

Министерство образования и науки РФ  
Государственное образовательное учреждение  
высшего профессионального образования  
Владимирский государственный университет

О. Н. МЕДВЕДЕВА

## ПРОГРАММИРОВАНИЕ

Курс лекций

Владимир 2011

УДК 004.42  
ББК 32.973.я73  
М42

Рецензенты:

Кандидат технических наук,  
генеральный директор ООО «ФС Сервис»  
*Д.С. Квасов*

Доктор технических наук, доцент кафедры информационных  
систем и информационного менеджмента  
Владимирского государственного университета  
*Д.В. Александров*

Печатается по решению редакционного совета  
Владимирского государственного университета

**Медведева, О. Н.**

М42 Программирование : курс лекций / О. Н. Медведева ; Вла-  
дим. гос. ун-т. – Владимир : Изд-во Владим. гос. ун-та, 2011. –  
145 с.

ISBN 978-5-9984-0122-0

Сформулированы основные принципы структурного программирования, под-  
робно изложены синтаксис и семантика языка высокоуровневого программирова-  
ния C++, рассмотрены вспомогательные вопросы, касающиеся построения алго-  
ритмов.

Предназначен для студентов первого курса очной формы обучения специаль-  
ности 010503 – математическое обеспечение и администрирование информаци-  
онных систем, изучающих дисциплину «Программирование».

Рекомендован для формирования профессиональных компетенций в соот-  
ветствии с ФГОС 3-го поколения.

Ил. 25. Табл. 7. Библиогр.: 5 назв.

УДК 004.42  
ББК 32.973.я73

ISBN 978-5-9984-0122-0

© Владимирский государственный  
университет, 2011

## ОГЛАВЛЕНИЕ

Предисловие.....	7
Введение.....	8
Лекция 1. ПОНЯТИЕ И ОСНОВНЫЕ СВОЙСТВА АЛГОРИТМА.....	9
§ 1.1. Понятие алгоритма.....	9
§ 1.2. Этапы подготовки вычислительных задач для их автоматического решения.....	10
<i>Вопросы и задания</i> .....	13
Лекция 2. СПОСОБЫ ЗАПИСИ АЛГОРИТМОВ. ТИПОВЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ.....	13
§ 2.1. Формы записи алгоритмов.....	13
§ 2.2. Основной набор элементарных предписаний алгоритма.....	14
§ 2.3. Основные типы простейших алгоритмических структур.....	16
<i>Вопросы и задания</i> .....	21
Лекция 3. КОМБИНИРОВАННЫЕ АЛГОРИТМЫ.....	21
§ 3.1. Вложенные циклы.....	21
§ 3.2. Циклы накопления конечной суммы (или произведения).....	23
§ 3.3. Итерационные циклы.....	25
<i>Вопросы и задания</i> .....	25
Лекция 4. АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ ПО ВРЕМЕНИ ИСПОЛНЕНИЯ И ПАМЯТИ.....	26
§ 4.1. Эффективность работы алгоритма.....	26
§ 4.2. Пример эффективности работы алгоритма.....	27
<i>Вопросы и задания</i> .....	28

Лекция 5. ВВЕДЕНИЕ В C++.....	28
§ 5.1. Структура программы.....	28
§ 5.2. Базовые типы данных.....	31
§ 5.3. Операции и функции на базовых типах данных C++. Приоритеты операций.....	35
§ 5.4. Основы ввода/вывода.....	43
<i>Вопросы и задания</i> .....	48
Лекция 6. ОПЕРАТОРЫ ЯЗЫКА.....	49
<i>Вопросы и задания</i> .....	58
Лекция 7. МАССИВЫ.....	59
§ 7.1. Объявление, определение и работа с массивами .....	59
§ 7.2. Массив с точки зрения языка C++ (на виртуальном уровне) и с точки зрения хранения.....	61
§ 7.3. Примеры работы с элементами массива.....	63
<i>Вопросы и задания</i> .....	63
Лекция 8. СТРОКИ И ОПЕРАЦИИ СО СТРОКАМИ.....	64
§ 8.1. Объявление строк в программах.....	64
§ 8.2. Инициализация символьной строки.....	66
§ 8.3. Функции работы со строками.....	66
<i>Вопросы и задания</i> .....	67
Лекция 9. УКАЗАТЕЛИ И ССЫЛКИ. РАБОТА С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ.....	67
§ 9.1. Понятие указателя.....	67
§ 9.2. Инициализация указателей.....	69
§ 9.3. Операции с указателями.....	71
§ 9.4. Ссылки.....	73
<i>Вопросы и задания</i> .....	75
Лекция 10. СТРУКТУРЫ.....	76
§ 10.1. Работа со структурами.....	76
§ 10.2. Битовые поля.....	80
<i>Вопросы и задания</i> .....	81

Лекция 11. СОСТАВНЫЕ ТИПЫ ДАННЫХ.....	81
§ 11.1. Перечисление.....	81
§ 11.2. Объединения.....	84
<i>Вопросы и задания</i> .....	85
Лекция 12. РАБОТА С ФУНКЦИЯМИ.....	86
§ 12.1. Прототип.....	86
§ 12.2. Определение функции.....	88
§ 12.3. Способы передачи параметров в функции.....	90
§ 12.4. Вызов функции.....	92
§ 12.5. Параметры функций по умолчанию.....	93
§ 12.6. Перегрузка функций.....	94
§ 12.7. Операторные функции. Перегрузка операторов.....	96
§ 12.8. Шаблоны функций.....	97
§ 12.9. Встраиваемые функции.....	98
§ 12.10. Параметры, передаваемые через командную строку.....	101
§ 12.11. Рекурсия.....	102
<i>Вопросы и задания</i> .....	105
Лекция 13. ФАЙЛЫ. БАЗОВЫЕ ФУНКЦИИ РАБОТЫ С ПОТОКАМИ.....	107
§ 13.1. Понятие физического и логического файлов.....	107
§ 13.2. Работа с файлами.....	108
§ 13.3. Текстовые (строковые) файлы.....	110
§ 13.4. Двоичные файлы.....	113
<i>Вопросы и задания</i> .....	117
Лекция 14. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ.....	118
§ 14.1. Пространство имен в C++.....	117
§ 14.2. Область видимости и время жизни переменных. Класс памяти.....	120

§ 14.3. Обработка исключительных ситуаций в C++.....	124
<i>Вопросы и задания</i> .....	127
Лекция 15. СТРУКТУРЫ КОМПЬЮТЕРНОЙ ОБРАБОТКИ ДАННЫХ.....	128
§ 15.1. Связные списки.....	128
§ 15.2. Табличное хранение.....	137
<i>Вопросы и задания</i> .....	142
Заключение.....	143
Библиографический список.....	144

## ПРЕДИСЛОВИЕ

В наше время, когда наблюдается активный спрос на специалистов в области информатики, учебные заведения испытывают сильное давление со стороны работодателей, желающих видеть выпускников, обладающих какими-либо конкретными навыками. С одной стороны, стремление выпускать профессионалов с необходимыми для рынка труда навыками, несомненно, является положительным. С другой стороны, необходимо иметь в виду, что для студентов эффективная подготовка заключается не в овладении специфическими навыками, которые могут быстро устареть, а в получении прочных теоретических и практических знаний, которые позволят им продолжительное время оставаться на современном уровне. Лучше всего данный аспект студенческой подготовки может быть сформулирован следующим образом: работодатели и сами студенты должны рассматривать выпускников в области информатики как специалистов, обладающих навыками, которые будут актуальны для организации долгое время.

Для того чтобы направить студентов на этот путь, учебный курс должен содействовать развитию набора универсальных умений, повышающих профессионализм выпускников.

Существует большое количество вариантов учебных программ в различных университетах и колледжах. При разработке учебно-методического комплекса, в состав которого входит курс лекций, учтены, прежде всего, государственные образовательные стандарты подготовки специалистов по специальности 010503 – математическое обеспечение и администрирование информационных систем, а также рекомендации Computing Curricula группы IEEE Computing Society/ACM.

## ВВЕДЕНИЕ

На современном этапе информатизации общества актуальна подготовка квалифицированных кадров в области информационных технологий. При этом ставятся задачи подготовки не только пользователей ПК, но и разработчиков программного обеспечения.

Основной упор в курсе делается на изучение вопросов и проблем, возникающих при использовании ЭВМ для автоматизации решения типовых инженерных задач. А это связано с понятием алгоритма, процессом его разработки как ядром решения какой-либо проблемы, типовыми приемами построения, формами представления. Практика обучения программированию показывает, что запись готового алгоритма на языке программирования такой же творческий процесс, как и построение самого алгоритма из его типовых конструкций. Отсюда возникает необходимость свободного владения алгоритмическим языком, в качестве которого выбран C++. Главная причина выбора языка – C++ поддерживает современную методику и технологию программирования, он удобен для обучения программированию вообще, т.к. изначально создавался именно для этой цели.

В результате изучения дисциплины студенты должны получить навыки разработки и программирования алгоритмов решения различных задач с последующей работой на IBM-совместимом ПК в некоторой среде программирования. Курс связан с последующими дисциплинами типа «Объектно-ориентированное программирование», «Дискретная математика», «Вычислительная математика», поэтому, с одной стороны, он должен быть базовым для них, а с другой – вопросы, подробно рассматриваемые в них, включены в настоящее издание в минимальном объеме или вообще не включены.



# Лекция 1. ПОНЯТИЕ И ОСНОВНЫЕ СВОЙСТВА АЛГОРИТМА

## § 1.1. Понятие алгоритма

Слово «*алгоритм*» произошло от имени узбекского 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», позднее – «алгоризм». Впоследствии произошло переименование термина в «алгоритм» как собирательное название правил определенного вида (не только арифметических действий) для выполнения какой-либо работы. Но 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 явилось 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 неё.

Сейчас алгоритмизацию (построение алгоритмов) 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 говоря, 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: автомат, робот, ЭВМ 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 растут с увеличением сложности задачи, тем более что писать необходимо только с помощью набора 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 ЭВМ. В последовательности этапов и в их содержании заключается то важное, что позволит решать не только учебные вычислительные задачи, но и сложные большие практические, с которыми сталкиваются специалисты в лю-

бой сфере профессиональной деятельности. Нам необходимо лишь понять и усвоить ее.

В литературных источниках необходимые мероприятия могут группироваться или, напротив, дробиться по-разному, образуя разное количество этапов. Принципиально лишь то, что все обязательные мероприятия надо выполнить в нужной взаимосвязи друг с другом. Последовательность этапов решения задачи на ЭВМ такова:

1. Постановка задачи (полная и исчерпывающая) – 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 задач). 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 – ММ) задачи или каждой подзадачи в случае ее разбиения. Выполняется замена всех физических параметров условными математическими обозначениями (по возможности передающими смысл параметра), составление таблицы идентификаторов, содержащей более полную информацию об этих объектах с указанием всех требований по форматам данных и результатов. Затем предполагается связь входных параметров 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 ММ, что очень важно для успеха всей последующей работы.

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

5. Составление алгоритма по выбранному численному методу (или по ММ, если п. 4 не выполнялся). На этом этапе, когда алгоритм только разрабатывается, его составляют в текстовой форме или графической – 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 высокого уровня (ЯВУ). По сути, эта работа сводится к переводу каждого пункта алгоритма с одного языка представления на другой. Иногда, конечно, оказывается целесообразным использовать более сложные конструкции ЯВУ, покрывающие работу сразу нескольких элементарных предписаний алгоритма. Здесь требуется хорошее знание этого ЯВУ.

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 «вручную» можно было вычислить результаты, используя лишь п. 1. Возможно, такие данные не будут иметь физического смысла для задачи, но зато заранее и просто вычисленный по ним результат может быть использован 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 ММ (пп. 2, 3), ошибки при реализации метода (п. 4), ошибки в алгоритме (п. 5), ошибки при переводе на алгоритмический язык (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 для решения с вводом исходных данных для всех входных параметров, заданных в постановке задачи (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 совершить при несовпадении контрольного решения задачи?
5. В чем преимущества формирования библиотек программ?

## **Лекция 2. СПОСОБЫ ЗАПИСИ АЛГОРИТМОВ. ТИПОВЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ**

### **§ 2.1. Формы записи алгоритмов**

Разработка алгоритма решения задачи выполняется сначала на языке, близком и понятном его разработчикам. Форма записи алгоритма определяется как раз языком, на котором записаны все его предписания. Нами будут рассмотрены три основные формы, две из которых могут быть использованы на предварительном этапе обдумывания и составления алгоритма, а одна, как заключительная, используется 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, поэтому под сомнение ставится свойство определенности алгоритма. Зато для записи в этой форме не нужны никакие специальные знания кроме знаний 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;
- наглядность алгоритма 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» разработанного алгоритма в любой из перечисленных выше форм на алгоритмический язык: переводится (заменяется) последовательно каждое предписание «своим» оператором. Для простых задач и при хорошем знании языка программирования возможно написание программы сразу, без промежуточных форм.

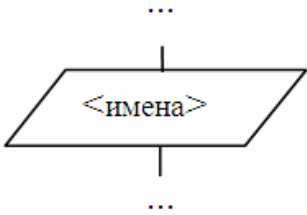
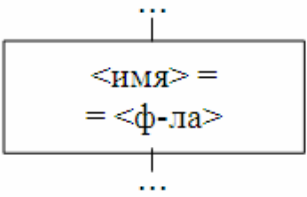
## § 2.2. Основной набор элементарных предписаний алгоритма

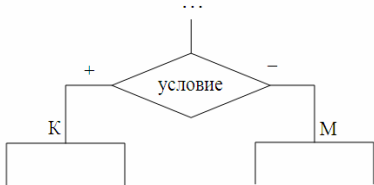
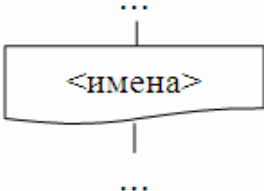
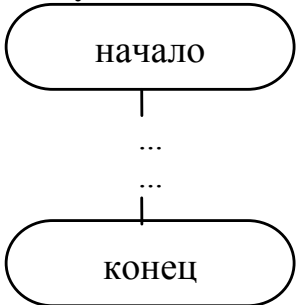
Все многообразие вычислительных задач может быть алгоритмически описано с использованием конечного *набора элементарных предписаний*. Элементарным предписанием называется такой приказ алгоритма, который однозначно понимается и выполняется исполнителем этого алгоритма. При алгоритмизации решения вычислительных задач для ЭВМ принято использовать основной набор, включающий всего пять типов элементарных предписаний.

Этот набор достаточен, какой бы сложности вычислительная задача не была, и имеет соответствующие операторы-аналоги в лю-

бом языке программирования высокого уровня. Представим этот перечень приказов (с необходимыми пояснениями) в двух формах: текстовой (на русском языке) и графической (блоками) (таблица). 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 (высота) обычно задают в 1,5 – 2 раза меньше размера по горизонтали (длина). Исключение составляет блок «пуск/остановка», вертикальный размер которого в четыре раза меньше габаритной длины, поэтому он легко отличается от других блоков и может не заполняться.

### Элементы блок-схемы

Форма		Пояснения
Текстовая	Графическая	
Ввести (значения для): <имена входных переменных>	<b>Блок-ввода</b> 	После ключевого слова или внутри блока указывается список имен входных переменных через запятую, но не сами числа
Вычислить: <имя> = <формула>	<b>Блок-процесс</b> 	После ключевого слова или внутри блока записывается имя вычисляемой переменной, знак = (присваивания) и формула, содержащая переменные, числа, арифметические операции, элементарные функции

Форма		Пояснения
Текстовая	Графическая	
Сравнить: если <условие>, то п. К иначе п. М	<b>Блок-принятие решения</b> 	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)
<b>Вывести</b> (напечатать): <имена выходных переменных>	<b>Блок-документ</b> 	После ключевого слова или внутри блока указывается список имен выходных переменных через запятую, но не сами числа
<b>Начало</b> решения ... ...  <b>Остановка</b> (конец решения)	<b>Блоки пуск/остановка</b> 	Если 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

Среди этапов подготовки задач к автоматическому решению ключевое место занимает этап построения собственно алгоритма. Любой начинающий программист в качестве основного элемента подготовки должен научиться строить алгоритмы решения вычислительных задач. Для их построения требуется минимум пять типов элементарных операций. Опыт показывает, что все многообразие вычислительных задач при их решении можно описать также ограниченным набором сочетаний этих элементарных предписа-



ний в более крупные вычислительные структуры. Постепенно, расчленяя задачи на подзадачи, сводим их решение к некоторым типовым фрагментам алгоритмов.

Можно указать три основных типа структур вычислительных схем, сочетая которые составляют алгоритмы любой сложности. Тип вычислительной схемы, или тип алгоритма, определяется исходя из порядка исполнения всех его предписаний при выполнении алгоритма по конкретным данным задачи. В соответствии с этим он получает и свое название.

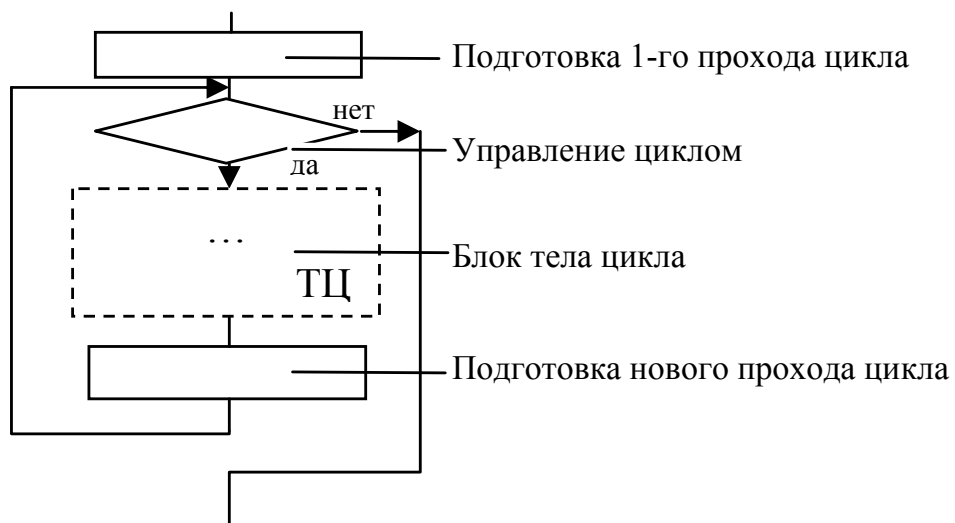
1. **Линейный тип алгоритма** – это такая вычислительная структура, при которой все предписания выполняются в строго линейной последовательности друг за другом, как записаны; сам порядок исполнения называется естественным. В чистом виде линейные алгоритмы встречаются редко, они чрезвычайно просты, но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

2. **Разветвляющийся тип алгоритма** – это такая вычислительная схема, которая содержит не одну, а несколько возможных ветвей решения по ММ, 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. Она выбирается автоматически, только в процессе исполнения алгоритма (остальные ветви для этого варианта данных останутся не востребованными). Следовательно, пользователю труднее заранее (на этапе разработки) предусмотреть все возможные ветви решения и изобразить их, ничего не упустив.

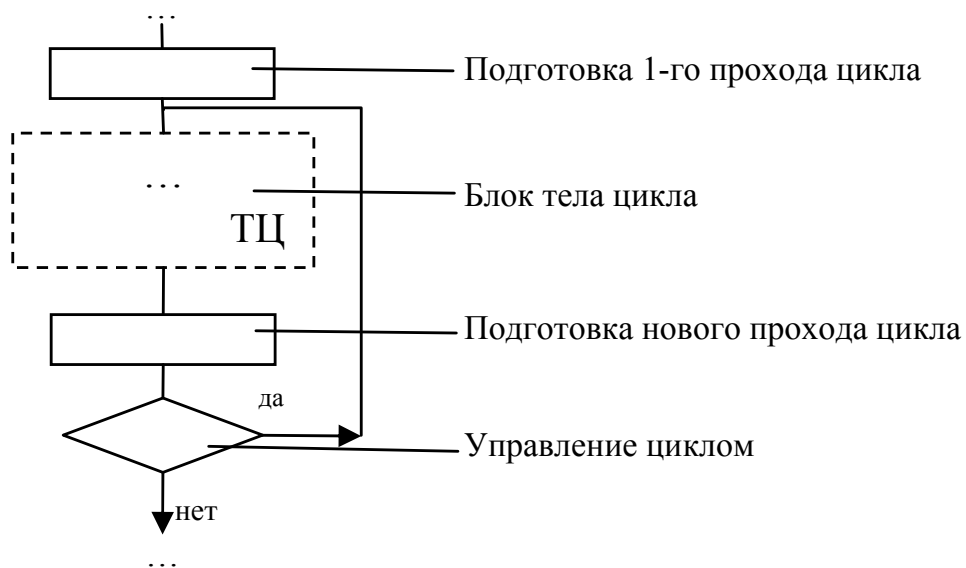
3. **Циклический тип алгоритма** – это такая схема разветвленной структуры, в которой одна ветвь операции сравнения является обратной связью (ОС) на предыдущую часть алгоритма (т.е. идет назад). Таким образом, некоторая последовательность операций алгоритма будет выполняться многократно (т.е. в цикле),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

- а) надо организовать данные для первого прохождения ТЦ;
- б) после каждого выполнения ТЦ надо обновлять данные для очередного прохождения цикла;
- в) надо управлять циклом, организовать условие выхода из цикла или его продолжения (ОС не должна охватывать указанные в п. а операции!).

В зависимости от конкретных заданий 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**

В первом случае (см. рис. 2.1) блок управления стоит перед телом цикла (цикл с предусловием), его используют для задач, требующих проверки и условия выполнения цикла перед входением в него (цикл может вообще ни разу не выполниться). Во втором случае (см. 2.2) блок управления после тела цикла (цикл с постусловием) используется 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:  $x_n$  – начальное значение  $x$ ;  $x_k$  – конечное значение  $x$ ;  $dx$  – 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):

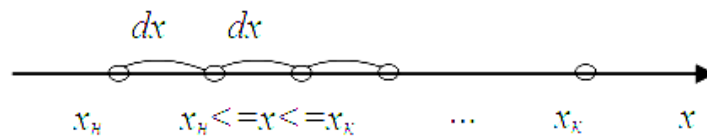


Рис. 2.3. Задание исходных данных (способ 1)

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

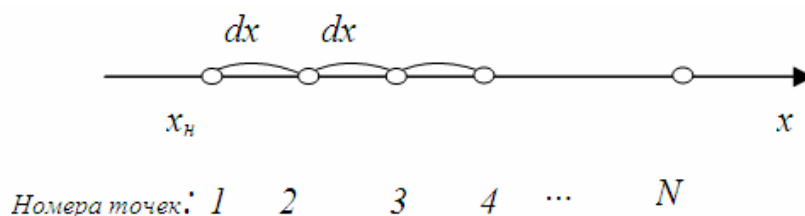


Рис. 2.4. Задание исходных данных (способ 2)

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

удаётся. В данном случае удобно иметь в распоряжении все заданные значения  $x$ :  $X = (x_1, x_2, \dots, x_n)$ . В программировании такой вариант задания аргумента называется числовым массивом, а каждое отдельное значение в нём – элементом массива (рис. 2.5).

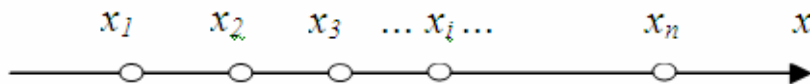


Рис. 2.5. Задание исходных данных (способ 3)

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

В современных ЯВУ 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 опера-

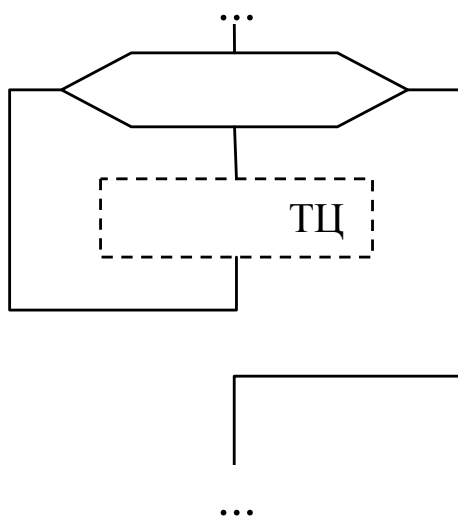


Рис. 2.6. Блок-цикл

торе цикла (обычно в языке их несколько видов, и нужный обязательно найдётся), который ставят перед операторами ТЦ. Это способствует приближению текста программы к логике человеческого (а не машинного) мышления, ускоряя и облегчая процесс программирования. Для того чтобы приблизиться к выбранному языку программирования, полнее использовать его возможности, в литературе по алгоритмизации вычислений стали использовать специальный блок, который на-

зывают блоком цикла, или блок-модификатором, или блок-заголовком цикла (рис. 2.6).

Этот блок имеет специальную структуру, предшествует телу цикла и размещает в себе все дополнительные операции по организации цикла. Переменная, управляющая циклом, называется параметром цикла. В блоке цикла указываются параметр цикла, знак присваивания ( $=$ ), начальное значение параметра, конечное значе-

ние параметра, шаг изменения параметра (например,  $i = 1, n, 1$ ). Некоторые ЯВУ 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.

## Лекция 3. КОМБИНИРОВАННЫЕ АЛГОРИТМЫ

### § 3.1. Вложенные циклы

Вложенные циклы (цикл в цикле) – это такие циклические алгоритмы, которые имеют несколько переменных (более одной), 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.

Исходные данные для  $x$  и  $a$ :

$x_n$  – начальное значение  $x$ ;

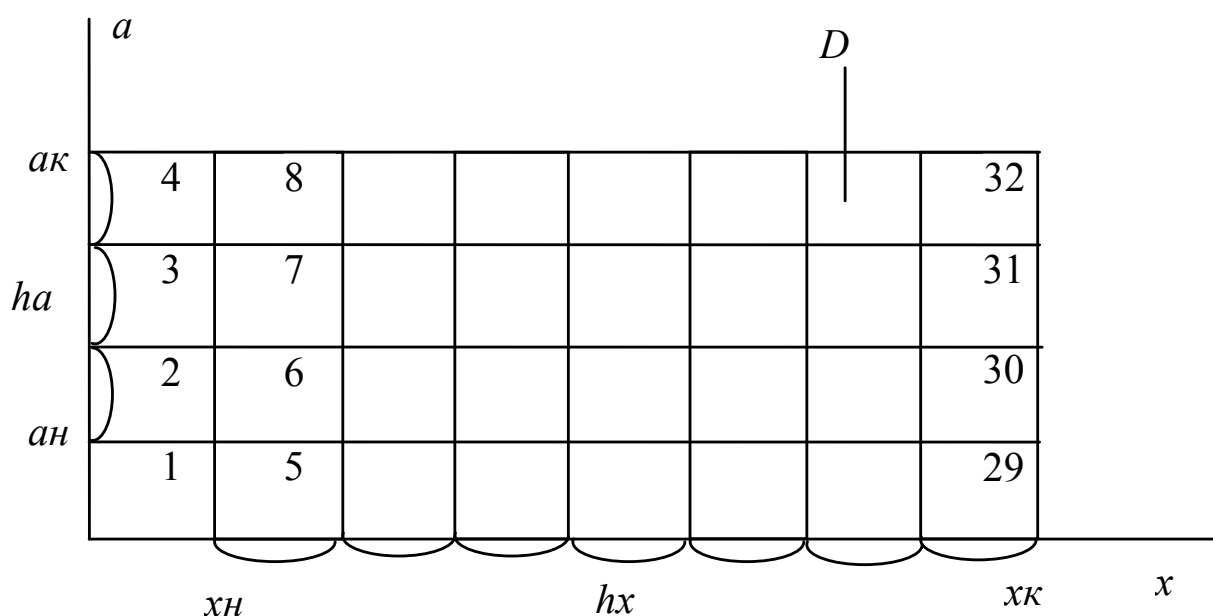
$a_n$  – начальное значение  $a$ ;

$x_k$  – конечное значение  $x$ ;  $a_k$  – 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 важно предварительно определить, какой аргумент образует внутренний цикл, а 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).

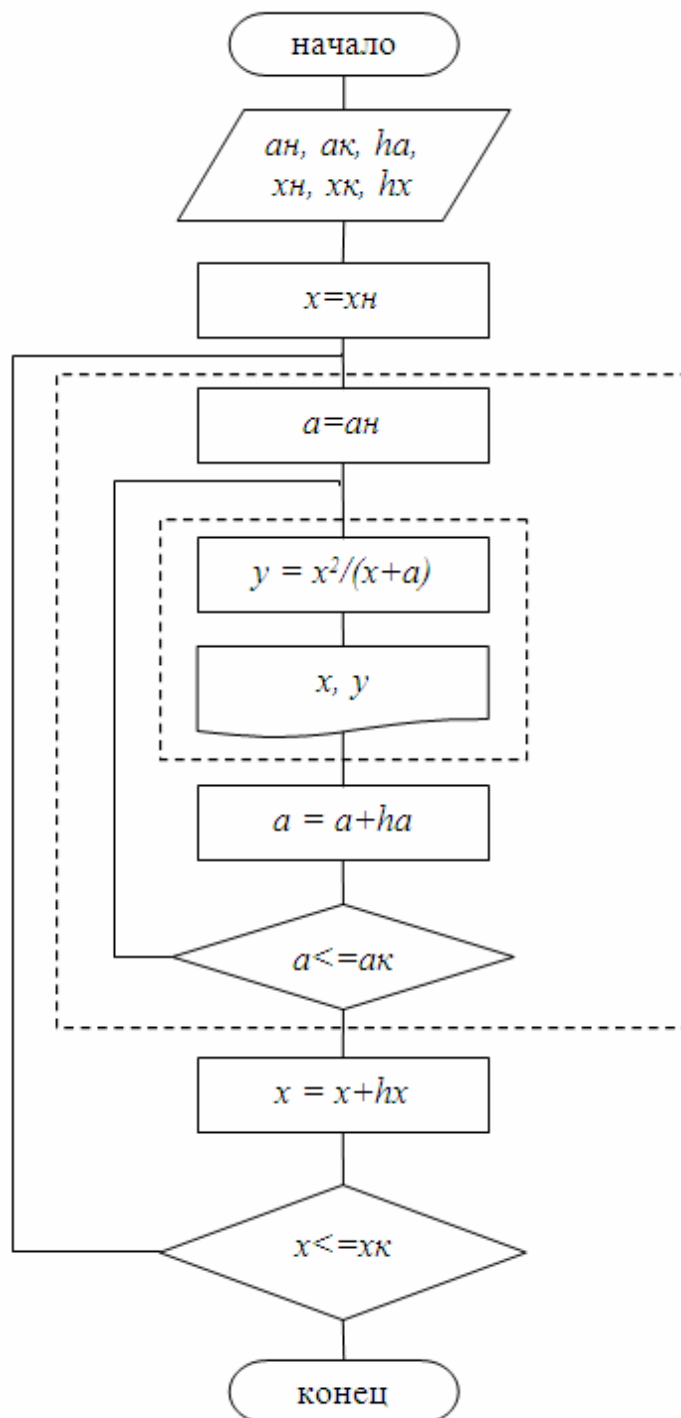


Рис. 3.2. Блок-схема алгоритма со вложенным циклом

### § 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:

$$\sum_{i=1}^n (\langle i\text{-е слагаемое} \rangle) \text{ или } \prod_{i=1}^n (\langle i\text{-й множитель} \rangle),$$

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,

$$S = \sum_{i=1}^{50} x_i^2 = x_1^2 + x_2^2 + x_3^2 + \dots + x_i^2 + \dots + x_{50}^2 \text{ или}$$

$$P = \prod_{i=1}^{40} \frac{x_i^2}{2} = \frac{x_1^2}{2} \cdot \frac{x_2^2}{2} \cdot \frac{x_3^2}{2} \cdot \dots \cdot \frac{x_i^2}{2} \cdot \dots \cdot \frac{x_{40}^2}{2}.$$

В алгоритмах знаки макроопераций  $\sum$ ,  $\prod$  не употребляются (не считаются элементарными). В развернутом виде формулу при записи алгоритма использовать нежелательно из-за громоздкости, а заменять часть операндов по формуле многоточием – не допустимо, поэтому 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 вводится дополнительная переменная «накопления»  $S$ , которая предварительно (перед телом цикла) обнуляется при накоплении суммы или получает значение «единица» при накоплении произведения. При  $S$  операция «накопления»: для суммы равна  $S + \langle i\text{-е слагаемое} \rangle$ ; для произведения –  $P \cdot \langle i\text{-й множитель} \rangle$ .



### § 3.3. Итерационные циклы

Существует множество практических задач, которые требуют циклических вычислений, но заранее неизвестно ни количество этих вычислений, ни конечное значение переменной цикла, т.е. воспользоваться рассмотренными ранее способами выхода из цикла нельзя. Эти циклы характеризуются последовательным приближением к искомой величине с заданной точностью, и для автоматического прекращения циклических вычислений используется дополнительное условие задач – точность решения. Примером таких вычислений могут служить итерационные методы нахождения, например, приближенного решения алгебраического уравнения или аналогичной системы уравнений. Алгоритмически они вырождаются в так называемые итерационные циклы, в основе которых лежит итерационная (или рекуррентная) формула, связывающая «новое» значение искомой величины ( $U_{\text{нов}}$ ) с ее старым значением ( $U_{\text{стар}}$ ), вычисленным на предыдущем шаге (в предыдущем выполнении цикла):

$$U_{\text{нов}} = f(U_{\text{стар}}).$$

Последовательное уточнение решения выполняется до тех пор, пока не будет достигнута заданная точность  $\varepsilon$ :

$$|U_{\text{нов}} - U_{\text{стар}}| \leq \varepsilon.$$

#### 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 является цикл с досрочным выходом из цикла.
4. Для чего применяют итерационные алгоритмы?

## Лекция 4. АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ ПО ВРЕМЕНИ ИСПОЛНЕНИЯ И ПАМЯТИ

Предположим, что быстродействие компьютера и объем его памяти можно увеличивать до бесконечности. Была бы в таком случае необходимость в изучении алгоритмов? Была бы, но только для того, чтобы продемонстрировать, что метод решения имеет конечное время и что он дает правильный ответ.

Если бы компьютеры были неограниченно быстрыми, подошел бы любой корректный метод решения задачи. Возможно, вы бы предпочли, чтобы реализация решения была выдержана в хороших традициях программирования (т.е. качественно разработана и аккуратно занесена в документацию), но чаще всего выбирался бы метод, который легче всего реализовать.

Конечно же, сегодня есть весьма производительные компьютеры, но их быстродействие не может быть бесконечно большим. Память тоже дешевет, но она не может быть бесплатной. Таким образом, время вычисления – это такой же ограниченный ресурс, как и объем необходимой памяти. Этими ресурсами следует распоряжаться разумно, чему и способствует применение алгоритмов, эффективных в плане расходов времени и памяти.

### § 4.1. Эффективность работы алгоритма

Алгоритмы, разработанные для решения одной и той же задачи, часто очень сильно различаются по эффективности. Эти различия могут быть намного значительнее, чем те, что вызваны применением неодинакового аппаратного и программного обеспечения.

В качестве примера можно привести два алгоритма сортировки. Для выполнения первого из них, известного как сортировка вставкой, требуется время, которое оценивается как  $c_1 n^2$ , где  $n$  – количество сортируемых элементов, а  $c_1$  – константа, не зависящая от  $n$ . Таким образом, время работы этого алгоритма приблизительно пропорционально  $n^2$ . Для выполнения второго алгоритма, сортировка слиянием, требуется время, приблизительно равное  $c_2 n \lg n$ , где  $\lg n$  – краткая

запись  $\log_2 n$ , а  $c_2$  – некоторая другая константа, не зависящая от  $n$ . Обычно константа метода вставок меньше константы метода слияния, т.е.  $c_1 < c_2$ . Убедимся, что постоянные множители намного меньше влияют на время работы алгоритма, чем множители, зависящие от  $n$ . Для двух приведенных методов последние относятся как  $\lg n$  и  $n$ . Для небольшого количества сортируемых элементов сортировка включением обычно работает быстрее, однако когда  $n$  становится достаточно большим, все заметнее проявляется преимущество сортировки слиянием, возникающее благодаря тому, что для больших  $n$  незначительная величина  $\lg n$  по сравнению с  $n$  полностью компенсирует разницу величин постоянных множителей. Не имеет значения, во сколько раз константа  $c_1$  меньше, чем  $c_2$ . С ростом количества сортируемых элементов обязательно будет достигнут переломный момент, когда сортировка слиянием окажется более производительной.

#### § 4.2. Пример эффективности работы алгоритма

В качестве примера рассмотрим два компьютера – А и Б. Компьютер А более быстрый, и на нем работает алгоритм сортировки вставкой, а компьютер Б более медленный, и на нем работает алгоритм сортировки слиянием. Предположим, что компьютер А выполняет миллиард инструкций в секунду, а компьютер Б – лишь десять миллионов. Таким образом, компьютер А работает в 100 раз быстрее, чем компьютер Б. Для того чтобы различие стало еще большим, предположим, что код для метода вставок написан самым лучшим программистом с использованием команд процессора и для сортировки  $n$  чисел надо выполнить  $2n^2$  команд (т.е.  $c_1 = 2$ ). Сортировка методом слияния реализована программистом среднего уровня с помощью языка высокого уровня. При этом компилятор оказался не слишком эффективным, и в результате получился код, требующий выполнения  $50n \lg n$  команд (т.е.  $c_2 = 50$ ). Для сортировки миллиона чисел компьютеру А понадобится  $\frac{2 \cdot (10^6)^2 \text{ команд}}{10^9 \text{ команд/с}} = 2000 \text{ с}$ , а компьютеру Б –  $\frac{50 \cdot 10^6 \cdot \lg 10^6 \text{ команд}}{10^7 \text{ команд/с}} \approx 100 \text{ с}$ .

Как видно, использование кода, время работы которого возрастает медленнее, даже при плохом компиляторе на более медленном компьютере требует на порядок меньше процессорного времени. Если же нужно выполнить сортировку 10 миллионов чисел, то преимущество метода слияния становится еще более очевидным: если для сортировки вставкой потребуется приблизительно 2,3 дня, то для сортировки слиянием – меньше 20 минут. Таким образом, можно сделать следующий вывод: чем больше количество сортируемых элементов, тем заметнее преимущество сортировки слиянием.

### Вопросы и задания

1. По каким параметрам осуществляется оценка эффективности работы алгоритма?
2. Как оценить время работы алгоритма?
3. С помощью какого показателя можно оценить быстродействие вычислительного устройства?

## Лекция 5. ВВЕДЕНИЕ В C++

### § 5.1. Структура программы

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

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

При необходимости можно с помощью директив препроцессора определить для ключевых слов другие имена. Например, при наличии в программе макроопределения `#define BOOL int` слово *BOOL* можно использовать в объявлениях вместо слова *int*. Смысл

объявлений (спецификация целого типа данных) от этого не изменится, однако программа станет более читабельной, если речь идет не просто о целых переменных, а о переменных, предназначенных для хранения значений булевского типа (булевский тип не реализован в языке C++ как самостоятельный тип данных).

В языке C++ различаются верхний и нижний регистры символов: *else* – ключевое слово, а *ELSE* – нет. В программе ключевое слово может быть использовано только как ключевое, то есть никогда не допускается его использование в качестве имени переменной или функции.

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

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

Указания компилятору – это специальные инструкции, которым компилятор языка C++ следует во время компиляции.

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

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

Определение функции специфицирует тело функции, которое представляет собой составной оператор (блок), содержащий объявления и операторы. Определение функции также задает имя функции, 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 существенен,

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 определяет действия, выполняемые программой. Если же программа содержит несколько функций, то среди них выделяется одна главная функция, которая должна иметь имя *main*. С нее начинается выполнение программы; она определяет действия, выполняемые программой, и вызывает другие функции. Порядок следования определений функций в исходной программе несущественен.

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 f2 (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
подключения заголовочных файлов */
```

```

int x = 1; /* определения переменных */
int y = 2;
extern int printf(char *, ...); /* объявление функции */
void main () /* определение главной функции */
{
    int z; /* объявления переменных */
    int w;
    z = y + x; /* выполняемые операторы */
    w = y - x;
    printf("z = %d\nw = %d\n", z, w);
}

```

Эта исходная программа определяет функцию с именем *main* и объявляет функцию *printf*. Переменные *x* и *y* определяются на внешнем уровне, а переменные *z* и *w* объявляются внутри функции.

## § 5.2. 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 4.1) *int* занимает 16 бит, а в 32-разрядной (Windows 95/98/NT/2000) – 32 бита. Однако полностью полагаться на это нельзя, особенно при переносе программы в другую среду. Необходимо помнить, что стандарт C++ обуславливает только *минимальный диапазон значений* каждого типа данных, но не размер в байтах.

Конкретный формат числа с плавающей точкой зависит от его реализации в трансляторе. Переменные типа *char* обычно используются для обозначения набора символов стандарта ASCII, симво-

лы, не входящие в этот набор, разными 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*. (Стандарт C99 также позволяет модифицировать *long* с помощью *long*, создавая, таким образом, *long long*). В табл. 5.1 приведены все допустимые комбинации типов данных с их минимальным диапазоном значений и типичным размером. Обратите внимание, что в таблице приведены *минимально возможные*, а не типичные диапазоны значений. Например, если в компьютере арифметические операции выполняются над числами в дополнительных кодах (а именно так спроектированы почти все компьютеры), то в диапазон значений целых попадут все целые числа от  $-32767$  до  $32768$ .

Типы данных языка C++

Табл. 5.1

Тип	Типичный размер		Минимально допустимый размер значений
	бит	байт	
<i>char</i>	8	1	от $-128$ до $127$
<i>unsigned char</i>	8	1	от $0$ до $255$



Окончание табл. 5.1

Тип	Типичный размер		Минимально допустимый размер значений
	бит	байт	
<i>signed char</i>	8	1	от -128 до 127
<i>int</i>	16 или 32	2 или 4	от -32768 до 32767
<i>unsigned int</i>	16 или 32	2 или 4	от 0 до 65535
<i>signed int</i>	16 или 32	2 или 4	то же, что <i>int</i>
<i>short int</i>	16	2	от -32768 до 32767
<i>unsigned short int</i>	16	2	от 0 до 65535
<i>signed short int</i>	16	2	то же, что <i>short int</i>
<i>long int</i>	32	4	от -2 147 483 648 до 2 147 483 647
<i>long long int</i>	64	8	от $-2^{63}$ до $(2^{63}-1)$ , добавлен стандартом C99
<i>signed long int</i>	32	4	то же, что <i>long int</i>
<i>unsigned long int</i>	32	4	от 0 до 4 294 967 295
<i>unsigned long long int</i>	64	8	от 0 до $(2^{64}-1)$ , добавлен в C99
<i>float</i>	32	4	от $1E-37$ до $1E+37$ , с точностью не менее 6 значащих десятичных цифр
<i>double</i>	64	8	от $1E-37$ до $1E+37$ , с точностью не менее 10 значащих десятичных цифр
<i>long double</i>	80	10	от $1E-37$ до $1E+37$ , с точностью не менее 10 значащих десятичных цифр

Для целых можно использовать спецификатор *signed*, но в этом нет необходимости, потому что при объявлении целого он предполагается по умолчанию. Спецификатор *signed* чаще всего исполь-

зуется типом *char*, который в некоторых реализациях по умолчанию может быть беззнаковым.

Целые числа со знаком и без него 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, а 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 32767 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 65535.

Если спецификатор типа записать сам по себе (без следующего за ним базового типа), то предполагается, что он модифицирует тип *int*. Таким образом, следующие спецификаторы типов эквивалентны:

<i>signed</i>	↔	<i>signed int</i>
<i>unsigned</i>	↔	<i>unsigned int</i>
<i>long</i>	↔	<i>long int</i>
<i>short</i>	↔	<i>short int</i>

Хотя базовый тип *int* и предполагается по умолчанию, его, тем не менее, обычно указывают явно.

## § 5.3. Операции и функции на базовых типах данных C++.

### Приоритеты операций

В языке C++ операции с высшими приоритетами вычисляются первыми. Наивысшим приоритетом является приоритет, равный единице. Приоритеты и порядок операций приведены в табл. 5.2.

*Приоритеты операций*

Табл. 5.2

Приоритет	Знак операции	Тип операции	Порядок выполнения
1	() [] . ->	Выражение	Слева направо
2	- ~ ! * & ++ -- <i>sizeof</i> (приведение типов)	Унарный	Справа налево
3	* / %	Мультипликативные	Слева направо
4	+ -	Аддитивные	
5	<< >>	Сдвиг	
6	< > <= >=	Отношение	
7	== !=	Отношение (равенство)	
8	&	Поразрядное И	
9	^	Поразрядное исключающее ИЛИ	
10		Поразрядное ИЛИ	
11	&&	Логическое И	
12		Логическое ИЛИ	
13	? :	Условная	
14	= *= /= %= += -= &=  = >>= <<= ^=	Простое и составное присваивания	Справа налево
15	,	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

дов операций умножения и деления могут отличаться, и для них справедливы правила преобразования типов. Типом результата является тип операндов после преобразования (преобразование происходит к типу, имеющему больший размер).

Операция умножения (\*) выполняет умножение операндов.

Пример:

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

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

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 «остаток 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 они не обес-

печивают 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:

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

В результате сложения  $k$  получит значение, равное  $-5536$ .

Результат выполнения операции сложения – 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 является 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 являются указателями одного типа.
3. Первый операнд – указатель, а второй – целое.

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

Пример:

```
double d[10], *u;  
int i;  
u = d + 2; //u указывает на третий элемент массива  
i = u - d; /* i принимает значение равное 2 */
```

### ***Операции сдвига***

Операции сдвига осуществляют смещение операнда влево ( $\ll$ ) или вправо ( $\gg$ ) на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами. При этом выполня-

ются обычные арифметические преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в ноль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип *unsigned*, то свободные левые биты устанавливаются в ноль. 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 результат операции сдвига не может быть представлен типом первого операнда после преобразования.

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:

```
int 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 логического И (&), операция поразрядного логического 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.

Операция поразрядного И (&) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита – единицы, то соответствующий бит результата устанавливается в единицу, в противном случае – в ноль.

Операция поразрядного ИЛИ (|) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой (или оба) из сравниваемых битов равен единице, то соот-

ветствующий бит результата устанавливается в единицу, в противном случае результирующий бит равен нулю.

Операция поразрядного исключающего ИЛИ (^) сравнивает каждый бит первого операнда с соответствующими битами второго операнда. Если один из сравниваемых битов равен нулю, а второй бит – единице, то соответствующий бит результата устанавливается в единицу, в противном случае, т.е. когда оба бита равны единице или нулю, бит результата устанавливается в ноль.

Пример:

```
int 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, который 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 (!) вырабатывает значение ноль, если операнд есть истина (не ноль), 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  $\sim f$  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 может быть использовано имя типа *void*, а идентификатор не может относиться к полю битов или быть именем функции.

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

Когда *sizeof* применяются к имени типа структуры или объединения или к идентификатору, имеющему тип структуры или объединения, то результатом является фактический размер структуры или объединения, который может включать участки памяти, используемые для выравнивания элементов структуры или объединения. Таким образом, этот результат может не соответствовать размеру, получаемому путем сложения размеров элементов структуры.

Пример:

```
struct  
{  
    char h;  
    int b;  
    double f;  
} str;  
int a1;  
a1 = sizeof(str);
```



Переменная *a1* получит значение, равное 12, в то же время если сложить длины всех используемых в структуре типов, то получим, что длина структуры *str* равна 7.

Несоответствие возникает из-за того, что после размещения в памяти первой переменной *h* длиной один байт добавляется один байт для выравнивания адреса переменной *b* на границу слова (слово имеет длину два байта для машин серии IBM PC AT /286/287), далее осуществляется выравнивание адреса переменной *f* на границу двойного слова (четыре байта), таким образом в результате операций выравнивания для размещения структуры в оперативной памяти требуется на пять байт больше.

В связи с этим целесообразно рекомендовать при объявлении структур и объединений располагать их элементы в порядке убывания длины типов, т.е. приведенную выше структуру следует записать в следующем виде:

```
struct
{
    double f;
    int b;
    char h;
} str;
```

### ***Логические операции***

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

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

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

Операция логического И (&&) вырабатывает значение «единица», 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 операнд не вычисляется.

Операция логического ИЛИ ( $\parallel$ ) вырабатывает значение «ноль», если оба операнда имеют значение ноль; если какой-либо из операндов имеет ненулевое значение, то результат операции равен единице. Если первый операнд имеет ненулевое значение, то второй операнд 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 вычисляет 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 или знака операции.

### ***Операции увеличения и уменьшения***

Операции увеличения (++) и уменьшения (--) – унарные операции присваивания. 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 увеличивает (уменьшается) на единицу. Тип результата соответствует типу операнда. Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует. В языке допускается префиксная или постфиксная формы операций увеличения (уменьшения), поэтому значение выражения, использующего операции увеличения (уменьшения), зависит от того, какая из форм указанных операций используется.

Если знак операции стоит перед операндом (префиксная форма записи), то изменение операнда происходит до его использова-

ния в выражении и результатом операции является увеличенное или уменьшенное значение операнда.

В том случае, если знак операции стоит после операнда (постфиксная форма записи), то операнд вначале используется 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  $s = 3, f = 1$ .

В случае, если операции увеличения и уменьшения используются как самостоятельные операторы, префиксная и постфиксная формы записи становятся эквивалентными:

```
z++; /* эквивалентно */ ++z;
```

## § 5.4. 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!”.

В функции *printf()*, как видно из табл. 5.3, имеется широкий набор спецификаторов преобразования.

*Спецификаторы вывода данных*

Табл. 5.3

Код	Формат
%a	Шестнадцатеричное число в виде 0xh.hhhhP+d (только C99)
%A	Шестнадцатеричное число в виде 0xh.hhhhP+d (только C99)
%c	Символ
%d %i	Десятичное знаковое число, размер по умолчанию, sizeof(int). По умолчанию записывается с правым выравниванием, знак пишется только для отрицательных чисел
%e %E	Числа с плавающей точкой в экспоненциальной форме записи (вида 1.1e+44); %e выводит символ «e» в нижнем регистре, %E – в верхнем
%f	Десятичное число с плавающей точкой
%g %G	Число с плавающей точкой; форма представления зависит от значения величины (f или e)
%o	Восьмеричное беззнаковое число, размер по умолчанию sizeof(int)
%s	Вывод строки с нулевым завершающим байтом
%u	Десятичное беззнаковое число, размер по умолчанию sizeof(int)
%x %X	Шестнадцатеричное число, x использует маленькие буквы (abcdef), X большие (ABCDEF), размер по умолчанию sizeof(int)
%p	Вывод указателя, внешний вид может существенно различаться в зависимости от внутреннего представления в компиляторе и платформе (например, 16-битная платформа MS DOS использует форму записи вида FFEC:1003, 32-битная платформа с плоской адресацией использует адрес вида 00FA0030)
%n	Запись по указателю, переданному в качестве аргумента, количества символов, записанных на момент появления командной последовательности, содержащей %n
%%	Символ для вывода знака процента (%)

### ***Форматный ввод с использованием функции scanf()***

Гораздо более сложным для начинающих программистов является 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 осуществляется нажатием клавиши *Enter*.

Первый аргумент функции *scanf* представляет 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 этом случае 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 набирать 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. Обязательными элементами любого форматного указателя являются начальный символ % и последний символ, определяющий тип вводимого значения.

Символ \* после начального символа является 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. 5.4.

## Допустимое значение спецификаторов в строке ввода

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

Форматный ввод, так же как и потоковый, не позволяет вводить числовые значения в переменные типа *char*. Дело в том, что минимальная длина числового значения, вводимого с помощью функции *scanf*, – два байта. И значение введенного старшего байта затирает в памяти еще один байт вслед за переменной типа *char*. Заметить такую ошибку удастся не каждому, но на работу программы такой ввод может повлиять довольно серьезно, поэтому возьмите за правило – в однобайтовые переменные типа *char* числовую информацию вводить нельзя.

**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", &stag);
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';
```

В первом операторе используется *стандартный выходной поток* *cout* и операция << (операция передачи в поток, произносится «послать в»). Оператор читается так:

"Enter new tag: " // послать строку в выходной поток *cout*.

Обратите внимание, что операция передачи в поток обозначается так же, как поразрядная операция сдвига влево. Во втором операторе используется *стандартный входной поток* *cin* и операция >> (извлечения из потока, произносится «взять из»). Этот оператор читается:

Взять из входного потока значение для переменной *tag*.

Обратите внимание, что операция извлечения из потока становится операцией поразрядного сдвига вправо, если левый параметр является целочисленным типом. Операции передачи и извлечения из потока, в отличие от функций *printf* и *scanf*, не требуют форматизирующих строк и спецификаторов преобразования для указания на тип входных и выходных данных. В C++ есть много примеров, подобных этому, когда он автоматически «знает», какие типы должны участвовать в операциях. Стоит обратить внимание на то, что при использовании операций извлечения из потока переменной *tag* не предшествует операция взятия адреса &, которую требует *scanf*.

Для организации потокового ввода/вывода программы на C++ должны включать заголовочный файл *iostream.h*.

### Вопросы и задания

1. Можно ли использовать ключевые слова для именования объектов программы?
2. Перечислите основные элементы программы.
3. В чем различия 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:

$a = b + c * d \ll 2 \& 8$

$a = -1 ++ b -- -5$

$a \& 077 != 3$

$a = b == c ++$

$a == b \parallel a == c \&\& c < 5$

$a = b = c = 0$

$c = x != 0$

$a - b, c = d$

## Лекция 6. ОПЕРАТОРЫ ЯЗЫКА

1. *Пустой оператор (;)* используется, когда синтаксически должен быть оператор, а мы не хотим его встраивать. Можно его помечать меткой «пустой оператор» (метка1: ;).

2. *Составной оператор { }* используется, когда синтаксически возможен хотя бы один оператор.

3. *Оператор безусловного перехода goto M* осуществляет передачу управления на строку кода, помеченную меткой *M*; видимость оператора внутри функции, т.е. локальная. Переход на метку из другой функции (осуществить дальний переход) невозможен. 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;
    }
    return c;
}
```

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 exit (c)* – аварийный выход в среду (точку) вызова (запуска) программы.

6. *Оператор-выражение.*

Любое выражение, которое заканчивается точкой с запятой, является оператором. Выполнение данного оператора заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызвать функцию, не возвращающую значение, 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, со-
стоящее из вызова функции. */
```

7. *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.

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

9. *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++;
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*. Для того чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах *if*, используя фигурные скобки. Если же они 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 двум.

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 (a > b)
    if (b < c) t = b;
    else r = t;
  return 0;
}
```

В этом случае *r* получит значение, равное трем, 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*, являются довольно громоздкими и не всегда достаточно надежными. Другой способ организации выбора из множества различных вариантов – использование специального оператора выбора *switch*.

10. *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 в круглых скобках, следующее за ключевым словом *switch*, 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 – ключевое 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.

Константное выражение не может содержать переменные или вызовы функций, так как вычисляется во время трансляции. Обычно в качестве константного выражения используются 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*.

Список операторов может быть пустым либо содержать 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 должна использоваться инициализация.

Схема выполнения оператора *switch* следующая:

- 1) вычисляются 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 управление передается 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 начальной 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-либо 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 используется 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 теле оператора *switch* можно использовать вложенные операторы *switch*, при этом в ключевых словах *case* можно использовать 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;
    :
}
```

## 11. Операторы циклов

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

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 выполняться оператор *break*, при его отсутствии цикл будет выполняться 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 итера-
ции цикла */
    if (i=N) break; // условие выхода из цикла
}
```



Тело цикла может быть 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, то есть содержать в качестве оператора цикл (вложенные циклы). Если во вложенном цикле тело цикла 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 заключается в том, что тело цикла выполняется один раз, затем происходит проверка условия. Если условие «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=f(i);
    if (x==1)
    {
        printf ("%d\n" , x);
        break;
    }
}
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* следующая:

а) вычисляется выражение;

б) если выражение ложно, то выполнение оператора *while* заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора *while*;

в) процесс повторяется сначала.

Оператор цикла с пошаговым исполнением

```
for (выражение1; выражение2; выражение3) тело;
```

можно заменить оператором *while* следующим образом:

```
выражение1;  
while (выражение2)  
{  
    тело цикла;  
    выражение3;  
}
```

Так же как и при выполнении оператора *for*, в операторе *while* вначале происходит проверка условия, поэтому оператор *while* удобно использовать в ситуациях, когда тело оператора не всегда нужно выполнять.

Внутри операторов *for* и *while* можно использовать локальные переменные, которые должны быть объявлены с определением соответствующих типов.

### Вопросы и задания

1. Перечислите циклические операторы, их синтаксис и семантику.

2. Назначение блочного оператора.

3. Поясните принцип работы операторов *break* и *continue*.

4. Является ли обязательной метка *default* в операторе-переключателе?

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=='?'*.

## Лекция 7. МАССИВЫ

### § 7.1. 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 тип элемента тогда 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 будет отведено место 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 С++ ни на стадии компиляции, ни на шаге выполнения не отслеживается факт выхода за пределы массива.

В языке С++ 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 двумерного массива содержит два константных выражения, трехмерного – три и т.д. Отметим, что в языке С++ 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 в фигурные скобки, соответствуют строкам массива, в случае отсутствия скобок инициализация будет выполнена неправильно.

В языке С++ можно использовать сечения массива, как и в других языках высокого уровня, однако на использование сечений на-

кладывается ряд ограничений. Сечения формируются вследствие опускания одной или нескольких пар квадратных скобок. Пары квадратных скобок можно отбрасывать только справа налево и строго последовательно. Сечения массивов используются 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  $3 \times 4$ . 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 является управляющей последовательностью `'\0'`.

## § 7.2. Массив с точки зрения языка C++ (на виртуальном уровне) и с точки зрения хранения

Имя массива трактуется как константный указатель на элементы массива. Имя массива не может переопределяться.

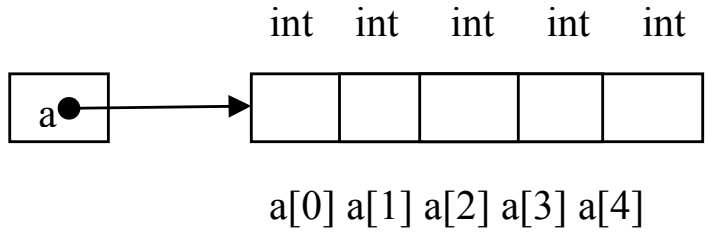
Рассмотрим одномерный массив `int a[5]`, который размещен в памяти следующим образом (рис. 7.1).

a[0] a[1] a[2] a[3] a[4]



Рис. 7.1. Размещение массива в памяти

Массив с точки зрения языка Си/C++ выглядит так, как показано на рис. 7.2.



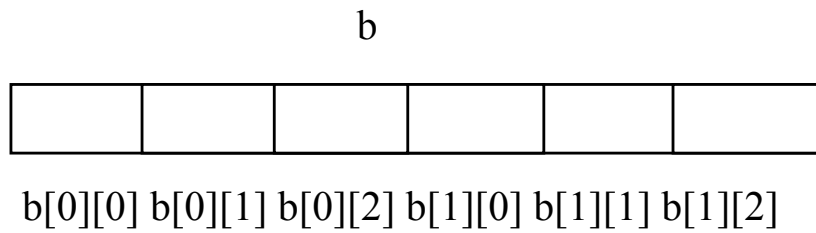
**Рис. 7.2. Массив с точки зрения языка Си/C++**

При этом  $a \equiv \&a$ .

Теперь рассмотрим двумерный массив с двух точек зрения:

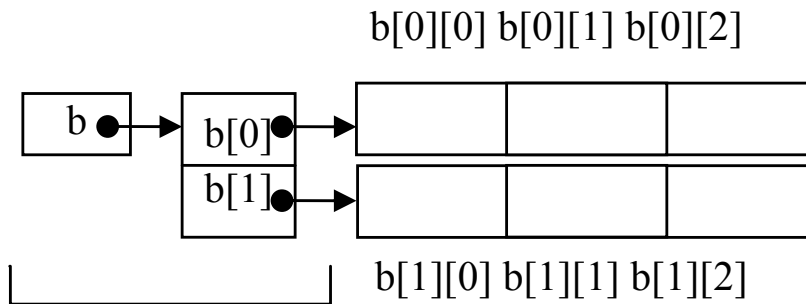
```
int b[2][3]; /* массив содержит 2 строки по 3 элемента в каждой */
```

Массив размещен в памяти так, как это показано на рис. 7.3.



**Рис. 7.3. Размещение двумерного массива в памяти**

С точки зрения языка Си/C++ двумерный массив выглядит так, как это показано на рис. 7.4.



Эти ячейки виртуальны,  
их тип – указатель

**Рис. 7.4. Двумерный массив с точки зрения языка Си/C++**

Массив  $b$  можно проинтерпретировать двумя способами:

- 1) указатель на указатель на  $int$ ;
- 2) массив указателей на  $int$ .

### § 7.3. Примеры работы с элементами массива

К элементу массива можно обращаться с помощью индексного выражения:

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

Пример:

```
a[1].
```

В C++ обязательным требованием при таком обращении является 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) вычисляется индексное выражение, к указателю будет прибавляться смещение типа умножение на длину специфицированного типа;

2) далее выполняется косвенная адресация, 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 можно определить размер массива?

5. Объявите трехмерный массив и изобразите его с точки зрения языка C++.

6. Объявите двумерный массив вещественных элементов размером 5 строк и 4 столбца. Далее:
  - а) проинициализируйте при объявлении;
  - б) организуйте заполнение массива с клавиатуры;
  - в) организуйте заполнение случайными числами.
7. Посчитайте сумму элементов массива.
8. Может ли элементом массива быть массив?

## Лекция 8. СТРОКИ И ОПЕРАЦИИ СО СТРОКАМИ

### § 8.1. 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 объявляют массив типа *char* с количеством элементов, доста-



filename [0]  
filename [1]

filename [62]  
filename [63]

точным 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. 8.1).

**Рис. 8.1.** 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 программист присваивает символы символьной строке, он должен 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';
        letter++, 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 выводит символы строки, пока не встретит символ *NULL*.

Цикл инициализирует и увеличивает две переменные (*letter* и *index*). Когда цикл *for* инициализирует или увеличивает несколько переменных, необходимо разделять 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. 8.2.

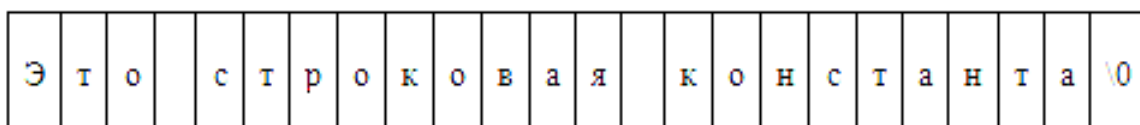


Рис. 8.2. Автоматическое добавление символа *NULL* к строковым константам

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

## § 8.2. Инициализация символьной строки

C++ позволяет инициализировать массивы при объявлении. Символьные строки C++ не являются 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 программист не указывает размер массива, который инициализирует при объявлении, компилятор C++ распределит достаточно памяти для размещения указанных букв и символа *NULL*:

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

## § 8.3. 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 ре-
гистра.
```

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

### Вопросы и задания

1. Что такое строка с точки зрения языка C++?
2. В чем отличие строки от массива символов?
3. Можно ли работать поэлементно со строкой?
4. Напишите программу, вычисляющую сумму цифр в строке вида "1ab3c405". Ввод строки осуществите 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.

## Лекция 9. УКАЗАТЕЛИ И ССЫЛКИ. РАБОТА С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

### § 9.1. Понятие указателя

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 ком-

пильтором на адрес области памяти, в которой хранится значение переменной. Можно определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями.

Через указатель можно обращаться к некоторому объекту, т.к. указатель – это его адрес.

#### Пример:

```
y=&x /* переменной y присваивается адрес переменной x, указывающий, где в оперативной памяти находится значение x */
```

```
z=*y /* переменной z присваивается значение переменной, записанной по адресу y, если y=&x и z=*y, то z=x */
```

В языке C++ различают три вида указателей:

- 1) 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.

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

## § 9.2. 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++ используют два способа работы с динамической памятью: первый использует семейство функций *malloc* и достался в наследство от языка Си, второй – операции *new* и *delete*.

При определении указателя надо стремиться выполнить его инициализацию, то есть присвоение начального значения. Непреднамеренное использование неинициализированных указателей – распространенная причина 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:

а) 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.
```

б) с помощью значения другого инициализированного указателя:

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

2. Присваивание указателю адреса области памяти в явном виде:

```
char* vp = (char *)0xB8000000;
```

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, ссылается ли указатель на конкретный объект или нет.

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

а) с помощью операции *new*:

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

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

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

В операторе 1 операция *new* выполняет выделение достаточно для размещения величины типа *int* участка динамической памяти и записывает адрес начала этого участка в переменную *n*. Память под саму переменную *n* (размер, достаточный 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 десять величин типа *int* (массива из десяти 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 называются 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.

### § 9.3. Операции с указателями

С указателями можно производить операции присваивания и арифметические отношения.

Операция присваивания для указателей аналогична этой операции для других типов данных. В отличие от других типов 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 переменной a=8*/  
p1 = p /* значение 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 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  $sizeof(tun)$ . 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-й элемент массива */
```



```

        printf("%c", *s); /* выводим значение по указателю */
        s++; //настраиваем указатель на 5-й элемент
        printf("%c", *s); /* выводим значение по указателю */
        s++; //настраиваем указатель на 6-й 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.

## § 9.4. Ссылки

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

```

    int& alias_name = variable; //Объявление ссылки

```

После объявления ссылки программа может использовать или переменную, или ссылку:

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

В следующем примере необходимо создать ссылку с именем *alias\_name* и присвоить псевдониму переменную *number*. Далее в коде используется как ссылка, так и переменная.

Пример:

```
#include <iostream >
void main(void)
{
    int number = 501;
    int& alias_name = number; // Создать ссылку

    cout << "Переменная number содержит " << number << endl;

    cout << "Псевдоним для number содержит " << alias_name << endl;

    alias_name = alias_name + 500;

    cout << "Переменная number содержит " << number << endl;

    cout << "Псевдоним для number содержит " << alias_name << endl;
}
```

Программа прибавляет 500 к ссылке *alias\_name*. В итоге программа прибавляет 500 также и к соответствующей переменной *number*.

**Результаты работы программы:**

```
Переменная number содержит 501
Псевдоним для number содержит 501
Переменная number содержит 1001
Псевдоним для number содержит 1001
```

Использование ссылок значительно упрощает процесс изменения параметров внутри функции.

После своей инициализации ссылка не может быть изменена 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. Объявите массив из трех указателей на вещественные переменные и задайте значения переменных через указатели.
6. Разместите 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.

## Лекция 10. СТРУКТУРЫ

### § 10.1. Работа со структурами

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 C++ структура является видом 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 быть указателями на него. Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов. В этом случае описание структуры служит определением 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 не обязательно 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 является 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, что 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*.

Представим, что необходимо описать структуру с именем *avto*, содержащую поля:

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

Необходимо 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 таких автомобилей нет, то на экран дисплея вывести соответствующее сообщение.

Программа решения задачи будет иметь вид:

```

#include <iostream >
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <windows.h>
using namespace std;

char buf[256];
char * Rus(char *text)
{
    CharToOem(text, buf);
    return buf;
}

```

```

struct avto {
    char fam[25];
    char marka[20];
    char nomer[10];
};

void output_gai(avto *,int );

void main(void)
{
    int i,n;
    bool flag;
    cout<<Rus("Введите количество записей: ");
    cin>>n;
    avto *gai = new avto[n], temp;
    char m[20];
    for(i=0;i<n;i++)
    { // Ввод данных
        cout<<Rus("Введите фамилию владельца-
ца")<<i+1;
        cout<<Rus("-го автомобиля: ");
        cin >> gai[i].fam;
        cout<<Rus("Введите марку ")<<i+1;
        cout<<Rus("-го автомобиля: ");
        cin >> gai[i].marka;
        cout<<Rus("Введите номер ")<<i+1;
        cout<<Rus("-го автомобиля: ");
        cin >> gai[i].nomer;
    }
    cout << Rus("Исходные данные до сортировки")
<< endl;
    output_gai(gai,n);
    // 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<<Rus("После сортировки")<<endl;
    output_gai(gai,n);
    cout << endl << Rus("Введите марку автомобиля:
");
    cin >> m;
    cout << endl << Rus("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<<Rus("\nМарки ")<<m;
            cout<<Rus("нет в списках");
        }
        delete [] gai;
    }
}
void output_gai( 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;
    }
}

```

## § 10.2. Битовые поля

*Битовые поля* – это особый вид полей структуры, которые обеспечивают доступ к отдельным битам памяти. Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы



вы битовых полей и применять к полям операцию определения адреса. В общем случае тип структуры с битовым полем задается в следующем виде:

```
struct { unsigned идентификатор 1 : длина поля 1;  
        unsigned идентификатор 2 : длина поля 2;};
```

Длина поля задается целым выражением или константой, которая определяет число битов, отведенное соответствующему полю. Поле нулевой длины обозначает выравнивание на границу следующего слова.

Пример:

```
struct {  
    unsigned a1 : 1;  
    unsigned a2 : 2;  
    unsigned a3 : 5;  
    unsigned a4 : 2;  
} prim;
```

Структуры битовых полей могут содержать и знаковые компоненты. Такие компоненты автоматически размещаются на соответствующих границах слов, при этом некоторые биты слов могут оставаться неиспользованными.

Ссылки на поле битов выполняются точно так же, как и компоненты общих структур. Само же битовое поле рассматривается как целое число, максимальное значение которого определяется длиной поля.

### **Вопросы и задания**

1. Назовите отличия структуры от массива.
2. Назовите операции доступа к полям структуры по указателю и через объект.
3. Назначение битовых полей и особенности работы с ними.

## **Лекция 11. СОСТАВНЫЕ ТИПЫ ДАННЫХ**

### **§ 11.1. Перечисление**

Перечисление – это набор именованных целочисленных констант, определяющий все допустимые значения, которые может при-

нимать переменная. Перечисления можно встретить в повседневной жизни. Например, в качестве перечислений можно использовать дни недели:

```
понедельник, вторник, среда, четверг, пятница, суббота, воскресенье
```

Объявление перечисления начинается с ключевого слова *enum*, которое указывает на начало перечисления. Стандартный формат перечисления следующий:

```
enum [имя_типа_перечисления] {список_перечислений}
    [список_переменных];
```

Как *имя\_типа\_перечисления*, так и *список\_переменных* необязательны, но один из них обязательно должен присутствовать.

*Список\_перечислений* – это разделенный запятыми список идентификаторов. *Имя\_типа\_перечисления* используется для объявления переменных данного типа. Следующий фрагмент определяет перечисление *week* и объявляет переменную *day* этого типа:

```
enum week {monday, tuesday, wednesday, thursday, friday,
saturday, sunday};
enum week day;
```

Имея данное объявление и определение, следующие операторы совершенно корректны.

Пример:

```
day = wednesday;
if(day == friday) printf("Это пятница");
```

В перечислениях каждому символу ставится в соответствие целочисленное значение, поэтому перечисления можно использовать в любых выражениях.

Пример:

```
printf("Переменная friday имеет значение %d", friday);
```

Если явно не проводить инициализацию, значение первого символа перечисления будет ноль, второго – единица и т.д.

Можно определить значения одного или нескольких символов, используя инициализатор. Это делается путем помещения за символом знака равенства целочисленного значения. При использовании инициализатора символы, следующие за инициализационным значением, получают значение большее на единицу, чем указанное перед этим.

Пример:

```
enum week{monday, tuesday, wednesday, thursday=100,  
          friday, saturday, sunday};
```

В примере символы получают следующие значения:

```
monday 0  
tuesday 1  
wednesday 2  
Thursday 100  
friday 101  
Saturday 102  
sunday 103
```

Использование элементов перечисления должно подчиняться следующим правилам:

1. Переменная может содержать повторяющиеся значения.
2. Идентификаторы в списке перечисления должны быть отличны от всех других идентификаторов в той же области видимости, включая имена обычных переменных и идентификаторы из других списков перечислений.
3. Имена типов перечислений должны быть отличны от других имен типов перечислений, структур и объединений в этой же области видимости.
4. Значение может следовать за последним элементом списка перечисления.

Невозможен прямой ввод или вывод символов перечислений. Фрагмент кода, представленный в примере, не работает.

Пример:

```
day = sunday;  
printf("%s", day);
```

Надо помнить, что символ *sunday* – это просто имя для целого числа, а не строка. Следовательно, невозможно с помощью *printf()* вывести строку «*sunday*», используя значение в *day*. Аналогично нельзя сообщить значение переменной перечисления, используя строковый эквивалент. Таким образом, следующий код также не работает:

```
day = "wednesday";
```

На самом деле, создание кода для ввода и вывода символов перечислений – это довольно скучное занятие. Например, следующий код необходим для вывода состояния *day* с помощью слов:

```

switch (day) {
case Monday: printf("Monday");
             break;
case Tuesday: printf("Tuesday");
             break;
case wedday: printf("wedday");
             break;
case Thursday: printf("Thursday");
             break;
case Friday: printf("Friday");
             break;
case Saturday: printf("Saturday");
             break;
case Sunday: printf("Sunday");
             }

```

Поскольку значения перечислений должны преобразовываться вручную к строкам, которые могут читать люди, они наиболее полезны в подпрограммах, не выполняющих такие преобразования.

## § 11.2. Объединения

Объединение подобно структуре, однако в каждый момент времени может использоваться (другими словами, быть доступным) только один из элементов объединения. Тип объединения может задаваться в следующем виде:

```

union [<название типа (тег)>] {
    описание элемента 1;
    ...
    описание элемента n;
} [описатели переменных];

```

Главная особенность объединения – то, что для каждого из объявленных элементов выделяется одна и та же область памяти, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

Доступ к элементам объединения осуществляется тем же способом, что и к структурам. Тег объединения может быть формализован точно так же, как и тег структуры.

Объединение применяется для следующих целей:

1) инициализации используемого объекта памяти, если в каждый момент времени только один объект из многих является активным;

2) интерпретации основного представления объекта одного типа, как если бы этому объекту был присвоен другой тип.

Память, которая соответствует переменной типа объединения, определяется величиной, необходимой для размещения наиболее длинного элемента объединения. Когда используется элемент меньшей длины, то переменная типа объединения может содержать неиспользуемую память. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

Пример:

```
union {
    char fio[30];
    char adres[80];
    int vozrast;
    int telefon;
} inform;
union {
    int ax;
    char al[2];
} ua;
```

При использовании объекта *inform* типа *union* можно обрабатывать только тот элемент, который получил значение, т.е. после присвоения значения элементу *inform.fio* не имеет смысла обращаться к другим элементам. Объединение *ua* позволяет получить отдельный доступ к младшему *ua.al[0]* и к старшему *ua.al[1]* байтам двухбайтного числа *ua.ax*.

### Вопросы и задания

1. С помощью какого ключевого слова можно объявить перечисление?
2. С какого значения начинается нумерация элементов перечисления?
3. Можно ли изменить нумерацию элементов перечисления? Если да, то каким образом?
4. Назовите синтаксис объявления объединения.
5. В чем отличия объединения от структур?

## Лекция 12. РАБОТА С ФУНКЦИЯМИ

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

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. использование однажды написанных функций в качестве конструктивных блоков для создания новых программ. Многократное использование кода – 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 разбиения программ на функции – 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 должно отражать смысл данной задачи. Это облегчает абстракцию и способствует многократному использованию программного кода.

### § 12.1. Прототип

Одна из наиболее важных особенностей *ANSI Си* – *прототипы функций*. 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 языка Си не выполняли этот вид проверки, так что существовала возможность неправильного вызова функции, при котором компилятор не регистрировал ошибки. Такие обращения могли приводить к фатальным ошибкам при выполнении программы или к ошибкам, которые, не вызывая краха программы, приводили тем не менее к трудно обнаруживаемым логическим ошибкам. Прототипы функций в *ANSI* Си исправляют такое положение дел.

Прототип функции в общем виде:

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

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

```
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 включаются в прототип функции с целью документирования. Компилятор игнорирует эти имена. 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* заголовка функции.

Другое важное следствие применения прототипов функций – *автоматическое приведение аргументов*, т.е. принудительное преобразование аргументов функции к соответствующему типу. Преобразование значений к более высоким типам происходит автома-

тически. 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 определения функции, 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.

## § 12.2. 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. Тип результата, возвращаемый вызывающей функции, – *тип\_возвращаемого\_значения*. Если он 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*.

Стоит сделать несколько замечаний:

1. Ошибкой является 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 состояться 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, уместиться на одной строке.

7. Прототип функции, заголовок определенной функции и вызов функции 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.

### § 12.3. Способы передачи параметров в функции

Существуют 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 указывается 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 - фактические
    параметры */
    cout<<a3;
    return 0;
}
```

Если функция не возвращает значения, то используется оператор *return* без параметров. В этом случае оператор *return* 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 адреса параметра. Ссылка, по определению, есть синоним переменной, поэтому 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;
}

```

В языке C++ отсутствует высокоуровневое понятие массива, поэтому его передача в качестве параметра сводится к передаче указателя на начальный элемент массива. Следствие этого – 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];
    return 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:

```

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.

## § 12.4. Вызов функции

Во многих языках программирования существуют 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 ней, не отражаются на значении исходной переменной в вы-

зывающей 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.

## § 12.5. Параметры функций по умолчанию

Для того 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 языке C++ требуется выполнение следующих правил для использования параметров по умолчанию:

- 1) когда назначается параметр (значение) по умолчанию какому-либо параметру, то необходимо назначать параметры (значение) по умолчанию всем последующим параметрам;
- 2) принимаемые по умолчанию параметры делят список на две части: первая не содержит значений, вторая – содержит;
- 3) для того чтобы использовать значение параметра по умолчанию в качестве фактического параметра, 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);
```

## § 12.6. Перегрузка функций

В языке Си объявление двух функций с одним и тем же именем в одной и той же программе является синтаксической ошибкой. С++ 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 С++ 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 *сигатурой* – комбинацией имени функции и типов ее параметров. Компилятор специальным образом кодирует идентификатор каждой функции, используя количество и типы ее параметров для обеспечения безо-

пасного 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* – тип *int*, *pf* – тип *float \** и *pd* – тип *double \**. В списке параметров функции *nothing1* *i* представляет тип *int*, *f* – тип *float*, *zc* – тип *char* и *pi* – тип *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 nothing1 (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$qzqipfpd
public @nothing1$qifzcpd
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. Имейте в виду, что кодирование имен функций зависит от компилятора, поэтому функция, компилируемая в Borland C++, может иметь декорированное имя,

отличное от того, которое она получила бы при использовании других компиляторов.

Перегруженные функции могут иметь разные типы 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 иметь одно и то же количество 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.

## § 12.7. 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 смысла оператора.

тип\_возвращаемого\_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

```

## § 12.8. Шаблоны функций

Многие алгоритмы не зависят от типа данных, с которыми они работают (классический пример – сортировка), поэтому естественно 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 передается 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 эле-
ментов
    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;
    }
}

```

## § 12.9. Встраиваемые функции

Когда в программе определена функция, компилятор С++ переводит код функции в машинный язык, сохраняя только одну копию инструкций функции внутри программы. Каждый раз, когда программа вызывает функцию, компилятор С++ помещает в программу специальные инструкции, которые заносят параметры функции в стек и затем выполняют переход к инструкциям этой функции. Когда операторы функции завершаются, выполнение програм-

мы продолжается с первого оператора, который следует за вызовом функции. Помещение аргументов в стек и переход в функцию и из нее вносит 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 следующем примере, вызывает функцию *show\_message*, которая указанное число раз выдает сигнал на динамик компьютера и затем выводит сообщение на дисплей.

Пример:

```
#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 далее, не вызывает функцию *show\_message*. Вместо этого она помещает внутри себя те же операторы функции при каждой ссылке на функцию.

Пример:

```
#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, 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).

## § 12.10. Параметры, передаваемые через командную строку

Программы могут принимать аргументы. Функция *main* может содержать в качестве аргументов два параметра: *argc* – целочисленную переменную, содержащую 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, которые были переданы 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 arguments\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
```

## § 12.11. Рекурсия

### *Понятие рекурсии*

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

Прямая (непосредственная) рекурсия – вызов функции внутри тела этой функции:

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

Косвенная рекурсия – 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 это 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 передачу ей копий параметров и, 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  $\{1, 1, 2, 3, 5, 8, \dots\}$ , 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 Хоара в 1961 г.

В массиве выбирается элемент, относительно которого ведётся сортировка, как правило, это средний элемент  $a[k]$ . Сначала перебираются все элементы, расположенные слева от  $a[k]$ , начиная с

первого, до элемента, большего либо равного по значению  $a[i] \geq a[k]$ . После этого перебираются 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 оно не выполняется, то массив разбивается на два новых массива 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:

- а) в качестве параметра функции;
- б) 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, описанные в § 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?

## Лекция 13. ФАЙЛЫ. БАЗОВЫЕ ФУНКЦИИ РАБОТЫ С ПОТОКАМИ

### § 13.1. Понятие физического и логического файлов

У понятия «файл» есть два значения. С одной стороны, *файл* – это именованная область внешней памяти, содержащая какую-либо информацию. Файл в таком понимании называют *физическим*, то есть 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*, то есть 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. Об-

разно 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 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.

## § 13.2. Работа с файлами

### *Указатель файла*

Указатель файла – это указатель на структуру типа *FILE*. Он указывает на структуру, содержащую различные сведения о файле, например его имя, статус, указатель текущей позиции в файле. Для выполнения в файлах операций чтения и записи программы должны использовать соответствующие указатели файлов. Для объявления переменной-указателя файла используется 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 быть выполнены 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*, *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 дан-

ные, которые ещё оставались в дисковом буфере, и проводит, обратно говоря, 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*. Для того 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 feof()***

Для проверки факта достижения конца файла применяется функция *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.

### § 13.3. Текстовые (строковые) файлы

Текстовые файлы относятся к файлам последовательного доступа, так как единицами хранения информации в них являются строки переменной длины. Каждая строка заканчивается специальным признаком, обычно его функцию выполняет пара символов `'\r'` и `'\n'` – "возврат каретки" и "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 – преобразование числовых данных из машинных форматов в символьный вид при выводе и обратное преобразование при вводе сопряжено 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
                               // открытие 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
```

### § 13.4. Двоичные файлы

Двоичные файлы отличаются от текстовых тем, что представляют собой последовательность байтов, содержимое которых может иметь различную природу. Это могут быть байты, представляющие 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 это делается при обработке текстовой информации. Естественно, что единицей обмена 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*. Запись данных в двоичный файл осуществляется 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.

Считывание данных из двоичного файла осуществляется 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, 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, не обязательно является ошибкой. Типичная картина – количество данных в файле не кратно размеру заказанных порций.

Двоичные файлы допускают не только последовательный обмен данными. Имеется возможность пропустить часть данных или вернуться повторно к ранее обработанной информации, так как размеры порций данных и их количество, участвующее в очередном обмене, диктуются программистом, а не смыслом информации, хранящейся в файле. 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 4×10 порций данных в машинном формате (строки, целые и вещественные числа). После записи данных файл закрывается и вновь открывается для чтения. Для демонстрации прямого доступа к данным информация из фай-

ла 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 двоично-
го 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();
}

```

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

```

**Использованные в этом примере операторы:**

```

fclose(f1); //закрытие файла
f1=fopen("c_bin","rb"); //открытие двоичного фай-

```

ла для чтения

Они могут быть заменены обращением к единственной функции *freopen*, которая повторно открывает ранее открытый файл:

```
f1=freopen("c_bin","rb");
```

Основное правило, которого надо придерживаться при обмене с двоичными файлами, звучит примерно так: как данные записывались в файл, так они и должны читаться.

### **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 совершить для работы с файлом?
6. Какая функция используется для открытия файла? Опишите параметры функции.
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.

## **Лекция 14. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ**

### **§ 14.1. Пространство имен в C++**

В программе на языке C++ имена (идентификаторы) используются для ссылок на различного рода объекты – функции, переменные, формальные параметры и т.п. При соблюдении определенных правил допускается использование одного и того же идентификатора более чем для одного программного объекта.

Для того чтобы различать идентификаторы объектов различного рода, компилятор языка C++ устанавливает так называемые «пространства имен». Во избежание противоречий имена внутри одного пространства должны быть уникальными, однако в различных пространствах могут содержаться идентичные имена. Это означает, что можно использовать один и тот же идентификатор для двух или более различных объектов, если их имена принадлежат к различным пространствам. Однозначное разрешение вопроса о том, на какой объект ссылается идентификатор, компилятор языка C++ осуществляет по контексту появления данного идентификатора в

программе. В табл. 14.1 перечислены виды объектов, которые можно именовать в программе на языке C++ и соответствующие им четыре пространства имен.

*Виды объектов и пространства имён*                      Табл. 14.1

Объекты	Пространство имен
Переменные, функции, формальные параметры, элементы списка перечисления, <i>typedef</i>	Уникальность имен в пределах этого пространства тесно связана с понятием области действия. Это выражается в том, что в данном пространстве могут содержаться совпадающие идентификаторы, если области действия именуемых ими объектов не пересекаются. Другими словами, совпадение идентификаторов возможно только при локальном переобъявлении. Обратите внимание на то, что имена формальных параметров функции сгруппированы в одном пространстве с именами локальных переменных. Следовательно, переобъявление формальных параметров внутри любого из блоков функции недопустимо, <i>typedef</i> – это объявления имен типов
Теги	Теги всех переменных перечисляемого типа, структур и объединений сгруппированы в одном пространстве имен. Каждый тег переменной перечисляемого типа, структуры или объединения должен быть отличен от других тегов с той же самой областью действия. Ни с какими другими именами имена тегов не конфликтуют
Элементы структур и объединений	Элементы каждой структуры или объединения образуют свое пространство имен, поэтому имя каждого элемента должно быть уникальным внутри структуры или объединения, но не обязано отличаться от любого другого имени в программе, включая имена элементов других структур и объединений
Метки операторов	Метки операторов образуют отдельное пространство имен. Каждая метка должна быть отлична от всех других меток операторов в той же самой функции. В разных функциях могут быть одинаковые метки

Пример:

```
struct student
{
    char student [20]; /*массив из 20 элементов
    типа char*/
```

```

        int class;
        int id;
    } student; /* структура из трех элементов */

```

В этом примере имя тега структуры, элемента структуры и самой структуры относится к трем различным пространствам имен, поэтому не возникает противоречия между тремя объектами с одинаковым именем *student*. Компилятор языка C++ определит по контексту использования, на какой из объектов ссылается идентификатор в каждом конкретном случае. Например, когда идентификатор *student* появится после ключевого слова *struct*, это будет означать, что именуется тег структуры. Когда идентификатор *student* появится после операции выбора элемента ( $\rightarrow$  или  $.$ ), то это будет означать, что именуется элемент структуры. В любом другом контексте идентификатор *student* будет рассматриваться как ссылка на переменную структурного типа.

## § 14.2. Область видимости и время жизни переменных.

### Класс памяти

Атрибуты переменных включают в себя имя, тип и значение. Каждый идентификатор в программе имеет и другие атрибуты, включая *класс памяти*, *период хранения*, *область действия* и *тип компоновки*.

Язык C++ поддерживает четыре класса памяти, обозначаемые *спецификаторами класса памяти*: *auto*, *register*, *extern* и *static*. Класс памяти идентификатора помогает определить его период хранения, область действия и тип компоновки. *Период хранения* идентификатора – это время, в течение которого данный идентификатор существует в памяти. Некоторые идентификаторы существуют короткое время, некоторые неоднократно создаются и разрушаются, другие существуют в течение всего времени выполнения программы. *Область действия* идентификатора характеризует возможность обращения к нему из различных частей программы. Некоторые идентификаторы доступны во всей программе, другие – только в отдельных ее частях. *Тип компоновки* идентификатора определяется для программ, состоящих из нескольких исходных файлов, объединяемых на этапе компоновки. Эта характеристика показы-

вает, известен ли идентификатор только в текущем исходном файле или в любом исходном файле с соответствующими объявлениями. Автоматический период хранения могут иметь только переменные. Локальные переменные функции (объявленные в списке параметров или в теле функции) обычно имеют автоматический период хранения. Ключевое слово *auto* объявляет переменные с автоматическим хранением явным образом. Например, следующее объявление показывает, что переменные *x* и *y* типа *float* – автоматические локальные переменные, существующие только в теле функции, в которой находится данное объявление:

```
auto float x, y;
```

Локальные переменные имеют автоматический период хранения по умолчанию, так что ключевое слово *auto* используется редко.

Автоматическое хранение способствует экономии памяти, поскольку автоматические переменные существуют только тогда, когда они необходимы. Они создаются при запуске функции, в которой они объявлены, и уничтожаются, когда происходит выход из функции.

Перед объявлением автоматической переменной может быть помещен спецификатор класса памяти *register*, чтобы рекомендовать компилятору поместить ее в одном из быстродействующих аппаратных регистров компьютера. Если интенсивно используемые переменные типа счетчиков или сумм будут реализованы в аппаратных регистрах, то можно исключить непроизводительные затраты на неоднократную загрузку переменных из памяти в регистры и обратно.

Компилятор может проигнорировать объявление *register*. Например, в распоряжении компилятора может не оказаться *достаточно числа* регистров. Следующее объявление предлагает, чтобы целая переменная *counter* была размещена в одном из регистров компьютера и инициализирована значением «единица»:

```
register int counter = 1;
```

Ключевое слово *register* может использоваться только с переменными, имеющими автоматический период хранения.

Объявления *register* зачастую не нужны. Современные оптимизирующие компиляторы способны распознать часто используемые



переменные и могут размещать их в регистрах самостоятельно, не требуя от программиста объявления *register*.

Ключевые слова *extern* и *static* используются для объявления идентификаторов переменных и функций со статическим периодом хранения. Идентификаторы статического периода хранения существуют с того момента, как программа начинает выполняться. Для переменных память распределяется и инициализируется один раз, когда программа запускается. Для функций имя также начинает существовать с момента начала выполнения программы. Однако даже при том, что переменные и имена функций существуют с момента начала выполнения программы, это не означает, что эти идентификаторы доступны во всей программе. Период хранения и область действия (область, где имя доступно) – разные вещи.

Существуют два типа идентификаторов со статическим периодом хранения: внешние идентификаторы (вроде глобальных переменных и имен функций) и локальные переменные, объявленные со спецификатором класса памяти *static*. Глобальные переменные и имена функций имеют по умолчанию класс памяти *extern*. Глобальные переменные создаются при помещении их объявлений вне любого определения функции, и они сохраняют свои значения в течение всего времени выполнения программы. Обращение к глобальным переменным и функциям возможно из любой функции, которая следует после их объявления или определения в файле – это одна из причин использования прототипов функций. Когда мы включаем *stdio.h* в программу, которая вызывает *printf*, прототип функции помещается в начало нашего файла, делая имя *printf* известным для остальной части файла.

Объявление переменной как глобальной, а не локальной, может приводить к случайным побочным эффектам, когда функция, которой не нужен доступ к этой переменной, случайно или преднамеренно ее изменяет. В целом нужно избегать использования глобальных переменных, за исключением некоторых ситуаций с уникальными требованиями к производительности.

Переменные, используемые только в определенной функции, должны быть объявлены локальными переменными этой функции, а не внешними переменными.

Локальные переменные, объявленные с ключевым словом *static*, остаются известными только той функции, в которой они определены, но в отличие от автоматических статических локальные переменные сохраняют свое значение и после выхода из функции. При следующем вызове статическая локальная переменная будет содержать то значение, которое она имела при последнем выходе из функции. Следующий оператор объявляет, что локальная переменная *static* будет статической, и инициализирует ее значением 1.

```
static int count = 1;
```

Все числовые переменные со статическим хранением инициализируются нулем, если они явно не инициализированы программистом.

Использование нескольких спецификаторов класса памяти для идентификатора недопустимо. К данному идентификатору может применяться только один спецификатор класса памяти.

Ключевые слова *extern* и *static* имеют специальное значение, когда применяются к внешним идентификаторам явным образом.

### ***Правила области действия***

*Область действия* идентификатора – это та часть программы, в которой возможно обращение к нему. Например, когда мы объявляем в некотором блоке локальную переменную, к ней можно обратиться только из этого блока или из блоков, вложенных в данный блок. Область действия идентификатора делится на четыре вида: *область действия функции*, *область действия файла*, *область действия блока* и *область действия прототипа функции*.

Метки (идентификаторы, сопровождающиеся двоеточием, вроде *start*.) – единственные идентификаторы, принадлежащие *области действия функции*. Метки могут использоваться в произвольном месте функции, в которой они появляются, но на них нельзя сослаться вне тела функции. Метки используют в структурах *switch* (в качестве меток *case*) и в операторах *goto*. Метки относятся к подробностям реализации, которые функции скрывают друг от друга. Эта скрытность, более формально называемая *сокрытием информации*, – один из наиболее фундаментальных принципов хорошего стиля программирования.

Идентификатор, объявленный вне любой функции, имеет *область действия файла*. Такой идентификатор «известен» всем функциям, начиная с того места, где он объявлен, и до конца файла. Глобальные переменные, определения функций и прототипы функций, помещенные вне функции, – все они имеют область действия файла.

Идентификаторы, объявленные внутри блока, имеют *область действия блока*, заканчивающуюся завершающей правой фигурной скобкой блока (}). Локальные переменные, объявленные в начале функции, имеют область действия блока, так же как и параметры функции, которые рассматриваются как ее локальные переменные. Любой блок может содержать объявления переменных. Когда блоки вложены, а идентификатор во внешнем блоке имеет то же самое имя, что и идентификатор во внутреннем блоке, идентификатор во внешнем блоке «скрывается», пока внутренний блок не завершит работу. Это означает, что пока выполняется внутренний блок, он видит значение собственного локального идентификатора, а не значение идентификатора с тем же именем, находящегося в объемлющем блоке. Локальные переменные, объявленные как *static*, имеют область действия блока, несмотря на то, что существуют с момента начала выполнения программы. Таким образом, период хранения не влияет на область действия идентификатора.

Единственные идентификаторы с *областью действия прототипа функции* – идентификаторы, которые используются в списке параметров прототипа функции. Как было отмечено, прототипы функций не требуют, чтобы в списке параметров стояли имена идентификаторов; требуется только их тип. Если в списке параметров прототипа используется имя, то компилятор его игнорирует. Идентификаторы, указанные в прототипе функции, могут неоднократно встречаться в других местах программы, и здесь не возникает никакой неоднозначности.

Случайное объявление во внутреннем блоке имени идентификатора, которое уже используется во внешнем блоке, в то время как программист на самом деле хотел, чтобы идентификатор внешнего блока был доступен для внутреннего блока, приводит к сокрытию идентификатора внешнего блока.

Избегайте применения имен переменных, которые скрывают имена во внешних областях действия. Можно просто избегать любого дублирования идентификаторов в программе.

### § 14.3. Обработка исключительных ситуаций в C++

Исключения в C++ – это механизм обработки ошибок. Все знают о стандартных исключениях C++, которые представлены ключевыми словами *try/catch*. Исключение – это событие, которое возникает во время выполнения программы и требует выполнения кода за пределами нормального потока выполнения. Исключения могут быть вызваны либо «железом», либо самой программой. Примером исключения первого типа может служить деление на ноль; второй тип возникает только в том случае, когда программа явно его иницирует.

#### *Try-catch-throw*

Для того чтобы комфортно работать с исключениями в C++, нужно знать лишь три ключевых слова:

- 1) *try* («пытаться») – начало блока исключений;
- 2) *catch* («поймать») – начало блока, «ловящего» исключение;
- 3) *throw* («бросить») – ключевое слово, «создающее» («возбуждающее») исключение.

```
try
{
    // код, который может вызвать исключительную ситуа-
цию
}
[ catch (exception-declaration)
  { // код, выполняемый при генерации исключительной
ситуации в блоке try
  }
  [ catch (exception-declaration)
    { // код, выполняемый при генерации исключения
другого типа
    }
  ]...
]
//Синтаксис для генерации исключительной ситуации
throw [expression]
```

Код в разделе *try* называется защищенным. Выражение *throw* генерирует (возбуждает) исключение. Блок кода после *catch* – об-

работчик исключительных ситуаций, сгенерированных с помощью конструкции *throw*. *Exception-declaration* определяет тип исключительной ситуации. Операнд *throw* некоторым образом похож на *return*.

Процесс выполнения программы:

1. Управление передается в блок *try*; выполняется кусок кода, защищенного блоком *try*.

2. Если не сгенерировано ни одного исключения, то следующие за блоком *try* блоки *catch* не выполняются. Выполнение программы продолжается с команды, следующей за последним *catch*.

3. Если же сгенерировано исключение, которое создано оператором *throw*, компилятор начинает просматривать *catch* и искать подходящий блок для обработки исключения сгенерированного типа (либо блока, который сможет обработать исключение любого типа). Если обработчик не найден, то по порядку просматривается внешний блок *try*, и так до последнего блока *try*.

4. Когда возбуждается исключительная ситуация, программа просматривает стек функций до тех пор, пока не находит соответствующий *catch*. Если оператор *catch* не найден, то исключение будет обработано в стандартном обработчике, который делает все менее изящно, чем мог бы сделать программист, показывая какие-то непонятные (для конечного пользователя) сообщения и обычно аварийно завершая программу.

5. Если найден обработчик исключения, который был задан по значению, то значение формального параметра инициализируется копированием исключительного объекта. Если при объявлении была указана ссылка на исключительный объект, то будет установлена ссылка на него. Далее обработчик выполняется, и программа возобновляет выполнение за последним обработчиком.

В C++ можно отмечать функции, которые могут создать исключительную ситуацию, – для этого при объявлении функции используют спецификатор функции *throw [(...)]*. Например, спецификатор *throw (...)* сообщает компилятору, что функция может сгенерировать исключение, но тип его не определен (табл. 14.2).

*Спецификации исключительных ситуаций*

Спецификация исключения	Значение
<i>throw ()</i>	Такой метод может применяться в случаях, когда не нужно передавать никаких данных в блок <i>catch</i>
<i>throw (...)</i>	Функция может сгенерировать исключение
<i>throw (type)</i>	Функция может сгенерировать исключение определенного типа

Пример:

```
void MyFunc() throw(...)
{
    throw 1;
}
```

Пример:

```
void func()
{
    try
    {
        throw 1;
    }
    catch(int a)
    {
        cout << "Caught exception number: " << a;
        return;
    }
    cout << "No exception detected!" << endl;
    return;
}
```

Если выполнить этот фрагмент кода, то мы получим следующий результат:

```
Caught exception number: 1
```

Если закомментировать строку `throw 1`, то функция выдаст такой результат:

```
No exception detected!
```

Подход, приведённый выше, – очень мощное средство обработки ошибок. *Catch* может «ловить» любой тип данных, так же как и *throw* может сгенерировать данные любого типа, т.е. *throw AnyClass()*; будет правильно работать, так же как и *catch (AnyClass &d) {}*;

Как уже было сказано, *catch* может «ловить» данные любого типа, но вовсе не обязательно при этом указывать переменную, т.е. прекрасно будет работать что-нибудь типа этого:

```
catch(dumbclass) { }
```

так же, как и

```
catch(dumbclass&) { }
```

Так же можно «поймать» и все исключения:

```
catch(...) { }
```

Троеточие в этом случае показывает, что будут «пойманы» все исключения. При таком подходе нельзя указать имя переменной. В случае, если «кидаются» данные нестандартного типа (экземпляры определенных программистом классов, структур и т.д.), лучше «ловить» их по ссылке, иначе вся «кидаемая» переменная будет скопирована в стек вместо того, чтобы просто передать указатель на нее. Если кидаются данные нескольких типов и необходимо «поймать» конкретную переменную (вернее, переменную конкретного типа), то можно использовать несколько блоков *catch*, «ловящих» «свой» тип данных:

```
try {
    throw 1;
    // throw 'a';
}
catch (long b) {
    cout << "пойман тип long: " << b << endl;
}
catch (char b) {
    cout << "пойман тип char: " << b << endl;
}
```

### ***Операторы throw без параметров***

Блок *try-catch* может содержать вложенные блоки *try-catch*, и, если не будет определено соответствующего оператора *catch* на текущем уровне вложения, исключение будет поймано на более высоком уровне. Единственная вещь, о которой необходимо помнить, — это то, что операторы, следующие за *throw*, никогда не выполняются. Такой метод может применяться в случаях, когда не нужно передавать никаких данных в блок *catch*.

```
try
{
```

```

        throw;
        /* ни один оператор, следующий далее (до за-
        крывающей скобки), выполнен не будет */
    }
    catch(...)
    {
        cout << "Исключение!" << endl;
    }

```

## Вопросы и задания

1. Какие классы памяти создают переменные, локальные для содержащей их функции?

2. Какие классы памяти создают переменные, которые существуют в течение выполнения содержащей их программы?

3. Файл начинается со следующих объявлений:

```

static int plink;
int value_ct (const int arr[], int value, int n);

```

а) что эти объявления говорят о намерениях программиста?

б) приведет ли замена операторов *int value* и *int n* на *const int value* и *const int n* к усилению степени защиты значений в вызывающей программе?

4. Напишите и протестируйте в цикле функцию, которая возвращает количество ее вызовов.

5. Назовите пространства имен в C++.

6. Можно ли создать структурный тип данных и переменную такого типа с одинаковыми именами? Почему?

7. В каком разделе программы объявляются глобальные переменные, локальные переменные?

8. Дайте определение понятию «область действия идентификатора».

9. Пусть переменная объявлена как регистровая, но при работе программы свободных регистров не оказалось. Какого класса памяти будет эта переменная?

10. Что такое исключение?

11. Назовите типы исключений.

12. Поясните принцип работы операторов обработки исключительных ситуаций *try-catch-throw*.



13. Что означает оператор, встреченный при определении функции *throw (...)*?

14. В чем заключается принцип работы механизма глобальной раскрутки, применяемый при обработке исключительных ситуаций?

## Лекция 15. СТРУКТУРЫ КОМПЬЮТЕРНОЙ ОБРАБОТКИ ДАННЫХ

### § 15.1. Связные списки

Связный список (*linked list*) – это структура данных, в которой объекты расположены в линейном порядке. Однако в отличие от массива, в котором этот порядок определен индексами, порядок в связном списке определяет указатель на каждый объект.

Связные списки могут быть односвязными (однонаправленными) и двусвязными (двунаправленными). Односвязный список содержит ссылку на следующий элемент данных. Двусвязный список содержит ссылки как на последующий, так и на предыдущий элементы списка. Выбор типа применяемого списка зависит от конкретной задачи.

#### *Односвязные списки*

В односвязном списке каждый элемент информации содержит ссылку на следующий элемент списка. Каждый элемент данных обычно представляет собой структуру, которая состоит из информационных полей и указателя связи. Концептуально односвязный список выглядит так, как показано на рис. 15.1.

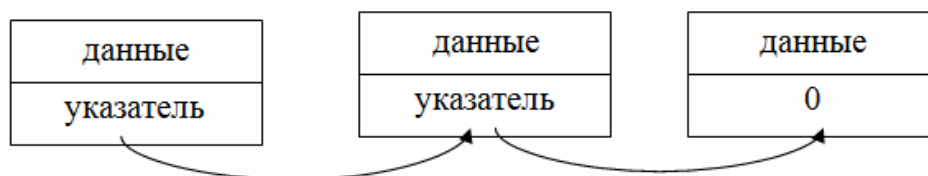


Рис. 15.1. Односвязный список

Существуют два основных способа построения односвязного списка: первый способ – помещать новые элементы в конец списка,

второй – вставлять элементы в определенные позиции списка, например в порядке возрастания. От способа построения списка зависит алгоритм функции добавления элемента.

Для создания динамического списка необходимо определить структуру, которая помимо информационных полей будет содержать поле-указатель на объявленную структуру.

Пример:

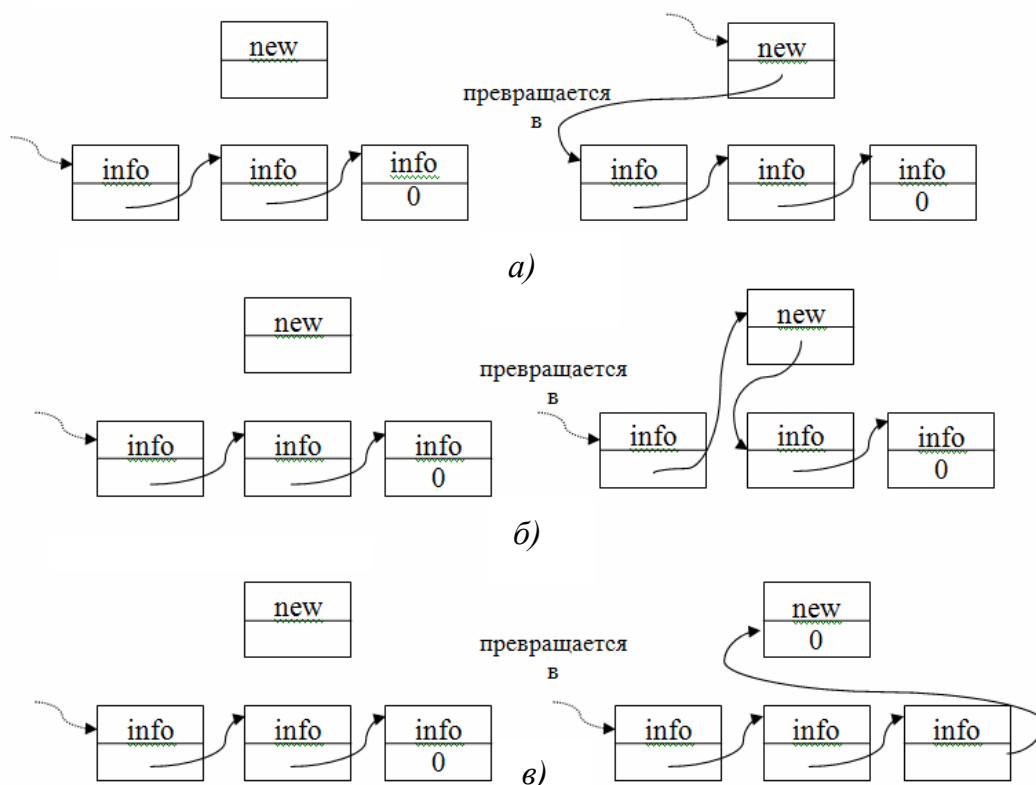
```
struct sp
{
    int x;
    sp *next;
};
```

Принцип построения динамического списка по первому способу таков: необходимо объявить указатель, который называется *головным указателем*. Если значение головного указателя равно *NULL*, это означает, что список пуст. Добавление нового элемента происходит в конец списка.

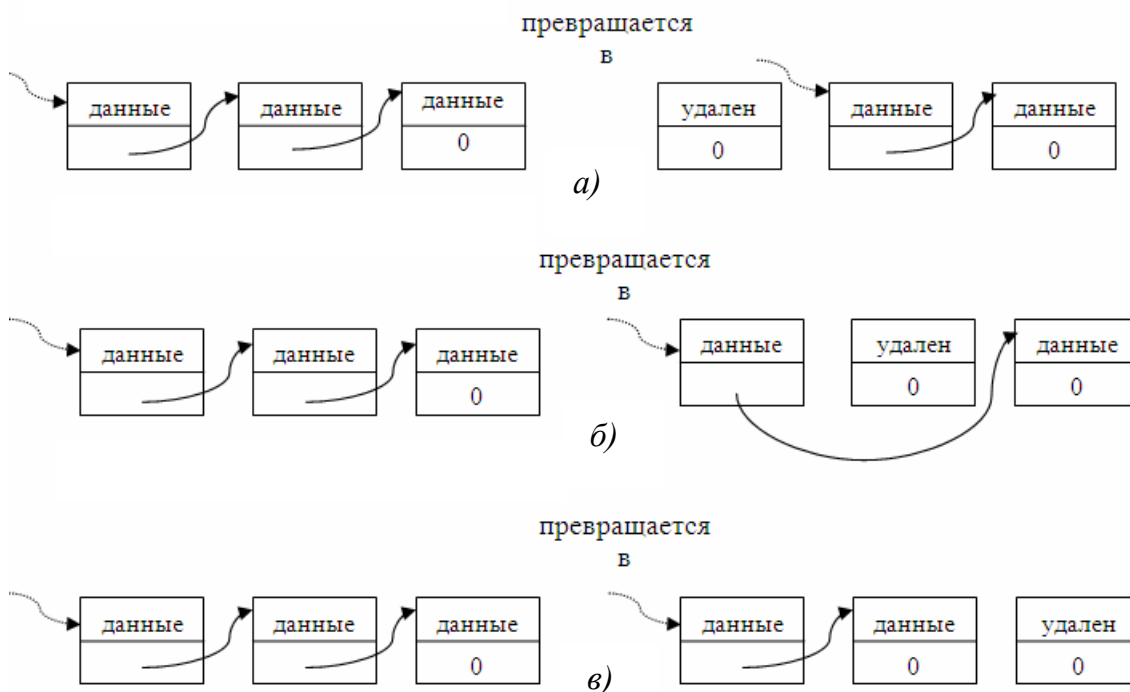
Полученный список можно отсортировать отдельной операцией уже после его создания, но легче сразу создавать упорядоченный список, вставляя новый элемент в нужное место в последовательности. Кроме того, если список уже отсортирован, имеет смысл поддерживать его упорядоченность, вставляя новые элементы в соответствующие позиции. Для вставки элемента таким способом требуется последовательно просматривать список до тех пор, пока не будет найдено место нового элемента, затем вставить в найденную позицию новую запись и переустановить ссылки.

При вставке элемента в односвязный список может возникнуть одна из трех ситуаций: элемент становится первым, элемент вставляется между двумя другими, элемент становится последним (рис. 15.2). Следует помнить, что при вставке элемента в начало списка необходимо изменить адрес входа в список где-то в другом месте программы.

Удаление элемента из односвязного списка выполняется просто. Так же как и при вставке, возможны три случая: удаление первого элемента, удаление элемента в середине, удаление последнего элемента (рис. 15.3).



**Рис. 15.2. Вставка нового элемента в односвязный список: а – вставка в начало списка; б – вставка в середину списка; в – вставка в конец списка**



**Рис. 15.3. Удаление элемента из односвязного списка: а – удаление первого элемента списка; б – удаление среднего элемента списка; в – удаление последнего элемента списка**

### Двусвязные списки

Двусвязный список (рис. 15.4) состоит из элементов данных, каждый из которых содержит ссылки как на следующий, так и на предыдущий элементы.

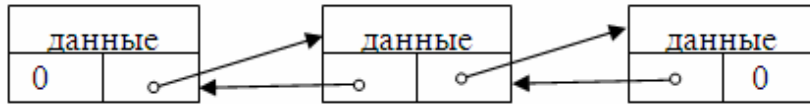


Рис. 15.4. Двусвязный список

Наличие двух ссылок вместо одной предоставляет несколько преимуществ. Наиболее важное из них состоит в том, что перемещение по списку возможно в обоих направлениях. Это упрощает работу со списком, в частности вставку и удаление. Помимо этого, пользователь может просматривать список в любом направлении. Еще одно преимущество имеет значение только при некоторых сбоях: поскольку весь список можно пройти не только по прямым, но и по обратным ссылкам, то в случае, если какая-то из ссылок станет неверной, целостность списка можно восстановить по другой ссылке.

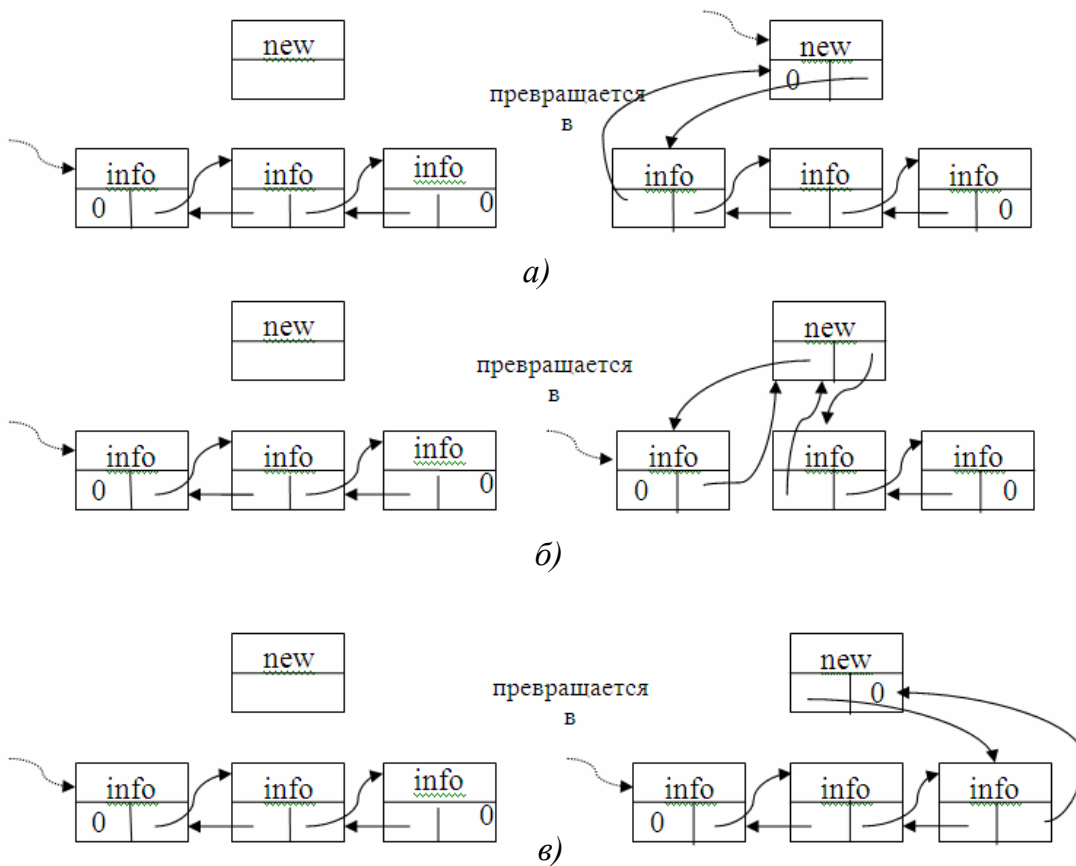
При вставке нового элемента в двусвязный список могут возникнуть три ситуации (рис. 15.5): элемент вставляется в начало, в середину и в конец списка.

Построение двусвязного списка выполняется аналогично построению односвязного за исключением того, что необходимо установить две ссылки, поэтому в структуре должны быть описаны два указателя связи:

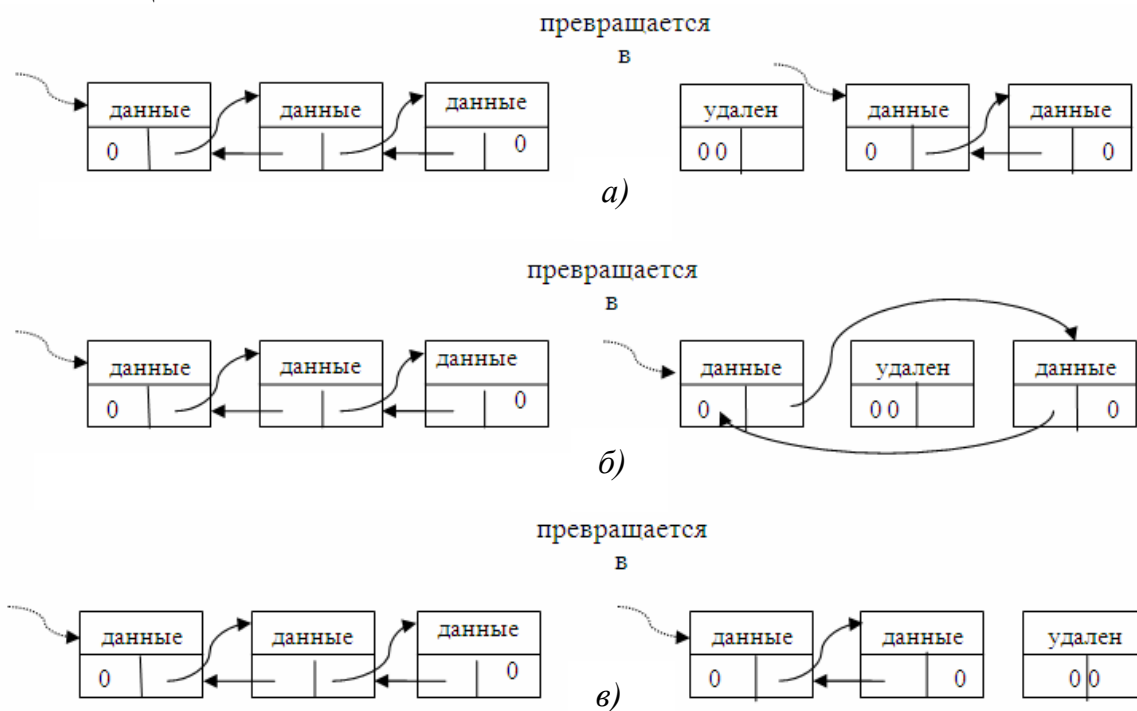
```
struct sp {  
    int x;  
    sp *next, *prev;  
};
```

Как и в односвязных списках, для получения элемента данных двусвязного списка необходимо переходить по ссылкам до тех пор, пока не будет найден искомый элемент.

При удалении элемента двусвязного списка могут возникнуть три ситуации (рис. 15.6): удаление первого элемента, удаление элемента из середины и удаление последнего элемента.



**Рис. 15.5.** Добавление элемента в двусвязный список: а – вставка элемента в начало списка; б – вставка элемента в середину списка; в – вставка элемента в конец списка



**Рис. 15.6.** Удаление элемента двусвязного списка: а – удаление первого элемента списка; б – удаление элемента из середины списка; в – удаление последнего элемента списка

## Стеки

Стек является противоположностью очереди, поскольку работает по принципу «последним вошел – первым вышел» (*last-in, first-out, LIFO*).

Операцию вставки применительно к стекам часто называют *PUSH* (запись в стек), а операцию удаления – *POP* (снятие со стека).

Стек, способный вместить не более  $n$  элементов, можно реализовать с помощью массива  $S[1..n]$ , который обладает атрибутом  $top[S]$ , представляющим собой индекс последнего помещенного в стек элемента. Стек состоит из элементов  $S[1..top[S]]$ , где  $S[1]$  – элемент на дне стека, а  $S[top[S]]$  – элемент на его вершине.

Если  $top[S]=0$ , то стек не содержит ни одного элемента и является пустым. Протестировать стек на наличие в нем элементов можно с помощью операции запроса *Stack\_Empty*. Если элемент снимается с пустого стека, говорят, что он *опустошается (underflow)*, что обычно приводит к ошибке. Если значение  $top[S]$  больше  $n$ , то стек *переполняется (overflow)*. (В приведенном ниже примере переполнение стека не учитывается).

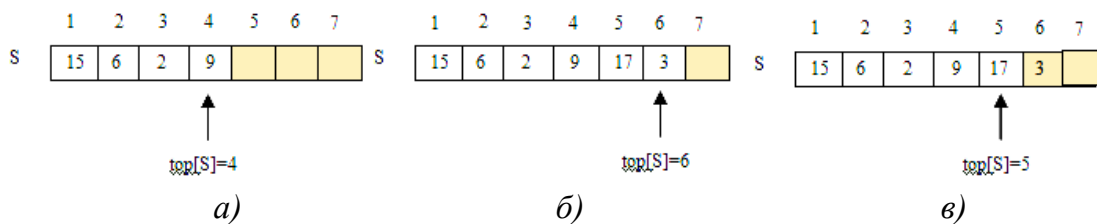
Каждую операцию над стеком можно легко реализовать несколькими строками кода:

```
Stack_Empty (S)
1 if top[S] = 0
2 then return TRUE
3 else return FALSE
```

```
PUSH(S, x)
1 top[S] ← top[S] + 1
2 S[top[S]] ← x
```

```
POP(S)
1 if Stack_Empty(S)
2 then error "underflow"
3 else top[S] ← top[S] - 1
4 return S[top[S] + 1]
```

На рис. 15.7 видно, какое воздействие на стек оказывают модифицирующие операции *PUSH* и *POP*. Элементы стека находятся только в тех позициях массива, которые отмечены белым цветом.



**Рис. 15.7. Реализация стека S в виде массива**

На рис. 15.7, а изображен стек  $S$ , состоящий из четырех элементов, на вершине которого находится элемент 9. На рис. 15.7, б представлен этот же стек после вызова процедур  $PUSH(S, 17)$  и  $PUSH(S, 3)$ , а на рис. 15.7, в – после вызова процедуры  $POP(S)$ , которая возвращает помещенное в стек последним значение 3. Несмотря на то, что элемент 3 все еще показан в массиве, он больше не принадлежит стеку; теперь на вершине стека – 17.

### Очереди

Очередь – это линейный список информации, работа с которой происходит по принципу «первый пришел – первым вышел» (*first-in, first-out*). Этот принцип (и очередь как структура данных) иногда еще называют *FIFO*. Это значит, что первый помещенный в очередь элемент будет получен первым, второй помещенный элемент будет вторым и т.д. Это единственный способ работы с очередью; произвольный доступ к отдельным элементам не разрешается.

Применительно к очередям операция вставки называется *ENQUEUE* («поместить в очередь»), а операция удаления – *DEQUEUE* («вывести из очереди»). Подобно стековой операции *POP*, операция *DEQUEUE* не требует передачи элемента массива в виде аргумента. У очереди имеется *голова (head)* и *хвост (tail)*. Когда элемент ставится в очередь, он занимает место в хвосте. Из очереди всегда выводится элемент, который находится в ее головной части.

На рис. 15.8 показан один из способов, который позволяет с помощью массива  $Q[1..n]$  реализовать очередь, состоящую не более чем из  $n - 1$  элементов. Эта очередь обладает атрибутом  $head[Q]$ , который является индексом головного элемента или указателем на него; атрибут  $tail[Q]$  индексирует местоположение, куда будет до-

бавлен новый элемент. Элементы очереди расположены в ячейках  $head[Q]$ ,  $head[Q]+1$ , ...,  $tail[Q]-1$ , которые циклически замкнуты в том смысле, что ячейка 1 следует сразу же после ячейки  $n$  в циклическом порядке. При выполнении условия  $head[Q] = tail[Q]$  очередь пуста. Изначально выполняется условие  $head[Q] = tail[Q] = 1$ . Если очередь пустая, то при попытке удалить из нее элемент происходит ошибка опустошения. Если  $head[Q] = tail[Q] + 1$ , то очередь заполнена, и попытка добавить в нее элемент приводит к ее переполнению.

В следующих процедурах *ENQUEUE* и *DEQUEUE* проверка опустошения и переполнения не производится.

```

Enqueue(Q, x)
1 Q[tail[Q]] ← x
2 if tail[Q] = length[Q]
3 then tail[Q] ← 1
4 else tail[Q] ← tail[Q] + 1

```

```

Dequeue(Q)
1 x ← Q[head[Q]]
2 if head[Q] = length[Q]
3 then head[Q] ← 1
4 else head[Q] ← head[Q] + 1
5 return x

```

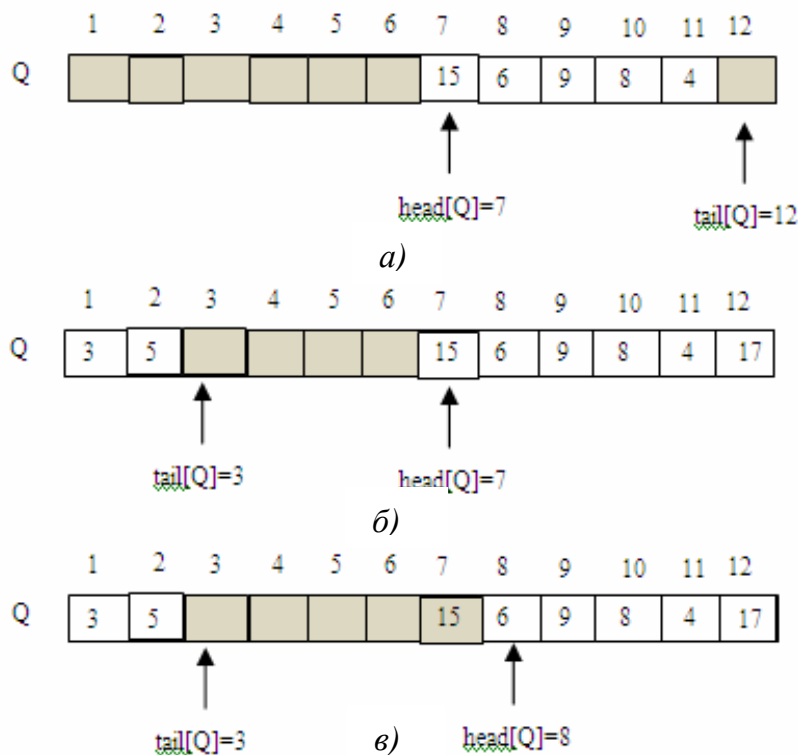


Рис. 15.8. Очередь, реализованная с помощью массива  $Q[1..12]$



На рис. 15.8 показана работа процедур *ENQUEUE* и *DEQUEUE*. Элементы очереди содержатся только в белых ячейках. На рис. 15.8, *а* изображена очередь, состоящая из пяти элементов, расположенных в ячейках  $Q[7..11]$ . На рис. 15.8, *б* показана эта же очередь после вызова процедур *ENQUEUE(Q, 17)*, *ENQUEUE(Q, 3)*, *ENQUEUE(Q, 5)*, а на рис. 15.8, *в* – конфигурация очереди после вызова процедуры *DEQUEUE(Q)*, возвращающей значение ключа 15, которое до этого находилось в голове очереди. Значение ключа новой головы очереди равно 6.

## § 15.2. Табличное хранение

Для многих приложений достаточно использовать динамические множества, поддерживающие только стандартные словарные операции вставки, поиска и удаления. Например, компилятор языка программирования поддерживает таблицу символов, в которой ключами элемента являются произвольные символьные строки, соответствующие идентификаторам в языке. Хеш-таблица представляет собой эффективную структуру данных для реализации словарей. Хотя на поиск элемента в хеш-таблице может в крайнем случае потребоваться столько же времени, что и в связном списке, на практике хеширование эффективно.

Хеш-таблица представляет собой обобщение обычного массива. Возможность прямой адресации элементов обычного массива обеспечивает доступ к произвольной позиции в массиве за время  $O(1)$ . Прямая адресация применима, если мы в состоянии выделить массив размера, достаточного для того, чтобы для каждого возможного значения ключа имелась своя ячейка.

Если количество реально хранящихся в массиве ключей мало по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится хеш-таблица, которая обычно использует массив с размером, пропорциональным количеству реально хранящихся в нем ключей. Вместо непосредственного использования ключа в качестве индекса массива, индекс вычисляется по значению ключа.

### Таблицы с прямой адресацией

Прямая адресация представляет собой простейшую технологию, которая хорошо работает для небольших множеств ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из множества  $U = \{0, 1, \dots, m-1\}$ , где  $m$  не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей.

Для представления динамического множества программисты используют массив, или *таблицу с прямой адресацией*, который обозначают как  $T[0..m-1]$ , каждая *позиция*, или *ячейка*, которого соответствует ключу из пространства  $U$  (рис. 15.9).

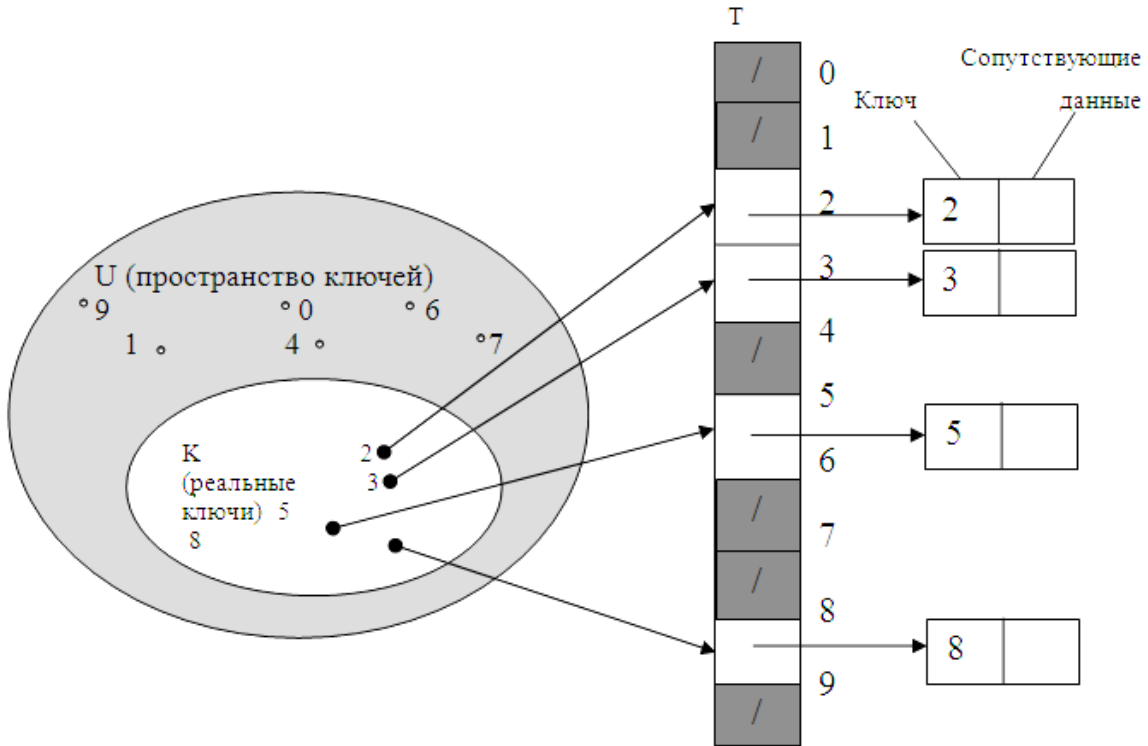


Рис. 15.9. Реализация динамического множества с использованием таблицы с прямой адресацией

Ячейка  $T[k]$  указывает на элемент множества с ключом  $k$ . Если множество не содержит элемента с ключом  $k$ , то  $T[k] = NIL$ . На рис. 15.9 каждый ключ из пространства  $U = \{0, 1, \dots, 9\}$  соответствует индексу таблицы. Множество различных ключей  $K = \{2, 3, 5, 8\}$  определяет ячейки таблицы, которые содержат указатели на элементы. Остальные ячейки (закрашенные темным цветом) содержат  $NIL$ .

Реализация словарных операций тривиальна:

```
Direct_Address_Search (T, k)
    return T[k]
```

```
Direct_Address_Insert (T, x)
    T[key[x]] ← x
```

```
Direct_Address_Delete (T, x)
    T[key[x]] ← NIL
```

В некоторых приложениях элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией, т.е. месте хранения ключей и сопутствующих данных элементов в объектах, внешних по отношению к таблице с прямой адресацией, а в таблице – указателей на эти объекты, которые можно хранить непосредственно в ячейках таблицы (что приводит к экономии используемой памяти). Кроме того, зачастую хранение ключа не является необходимым условием, поскольку если мы знаем индекс объекта в таблице, то мы знаем и его ключ. Однако если ключ не хранится в ячейке таблицы, то нам нужен какой-то иной механизм для того, чтобы помечать пустые ячейки.

### *Хеш-таблицы*

Недостаток прямой адресации очевиден: если пространство ключей  $U$  велико, хранение таблицы  $T$  размером  $|U|$  непрактично, а то и вовсе невозможно – в зависимости от количества доступной памяти и размера пространства ключей. Кроме того, множество  $K$  реально сохраненных ключей может быть мало по сравнению с пространством ключей  $U$ , а в этом случае память, выделенная для таблицы  $T$ , в основном, расходуется напрасно.

Когда множество  $K$  хранящихся в словаре ключей гораздо меньше пространства возможных ключей  $U$ , хеш-таблица требует существенно меньше места, чем таблица с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до  $\Theta(|K|)$ , при этом время поиска элемента в хеш-таблице остается равным  $O(1)$ .

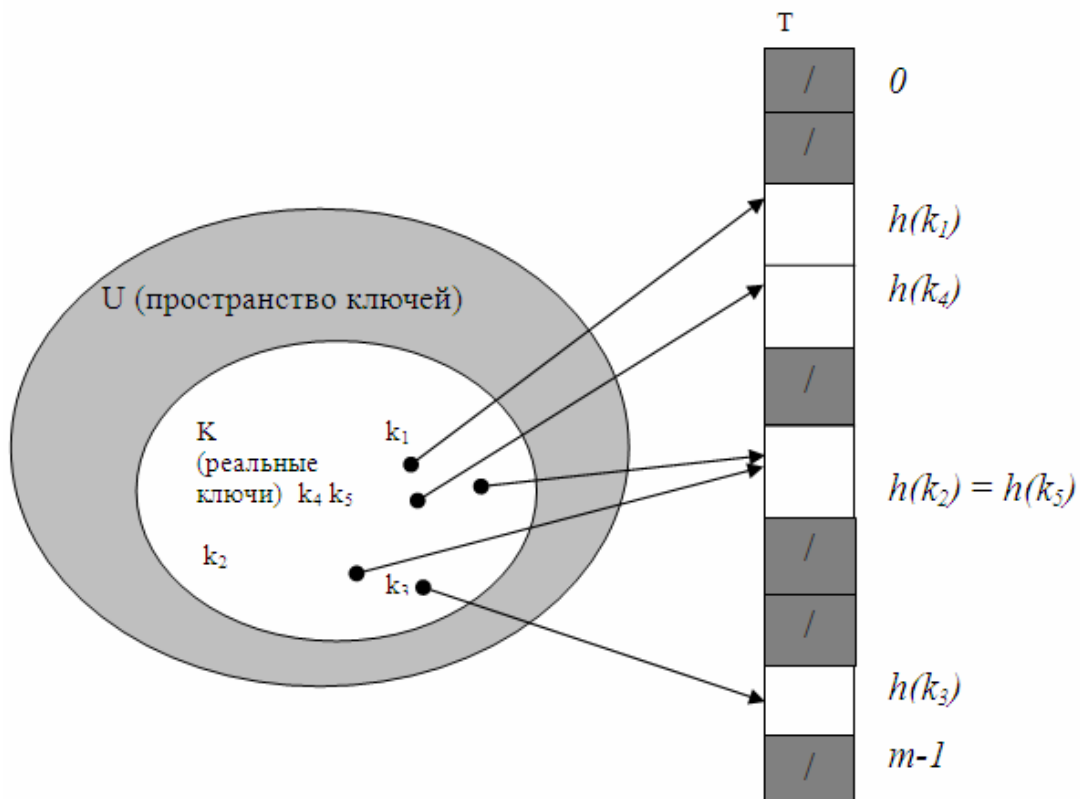
В случае прямой адресации элемент с ключом  $k$  хранится в ячейке  $k$ . При хешировании этот элемент хранится в ячейке  $h(k)$ , т.е. мы используем *хеш-функцию*  $h$  для вычисления ячейки для дан-

ного ключа  $k$ . Функция  $h$  отображает пространство ключей  $U$  на ячейки хеш-таблицы  $T[0 .. m - 1]$ :

$$h : U \rightarrow \{0, 1, \dots, m - 1\}.$$

Будем говорить, что элемент с ключом  $k$  хешируется в ячейку  $h(k)$ ; величина  $h(k)$  называется хеш-значением ключа  $k$ . На рис. 15.10 представлена основная идея хеширования.

Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива, и вместо  $|U|$  значений мы можем обойтись всего лишь  $m$  значениями. Соответственно снижаются и требования к количеству памяти.



**Рис. 15.10.** Использование хеш-функции  $h$  для отображения ключей в ячейки хеш-таблицы

Однако здесь есть одна проблема: два ключа могут быть хешированы в одну и ту же ячейку. Такая ситуация называется *коллизией*. Решения данной проблемы существуют.

Конечно, идеальным решением было бы полное устранение коллизий. Мы можем попытаться добиться этого путем выбора подходящей хеш-функции  $h$ . Одна из идей заключается в том, что-

бы сделать  $h$  «случайной», что позволило бы избежать коллизий или хотя бы минимизировать их количество. Функция  $h$  должна быть детерминистической и для одного и того же значения  $k$  всегда давать одно и то же хеш-значение  $h(k)$ . Однако, поскольку  $|U| > m$ , должны существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Таким образом, полностью избежать коллизий невозможно в принципе, и хорошая хеш-функция в состоянии только минимизировать их количество.

### Разрешение коллизий при помощи цепочек

При использовании данного метода мы объединяем все элементы, хешированные в одну и ту же ячейку, в связный список, как показано на рис. 15.11.

Ячейка  $j$  содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно  $j$ ; если таких элементов нет, ячейка содержит значение  $NIL$ . На рис. 15.11 показано разрешение коллизий, возникающих из-за того, что  $h(k_1) = h(k_4)$ ,  $h(k_5) = h(k_2) = h(k_7)$  и  $h(k_8) = h(k_6)$ .

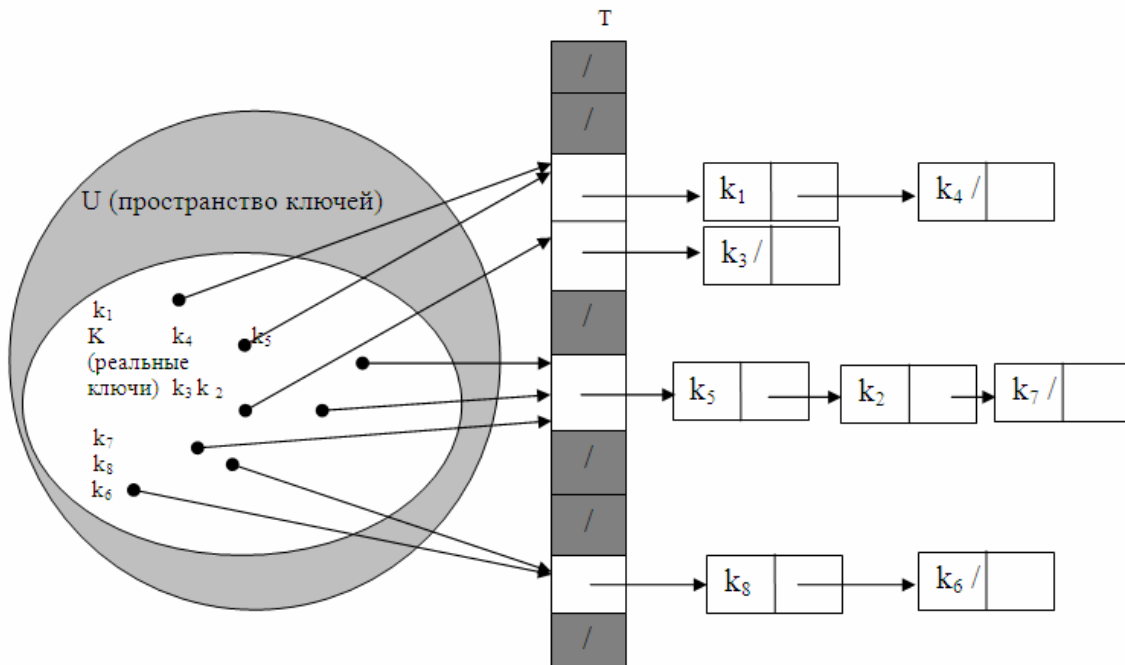


Рис. 15.11. Разрешение коллизий при помощи цепочек

Словарные операции в хеш-таблице с использованием цепочек для разрешения коллизий реализуются очень просто:

`Chained_Hash_Insert (T, x)`  
Вставить  $x$  в заголовок списка  $T[h(key[x])]$

`Chained_Hash_Search (T, k)`  
Поиск элемента с ключом  $k$  в списке  $T[h(k)]$

`Chained_Hash_Delete (T, x)`  
Удаление  $x$  из списка  $T[h(key[x])]$

Время, необходимое для вставки, в крайнем случае равно  $O(1)$ . Процедура вставки выполняется очень быстро, поскольку предполагается, что вставляемый элемент отсутствует в таблице. При необходимости это предположение может быть проверено путем выполнения поиска перед вставкой. Время работы поиска в крайнем случае пропорционально длине списка. Удаление элемента может быть выполнено за время  $O(1)$  при использовании двусвязных списков. Обратите внимание на то, что процедура *Chained\_Hash\_Delete* принимает в качестве аргумента элемент  $x$ , а не его ключ, поэтому нет необходимости в предварительном поиске  $x$ . Если список односвязный, то передача в качестве аргумента  $x$  не дает нам особого выигрыша, поскольку для корректного обновления поля *next* предшественника  $x$  нам все равно надо выполнить поиск  $x$  в списке  $T[h(key[x])]$ . В таком случае, как нетрудно понять, удаление и поиск имеют по сути одно и то же время работы.

### Вопросы и задания

1. Что такое линейный список?
2. Перечислите виды линейных списков.
3. Какого вида структуру данных необходимо определить для создания двусвязного линейного списка?
4. Что такое стек? Назовите принцип работы стека.
5. Что такое очередь? Назовите принцип работы очереди.
6. Какие операции можно выполнять с линейными списками?
7. Что такое хеш-таблицы?

## ЗАКЛЮЧЕНИЕ

Курс лекций по дисциплине «Программирование», не делая акцент ни на одной области практики, знакомит студентов с основными принципами автоматизированного решения задач с применением ЭВМ. В частности, рассмотрены принципы разработки алгоритмов (как простейших, так и комбинированных), изложены семантика и синтаксис высокоуровневого языка программирования C++.

Конечно, данное издание не может охватить весь объем знаний по программированию, однако имеющаяся в нем информация достаточно полно отражает все главные задачи дисциплины, демонстрирует основные способы их решения.

После изучения материалов данного пособия студент или любой другой изучающий данную дисциплину человек получит достаточный объем знаний, необходимых для решения серьезных прикладных задач. Курс лекций снабжен списком литературы, в которой любой желающий может найти более полную информацию по узким вопросам данной дисциплины.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

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. *Кормен, Т. Х.* Алгоритмы: построение и анализ : пер. с англ. / Т. Х. Кормен [и др.]. – 2-е изд. – М. : Вильямс, 2007. – 1296 с. – ISBN 5-8459-0857-4 (рус.).

3. *Прата, С.* Язык программирования С. Лекции и упражнения : пер. с англ. / Стивен Прата. – Киев : ДиаСофт, 2000. – 432 с. – ISBN 966-7393-50-X.

4. *Страуструп, Б.* Язык программирования С++ / Б. Страуструп. – М. : Бином-Пресс, 2008. – 1054 с. – ISBN 5-7989-0226-2.

5. *Шилдт, Г.* Полный справочник по С : пер. с англ. / Г. Шилдт. – 4-е изд. – М. : Вильямс, 2004. – 704 с. – ISBN 5-8459-0226-6.



*Учебное издание*

МЕДВЕДЕВА Ольга Николаевна

ПРОГРАММИРОВАНИЕ

Курс лекций

Подписано в печать 21.02.11.

Формат 60x84/16. Усл. печ. л. 8,60. Тираж 100 экз.

Заказ

Издательство

Владимирского государственного университета.

600000, Владимир, ул. Горького, 87.