protected*) данные и методы доступны внутри данного класса и для классов-потомков.

Общедоступные (*public*) данные и методы доступны в любом месте программы.

Существуют специальные элементы-функции класса (конструкторы и деструкторы), служащие для создания, копирования, преобразования и уничтожения объектов класса.

Конструктор — функция, автоматически вызываемая при создании объекта данного класса. Обычно конструкторы используют для инициализации переменных объекта, выполнения каких-либо первоначальных действий и т.д. Конструктор не имеет возвращаемого типа. Имя конструктора должно совпадать с именем класса. Класс может иметь несколько конструкторов, отличающихся набором аргументов.

Деструктор — это функция, автоматически вызываемая при уничтожении объекта. Обычно деструктор используется для выполнения каких-либо финальных действий и чистки памяти. В классе может быть только один деструктор. Деструктор не имеет возвращаемого значения и должен иметь имя вида **~имя_класса**.

В данном параграфе рассмотрена лишь очень малая часть особенностей языка C++. Для его подробного изучения следует воспользоваться специальной литературой.

Глава 6
РАЗРАБОТКА ПРОГРАММНОГО ПРИЛОЖЕНИЯ НА ЯЗЫКЕ С

Требуется разработать программное приложение «Формирование заказов на использование строительных механизмов».

Приложение должно обеспечить выполнение следующих действий.

1. Создание входных файлов; требуется создать файлы:

• *Справочник строительных механизмов*, содержащий поля: Код механизма, Наименование, Стоимость одного машиночаса работы;

• *Заказчики*, содержащий поля: Код заказчика, Наименование организации, Форма оплаты.

2. Получение выходного файла *Заказы* на основе входных файлов и данных, введенных с клавиатуры. С клавиатуры вводятся Номер заказа, Дата заказа, Код заказчика, Код механизма и Объем работы в часах. Выходной файл *Заказы* должен содержать поля: Номер заказа, Дата заказа, Код заказчика, Наименование организации заказчика, Код механизма, Наименование механизма, Объем работы в часах.

3. Вывод Стоимости заказа по введенному с клавиатуры Номеру заказа.

4. Создание меню для выполнения всех указанных в задании пунктов.

Создание приложения осуществляется с помощью следующего программного кода:

```
#include <stdio.h>
#include <string.h>
// объявляем структуры
structc_materials
{
  int mtr_code;
  char mtr_name[20];
  int cash;
};
```

```
struct customer
{
  int customer_code;
  char organization_name[20];
  char form_payment[20];
};

struct order
{
  int orders;
  char order_date[15];
  int customer_code;
  char organization_name[20];
  int mtr_code;
  char mtr_name[20];
  int time;
};

int main()
{
  int operation = 0, quantity, pos = 0;
  char selection;
  int mark_code, summ, ord;
  // Объявляем указатели на файлы
  FILE *MN = NULL, *AB = NULL, *IR = NULL;
  printf("  ------------------------------------\n");
  printf("  |          Good day, user!         |\n");
  printf("  |      Please, select action:      |\n");
  printf("  |       1) Add customer            |\n");
  printf("  |       2) Add order               |\n");
  printf("  |       3) Search by brand         |\n");
  printf("  |       4) Exit from the program   |\n");
  printf("  |    (Use the number operation!!   |\n");
  printf("  ------------------------------------\n\n\n");

  // выделяем память под структуры
  struct customer cus;
  struct order od;
  struct c_materials cm;

  // запускаем цикл проверки введенного значения
  while (operation != 4)
  {
    printf("\nOperation: ");
    scanf("%d", &operation);
    // запускаем выбор действия по введенному значению
    switch (operation)
    {
    case 1:
      {
        printf("Enter the number of customers: ");
        scanf("%d", &quantity);
        for (int i = 0; i < quantity; i++)
```

```
    {
      printf("\nEnter the customer code: ");
      scanf("%d", &cus.customer_code);
      printf("\nEnter the organization name: ");
      scanf("%s", &cus.organization_name);
      printf("\nEnter the form of payment: ");
      scanf("%s", &cus.form_payment);
      IR = fopen("customer.txt", "a");
      fprintf(IR, "%d %s %s\n", cus.customer_code,
        cus.organization_name, cus.form_payment);
      fclose(IR);
    }
    break;
  }

case 2:
  {
    // вводим данные с клавиатуры для формирования
    // заказа
    printf("\nEnter the quantity of order: ");
    scanf("%d", &quantity);
    for (int i = 0; i < quantity; i++)
    {
      printf("\nEnter the order number: ");
      scanf("%d", &od.orders);
      printf("\nEnter the order date: ");
      scanf("%s", &od.order_date);
      printf("\nEnter the customer code: ");
      scanf("%d", &od.customer_code);
      printf("\nEnter the materials code: ");
      scanf("%d", &od.mtr_code);
      printf("\nEnter the time in hours: ");
      scanf("%d", &od.time);
      MN = fopen("order.txt", "a");
      IR = fopen("customer.txt", "r");
      AB = fopen("c_materials.txt", "r");
      // цикл поиска нужного значения
      while (!feof(IR))
      {
        fscanf(IR, "%d%s%s", &cus.customer_code,
          &cus.organization_name, &cus.form_payment);
        if (od.customer_code == cus.customer_code)
          strcpy(od.organization_name,
          cus.organization_name);
      }
      // цикл поиска нужного значения
      while (!feof(AB))
      {
        fscanf(AB, "%d %s %d", &cm.mtr_code,
        &cm.mtr_name, &cm.cash);
        printf("\n%s", cm.mtr_name);
```

```
        if (od.mtr_code == cm.mtr_code)
          strcpy(od.mtr_name, cm.mtr_name);
      }
      fclose(IR);
      fclose(AB);
      fprintf(MN, "%d %s %d %s %d %s %d\n",
        od.orders, od.order_date, od.customer_code,
        od.organization_name, od.mtr_code,
        od.mtr_name, od.time); fclose(MN);
    }
    printf("\nDone!");
    break;
  }

case 3:
  { // вывод стоимости перевозок по номеру заказа
    printf("\nEnter the number of order: ");
    scanf("%d", &ord);
    MN = fopen("order.txt", "r");
    AB = fopen("c_materials.txt", "r");
    while (!feof(MN))
  {
      // поиск заказа
      fscanf(MN, "%d %s %d %s %d %s %d\n",
        od.orders, od.order_date, od.customer_code,
        od.organization_name, od.mtr_code,
        od.mtr_name, od.time);

      if (ord == od.orders)
      {
        while (!feof(AB))
        {// поиск стоимости перевозки по машине
          fscanf(AB, "%d%s%d", &cm.mtr_code,
            &cm.mtr_name, &cm.cash);
            if (od.mtr_code == cm.mtr_code)
            {
              summ = cm.cash*od.time;
            }
          }
        }
      }
    printf("\nCost by order number: %d", summ);
    fclose(AB);
    fclose(MN);
    break;
  }

case 4:
  break;
```

```
    default:
      printf("Use number 1-4!!!!\n\n");
      break;
  }

  if (operation >= 1 && operation < 4)
  {
    printf("\nSelect another operation? (y|||n):");
    scanf("%c", &selection);
    scanf("%c", &selection);
    if (selection == 'n')
      operation = 4;
  }
}

printf("\n       Good bye! Have a nice day!");
return 0;
}
```

Глава 7
ИНТЕГРАЦИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ PYTHON И C

Знание языка программирования C позволяет создавать новые модули Python, расширяющие возможности языка. Для создания такого модуля воспользуемся пакетом distutils[1], входящим в состав Python.

Рассмотрим пример создания модуля Python (для версии 3 и выше) на языке C в операционной системе Debian 4.9.2-10[2]. Для этого в первую очередь понадобится создать файл на языке C (*ownmod.c*), содержащий функциональные[3] возможности будущего модуля:

```
/* ownmod.c */

#include <Python.h> /* Определяет набор функций,
макросов и переменных, которые обеспечивают доступ
к большинству возможностей интерпретатора Python */

/*
Функция на языке C, которая будет вызываться
при обращении к функции echo из языка Python.
*/
static PyObject* py_echo( PyObject* self, PyObject*
args )
{
  printf( "вывод из экспортированного кода!\n" );
  return Py_None;
}

/*
Массив структур типа PyMethodDef, содержащий имя
функции "echo"; адрес функции py_echo, которая будет
вызываться; флаг METH_NOARGS, указывающий на отсутствие
```

[1] Подробнее см.: https://docs.python.org/3.6/library/distutils.html.

[2] Сборка в Windows: https://docs.python.org/3.6/extending/windows.html.

[3] Подробнее см.: https://docs.python.org/3.6/extending/index.html; https://docs.python.org/3.6/c-api/index.html.

```
аргументов; строка документации - "echo function".
Массив должен оканчиваться нулем.
*/
static PyMethodDefownmod_methods[] =
{
    { "echo", py_echo, METH_NOARGS, "echo function" },
    { NULL, NULL }
};

/*
Определяем структуру, необходимую для создания
модульного объекта.
*/
static struct PyModuleDef ownmodule =
{
  PyModuleDef_HEAD_INIT,
  "ownmod", // имя модуля
  NULL,     // документация, может содержать NULL
  -1,       // модуль хранит состояние в глобальной
            // переменной
  ownmod_methods
};

/*
Функция для инициализации модуля.
*/
PyMODINIT_FUNC PyInit_ownmod()
{
  PyObject *m;
  /* Создание нового модульного объекта */
  m = PyModule_Create(&ownmodule);
  if (m == NULL)
    return NULL;
}
```

Затем формируем файл с именем *setup.py*, который необходим для сборки и установки модуля:

```
# setup.py
from distutils.core import setup, Extension
module1 = Extension( 'ownmod', sources = ['ownmod.c'] )
setup( name = 'ownmod',
       version = '1.1',
       description = 'This is a first package',
       ext_modules = [module1]
     )
```

Для сборки модуля[1] (создания динамической библиотеки .so) в командной строке Linux необходимо выполнить:

```
python3 setup.py build
```

[1] Подробнее см.: https://docs.python.org/3/install/.

Для установки модуля (копирования созданной библиотеки в рабочие каталоги) в командной строке Linux с правами администратора необходимо выполнить

```
python3 setup.py install
```

Теперь можем запустить интерпретатор Python, импортировать созданный модуль **ownmod** и вызвать функцию **echo**:

```
>>> import ownmod
>>> ownmod.echo()
вывод из экспортированного кода!
>>>
```

Контрольные вопросы

1. Какова структура программы на языке С?
2. Что такое модуль?
3. Что такое функция?
4. Какие символы входят в алфавит языка С?
5. Какие типы констант существуют в языке С?
6. Что такое переменная?
7. Что определяет тип данных?
8. Какие стандартные типы переменных есть в языке С?
9. Когда происходит выделение памяти под переменные?
10. Какая операция используется для явного преобразования типа данных?
11. Что понимается под временем жизни переменной?
12. Что понимается под областью видимости переменной?
13. Что определяет класс памяти?
14. В чем особенности статических переменных?
15. В чем отличия префиксной и постфиксной форм операций?
16. Какая операция с тремя операндами существует в языке С?
17. Какие типы алгоритмических структур применяются в структурном программировании?
18. Какие операторы используются для реализации ветвящихся алгоритмов?
19. Какие виды циклов существуют?
20. В чем отличие циклов с предусловием от циклов с постусловием?
21. Для чего предназначен оператор break?
22. Для чего предназначен оператор continue?
23. Что такое директива препроцессора?
24. Для чего предназначена директива #define?
25. Какие директивы условной компиляции вы знаете?
26. Что такое указатель?
27. Какие операции применяются для работы с указателями?
28. Что такое массив?
29. Что означает операция индексации для массива?
30. В чем особенности хранения многомерных массивов?

31. Как производится инициализация массивов?
32. Что такое динамический массив?
33. Какие функции применяются для управления динамическим выделением памяти?
34. Каковы особенности динамического выделения памяти для многомерных массивов?
35. Что такое прототип функции?
36. Что такое определение функции?
37. Чем различаются формальные и фактические параметры функции?
38. Какие существуют способы передачи параметров функции?
39. Что такое рекурсивная функция?
40. Какие функции форматированного ввода/вывода вы знаете?
41. Что такое спецификатор формата?
42. Какие функции неформатированного ввода/вывода вы знаете?
43. Каковы особенности хранения строковых данных?
44. Что такое поток ввода/вывода?
45. Какие стадии включает процесс работы с файлами?
46. Что такое курсор?
47. Какие пользовательские типы данных вы знаете?
48. Что такое перечисление?
49. Что такое объединение?
50. Что такое структура?

Задания для самостоятельного выполнения

1. С клавиатуры ввести три числа. Найти произведение наибольшего и наименьшего среди них.

2. С клавиатуры ввести два целых числа x и y. Вычислить значение $f(x, y)$, равное

$$f(x, y) = \begin{cases} 2x + y, \text{ если } |x| > |y|, \\ x^2, \text{ если } x = y, \\ x - y \text{ в остальных случаях.} \end{cases}$$

3. Ввести координаты (x и y) трех точек на плоскости. Определить, могут ли эти точки являться вершинами треугольника.

4. С клавиатуры ввести натуральное число A. Найти наибольшее простое число, меньшее A.

5. Ввести два натуральных числа n и m — числитель и знаменатель дроби. Сократить дробь и вывести числитель и знаменатель сокращенной дроби.

6. Ввести два натуральных числа. Найти их наименьшее общее кратное.

7. Ввести натуральное число. Поменять в этом числе местами максимальную и минимальную цифру. Вывести полученное число.

8. С клавиатуры ввести n целых чисел. Вывести число с наибольшим количеством целочисленных делителей. Если таких чисел несколько, вывести их количество.

9. С клавиатуры ввести n целых чисел. Вывести число с максимальной суммой цифр. Если таких чисел несколько, вывести их количество.

10. С клавиатуры вводится величина относительной погрешности ξ. С точностью ξ найти сумму ряда, общий член которого задан формулой:

$$a_n = \frac{10^n}{n!}.$$

11. Ввести целое число. Найти максимальную и минимальную цифру в этом числе. Для поиска цифр использовать функцию, параметры которой передаются по указателям.

12. Ввести n целых чисел. Найти наибольше четное и наименьшее нечетное число среди введенных.

13. Ввести число целое число n. Вычислить n^n и $n!$ Для вычислений использовать функцию, параметры которой передаются по указателям.

14. Ввести одномерный массив, состоящий из n элементов. Вывести элемент, наименее отличающийся от среднего арифметического значения массива.

15. Ввести одномерный массив, состоящий из n элементов. Поменять в этом массиве местами первый положительный и последний отрицательный элементы.

16. Ввести одномерный массив, состоящий из n элементов. Найти максимальную сумму подряд идущих одинаковых элементов массива.

17. Ввести одномерный массив, состоящий из n элементов. Найти максимальный элемент, значение которого меньше его индекса.

18. Ввести одномерный массив, состоящий из n элементов. Сжать массив, удалив из него все отрицательные элементы. Дополнительный массив не использовать.

19. Ввести одномерный массив, состоящий из n элементов. Добавить в начало массива минимальный элемент, а в конец — максимальный. Дополнительный массив не использовать.

20. Ввести одномерный массив, состоящий из n элементов, где n — четное число. Вставить в середину массива среднее арифметическое значение. Дополнительный массив не использовать.

21. Ввести двумерный массив размером n на m. Найти среднее арифметическое значение. Все элементы массива, большие среднего арифметического, заменить на 1, остальные — на 0. Вывести полученный массив.

22. Ввести целочисленный двумерный массив размером n на m. Подсчитать количество четных и нечетных элементов в массиве. Если четных больше, найти максимальное значение в массиве, в противном случае — минимальное. Найденное значение вывести.

23. Ввести двумерный массив размером n на m. В каждом столбце массива найти максимальный элемент. Среди найденных максимальных элементов определить и вывести минимальный.

24. Ввести двумерный массив размером n на m. Ввести целое число k, поменять в массиве местами элементы 1-й и k-й строки. Полученный массив вывести.

25. Ввести массив размером n на m. В каждой строке массива поменять местами максимальный и минимальный элементы. Полученный массив вывести.

26. Ввести массив размером n на m. В каждой строке массива найти максимальный элемент и заменить его средним арифметическим значением того столбца, в котором находится этот элемент.

27. Ввести массив размером n на m. В каждом столбце массива заменить минимальный элемент средним арифметическим значением данного столбца.

28. Ввести двумерный массив размером n на m. В каждой строке массива найти произведение отрицательных элементов. Для нахождения произведения написать соответствующую функцию. Найденные значения вывести.

29. Ввести одномерный массив, состоящий из n элементов. Написать рекурсивную функцию, вычисляющую сумму элементов этого массива.

30. Ввести целое число. Найти максимальную и минимальную цифру. Для нахождения цифр использовать функцию с передачей параметров по указателям.

31. Ввести натуральное число n. Вычислить $n \cdot n$ и $n!$. Для вычисления значений написать функцию с передачей параметров по указателям.

32. Ввести массив размером n на m. В каждом столбце массива поменять местами максимальный и минимальный элементы. Для замены переменных написать соответствующую функцию. Полученный массив вывести.

33. Ввести предложение, в котором слова разделены пробелами. Ввести одну букву. Вывести все слова, в которых отсутствует введенная буква.

34. Ввести предложение на русском языке, в котором слова разделены пробелами. Вывести слово, содержащее наибольшее количество гласных букв. Если таких слов несколько, вывести первое из них.

35. Ввести предложение, в котором слова разделены пробелами. Инвертировать в этой строке все слова с нечетными номерами.

36. Ввести предложение, в котором слова разделены пробелами. Вставить после каждого слова число, равное длине слова.

37. Ввести предложение, в котором слова разделены пробелами. Подсчитать, сколько в этом предложениии слов-палиндромов, одинаково читающихся слева направо и справа налево.

38. Создать текстовый файл, содержащий произвольные числа. Найти произведение положительных чисел и сумму отрицательых. Вычисленные значения записать в другой файл.

39. Создать текстовый файл, содержащий произвольные символьные значения. Определить, сколько в этом файле цифровых символов. Результат записать в другой текстовый файл.

40. Создать текстовый файл, содержащий произвольные числа. Определить максимальное значение из элементов с четными номерами и минимальное — из элементов с нечетными номерами. Полученные значения записать в другой текстовый файл.

41. Сформировать массив, содержащий сведения о сотрудниках. Структурный тип содержит поля: фамилия сотрудника, наименование от-

дела, стаж работы, должность, оклад. Вывести информацию о среднем стаже сотрудников отдела (номер отдела вводится с клавиатуры).

42. Сформировать массив, содержащий сведения о погоде за месяц. Структурный тип содержит поля: число, температура дневная, температура ночная, количество осадков. Определить день с максимальной дневной температурой, с минимальной ночной температурой, среднее количество осадков за месяц.

43. Сформировать массив, содержащий сведения о сдаче студентами сессии. Структурный тип содержит поля: номер группы, фамилия студента, оценки по четырем предметам. Вывести фамилии всех неуспевающих студентов с указанием номера группы и количества задолженностей.

44. Сформировать массив, содержащий сведения о сдаче студентами экзамена. Структурный тип содержит поля: номер группы, фамилия студента, оценка. Вывести средний балл по каждой группе, а также номер группы, в которой получено больше всего неудовлетворительных оценок.

Литература

1. *Столяров, А. В.* Программирование: введение в профессию. Т. 1: Азы программирования / А. В. Столяров. — М. : МАКС Пресс, 2016.

2. *Столяров, А. В.* Программирование: введение в профессию. Т. 2: Низкоуровневое программирование / А. В. Столяров. — М. : МАКС Пресс, 2016.

3. *Керниган, Б. В.* Язык программирования Си : пер. с англ. / Б. В. Керниган, Д. М. Ритчи — 3-е изд. — СПб. : Невский Диалект, 2001.

4. *Подбельский, В. В.* Программирование на языке Си / В. В. Подбельский — М. : Финансы и статистика, 2004.

5. *Березин, Б. И.* Начальный курс С и С++ : учеб. пособие / Б. И. Березин, С. Б. Березин. — М. : ДИАЛОГ-МИФИ, 2004.

6. *Культин, Н. Б.* С/С++ в задачах и примерах : сб. задач / Н. Б. Культин. — СПб. : БХВ-Петербург, 2004.

7. *Лучано, Р.* Python. К вершинам мастерства / Р. Лучано. — М. : ДМК Пресс, 2016.

8. *Лутц, М.* Изучаем Python : пер. с англ. / М. Лутц. — 4-е изд.— СПб. : Символ-Плюс, 2011.

9. *Лутц, М.* Программирование на Python, Т. I : пер. с англ. / М. Лутц. — 4-е изд. — СПб. : Символ-Плюс, 2011.

10. *Лутц, М.* Программирование на Python, Т. II : пер. с англ. / М. Лутц. — 4-е изд. — СПб. : Символ-Плюс, 2011.

11. *Лаврищева, Е. М.* Технология программирования и программная инженерия : учебник для вузов / Е. М. Лаврищева. — М. : Издательство Юрайт, 2017.

12. *Лаврищева, Е. М.* Технология разработки и моделирования вариантов программных систем : монография для вузов / Е. М. Лаврищева. — М. : Издательство Юрайт, 2017.