Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

О. Н. ПАВЛОВА

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Владимир 2019

УДК 004.42
ББК 32.973
П12

Рецензенты:
Кандидат технических наук
генеральный директор ООО «ФС Сервис»
Д. С. Квасов

Кандидат физико-математических наук
доцент кафедры функционального анализа и его приложений
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
И. А. Петренко

Издается по решению редакционно-издательского совета ВлГУ

Павлова, О. Н.

П12 Основы программирования : учеб. пособие / О. Н. Павлова ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2019. – 108 с. – ISBN 978-5-9984-1070-3.

Формулирует основные принципы построения алгоритмов. Подробно излагает синтаксис и семантику языка высокоуровневого программирования C++.

Предназначено для бакалавров, обучающихся по направлениям 28.03.01 «Нанотехнологии и микросистемная техника», 12.03.05 «Лазерная техника и лазерные технологии».

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 14. Табл. 5. Библиогр.: 5 назв.

УДК 004.42
ББК 32.973

ISBN 978-5-9984-1070-3

ПРЕДИСЛОВИЕ

Компьютеризация – это неотъемлемая часть практически всех сфер деятельности современного человека. Компьютеры используются при обучении, работе и даже отдыхе.

Применение компьютеров значительно облегчает процесс получения любой информации, обмена ею с другими людьми. Благодаря компьютерным технологиям вычислительные процессы, на которые раньше тратилась масса времени, стали значительно быстрее и точнее, потому что компьютер в отличие от человека ошибается гораздо реже, практически никогда.

В своей профессиональной деятельности выпускники направлений подготовки бакалавров 28.03.01 «Нанотехнологии и микросистемная техника», 12.03.05 «Лазерная техника и лазерные технологии» занимаются не только непосредственной работой с приборами, но и компьютерным моделированием задач лазерной техники и нанотехнологий, проведением экспериментов, измерениями и обработкой полученных результатов. Для автоматизации данных процессов можно применять ЭВМ, но для этого потребуется разработка компьютерной программы на языке программирования, а следовательно, и знания в области программирования.

В различных университетах и колледжах существует большое количество вариантов учебных программ. При разработке учебно-методического комплекса дисциплины «Основы программирования», в состав которого входит данное пособие, учтены прежде всего федеральные государственные образовательные стандарты подготовки бакалавров по указанным направлениям. Стоит отметить, что издание соответствует рабочим программам по данной дисциплине для обоих направлений подготовки бакалавров.

ВВЕДЕНИЕ

На современном этапе информатизации общества актуальна подготовка квалифицированных кадров, умеющих решать профессиональные задачи, в том числе и с применением компьютерной техники, для проведения экспериментов и обработки их результатов. Изучение основ программирования даже для специалистов в областях лазерной техники и лазерных технологий, а также нанотехнологий – неотъемлемая часть подготовки квалифицированных кадров.

В качестве изучаемого языка программирования выбран C++. Это не случайно, так как C++ поддерживает такие парадигмы программирования, как процедурное, объектно-ориентированное, обобщённое программирование, обеспечивает модульность, раздельную компиляцию, обработку исключений, абстракцию данных, объявление типов (классов) объектов, виртуальные функции. Стандартная библиотека включает в том числе общеупотребительные контейнеры и алгоритмы. C++ сочетает свойства как высокоуровневых, так и низкоуровневых языков.

В ВлГУ согласно учебным планам подготовки бакалавров по направлениям 28.03.01 «Нанотехнологии и микросистемная техника», 12.03.05 «Лазерная техника и лазерные технологии» для освоения дисциплины «Основы программирования» отведено 36 часов лекций, 18 часов практических и 36 часов лабораторных занятий с обязательной самостоятельной работой.

Пособие предназначено для проведения лекционных пар. Изучение одной лекции рассчитано на период от двух до четырех часов в зависимости от объема изложенного материала. Рекомендации по времени изучения: темы 1, 6, 8, 9 должны занимать около двух часов аудиторного изучения, остальные – четыре часа с обязательной самостоятельной проработкой тем по материалам пособия и дополнительной литературы.

В конце каждой темы приведены вопросы и задания, предназначенные для проверки качества усвоения материала. Вопросы могут быть вынесены также в качестве рейтинговых заданий и дополнительных вопросов при сдаче экзамена.

Тема 1. ПОНЯТИЕ И ОСНОВНЫЕ СВОЙСТВА АЛГОРИТМА

1.1. Понятие алгоритма

Слово *алгоритм* произошло от имени восточного математика Хорезми (по-арабски Ал-Хорезми), который в IX веке разработал правила выполнения четырех арифметических действий над числами в десятичной системе счисления. В Европе совокупность этих правил получила название «алхоризм», затем, ближе к европейскому, – «алгоризм». Впоследствии произошло переименование термина в «алгоритм» как собирательное название правил определенного вида (не только арифметических действий) для выполнения какой-либо работы. Очень долгое время его употребляли только математики, обозначая конкретную последовательность некоторых правил для решения различных вычислительных задач.

В 30-х годах XX века это понятие стало объектом математического изучения, а с появлением ЭВМ получило известность и широкое распространение. Одним из замечательных достижений науки XX века стало создание и развитие теории алгоритмов, новой математической дисциплины, предметом изучения которой и стали *алгоритмы.* Возникновение ЭВМ и программирования обнаружило тот факт, что разработка алгоритма – необходимый этап автоматизации вычислений (что сегодня записано в виде алгоритма – завтра будет выполняться роботами). Развитие вычислительной техники и методов программирования нуждается в теории алгоритмов, но и порождает, в свою очередь, новые идеи для теории алгоритмов.

Сейчас алгоритмизацию применяют в различных областях человеческой деятельности, понимая это как создание руководства для достижения результата.

Алгоритм – это некоторая конечная последовательность точных элементарных предписаний (команд, инструкций, правил), однозначно определяющих процесс преобразования исходных данных и промежуточных результатов в результат решения задачи (иначе: это план к решению задачи). Свойства алгоритма:

1) **дискретность** (конечный набор отдельных шагов);

2) **определенность** (восприятие и исполнение нужных пунктов вполне однозначно);

3) **результативность** (всегда найдется путь от начала к концу решения, может быть, и с отсутствием конкретных выходных результатов);

4) **массовость** (алгоритм работает для конкретного типа задачи при различных наборах исходных данных, которые могут меняться в определенных пределах);

5) **понятность** (создается в расчете на определенного исполнителя, способного выполнить каждый шаг предписания).

В качестве исполнителя алгоритма может выступать не только человек, но и техническое устройство: станок, робот, ЭВМ, телефон, бытовой прибор и т. д. Для правильного построения алгоритма необходимо обязательно учитывать конкретного исполнителя, его возможности, знания и пр. Для исполнителя ЭВМ – это набор допустимых операций, скорость выполнения отдельных операций, возможность замены через другие операции, разрешенный объем данных, программы и т. д. Только учитывая свойства и возможности ЭВМ, можно получить эффективные алгоритмы.

1.2. Этапы подготовки вычислительных задач для их автоматического решения

На начальной стадии обучения бывает довольно трудно для решения задачи писать сразу программу. Трудности увеличиваются с ростом сложности задачи, тем более что писать надо только набором допустимых предписаний. Однако напомним, что всякий сложный процесс базируется на последовательности более простых. В случае автоматизации решения задач эта цепочка давно сложилась и известна как последовательность этапов подготовки и решения задачи с использованием компьютера. В последовательности этапов и в их содержании заключается то важное, что позволит решать не только учебные вычислительные задачи, но и большие сложные практические, с которыми сталкиваются специалисты в любой сфере профессиональной деятельности. Необходимо лишь понять и усвоить ее.

В литературных источниках нужные действия могут группироваться или, напротив, дробиться по-разному, образуя различное количество этапов. Последовательность этапов решения задачи на компьютере:

1. Постановка задачи – текстовая формулировка в терминах конкретной предметной области, включающая полный перечень исходных данных и требования к результатам (выполняется обычно заказчиком).

2. Тщательный анализ задачи с целью правильного понимания и осмысления всех ее объектов (параметров), включая исходные, промежуточные и выходные. Для сложных задач может выполняться ее структурирование, пошаговая детализация. В технологиях программирования это называют нисходящим проектированием, или проектированием сверху вниз (от сложной задачи к системе более простых). Такая детализация с переходом к подзадачам все более низкого уровня может выполняться многократно, приближаясь к системе предписаний, пригодной для непосредственного исполнителя решения задачи.

3. Формальное описание (математическая модель) задачи или каждой подзадачи в случае ее разбиения. Все физические параметры заменяют условными математическими обозначениями (по возможности передающими смысл параметра), составляют таблицу идентификаторов, содержащую более полную информацию об этих объектах с указанием всех требований по форматам данных и результатам. Затем предполагается связь входных параметров с необходимыми математическими соотношениями, формулами, обеспечивающими получение результатов. Удачный выбор математической символики (т. е. списка идентификаторов), четкое представление математических разделов, на которых базируется изучаемый вопрос, обеспечат понятность и простоту математической модели, что очень важно для успеха всей последующей работы.

4. Выбор специального метода, дающего возможность получить результаты. Этот этап выполняется только в том случае, когда вычисления выходных параметров по математической модели неочевидны.

5. Составление алгоритма по выбранному численному методу, если п. 4 не выполнялся. На этом этапе, когда алгоритм только разрабатывается, его составляют в текстовой или графической форме – по усмотрению разработчика. С точки зрения решаемой задачи выбор формы представления алгоритма значения не имеет. Если выполнялось разбиение задачи на подзадачи, то алгоритмы разрабатываются для каждой подзадачи, а также для управления их совместной согласованной работой в соответствии с общей методикой получения результата (проектирование снизу вверх).

6. Написание программы по составленному алгоритму на требуемом языке программирования. По сути, эта работа сводится к переводу каждого пункта алгоритма с одного языка представления на другой. Иногда, конечно, целесообразно использовать более сложные конструкции

языка программирования, покрывающие работу сразу нескольких элементарных предписаний алгоритма.

7. Ввод программы в оперативную память, трансляция, редактирование синтаксических ошибок, получение контрольных результатов (используется среда системы программирования). Заранее подбирают контрольную точку по входным данным так, чтобы легко можно было вычислить результаты «вручную», используя лишь п. 1. Возможно, такие данные не будут иметь физического смысла для задачи, но зато заранее и просто вычисленный по ним результат может использоваться как тест, критерий семантической правильности работы программы. Контрольных точек может быть и несколько.

8. Если результаты контрольного решения не совпадают с «ручными», необходима отладка программы. При этом следует учесть, что расхождение результатов возможно не только вследствие семантических (смысловых) ошибок по тексту программы, но и ввиду ошибок, допущенных на любом этапе подготовки задачи: ошибки в составлении математической модели (см. пп. 2, 3), ошибки при реализации метода (см. п. 4), ошибки в алгоритме (см. п. 5), ошибки при переводе на алгоритмический язык (см. п. 6). Процесс отладки – довольно серьезный этап, требующий кропотливой, вдумчивой проверки всей уже проделанной работы.

9. Если тестовая проверка увенчалась успехом, необходим заключительный запуск программы на решение с вводом исходных данных для всех входных параметров, заданных в постановке задачи (см. п. 1). Далее – получение практических результатов, их анализ и физическая интерпретация в соответствии с п. 1.

❓ Вопросы и задания

1. Дайте определение понятия «алгоритм».
2. Перечислите свойства алгоритма.
3. Назовите основные этапы подготовки задач к их автоматизированному решению с помощью ЭВМ.
4. Какие действия необходимо сделать при несовпадении контрольного решения задачи?
5. Что происходит на этапе построения математической модели задачи?
6. Каким образом осуществляется отладка программы? Разработайте словесный алгоритм, описывающий действия данного этапа.

Тема 2. СПОСОБЫ ЗАПИСИ АЛГОРИТМОВ. ТИПОВЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ

2.1. Формы записи алгоритмов

Разработку алгоритма решения задачи выполняют сначала на языке, близком и понятном его разработчикам. Форма записи алгоритма определяется как раз языком, на котором сформулированы все его предписания. Будем рассматривать три основные формы. Две из них могут быть использованы на предварительном этапе обдумывания и составления алгоритма, а одна – как заключительная – применяется для ввода в ЭВМ.

1. Словесная, или текстовая, форма – это пронумерованная последовательность предложений на естественном языке (разговорном языке пользователя) с применением математических обозначений и формул. Такое представление весьма слабо формализовано, а поэтому под сомнением и свойство определенности алгоритма. Зато для записи в этой форме не нужны никакие специальные знания, кроме знания собственно самого плана решения задачи.

2. Графическая, или схемная, форма (блок-схема) – это пронумерованная последовательность блоков различной конфигурации в зависимости от типа выполняемого предписания (т. е. каждая фигура – это очередное предписание алгоритма). Блоки соединены линиями связи в соответствии с порядком исполнения. Внутри блоков допустимы только математические записи. Конфигураций основных блоков столько, сколько выделено основных предписаний для построения вычислительных алгоритмов.

Преимущества графической записи:

- представление алгоритмов очень формализовано;
- алгоритм нагляден по структуре и характеру исполнения.

3. Алгоритм, представленный ***на языке программирования,*** – это последовательность (не всегда пронумерованная) «предложений» на выбранном алгоритмическом языке с использованием математической символики в алфавите этого языка. Каждое «предложение» (очередное предписание алгоритма) называется ***оператором,*** а весь алгоритм, записанный в виде последовательности операторов, – ***программой.*** Такое представление формализовано, используется обычно для ввода алгоритма в ЭВМ, а получается в результате «перевода» разра-

ботанного алгоритма в любой из перечисленных выше форм на алгоритмический язык: каждое предписание переводится (заменяется) последовательно «своим» оператором. Для простых задач и при хорошем знании языка программирования возможно написание программы сразу, без промежуточных форм.

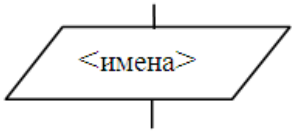
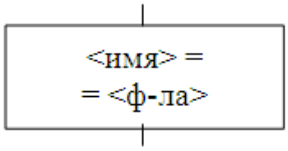
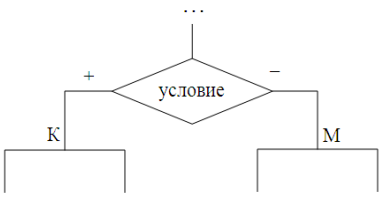
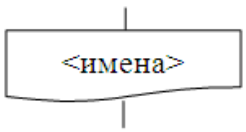
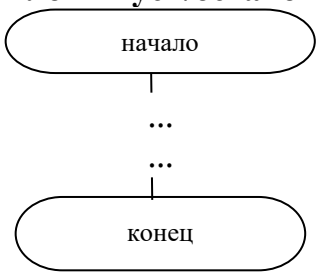
2.2. Основной набор элементарных предписаний алгоритма

Все многообразие вычислительных задач может быть алгоритмически описано с использованием конечного *набора элементарных предписаний.* Элементарным предписанием называется такая инструкция алгоритма, которая однозначно понимается и выполняется исполнителем этого алгоритма. Этот набор достаточен, какой бы сложности вычислительная задача ни была, и имеет соответствующие операторы-аналоги в любом языке программирования высокого уровня (ЯВУ). Представим этот перечень приказов (с необходимыми пояснениями) в двух формах: текстовой (на русском языке) и графической (блоками) в таблице ниже. Разрешенные геометрические размеры блоков сообщаются в соответствующих стандартах, для их изображения существуют специальные формы-лекала, а для их компьютерного «рисования» могут быть использованы соответствующие автофигуры. Если не требуется строгого следования размерам стандартов, то для приемлемого внешнего вида рекомендуется соблюдать одинаковые габаритные размеры всех блоков в пределах одного алгоритма, причем размер по вертикали (высоту) обычно задают в 1,5 – 2 раза меньше размера по горизонтали (длины). Исключение составляет блок «пуск/останов», вертикальный размер которого в четыре раза меньше габаритной длины, поэтому он легко отличается от других блоков и может не заполняться.

В случае когда блок-схема не умещается на один лист, стоит организовывать переносы блок-схемы по правилу: неоконченную ветвь в конце помечают буквой латинского алфавита, заключенной в круг, и начинается соответствующая ветвь на другой странице с той же самой буквы в круге.

В табл. 1 приведены наиболее часто используемые элементы блок-схем, расширенный перечень обозначений изложен в ГОСТ 19.701-90 «ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения».

Таблица 1. Основные элементы блок-схемы

Тип	Текстовая форма	Графическая форма	Пояснения
1	Ввести (значения для): <имена входных переменных>	**Блок-ввод** …  …	После ключевого слова или внутри блока указывают список имен входных переменных через запятую, но не сами числа
2	Вычислить: <имя>=<формула>	**Блок-процесс** …  …	После ключевого слова или внутри блока записывают имя вычисляемой переменной, знак = (присваивания) и формулу, содержащую переменные, числа, арифметические операции, элементарные функции
3	Сравнить: если <условие>, то п. К иначе п. М	**Блок-принятие решения**  	Условие – это логическое выражение (например, сравнения чего-то с чем-то). При выполнении условия (истина) – переход к одному пункту (например, к п. К), при невыполнении (ложь) – переход к другому (например, к п. М)
4	**Вывести** (напечатать): <имена выходных переменных>	**Блок-документ** …  …	После ключевого слова или внутри блока указывают список имен выходных переменных через запятую, но не сами числа
5	**Начало** решения …. …. **Остановка** (конец решения)	**Блоки пуск/останов**  	Если блок находится в начале алгоритма, то допустимы слова *вкл., вход, пуск, готов, начало* или оставить пусто. Конец решения помечают обязательно, допустимы слова *конец*, *останов.*, *выкл.*, *выход*, *стоп* или оставить блок пустым

2.3. Основные типы простейших алгоритмических структур

Для построения алгоритмов решения вычислительных задач требуется минимум пять типов элементарных операций. Опыт показывает, что все многообразие вычислительных задач при их решении можно описать также ограниченным набором сочетаний этих элементарных предписаний в более крупные вычислительные структуры. Постепенно, разбивая задачи на подзадачи, сводим их решение к некоторым типовым фрагментам алгоритмов.

Можно выделить три основных типа алгоритмов:

1. Линейный – это такая вычислительная структура, при которой все предписания выполняются в строго линейной последовательности друг за другом, как записаны; сам порядок исполнения называется естественным. В чистом виде линейные алгоритмы встречаются редко, они чрезвычайно просты. Как фрагменты они присутствуют почти во всех вычислительных схемах, так как именно на этих участках вводятся исходные данные, формируются и выводятся пользователю новые результаты (блок-ввод, блок-процесс, блок-документ).

2. Разветвляющийся – это такая вычислительная схема, которая содержит не одну, а несколько возможных ветвей решения по математической модели; т. е. в структуре алгоритма есть хотя бы одна операция сравнения, порождающая две ветви последующего решения. Заранее нельзя сказать, какая ветвь алгоритма будет выполняться, все зависит от конкретных данных. Выбирается эта ветвь автоматически, только в процессе исполнения алгоритма (остальные ветви для этого варианта данных останутся невостребованными). Следовательно, на этапе разработки пользователю труднее предусмотреть все возможные ветви решения и изобразить их, ничего не упустив.

3. Циклический – это такая схема разветвленной структуры, в которой одна ветвь операции сравнения представляет собой обратную связь (ОС) на предыдущую часть алгоритма (т. е. идет назад). Таким образом, некоторая последовательность операций алгоритма будет выполняться многократно (в цикле), образуя тело цикла (ТЦ). В крайнем случае тело цикла может состоять всего из одной операции, в некоторых случаях операции могут отсутствовать вовсе. Для того чтобы результаты всякий раз получались новые, необходимо алгоритмически предусмотреть три момента:

а) надо подготовить данные для первой итерации цикла;

b) после каждого выполнения тела цикла нужно обновлять данные для очередного прохождения цикла;

c) необходимо управлять циклом, организовать условие выхода из цикла или его продолжения (ОС не должна охватывать указанные в пункте *a* операции!).

В зависимости от конкретных вариантов исходных данных указанные мероприятия реализуются по-разному, но в общем случае обобщенная структура простого цикла имеет вид, представленный на рис. 2.1 и рис. 2.2.

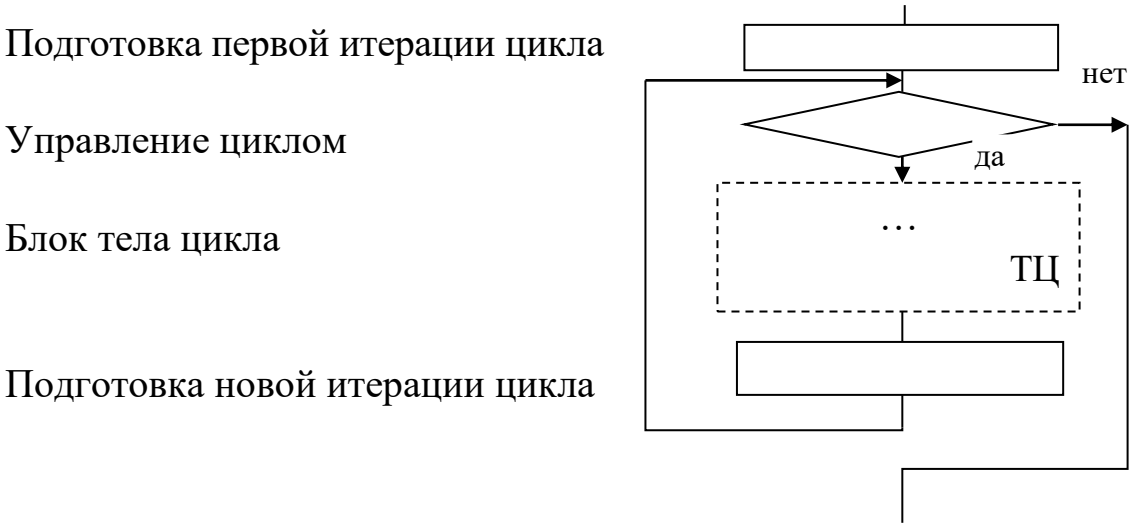


Рис. 2.1. Схема цикла с предусловием

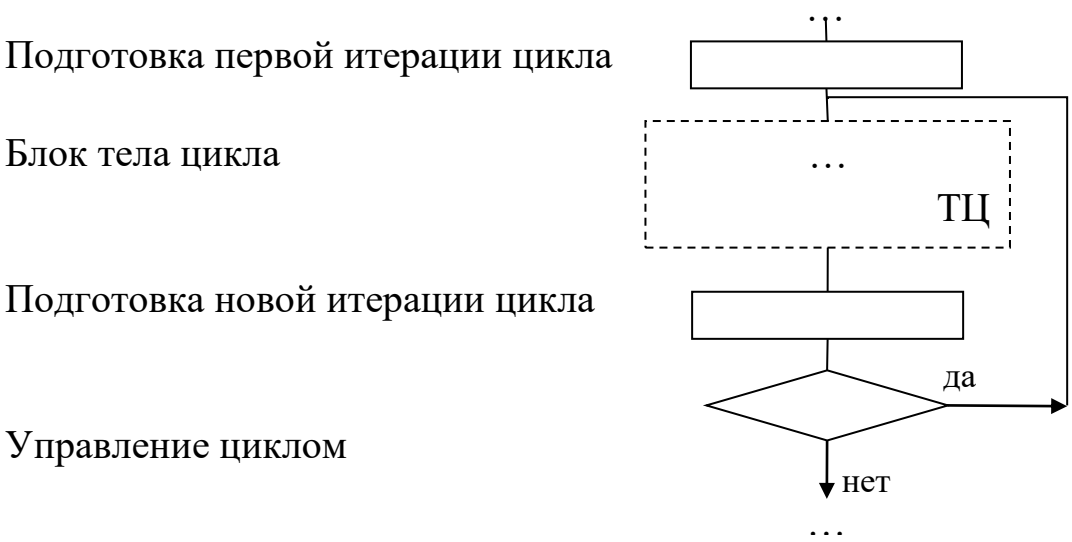


Рис. 2.2. Схема цикла с постусловием

Первый случай (см. рис. 2.1) – блок управления стоит перед телом цикла (цикл с предусловием), его используют для задач, требующих проверки условия выполнения цикла перед вхождением в него (цикл может вообще ни разу не выполниться). Второй случай (см. рис. 2.2) – блок управления после тела цикла (цикл с постусловием) – для задач, в которых один раз цикл выполняется обязательно. Примеры существующих способов заполнения обязательных блоков цикла будут рассмотрены далее.

Можно выделить три способа задания исходных данных для задач, реализуемых простыми циклами, а следовательно, и три способа заполнения обязательных блоков типовой алгоритмической структуры:

Способ 1: $x_н$ – начальное значение x, $x_к$ – конечное значение x, d – шаг изменения x (узлы по x равноотстоящие), y – многократно вычисляемый результат с выводом на экран.

Представим графически по исходным данным область и допустимые значения x (рис. 2.3):

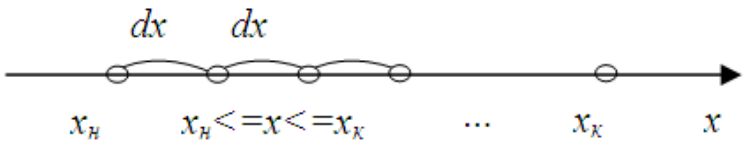


Рис. 2.3. Задание исходных данных. Способ 1

Способ 2: $x_н$ – начальное значение x, dx – шаг изменения x (узлы по x равноотстоящие), N – количество расчетных точек, y – многократно вычисляемый и выводимый на экран результат. Представим графически по исходным данным область и допустимые значения x (рис. 2.4):

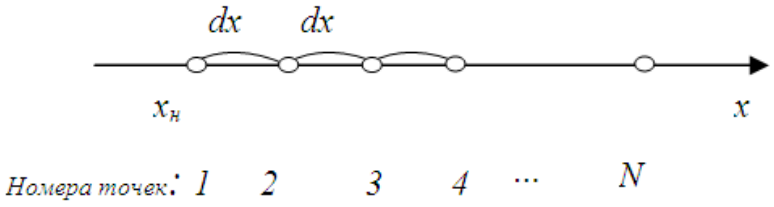


Рис. 2.4. Задание исходных данных. Способ 2

Способ 3: задание исходных данных для изменения x в процессе счета: известно, что узлов по оси x N штук, но они неравноотстоящие,

поэтому вычислить следующий x по предыдущему не удается. В данном случае удобно иметь в распоряжении все заданные значения x: $X = (x_1, x_2, \ldots, x_n)$. В программировании такой вариант задания аргумента называется числовым массивом, а каждое отдельное значение в нем – элементом массива (рис. 2.1).

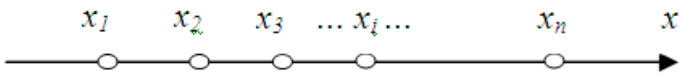


Рис. 2.1. Задание исходных данных. Способ 3

Для ввода всех элементов массива в алгоритме используется конструкция вида $\{x_i\},\ i = \overline{1,n}$ – поэлементный ввод массива X.

В современных высокоуровневых языках программирования существуют операторы, вбирающие в себя несколько элементарных предписаний разработанного алгоритма. В первую очередь это касается типовых циклов как наиболее востребованных при построении вычислений. Все подготовительные блоки цикла размещаются в одном специальном операторе цикла (обычно в языке их несколько видов, и нужный обязательно найдется), который ставят перед операторами тела цикла. Это способствует приближению текста программы к логике человеческого (а не машинного) мышления, ускоряя и облегчая процесс программирования. Чтобы приблизиться к выбранному языку программирования, полнее использовать его возможности, в литературе по алгоритмизации вычислений стали применять специальный блок, который называют блоком цикла, или блоком-модификатором, или блоком-заголовком цикла (рис. 2.6).

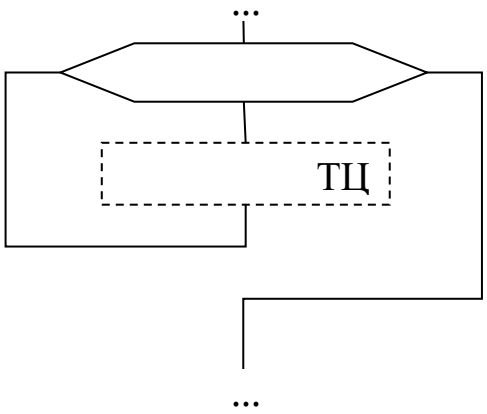


Рис. 2.6. Блок-цикл

Этот блок имеет специальную структуру, предшествует телу цикла и размещает в себе все дополнительные операции по организации цикла. Переменная, которая управляет циклом, называется его параметром. В блоке цикла указывают его параметр, знак присваивания (=), начальное значение параметра, конечное значение параметра, шаг изменения параметра. Например, $i = 1, n, 1$. Некоторые языки программирования допускают параметр

лишь целого типа, другие – и целого, и вещественного (при построении алгоритма это надо знать и учитывать).

? Вопросы и задания

1. Назовите и охарактеризуйте формы записи алгоритмов.
2. В чем преимущества графической формы записи алгоритма?
3. Изобразите элемент «Блок принятия решений» и поясните принцип его заполнения и работы.
4. Перечислите типы циклов.
5. Перечислите способы задания исходных данных циклов. Приведите примеры конкретных задач.
6. Изобразите блок-схему алгоритма решения задачи вычисления корней квадратного уравнения.
7. Изобразите блок-схему алгоритма решения задачи подсчета суммы элементов конечной последовательности.

Тема 3. КОМБИНИРОВАННЫЕ АЛГОРИТМЫ

3.1. Вложенные циклы

Вложенные циклы (цикл в цикле) – это такие циклические алгоритмы, которые имеют несколько переменных (более 1), каждая из которых меняется по своему закону (одному из предложенных выше).

Рассмотрим функцию двух аргументов $Y = \frac{x^2}{x + a}$, где x и a меняются независимо друг от друга. Требуется провести вычисление значений функции в наборе точек координатной сетки.

Исходные данные для x и a:

$x_н$ – начальное значение x; $a_н$ – начальное значение a;

$x_к$ – конечное значение x; $a_к$ – конечное значение a;

h_x – шаг изменения x; h_a – шаг изменения a.

Точки, в которых вычисляется Y, можно изобразить некоторой областью D. Для построения алгоритма нельзя использовать в чистом виде ни один приведенный ранее простой цикл, так как он дает одновременное изменение x и a и вычисление Y лишь в диагональных точках области D.

Необходимо построить два независимых цикла, вкладывая их один в другой. Для правильной вложенности циклов нужно предварительно определить, какой аргумент образует внутренний цикл и какой – внешний.

С точки зрения решаемой задачи совершенно безразлично, как сделать эти назначения; меняется только порядок перебора расчетных точек в области *D* (но ни одна пропущена не будет). Пусть *a* – внутренний аргумент, т. е. меняется в первую очередь, тогда *x* образует внешний цикл (порядок перебора точек при этом показан на рис. 3.1).

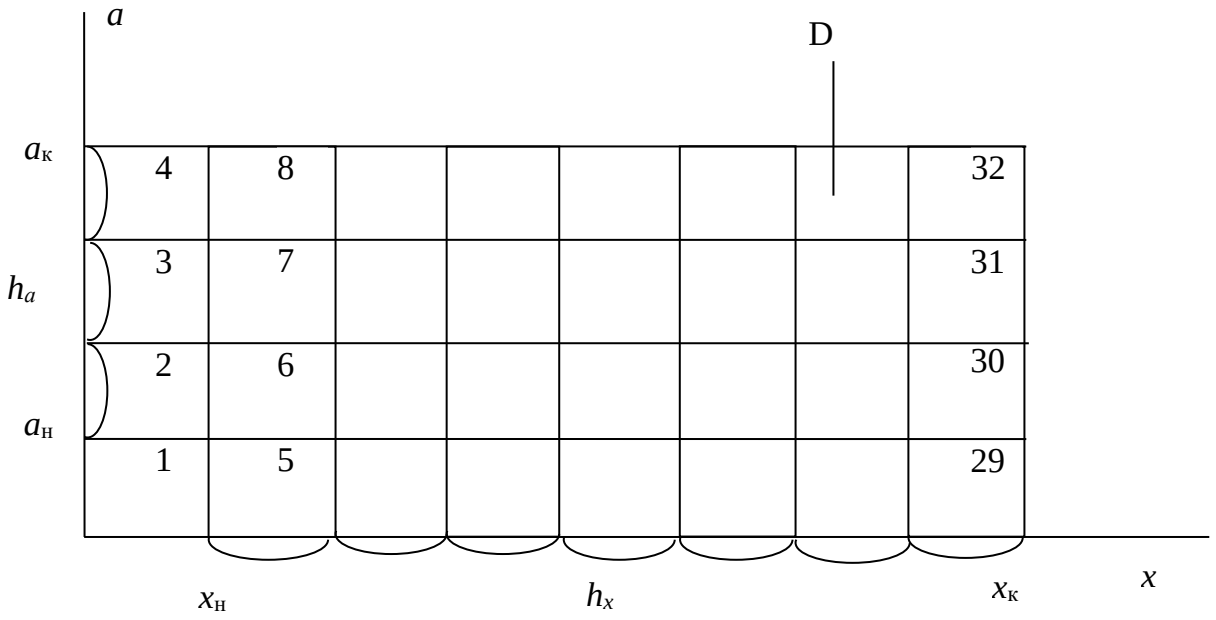


Рис. 3.1. Схема перебора точек для вложенных циклов

Далее строим блок-схему алгоритма реализации поставленной задачи (рис. 3.2).

Для соблюдения свойств алгоритма все исходные данные для него вводим с клавиатуры. Далее указываем блок подготовки первого прохода внешнего цикла. В теле внешнего цикла по параметру *x* будет располагаться цикл по перебору значений параметра *a*. Как только все значения параметра *a* закончатся, то произойдет переход к новой точке

по x. Алгоритм будет работать до тех пор, пока все точки не будут просчитаны. Заметим, что в теле внутреннего цикла осуществляются непосредственный расчет значения функции и вывод рассчитанных значений на экран.

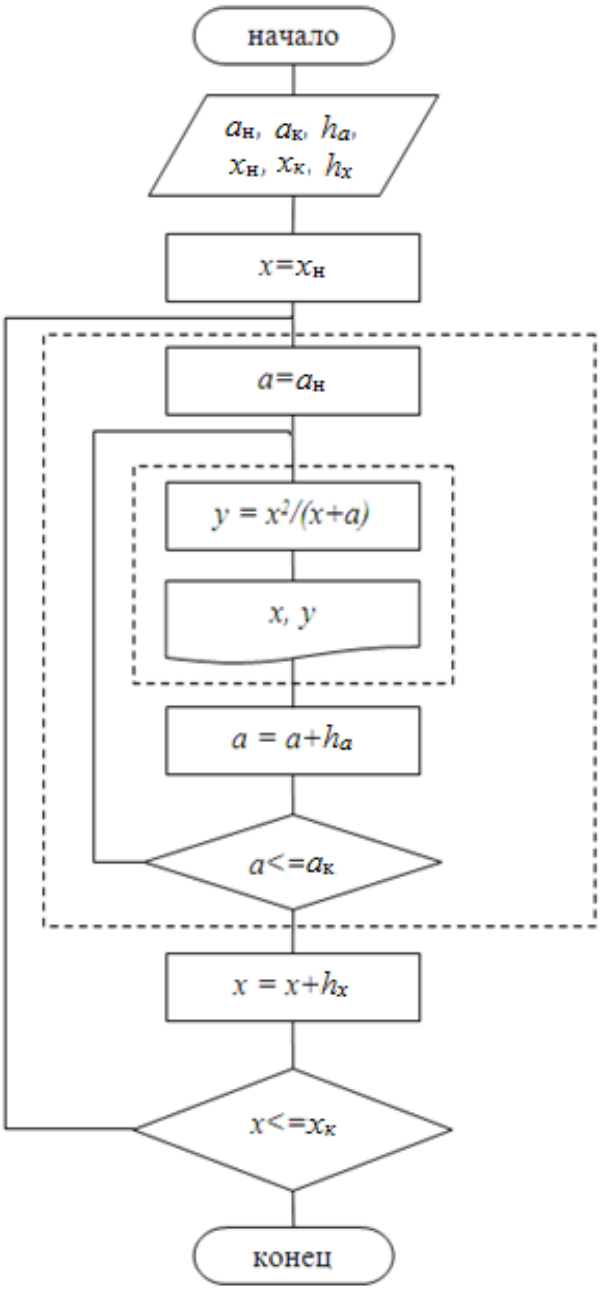


Рис. 3.2. Блок-схема алгоритма с вложенным циклом

3.2. Циклы накопления конечной суммы (или произведения)

Часто в математических моделях встречаются макрооперации многоместного суммирования или многоместного произведения однотипных слагаемых (множителей), сконструированных на элементах заданного числового массива:

$\sum_{i=1}^{n}$ *(<i-е слагаемое>)*

или $\prod_{i=1}^{n}$ *(<i-й множитель>)*,

где $\sum_{i=1}^{n}$ – операция многоместной суммы, $\prod_{i=1}^{n}$ – операция многоместного произведения, $i=1$ – номер первого операнда, n – его последний номер.

Таким образом, можно изобразить сумму (или произведение) однотипных операндов, построенных по какому-либо правилу на элементах заданного числового массива конечной длины. Например,

$$S=\sum_{i=1}^{50} x_i^2 = x_1^2+\ldots+x_i^2+\ldots+x_{50}^2 \text{ или } P=\prod_{i=1}^{40}\frac{x_i^2}{2}=\frac{x_1^2}{2}\cdot\ldots\cdot\frac{x_i^2}{2}\cdot\ldots\cdot\frac{x_{40}^2}{2}.$$

В алгоритмах знаки макроопераций Σ, П не употребляются (не считаются элементарными). В развернутом виде формулу при записи алгоритма использовать нежелательно из-за громоздкости, а заменять часть операндов по формуле многоточием недопустимо. Для алгоритмизации вычислений подобного типа существует прием циклического «накопления» результата последовательным добавлением к предыдущей частичной сумме нового слагаемого или последовательным домножением предыдущего частичного произведения на новый множитель.

Циклов получается ровно столько, сколько операндов, а операндов ровно столько, сколько элементов в заданном массиве. Таким образом, цикл должен быть организован по счетчику. Для выполнения операции накопления вводят дополнительную переменную «накопления», которая предварительно (перед телом цикла) обнуляется при накоплении суммы или получает значение «единица» при накоплении произведения. В операции «накопления» в зависимости от ее вида используют следующее формулы:

для суммы	для произведения
$S = S + <$ *i-е слагаемое*$>$	$P = P \cdot <$ *i-й множитель*$>$.

3.3. Разветвленное тело цикла. Досрочный выход из тела цикла

В данном типе алгоритмов совмещаются схемы разветвленного алгоритма тела цикла. Досрочный выход означает, что одна из ветвей выходит за пределы тела цикла, а другая продолжает цикл.

Пример такой задачи: «Требуется выполнить расчет значений функции, кроме точки $x = 0$, так как в этой ситуации значение функции не существует».

❓ Вопросы и задания

1. Поясните принципы построения вложенных циклов.
2. Может ли при создании вложенных циклов внешний цикл быть с предусловием, а внутренний – с постусловием?
3. Приведите пример задачи, алгоритмическим решением которой станет цикл с досрочным выходом.

Тема 4. ВВЕДЕНИЕ В C++

4.1. Структура программы

Исходная программа представляет собой совокупность следующих элементов: директив препроцессора, указаний компилятору, объявлений и определений.

Директивы препроцессора специфицируют действия препроцессора по преобразованию текста программы перед компиляцией.

Наиболее часто применяемые директивы препроцессора приведены ниже.

1. Директива *#define идентификатор_«строка_токена»* – создает макрос, представляющий собой ассоциацию обычного или параметризованного идентификатора со строкой_токена. После определения макроса компилятор может подставить строку_токена для каждого обнаруженного идентификатора в исходном файле.

Примеры применения:

а) #define getMax(a,b) a<b?b:a – если в тексте программы встретится текст getMax(a,b), то все такие фрагменты будут заменены на код условной операции a<b?b:a;

б) #define a 5 – вместо переменной a будет проведена замена на значение 5.

2. Директива #include "path-spec" или #include <path-spec> – позволяет подключить внешние включаемые файлы, содержащие объявления констант и функций, которые мы будем применять в программе.

Пример применения:

#include <iostream> – подключение библиотеки потокового ввода-вывода.

Существуют такие понятия, как сборка, условная компиляция, которые также задаются с помощью директив препроцессора. Более подробно с ними можно ознакомиться в официальной документации по языку C++ на сайте msdn.microsoft.com.

Указания компилятору – это специальные инструкции, которым компилятор языка C++ следует во время компиляции. Примерами могут служить различные операторы, например, циклические, условный, различные операции над переменными в программе, вызовы функций и т. д.

Объявление переменной задает имя, атрибуты переменной и приводит к выделению для нее памяти. Определение переменной, помимо задания ее имени, атрибутов, выделения для нее памяти, устанавливает начальное значение переменной (явно или неявно).

Объявление функции задает ее имя, тип возвращаемого значения и может настраивать атрибуты ее формальных параметров.

Определение функции специфицирует тело функции, которое представляет собой составной оператор (блок), содержащий объявления и операторы. Определение функции также задает ее имя, тип возвращаемого значения и атрибуты ее формальных параметров.

Объявление типа позволяет программисту создать собственный тип данных. Оно состоит в присвоении имени некоторому базовому или составному типу языка C++. Для типа понятия объявления и определения совпадают.

Исходная программа может содержать произвольное число директив, указаний компилятору, объявлений и определений. Порядок появления этих элементов в программе весьма существен: в частности, он влияет на возможность использования переменных, функций и типов в различных частях программы.

Для того чтобы программа на языке C++ могла быть скомпилирована и выполнена, она должна содержать по крайней мере одно определение – определение функции. Эта функция содержит действия, выполняемые программой. Если же программа содержит несколько функций, то среди них выделяется одна главная, которая должна иметь имя *main* или *_tmain* для работы в среде Microsoft Visual Studio. С нее начинается выполнение программы; она определяет действия, выполняемые программой, и вызывает другие функции. Порядок следования определений функций в исходной программе несуществен.

Структура программы C++ представлена ниже, здесь f1() – fN() – функции, написанные программистом.

```
// Директивы препроцессора
// Объявление глобальных переменных
```

```
// Объявление функций
int main (список параметров)
{
     //Последовательность операторов
}
Тип_возвращаемого_значения f1 (список параметров)
{
     //Последовательность операторов
}
.
.
.
Тип_возвращаемого_значения fN (список параметров)
{
     //Последовательность операторов
}
```

В следующем примере приведена простая программа на языке C++:

```
#include <stdio.h> /* директива препроцессора для под-
ключения заголовочных файлов */
int x = 1; /* определения глобальных переменных */
int y = 2;
void main () /* определение главной функции */
{
     int z, w; /* объявления переменных */
     z = y + x; /* выполняемые операторы */
     w = y - x;
     printf("z = %d\nw = %d.", z, w); /*вывод на
экран*/
}
```

Результатом работы данной программы будут строки z = 3
w = 1.

4.2. Ключевые слова языка C++

Ключевые слова – это предопределенные идентификаторы, которые имеют специальное значение для компилятора языка C++. Их использование строго регламентировано. Имена объектов программы не могут совпадать с ключевыми словами.

Приведем перечень ключевых слов с оговоркой, что он не полный: нам важно понимать, какие объекты в программе описываются ключевыми словами – названия типов данных и их модификаторов, операторы, метки видимости:

auto	continue	else	for	long	signed	switch	void
break	default	enum	goto	register	sizeof	typedef	while
case	do	extern	if	return	static	union	
char	double	float	int	short	struct	unsigned	

В языке C++ различаются верхний и нижний регистры символов: else – ключевое слово, а ELSE – нет. В программе ключевое слово можно применять только как ключевое, т. е. никогда не допускается его использование в качестве имени переменной или функции.

Идентификатор должен удовлетворять правилам именования переменных в C++, и нельзя использовать в одной области переменные с одинаковыми именами. Переменные автоматически не инициализируются значениями, они имеют значение («мусор»), которое хранится в памяти по адресу переменной.

Правила именования переменных гласят:

1) имя переменной не должно начинаться с цифры;

2) имя переменной не должно включать следующие символы: пробел, /, :, *, ?, ", <, >, |.

4.3. Базовые типы данных

В C++ определены пять фундаментальных типов данных:

1) *char* – символьные данные;
2) *int* – целочисленные;
3) *float* – числа с плавающей точкой;
4) *double* – с плавающей точкой двойной точности;
5) *void* – без значения.

На основе этих типов формируются другие типы данных. Размер (объем занимаемой памяти) и диапазон значений этих типов данных для разных процессов и компиляторов могут быть разными. Однако объект типа *char* всегда занимает 1 байт. Размер объекта *int* обычно совпадает с размером слова в конкретной среде программирования. В большинстве

случаев в 16-разрядной среде (DOS или Windows 4.l) *int* занимает 16 бит, а в 32-разрядной (Windows 95/98/NT/2000) – 32 бита.

При разработке программ в Microsoft Visual Studio тип *int* занимает 32 бита или 4 байта. Однако полностью полагаться на это нельзя, особенно при переносе программы в другую среду. Необходимо помнить, что стандарт C++ обусловливает только *минимальный диапазон значений* каждого типа данных, но не размер в байтах. Для определения размера типа в байтах можно применить команду *sizeof (int).*

Конкретный формат числа с плавающей точкой зависит от его реализации в трансляторе. Переменные типа *char* обычно используются для обозначения набора символов стандарта ASCII. Символы, не входящие в этот набор, различными компиляторами обрабатываются по-разному.

Диапазоны значений типов *float* и *double* зависят от формата представления чисел с плавающей точкой.

Тип *void* служит для объявления функции, не возвращающей значения, или для создания универсального (нетипизированного) указателя.

Базовые типы данных (кроме *void*) могут иметь различные *спецификаторы,* предшествующие им в тексте программы. Спецификатор типа так изменяет значение базового типа, чтобы он более точно соответствовал своему назначению в программе.

Полный список спецификаторов типов:

- ✓ signed;
- ✓ unsigned;
- ✓ long;
- ✓ short.

Базовый тип *int* может быть модифицирован каждым из этих спецификаторов. Тип *char* модифицируется с помощью *unsigned* и *signed, double* – с помощью *long.* В табл. 2 приведены все допустимые комбинации типов данных с их минимальным диапазоном значений и типичным размером.

Таблица 2. Типы данных языка C++

Имя типа	Байты	Синонимы	Диапазон значений
int	4	signed	От –2147483648 до 2147483647
unsigned int	4	без знака	От 0 до 4 294 967 295
__int8	1	char	От –128 до 127
unsigned __int8	1	unsigned char	От 0 до 255
__int16	2	short, signed short int	От –32 768 до 32 767
unsigned __int16	2	unsigned short, unsigned short int	От 0 до 65 535
__int32	4	signed, int, signed int	От –2 147 483 648 до 2 147 483 647
unsigned __int32	4	unsigned, unsigned int	От 0 до 4 294 967 295
__int64	8	long long, signed long long	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
unsigned __int64	8	unsigned long long	От 0 до 18 446 744 073 709 551 615
bool	1	Нет	false или true
char	1	Нет	По умолчанию – от –128 до 127. При компиляции при помощи /J – от 0 до 255
signed char	1	Нет	От –128 до 127
unsigned char	1	Нет	От 0 до 255
short	2	short int, signed short int	От –32 768 до 32 767
unsigned short	2	unsigned short int	От 0 до 65 535
long	4	long int, signed long int	От –2 147 483 648 до 2 147 483 647
unsigned long	4	unsigned long int	От 0 до 4 294 967 295
long long	8	Нет (эквивалентно __int64)	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
unsigned long long	8	Нет (эквивалентно unsigned__int64)	От 0 до 18 446 744 073 709 551 615
float	4	Нет	3,4E +/–38 (7 знаков)
double	8	Нет	1,7E +/–308 (15 знаков)
long double	8	Нет	1,7E +/–308 (15 знаков)
wchar_t	2	__wchar_t	От 0 до 65 535

Для целых можно использовать спецификатор *signed*, но в этом нет необходимости, потому что при объявлении целого он предполагается по умолчанию. Спецификатор *signed* чаще всего используется типом *char,* который в некоторых реализациях по умолчанию может быть беззнаковым.

Целые числа со знаком и без знака отличаются интерпретацией нулевого бита. Если целое объявлено со знаком, компилятор считает, что нулевой бит содержит знак числа. Если в нулевом бите записан ноль, число считается положительным, а если единица – отрицательным.

В большинстве реализаций отрицательные числа представлены в *двоичном дополнительном коде.* Это означает, что для отрицательного числа все биты, кроме нулевого, инвертируются, к полученному числу добавляется единица, нулевой бит устанавливается в единицу.

Целые числа со знаком используются почти во всех алгоритмах, но абсолютная величина наибольшего из них составляет примерно только половину максимального целого без знака. Например, знаковое целое число 32 767 в двоичном коде имеет вид

0111111111111111.

Если в нулевой бит записать 1, то оно будет интерпретироваться как –1. Однако если полученную запись рассматривать как представление числа, объявленного как *unsigned int*, то оно будет интерпретироваться как 65 535.

Если спецификатор типа записать сам по себе (без следующего за ним базового типа), то предполагается, что он модифицирует тип *int*. Хотя базовый тип *int* и предполагается по умолчанию, его тем не менее обычно указывают явно.

4.4. Операции и функции на базовых типах данных C++. Приоритеты и ассоциативность операций

В языке C++ операции с высшими приоритетами вычисляются первыми. Приоритет операторов определяет порядок операций в выражениях, содержащих более одного оператора. Ассоциативность оператора определяет, группируется ли операнд в выражении, содержащем несколько операторов с одинаковым приоритетом, с оператором слева от него или справа. В табл. 3 показан приоритет и ассоциативность операторов C++ (в порядке убывания приоритета). Операторы с тем же номером приоритета имеют равный приоритет, если другие связи не заданы явно с помощью круглых скобок.

Таблица 3. Приоритеты операций

Приоритет	**Знак операции**	**Тип операции**	**Порядок выполнения**
1	:: – разрешение области	Унарные	Нет
2	( ) – вызов функции; [ ] – нижний индекс массива; . – выбор члена для объекта; –> – выбор члена для указателя; ++ – постфиксный инкремент; -- – постфиксный декремент; typeid() – имя типа; Приведение типов	Выражение	Слева направо
3	sizeof – размер объекта или типа; – унарный минус; + – унарный плюс; ~ – дополнение до единицы; ! – логическое НЕ; * – косвенное обращение; & – взятие адреса; ++ – префиксный инкремент; -- – префиксный декремент; new – создание объекта; delete – удаление объекта	Унарные	Справа налево
4	.* – указатель на член составного объекта; ->*– указатель на член составного объекта	Унарные	Слева направо
5	* – умножение; / – деление; % – остаток от деления	Мультипликативные	
6	+ – сложение; – – вычитание	Аддитивные	
7	<< – сдвиг влево; >> – сдвиг вправо	Сдвиг	
8	< – меньше; > – больше; <= – меньше или равно; >= – больше или равно	Отношение	
9	== – равенство; != – неравенство	Отношение (равенство)	
10	& – побитовое И	Поразрядное И	

Окончание табл. 3

Приоритет	Знак операции	Тип операции	Порядок выполнения
11	^ – побитовое исключающее ИЛИ	Поразрядное исключающее ИЛИ	Слева направо
12	| – побитовое ИЛИ	Поразрядное ИЛИ	
13	&& – логическое И	Логическое И	
14	|| – логическое ИЛИ	Логическое ИЛИ	
15	? : – условие	Условная	
16	= – присваивание; *= – присваивание умножения; /= – присваивание деления; %= – присваивание модуля; += – присваивание сложения; –= – присваивание разности; >>= – присваивание сдвига вправо; <<= – присваивание сдвига влево; &= – назначение побитового И; |= – назначение побитового ИЛИ; ^= – назначение побитового исключающего ИЛИ	Простое и составное присваивания	Справа налево
17	throw – обработчик исключений	Обработка исключительных ситуаций	
18	, – операция запятая	Последовательное вычисление	Слева направо

Мультипликативные операции

К этому классу операций относятся операции умножения (*), деления (/) и получения остатка от деления (%). Операндами операции (%) должны быть целые числа. Отметим, что типы операндов операций умножения и деления могут отличаться, и для них справедливы правила преобразования типов. Тип результата – это тип операндов после преобразования (преобразование происходит к типу, имеющему большой размер).

Операция умножения (*) выполняет умножение операндов.

Пример:

```
int i = 5;
float f = 0.2;
double g;
g = f*i; /* результат 1.0 */
```

Тип произведения i и f преобразуется к типу double, затем результат присваивается переменной g = 1.0.

Операция деления (/) выполняет деление первого операнда на второй. Если две целые величины не делятся нацело, то результат округляется в сторону нуля.

При попытке деления на ноль выдается сообщение во время выполнения.

Пример:

```
int i=49, j=10, n, m;
n = i/j; /* результат 4 */
m = i/(-j); /* результат -4 */
```

Операция остатка от деления (%) дает остаток от деления первого операнда на второй. При этом знак результата зависит от конкретной реализации. В данной реализации знак результата совпадает со знаком делимого. Если второй операнд равен нулю, то выдается сообщение.

Пример:

```
int n = 49, m = 10, i, j, k, l;
i = n % m; /* 9 */
j = n % (-m); /* 9 */
k = (-n) % m; /* -9 */
l = (-n) % (-m); /* -9 */
```

Аддитивные операции

К аддитивным операциям относятся сложение (+) и вычитание (–). Операнды могут быть целого или плавающего типов. В некоторых случаях над операндами аддитивных операций выполняются общие арифметические преобразования. Однако такие преобразования не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат аддитивной операции не может быть представлен типом операндов после преобразования. При этом сообщение об ошибке не выдается.

Пример:

```
short i = 30000, j = 30000, k;
k = i + j;
```

В результате сложения *k* получит значение равное –5536.

Результатом выполнения операции сложения становится сумма двух операндов. Операнды могут быть целого или плавающего типа или один операнд может быть указателем, а второй – целой величиной.

Когда целая величина складывается с указателем, то она преобразуется путем умножения ее на размер памяти, занимаемой величиной, адресуемой указателем.

Когда преобразованная целая величина складывается с величиной указателя, то результатом становится указатель, адресующий ячейку памяти, расположенную на целую величину дальше от исходного адреса. Новое значение указателя адресует тот же самый тип данных, что и исходный указатель.

Операция вычитания (–) вычитает второй операнд из первого. Возможны следующие комбинации операндов:

1. Оба операнда целого или плавающего типа.
2. Оба операнда – указатели одного типа.
3. Первый операнд – это указатель, а второй – целое.

Отметим, что операции сложения и вычитания над адресами в единицах, отличных от длины типа, могут привести к непредсказуемым результатам.

Операции сдвига

Операции сдвига осуществляют смещение операнда влево (<<) или вправо (>>) на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами. Выполняются обычные арифметические преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип *unsigned*, то свободные левые биты устанавливаются в нуль. В противном случае они заполняются копией знакового бита. Результат операции сдвига не определен, если второй операнд отрицательный.

Преобразования, выполненные операциями сдвига, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если после преобразования результат операции сдвига не может быть представлен типом первого операнда.

Отметим, что сдвиг влево соответствует умножению первого операнда на степень числа два, равную второму операнду, а сдвиг вправо соответствует делению первого операнда на два в степени, равной второму операнду.

Пример:

```
short i = 0x1234, j, k;
k = i << 4; /* k="0x0234" */
j = i << 8; /* j="0x3400" */
i = j >> 8; /* i = 0x0034 */
```

Поразрядные операции

К поразрядным операциям относятся операция поразрядного И (&), операция поразрядного ИЛИ (|), операция поразрядного исключающего ИЛИ (^).

Операнды поразрядных операций могут быть любого целого типа. При необходимости над операндами выполняются преобразования по умолчанию, тип результата – это тип операндов после преобразования.

При выполнении поразрядных операций сравнивают каждый бит первого операнда с соответствующим битом второго операнда, и результат операции выводится по табл. 4.

Таблица 4. Таблица истинности логических операций

Логические переменные		**Операции**		
		поразрядное И	поразрядное ИЛИ	поразрядное исключающее ИЛИ
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

Пример:

```
short i = 0x45FF, /* i= 0100 0101 1111 1111 */
j=0x00FF; /* j= 0000 0000 1111 1111 */
char r;
r = i^j; /* r=0x4500 = 0100 0101 0000 0000 */
r = i|j; /* r=0x45FF = 0100 0101 0000 0000 */
r = i&j /* r=0x00FF = 0000 0000 1111 1111 */
```

Операции отрицания и дополнения

Операция арифметического отрицания (–) вырабатывает отрицание своего операнда. Операнд должен быть целой или плавающей величиной. При выполнении осуществляются обычные арифметические преобразования.

Пример:

```
double u = 5;
u = -u; /* переменной u присваивается ее
     отрицание, т. е. u принимает значение –5 */
```

Операция логического отрицания «НЕ» (!) вырабатывает значение «нуль», если операнд есть истина (не нуль), и значение «единица», если операнд равен нулю. Результат имеет тип *int*. Операнд должен быть целого, или плавающего типа, или типа «указатель».

Пример:

```
int t, z=0;
t=!z;
```

Переменная t получит значение, равное единице, так как переменная z имела значение, равное нулю.

Операция двоичного дополнения (~) вырабатывает двоичное дополнение своего операнда. Операнд должен быть целого типа. Осуществляется обычное арифметическое преобразование, результат имеет тип операнда после преобразования.

Пример:

```
char b = '9';
unsigned char f;
b = ~f;
```

Шестнадцатеричное значение символа '9' равно 39. В результате операции ~f будет получено шестнадцатеричное значение C6, что соответствует символу 'ц'.

Операция sizeof

С помощью операции *sizeof* можно определить размер памяти, которая соответствует идентификатору или типу.

Операция *sizeof* имеет следующий формат:

```
sizeof выражение.
```

В качестве выражения может быть использован любой идентификатор либо имя типа, заключенное в скобки. Отметим, что не сле-

дует использовать имя типа *void*, а идентификатор не должен относиться к полю битов или быть именем функции.

Если в качестве выражения указано имя массива, то результатом становится размер всего массива (т. е. произведение числа элементов на длину типа), а не размер указателя, соответствующего идентификатору массива.

Когда *sizeof* применяют к имени типа структуры или объединения или к идентификатору, имеющему тип структуры или объединения, то результат представляет собой фактический размер структуры или объединения, который может включать участки памяти, используемые для выравнивания элементов структуры или объединения.

Логические операции

К логическим операциям относят операцию логического И (&&) и операцию логического ИЛИ (||). Операнды логических операций могут быть целого типа, плавающего типа или типа указателя, при этом в каждой операции допустимы операнды различных типов.

Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, то второй не вычисляется.

Логические операции не вызывают стандартных арифметических преобразований. Они оценивают каждый операнд с точки зрения его эквивалентности нулю. Результат логической операции – ноль или единица, тип результата – *int*.

Операция логического И (&&) вырабатывает значение единицы, если оба операнда имеют единые значения. Если оба или один из операндов равен нулю, то результат также равен нулю. Если значение первого операнда равно нулю, то второй не вычисляется.

Операция логического ИЛИ (||) вырабатывает значение нуля, если оба операнда имеют значение нуль; если какой-либо из операндов имеет ненулевое значение, то результат операции равен единице. Если первый операнд имеет ненулевое значение, то второй не вычисляется.

Операция последовательного вычисления

Операция последовательного вычисления обозначается запятой (,) и используется для вычисления двух и более выражений там, где по синтаксису допустимо только одно выражение. Эта операция вычис-

ляет два операнда слева направо. При выполнении операции последовательного вычисления преобразование типов не производится. Операнды могут быть любых типов. Результат операции имеет значения и тип второго операнда. Отметим, что запятая может использоваться также как символ-разделитель, поэтому необходимо по контексту различать запятую, используемую в качестве разделителя и в качестве знака операции.

Операции увеличения и уменьшения

Операции увеличения (++) и уменьшения (--) – это унарные операции присваивания, которые существуют в двух формах – префиксной и постфиксной. Они соответственно увеличивают или уменьшают значения операнда на единицу. Операнд может быть целого, или плавающего типа, или типа «указатель» и должен быть модифицируемым. Операнд целого или плавающего типа увеличивается (уменьшается) на единицу. Тип результата соответствует типу операнда. Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует.

Если знак операции стоит перед операндом (префиксная форма записи), то изменение операнда происходит до его использования в выражении и результатом операции становится увеличенное или уменьшенное значение операнда.

В том случае, если знак операции стоит после операнда (постфиксная форма записи), операнд вначале используют для вычисления выражения, а затем происходит изменение операнда.

Пример:

```
int t = 1, s = 2, z, f;
z = (t++)*5;
```

Вначале происходит умножение t на пять, а затем увеличение t на единицу. В результате получится z = 5, t = 2.

```
f=(++s)/3;
```

Вначале значение s увеличивается, а затем используется в операции деления. В результате получим s = 3, f = 1.

Если операции увеличения и уменьшения используются как самостоятельные операторы, префиксная и постфиксная формы записи становятся эквивалентными.

```
z++; /* эквивалентно */ ++z;
```

4.5. Основы ввода/вывода

Форматированный вывод с помощью функции printf()

Прототип функции *printf():*

```
int printf (const char * управляющая_строка, …).
```

Функция *printf()* возвращает число выведенных символов или отрицательное значение в случае ошибки.

Управляющая_строка состоит из элементов двух видов. Первый из них – это символы, которые предстоит вывести на экран; второй – это *спецификаторы преобразования*, которые определяют способ вывода стоящих за ним аргументов. Каждый такой спецификатор начинается со знака процента, за которым следует код формата. Аргументов должно быть ровно столько, сколько и спецификаторов, причем спецификаторы преобразования и аргументы должны попарно соответствовать друг другу в направлении слева направо. Например, в результате такого вызова *printf()*

```
printf (“Мне нравится язык %c %s”, ‘C’, “и к тому же очень сильно!”);
```

будет выведено

```
Мне нравится язык C и к тому же очень сильно!
```

В этом примере первому спецификатору преобразования (%c), соответствует символ ‘C’, а второму (%s) – строка “и к тому же очень сильно!”.

Форматный ввод с использованием функции scanf()

Гораздо более сложен для начинающих программистов ввод числовых данных, организуемый с помощью функции *scanf*, использующей так называемую форматную строку:

Пример:

```
#include <stdio.h>
void int main()
{
     int i;
     float f;
     double d;
     ..........
     scanf("%d %f %lf", &i, &f, &d);
}
```

Строка вводимых данных поступает со стандартного устройства ввода (*stdin*), которым по умолчанию считается клавиатура. Завершение набора строки ввода – нажатие клавиши *Enter*.

Первый аргумент функции *scanf* представляет собой форматную строку, управляющую процессом преобразования числовых данных, набранных пользователем в строке ввода, в машинный формат, соответствующий типам переменных, адреса которых указаны вслед за форматной строкой. Числовые значения в строке ввода рекомендуется разделять одним или несколькими пробелами.

В приведенном примере переменной i (в списке ввода указан ее адрес – &i), объявленной с помощью спецификатора типа *int*, соответствует форматный указатель %d. Это означает, что первым числовым значением в строке ввода может быть только целое десятичное число со знаком (d – от *decimal*, десятичный). Вещественной переменной f типа *float* в форматной строке соответствует указатель %f. Это означает, что второе числовое значение в строке ввода должно принадлежать диапазону, предусмотренному для коротких вещественных данных. Для переменной d типа *double* использован форматный указатель %lf (*l* – от *long*).

Как правило, количество форматных указателей, перечисленных в первом аргументе функции *scanf*, должно совпадать с количеством адресов переменных, следующих за форматной строкой. Исключение составляет случай, когда форматный указатель предписывает программе пропустить очередное значение из введенной строки. В этой ситуации количество адресов в списке ввода уменьшается соответствующим образом. Например:

```
scanf("%d %*l %lf", &i, &d);
```

При выполнении такого оператора ввода программа проигнорирует второе числовое значение, набранное пользователем. Конечно, при ручном наборе вводимых значений нелепо заставлять пользователя печатать данные, которые программе не понадобятся. Но такая возможность может оказаться полезной, когда строка ввода поступает не с клавиатуры, а из других источников (считана с диска, сформирована другой программой в оперативной памяти).

Вообще говоря, функция *scanf* возвращает числовое значение, равное количеству правильно обработанных полей из строки ввода. Это полезно помнить при организации проверки правильности ввода,

так как сообщения об ошибках ввода функция *scanf* не выдает, но после первой же ошибки прерывает свою работу.

Основная сложность в овладении тонкостями ввода, управляемого списком форматных указателей, заключается в многообразии последних. В самом общем виде числовой форматный указатель, используемый функцией *scanf*, представляется как следующая последовательность управляющих символов и дополнительных признаков:

```
%[*][ширина][{l|h|L}]{d|i|u|o|x|X|f|e|E|g|G}.
```

Квадратные скобки здесь означают, что соответствующий элемент форматного указателя может отсутствовать. В фигурных скобках указаны символы, один из которых может быть выбран. Обязательные элементы любого форматного указателя – начальный символ % и последний символ, определяющий тип вводимого значения.

Символ * после начального символа служит указанием о пропуске соответствующего значения из строки ввода. Необязательное и, как правило, не используемое при вводе поле «ширина» задает количество символов во вводимом значении. Дополнительные признаки l, h и L уточняют длину машинного формата соответствующей переменной (l, L – *long*; h – *short*). Значение последнего обязательного символа форматного указателя расшифровано в табл. 5.

Таблица 5. Значение спецификаторов в строке ввода/вывода

Код	**Допустимое значение в строке ввода**
%a	значение с плавающей точкой
%c	одиночный символ
%d	целое десятичное число со знаком
%i	целое число
%e	вещественное число
%E	вещественное число
%f	число с плавающей точкой
%g	вещественное число
%G	вещественное число
%o	целое восьмеричное число без знака
%s	строка
%u	целое число без знака
%x	целое шестнадцатеричное число без знака
%X	целое шестнадцатеричное число без знака
%p	указатель
%n	целое значение, равное количеству уже считанных символов
%%	знак процента

Потоковый ввод/вывод C++

Язык C++ предусматривает альтернативную обращениям к функциям *printf* и *scanf* возможность обработки ввода/вывода стандартных типов данных и строк. Например, простой диалог

```
printf("Enter new tag: ");
scanf("%d", &tag);
printf("The new tag is: %d\n", tag);
```

записывается на C++ в виде

```
cout << "Enter new tag: ";
cin >> tag;
cout << "The new tag is: " << tag << '\n';
```

Для организации потокового ввода/вывода программы на C++ в код нужно добавить строки в раздел описания библиотек #include <iostream> и перед объявлением функции main строку

```
using namespace std;
```

? Вопросы и задания

1. Можно ли использовать ключевые слова для именования объектов программы?
2. Перечислите основные элементы программы.
3. В чем отличия определения и объявления объектов программы?
4. Перечислите базовые типы данных языка и их характеристики.
5. Поясните принцип работы префиксной и постфиксной операций увеличения.
6. Каким образом можно, не используя операцию умножения, умножить число на 4.
7. Укажите порядок вычисления следующих выражений, задав полную скобочную структуру:

a = b + c * d << 2 & 8

a & 077 != 3

a == b || a == c && c < 5

c = x != 0

a = –1 + + b -- –5

a = b == c ++

a = b = c = 0

a-b, c=d

Тема 5. ОПЕРАТОРЫ ЯЗЫКА

1. *Пустой оператор (;)* используют, когда синтаксически должен быть оператор, а встраивать его для программиста нежелательно. Можно его помечать меткой «пустой оператор» (метка1: ;).

2. *Составной оператор { }* применяют, когда синтаксически имеет место быть хотя бы один оператор.

3. *Оператор безусловного перехода goto M* передает управление на строку кода, помеченную меткой *M*. Видимость оператора внутри функции – локальная, т. е. перейти на метку из другой функции (осуществить дальний переход) нельзя. С помощью метки можно войти в тело цикла, в операцию составного оператора.

4. *Оператор возврата*

```
return [<выражение>];
```

Оператор *return* завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом. Функция *main* передает управление операционной системе.

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие не обязательно. Если в какой-либо функции отсутствует оператор *return*, то передача управления в вызывающую функцию происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом *void*.

Таким образом, использование оператора *return* необходимо либо для немедленного выхода из функции, либо для передачи возвращаемого значения.

Пример:

```
int sum (int a, int b)
{
return a+b;
}
```

Функция *sum* имеет два формальных параметра *a* и *b* типа *int* и возвращает значение типа *int*, о чем говорит описатель, стоящий перед именем функции. Возвращаемое оператором *return* значение равно сумме фактических параметров.

Пример:

```
void prov (int a, double b)
{
     double c;
     if (a<3) return;
     else
     if (b>10) return;
     else
     {
          c = a + b;
          if ((2 * c - b) == 11) return;
     }
}
```

В этом примере оператор *return* используется для выхода из функции в случае выполнения одного из проверяемых условий. Оператор return в функции может использоваться несколько раз.

5. *Оператор-выражение*

Любое выражение, которое заканчивается точкой с запятой, представляет собой оператор. Выполнение данного оператора заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения чреваты побочными эффектами. Заметим, что вызвать функцию, не возвращающую значения, можно только при помощи оператора выражения. Правила вычисления выражений были сформулированы выше.

Пример:

```
++i; /* этот оператор представляет выражение, которое увеличивает значение переменной i на единицу. */
a=cos(b*5); /* этот оператор представляет выражение, включающее в себя операции присваивания и вызова функции. */
a(x,y); /* этот оператор представляет выражение, состоящее из вызова функции. */
```

6. *Оператор break* обеспечивает прекращение выполнения самого внутреннего из включающих его операторов *switch, do, for, while*. После выполнения оператора *break* управление передается оператору, следующему за прерванным.

7. *Оператор continue* – оператор продолжения. Встречается внутри тела цикла и означает переход на следующую итерацию цикла.

8. *Оператор if* – оператор условного перехода:

Формат оператора:

```
if (выражение) оператор1; [else оператор2;]
```

Выполнение *оператора if* начинается с вычисления выражения. Далее выполнение осуществляется по следующей схеме:

Если выражение истинно (т. е. отлично от нуля), то выполняется оператор1, иначе выполняется оператор2. Если выражение ложно и отсутствует оператор2, то выполняется следующий за if оператор.

После выполнения оператора *if* значение передается на следующий оператор программы, если последовательность выполнения операторов программы не будет принудительно нарушена использованием операторов перехода.

Пример:

```
if (i < j) i++; //если i<j, то увеличиваем i на единицу
else
{
     j = i-3;
     i++;
}
```

Этот пример иллюстрирует также и тот факт, что на месте *оператора1*, так же, как и на месте *оператора2*, могут находиться сложные конструкции.

Допускается использование вложенных операторов *if*. Оператор *if* может быть включен в конструкцию *if* или в конструкцию *else* другого оператора *if*. Чтобы программа лучше читалась, рекомендуется группировать операторы и конструкции во вложенных операторах *if*, используя фигурные скобки. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово *else* с наиболее близким *if*, для которого нет *else*.

Пример:

```
int main ( )
{
int t=2, b=7, r=3;
 if (t>b)
 {
      if (b < r) r=b;
 }
 else r=t;
 return 0;
}
```

В результате выполнения этой программы значение переменной r станет равным 2.

Если же в программе опустить фигурные скобки, стоящие после оператора *if*, то программа будет иметь следующий вид:

```
int main ( )
{
     int t = 2, b = 7, r = 3;
     if (t > b)
      if (b < r) r = b;
          else r = t;
     return 0;
}
```

В этом случае r останется равной 3, так как ключевое слово *else* относится ко второму оператору *if*, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе *if*.

Следующий фрагмент иллюстрирует вложенные операторы *if*:

```
char ZNAC;
int x, y, z;
:
if (ZNAC == '-') x = y - z;
else if (ZNAC == '+') x = y + z;
else if (ZNAC == '*') x = y * z;
      else if (ZNAC == '/') x = y / z;
           else ...
```

Из этого примера можно сделать вывод, что конструкции, использующие вложенные операторы *if*, довольно громоздки и не всегда достаточно надежны. Другой способ организации выбора из множества различных вариантов – это работа со специальным оператором выбора *switch*.

9. *Оператор-переключатель switch* предназначен для организации выбора из множества различных вариантов. Формат оператора следующий:

```
switch (<выражение>)
{[объявления]
 :
 [case константное-выражение1]:[операторы];
 [case константное-выражение2]:[операторы];
 :
 :
 [default:[операторы]];
}
```

Выражение, следующее за ключевым словом *switch* в круглых скобках, может быть любым выражением, допустимым в языке C++, значение которого должно быть целым. Отметим, что можно использовать явное приведение к целому типу.

Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора *switch* состоит из нескольких операторов, помеченных ключевым словом *case* с последующим *константным выражением*. Следует отметить, что использование целого константного выражения – существенный недостаток, присущий рассмотренному оператору.

Так как константное выражение вычисляется во время трансляции, оно не может содержать переменные или вызовы функций. Обычно в качестве константного выражения используют целые или символьные константы.

Все константные выражения в операторе *switch* должны быть уникальны. Кроме операторов, помеченных ключевым словом *case*, может быть (но обязательно один) фрагмент, помеченный ключевым словом *default*.

Допускают либо пустой список операторов, либо содержащий один или более операторов. Причем в операторе *switch* не требуется заключать последовательность операторов в фигурные скобки.

Отметим также, что в операторе *switch* можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом *case*, однако в объявлениях не следует применять инициализацию.

Схема выполнения оператора *switch*:

1) вычисляется выражение в круглых скобках;

2) вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами *case*;

3) если одно из константных выражений совпадает со значением выражения, то управление передается оператору, помеченному соответствующим ключевым словом *case*;

4) если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом *default*, а в случае его отсутствия – следующему оператору после *switch*.

Отметим интересную особенность использования оператора *switch*: конструкция со словом *default* может быть не последней в теле оператора *switch*. Ключевые слова *case* и *default* в теле оператора *switch* существенны только при первичной проверке, когда определяется начальная точка выполнения тела оператора *switch*. Все операторы между начальным оператором и концом тела выполняются вне зависимости от меток, если только какой-то из операторов не передаст управления из тела оператора *switch*. Таким образом, программист должен сам позаботиться о выходе из *case*, если это необходимо. Чаще всего для этого используют оператор *break*.

Для того чтобы выполнить одни и те же действия для различных значений выражения, можно пометить один и тот же оператор несколькими ключевыми словами *case*.

Пример:

```
int i = 2;
switch (i)
{
 case 1: i += 2;
 case 2: i *= 3;
 case 0: i /= 2;
 case 4: i -= 5;
}
```

Выполнение оператора *switch* начинается с оператора, помеченного *case* 2. Таким образом, переменная i получает значение, равное 6, далее выполняется оператор, помеченный ключевым словом *case* 0, а затем *case* 4, переменная i примет значение 3, а затем значение –2.

Рассмотрим ранее приведенный пример, в котором иллюстрировалось использование вложенных операторов *if*, переписанный теперь с использованием оператора *switch*.

Пример:

```
char ZNAC;
int x, y, z;
switch (ZNAC)
{
 case '+': x = y + z; break;
 case '-': x = y - z; break;
 case '*': x = y * z; break;
 case '/': x = u / z; break;
 default : ;
}
```

Использование оператора *break* позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора *switch* путем передачи управления оператору, следующему за *switch*.

Отметим, что в тело оператора *switch* можно включать вложенные операторы *switch*, при этом в ключевых словах *case* можно применять одинаковые константные выражения.

Пример:

```
switch (a)
{
  case 1: b = c; break;
  case 2:
     switch (d)
     {    case 0: f = s; break;
          case 1: f = 9; break;
          case 2: f -= 9; break;
     }
  case 3: b -= c; break;
  :
}
```

10. *Операторы циклов*

Существуют три типа циклов: циклы с предусловием, циклы с постусловием, с пошаговым исполнением.

1) Оператор цикла с пошаговым исполнением:

```
for (<инициализация>; <условие>; <приращение>) <оператор>;
```

Цикл *for* может иметь большое количество вариаций. *Инициализация* – это присваивание начального значения переменной, которая называется параметром цикла. *Условие* представляет собой условное выражение, определяющее, следует ли выполнять *оператор* цикла (тело цикла) в очередной раз. Оператор *приращение* осуществляет изменение параметра цикла при каждой итерации. Эти три оператора обязательно разделяются точкой с запятой. Цикл *for* выполняется, если выражение *условие* принимает значение ИСТИНА. Если оно хотя бы один раз принимает значение ЛОЖЬ, то программа выходит из цикла и выполняется оператор, следующий за телом цикла *for*.

Любое из указанных выражений может отсутствовать (пустые операторы).

Если отсутствует *условие*, то оно всегда равно ИСТИНА. Следовательно, в теле цикла должен выполниться оператор *break*, при его отсутствии цикл будет повторяться бесконечно.

Пример:

```
/* циклическое обнуление элементов массива */
for (int i=0; i<N; i++)
     a[i]=0;

int i=0; /*инициализация первого прохода цикла*/
for ( ; i<N; )
     a[i++]=0;/* i++ - переход к следующей итерации
цикла */

int i=0; /*инициализация первого прохода цикла*/
for ( ; ; ) //бесконечный цикл
{
     a[i++]=0; /* i++ - переход к следующей итерации
цикла */
     if (i=N) break; // условие выхода из цикла
}
```

Тело цикла бывает и пустым. Такую особенность цикла *for* можно использовать для упрощения некоторых программ, а также в циклах, предназначенных для того, чтобы отложить выполнение последней части программы на некоторое время.

```
for (printf ("нажать Enter" \n); printf ("Не клавиша
Enter"); getchar()!='\n') ;
```

Тело цикла может быть составным, т. е. содержать в качестве оператора цикл (вложенные циклы). Если во вложенном цикле его тело содержит break, это означает осуществление передачи управления за пределы вложенного цикла во внешний.

2) Оператор цикла с постусловием:

```
do
     <оператор> ;
while (<выражение>);
```

Выражение – условие выполнения цикла (в качестве условия может выступать любое логическое выражение).

Особенность цикла с постусловием: тело цикла выполняется один раз, затем происходит проверка условия. Если условие равно ИСТИНА, то выполняется тело цикла, иначе управление передается оператору, следующему за циклом.

Досрочный выход из цикла:

```
i = n; //вычисления происходят при n>0
do
{
  x=rand()%10; //генерируем случайное число от 0 до 9
  if (x==1) break; // если сгенерировано число 1, то прерываем работу цикла
printf (“%d\n”, x}; //все остальные числа будут выведены на экран
}
while (--i);
```

3) Оператор цикла с предусловием:

Оператор цикла *while* называется циклом с предусловием и имеет следующий формат:

```
while (<выражение>) <тело цикла>;
```

В качестве выражения допускается использовать любое выражение языка C++, а в качестве тела цикла – любой оператор, в том числе пустой или составной. Схема выполнения оператора *while* такова:

1. Вычисляется выражение.

2. Если выражение ложно, то выполнение оператора *while* заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора *while*.

3. Процесс повторяется с п. 1.

❓ Вопросы и задания

1. Перечислите циклические операторы: синтаксис и семантику.
2. Каково назначение блочного оператора?
3. Поясните принцип работы операторов break и continue.
4. Обязательна ли в операторе-переключателе метка default?
5. Следующий цикл for перепишите с помощью оператора while:

```
for (i=0; i<max_length; i++)
if (input_line[i] == '?') quest_count++;
```

Запишите цикл, используя в качестве его управляющей переменной указатель так, чтобы условие имело вид *p=='?'.

6. Переделайте пример из пункта «Оператор цикла с постусловием») для использования цикла *while*.

Тема 6. МАССИВЫ

6.1. Объявление, определение и работа с массивами

На основе базовых типов пользователь может конструировать составные типы, используя модификаторы типов [], () , *.

Массив – это последовательность элементов одинакового типа, плотно расположенных в памяти друг за другом. Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет два формата:

```
спецификатор_типа описатель [константное_выражение];
спецификатор_типа описатель [ ];
```

Описатель – это идентификатор массива. Описатель может быть простым идентификатором, если описатель, кроме идентификатора, включает в себя признаки других типов [], (), *, то тогда тип элемента складывается из оставшейся части описателя и спецификации типа.

Пример:

```
int *a[10]; /* массив указателей на тип int; приоритет у скобок */
int (*a)[10]; /* указатель на массив из 10 int */
```

Спецификатор_типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа *void*.

Константное_выражение в квадратных скобках задает количество элементов массива.

Пример:

```
int a[10]; /* объявление целочисленного массива на 10 элементов */
int a[10] = {0}; /* объявление целочисленного массива на 10 элементов и инициализация каждого элемента нулевым значением */
```

Константное_выражение при объявлении массива может быть опущено в следующих случаях:

1) при объявлении массив инициализируется:

```
int a[]={1,2,3}; /* место в памяти отводится под 3 элемента целого типа*/
```

2) массив объявлен как формальный параметр функции;

3) массив объявлен как ссылка на массив, явно определенный в другом файле.

Если инициализация массива совмещается с объявлением, то элементов может быть меньше, чем в квадратных скобках. В этом случае место отведётся для количества элементов, указанных в квадратных скобках, а проинициализированы будут от начала массива столько элементов, сколько реально указано в списке, остальные обнулятся.

Способ обращения к элементу массива:

```
<имя массива> [номер элемента]
```

Пример:

```
a[1]; //обращение к первому элементу массива
```

В C++ ни на стадии компиляции, ни на шаге RUN не отслеживается факт выхода за пределы массива.

В языке C++ определены только одномерные массивы, но поскольку элементом массива может быть массив, можно определить и многомерные массивы. Они формализуются списком константных_выражений, следующих за идентификатором массива, причем каждое константное_выражение заключается в свои квадратные скобки.

Каждое константное_выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных_выражения, трехмерного – три и т. д. Отметим, что в языке C++ первый элемент массива имеет индекс, равный ***нулю***.

Примеры:

```
int a[2][3]; /* представлено в виде матрицы
 a[0][0] a[0][1] a[0][2]
 a[1][0] a[1][1] a[1][2] */
double b[10]; /* массив из 10 элементов */
int w[3][3] = { {2, 3, 4}, {3, 4, 8}, {1, 0, 9 } };
```

В последнем примере объявлен массив w[3][3]. Списки, выделенные фигурными скобками, соответствуют строкам массива, в случае отсутствия скобок инициализация будет выполнена неправильно.

В языке C++ можно использовать сечения массива, как и в других языках высокого уровня, однако на применение сечений накладывается ряд ограничений. Сечения формируются вследствие опускания одной или нескольких пар квадратных скобок. Пары квадратных скобок можно отбрасывать только справа налево и строго последовательно. С сечениями массивов имеют дело при организации вычислительного процесса в функциях языка C++, разрабатываемых пользователем.

Пример:

```
int s[2][3];
```

Если при обращении к некоторой функции написать s[0], то будет передаваться нулевая строка массива *s*.

Пример:

```
int b[2][3][4];
```

При обращении к массиву *b* можно написать, например, b[1][2] и будет передаваться вектор из четырех элементов, а обращение b[1] даст двухмерный массив размером 3 на 4. Нельзя написать b[2][4], подразумевая, что передаваться будет вектор, потому что это не соответствует ограничению, наложенному на использование сечений массива.

Пример объявления символьного массива:

```
char str[] = "объявление символьного массива";
```

Следует учитывать, что в символьном литерале находится на один элемент больше, так как последний из элементов представляет собой управляющую последовательность '\0'.

6.2. Массив с точки зрения языка C++ (на виртуальном уровне) и с точки зрения хранения

Стандарт языка гарантирует плотное расположение элементов массива в памяти компьютера (рис. 6.1). Имя массива трактуется как константный указатель на элементы массива (рис. 6.2). Имя массива не может переопределяться. При этом $a \equiv \&a$.

```
int a[5];
```

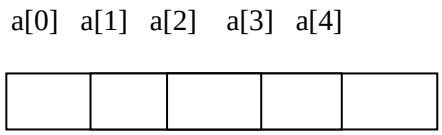


Рис. 6.1. Размещение массива в памяти

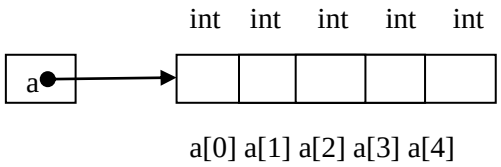


Рис. 6.2. Массив с точки зрения языка C++

Теперь рассмотрим двумерный массив с двух точек зрения.

Двумерный массив вытягивается в линию при записи в память компьютера: вначале идут элементы первой строки, затем со следующей ячейки записывается вторая строка и т. д. (рис. 6.3).

С точки зрения языка C++ двумерный массив можно интерпретировать двумя способами:

1) указатель на указатель на *int*;

2) массив указателей на *int*.

```
int b[2][3]; /* массив содержит 2 строки по 3 элемен-
та в каждой */
```

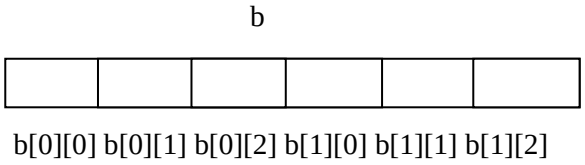


Рис. 6.3. Размещение двумерного массива в памяти

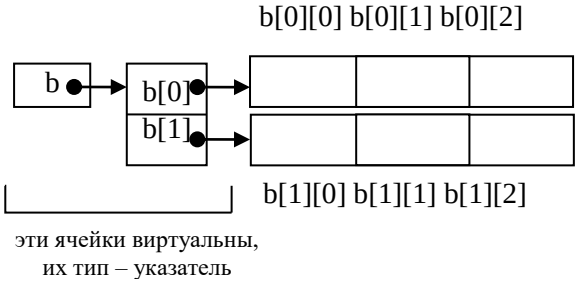


Рис. 6.4. Двумерный массив с точки зрения языка C++

6.3. Примеры работы с элементами массива

К элементу массива можно обращаться с помощью индексного выражения

```
<указатель> [<выражение типа int>].
```

Пример:

```
a[1].
```

В C++ обязательное требование при таком обращении – наличие в этом обращении одного указателя и одного целого числа. Местоположение в этом объявлении роли не играет.

Пример:

```
a[i] ≡ i[a].
```

В этом случае происходит следующее:

1. Вычисляют индексное выражение, к указателю будет прибавляться смещение типа «умножение» на длину специфицированного типа.

2. Далее выполняют косвенную адресацию, т. е. вычисленное значение воспринимается как адрес.

```
(a+1)[0] ≡ a[1];
(a-1)[2] ≡ a[1];
```

```
a[2] ≡ *(a+2); // 2*sizeof(int)=4 - размер сдвига
```

Обращение к элементу двумерного массива:

```
b[2][3]= {{1,2,3},{4,5,6}};
```

Запись индексного выражения с использованием косвенной адресации:

```
b[1][2] ≡ *(*(b+1) +2);
```

Общий принцип адресации:

```
b[i][j] = *(*(b+ i*<количество_элементов_строки>)+ +
j*sizeof(тип_элемента)).
```

? Вопросы и задания

1. Дайте определение массива с точки зрения хранения.
2. Приведите общий принцип обращения к элементам массива.
3. С какого числа начинается нумерация элементов массива? Можно ли нумерацию изменить?
4. Каким образом определить размер массива?
5. Объявите трехмерный массив и изобразите его с точки зрения языка C++.
6. Объявите двухмерный массив вещественных элементов размером 5 строк и 4 столбца. Далее:
   a) проинициализируйте при объявлении;
   b) заполните массив с клавиатуры;
   c) заполните массив случайными числами.
7. Посчитайте сумму элементов массива.
8. Может ли элементом массива быть массив?

Тема 7. СТРОКИ И ОПЕРАЦИИ СО СТРОКАМИ

7.1. Объявление строк в программах

Программисты на C++ широко используют символьные строки для хранения имен пользователей, имен файлов и другой символьной информации.

Для объявления символьной строки внутри программы просто объявите массив типа *char* с количеством элементов, достаточным для хранения требуемых символов. Например, следующее объявление создает переменную символьной строки с *именем filename,* способную хранить 64 символа:

```
char filename[64];
```

Это объявление создает массив с элементами, индексируемыми от *filename[0]* до *filename[63]* (рис. 7.1).

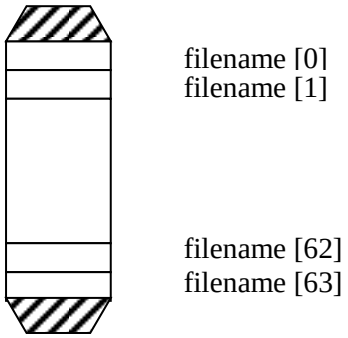


Рис. 7.1. Трактовка символьной строки как массива типа char

Главное различие между символьными строками и другими типами массивов заключается в том, как C++ указывает последний элемент массива. Программы на C++ представляют конец символьной строки с помощью символа NULL, который в C++ изображается как специальный символ '\0'. Когда вы присваиваете символы символьной строке, вы должны поместить символ NULL ('\0') после последнего символа в строке. Например, следующая программа присваивает буквы от А до Я переменной *alphabet*, используя цикл *for*. Затем программа добавляет символ NULL в эту переменную и выводит ее с помощью *cout.*

Пример:

```
#include <iostream>
void main(void)
{
      char alphabet [34]; // 33 буквы плюс NULL char
      letter;
     int index;
     for (letter = 'A', index = 0; letter <= 'Я'; let-
 ter++, index++)
          alphabet[index] = letter;
     alphabet[index] = NULL;
     cout << "Буквы " << alphabet;
}
```

Программа присваивает строке символ NULL, чтобы указать последний символ строки:

```
alphabet[index] = NULL;
```

Когда выходной поток *cout* выводит символьную строку, он по одному печатает символы строки, пока не встретит символ NULL.

Цикл инициализирует и увеличивает две переменные *(letter* и *index).* Когда цикл *for* инициализирует или увеличивает несколько переменных, разделяйте операции запятой (запятая тоже является оператором C++):

```
     for (letter = 'A', index = 0; letter <= 'Я'; let-
ter++, index++)
```

Все созданные ранее программы использовали символьные строковые константы, заключенные внутри двойных кавычек, как показано ниже:

```
"Это строковая константа".
```

При создании символьной строковой константы компилятор C++ автоматически добавляет символ NULL, как показано на рис. 7.2.

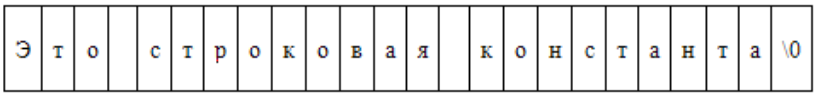


Рис. 7.2. Автоматическое добавление символа NULL
к строковым константам

Когда в программе вывод строковых констант осуществляется с помощью выходного потока *cout*, *cout* использует символ NULL (который компилятор добавляет к строке автоматически) для определения последнего символа вывода.

7.2. Инициализация символьной строки

C++ позволяет инициализировать массивы при объявлении. Символьные строки C++ не считаются исключением. Для инициализации символьной строки при объявлении необходимо указать требуемую строку внутри двойных кавычек, как показано ниже:

```
char title[64] = "Учимся программировать на языке Си";
```

Если количество символов, присваиваемое строке, меньше размера массива, большинство компиляторов C++ будут присваивать символы NULL остающимся элементам строкового массива. Как и в случае с массивами других типов, если вы не указываете размер массива, который инициализируете при объявлении, компилятор C++ распределит достаточно памяти для размещения указанных букв и символа NULL:

```
char title[] = "Учимся программировать на языке Си";
```

7.3. Функции работы со строками

Для использования специальных функций работы со строками необходимо подключить заголовочный файл *string.h*.

Приведем описание основных функций:

```
int strlen(const char *s) //Возвращает длину строки s.
```

```
char *strchr(const char *s, char c) //Отыскивает первое вхождение в строку s символа c. Возвращает указатель на символ c в строке s. В противном случае возвращает нулевой указатель.

char *strcpy(char * to, const char *from) //Копирует строку, на которую указывает from, в строку, на которую указывает to. Область, на которую указывает to, должна иметь достаточный размер для размещения копируемой строки.

char *strncpy(char *to, const char *from, int limit) //Аналогично предыдущей, но копирует не более, чем limit символов.

int strcmp(const char *s1, const char *s2) // Функция сравнения двух строк. Возвращает –1, 0, или +1, если s1 соответственно меньше, равна или больше s2.

int strncmp(const char *s1, const char *s2, size_t m); // Функция сравнения строк по первым m символам. Результат работы функции аналогичен strcmp.

int strcmpi(const char *s1, const char *s2); //Функция сравнения строк без учета регистра.

int strncmpi(const char *s1, const char *s2, size_t n); // Функция сравнения строк по первым m символам без учета регистра.

char *strcat(char *s1, const char *s2); // Функция конкатенации строк добавляет строку s2 в конец строки s1.
```

7.4. Работа со строками через класс string

В современном стандарте C++ определен класс с функциями и свойствами (переменными) для организации работы со строками:

```
#include <string>
```

Для работы со строками также нужно подключить стандартное пространство имен

```
using namespace std;
```

В противном случае придётся везде указывать описатель класса *std::string* вместо *string*.

Ниже следует пример программы, работающей со *string*:

```
#include <iostream>
#include <string>
```

```
#include <malloc.h>
using namespace std;

int main () {
 string s = "Test";
 s.insert (1,"!");
 cout << s.c_str() << endl;
 string *s2 = new string("Hello");
 s2->erase(s2->end());
 cout << s2->c_str();
 cin.get(); return 0;
}
```

Инициализация строк при описании и длина строки (не включая завершающий нуль-терминатор):

```
string st( "Моя строка\n" );
cout << "Длина " << st << ": " << st.size()
     << " символов, включая символ новой строки\n";
```

Строка может быть задана и пустой:

```
string st2;
```

Для проверки того, пуста ли строка, можно сравнить ее длину с 0:

```
if ( ! st.size() ) // пустая
```

или применить метод empty(), возвращающий true для пустой строки и false для непустой:

```
if ( st.empty() ) // пустая
```

Третья форма создания строки инициализирует объект типа *string* другим объектом того же типа:

```
string st3( st );
```

Строка *st3* инициализируется строкой *st*. Как мы можем убедиться, что эти строки совпадают? Воспользуемся оператором сравнения (==):

```
if ( st == st3 ) // инициализация сработала
```

Как скопировать одну строку в другую? С помощью обычной операции присваивания:

```
st2 = st3; // копируем st3 в st2
```

Для сцепления строк используют операцию сложения (+) или операцию сложения с присваиванием (+=). Пусть даны две строки

```
string s1( "hello, " );
string s2( "world\n" );
```

Мы можем получить третью строку, состоящую из конкатенации первых двух, таким образом

```
string s3 = s1 + s2;
```

Если же мы хотим добавить s2 в конец s1, мы должны написать

```
s1 += s2;
```

Операция сложения может сцеплять объекты класса *string* не только между собой, но и со строками встроенного типа. Можно переписать пример, приведенный выше, так, чтобы специальные символы и знаки препинания представлялись встроенным типом *char* *, а значимые слова – объектами класса *string*:

```
const char *pc = ", ";
string s1( "hello" ), s2( "world" );
string s3 = s1 + pc + s2 + "\n";
cout << endl << s3;
```

Подобные выражения работают потому, что компилятор «знает», как автоматически преобразовывать объекты встроенного типа в объекты класса *string*. Возможно и простое присваивание встроенной строки объекту *string*:

```
string s1;
const char *pc = "a character array";
s1 = pc; // правильно
```

Обратное преобразование при этом не работает. Попытка выполнить следующую инициализацию строки встроенного типа вызовет ошибку компиляции:

```
char *str = s1; // ошибка компиляции
```

Чтобы осуществить такое преобразование, необходимо явно вызвать функцию-член с названием *c_str*() ("строка Си"):

```
const char *str = s1.c_str();
```

Функция *c_str*() возвращает указатель на символьный массив, содержащий строку объекта *string* в том виде, в каком она находилась бы во встроенном строковом типе. Ключевое слово *const* здесь предотвращает опасную в современных визуальных средах возможность непосредственной модификации содержимого объекта через указатель.

К отдельным символам объекта типа *string*, как и встроенного типа, можно обращаться с помощью операции взятия индекса. Вот, например, фрагмент кода, заменяющего все точки символами подчеркивания:

```
string str( "www.disney.com" );
int size = str.size();
for ( int i = 0; i < size; i++ )
 if ( str[i] == '.' ) str[ i ] = '_';
cout << str;
```

Перечень функций-членов класса string

Задание символов в строке	
operator=	присваивает значения строке
assign	назначает символы строке
Доступ к отдельным символам	
at	получение указанного символа с проверкой выхода индекса за границы
operator[]	получение указанного символа
front	получение первого символа
back	получение последнего символа
data	возвращает указатель на первый символ строки
c_str	возвращает немодифицируемый массив символов, содержащий символы строки
Проверка на вместимость строки	
empty	проверяет, является ли строка пустой
size length	возвращает количество символов в строке
max_size	возвращает максимальное количество символов
reserve	резервирует место под хранение
Операции над строкой	
clear	очищает содержимое строки
insert	вставка символов
erase	удаление символов
push_back	добавление символа в конец строки
pop_back	удаляет последний символ
append	добавляет символы в конец строки
operator+=	добавляет символы в конец строки
compare	сравнивает две строки
replace	заменяет каждое вхождение указанного символа
substr	возвращает подстроку
copy	копирует символы
resize	изменяет количество хранимых символов
swap	обменивает содержимое
Поиск в строке	
find	поиск символов в строке
rfind	поиск последнего вхождения подстроки
find_first_of	поиск первого вхождения символов
find_first_not_of	найти первое вхождение отсутствия символов
find_last_of	найти последнее вхождение символов
find_last_not_of	найти последнее вхождение отсутствия символов

Следует обратить внимание, что у любой функции класса *string* может быть несколько *перегрузок* – разновидностей с одинаковыми именами, отличающихся между собой списками и типами аргументов.

В качестве недостатков класса *string* можно отметить следующие:

1) отсутствие в классе встроенных средств для разбора строк по набору разделителей (аналога функции *strtok* для строк *char* *);

2) возможное замедление быстродействия по отношению к *char* * при сложной обработке данных.

? Вопросы и задания

1. Что такое строка с точки зрения языка C++?
2. В чем отличие строки от массива символов?
3. Можно ли со строкой работать поэлементно?
4. Напишите программу, вычисляющую сумму цифр в строке вида “1ab3c405”. Ввод строки организуйте с клавиатуры.
5. Напишите программу, удаляющую все цифры из символьной строки.
6. Напишите фрагмент кода, осуществляющий смену двух строк str1 и str2, если они одинаковой длины.

Тема 8. УКАЗАТЕЛИ И ССЫЛКИ. РАБОТА С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

Когда компилятор обрабатывает оператор определения переменной, например *int i = 10*, он выделяет память в соответствии с типом (*int*) и инициализирует ее указанным значением (10). Все обращения в программе к переменной по ее имени (i) заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Можно определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями.

Так как указатель – это адрес некоторого объекта, то через него можно обращаться к этому объекту.

Пример:

```
    y=&x /* переменной y присваивается адрес переменной x,
указывающий, где в оперативной памяти находится значение x */
    z=*y /* переменной z присваивается значение перемен-
ной, записанной по адресу y, если y=&x и  z=*y, то z=x */
```

В C++ различают три вида указателей:

1) указатели на объект;
2) функцию;
3) *void*.

Эти указатели отличаются свойствами и набором допустимых операций. Указатель, имеющий тип, соответствующий типу данных, хранящемуся в указателе – типизированный указатель. Указатель на *void* – нетипизированный.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа (основного или составного). Простейшее объявление указат*еля на объект* имеет вид

```
тип *имя;
```

где тип может быть любым: как базовым, так и пользовательским.

Модификатор «звездочка» при объявлении указателя относится непосредственно к имени переменной, поэтому для того чтобы объявить несколько указателей, требуется ставить ее перед именем каждого из них. Например, в операторе

```
int *a, b, *c;
```

описываются два указателя на целое с именами *a* и *c*, а также целая переменная *b*.

Указатель на void применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Указателю на *void* можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать тип.

8.1. Инициализация указателей

Указатели чаще всего используют при работе с динамической памятью, называемой *кучей*. Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти, называемым *динамическими переменными,* производится только через указатели. Время жизни динамических переменных – от точки создания до конца программы или до явного освобождения памяти. В C++ существуют два способа работы с динамической памятью. Первый

использует семейство функций *malloc* и достался в наследство от Си, второй применяет операции *new* и *delete*.

При определении указателя надо стремиться выполнить его инициализацию, т. е. присвоение начального значения. Непреднамеренное использование неинициализированных указателей – распространенный источник ошибок в программах. Инициализатор записывается после имени указателя либо в круглых скобках, либо после знака равенства.

Существуют следующие способы инициализации указателя:

1. Присваивание указателю адреса существующего объекта:

a) с помощью операции получения адреса;

```
int a = 5; // целая переменная
int* p = &a; // в указатель записывается адрес a
int* p (&a); // то же самое другим способом.
```

b) с помощью значения другого инициализированного указателя:

```
int* u = p;
```

2. Присваивание указателю адреса области памяти в явном виде:

```
char* vp = (char *)OxB8000000;
```

Здесь 0xB8000000 – шестнадцатеричная константа, (*char* *) – операция приведения типа: константа преобразуется к типу «указатель на char».

3. Присваивание пустого значения:

```
int* suxx = NULL;
int* rulez = 0;
```

Пустой указатель можно использовать для проверки того, ссылается указатель на конкретный объект или нет.

4. Выделение участка динамической памяти и присваивание ее адреса указателю:

a) с помощью операции new:

```
int* n = new int; // 1
int* m = new int (10); // 2
int* q = new int[10];  // 3
```

b) с помощью функции *malloc*, которая находится в библиотеке *malloc.h*:

```
int* u = (int *)malloc(sizeof(int)); // 4
```

В операторе 1 операция *new* выполняет выделение достаточного для размещения величины типа *int* участка динамической памяти и записывает адрес начала этого участка в переменную *n*. Память под саму переменную *n* (размера, достаточного для размещения указателя) выделяется на этапе компиляции.

В операторе 2, кроме описанных выше действий, производится инициализация выделенной динамической памяти значением 10.

В операторе 3 операция *new* выполняет выделение памяти под 10 величин типа *int* (массива из 10 элементов) и записывает адрес начала этого участка в переменную *q*, которая может трактоваться как имя массива.

В операторе 4 делается то же самое, что и в операторе 1, но с помощью функции выделения памяти *malloc*, унаследованной из библиотеки Си. В функцию передается один параметр – количество выделяемой памяти в байтах. Конструкция (*int**) используется для приведения типа указателя, возвращаемого функцией, к требуемому типу. Если память выделить не удалось, функция возвращает NULL.

Операцию *new* использовать предпочтительнее, чем функцию *malloc*, особенно при работе с объектами.

Освобождение памяти, выделенной с помощью операции *new*, должно выполняться с помощью *delete*, а памяти, выделенной функцией *malloc*, – посредством функции *free*. При этом переменная-указатель сохраняется и может инициализироваться повторно. Приведенные выше динамические переменные уничтожаются следующим образом:

```
delete n;
delete m;
delete [] q;
free (u);
```

Если память выделялась с помощью *new[]*, для ее освобождения необходимо применять *delete[]*. Размерность массива при этом не указывается. Если квадратных скобок нет, то никакого сообщения об ошибке не выдается, но помечен как свободный будет только первый элемент массива, а остальные окажутся недоступны для дальнейших операций. Такие ячейки памяти называют «мусором».

Если переменная-указатель выходит из области своего действия, отведенная под нее память освобождается. Следовательно, динамическая переменная, на которую ссылался указатель, становится недоступной. При этом память из-под самой динамической переменной не освобождается. Другой случай появления «мусора» – когда инициализированному указателю присваивается значение другого указателя. При этом старое значение бесследно теряется.

8.2. Операции с указателями

С указателями можно производить следующие операции: присваивания и арифметические отношения.

Операция присваивания для указателей аналогична такой же для других типов данных. В отличие от работы с переменными при операции присваивания достаточно, чтобы оба операнда были типа «указатель», но типы этих указателей могут быть различными.

Пример:

```
int a;
int *p=&a, *p1;
*p = 8 /* значение 8 заносится по адресу p, т. е. значение переменой a=8*/
p1 = p /*переменной p1=p, т. е. адрес переменной a*/
```

Допустимые арифметические операции: +, – (соответственно +=, –=), ++, – –.

Арифметические действия с указателями учитывают тип переменной, на которую они указывают. Эти операции изменяют адрес указателя на величину пропорциональную размеру типа данных.

Пример:

```
float f,*pf; /* указатель на вещественное число, в памяти вещественное число занимает 4 байта */
pf=&f; /* например, переменная f хранится по адресу 200, тогда pf = 200 */
pf--; /* pf=196*/
pf+=4; /* pf=196+4*4=212*/
```

Арифметические операции с указателями (сложение с константой, вычитание, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, последовательно размещенными в памяти, например, с массивами. *Инкремент* перемещает указатель к следующему элементу массива, *декремент* – к предыдущему. Фактически значение указателя изменяется на величину *sizeof(mun)*. Если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа.

Выражение *(*p)++* инкрементирует значение, на которое ссылается указатель, при этом запись **p* означает доступ к значению объекта, на который настроен указатель *p*, и читается как «по указателю».

Пример:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char *s, s1[10];
    scanf("%s", s1);
    s=s1; /*настраиваем указатель на начало массива s1*/
    printf("%c \n", *s);
    s+=4; /*настраиваем указатель на 4 элемент массива*/
    printf("%c", *s); /* выводим значение по указателю*/
    s++; //настраиваем указатель на 5 элемент
    printf("%c", *s); /* выводим значение по указателю*/
    s++; //настраиваем указатель на 6 элемент
    printf("%c", *s); /* выводим значение по указателю*/
    getch();
}
```

Допустимые операции отношений для указателей: ==, >=, <=, >,<,!=.

Пример:

```
p1 == p2 /* результат операции ИСТИНА, если p1 и p2 указывают на один и тот же объект в оперативной памяти */
p1 >= p2 /* результат операции ИСТИНА, если переменная, на которую указывает p1, расположена в памяти правее (т. е. с большим адресом), чем p2 */
```

Унарная операция получения адреса & применима к величинам, имеющим имя и размещенным в оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной. Примеры операции приводились выше.

Указатель типа *void** используется для доступа к любым типам данных. Указатель типа *void* указывает на место в оперативной памяти и не содержит информации о типе объекта, т. е. он указывает не на сам объект, а на его возможное расположение. К указателю типа *void* не применяются арифметические операции.

8.3. Ссылки

Ссылка позволяет создать псевдоним (или второе имя) для переменных в вашей программе. Для объявления ссылки внутри программы укажите знак амперсанда (&) непосредственно после типа параметра. Объявляя ссылку, вы должны сразу же присвоить ей переменную, для которой эта ссылка будет псевдонимом, как показано ниже:

```
int& alias_name = variable; //объявление ссылки
```

После объявления ссылки ваша программа может применить или переменную, или ссылку:

```
alias_name = 1001;
variable = 1001;
```

Использование ссылок значительно упрощает процесс изменения параметров внутри функции.

После своей инициализации ссылку нельзя изменить на ссылку на другой объект.

Для создания ссылки на постоянный объект необходимо выполнить следующие действия:

```
const double d = 10.5;
const double& rcd=d; /*объявление ссылки на неизменяемую переменную d */
```

? Вопросы и задания

1. Что такое указатель?
2. Перечислите проблемы, которые могут возникнуть при работе с неинициализированными указателями.
3. Каково назначение нетипизированного указателя? Каким образом можно объявить нетипизированный указатель?
4. Будет ли корректно работать следующий код:

```
int a = 5;
int *pf = &a;
float *p;
p = pf;
```

Если код работает некорректно, то внесите исправления.

5. Объявить массив из трех указателей на вещественные переменные. Задайте значения переменных через указатели.
6. Разместить в динамической памяти одномерный массив, двумерный массив.
7. Поясните, что объявлено, проинициализируйте все объявленные переменные и нарисуйте картинки в памяти и с точки зрения языка Си.

```
int (*pM)[3];
int *(*pMM)[2];
int m[2][3];
```

8. Напишите фрагмент программы, используя оператор выделения динамической памяти *new*. В программе должен выполняться захват памяти для пяти символов, ввод строки с клавиатуры и освобождение захваченной памяти.
9. Что такое ссылка?
10. Объявите ссылку на константу.

Тема 9. СТРУКТУРЫ

Структура – это пользовательский тип данных, включающий поля данных различных типов. В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов. В языке С++ структура относится к виду класса и обладает всеми его свойствами, но во многих случаях достаточно использовать структуры так, как они определены в языке Си:

```
struct [ имя_типа ]
{
     тип_1 элемент_1;
     тип_2 элемент_2;
     ...
     тип_n элемент_n;
} [ список_описателей ];
```

Элементы структуры называются *полями структуры* и могут иметь любой тип, кроме типа этой же структуры, но могут служить указателями на него. Если отсутствует имя типа, должен быть отмечен список описателей переменных, указателей или массивов. В этом случае описание структуры представляет собой определение элементов этого списка.

Пример:

```
struct {
     int year;
     char moth;
     int day;
} date, date2;
```

Если список отсутствует, описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами.

Пример:

```
struct student{ // описание нового типа student
     char fio[30];
     long int num_zac;
     double sr_bal;
}; // описание заканчивается точкой с запятой

student gr[30], *p; /* определение массива типа student и указателя на тип student */
```

Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания.

Пример:

```
struct {
     char fio[30];
     long int num_zac;
     double sr_bal;
}    student1 = {"Necto", 011999,3.66};
```

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив – это массив массивов).

Пример:

```
struct complex{
     float real, im;
} compl [2][3] = {{{1, 1}, {1, 1}, {1, 1}},{{2, 2},
{2, 2}, {2, 2}}};
```

Для переменных одного и того же структурного типа определена операция присваивания, при этом происходит поэлементное копирование. Структуру можно передавать в функцию и возвращать в качестве значения функции. Размер структуры необязательно равен сумме размеров ее элементов, поскольку они могут быть выровнены по границам слова.

Доступ к полям структуры выполняется с помощью операций выбора “.” (точка) при обращении к полю через имя структуры и операции –> при обращении через указатель.

Пример:

```
student student1, gr[30], *p;
student.fio = "Иванов И.И.":
gr[8].sr_bal=5;
p->num_zac = 012001;
```

Если элементом структуры предстает другая структура (вложенные структуры), то доступ к ее элементам выполняется через две операции выбора.

Пример:

```
struct A {
     int a;
     double x;
};
```

```
struct B {
     A a;
     double x,
} x[2];
x[0].a.a = 1;
x[1].x = 0.1;
```

Из примера видно: поля разных структур могут иметь одинаковые имена, поскольку у них разная область видимости. Более того, можно объявлять в одной области видимости структуру и другой объект (например, переменную или массив) с одинаковыми именами, если при определении структурной переменной использовать слово *struct*.

Задание:

Описать структуру с именем avto, содержащую поля:

1) fam – фамилия и инициалы владельца автомобиля;
2) marka – марка автомобиля;
3) nomer – номер автомобиля.

Написать программу, выполняющую следующие действия:

1) ввод с клавиатуры данных в массив gai, состоящий из шести элементов типа avto; записи должны быть упорядочены в алфавитном порядке по фамилиям и именам владельцев;
2) вывод на экран информации о владельцах автомобиля, марка которого вводится с клавиатуры. Если таковых автомобилей нет, то на экран дисплея вывести соответствующее сообщение.

Программа решения задачи будет иметь вид

```
#include "stdafx.h"
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include <iostream>
using namespace std;

struct avto { //объявление структурного типа данных
 char fam[25];
 char marka[20];
 char nomer[10];
};
```

```
    //объявляем функцию вывода базы данных автомобилей на
экран
    void output_gai(const avto *,int );
    void main(void)
    {
     setlocale(LC_ALL,"Russian");
     int i,n;
     bool flag;
     cout<<"Введите количество записей: ";
     cin>>n;
     avto *gai = new avto[n], temp;
     char m[20];
     for(i=0;i<n;i++)
     { // Ввод данных
          cout<<"Введите фамилию владельца "<<i+1<<"-го ав-
томобиля: ";
          _flushall();//очистка буфера ввода
          cin.getline(gai[i].fam,255);  //считываем фамилию
с клавиатуры
          cout<<"Введите марку "<<i+1<<"-го автомобиля: ";
          cin.getline(gai[i].marka,255);//считываем   марку
авто с клавиатуры
          cout<<"Введите номер "<<i+1<<"-го автомобиля: ";
          cin >> gai[i].nomer; //считываем номер автомобиля
с клавиатуры
      }
     cout << "Исходные данные до сортировки" << endl;
     output_gai(gai,n); //вывод на экран базы данных авто-
мобилей
     // Сортировка
     flag = true;
     while(flag)
     {
          flag = false;
          for(i=0;i<n-1;i++)
          if(strcmp(gai[i].fam,gai[i+1].fam)>0)
          {
               temp=gai[i];
               gai[i]=gai[i+1];
               gai[i+1]=temp;
               flag = true;
          }
     }
```

```
 cout << endl<<"После сортировки"<<endl;
 output_gai(gai,n);
 cout << endl << "Введите марку автомобиля: ";
 cin >> m;
 cout << endl << "Искомые автомобили" << endl;
 flag = true;
 for(i=0;i<n;i++)
 if(!strcmp(m,gai[i].marka))
 {
      output_gai(&gai[i],1);
      flag = false;
 }
 if(flag)
 {
      cout<<"\nМарки "<<m;
      cout<<"нет в списках";
 }
 delete [] gai;
 system("pause");
}
void output_gai(const avto *gai, int n)
{
 int i;
 for(i=0;i<n;i++)
 {
      cout.setf(ios::left);
      cout.width(15);
      cout<<gai[i].fam;// *(gai+i)->fam;
      cout.width(15);
      cout<<gai[i].marka;
      cout.setf(ios::right);
      cout.width(5);
      cout<<gai[i].nomer<<endl;
 }
}
```

? Вопросы и задания

1. Назовите отличия структуры от массива.

2. Назовите операции доступа к полям структуры по указателю и через объект.

Тема 10. РАБОТА С ФУНКЦИЯМИ

Функции – это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Все переменные, объявленные в определениях функций, являются *локальными* – они известны только той функции, в которой определены. Большинство функций имеют список *параметров*, которые позволяют функциям обмениваться информацией. Параметры функции – это также локальные переменные.

В программах, содержащих большое число функций, *main* должна быть реализована как группа обращений к функциям, выполняющим основную часть работы.

Существует несколько оснований для разбиения программы на функции. Подход «разделяй и властвуй» делает разработку программы более контролируемой. Другой мотив – это *повторное использование кода,* т. е. применение однажды написанных функций в качестве конструктивных блоков для создания новых программ. Многократное использование кода – это основной определяющий фактор при переходе к объектно-ориентированному программированию. Имея хорошо продуманные имена и определения функций, можно создавать программы из стандартизованных модулей, не создавая специализированного программного кода. Эта методика известна как *абстракция.* Мы прибегаем к абстракции всякий раз, когда пишем программу, вызывающую функции стандартной библиотеки, например, *printf, scanf* и *pow*. Третья причина – правило избегания дублирования кода в программе. Оформление кода в виде функции позволяет выполнять его в разных частях программы посредством простого вызова функции.

Каждая функция должна ограничиваться выполнением одной, точно определенной задачи, а имя функции – отражать смысл данной задачи. Это облегчает абстракцию и способствует многократному использованию программного кода.

10.1. Прототип

Прототип функции сообщает компилятору тип данных, возвращаемых функцией, число параметров, получаемых функцией, тип и порядок следования параметров. Компилятор использует прототипы функций для проверки корректности обращений к функции. Предыдущие версии Си не выполняли этот вид проверки, так что существовала возможность неправильного вызова функции, при котором компилятор не регистрировал ошибки. Такие обращения могли приводить к некорректному завершению выполнения программы или исключительным ситуациям, которые, не вызывая краха программы, приводили тем не менее к трудно обнаруживаемым логическим ошибкам.

Прототип функции в общем виде:

```
тип_возвращаемого_значения имя_функции (список абстрактных типов);
```

Пример объявления прототипа функции, вычисляющей максимальное значение из трёх целочисленных:

```
int maximum (int, int, int);
```

Этот прототип объявляет, что *maximum* получает три параметра типа *int* и возвращает результат типа *int*. Обратите внимание, что прототип функции имеет тот же вид, что и первая строка определения функции *maximum*, за исключением того, что в него не включены имена параметров (*x*, *y* и *z*).

Имена параметров иногда входят в прототип функции с целью документирования. Компилятор их игнорирует. Пропуск точки с запятой в конце прототипа функции вызывает синтаксическую ошибку. Вызов функции, который не соответствует ее прототипу, вызывает синтаксическую ошибку. Ошибка генерируется также в том случае, если не согласованы прототип и определение функции. Например, если прототип функции записать в виде

```
void maximum(int, int, int);
```

то компилятор зарегистрировал бы ошибку, поскольку возврат типа *void* в прототипе функции отличался бы от возврата типа *int* ее заголовка.

Другое важное следствие применения прототипов функций – *автоматическое приведение аргументов,* т. е. принудительное преоб-

разование аргументов функции к соответствующему типу. Преобразование значений к более высоким типам происходит автоматически. Преобразование значений к более низким типам обычно приводит к неверному результату. Следовательно, значение может быть преобразовано в более низкий тип только посредством явного указания переменной более низкого типа или с помощью операции приведения. Значения аргументов функции преобразуются к типу параметров прототипа функции таким образом, как будто они непосредственно присваиваются переменным этого типа. Если прототип для данной функции не был включен в программу, компилятор формирует собственный прототип функции, используя ее первое вхождение в программу: или определение, или обращение к функции. По умолчанию компилятор предполагает, что функция возвращает тип *int*, ничего не предполагая относительно типа аргументов. Следовательно, если аргументы, переданные функции, некорректны, то компилятор не обнаружит ошибку.

Пропуск прототипа функции вызывает синтаксическую ошибку, если функция возвращает тип, отличный от *int,* а определение функции появляется в программе после вызова функции. В других случаях пропуск прототипа функции может вызвать ошибку во время выполнения программы или привести к непредвиденному результату. Прототип функции, размещенный вне любого ее определения, применяется ко всем вызовам, появляющимся в файле после прототипа функции. Прототип функции, помещенный в функцию, применяется только к тем вызовам, которые делаются из этой функции.

10.2. Определение функции

Определение функции имеет следующий формат:

```
тип_возвращаемого_значения имя_функции (список_параметров)
{
  тело функции;
};
```

В качестве имени функции *(имя_функции)* может использоваться любой допустимый идентификатор. Типом результата, возвращаемого вызывающей функции, является *тип_возвращаемого_значения.* Если *тип_возвращаемого_значения* задан ключевым словом *void*, то это означает, что функция ничего не возвращает. Если

тип_возвращаемого_значения не указан, то компилятор будет предполагать тип *int*.

Если *тип_возвращаемого_значения* в определении функции опущен, то это вызовет синтаксическую ошибку в том случае, если прототип функции определяет, что возвращаемый тип не является целым (*int*).

Если функция должна возвращать некоторое значение, а в функции опущен оператор возврата, то это может привести к непредвиденным ошибкам. Стандарт ANSI гласит, что в этом случае результат не определен. Возврат значения из функции, для которой возвращаемый тип был объявлен как *void*, вызывает синтаксическую ошибку. Хотя опущенный возвращаемый тип по умолчанию расценивается как *int*, всегда задавайте возвращаемый тип явным образом. Однако для функции *main* возвращаемый тип обычно опускается.

Список_параметров – это список объявлений параметров (отделенных запятыми), получаемых функцией при ее вызове. Если функция не получает значения, *список_параметров* обозначается ключевым словом *void*. Тип каждого параметра должен быть указан явно за исключением параметров типа *int*. Если тип параметра не указан, то по умолчанию принимается тип *int*.

Замечания:

1. Ошибочна запись *float x, y* вместо *float x, float y* при объявлении параметров функции, относящихся к одному типу. Такое объявление параметров, как *float x, y,* фактически присвоило бы параметру *y* тип *int*, поскольку *int* принимается по умолчанию.

2. Символ точки с запятой после правой круглой скобки, закрывающей список параметров в определении функции, – синтаксическая ошибка.

3. Повторное определение параметра функции как локальной переменной внутри функции – синтаксическая ошибка.

4. *Объявления* и *операторы* внутри фигурных скобок образуют *тело функции.* Тело функции называют *блоком.* Блок – это просто составной оператор, который включает в себя объявления. Переменные могут быть объявлены в любом блоке, а блоки могут быть вложены. *В любом случае функция не может быть определена внутри другой*

функции. Определение функции внутри другой функции – синтаксическая ошибка.

5. Выбор осмысленных имен функций и параметров делает программы более удобочитаемыми и помогает избежать чрезмерного использования комментариев.

6. Программы должны составляться в виде совокупности небольших функций. Это упрощает создание программ, их отладку, поддержку и модификацию. Функция, требующая большого количества параметров, может выполнять слишком много задач. Рассмотрите возможность ее разделения на меньшие функции, которые выполняют выделенные задачи. Заголовок функции должен, по возможности, умещаться на одной строке.

7. Прототип функции, заголовок определенной функции и ее вызов должны иметь взаимное соответствие по числу, типу и порядку следования аргументов и параметров, а также по типу возвращаемого значения.

Существует три способа возвращения управления в ту точку программы, в которой была вызвана функция. Если функция не возвращает результат, управление возвращается просто при достижении правой фигурной скобки, завершающей функцию, или при исполнении оператора:

```
return;
```

Если функция возвращает результат, применяется оператор

```
return выражение;
```

который возвращает вызывающей функции значение выражения.

10.3. Способы передачи параметров в функции

Существует три способа передачи параметров в функции:

1) по значению;
2) указателю;
3) ссылке.

Приведём пример передачи параметров по значению.

Пример:

```
#include <iostream.h>
double max(double x1, double x2){
 if(x1>x2)  return  x1;  /* выход  из  функции  и  возврат
 значения */
```

```
 else return x2; /* выход из функции и возврат значе-
 ния */
}
```

При вызове функции указываются ее идентификатор и фактические параметры. Функция, возвращающая значение, обычно вызывается в выражении.

Пример:

```
int main(){
 double a1=3;a2=2;a3;
 a3=max(a1,a2); /* вызов функции, a1 и a2 – фактиче-
 ские параметры */
 cout<<a3;
 return 0;
}
```

Если функция не возвращает значения, то используется оператор *return* без параметров или он может быть опущен. Передача адреса переменной позволяет изменять значение внешних переменных внутри функции.

Пример:

```
void swap(double *a, double *b)
{/* передача параметров по значению, но значениями
являются адреса a и b */
 double c;
 c=*a;
 *a=*b;
 *b=c;
}
main(){
 double x=5,y=1;
 swap(&x,&y);
 return 0;
}
```

Передача параметра по ссылке реализуется как передача его адреса. Так как ссылка по определению есть синоним переменной, то при вызове функции формальный параметр становится синонимом фактического параметра. Следовательно, изменения параметра, переданного по ссылке, вызовут изменения внешнего фактического параметра.

Пример:

```
void swap(double &a, double &b){
 double c;
 c=a;
 a=b;
 b=c;
}
main() {
 double x=5, y=1;
 swap(x,y); /* в качестве фактических параметров пе-
 редаются синонимы */
 return 0;
}
```

Так как в языке С++ отсутствует высокоуровневое понятие массива, то передача массива в качестве параметра сводится к передаче указателя на начальный элемент массива. Следствием этого становится необходимость передачи в качестве параметра функции размера массива.

Пример:

```
// вычисление суммы элементов массива
double sum(int n, double *a) {
 double s = 0.0;
 for(int i = 0; i < n; i++)
 s = s + a[i];
 retutn s;
}
int main(){
 double arr[3] ={3,2,1};//arr – указатель на массив
 cout<<sum(3, arr);
 return 0;
}
```

Для передачи адреса массива возможны следующие эквивалентные формы записи.

Пример:

```
double sum(int n, double a[10]){...}
double sum(int n, double a[]){...}
double sum(int n, double *a){...}
```

Передача параметров по значению при вызове функции предполагает, что копия параметра помещается в программный стек. Кроме этого все локальные переменные функции размещаются в стеке при каждом входе в процедуру.

10.4. Вызов функции

Во многих языках программирования существует два способа вызова функций – *вызов по значению* и *вызов по ссылке.* Когда аргумент используется в вызове по значению, то вызываемой функции передается *копия* значения аргумента. Изменения, происходящие с копией, не отражаются на значении исходной переменной в вызывающей функции. Когда аргумент функции передается по ссылке, вызывающая функция фактически позволяет вызываемой функции изменять значение исходной переменной.

Передача аргумента по значению должна использоваться, когда вызываемой функции не нужно менять значение исходной переменной в вызывающей функции. Это предотвращает случайные *побочные эффекты,* которые мешают разработке правильных и надежных систем программного обеспечения. Передача аргумента по ссылке должна применяться только с «доверенными» вызываемыми функциями, которым необходимо менять первоначальную.

10.5. Параметры функций по умолчанию

Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию. Эти параметры должны быть последними в списке и могут опускаться при вызове функции. Синтаксис для параметров функции по умолчанию подобен синтаксису для инициализации переменных:

```
тип имя = значение
```

В C++ требуется выполнение следующих правил для использования параметров по умолчанию:

1) когда назначается значение по умолчанию какому-либо параметру, то необходимо назначать значения по умолчанию всем последующим параметрам;

2) принимаемые по умолчанию параметры делят список на две части: первая не содержит значений, вторая содержит;

3) чтобы использовать значение параметра по умолчанию в качестве фактического, следует опустить фактический параметр на месте, соответствующем формальному параметру при вызове функций;

4) если при вызове параметр опущен, должны быть опущены и все параметры, стоящие за ним.

```
double kor(double a, double n=2.0)
{
    return pow(a,1./n);
}
//вызов функции
y=kor(64,3);
y=kor(64);
```

10.6. Перегрузка функций

В языке Си объявление двух функций с одним и тем же именем в одной и той же программе является синтаксической ошибкой. C++ допускает определение нескольких функций с одним и тем же именем, пока эти функции различаются наборами параметров, типами или порядком следования. Такая возможность называется *перегрузкой функций.* При вызове перегруженной функции компилятор C++ автоматически выбирает соответствующую функцию исходя из анализа числа, типа и порядка параметров вызова. Перегрузка функций обычно используется для создания нескольких функций с одним и тем же именем, производящих схожие действия над различными типами данных.

Перегруженные функции, выполняющие аналогичные задачи, делают программы удобочитаемыми и понятными.

В представленном ниже примере перегруженная функция *square* используется для вычисления квадрата чисел типов *int* и *double*.

Пример:

```
// Использование перегруженных функций
#include <iostream>
using namespace std;

int square(int x) { return x * x; }

double square(double y) { return y * y; }

main () {
     cout <<"The square of integer 7 is
"<<square(7)<<"\nThe square of double 7.5 is
"<<square(7.5)<< '\n';
return 0 ;
}
```

Результаты работы программы:

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Перегруженные функции различаются своей *сигнатурой* – комбинацией имени функции и типов ее параметров. Компилятор специальным образом кодирует идентификатор каждой функции, используя количество и типы ее параметров, для обеспечения *безопасного по типу* редактирования связей (компоновки) (иногда этот процесс называют *декорированием*). Безопасное по типу редактирование связей гарантирует вызов соответствующей перегруженной функции и должное согласование аргументов с параметрами. Компилятор выявляет ошибки компоновки и сообщает о них.

Программа, представленная ниже, транслировалась с помощью компилятора Borland C++. Закодированные имена функций на языке ассемблера, генерированные компилятором, показаны в окне вывода. Каждое декорированное имя начинается с символа @, за которым следует имя функции. Закодированный список параметров начинается с символов *$q*. В списке параметров функции *nothing2 zc* представляет тип *char, i* представляет тип *int, pf* представляет *float* * и *pd* представляет *double* *. В списке параметров функции *nothingl i* представляет тип *int, f* представляет тип *float, zc* представляет тип *char* и *pi* представляет *int* *.

Пример:

```
/* Декорирование имен, обеспечивающее безопасное по
типу редактирование связей */

int square(int x) { return x * x; }
double square(double y) { return y * y; }
void nothingl (int a, float b, char c, int *d) {}
char *nothing2 (char a, int b, float *c, double *d)
{ return 0; }
main ()
{
return 0;
}

____
public _main
```

```
public @nothing2$qzcipfpd
public @nothingl$qifzcpi
public @square$qd
public @square$qi
```

Две функции *square* различаются своими списками параметров; в одном *d* объявляется как *double*, а в другом *i* объявляется как *int*. Типы возвращаемых значений всех четырех функций в закодированных именах не специфицируются. Кодирование имен функций зависит от компилятора, поэтому функция, компилированная в Borland C++, может иметь декорированное имя, отличное от того, которое она получила бы при использовании других компиляторов.

Перегруженные функции могут отличаться разными типами возвращаемых значений, но при этом они обязаны все равно иметь различные списки параметров. В процессе декорирования тип возвращаемого значения не участвует.

Перегруженные функции не обязаны обладать одним и тем же количеством параметров. Программисты должны проявлять осторожность при перегрузке функций с аргументами по умолчанию, поскольку это может приводить к неоднозначности.

Функция с опущенными аргументами по умолчанию может вызываться точно так же, как другая перегруженная функция; это является синтаксической ошибкой.

10.7. Операторные функции. Перегрузка операторов

Операторные функции определяются с помощью ключевого слова *operator*. Для базовых типов операторные функции уже определены и перегружены. Для упрощения работы с пользовательскими типами данных можно также перегружать основные операции. *Перегрузка оператора* состоит в изменении его смысла.

```
тип_возращаемого_значения operator <знак_операции> (тип, тип).
```

Пример:

```
//объявление пользовательского типа данных
struct Complex
{
     double re, im;
};
```

```
//объявление операторной функции
Complex operator+ (Complex c1, Complex c2)
{
     Complex c3;
     c3.re=c1.re+c2.re;
     c3.im=c1.im+c2.im;
     return c3;
}
//использование в функциях:
Complex c1, c2, c3;
c1.re=1;
c1.im=1;
c2.re=2;
c2.im=2;
c3=c1+c2; //неявный вызов перегруженного оператора
c3 = operator +(c1, c2);//явный вызов перегруженного
оператора
```

10.8. Шаблоны функций

Многие алгоритмы не зависят от типа данных, с которыми они работают (классический пример – сортировка). Естественно, возникает желание параметризовать алгоритм таким образом, чтобы его можно было использовать для различных типов данных.

В C++ есть мощное средство параметризации – шаблоны. С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных – передаваться функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу. Таким образом, создается функция, которая автоматически перегружает сама себя и при этом не содержит накладных расходов, связанных с параметризацией.

Формат простейшей функции-шаблона:

```
template <class Type>

заголовок функции {
/* тело функции */
}
```

Вместо слова *Type* может использоваться произвольное имя.

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только типом, но и просто переменной, например:

```
template<class A, class B, int C>
void f(){ ... }
```

Например, функция, сортирующая методом выбора массив из *n* элементов любого типа, в виде шаблона может выглядеть так:

```
#include<iostream>
using namespace std;

template <class Type>

void sort_vybor(Type *b, int n);

int main()
{
    const int n=20;
    int i, b[n];
    for (i= 0; i<n-1; i++) cin>>b[i];
    sort_vybor(b, n);
    for (i=0; i<n-1; i++) cout<<b[i]<<' ';
    cout<<'\n';
    double a[]={3,2,5,1,7};
    sort_vybor(a, 5);
    for (i= 0; i<n-1; i++) cout<<a[i]<<' ';
    cout<<'\n';
    return 0;
}

template <class Type>
void sort_vybor(Type *b, int n)
{
   Type a; //буферная переменная для обмена элементов
    for (int i= 0; i<n-1; i++)
    {
        int imin = i;
        for (int j = i + 1;j<n; j++)
        if (b[j] < b[imin])     imin= j;
        a=b[i]; b[i]=b[imin]; b[imin]=a;
    }
}
```

10.9. Встраиваемые функции

Когда в программе определена функция, компилятор C++ переводит код функции в машинный язык, сохраняя только одну копию инструкций функции внутри программы. Каждый раз, когда программа вызывает функцию, компилятор C++ помещает в программу специальные инструкции, которые заносят параметры функции в стек и затем выполняют переход к инструкциям этой функции. Когда операторы функции завершаются, выполнение программы продолжается с первого оператора, который следует за вызовом функции. Помещение аргументов в стек и переходы в функцию и из нее вносят издержки, из-за которых программа выполняется немного медленнее, чем если бы она размещала те же операторы прямо внутри программы при каждой ссылке на функцию. Например, предположим, что программа, приведённая в примере 1, вызывает функцию *show_message*, которая указанное число раз выдает сигнал на динамик компьютера и затем выводит сообщение на дисплей:

Пример 1:

```
#include <iostream>
using namespace std;

void show_message(int count, char *message)
{
    int i;
    for (i = 0; i < count; i++) cout << '\a';
    cout << message << endl;
}

void main(void)
{
   show_message(3, "Учимся программировать на языке
   C++");
   show_message(2, "Пример");
}
```

Программа, приведённая в примере 2, не вызывает функцию *show_message*. Вместо этого она помещает внутри себя те же операторы функции при каждой ссылке на функцию.

Пример 2:

```
#include <iostream>
using namespace std;

void main (void){
   int i;
   for (i = 0; i < 3; i++) cout << 'a';
   cout << " Учимся программировать на языке C++" <<
   endl;
   for (i = 0; i < 2; i++) cout << 'a';
   cout << "пример" << endl;
}
```

Обе программы выполняют одно и то же. Поскольку вторая программа не вызывает функцию *show_message*, она выполняется немного быстрее, чем первая. В данном случае разницу во времени выполнения определить невозможно, но если в обычной ситуации функция будет вызываться 1000 раз, вероятно, будет заметно небольшое увеличение производительности. Однако вторая программа более запутана, чем первая, а следовательно, более тяжела для восприятия.

При объявлении функции внутри программы C++ позволяет предварить имя функции ключевым словом *inline*. Если компилятор C++ встречает ключевое слово *inline*, он помещает в выполнимый файл (машинный язык) операторы этой функции в месте каждого ее вызова. Таким образом, можно улучшить читаемость программы, используя функции, и в то же время увеличить производительность, избегая издержек на вызов функций. Следующая программа определяет функции *max* и *min* как *inline* функции.

Пример:

```
#include <iostream>
using namespace std;

inline int max(int a, int b)
{
      (a > b)? return(a): return(b);
}
inline int min(int a, int b)
{
      (a < b)? return(a): return(b);
}
```

```
void main(void)
{
     cout << "Минимум из 1001 и 2002 равен " <<
     min(1001, 2002) << endl;
     cout << "Максимум из 1001 и 2002 равен " <<
     max(1001, 2002) << endl;
}
```

В данном случае компилятор C++ заменит каждый вызов функции на операторы функции. Производительность программы увеличивается без ее усложнения.

Код функции, помеченной ключевым словом *inline*, будет подставлен в точку ее вызова в следующих случаях:

1) в функции отсутствуют операторы условного перехода;
2) в функции отсутствуют циклические операторы;
3) определение функции следует до ее вызова;
4) в коде программы нет вызова функции по указателю.

Если компилятор C++ перед определением функции встречает ключевое слово *inline* и выполнены все перечисленные выше условия, он будет заменять обращения к этой функции (вызовы) на последовательность операторов, эквивалентную выполнению функции. Если же хоть одно условие не выполняется, то функция становится статической, т. е. будет сгенерирован вызов функции (передача управления).

10.10. Параметры, передаваемые через командную строку

Программы могут принимать аргументы. Функция *main* может содержать в качестве аргументов два параметра: *argc* – целочисленная переменная, содержащая количество переданных аргументов через командную строку, и *argv* – массив строк, в котором содержатся значения переданных аргументов. Отметим, что *argv[0]* содержит путь и имя запущенной программы, а начиная с первого и до *argc-1* – программные параметры.

Ниже приведен пример программы, которая печатает список аргументов, переданных ей в командной строке.

Пример:

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i;
```

```
    for( i = 0 ; i < argc; i++)
     printf("Argument %d: %s\n", i, argv[i]);

    if(argc == 1)
     printf("Command  line  has  no  additional  argu-
     ments\n");
    return 0;
}

bash$ ./argv alpha beta gamma last
Argument 0: ./a.out
Argument 1: alpha
Argument 2: beta
Argument 3: gamma
Argument 4: last
```

10.11. Рекурсия

Понятие рекурсии

Функция называется рекурсивной, если во время обработки возникает ее повторный вызов либо непосредственно (функция вызывает сама себя), либо косвенно, путем цепочки вызовов других функций.

Прямую (непосредственную) рекурсию представляет собой вызов функции внутри тела этой функции:

```
int a()
{.....a().....}
```

Косвенной называется рекурсия, осуществляющая рекурсивный вызов функции посредством цепочки вызова других функций. Все функции, входящие в цепочку, тоже считаются рекурсивными.

Например:

```
a(){.....b().....}
b(){.....c().....}
c(){.....a().....}.
```

Все функции *a, b, c* рекурсивные, так как при вызове одной из них осуществляется вызов других и самой себя.

Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции. При повторном вызове этот процесс повторяется. Ясно, что для завершения вычислений каждая рекурсивная функция должна содержать хотя

бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

При написании рекурсивных функций следует использовать оператор условия, чтобы заставить функцию вернуться без рекурсивного вызова. Если этого не сделать, то, однажды вызвав функцию, выйти из нее будет невозможно.

Достоинство рекурсии – компактная запись, а недостатки – расход времени и памяти на повторные вызовы функции и передача ей копий параметров, а главное – опасность переполнения стека.

Рекурсивные математические функции

Простой пример рекурсии – функция *fact()*, вычисляющая факториал целого числа. Факториал числа N есть произведение целых чисел от 1 до N. Для вычисления факториала следует число n умножить на факториал числа n – 1. Известно также что 0! = 1 и 1! = 1.

Пример:

```
/* Вычисление факториала числа */
int factorial(int n) /* нерекурсивная реализация */
{
   int t, answer;
   answer = 1;
   for(t=1; t<=n; t++)
   answer = answer*t;
   return answer;
}
/* Вычисление факториала числа */
 long fact(int n) /* рекурсивно */
{
   if(n==1 || n==0)) return 1;
   return fact(n-1)*n;
}
/* второй вариант реализации функции fact*/
long fact(int n) /* рекурсивно */
{
   return (n>1)?fact(n-1)*n:1;
}
```

Еще один пример – задача вычисления чисел Фибоначчи. Числа Фибоначчи образуют последовательность $\{1, 1, 2, 3, 5, ...\}$, вычисляемую по правилу

$f_1 = 1,$

$f_2 = 1,$

$f_k = f_{k-1} + f_{k-2}.$

Приведём пример применения рекурсии для упорядочивания по возрастанию чисел методом быстрой сортировки, предложенной Хоаром в 1961 г.

Метод быстрой сортировки состоит в следующем.

В массиве выбирается элемент, относительно которого ведётся сортировка (как правило, это средний элемент $a[k]$). Сначала перебирают все элементы, расположенные слева от $a[k]$, начиная с первого до элемента большего либо равного по значению $a[i] \geq a[k]$. После этого перебирают все элементы, расположенные справа от $a[k]$, начиная с последнего до элемента меньшего либо равного по значению $a[j] \leq a[k]$. Элементы $a[i]$ и $a[j]$ меняются местами. Эта процедура продолжается до тех пор, пока выполняется условие $i < j$. Если же это условие не выполняется, то массив разбивается на два новых и все действия повторяются до тех пор, пока новый массив содержит не менее двух элементов.

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;

void quicksort(int *a, int l, int r);

int main ()
{
    const int n=10;
    int a[n],i;
    randomize();
    for(i=0;i<n;i++)
    {
        a[i]=random(50);
        cout<<a[i]<<" ";
    }
```

```
     quicksort(a,0,n-1);
     cout<<"\nPosle\n";
     for(i=0;i<n;i++)
          cout<<a[i]<<" ";
     getch();
}
void quicksort(int *a, int l, int r)
{
     int i,j;
     int buf,p;
     if(l<r)
     {
          p=a[(l+r)/2];
          i=l;
          j=r;
          do
          {
               while(a[i]<p) i++;
               while(a[j]>p) j--;
               if(i<=j)
               {
                    buf=a[i];
                    a[i]=a[j];
                    a[j]=buf;
                    i++;
                    j--;
               }
          } while(i<j);
     quicksort(a,l,j);
     quicksort(a,i,r);
     }
}
```

❓ Вопросы и задания

1. Что такое функция?
2. Для чего применяют прототипы функций?
3. В чем отличия объявления и определения функции?
4. Что означает передача аргумента в функцию по ссылке? Приведите пример.

5. Напишите функцию ввода с клавиатуры переменной структурного типа (структура *Студент* содержит поля: ФИО, дата рождения, успеваемость по пяти дисциплинам). Сформированные структуры из функции получите следующими способами:

a) в качестве параметра функции;

b) в качестве возвращаемого значения.

Продемонстрируйте работу функции.

6. Верните из функции массив указателей на целочисленные переменные. Продемонстрируйте работу функции.

7. Напишите шаблонную функцию для нахождения суммы элементов числового массива, массив передайте в функцию в качестве параметра. Приведите пример использования функции для нескольких числовых типов.

8. Объявите указатель на функцию, имеющую следующий прототип:

```
int func(char *, int);
```

9. Каким образом можно передавать одномерные массивы в функции? Приведите примеры.

10. Поясните принцип передачи многомерных массивов в функции.

11. Что такое операторные функции? Назначение перегрузки операторов.

12. Дайте определение перегруженной функции.

13. За счет какого механизма возможна перегрузка функций в C++?

14. Перегрузите операторы /, *, << для комплексных чисел, описанных в разделе 12.7. Оператор << отвечает за вывод на экран комплексного числа.

15. Объявите *inline* функцию, например вычисления квадрата числа. Объявите указатель на эту функцию. Вызовите *inline* функцию через указатель. Будет ли функция в этом случае трактоваться как встраиваемая?

Тема 11. ФАЙЛЫ. БАЗОВЫЕ ФУНКЦИИ РАБОТЫ С ПОТОКАМИ

11.1. Понятие физического и логического файлов

У понятия файла есть две стороны. С одной стороны, *файл* – это именованная область внешней памяти, содержащая какую-либо информацию. Файл в таком понимании называют *физическим*, т. е. существующим физически на некотором материальном носителе информации. С другой стороны, файл – это одна из многих структур данных, используемых в программировании. Файл в таком понимании называют *логическим*, т. е. существующим только в нашем логическом представлении при написании программы. В программах логические файлы представляются файловыми переменными определенного типа.

Структура физического файла представляет собой простую последовательность байт памяти носителя информации. Структура логического файла – это способ восприятия файла в программе. Образно говоря, это «шаблон» («окно»), через который мы смотрим на физическую структуру файла. В языках программирования таким «шаблонам» соответствуют типы данных, допустимые в качестве компонент файлов.

Логическая структура файла в принципе очень похожа на структуру массива. Различия между массивом и файлом заключаются в следующем.

У массива количество элементов фиксируется в момент распределения памяти, и он целиком располагается в оперативной памяти. Нумерация элементов массива выполняется соответственно нижней и верхней границам, указанным при его обновлении.

У файла количество элементов в процессе работы программы может изменяться, и он располагается на внешних носителях информации. Нумерация элементов файла выполняется слева направо, начиная от нуля (кроме текстовых файлов). Количество элементов файла в каждый момент времени неизвестно. Зато известно, что в конце файла располагается специальный символ конца файла *EOF*, в качестве которого используется управляющий символ ASCII с кодом 26 (Ctrl + Z). Кроме того, определить длину файла и выполнить другие часто требуемые операции можно с помощью стандартных процедур и функций, предназначенных для работы с файлами.

11.2. Работа с файлами

Указатель файла

Указатель файла – это указатель на структуру типа FILE, содержащую различные сведения о файле, например, его имя, статус, текущую позицию в файле. Для выполнения в файлах операций чтения и записи программы должны использовать соответствующие указатели файлов. Для объявления переменной-указателя файла используют оператор

```
FILE * <название указателя файла>;
```

Открытие файла

Функция *fopen()* открывает поток и связывает с этим потоком определенный файл, затем возвращает указатель этого файла. Прототип функции *fopen()*

```
FILE* fopen (const char *filename, const char *mode);
```

Параметр *filename* – допустимое имя файла, в которое может входить спецификация пути к этому файлу. Строка *mode* – режим открытия файла:

r – открытие файла только для чтения;

w – создание файла для записи;

a – присоединение; открытие для записи в конец файла или создание для записи, если файл не существует;

r+ – открытие существующего файла для обновления (чтения и записи);

w+ – создание нового файла для изменения;

a+ – открытие для присоединения; открытие (или создание, если файл не существует) для обновления в конец файла.

Если данный файл открывается или создается в текстовом режиме, вы можете приписать символ *t* к значению параметра *type* (*rt, w+t* и т. д.); аналогично для спецификации бинарного режима вы можете к значению параметра *type* добавить символ *b* (*wb, a+b* и т. д.). Если в параметре *type* отсутствуют символы *t* или *b*, режим будет определяться глобальной переменной *_fmode*. Если переменная *_fmode* имеет значение *O_BINARY*, файлы будут открываться в бинарном режиме, иначе, если *_fmode* имеет значение *O_TEXT*, файлы открываются в текстовом режиме. Данные переменной *_fmode* определены в файле *fcntl.h*. При открытии файла в режиме обновления над

результирующим потоком *stream* могут выполняться как операции ввода, так и вывода. Тем не менее, вывод не может следовать непосредственно за вводом без вмешательства функций *fseek* или *rewind,* а также ввод без применения функций *fseek* и *rewind* не может непосредственно следовать за выводом или вводом, который встречает конец файла (*EOF*).

При успешном завершении *fopen* возвращает указатель на открытый поток *stream*. В случае ошибки функция возвращает нуль (*NULL*).

Закрытие файла

Функция *fclose()* закрывает поток, который был открыт с помощью вызова *fopen()*. Функция *fclose()* записывает в файл все данные, которые ещё оставались в дисковом буфере, и проводит официальное закрытие файла на уровне операционной системы. Отказ при закрытии потока влечет всевозможные неприятности, включая потерю данных, испорченные файлы и возможные периодические ошибки в программе. Функция *fclose()* также освобождает блок управления файлом, связанный с этим потоком, давая возможность использовать этот блок снова. Так как количество одновременно открытых файлов ограничено, то, возможно, придется закрывать один файл, прежде чем открывать другой.

Прототип функции *fclose()* такой:

```
int fclose(FILE *уф);
```

где уф – указатель на файл, возвращенный в результате вызова *fopen()*. Возвращение нуля означает успешную операцию закрытия. В случае ошибки возвращается *EOF*. Чтобы точно узнать, в чем причина этой ошибки, можно использовать стандартную функцию *ferror()*.

Пример:

```
#include <stdio.h>
void main()
 {
     FILE *fp;
     fp = fopen("file.dat", "r");
     if (fp==NULL)
     printf("Файл данных не открыт\n");
     else
```

```
        {
             fclose(fp);
             printf("Файл file.dat\n");
        }
    }
```

Использование foef()

Для проверки факта достижения конца файла применяют функцию *feof()*, прототип которой имеет следующий вид:

```
int feof(FILE *уф);
```

Если достигнут конец файла, то *feof()* возвращает *true*; в противном случае эта функция возвращает нуль.

11.3. Текстовые (строковые) файлы

Текстовые файлы относятся к файлам последовательного доступа, так как единицей хранения информации в них служат строки переменной длины. Каждая строка заканчивается специальным признаком, обычно его функцию выполняет пара символов `'\r'` («возврат каретки») и `'\n'` («перевод строки»). Самое важное преимущество текстовых файлов – универсальность формата хранения информации: числовые данные в символьном виде доступны на любом компьютере, при необходимости их может прочитать и человек. Однако это преимущество имеет и обратную сторону медали: преобразование числовых данных из машинных форматов в символьный вид при выводе и обратное преобразование при вводе сопряжены с дополнительными расходами. Кроме того, объем числовых данных в символьном формате занимает в несколько раз больше памяти по сравнению с их машинным представлением.

Для работы с текстовым файлом необходимо завести указатель на структуру типа *FILE* и открыть файл по оператору *fopen* в одном из нужных режимов: «*rt*» (текстовый для чтения), «*wt*» (текстовый для записи), «*at*» (текстовый для дозаписи в уже существующий набор данных):

```
FILE *f1;
.........
f1=fopen(имя_файла, "режим");
```

Формат оператора обмена с текстовыми файлами мало чем отличается от операторов форматного ввода (*scanf*) и вывода (*printf*). Вместо них при работе с файлами используются функции *fscanf* и

fprintf, у которых единственным дополнительным аргументом является указатель на соответствующий файл:

```
fscanf(f1,"список_форматов", список_ввода);
fprintf(f1,"список_форматов \n",список_вывода);
```

Если очередная строка текстового файла формируется из значения элементов символьного массива *str*, то вместо функции *fprintf* проще воспользоваться функцией *fputs(f1, str).* Чтение полной строки из текстового файла удобнее выполнить с помощью функции *fgets(str, n, f1).* Здесь параметр *n* означает максимальное количество считываемых символов, если раньше не встретится управляющий байт *0A.*

Библиотека C++ предусматривает и другие возможности для работы с текстовыми файлами – функции *open, creat, read, write*.

Пример:

Рассмотрим программу, которая создает в текущем каталоге (т. е. в каталоге, где находится наша программа) текстовый файл с именем *c_txt* и записывает в него 10 строк. Каждая из записываемых строк содержит символьное поле с текстом "Line" (5 байт, включая нулевой байт – признак конца строки), пробел, поле целочисленного значения переменной j, пробел и поле вещественного значения квадратного корня из j. Очевидно, что числовые поля каждой строки могут иметь разную длину. После записи информации файл закрывается и вновь открывается, но уже для чтения. Для контроля содержимое записываемых строк и содержимое считанных строк дублируется на экране.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
void main( )
{
 FILE *f;//указатель на блок управления файлом
 int j,k;
 double d;
 char s[]="Line";
 f=fopen("c_txt","wt"); //создание нового или // от-
                крытие существующего
                //файла для записи
 for(j=1;j<11;j++)
 {
      //запись в файл
      fprintf(f,"%s %d %lf\n",s,j,sqrt(j));
```

```
     //вывод на экран
     printf("%s %d %lf\n",s,j,sqrt(j));
}
fclose(f);              //закрытие файла
printf("\n");
//открытие файла для чтения
f=fopen("c_txt","rt");
for(j=10; j>0; j--)
{
     //чтение из файла
     fscanf(f,"%s %d %lf",s,&k,&d);
     //вывод на экран
     printf("%s %d %lf\n",s,k,d);
}
getch();
}
//== Результат работы ===
Line 1 1.000000
Line 2 1.414214
Line 3 1.732051
Line 4 2.000000
Line 5 2.236068
Line 6 2.449490
Line 7 2.645751
Line 8 2.828427
Line 9 3.000000
Line 10 3.162278

Line 1 1.000000
Line 2 1.414214
Line 3 1.732051
Line 4 2.000000
Line 5 2.236068
Line 6 2.449490
Line 7 2.645751
Line 8 2.828427
Line 9 3.000000
Line 10 3.162278
```

Обратите внимание на возможную ошибку при наборе этой программы. Если между форматными указателями *%s* и *%d* не сделать пробел, то в файле текст "Line" склеится с последующим целым числом. После этого при чтении в переменную s будут попадать строки

вида "Line1", "Line2", …, "Line10", в переменную k будут считываться старшие цифры корня из j (до символа «точка»), а в переменной d окажутся дробные разряды соответствующего корня. Тогда результат работы программы будет выглядеть следующим образом:

```
Line1 1.000000
Line2 1.414214
…
```

При считывании данных из текстового файла надо следить за ситуацией, когда данные в файле исчерпаны. Для этой цели можно воспользоваться функцией *feof*:

```
if(feof(f1))...    //если данные исчерпаны
```

11.4. Двоичные файлы

Двоичные файлы отличаются от текстовых тем, что представляют собой последовательность байтов, содержимое которых может иметь различную природу. Это могут быть байты, отображающие числовую информацию в машинном формате, байты с графическими изображениями, байты с аудиоинформацией и т. п. Содержимое таких байтов может случайно совпасть с управляющими кодами таблицы ASCII, но на них нельзя реагировать так, как при обработке текстовой информации. Естественно, что единицей обмена с такими данными могут быть только порции байтов указанной длины.

Создание двоичных файлов с помощью функции *fopen* отличается от создания текстовых файлов только указанием режима обмена – «*rb*» (двоичный для чтения), «*rb+*» (двоичный для чтения и записи), «*wb*» (двоичный для записи), «*wb+*» (двоичный для записи и чтения):

```
FILE *f1;
.........
f1=fopen(имя_файла, "режим");
```

Обычно для обмена с двоичными файлами используют функции *fread* и *fwrite*. Данные в двоичный файл записывают с помощью функции *fwrite*:

```
c_w = fwrite(buf, size_rec, n_rec, f1);
```

Здесь *buf* – указатель типа *void** на начало буфера в оперативной памяти, из которого информация переписывается в файл;

size_rec – размер передаваемой порции в байтах;

n_rec – количество порций, которое должно быть записано в файл;

f1 – указатель на блок управления файлом;

c_w – количество порций, которое фактически записалось в файл.

Считывают данные из двоичного файла с помощью функции *fread* с таким же набором параметров:

```
c_r = fread(buf, size_rec, n_rec, f1);
```

Здесь *c_r* – количество порций, которое фактически прочиталось из файла;

buf – указатель типа *void** на начало буфера в оперативной памяти, в который информация считывается из файла.

Обратите внимание на значения, возвращаемые функциями *fread* и *fwrite*. В какой ситуации количество записываемых порций может не совпасть с количеством записавшихся данных? Как правило, когда на диске не хватило места и на такую ошибку надо реагировать. А вот при чтении ситуация, когда количество прочитанных порций не совпадает с количеством запрашиваемых порций, не обязательно является ошибкой. Типичная картина – количество данных в файле не кратно размеру заказанных порций.

Двоичные файлы допускают не только последовательный обмен данными. Так как размеры порций данных и их количество, участвующее в очередном обмене, диктует программист, а не смысл информации, хранящейся в файле, то есть возможность пропустить часть данных или вернуться повторно к ранее обработанной информации. Контроль за текущей позицией доступных данных в файле осуществляет система с помощью указателя, находящегося в блоке управления файлом. С помощью функции *fseek* программист имеет возможность переместить этот указатель:

```
fseek(f1,delta,pos);
```

Здесь *f1* – указатель на блок управления файлом;

delta – величина смещения в байтах, на которую следует переместить указатель файла;

pos – позиция, от которой производится смещение указателя (0 или *SEEK_SET* – от начала файла, 1 или *SEEK_CUR* – от текущей позиции, 2 или *SEEK_END* – от конца файла).

Пример:

Рассмотрим программу, которая создает двоичный файл для записи с именем *c_bin* и записывает в него $4 \cdot 10$ порций данных в машинном формате (строки, целые и вещественные числа). После запи-

си данных файл закрывается и вновь открывается для чтения. Для демонстрации прямого доступа к данным информация из файла считывается в обратном порядке – с конца. Контроль записываемой и считываемой информации обеспечивается дублированием данных на экране дисплея.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
void main( )
{
    FILE *f1;  //указатель на файл
    int j, k;
    char s[]="Line";
    int n;
    float r;
    f1=fopen("c_bin","wb");  //создание двоичного файла
для записи
    for(j=1;j<11;j++)
    {
        r=sqrt(j);
        //запись строки в файл
        fwrite(s,sizeof(s),1,f1);
        //запись целого числа в файл
        fwrite(&j,sizeof(int),1,f1);
        //запись вещественного числа
        fwrite(&r,sizeof(float),1,f1);
        //контрольный вывод
        printf("\n%s %d %f", s, j, r);
    }
    fclose(f1);     //закрытие файла
    printf("\n");
    //открытие двоичного файла для чтения
    f1=fopen("c_bin", "rb");
    for(j=10; j>0; j--)
    {
     //перемещение указателя файла
     fseek(f1,(j-1)  *  (sizeof(s)  +  sizeof(int)  +
 sizeof(float)), SEEK_SET);
     fread(&s,sizeof(s),1,f1);   //чтение строки
     // чтение целого числа
     fread(&n,sizeof(int),1,f1);
     // чтение вещественного числа
     fread(&r,sizeof(float),1,f1);
```

```
        //контрольный вывод
        printf("\n%s %d %f",s,n,r);
       }
       getch();
}
```

? Вопросы и задания

1. Что такое файл?
2. Приведите отличия логического и физического файлов.
3. Перечислите типы файлов.
4. Приведите классификацию файлов по способам доступа к информации.
5. Какие действия необходимо выполнить для работы с файлом?
6. Какая функция используется для открытия файла? Опишите ее параметры.
7. Каким образом можно определить, достигнут ли конец файла?
8. Напишите программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.
9. Если требуется осуществить быстрое копирование файлов неизвестной структуры, какого типа файл нужно использовать?
10. Напишите программу, которая считывает текст из файла и выводит на экран, после каждого предложения добавляя, сколько раз встречалось в нем введенное с клавиатуры слово.

ЗАКЛЮЧЕНИЕ

Учебное пособие, не делая акцента ни на одной области практики, знакомит читателей с основными принципами автоматизированного решения задач с применением ЭВМ. В частности, рассматриваются принципы разработки алгоритмов как простейших, так и комбинированных, излагаются семантика и синтаксис высокоуровневого языка программирования C++.

Конечно, книга не может дать все знания по программированию, однако имеющаяся в ней информация достаточно полно отражает все главные задачи дисциплины, демонстрирует основные способы их решения.

После освоения материала пособия студент или любой другой изучающий дисциплину человек получит достаточный объем знаний, необходимых для решения серьезных прикладных задач. Пособие снабжено списком рекомендуемой литературы, в которой любой желающий может найти более полную информацию по узким вопросам дисциплины.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Воронова, Л. М.* Типовые алгоритмические структуры для вычислений : учеб. пособие / Л. М. Воронова ; Владим. гос. ун-т. – 2-е изд., перераб. и доп. – Владимир : Ред.-издат. комплекс ВлГУ, 2004. – 96 с. – ISBN 5-89368-503-2.

2. Алгоритмы: построение и анализ / Т. Х. Кормен [и др.]. – М. : Вильямс, 2016. – 1328 с. – ISBN 978-5-8459-2016-4.

3. *Прата, С.* Язык программирования C++. Лекции и упражнения : пер. с англ. / С. Прата – М. : Вильямс, 2017. – 1248 с. – ISBN 978-5-8459-2048-5. – ISBN 978-5-8459-1778-2.

4. *Страуструп, Б.* Программирование. Принципы и практика с использованием C++ / Б. Страуструп. – М. : Вильямс, 2016. – 1328 с. – ISBN 978-5-8459-1949-6. – ISBN 978-0-321-99278-9.

5. *Шилдт, Г.* Полный справочник по C++ : пер. с англ. / Г. Шилдт. – 4-е изд. – М. : Вильямс, 2015. – 800 с. – ISBN 978-5-8459-2047-8. – ISBN 0-07-222680-3.

ПРИЛОЖЕНИЕ

Таблицы ASCII-кодов

Символы с кодами от 0 до 127

0 -	16 - ▶	32 -	48 - 0	64 - @	80 - P	96 - '	112 - p
1 - ☺	17 - ◀	33 - !	49 - 1	65 - A	81 - Q	97 - a	113 - q
2 - ☻	18 - ↕	34 - "	50 - 2	66 - B	82 - R	98 - b	114 - r
3 - ♥	19 - ‼	35 - #	51- 3	67 - C	83 - S	99 - c	115 - s
4 - ♦	20 - ¶	36 - $	52 - 4	68 - D	84 - T	100 - d	116 - t
5 - ♣	21 - §	37 - %	53 - 5	69 - E	85 - U	101 - e	117 - u
6 - ♣	22 - ▬	38 -&	54 - 6	70 - F	86 - V	102 - f	118 - v
7 -	23 - ↕	39 - '	55 - 7	71 - G	87 - W	103 - g	119 - w
8 -	24 - ↑	40 - (	56 - 8	72 - H	88 - X	104 - h	120 - x
9 -	25 - ↓	41 -)	57 - 9	73 - I	89 - Y	105 - i	121 - y
10 -	26 - →	42 - *	58 - :	74 - J	90 - Z	106 - j	122 - z
11 -	27 - ←	43 - +	59 - ;	75 - K	91 - [	107 - k	123 - {
12 -	28 - ∟	44 - ,	60 - <	76 - L	92 - \	108 - l	124 - |
13 -	29 - ↔	45 - -	61 - =	77 - M	93 - j	109 - m	125 - }
14 - ♫	30 - ▲	46 - .	62 ->	78 - N	94 - ^	110 - n	126 - ~
15 - ☼	31 - ▼	47 - /	63 - ?	79 - O	95 - ÷	111 - o	127 - ⌂
16 - ▶	32 -	48 -0	64 - @	80 - P	96 -	112 - p	

Символы с кодами от 128 до 255

128 -А	144 - Р	160 - а	176 - ▒	192 - └	208 - ╨	224 -р	240 - Ё
129 - Б	145 - С	161 - б	177 - ▒	193 - ┴	209 - ╤	225 -с	241- ё
130 -В	146 - Т	162 - в	178 - ▓	194 - ┬	210 - ╥	226 -т	242 - Є
131 - Г	147 - У	163 - г	179 - |	195 - ├	211 - ╙	227 -у	243 - є
132 -Д	148 - Ф	164 - д	180 - ┤	196 - —	212 - ╘	228 -ф	244 - Ї
133 -Е	149 - Х	165 - е	181 - ╡	197 - ┼	213 - ╒	229 - х	245 - ї
134 - Ж	150 - Ц	166 - ж	182 - ╢	198 - ╞	214 - ╓	230 - ц	246 - Ў
135 -З	151 - Ч	167 - з	183 - ╖	199 - ╟	215 - ╫	231 - ч	247 - ў
136 -И	152 -Ш	168 - и	184 - ╕	200 -	216 - ╪	232 - ш	248 - °
137 - Й	153 -Щ	169 - й	185 - ╣	201 - ╔	217 - ┘	233 - щ	249 - ●
138- К	154 - Ъ	170 - к	186 - ║	202 - ╩	218 - ┌	234 - ъ	250 - .
139 -Л	155 - Ы	171 - л	187 - ╗	203 - ╦	219 - ▉	235 - ы	251 - √
140 -М	156 - Ь	172 - м	188 - ╝	204 - ╠	220 - ▄	236 - ь	252- №
141 -Н	157 - Э	173 - н	189 - ╜	205 - ═	221 - ▌	237 - э	253 - ¤
142 -О	158 - Ю	174 - о	190 - ╛	206 - ╬	222 - ▐	238 - ю	254 - ▪
143- П	159 - Я	175 - п	191 - ┐	207 - ╧	223 - ▀	239 - я	255 -
144 - Р	160 - а	176 - ▒	192 - └	208 - ╨	224 - р	240 - Ё	

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ 3

ВВЕДЕНИЕ 4

Тема 1. ПОНЯТИЕ И ОСНОВНЫЕ СВОЙСТВА АЛГОРИТМА 5
1.1. Понятие алгоритма 5
1.2. Этапы подготовки вычислительных задач для их автоматического решения 6
Вопросы и задания 8

Тема 2. СПОСОБЫ ЗАПИСИ АЛГОРИТМОВ. ТИПОВЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ 9
2.1. Формы записи алгоритмов 9
2.2. Основной набор элементарных предписаний алгоритма 10
2.3. Основные типы простейших алгоритмических структур 12
Вопросы и задания 16

Тема 3. КОМБИНИРОВАННЫЕ АЛГОРИТМЫ 16
3.1. Вложенные циклы 16
3.2. Циклы накопления конечной суммы (или произведения) 18
3.3. Разветвленное тело цикла. Досрочный выход из тела цикла 19
Вопросы и задания 19

Тема 4. ВВЕДЕНИЕ В C++ 20
4.1. Структура программы 20
4.2. Ключевые слова языка C++ 22
4.3. Базовые типы данных 23
4.4. Операции и функции на базовых типах данных C++. Приоритеты и ассоциативность операций 26

4.5. Основы ввода/вывода 35
Вопросы и задания 38

Тема 5. ОПЕРАТОРЫ ЯЗЫКА 39
Вопросы и задания 47

Тема 6. МАССИВЫ 48
6.1. Объявление, определение и работа с массивами 48
6.2. Массив с точки зрения языка C++ (на виртуальном уровне) и с точки зрения хранения 50
6.3. Примеры работы с элементами массива 51
Вопросы и задания 52

Тема 7. СТРОКИ И ОПЕРАЦИИ СО СТРОКАМИ 52
7.1. Объявление строк в программах 52
7.2. Инициализация символьной строки 54
7.3. Функции работы со строками 54
7.4. Работа со строками через класс string 55
Вопросы и задания 59

Тема 8. УКАЗАТЕЛИ И ССЫЛКИ. РАБОТА С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ 59
8.1. Инициализация указателей 60
8.2. Операции с указателями 63
8.3. Ссылки 64
Вопросы и задания 65

Тема 9. СТРУКТУРЫ 66
Вопросы и задания 70

Тема 10. РАБОТА С ФУНКЦИЯМИ 71
10.1. Прототип 72
10.2. Определение функции 73
10.3. Способы передачи параметров в функции 75
10.4. Вызов функции 78
10.5. Параметры функций по умолчанию 78

10.6. Перегрузка функций 79
10.7. Операторные функции. Перегрузка операторов 81
10.8. Шаблоны функций 82
10.9. Встраиваемые функции 84
10.10. Параметры, передаваемые через командную строку 86
10.11. Рекурсия 87
Вопросы и задания 90

Тема 11. ФАЙЛЫ. БАЗОВЫЕ ФУНКЦИИ РАБОТЫ
С ПОТОКАМИ 92
11.1. Понятие физического и логического файлов 92
11.2. Работа с файлами 93
11.3. Текстовые (строковые) файлы 95
11.4. Двоичные файлы 98
Вопросы и задания 101

ЗАКЛЮЧЕНИЕ 102

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ 103

ПРИЛОЖЕНИЕ 104

Учебное издание

ПАВЛОВА Ольга Николаевна

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Редактор Е. С. Глазкова

Технический редактор С. Ш. Абдуллаева

Корректор Н. В. Пустовойтова

Компьютерная верстка Л. В. Макаровой, А. Н. Герасина

Выпускающий редактор А. А. Амирсейидова

Подписано в печать 26.12.19.
Формат 60×84/16. Усл. печ. л. 6,28. Тираж 50 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.