М. П. Батура, В. Л. Бусько, А. Г. Корбит, Т. М. Кривоносова

# *ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ. ЯЗЫК СИ*

*Допущено Министерством образования Республики Беларусь*
*в качестве учебного пособия*
*для студентов учреждений, обеспечивающих получение*
*высшего образования по специальностям «Искусственный интеллект»,*
*«Программное обеспечение информационных технологий»,*
*«Автоматизированные системы обработки информации»,*
*«Электронные вычислительные средства»,*
*«Инженерно-психологическое обеспечение информационных технологий»*



Минск БГУИР 2007

УДК 621.3 (075.8)
ББК 22.193 я 73
Б 28

Р е ц е н з е н т ы :
зав. кафедрой алгоритмики и дискретной математики БГУ,
д-р техн. наук, проф. В. М. Котов;
начальник кафедры систем автоматического управления Военной академии
Республики Беларусь, д-р техн. наук, проф. В. А. Куренев



**Батура, М. П.**

Б 28 Основы алгоритмизации и программирования. Язык Си : учеб. пособие / М. П. Батура, В. Л. Бусько, А. Г. Корбит, Т. М. Кривоносова. – Минск : БГУИР, 2007. – 240 с. : ил.

ISBN 978-985-488-192-8

Материал пособия составлен на основе курса лекций по дисциплине «Основы алгоритмизации и программирования», читаемого авторами в Белорусском государственном университете информатики и радиоэлектроники на факультете информационных технологий и управления.

Содержание пособия охватывает темы, посвященные основным конструкциям языка Си. Приведенные примеры, иллюстрирующие основные возможности языка, прошли проверку в качестве консольных приложений среды программирования *Visual* C++ 6.0.

Неотъемлемой частью учебного пособия являются индивидуальные задания для практических и лабораторных работ. В приложениях рассматриваются некоторые элементы языка C++, приведены дополнительные задания.

**УДК 621.3 (075.8)**
**ББК 22.193 я 73**

**ISBN 978-985-488-192-8**

# СОДЕРЖАНИЕ

# ПРЕДИСЛОВИЕ

Алгоритмический язык Си был разработан в 1972 г. сотрудником фирмы *AT&T Bell Laboratory* ***Денисом Ритчи*** на базе языка *B* (автор К.Томпсон), который в свою очередь основывался на языке системного программирования *BCPL*. Первая версия языка была опубликована в книге авторов Б. Кернигана и Д. Ритчи и получила название стандарт *K&R*. Минимальная стандартная реализация, поддерживаемая любым компилятором, содержала всего 27 ключевых слов. Началось успешное развитие языка и, чтобы избежать путаницы, Американский институт стандартизации (*American National Standart Institute*) ввел в 1983 г. общий стандарт языка – ***ANSI***-стандарт.

Язык продолжает развиваться, и в 1985 г. появляется язык C++, который в основном сохраняет все черты обычного Си, но дополнен новыми существенными возможностями, которые позволили реализовать объектно-ориентированный стиль программирования.

Язык Си отражает возможности современных компьютеров и обладает следующими достоинствами – это современный, эффективный, переносимый (мобильный), мощный и гибкий, удобный и обладающий рядом присущих ассемблеру управляющих конструкций язык высокого уровня.

Программы на Си отличаются компактностью и быстротой исполнения. Структура языка Си побуждает программиста использовать в своей работе нисходящее программирование, структурное программирование, пошаговую разработку модулей.

Любой язык представляет собой описание определенного набора действий (операции, операторы, функции) над некоторыми данными. Данным, над которыми выполняются эти действия, вместо номеров ячеек в памяти принято давать имена (идентификаторы), а содержимое ячеек называть переменными, или константами, в зависимости от того, изменяется значение в процессе работы или нет.

Под ***системой программирования*** понимают совокупность языка программирования и виртуальной машины, обеспечивающей выполнение программ, составленных на этом языке.

*Виртуальная машина* – это программный комплекс, имитирующий работу реальной машины с определенным входным языком на ЭВМ с другим машинным языком, т.е. реализующий входной язык программирования. Виртуальная машина содержит транслятор и/или интерпретатор и может включать библиотеки стандартных подпрограмм, отладчик, компоновщик и другие сервисные средства.

***Транслятор*** представляет собой программу, осуществляющую перевод текстов с входного языка на машинный язык. Одной из разновидностей транслятора является ***компилятор***, обеспечивающий перевод программ с языка высокого уровня (приближенного к человеку) на язык более низкого уровня (близкий к ЭВМ), или машинозависимый язык.

Текст программы, записанный на *языке высокого уровня* и введенный с помощью клавиатуры в память компьютера, – *исходный модуль*. Программы, написанные в среде программирования, предназначенной для языка Си, например *Turbo C*, имеют расширение **.c*. Расширение **.cpp* имеют программы, написанные в интегрированных средах *Borland C++*, *Visual C++*, *Builder C++*, предназначенных для написания программ как на языке Си, так и на языке C++.

Большинство трансляторов языка Си – компиляторы.

Результат обработки исходного модуля компилятором – ***объектный модуль*** (расширение **.obj*). На этом этапе компилятор выделяет лексемы (элементарные конструкции языка), затем на основе грамматики распознает выражения и операторы, построенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и, в случае их отсутствия, создает объектный модуль.

Исполняемый (абсолютный, загрузочный) модуль создает вторая специальная программа – «компоновщик». Ее еще называют редактором связей (*Linker*). Она и создает ***загрузочный модуль*** (расширение **.exe*) на основе одного или нескольких объектных модулей – это программный модуль, представленный в форме, пригодной для выполнения.

Главным классифицирующим признаком языков и, следовательно, систем программирования, является принадлежность к одному из оформившихся к настоящему времени стилей программирования, основные среди которых – процедурное, функциональное, логическое и объектно-ориен-
тированное.

# ГЛАВА 1. Введение в алгоритмы

## *1.1. Этапы решения задач на ЭВМ*

Условно ***программированием*** можно назвать научную и практическую деятельность по созданию программ. Основной частью программирования является процесс решения задачи на ЭВМ, который можно разбить на следующие этапы:

1) математическая или информационная формулировка задачи;

2) выбор численного или иного метода решения поставленной задачи;

3) построение алгоритма решения поставленной задачи;

4) выбор языка программирования и запись построенного алгоритма по его правилам, т.е. написание текста программы;

5) отладка программы – это процесс обнаружения, локализации и устранения возможных ошибок;

6) выполнение программы, т.е. получение требуемого результата.

Рассмотрим более подробно некоторые наиболее важные из приведенных этапов.

## *1.2. Понятие алгоритма*

Понятие алгоритма занимает центральное место в современной математике и программировании.

***Алгоритмизация*** – сведение задачи к последовательным этапам действий так, что результаты предыдущих действий используются при выполнении последующих.

Рассмотрим вначале некоторые наиболее важные (фундаментальные) понятия программирования.

1. ***Действие*** – это некоторая операция, имеющая конкретную продолжительность и приводящая к совершенно конкретному результату.

2. Каждое действие предполагает наличие некоторых данных, над которыми это действие совершается и по изменению состояния которых определяют результат этого действия.

3. Каждое действие должно быть таким, чтобы его можно было описать при помощи какого-либо языка (или набора формул); такое описание называют ***инструкция***.

4. Если действие можно разложить на составные части, то его называют ***процессом*** (или вычислением).

5. Описание характера проведения процесса, т.е. последовательности выполняемых действий без привязки к какому-то конкретному процессору, называют ***алгоритмом.***

***Числовой алгоритм*** – детально описанный способ преобразования числовых входных данных в выходные при помощи математических операций. Существуют нечисловые алгоритмы, которые используются в экономике, технике и научных исследованиях.

В общем, ***алгоритм*** – строгий и четкий набор правил, определяющий последовательность действий, приводящих к достижению поставленной цели.

### *1.3. Свойства алгоритмов*

***Дискретность*** – значения новых величин (данных) вычисляются по определенным правилам из других величин с уже известными значениями.

***Определенность*** (детерминированность) – каждое правило из набора однозначно, а сами данные однозначно связаны между собой, т.е. последовательность действий алгоритма строго и точно определена.

***Результативность*** (конечность) – алгоритм решает поставленную задачу за конечное число шагов.

***Массовость*** – алгоритм разрабатывается так, чтобы его можно было применить для целого класса задач, например, алгоритм вычисления определенных интегралов с заданной точностью.

### *1.4. Сложность алгоритма*

Выполнение любого алгоритма требует определенного объема памяти компьютера для размещения данных и программы, а также времени по обработке этих данных – эти ресурсы ограничены и, следовательно, правомочен вопрос об эффективности их использования. Таким образом, в самом широком смысле понятие эффективности связано со всеми вычислительными ресурсами, необходимыми для работы алгоритма.

Однако обычно под *«самым эффективным»* понимается алгоритм, обеспечивающий наиболее быстрое получение результата, поэтому рассмотрим именно временну́ю сложность алгоритмов.

Время работы алгоритма удобно выражать в виде функции от одной переменной, характеризующей *«размер»* конкретной задачи, т.е. объем входных данных, необходимых для ее решения. Тогда сравнительная сложность задач и может оцениваться через ее размер.

Поскольку описание задачи, предназначенной для решения посредством вычислительного устройства, можно рассматривать в виде слова конечной длины, представленной символами конечного алфавита, в качестве формальной характеристики размера задачи можно принять длину входного слова. Например, если стоит задача определения максимального числа в некоторой последовательности из $n$ элементов, то и размер задачи будет $n$, поскольку любой вариант входной последовательности можно задать словом из $n$ символов.

***Временная сложность*** алгоритма – это функция, которая каждой входной длине слова $n$ ставит в соответствие *максимальное* (для всех конкретных задач длиной $n$) время, затрачиваемое алгоритмом на ее решение.

Различные алгоритмы имеют различную временную сложность и выяснение того, какие из них окажутся *достаточно эффективны*, а какие

нет, определяется многими факторами. Однако для сравнения эффективности алгоритмов был предложен простой подход, позволяющий прояснить ситуацию. Речь идет о различии между *полиномиальными* и *экспоненциальными* алгоритмами.

*Полиномиальным* называется алгоритм, временна̀я сложность которого выражается некоторой полиномиальной функцией размера задачи $n$. Алгоритмы, временна́я сложность которых не поддается подобной оценке, называются *экспоненциальными*.

Задача считается *труднорешаемой*, если для нее не удается построить полиномиального алгоритма. Это утверждение не является категорическим, поскольку известны задачи, в которых достаточно эффективно работают и экспоненциальные алгоритмы. Примером может служить симплекс-метод, который успешно используется при решении задач линейного программирования, имея функцию сложности $f(n) = 2^n$. Однако подобных примеров не очень много, и общей следует признать ситуацию, когда эффективно исполняемыми можно считать полиномиальные алгоритмы с функциями сложности $n$, $n^2$ или $n^3$.

Например, при решении задачи поиска нужного элемента из $n$ имеющихся в худшем варианте сложность равна $n$; если же оценить среднюю трудоемкость (продолжительность поиска), то она составит $(n+1)/2$ – в обоих случаях функция сложности оказывается линейной $n$.

Сложность задачи вычисления определителя системы $n$ линейных уравнений с $n$ неизвестными характеризуется полиномом 3-й степени. Повышение быстродействия элементов компьютера уменьшает время исполнения алгоритма, но не уменьшает степень полинома сложности.

### *1.5. Способы описания алгоритмов*

Существует несколько способов описания алгоритмов. Наиболее распространенные способы – это словесное и графическое описания алгоритма.

***Словесное описание алгоритма***

В любом алгоритме для обозначения данных используют некоторый набор символов, называемых *буквами*. Конечную совокупность букв называют *алфавитом*, из любой конечной последовательности которого можно составить *слово*, т.е. в любом алфавите реальным данным можно сопоставить некоторые слова, в дальнейшем обозначающие эти данные.

При словесной записи алгоритм описывается с помощью естественного языка с использованием следующих конструкций:

1) шаг (этап) обработки (вычисления) значений данных – «=»;

2) проверка логического условия: если (условие) истинно, то выполнить действие 1, иначе – действие 2;

3) переход (передача управления) к определенному шагу (этапу) $N$.

Для примера рассмотрим алгоритм решения квадратного уравнения вида $a \cdot x^2 + b \cdot x + c = 0$:

1) ввод исходных данных $a$, $b$, $c$ ($a,b,c \neq 0$);

2) вычислить дискриминант $D = b^2 - 4 \cdot a \cdot c$ ;

3) если $D < 0$, то перейти к п. 6, сообщив, что действительных корней нет;

4) иначе, если $D \geq 0$, вычислить $x_1 = (-b + \sqrt{D})/(2 \cdot a)$ и $x_2 = (-b - \sqrt{D})/(2 \cdot a)$;

5) вывести результаты $x_1$ и $x_2$ ;

6) конец.

***Графическое описание алгоритма***

Графическое изображение алгоритма – это представление его в виде схемы, состоящей из последовательности блоков (геометрических фигур), каждый из которых отображает содержание очередного шага алгоритма. А внутри фигур кратко записывают действие, выполняемое в этом блоке. Такую схему называют блок-схемой или структурной схемой алгоритма, или просто схемой алгоритма.

Правила изображения фигур сведены в единую систему программной документации (дата введения последнего стандарта ГОСТ 19.701.90 – 01.01.1992).

По данному ГОСТу графическое изображение алгоритма – это схема данных, которая отображает путь данных при решении задачи и определяет этапы их обработки.

Схема данных состоит из следующих элементов:

– символов данных (символы данных могут отображать вид носителя данных);

– символов процесса, который нужно выполнить над данными;

– символов линий, указывающих потоки данных между процессами и носителями данных;

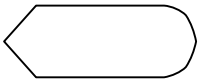
– специальных символов, которые используют для облегчения чтения схемы алгоритма.

Рассмотрим основные символы для изображения схемы алгоритма.

***Символы ввода-вывода данных:***

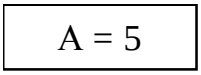
– данные ввода-вывода, если носитель не определен;

– ручной ввод с устройства любого типа, например с клавиатуры;

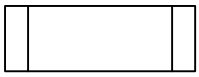


– отображение данных в удобочитаемой форме на устройстве, например дисплее.

***Символы процесса:***




– ***процесс*** – отображение функции обработки данных, т.е. операции, приводящей к изменению указанного значения;



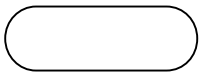
– ***предопределенный процесс*** – отображение группы операций, которые определены в другом месте, например в подпрограмме (функции);


– ***решение*** – отображение функции, имеющей один вход и ряд альтернативных выходов, из которых только один может быть активизирован после анализа условия, указанного внутри этого символа.

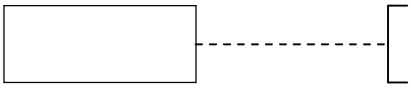
***Символы линий*** – отображают поток данных или управления. Линии – горизонтальные или вертикальные, имеющие только прямой угол перегиба. Стрелки – указатели направления не ставятся, если управление идет сверху вниз или слева направо.

***Специальные символы***

***Соединитель*** – используется при обрыве линии и продолжении ее в другом месте (необходимо присвоить название).



***Терминатор*** – вход из внешней среды или выход во внешнюю среду (начало или конец схемы программы).



***Комментарий.***

### *1.6. Способы реализации алгоритмов*

Любую программу можно разбить на блоки, реализованные в виде алгоритмов (процессов), которые можно разделить на три вида:

1) линейные (единственное направление выполнения);

2) разветвляющиеся (направление выполнения определяет условие);

3) циклические (отдельные участки вычислений выполняются многократно).

Любой циклический процесс включает в себя участок с разветвлением и может быть простым и сложным (вложенным).

Для решения вопроса о том, сколько раз нужно выполнить цикл, используется анализ переменной, которую называют параметром цикла.

Циклический процесс, в котором количество повторений заранее известно, называется циклом по счетчику, а циклический процесс, в котором количество повторений заранее неизвестно и зависит от получаемого в ходе вычислений результата, называют итерационным.

### *1.7. Пример простейшего линейного процесса*

Наиболее часто в практике программирования требуется организовать расчет некоторого арифметического выражения при различных исходных данных. Например, такого:

$$z = \frac{tg^2 x}{\sqrt{x^2 + m^2}} + x^{(m+1)} \sqrt{x^2 + m^2} \ ,$$

где $x > 0$ – вещественное, $m$ – целое.

Разработка алгоритма обычно начинается с составления схемы. Продумывается оптимальная последовательность вычислений, при которой, например, отсутствуют повторения. При написании алгоритма рекомендуется переменным присваивать те же имена, которые фигурируют в заданном арифметическом выражении либо иллюстрируют их смысл.

Для того чтобы не было «длинных» операторов, исходное выражение полезно разбить на ряд более простых. В нашей задаче предлагается схема вычислений, представленная на рис. 1.1.

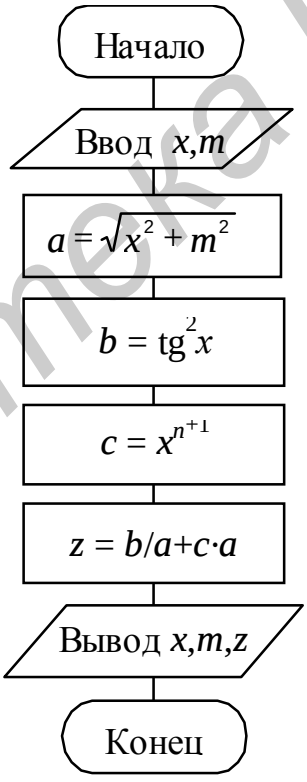



Рис. 1.1. Схема линейного процесса

Она содержит ввод и вывод исходных данных, линейный вычислительный процесс, вывод полученного результата. Заметим, что выражение $\sqrt{x^2 + m^2}$ вычисляется только один раз. Введя дополнительные переменные $a$, $b$, $c$, мы разбили сложное выражение на ряд более простых.

### *1.7. Пример циклического процесса*

Вычислить значение функции $y = \sin x$, представленной в виде разложения в ряд, с заданной точностью, т.е. до тех пор, пока разность между соседними слагаемыми не станет меньше заданной точности:

$$y = \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots.$$

Схема алгоритма, приведенная на рис. 1.2, реализует циклический процесс, в состав которого (в блоке проверки $|E| < eps$) входит участок разветвления.

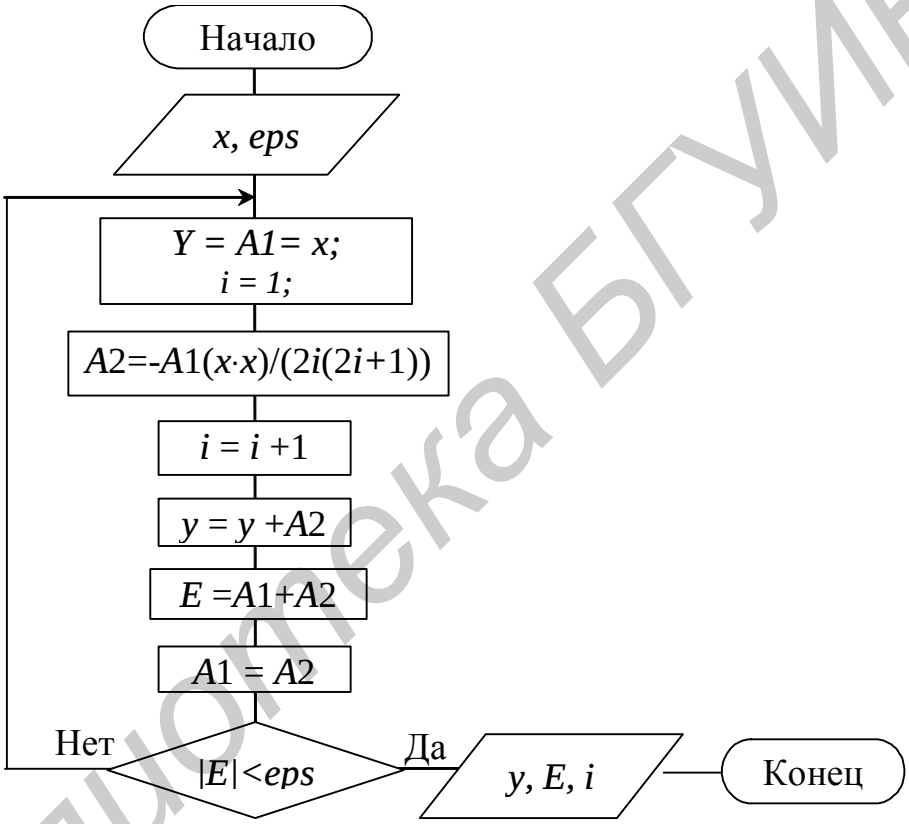



Рис. 1.2. Схема циклического алгоритма

# ГЛАВА 2. Базовые средства языка Си

Любая программа, написанная на языке высокого уровня, состоит из последовательности инструкций, оформленных в строгом соответствии с набором правил, составляющих ***синтаксис данного языка***.

При создании программ разработчик может допустить следующие ошибки: синтаксические и логические.

Синтаксические ошибки – это результат нарушения формальных правил написания программы на конкретном языке программирования.

Логические ошибки разделяются, в свою очередь, на ошибки алгоритма и семантические ошибки.

Причиной ошибки алгоритма является несоответствие построенного алгоритма ходу получения конечного результата сформулированной задачи.

Причина семантической ошибки – неправильное понимание смысла (семантики) операторов выбранного языка программирования.

## *2.1. Алфавит языка Си*

Алфавит любого языка составляет совокупность символов – тех неделимых знаков, при помощи которых записываются все тексты на данном языке.

Каждому из множества значений, определяемых одним байтом (от 0 до 255), в таблице знакогенератора ЭВМ ставится в соответствие символ. По кодировке фирмы *IBM* символы с кодами от 0 до 127, образующие первую половину таблицы знакогенератора, построены по стандарту *ASCII* и одинаковы для всех компьютеров, вторая половина символов (коды 128 – 255) может отличаться и обычно используется для размещения символов национального алфавита. Коды 176 – 223 отводятся под символы псевдографики, а коды 240 – 255 – под специальные знаки (прил. 1).

Алфавит языка Си включает:

– прописные и строчные буквы латинского алфавита и знак подчеркивания (код 95);

– арабские цифры от 0 до 9;

– специальные символы, смысл и правила использования которых будем рассматривать по тексту;

– пробельные (разделительные) символы: пробел, символы табуляции, перевода строки, возврата каретки, новой страницы и новой строки.

## *2.2. Лексемы*

Из символов алфавита формируются *лексемы* (или элементарные конструкции) языка – минимальные значимые единицы текста в программе:

– идентификаторы;

– ключевые (зарезервированные) слова;

– знаки операций;

– константы;

– разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими как разделители или знаки операций, а также комментариями.

### *2.3. Идентификаторы и ключевые слова*

Идентификатор (***ID***) – это имя программного объекта[*] (константы, переменной, метки, типа, функции и т.д.). В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания; первый символ *ID* – не цифра; пробелы внутри *ID* не допускаются.

Длина идентификатора определяется выбранной версией среды программирования. Например, в среде *Borland* C++ 6.0 идентификаторы могут включать любое число символов, из которых воспринимаются и используются только первые 32 символа. Современная тенденция – снятие ограничений длины идентификатора.

При именовании объектов следует придерживаться общепринятых соглашений:

– *ID* переменных и функций обычно пишутся строчными (малыми) буквами – *index, max*();

– *ID* типов пишутся с большой буквы, например, *Spis*, *Stack*;

– *ID* констант (макросов) – большими буквами – *INDEX*, *MAX_INT*;

– идентификатор должен нести смысл, поясняющий назначение объекта в программе, например, *birth_date* – день рождения, *sum* – сумма;

– если *ID* состоит из нескольких слов, как, например, *birth_date*, то принято либо разделять слова символом подчеркивания, либо писать каждое следующее слово с большой буквы – *birthDate*.

В Си прописные и строчные буквы – различные символы. Идентификаторы *Name*, *NAME*, *name* – различные объекты.

Ключевые (зарезервированные) слова не могут быть использованы в качестве идентификаторов.

Список ключевых слов, определенных в стандарте ANSI Си:

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

[*] Здесь и далее по тексту объектами будем называть элементы, участвующие в программе.

### *2.4. Комментарии*

Еще один базовый элемент языка программирования – *комментарий* – не является лексемой. Внутри комментария можно использовать любые допустимые на данном компьютере символы, поскольку компилятор их игнорирует.

В Си комментарии ограничиваются парами символов /* и */, а в С++ был введен вариант комментария, который начинается символами // и заканчивается символом перехода на новую строку.

### *2.5. Простейшая программа*

Программа, написанная на языке Си, состоит из одной или нескольких функций, одна из которых имеет идентификатор ***main***[*] – главная (основная). Она является первой выполняемой функцией (с нее начинается выполнение программы) и ее назначение – управлять работой всей программы (проекта).

Общая структура программы на языке Си имеет вид:

<директивы препроцессора>
<определение типов пользователя – *typedef*>
<описание прототипов функций>
<определение глобальных переменных>
<функции>

В свою очередь, каждая функция имеет следующую структуру:

<класс памяти> <тип> < *ID* функции> (<объявление параметров>)
{ – начало функции
код функции
} – конец функции

Код функции является блоком и поэтому заключается в фигурные скобки.

Функции не могут быть вложенными друг в друга.

Рассмотрим кратко основные части общей структуры программ.

Перед компиляцией программа обрабатывается ***препроцессором*** (прил. 3), который работает под управлением директив.

Препроцессорные ***директивы*** начинаются символом #, за которым следует наименование директивы, указывающее ее действие.

Препроцессор решает ряд задач по предварительной обработке программы, основной из которых является подключение (***include***) к программе так называемых заголовочных файлов (обычных текстов) с декларацией стандартных библиотечных функций, использующихся в программе. Общий формат ее использования

***#include < ID_файла.h>***

где *h* – расширение заголовочных файлов.

[*] Более подробное описание функции *main* рассматривается в п. 11.7.

Если идентификатор файла заключен в угловые скобки (< >), то поиск данного файла производится в стандартном каталоге, если – в двойные кавычки (” ”), то поиск файла производится в текущем каталоге.

К наиболее часто используемым библиотекам относятся:

***stdio.h*** – содержит стандартные функции файлового ввода-вывода;

***math.h*** – математические функции;

***conio.h*** – функции для работы с консолью (клавиатура, дисплей).

Второе основное назначение препроцессора – обработка макроопределений. Макроподстановка ***определить*** (***define***) имеет общий вид

***#define ID строка***

Например: *#define PI* 3.1415927

– в ходе препроцессорной обработки программы идентификатор *PI* везде будет заменяться значением 3.1415927.

Рассмотрим пример, позволяющий понять простейшие приемы программирования на языке Си:

```
#include <stdio.h>
void main(void)
{                          // Начало функции main
    printf(“ Высшая оценка знаний – 10 !”);
}                          // Окончание функции main
```

Отличительным признаком функции служат скобки ( ) после ее идентификатора, в которые заключается список параметров. Перед *ID* функции указывается тип возвращаемого ею результата. Если функция не возвращает результата и не имеет параметров, указывают атрибуты *void* – отсутствие значений.

Для начала будем использовать функцию *main* без параметров и не возвращающую значения.

Код функции представляет собой набор инструкций, каждая из которых оканчивается символом «**;**». В нашем примере одна инструкция – функция *printf*, выполняющая вывод данных на экран, в данном случае – указанную фразу.

---

Приемы отладки в среде программирования *Visual* C++ 6.0 рассматриваются в прил. 5.

---

### *2.6. Основные типы данных*

Данные в языке Си разделяются на две категории: простые (скалярные), будем их называть базовыми, и сложные (составные) типы данных.

Тип данных определяет:

– внутреннее представление данных в оперативной памяти;

– совокупность значений (диапазон), которые могут принимать данные этого типа;

– набор операций, которые допустимы над такими данными.

Основные типы базовых данных: целый – ***int*** (*integer*), вещественный с одинарной точностью – ***float*** и символьный – ***char*** (*character*).

В свою очередь, данные целого типа могут быть короткими – ***short***, длинными – ***long*** и беззнаковыми – ***unsigned***, а вещественные – с удвоенной точностью – ***double***.

Сложные типы данных – массивы, структуры – ***struct***, объединения – ***union***, перечисления – ***enum***.

Данные целого и вещественного типов находятся в определенных диапазонах, т.к. занимают разный объем оперативной памяти (табл. 2.1).

Таблица 2.1

| Тип данных | Объем памяти (байт) | Диапазон значений |
|---|---|---|
| *char* | 1 | –128 … 127 |
| *int* | 2 (4)* | –32768 … 32767 |
| *short* | 1 (2)* | –32768 … 32767(–128 … 127) |
| *long* | 4 | –2147483648 … 2147483647 |
| *unsigned int* | 4 | 0 … 65535 |
| *unsigned long* | 4 | 0 … 4294967295 |
| *float* | 4 | $3{,}14\cdot10^{-38}$ … $3{,}14\cdot10^{38}$ |
| *double* | 8 | $1{,}7\cdot10^{-308}$ … $1{,}7\cdot10^{308}$ |
| *long double* | 10 | $3{,}4\cdot10^{-4932}$ … $3{,}4\cdot10^{4932}$ |

* Размер памяти зависит от разрядности процессора, для 16-разрядных объем памяти определяется первой цифрой, для 32-разрядных – второй.

### *2.7. Декларация объектов*

Все объекты, с которыми работает программа, необходимо декларировать, т.е. объявлять компилятору об их присутствии. При этом возможны две формы декларации:

– описание, не приводящее к выделению памяти;

– определение, при котором под объект выделяется объем памяти в соответствии с его типом; в этом случае объект можно инициализировать, т.е. задать его начальное значение.

Кроме констант, заданных в исходном тексте, все объекты программы должны быть явно декларированы по следующему формату:

***<атрибуты> <список ID объектов>;***

элементы *списка ID объектов* разделяются запятыми, а *атрибуты* – разделителями, например: *int i, j, k*; *float a, b*;

Объекты программы могут иметь следующие атрибуты:

***класс памяти*** – характеристика способа размещения объектов в памяти (статическая, динамическая); определяет область видимости и время

жизни переменной (по умолчанию – ***auto***), данные атрибуты будут рассмотрены в гл. 12;

***тип*** – тип будущих значений декларируемых объектов (по умолчанию устанавливается тип ***int***).

Класс памяти и тип – атрибуты необязательные и при отсутствии одного из них (но не обоих одновременно) устанавливаются атрибуты по умолчанию.

Примеры декларации простых объектов:

*int i, j, k;* *char r;* *double gfd;*

Рассмотрим основные базовые типы данных более подробно.

### *2.8. Данные целого типа (integer)*

Тип ***int*** – целое число, обычно соответствующее естественному размеру целых чисел. Квалификаторы ***short*** и ***long*** указывают на различные размеры и определяют объем памяти, выделяемый под них (см. табл. 2.1), например:

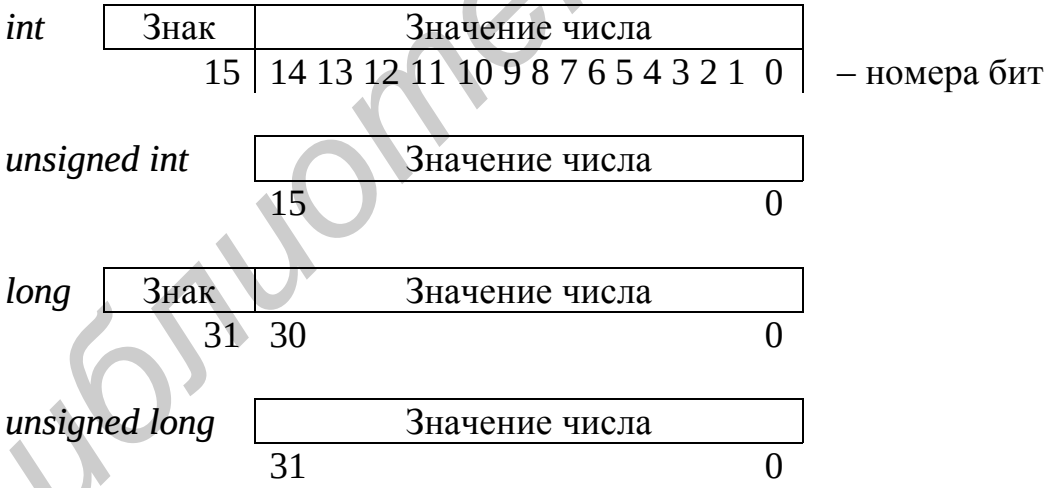
*short x*;

*long x*;

*unsigned x* = 8; – декларация с инициализацией числом 8;

атрибут *int* в этих случаях может быть опущен.

Атрибуты ***signed*** и ***unsigned*** показывают, как интерпретируется старший бит числа – как знак или как часть числа:




### *2.9. Данные символьного типа (char)*

Под величину символьного типа отводится такое количество байт, которое достаточно для любого символа. Поэтому символьная переменная занимает в памяти один байт. Закрепление конкретных символов за кодами производится кодовыми таблицами.

Для персональных компьютеров (ПК) наиболее распространена *ASCII* (*American Standard Code for Information Interchenge*) таблица кодов (см. прил. 1). Данные типа *char* рассматриваются компилятором как целые, поэтому

возможно использование *signed char*: величины со знаком (по умолчанию) – символы с кодами от –128 до +127 и *unsigned char* – беззнаковые символы с кодами от 0 до 255. Этого достаточно для хранения любого символа из 256-символьного набора *ASCII*. Величины типа *char* применяют еще и для хранения целых чисел из указанных диапазонов.

Примеры: *char res, simv*1, *simv*2;

*char let* = *'s'*; – декларация символьной переменной с инициализацией символом *s*.

### *2.10. Данные вещественного типа (float, double)*

Данные вещественного типа в памяти занимают (табл. 2.2): ***float*** – 4 байта (одинарная точность), ***double*** (удвоенная точность) – 8 байт; ***long double*** (повышенная точность) – 10 байт. Для размещения данных типа *float* обычно 8 бит выделено для представления порядка и знака и 24 бита под мантиссу.

Таблица 2.2

| Тип | Точность (мантисса) | Порядок |
|---|---|---|
| ***float*** (4 байта) | 7 цифр после запятой | ± 38 |
| ***double*** (8 байт) | 15 | ± 308 |
| ***long double*** (10 байт) | 19 | ± 4932 |

Типы данных с плавающей десятичной точкой хранятся в оперативной памяти иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей: мантиссы и порядка (см. разд. 3.2 «Константы вещественного типа»). В *IBM* совместимых ПК, как вы уже знаете, переменная типа *float* занимает 4 байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Мантисса – это число больше единицы и меньше двух. Поскольку старшая цифра мантиссы всегда равна единице, то ее не хранят.

Для величин типа *double*, занимающих 8 байт, под порядок и мантиссу отводится 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а порядок – его диапазон. Как можно видеть из приведенных выше таблиц, при одинаковом количестве байт, отводимом под величины типа *float* и *long int*, диапазоны их допустимых значений сильно различаются из-за внутренней формы представления значений таких данных.

При переносе программы с одной платформы на другую нельзя делать предположений, например, о типе *int*, так как для оперативной системы (ОС) *MS DOS* этот тип имеет размер в два байта, а для ОС *Windows* 9X – четыре байта. В стандарте *ANSI* поэтому диапазоны значений для основных типов не задаются, а определяются только соотношения между их размерами, например:

*sizeof* (***float***) < *sizeof* (***double***) < *sizeof* (***long double***) ,

*sizeof* (***char***) < *sizeof* (***short***) < *sizeof* (***int***) < *sizeof* (***long***) ,

где операция ***sizeof*** – возвращает количество байт для указанного аргумента – скалярного типа данных.

### *2.11. Использование модификаторов при декларации производных типов данных*

Ключевые слова *int, float, char* и т.д. называют конечными атрибутами декларации объектов программы. При декларации так называемых производных объектов используют еще дополнительные – промежуточные атрибуты или, как их иногда называют, «***модификаторы***».

К символам модификации текущего типа относятся:

– символ ***** перед идентификатором, обозначающий декларацию указателя на объект исходного типа (левый промежуточный атрибут);

– символы [ ] после идентификатора объекта – декларация массива объектов;

– символы ( ) после идентификатора объекта – декларация функции (правые промежуточные атрибуты).

Допускается использование более одного модификатора типа с учетом следующих правил:

1) чем ближе модификатор к *ID* объекта, тем выше его приоритет;

2) при одинаковом расстоянии от идентификатора объекта модификаторы [ ] и ( ) обладают приоритетом перед атрибутом звездочка *;

3) дополнительные круглые скобки позволяют изменить приоритет объединяемых ими элементов описания;

4) квадратные и круглые скобки, имеющие одинаковый приоритет, рассматриваются слева направо.

Конечный атрибут декларации принимается во внимание в последнюю очередь, т.е. тогда, когда все промежуточные атрибуты уже проинтерпретированы.

Примеры декларации объектов с конечным атрибутом ***int***:

*int a*; – переменная типа *int*;
*int a*[5]; – массив из пяти элементов типа *int*;
*int *a*; – указатель на объект типа *int*;
*int **a*; – указатель на указатель на объект типа *int*;
*int *a*[5]; – массив из пяти указателей на элементы типа *int*;
*int* (**a*)[10]; – указатель на массив из десяти элементов типа *int*;
*int *a*[3][4]; – 3-элементный массив указателей на одномерные целочисленные массивы по четыре элемента каждый;
*int a*[5][2]; – двухмерный массив элементов типа *int*;
*int a*(*void*); – функция без параметров, возвращающая значение типа *int*;
*int *a*(*void*); – функция без параметров, возвращающая указатель на элемент типа *int*;
*int* (**a*)(*void*); – указатель на функцию без параметров, возвращающую значение типа *int*;

*int *a*(*void*)[6]; – функция без параметров, возвращающая указатель на массив элементов типа *int*;

*int *a* [4](*void*); – массив указателей на функцию без параметров, возвращающую значение типа *int*.

Существуют и недопустимые последовательности промежуточных атрибутов, например, массив не может состоять из функций, а функция не может возвращать массив или другую функцию.

# ГЛАВА 3. Константы в программах

Константами называют величины, которые не изменяют своего значения во время выполнения программы, т.е. это объекты, не подлежащие использованию в левой части операции присваивания, т.к. константа – это неадресуемая величина и, хотя она хранится в памяти компьютера, не существует способа определить ее адрес. В языке Си константами являются:

– самоопределенные арифметические константы целого и вещественного типов, символьные и строковые данные;

– идентификаторы массивов и функций;

– элементы перечислений.

## *3.1. Целочисленные константы*

Общий формат записи: ±***n*** (+ обычно не ставится).

***Десятичные константы*** – это последовательность цифр 0...9, первая из которых *не должна быть 0*. Например, 22 и 273 – обычные целые константы, если нужно ввести длинную целую константу, то указывается признак *L*(*l*) – 273*L* (273*l*). Для такой константы будет отведено – 4 байта. Обычная целая константа, которая слишком длинна для типа *int*, рассматривается как *long*.

Существует система обозначений для восьмеричных и шестнадцатеричных констант.

***Восьмеричные константы*** – это последовательность цифр от 0 до 7, первая из которых *должна быть 0*, например: $020_8 = 16_{10}$.

***Шестнадцатеричные константы*** – последовательность цифр от 0 до 9 и букв от *A* до *F* (*a...f*), начинающаяся символами 0*X* (0*x*), например: $0X1F_{16}$ $(0x1f)_{16} = 31_{10}$.

Восьмеричные и шестнадцатеричные константы могут также заканчиваться буквой *L*(*l*) – *long*, например, 020*L* или 0*X*20*L*.

Примеры целочисленных констант:

| | | | |
|---|---|---|---|
| 1992 | 777 | 1000*L* | – десятичные; |
| 0777 | 00033 | 01*l* | – восьмеричные; |
| 0*x*123 | 0*X*00*ff* | 0*xb*8000*l* | – шестнадцатеричные. |

### *3.2. Константы вещественного типа*

Данные константы размещаются в памяти в формате *double*, а во внешнем представлении могут иметь две формы:

1) с фиксированной десятичной точкой, формат записи: ±***n.m***, где *n*, *m* – целая и дробная части числа;

2) с плавающей десятичной точкой (экспоненциальная форма) представляется в виде мантиссы и порядка. Мантисса записывается слева от знака экспоненты (*E* или *e*), а порядок – справа. Значение константы определяется как произведения мантиссы и числа 10, возведенного в указанную в порядке степень.

Общий формат таких констант: ±***n.mE*±*p***, где *n*, *m* – целая и дробная части числа, *p* – порядок; ±0.*xxxE*±*p* – нормализованный вид, например, $1{,}25 \cdot 10^{-8} = 0.125E{-}7$.

Примеры констант с фиксированной и плавающей точками:

1.0 –3.125 100*e*–10 0.12537*e*+12.

Пробелы внутри чисел не допускаются, а для отделения целой части числа от дробной используется *точка*. Можно опустить нулевую дробную или целую части числа, но не обе сразу, например, $1.0 \leftrightarrow 1.$ или $0.5 \leftrightarrow .5$ .

В любом случае при использовании вещественных констант наличие так называемой десятичной точки обязательно.

### *3.3. Символьные константы*

*Символьная константа* – это символ, заключенный в одинарные кавычки: '*A*', '*x*' (тип *char* занимает в памяти один байт).

Также используются специальные последовательности символов – управляющие (*escape*) последовательности:

\n – новая строка;
\t – горизонтальная табуляция;
\b – шаг назад;
\r – возврат каретки;
\v – вертикальная табуляция;
\f – перевод формата (переход на новую строку);
\\ – обратный слеш;
\' – апостроф;
\" – кавычки;
\0 – символ «пусто», не путать с символом ‘0’.

***Символьная константа '\0'*** – это нулевой байт, каждый бит которого равен нулю.

При присваивании символьным переменным значений констант значения констант заключаются в апострофы, например:

char ss = ‘У’;

Текстовые символы непосредственно вводятся с клавиатуры, а специальные и управляющие – представляются в исходном тексте парами символов, например: \\ , \' , \" .

Примеры символьных констант: '*A*', '9', '$', '*n*'.

### *3.4. Строковые константы*

*Строковая константа* представляет собой последовательность символов кода *ASCII*, заключенную в кавычки (”). Во внутреннем представлении к строковым константам добавляется пустой символ '\0', который не является цифрой 0, на печать не выводится (в таблице кодов *ASCII* имеет код = 0) и является признаком окончания строки.

Кавычки не являются частью строки, а служат только для ее ограничения. Строка в языке Си представляет собой массив, состоящий из символов. Внутреннее представление константы "1234*ABC*": '1' '2' '3' '4' '*A*' '*B*' '*C*' '\0' .

Примеры строковых констант:

"Система", "*n**t* Аргумент *n*", "Состояние \"*WAIT* \" " .

Строковые константы еще называют *строковыми литералами*.

В конец строковой константы компилятор автоматически помещает нуль-символ.

Длинную строковую константу можно разбить на несколько, используя символ переноса – обратный слеш (\). Например:

“*Вы поступили и* \
*учитесь на факультете информационных технологий* \
*Белорусского государственного университета* \
*информатики и радиоэлектроники*”

Компилятор Си воспримет такую запись как единое целое, игнорируя символы обратного слеша.

## ГЛАВА 4. Обзор операций

### *4.1. Операции, выражения*

Выражения используются для вычисления значений (определенного типа) и состоят из операндов, операций и скобок. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев – константой или переменной. Операнды задают данные для вычислений.

Знак операции – это один или более символов, определяющих действие над операндами, т.е. операции задают действия, которые необходимо выполнить. Внутри знака операции пробелы не допускаются.

Операции делятся на унарные, бинарные и тернарные – по количеству участвующих в них операндов, и выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки.

Большинство операций выполняются слева направо, например, $a+b+c \rightarrow$ $\rightarrow (a+b)+c$. Исключение составляют унарные операции, операции присваивания и условная операция (?:), которые выполняются справа налево.

В языке Си используются четыре унарные операции, имеющие самый высокий приоритет, их часто называют первичными:

– операция доступа к полям структур и объединений при помощи идентификаторов «**.**» – точка;

– операция доступа к полям структур и объединений при помощи указателей «**–>**» – стрелка;

– операция [ ] индексации, используемая при декларации массива и обращении к его элементам;

– операция ( ) обращения к функции.

Первичные операции будут рассмотрены в соответствующих разделах.

Полный список операций с указанием их приоритетов приводится в прил. 2.

### *4.2. Арифметические операции*

Обозначения арифметических операций:
+ (сложение); – (вычитание); / (деление, для *int* операндов – с отбрасыванием остатка); * (умножение); % (остаток от деления целочисленных операндов со знаком первого операнда – деление «по модулю»).

Операндами традиционных арифметических операций (+ – * /) могут быть константы, переменные, обращения к возвращающим значения функциям, элементы массивов, любые арифметические выражения, указатели (с ограничениями).

Порядок выполнения действий в арифметических выражениях следующий: выражения в круглых скобках; операции *, /, %; операции **+, –.**

Унарные операции «знак числа» (+, –) обладают самым высоким приоритетом и определены для операндов числовых типов (имеющих числовой результат), при этом «+» носит только информационный характер, «–» меняет знак операнда на противоположный (неадресная операция).

Операции *, /, % обладают высшим приоритетом над операциями **+, –**, поэтому при записи сложных выражений нужно использовать общепринятые математические правила: $x + y \cdot z - \dfrac{a}{b+c} \leftrightarrow x+y*z-a\ /(b+c)$ , т.е. использовать круглые скобки.

### *4.3. Операция присваивания*

Формат операции присваивания:

***Операнд*_1 = *Операнд*_2 ;**

*Операндом*_1 (левый операнд) может быть только переменная. Левый операнд операции присваивания получил название ***L–значение***, (*L–value*, *Left–value*) – *адресное выражение*. Так в Си называют любое выражение, адресующее некоторый участок оперативной памяти, в который можно записать некоторое значение. Переменная – это частный случай адресного выражения.

*Операндом*_2 (правый операнд) могут быть: константа, переменная или любое выражение, составленное в соответствии с синтаксисом языка Си. Правый операнд операции присваивания назвали ***R–значение***, (*R–value*, *Right–value*).

Присваивание значения в языке Си, в отличие от традиционной интерпретации, рассматривается как выражение, имеющее значение левого операнда после присваивания. Таким образом, присваивание может включать несколько операций присваивания, изменяя значения нескольких операндов, например:

```
int i, j, k;
float x, y, z;
   ...
i = j = k = 0;          ↔   k = 0, j = k, i = j;
x = i + (y = 3) – (z = 0);  ↔   z = 0, y = 3, x = i + y – z;
```

***Примеры недопустимых выражений:***

– присваивание константе: $2 = x + y$;
– присваивание функции: *getch*() = *i*;
– присваивание результату операции: $(i + 1) = 2 + y$;

### *4.4. Сокращенная запись операции присваивания*

В языке Си используются два вида сокращенной записи операции присваивания:

1) вместо записи: ***v = v # e***;

где **#** – любая арифметическая операция (операция над битовым представлением операндов), рекомендуется использовать запись ***v #= e***;

Например, *i* = *i* + 2; ↔ *i* += 2; (***знаки операций – без пробелов***);

2) вместо записи: *x* = *x* # 1;

где **#** – символы, обозначающие операцию инкремента (+1), либо декремента (–1), *x* – целочисленная переменная (или переменная-указатель), рекомендуется использовать запись:

***##x;*** – префиксную, или ***x##;*** – постфиксную.

Если эти операции используются в чистом виде, то различий между постфиксной и префиксной формами нет. Если же они используются в выражении, то в префиксной форме (##*x*) сначала значение *x* изменится на 1, а затем полученный результат будет использован в выражении; в постфиксной форме (*x*##) – сначала значение переменной *x* используется в выражении, а затем изменится на 1. Операции над указателями будут рассмотрены в разд. 9.4.

***Пример 1:***

| *int i, j, k;* | Смысл записи |
|---|---|
| *float x, y;* | |
| ... | |
| *x* *= *y*; | *x* = x*y; |
| *i* += 2; | *i* = *i* + 2; |
| *x* /= *y*+15; | *x* = *x*/(*y* + 15); |
| *--k*; | *k* = *k* – 1; |
| *k*--; | *k* = *k* – 1; |
| *j* = *i*++; | *j* = *i*; *i* = *i* + 1; |
| *j* = ++*i*; | *i* = *i* + 1; *j* = i; |

***Пример 2:***

| *int n, a, b, c, d;* | Значения |
|---|---|
| *n* = 2; *a* = *b* = *c* = 0; | |
| *a* = ++*n*; | *n*=3, *a*=3 |
| *a* += 2; | *a*=5 |
| *b* = *n*++; | *b*=3, *n*=4 |
| *b* –= 2; | *b*=1 |
| *c* = --*n*; | *n*=3, *c*=3 |
| *c* *= 2; | *c*=6 |
| *d* = *n*--; | *d*=3, *n*=2 |
| *d* %= 2; | *d*=1 |

### *4.5. Преобразование типов операндов арифметических операций*

Если операнды арифметических операндов имеют один тип, то и результат операции будет иметь такой же тип.

Но, как правило, в операциях участвуют операнды различных типов. В этом случае они преобразуются к общему типу в порядке увеличения их «размера памяти», т.е. объема памяти, необходимого для хранения их значений. Поэтому неявные преобразования всегда идут от «меньших» объектов к «большим». Схема выполнения преобразований операндов арифметических операций выглядит следующим образом:

***short, char*** ® ***int*** ® ***unsigned*** ® ***long*** ® ***double***
***float*** ® ***double***

Стрелки отмечают преобразования даже однотипных операндов перед выполнением операции. То есть действуют следующие правила:

– значения типов *char* и *short* всегда преобразуются в *int*;

– если один из операндов имеет тип *double*, то и другой преобразуется в *double*;

– если один из операндов *long*, то другой преобразуется в *long*.

***Внимание***. Результатом операции **1/3** будет значение **0**, чтобы избежать такого рода ошибок, необходимо явно изменить тип хотя бы одного операнда, т.е. записать, например: 1./3 .

Типы *char* и *int* могут свободно смешиваться в арифметических выражениях. Каждая переменная типа *char* автоматически преобразуется в

*int*, что обеспечивает значительную гибкость при проведении преобразований, т.к. над типом *int* действия выполняются быстрее, чем над любым другим типом.

При выполнении операции присваивания значение правого операнда преобразуется к типу левого, который и является типом полученного результата. И здесь необходимо быть внимательным, т.к. при некорректном использовании операций присваивания могут возникнуть неконтролируемые ошибки. Так, при преобразовании *int* в *char* старший байт просто отбрасывается.

Пусть: *float x*; *int i*; тогда и $x = i$; и $i = x$; приводят к преобразованиям, причем *float* преобразуется в *int* отбрасыванием дробной части.

Тип *double* преобразуется в *float* округлением.

Длинное целое преобразуется в более короткое целое и *char* посредством отбрасывания бит в старших разрядах.

Итак, безопасным преобразованием типов является преобразование в порядке увеличения «размера памяти», обратное преобразование может привести к потере значащих разрядов.

### *4.6. Операция приведения типа*

В любом выражении преобразование типов может быть осуществлено явно, для этого достаточно перед выражением поставить в круглых скобках атрибут соответствующего типа:

**(*тип*) *выражение***;

ее результат – значение *выражения*, преобразованное к заданному *типу*.

Операция приведения типа вынуждает компилятор выполнить указанное преобразование, но ответственность за последствия возлагается на программиста. Использовать эту операцию рекомендуется везде, где это необходимо, например:

*double x*;
*int n* = 6, *k* = 4;
$x = (n + k)/3;$ $\rightarrow$ $x = 3$, т.к. дробная часть будет отброшена;
$x = (double)(n + k)/3;$ $\rightarrow$ $x = 3.333333$ – использование операции приведения типа позволило избежать округления результата деления целочисленных операндов.

### *4.7. Операции сравнения*

В языке Си используются следующие операции сравнения, т.е. отношения между объектами:

| | | | |
|---|---|---|---|
| == | – равно или эквивалентно; | != | – не равно; |
| < | – меньше; | <= | – меньше либо равно; |
| > | – больше; | >= | – больше либо равно. |

Пары символов соответствующих операций разделять нельзя.

Общий вид операций отношений:

***Операнд_*1 *Знак операции Операнд_*2**

Указанные операции выполняют сравнение значений первого операнда со вторым. Операндами могут быть любые арифметические выражения и указатели.

Значения арифметических выражений перед сравнением вычисляются и преобразуются к одному типу.

Арифметические операнды преобразуются по правилам, аналогичным для арифметических операций. Операнды-указатели преобразуются в целые числа необходимого типа. Результат сравнения указателей будет корректным в арифметическом смысле лишь для объектов одного массива.

В языке Си нет логического типа данных. Результат операции отношения имеет значение **1**, если отношение истинно, или в результате вычислений получено не нулевое значение, воспринимаемое компилятором Си как истина *(true)*, или **0** – в противном случае, т.е. – ложно *(false)*. Следовательно, операция отношения может использоваться в любых арифметических выражениях.

Операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции отношений.

Примеры использования операций отношений:

$y > 0$ , $x == y$ , $x != 2$ .

Отношения между объектами сложных типов проверяются либо посредством последовательного сравнения их элементов (для массивов), либо используя стандартные библиотечные функции, которые будут рассмотрены позже.

### *4.8. Логические операции*

Приведем логические операции в порядке убывания относительного приоритета. Их обозначения:

**!** – отрицание (логическое «НЕТ»);

**&&** – конъюнкция (логическое «И»);

**||** – дизъюнкция (логическое «ИЛИ»).

Операндами (выражениями) логических операций могут быть любые скалярные типы. Ненулевое значение операнда трактуется как «истина», а нулевое – «ложь». Результатом логической операции, как и в случае операций отношения, может быть 1 или 0.

Общий вид операции ***отрицания***

**!** ***выражение***

Примеры использования операции отрицания:

!0 → 1

!5 → 0

$x = 10$;

! ($x > 0$) $\rightarrow 0$

Общий вид операций ***конъюнкции*** и ***дизъюнкции***:

*Выражение*_1 ***знак операции*** *Выражение*_2

Особенность операций конъюнкции и дизъюнкции – экономное последовательное вычисление выражений-операндов:

– если выражение_1 операции «конъюнкция» ложно, то результат операции – ноль и выражение_2 не вычисляется;

– если выражение_1 операции «дизъюнкция» истинно, то результат операции – единица и выражение_2 не вычисляется.

Например:

$y > 0$ && $x = 7$ $\rightarrow$ истина, если оба выражения истинны;

$e > 0$ || $x = 7$ $\rightarrow$ истина, если хотя бы одно выражение истинно.

Старшинство операции «И» выше, чем «ИЛИ» и обе они младше операций отношения и равенства.

Относительный приоритет логических операций позволяет пользоваться общепринятым математическим стилем записи сложных логических выражений, например:

$0 < x < 100$ $\leftrightarrow$ $0 < x$ && $x < 100$ ;

$x > 0, y \le 1$ $\leftrightarrow$ $x > 0$ && $y <= 1$ .

Учет этих свойств очень существенен для написания правильно работающих программ.

### *4.9. Побитовые логические операции, операции над битами*

В языке Си предусмотрен набор операций для работы с отдельными битами. Эти операции нельзя применять к переменным вещественного типа.

Обозначения операций над битами:

**~** – дополнение (унарная операция); инвертирование (одноместная операция);

**&** – побитовое «И» – конъюнкция;

**|** – побитовое включающее «ИЛИ» – дизъюнкция;

**^** – побитовое исключающее «ИЛИ» – сложение по модулю 2;

**>>** – сдвиг вправо;

**<<** – сдвиг влево.

Общий вид операции инвертирования (поразрядное отрицание):

***~ выражение***

инвертирует каждый разряд в двоичном представлении своего операнда.

Остальные операции над битами имеют вид:

*Выражение*_1 ***знак операции*** *Выражение*_2

Операндами операций над битами могут быть только *выражения*, приводимые к целому типу. Операции (~, &, |, ^) выполняются поразрядно над всеми битами операндов (знаковый разряд особо не выделяется):

~0*x*F0 ↔ *x*0F
0*x*FF & 0*x*0F ↔ *x*0F
0*x*F0 | 0*x*11 ↔ *x*F1
0*x*F4 ^ 0*x*F5 ↔ *x*01

Операция & часто используется для маскирования некоторого множества бит. Например, оператор *w* = *n* & 0177 передает в *w* семь младших бит *n*, полагая остальные равными нулю.

Операции сдвига выполняются также для всех разрядов с потерей выходящих за границы бит.

Операция (|) используется для включения бит *w* = *x* | *y*, устанавливает в единицу те биты в *x*, которые равны 1 в *y*.

Необходимо отличать побитовые операции & и | от логических операций && и || , если *x* = 1, *y* = 2, то *x* & *y* равно нулю, а *x* && *y* равно 1.

0*x*81 << 1 ↔ 0*x*02
0*x*81 >> 1 ↔ 0*x*40

Если *выражение*_1 имеет тип ***unsigned***, то при сдвиге вправо освобождающиеся разряды гарантированно заполняются нулями (логический сдвиг). Выражения типа ***signed*** могут, но необязательно, сдвигаться вправо с копированием знакового разряда (арифметический сдвиг). При сдвиге влево освобождающиеся разряды всегда заполняются нулями. Если *выражение*_2 отрицательно либо больше длины *выражения*_1 в битах, то результат операции сдвига не определен.

Унарная операция (~) дает дополнение к целому, т.е. каждый бит со значением 1 получает значение 0 и наоборот.

Операции сдвига << и >> применяются к целочисленным операндам и осуществляют соответственно сдвиг вправо (влево) своего левого операнда на число позиций, задаваемых правым операндом, например, *x* << 2 сдвигает *x* влево на две позиции, заполняя освобождающиеся биты нулями (эквивалентно умножению на 4).

Операции сдвига вправо на *k* разрядов весьма эффективны для деления, а сдвиг влево – для умножения целых чисел на 2 в степени *k*:

*x* << 1 ↔ *x**2; *x* >> 1 ↔ *x*/2 ;
*x* << 3 ↔ *x**8 .

Подобное применение операций сдвига безопасно для беззнаковых и положительных значений *выражения*_1.

Операции сдвига не учитывают переполнение и потерю значимости.

В математическом смысле операнды логических операций над битами можно рассматривать как отображение некоторых множеств с размерностью не более разрядности операнда на значения {0,1}.

Пусть единица означает обладание элемента множества некоторым свойством, тогда очевидна теоретико-множественная интерпретация рассматриваемых операций:

**~** – дополнение; **|** – объединение; **&** – пересечение.

Простейшее применение – проверка нечетности целого числа:

```
int i;
...
if ( i & 1) printf (" Значение i четно!");
```

Комбинирование операций над битами с арифметическими операциями часто позволяет упростить выражения.

### *4.10. Операция «,» (запятая)*

Данная операция используется при организации строго гарантированной последовательности вычисления выражений (обычно используется там, где по синтаксису допустима только одна операция, а необходимо разместить две и более, например, в операторе *for*). Форма записи:

***выражение_*1, …, *выражение_N*;**

*выражения* 1, 2,…, *N* вычисляются последовательно друг за другом и результатом операции становится значение последнего выражения *N*, например:

$$m = ( i = 1, j = i ++, k = 6, n = i + j + k );$$

получим последовательность вычислений: $i = 1, j = i = 1, i = 2, k = 6, n = 2+1+6,$ и в результате $m = n = 9$.

В заключение отметим следующую особенность языка Си – любые операции допускаются только со скалярными объектами, причем небольшого размера, порядка размера регистров процессора. Это объясняется ориентацией языка на задачи системного программирования. Любые действия с составными или сложными объектами – массивами, строками, структурами и т.п. реализуются с помощью стандартных библиотечных функций, работа с которыми будет рассмотрена позже.

## ГЛАВА 5. Обзор базовых инструкций языка Си

### *5.1. Стандартная библиотека языка Си*

В любой программе кроме операторов и операций используются средства библиотек, входящих в среду программирования. Часть библиотек стандартизована и поставляется с компилятором. Функции, входящие в библиотеку языка Си, намного облегчают создание программ. Расширение библиотечных файлов **.lib*.

В стандартную библиотеку входят также прототипы функций, макросы, глобальные константы. Это, как вы уже знаете, заголовочные файлы с расширением **.h*, которые хранятся в папке *include* и подключаются на этапе предпроцессорной обработки исходного текста программ.

Рассмотрим наиболее часто используемые функции из стандартной библиотеки языка Си.

### *5.2. Стандартные математические функции*

Математические функции языка Си декларированы в файлах ***math.h*** и ***stdlib.h.***

В приведенных здесь функциях аргументы и возвращаемый результат имеют тип *double*. Аргументы тригонометрических функций должны быть заданы в радианах ($2\pi$ радиан = $360^o$).

| Математическая функция | *ID* функции в языке Си |
|---|---|
| $\sqrt{x}$ | sqrt($x$) |
| $\|x\|$ | fabs($x$) |
| $e^x$ | exp($x$) |
| $x^y$ | pow($x$,$y$) |
| ln($x$) | log($x$) |
| $lg_{10}(x)$ | log10($x$) |
| sin($x$) | sin($x$) |
| cos($x$) | cos($x$) |
| tg($x$) | tan($x$) |
| arcsin($x$) | asin($x$) |
| arccos($x$) | acos($x$) |
| arctg($x$) | atan($x$) |
| arctg($x$ / $y$) | atan2($x$) |
| sh($x$)=0.5 ($e^x$–$e^{-x}$) | sinh($x$) |
| ch($x$)=0.5 ($e^x$+$e^{-x}$) | cosh($x$) |
| tgh($x$) | tanh($x$) |
| остаток от деления $x$ на $y$ | fmod($x$,$y$) |
| наименьшее целое >=$x$ | ceil($x$) |
| наибольшее целое <=$x$ | floor($x$) |

### *5.3. Функции вывода данных на дисплей*

В языке Си нет встроенных средств ввода/вывода данных. Ввод/вывод информации осуществляется с помощью библиотечных функций и объектов.

Декларации функций ввода/вывода, как уже упоминалось, приведены в заголовочном файле ***stdio.h.***

Для вывода информации на экран монитора (дисплей) в языке Си чаще всего используются функции: ***printf***() и ***puts***().

Формат функции форматного вывода на экран:

***printf(*** *управляющая строка , список объектов вывода);*

В ***управляющей строке***, заключенной в кавычки, записывают: поясняющий текст, который выводится на экран без изменения (комментарии), список модификаторов форматов, указывающих компилятору способ вывода объектов (признак модификатора формата – символ **%**) и специальные символы, управляющие выводом (признак – символ \).

В ***списке объектов вывода*** указываются идентификаторы печатаемых объектов, разделенных запятыми: переменные, константы или выражения, вычисляемые перед выводом.

Количество и порядок следования форматов должен совпадать с количеством и порядком следования выводимых на экран объектов.

Функция *printf* выполняет вывод данных в соответствии с указанными форматами, поэтому формат может использоваться и для преобразования типов выводимых объектов.

Если признака модификации (%) нет, то вся информация выводится как комментарии.

Основные модификаторы формата:

**%*d* (%*i*)** – десятичное целое число;
**%*c*** – один символ;
**%*s*** – строка символов;
**%*f*** – число с плавающей точкой, десятичная запись;
**%*e*** – число с плавающей точкой, экспоненциальная запись;
**%*g*** – используется вместо ***f***, ***e*** для исключения незначащих нулей;
**%*o*** – восьмеричное число без знака;
**%*x*** – шестнадцатеричное число без знака.

Для чисел ***long*** добавляется символ ***l***, например, %*ld* – длинное целое, %*lf* – число вещественное с удвоенной точностью – ***double***.

Если нужно напечатать сам символ %, то его нужно указать 2 раза:

***printf ("Только %d%% предприятий не работало. \n",5);***

Получим: ***Только 5% предприятий не работало***.

Управляют выводом специальные последовательности символов: ***n*** – новая строка; ***t*** – горизонтальная табуляция; ***b*** – шаг назад; ***r*** – возврат каретки; ***v*** – вертикальная табуляция; \\ – обратная косая; \' – апостроф; \" – кавычки; **\0** – нулевой символ (пусто).

Пример:

```
#define  PI   3.14159
                . . .
        int number = 5;
        float bat = 255;
        int cost = 11000;
                . . .
        printf(" %d студентов съели %f бутербродов. \n", number, but);
```

```
printf(" Значение числа pi равно %f. \n", pi);
printf(" Стоимость этой вещи %d %s. \n", cost, "Руб.");
. . .
```

В модификаторах формата функции ***printf*** после символа % можно указывать число, задающее минимальную ширину поля вывода, например, %5*d* – для целых, %4.2*f* – для вещественных – две цифры после запятой для поля шириной 4 символа. Если указанных позиций для вывода целой части числа не хватает, то происходит автоматическое расширение.

Если после «%» указан знак «минус», то выводимое значение будет печататься с левой позиции поля вывода, заданной ширины, например: % – 10d.

Использование функции ***printf*** для преобразования данных:

1) printf("%d", 336.65); получим: 336;
2) printf("%o", 336); получим: 520, т.е. 5*8**2+2*8+0*1 = 336;
3) printf("%x", 336); получим: 150 (шестнадцатеричное).

Можно использовать функцию ***printf*** для нахождения кода ASCII некоторого символа:

```
printf (" %c – %d\n", 'a', 'a');
```

получим десятичный код ASCII символа a: a – 65 .

Функция ***puts***(*ID* строки); выводит на экран дисплея строку символов, автоматически добавляя к ней символ перехода на начало новой строки (\n).

Аналогом такой функции будет: printf(“%s \n”, *ID* строки);

Функция ***putchar***() выдает на экран дисплея один символ без добавления символа ‘\n’.

### *5.4. Функции ввода информации*

Функция, предназначенная для форматированного ввода исходной информации с клавиатуры:

***scanf*** (*управляющая строка , список адресов объектов ввода*);

в *управляющей строке* указываются только модификаторы форматов, количество и порядок следования которых должны совпадать с количеством и порядком следования вводимых объектов, а тип данных будет преобразовываться в соответствии с модификаторами.

Список объектов ввода представляет собой адреса переменных, разделенные запятыми, т.е. для ввода значения переменной перед ее идентификатором указывается символ &, обозначающий операцию «***взять адрес***».

Если нужно ввести значение строковой переменной, то использовать символ & не нужно, т.к. строка – это массив символов, а *ID* массива является адресом его первого элемента. Например:

```
int course;
```

```
double grant;
char name[20];
printf (" Укажите курс, стипендию, имя \n ");
scanf ("%d %lf %s", &course, &grant,  name);
```

Вводить данные с клавиатуры можно как в одной строке через пробелы, так и в форме разных строк, нажимая после ввода текущего объекта клавишу *Enter*.

Функция *scanf*() использует практически тот же набор модификаторов форматов, что и *printf*(); отличия от функции вывода следующие: отсутствует формат %*g*, форматы %*e*,%*f* – эквивалентны. Для ввода коротких целых чисел введен модификатор формата %*h*.

***Внимание***. Функцией *scanf*() по формату %*s* строка вводится только до первого пробела.

Для ввода фраз, состоящих из слов, разделенных пробелами, используется функция

***gets*** (*ID строковой переменной*);

Символы вводятся при помощи функции ***getch***(). Причем простой ее вызов организует паузу, при которой система программирования приостановит выполнение программы и будет ждать нажатия любой клавиши. Так поступают в том случае, когда нужно просмотреть какие-то результаты работы, при выводе их на экран монитора.

Если же использовать ее в правой части операции присваивания, например:

```
char c;
. . .
c = getch();
```

то символьная переменная *c* получит значение кода нажатой клавиши.

С началом работы любой программы автоматически открываются стандартные потоки для ввода (***stdin***) и вывода данных (***stdout***), которые по умолчанию связаны с клавиатурой и экраном монитора соответственно.

***Внимание***. Ввод данных функциями *gets*(), *getch*() выполняется с использованием потока *stdin*. Если указанная функция не выполняет своих действий (проскакивает), перед использованием необходимо очистить поток (буфер) ввода с помощью функции

***fflush***(*stdin*);

---

В языке C++ существует наиболее простая с точки зрения использования возможность ввода-вывода – потоковый ввод-вывод, основы которого рассмотрены в разд. 16.1, 16.2.

---

***Советы по программированию***

При выполнении вариантов заданий придерживайтесь следующих ключевых моментов.

1. Выбирайте тип для переменных с учетом диапазона их возможных значений и требуемой точности представления данных.

2. Старайтесь давать переменным *ID* (имена), отражающие их назначение.

3. При вводе данных с клавиатуры выводите на экран пояснения: что нужно ввести, т.е. организуйте диалог. Для контроля сразу же после ввода выводите исходные данные на дисплей (хотя бы в процессе отладки).

4. До запуска программы подготовьте тестовые примеры, содержащие исходные данные и ожидаемые результаты. Отдельно нужно проверить реакцию программы на заведомо неверные исходные данные. Для таких ситуаций необходимо предусмотреть вывод сообщений, например, «Ошибка! Решения нет».

5. При составлении выражений учитывайте приоритет используемых операций.

6. В функциях ввода/вывода *printf* и *scanf* для каждой переменной указывайте спецификацию формата, соответствующую ее типу. Не забывайте, что в функции *scanf* передается адрес переменной, а не ее значение.

7. При использовании стандартных функций требуется с помощью директивы препроцессору *include* подключить к программе соответствующие заголовочные файлы. Установить, какой именно файл необходим, можно с помощью справочной системы *Visual* C++ 6.0 – «*MSDN*».

8. Данные при вводе разделяйте пробелами, символами перевода строки или табуляции, но не запятыми.

---

Не смешивайте в одной программе ввод/вывод с помощью стандартных функций (в стиле Си) с вводом/выводом в потоке (в стиле C++).

---

## ЗАДАНИЕ 1. Составление линейных алгоритмов

***Первый уровень сложности***

Составить программу для расчета двух значений $z_1$ и $z_2$, результаты которых должны совпадать [32]. Ввод исходных данных можно задавать при декларации или вводить с клавиатуры. Игнорировать возможность деления на ноль. Значение $\pi$ = 3,1415926.

1. $z_1 = 2\sin^2(3\pi - 2\alpha)\cos^2(5\pi + 2\alpha)$, $z_2 = \frac{1}{4} - \frac{1}{4}\sin\left(\frac{5}{2}\pi - 8\alpha\right)$.

2. $z_1 = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha$ , $z_2 = 2\sqrt{2}\cos\alpha \cdot \sin\left(\frac{\pi}{4} + 2\alpha\right)$.

3. $z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos\alpha + 1 - 2\sin^2 2\alpha}$, $z_2 = 2\sin\alpha$ .

4. $z_1 = \cos^2\left(\frac{3}{8}\pi - \frac{\beta}{4}\right) - \cos^2\left(\frac{11}{8}\pi + \frac{\beta}{4}\right)$, $z_2 = \frac{\sqrt{2}}{2}\sin\frac{\beta}{2}$.

5. $z_1 = 1 - \frac{1}{4}\sin^2 2\alpha + \cos 2\alpha$ , $z_2 = \cos^2\alpha + \cos^4\alpha$ .

6. $z_1 = \cos\alpha + \cos 2\alpha + \cos 6\alpha + \cos 7\alpha$ , $z_2 = 4\cos\frac{\alpha}{2} \cdot \cos\frac{5}{2}\alpha \cdot \cos 4\alpha$.

7. $z_1 = \cos^2\left(\frac{3}{8}\pi - \frac{\alpha}{4}\right) - \cos^2\left(\frac{11}{8}\pi + \frac{\alpha}{4}\right)$, $z_2 = \frac{\sqrt{2}}{2}\sin\frac{\alpha}{2}$ .

8. $z_1 = \cos^4 x + \sin^2 y + \frac{1}{4}\sin^2 2x - 1$ , $z_2 = \sin(y + x) \cdot \sin(y - x)$ .

9. $z_1 = (\cos\alpha - \cos\beta)^2 - (\sin\alpha - \sin\beta)^2$ , $z_2 = -4\sin^2\frac{\alpha - \beta}{2} \cdot \cos(\alpha + \beta)$.

10. $z_1 = \frac{\sin\left(\frac{\pi}{2} + 3\alpha\right)}{1 - \sin(3\alpha - \pi)}$ , $z_2 = ctg\left(\frac{5}{4}\pi + \frac{3}{2}\alpha\right)$.

11. $z_1 = \frac{1 - 2\sin^2\alpha}{1 + \sin 2\alpha}$ , $z_2 = \frac{1 - tg\alpha}{1 + tg\alpha}$ .

12. $z_1 = \frac{\sin 4\alpha}{1 + \cos 4\alpha} \cdot \frac{\cos 2\alpha}{1 + \cos 2\alpha}$ , $z_2 = ctg\left(\frac{3}{2}\pi - \alpha\right)$.

13. $z_1 = \frac{\sin\alpha + \cos(2\beta - \alpha)}{\cos\alpha - \sin(2\beta - \alpha)}$ , $z_2 = \frac{1 + \sin 2\beta}{\cos 2\beta}$ .

14. $z_1 = \frac{(m-1)\sqrt{m} - (n-1)\sqrt{n}}{\sqrt{m^3 n} + nm + m^2 - m}$ , $z_2 = \frac{\sqrt{m} - \sqrt{n}}{m}$ .

15. $z_1 = \frac{\sqrt{2b + 2\sqrt{b^2 - 4}}}{\sqrt{b^2 - 4} + b + 2}$ , $z_2 = \frac{1}{\sqrt{b+2}}$.

***Второй уровень сложности***

Составить программу для расчета заданных выражений. Вводить исходные данные с клавиатуры. Обязательно проверять исключительные ситуации. Значение $\pi = 3{,}1415926$.

1. $t = \dfrac{2\cos\left(x - \dfrac{\pi}{6}\right)}{0{,}5 + \sin^2 y}\left(1 + \dfrac{z^2}{3 - z^2/5}\right)$.

При $x = 14.26$, $y = -1.22$, $z = 3.5\times10^{-2}$, результат $t = 0.564849$.

2. $u = \dfrac{\sqrt[3]{8 + |x - y|^2 + 1}}{x^2 + y^2 + 2} - e^{|x-y|}\left(tg^2 z + 1\right)^x$ .

При $x = -4.5$, $y = 0.75\times10^{-4}$, $z = 0.845\times10^{2}$, результат $u = -55.6848$.

3. $v = \dfrac{1 + \sin^2(x + y)}{\left|x - \dfrac{2y}{1 + x^2 y^2}\right|} x^{|y|} + \cos^2\left(arctg\,\dfrac{1}{z}\right)$.

При $x = 3.74\times10^{-2}$, $y = -0.825$, $z = 0.16\times10^{2}$, результат $v = 1.0553$.

4. $w = \left|\cos x - \cos y\right|^{\left(1 + 2\sin^2 y\right)}\left(1 + z + \dfrac{z^2}{2} + \dfrac{z^3}{3} + \dfrac{z^4}{4}\right)$.

При $x = 0.4\times10^{4}$, $y = -0.875$, $z = -0.475\times10^{-3}$, результат $w = 1.9873$.

5. $\alpha = \ln\left(y^{-\sqrt{|x|}}\right)\left(x - \dfrac{y}{2}\right) + \sin^2 arctg(z)$.

При $x = -15.246$, $y = 4.642\times10^{-2}$, $z = 20.001\times10^{2}$, результат $\alpha = -182.036$.

6. $\beta = \sqrt{10\left(\sqrt[3]{x} + x^{y+2}\right)} \cdot \left(\arcsin^2 z - |x - y|\right)$ .

При $x = 16.55\times10^{-3}$, $y = -2.75$, $z = 0.15$, результат $\beta = -38.902$.

7. $\gamma = 5\,arctg(x) - \dfrac{1}{4}\arccos(x)\dfrac{x + 3|x - y| + x^2}{|x - y|z + x^2}$.

При $x = 0.1722$, $y = 6.33$, $z = 3.25\times10^{-4}$, результат $\gamma = -172.025$.

8. $\varphi = \dfrac{e^{|x-y|}\,|x - y|^{x+y}}{arctg(x) + arctg(z)} + \sqrt[3]{x^6 + \ln^2 y}$.

При $x = -2.235\times10^{-2}$, $y = 2.23$, $z = 15.221$, результат $\varphi = 39.374$.

9. $\psi = \left| x^{\frac{y}{x}} - \sqrt[3]{\frac{y}{x}} \right| + (y-x)\frac{\cos y - \dfrac{z}{(y-x)}}{1+(y-x)^2}.$

При $x = 1.825\times10^{2}$, $y = 18.225$, $z = -3.298\times10^{-2}$, результат $\psi = 1.2131$.

10. $a = 2^{-x}\sqrt{x + \sqrt[4]{|y|}\sqrt[3]{e^{x-1/\sin z}}}.$

При $x = 3.981\times10^{-2}$, $y = -1.625\times10^{3}$, $z = 0.512$, результат $a = 1.26185$.

11. $b = y^{\sqrt[3]{|x|}} + \cos^3(y)\dfrac{|x-y|\cdot\left(1+\dfrac{\sin^2 z}{\sqrt{x+y}}\right)}{e^{|x-y|}+\dfrac{x}{2}}.$

При $x = 6.251$, $y = 0.827$, $z = 25.001$, результат $b = 0.7121$.

12. $c = 2^{(y^x)} + \left(3^x\right)^y - \dfrac{y\cdot\left(arctgz - \dfrac{\pi}{6}\right)}{|x| + \dfrac{1}{y^2+1}}.$

При $x = 3.251$, $y = 0.325$, $z = 0.466\times10^{-4}$, результат $c = 4.025$.

13. $f = \dfrac{\sqrt[4]{y+\sqrt[3]{x-1}}}{|x-y|\left(\sin^2 z + tgz\right)}.$

При $x = 17.421$, $y = 10.365\times10^{-3}$, $z = 0.828\times10^{5}$, результат $f = 0.33056$.

14. $g = \dfrac{y^{x+1}}{\sqrt[3]{|y-2|}+3} + \dfrac{x+\dfrac{y}{2}}{2|x+y|}(x+1)^{-1/\sin z}.$

При $x = 12.3\times10^{-1}$, $y = 15.4$, $z = 0.252\times10^{3}$, результат $g = 82.8257$.

15. $h = \dfrac{x^{y+1}+e^{y-1}}{1+x|y-tgz|}\left(1+|y-x|\right) + \dfrac{|y-x|^2}{2} - \dfrac{|y-x|^3}{3}.$

При $x = 2.444$, $y = 0.869\times10^{-2}$, $z = -0.13\times10^{3}$, результат $h = -0.49871$.

# ГЛАВА 6. Составление разветвляющихся алгоритмов

## *6.1. Краткая характеристика операторов языка Си*

Операторы языка Си можно разделить на три группы: операторы-декларации (рассмотрены ранее), операторы преобразования объектов и операторы управления процессом выполнения алгоритма.

Программирование процесса преобразования объектов производится посредством записи операторов (инструкций).

Простейший вид операторов – ***выражение***, заканчивающееся символом «;» (точка с запятой). Выполнение такого оператора заключается в вычислении некоторого выражения.

Простые операторы: оператор присваивания (выполнение операций присваивания), оператор вызова функции (выполнение операции вызова функции), пустой оператор «;» – частный случай выражения. Пустой оператор используют тогда, когда по синтаксису оператор требуется, а по смыслу – нет (например, смотри бесконечный оператор цикла *for* в разд. 7.4).

Примеры операторов *«выражение»*:

*i*++; – выполняется операция инкремента (увеличение на 1);

*x*+*y*; – выполняется операция сложения (результат будет утерян);

*a* = *b*–*c*; – выполняется операция вычитания с одновременным присваиванием.

Операторы языка Си записываются в свободном формате с использованием разделителей между ключевыми словами. Любой оператор может помечаться меткой – идентификатор и символ «:» (двоеточие). Область действия метки – функция, где эта метка определена.

К ***управляющим операторам*** относятся: операторы условного и безусловного переходов, оператор выбора альтернатив (переключатель), операторы организации циклов и передачи управления (перехода).

Каждый из управляющих операторов имеет конкретную лексическую конструкцию, образуемую из ключевых слов языка Си, выражений и символов-разделителей.

Допускается вложенность операторов. В случае необходимости можно использовать составной оператор – блок, состоящий из любой последовательности операторов, заключенных в фигурные скобки – { и }, после закрывающей скобки символ «;» не ставится.

## *6.2. Условные операторы*

Условный оператор ***if*** используется для разветвления процесса выполнения кода программы на два направления.

В языке Си имеется две разновидности условного оператора: простой и полный. Синтаксис ***простого*** оператора:

***if*** **(*выражение*)** ***оператор*;**

***выражение*** – логическое или арифметическое выражение, вычисляемое перед проверкой, и, если выражение истинно (не равно нулю), то выполняется ***оператор***, иначе он игнорируется; ***оператор*** – простой или составной (блок) оператор языка Си. Если в случае истинности выражения необходимо выполнить несколько операторов (более одного), их необходимо заключить в фигурные скобки.

Структурная схема простого оператора приведена на рис. 6.1.

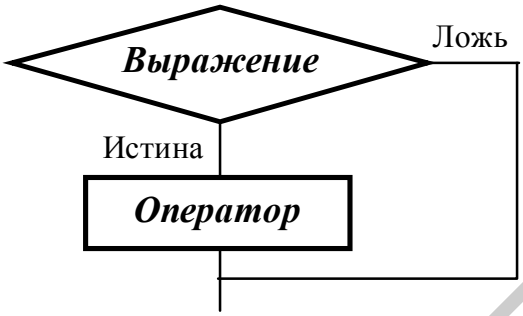



Рис. 6.1

Примеры записи условного оператора *if*:

```
if (x > 0) x = 0;
if (i != 1)   j++,  s = 1;     – используем операцию «запятая»;
if (i != 1) {
              j++;  s = 1;    – последовательность операций (блок);
}
if (getch() != 27) k = 0;   – если нажата любая клавиша кроме “Esc”.
if (!x) exit (1);       или   if (x == 0) exit(1);
```

if (i>0 && i<n) k++; – если нужно проверить несколько условий, то их объединяют знаками логических операций и заключают в круглые скобки (для улучшения читаемости программы можно ставить круглые скобки и там где они необязательны);

if (a++) b++; – необязательно в качестве выражения использовать логические выражения.

Синтаксис ***полного*** оператора условного выполнения:

***if* (*выражение*) *оператор* 1 ;**
**else *оператор* 2 ;**

Если ***выражение*** не равно нулю (истина), то выполняется ***оператор* 1**, иначе – ***оператор* 2**. Операторы 1 и 2 могут быть простыми или составными (блоками).

Наличие символа **«;»** перед словом ***else*** в языке Си обязательно.

Структурная схема такого оператора приведена на рис. 6.2.

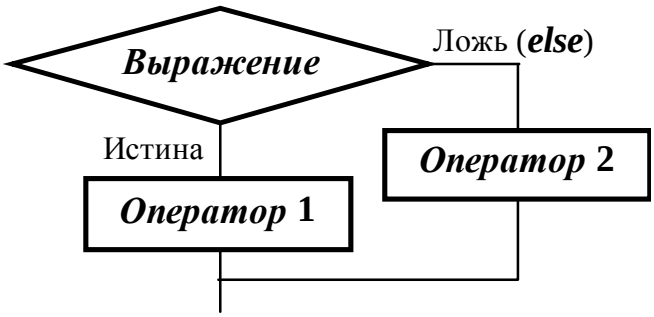



Рис. 6.2

Примеры записи:

```
if (x > 0) j = k+10;
        else m = i+10;

if ( x>0 &&  k!=0 ) {
                j = x/k;
                x += 10;
}
else m = k*i + 10;
```

*Операторы* 1 и 2 могут быть любыми операторами, в том числе и условными. Тогда, если есть вложенная последовательность операторов *if – else*, то фраза *else* связывается с ближайшим к ней предыдущим *if*, не содержащим ветвь *else*. Например:

```
if (n > 0)
   if(a > b)  z = a;
        else   z = b;
```

Здесь ветвь *else* связана со вторым *if* (a > b). Если же необходимо связать фразу *else* с внешним *if*, то используются операторные скобки:

```
if(n > 0) {
        if(a > b)  z = a;
        }
  else   z = b;
```

В следующей цепочке операторов *if – else – if* выражения просматриваются последовательно:

```
if (выражение 1)  оператор 1;
   else
        if (выражение 2)  оператор 2;
           else
                if (выражение 3)  оператор 3;
                   else   оператор 4 ;
```

Если какое-то *выражение* оказывается истинным, то выполняется относящийся к нему *оператор* и этим вся цепочка заканчивается. Каждый

оператор может быть либо отдельным оператором, либо группой операторов в фигурных скобках. ***Оператор* 4** будет выполняться только тогда, когда ни одно из проверяемых условий не подходит. Иногда при этом не нужно предпринимать никаких явных действий, тогда последний *else* может быть опущен или его можно использовать для контроля, чтобы зафиксировать «невозможное» условие (своеобразная экономия на проверке условий).

Пример:

```
if (x < 0) printf("\n X отрицательное \n");
        else if(x==0) printf ("\n X равно нулю \n");
                else printf("\n X положительное \n");
```

***Замечание***. Наиболее распространенной ошибкой при создании условных операторов является использование в выражении операции присваивания «=» вместо операции сравнения на равенство операндов «==» (два знака равно). Например, в следующем операторе синтаксической ошибки нет:

***if* (*x* = 5) *a*++;**

но значение ***a*** будет увеличено на единицу независимо от значения переменной ***x***, т.к. результатом операции присваивания $x = 5$ в круглых скобках является значение $5 \neq 0$ – истина.

### *6.3. Условная операция «? :»*

Условная операция – ***тернарная***, т.к. в ней участвуют три операнда. Формат написания условной операции следующий:

*Выражение* 1 **?** *выражение* 2 **:** *выражение* 3**;**

если *выражение* 1 (условие) отлично от нуля (истинно), то результатом операции является значение *выражения* 2, в противном случае – значение *выражения* 3. Каждый раз вычисляется только одно из выражений 2 или 3.

На рис. 6.3 приведена схема вычисления результата, которая аналогична схеме полного оператора *if* (см. рис. 6.2):

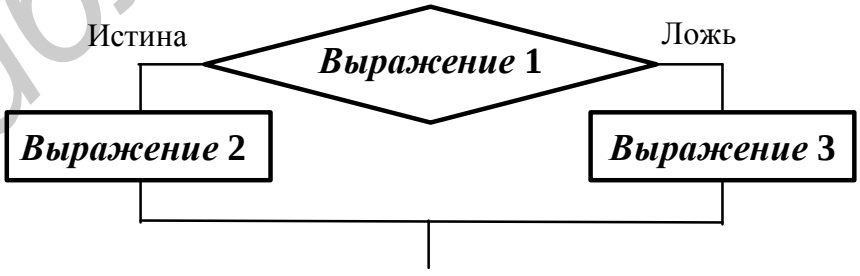



Рис. 6.3

Условное вычисление применимо к арифметическим операндам и операндам-указателям.

Рассмотрим участок программы для нахождения максимального значения ***z*** из двух чисел ***a*** и ***b***, используя оператор *if* и условную операцию.

1. Запишем оператор *if* :

```
if (a > b)  z = a;
       else  z = b;
```

2. Используя условную операцию, получим

```
z = (a > b) ? a : b;
```

Условную операцию можно использовать так же, как и любое другое выражение. Если выражения 2 и 3 имеют разные типы, то тип результата определяется по правилам преобразования. Например, если *f* имеет тип *double*, а *n – int*, то результатом операции

```
(n > 0) ? f : n;
```

по правилам преобразования типов будет *double*, независимо от того, положительно *n* или нет.

Использование условных выражений позволяет во многих случаях значительно упростить программу. Например:

```
int a, x;
   ...
x = (a < 0) ? –a : a;
printf("\n Значение %d  %s нулевое !", x, (x ? "не" : " ") );
```

### *6.4. Оператор выбора альтернатив (переключатель)*

Оператор *switch* (переключатель) предназначен для разветвления процесса вычислений на несколько направлений.

Общий вид оператора:

```
switch ( выражение ) {
    case  константа1:      список операторов 1
    case  константа2:      список операторов 2
                ...
    case  константаN:      список операторов N
    default:  список операторов N+1          – необязательная ветвь;
    }
```

Выполнение оператора начинается с вычисления ***выражения***, значение которого должно быть целого или символьного типа. Это значение сравнивается со значениями ***констант*** и используется для выбора ветви, которую нужно выполнить.

В данной конструкции ***константы*** фактически выполняют роль меток. Если значение выражения совпало с одной из перечисленных констант, то управление передается в соответствующую ветвь. После этого, если выход из переключателя в данной ветви явно не указан, последовательно выполняются все остальные ветви.

Все константы должны иметь разные значения, но быть одного и того же типа. Несколько меток могут следовать подряд, и тогда переход в указанную ветвь будет происходить при совпадении хотя бы одной из них. Порядок следования ветвей не регламентируется.

В случае несовпадения значения выражения ни с одной из констант выбора происходит переход на метку ***default*** либо, при ее отсутствии, к оператору, следующему за оператором *switch*.

Управляющий оператор ***break*** (разрыв) выполняет выход из оператора *switch*. Если по совпадению с каждой константой должна быть выполнена одна и только одна ветвь, схема оператора switch следующая:

```
switch (выражение) {
                case константа1: операторы 1;  break;
                case константа2: операторы 2;  break;
                        ...
                case константаN: операторы N;  break;
                default:  операторы (N+1);  break;
        }
```

Структурная схема рассмотренной конструкции (с использованием оператора *break*) приведена на рис. 6.4.

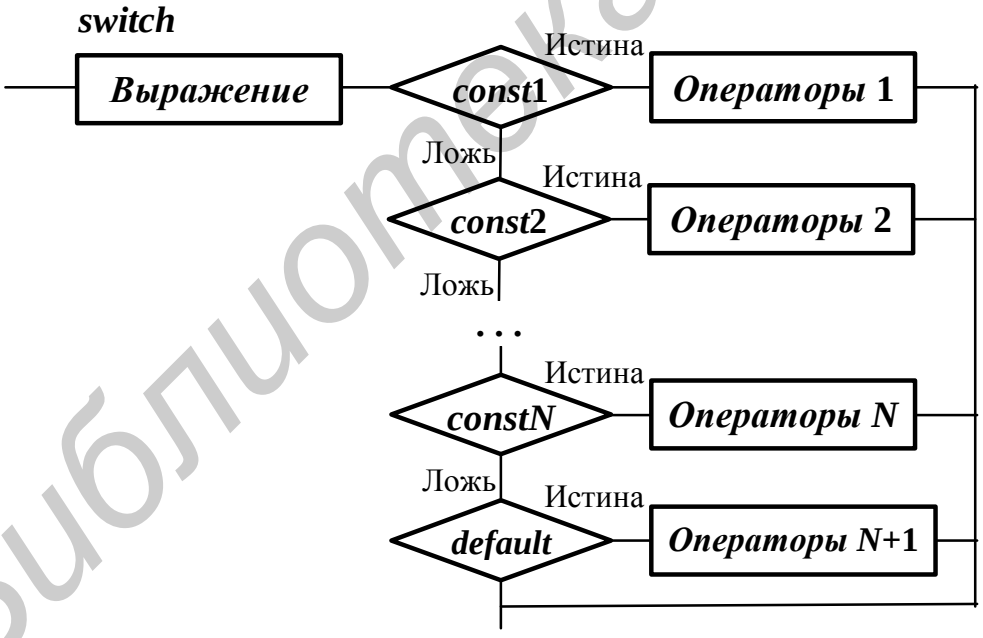



Рис. 6.4

Пример оператора *switch* с использованием оператора *break*:

```
void main(void)    {
        int i = 2;
        switch(i)      {
                case 1:  puts ( "Случай 1. "); break;
                case 2:  puts ( "Случай 2. "); break;
```

```
        case 3:  puts ( "Случай 3. "); break;
        default: puts ( "Случай default. "); break;
        }
}
```

Результатом выполнения данной программы будет:

*Случай 2.*

Аналогичный пример без использования оператора ***break*** (схема общего вида такой конструкции приведена рис. 6.5):

```
void main() {
        int i = 2;
        switch(i)      {
                case 1: puts ( "Случай 1. ");
                case 2: puts ( "Случай 2. ");
                case 3: puts ( "Случай 3. ");
                default: puts ( "Случай default. ");
                }
        }
```

В данном случае результат будет следующим:

*Случай 2.*
*Случай 3.*
*Случай default.*

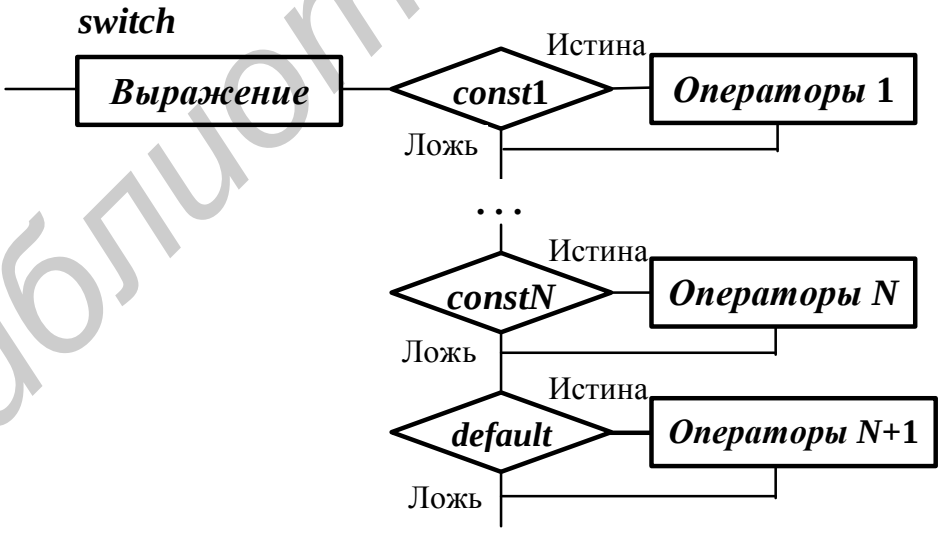



Рис. 6.5

Пример реализации простейшего калькулятора на четыре действия с контролем правильности ввода символа нужной операции. Ввод данных осуществляется следующим образом: операнд 1, символ нужной операции, операнд 2.

Текст программы может быть следующим:

```
#include <stdio.h>
void main(void)
{
        double a, b, c;
        char s;
        m1:  fflush(stdin);                  // Очистка буфера ввода stdin
        printf("\n Введите операнд 1, символ операции, операнд 2:");
        scanf("%lf%c%lf", &a, &s, &b);
        switch(s)  {
                case '+':       c = a+b;      break;
                case '–':       c = a–b;      break;
                case '*':       c = a*b;      break;
                case '/':       c = a/b;      break;
                default: printf("\n Ошибка, повторите ввод! "); goto m1;
                }
          printf("\n a %c b = %lf ", s, c);
          printf("\n Продолжим? (Y/y) ");
          s = getch();
          if ( (s=='Y') || (s=='y') ) goto m1;
          printf("\n Good bye! ");
}
```

После запуска программы на экран выводится подсказка, нужно набрать соответствующие значения без пробелов, например, как показано ниже, и нажать клавишу *Enter*:

*Введите операнд 1, символ операции, операнд 2:  2.4+3.6*

На экран будет выведен результат и дальнейший диалог:

*a + b = 6.000000*

*Продолжим? (Y/y)*

Введя символ **y** (**Y**), вернемся в начало функции и на экране вновь появится:

*Введите операнд 1, символ операции, операнд 2:*

Если ошибочно ввести – **2r3 ,** появятся следующие сообщения:

*Ошибка, повторите ввод!*

*Введите операнд 1, символ операции, операнд 2:  2 * 3*

*a*b = 6.000000*

*Continue? (Y/y)*

Нажимаем любую клавишу, кроме *y* или *Y* – следует сообщение

*Good bye!*

Программа закончена.

# ГЛАВА 7. Составление циклических алгоритмов

## *7.1. Понятие циклического кода*

Практически все алгоритмы решения задач содержат циклически повторяемые участки. Цикл – это одно из фундаментальных понятий программирования. Под циклом понимается организованное повторение некоторой последовательности операторов.

Любой цикл состоит из кода цикла, т.е. тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла.

Один проход цикла называется шагом или итерацией. Проверка условия продолжения цикла происходит на каждой итерации либо до выполнения кода цикла (с предусловием), либо после выполнения (с постусловием).

Для организации циклов используются специальные операторы. Перечень разновидностей операторов цикла языка Си следующий:

– оператор цикла с предусловием;
– оператор цикла с постусловием;
– оператор цикла с предусловием и коррекцией.

## *7.2. Оператор с предусловием while*

Цикл с предусловием реализует структурную схему, приведенную на рис. 7.1, *а*, и имеет вид

***while*** **(***выражение***)**
*код цикла;*

Выражение определяет условие повторения кода цикла, представленного простым или составным оператором.

Если выражение в скобках – истина (не равно 0), то выполняется *код цикла*. Это повторяется до тех пор, пока выражение не примет значение 0 (ложь). В этом случае происходит выход из цикла и выполняется оператор, следующий за конструкцией ***while***. Если выражение в скобках изначально ложно (т.е. равно 0), то цикл не выполнится ни разу.

*Код цикла* может включать любое количество операторов, связанных с конструкцией ***while***, которые нужно заключить в фигурные скобки (организовать блок), если их более одного.

Переменные, изменяющиеся в коде цикла и используемые при проверке условия продолжения, называются параметрами цикла. Целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации, называются счетчиками цикла.

Начальные установки могут явно не присутствовать в программе, их смысл состоит в том, чтобы до входа в цикл задать значения переменным, которые в этом цикле используются.

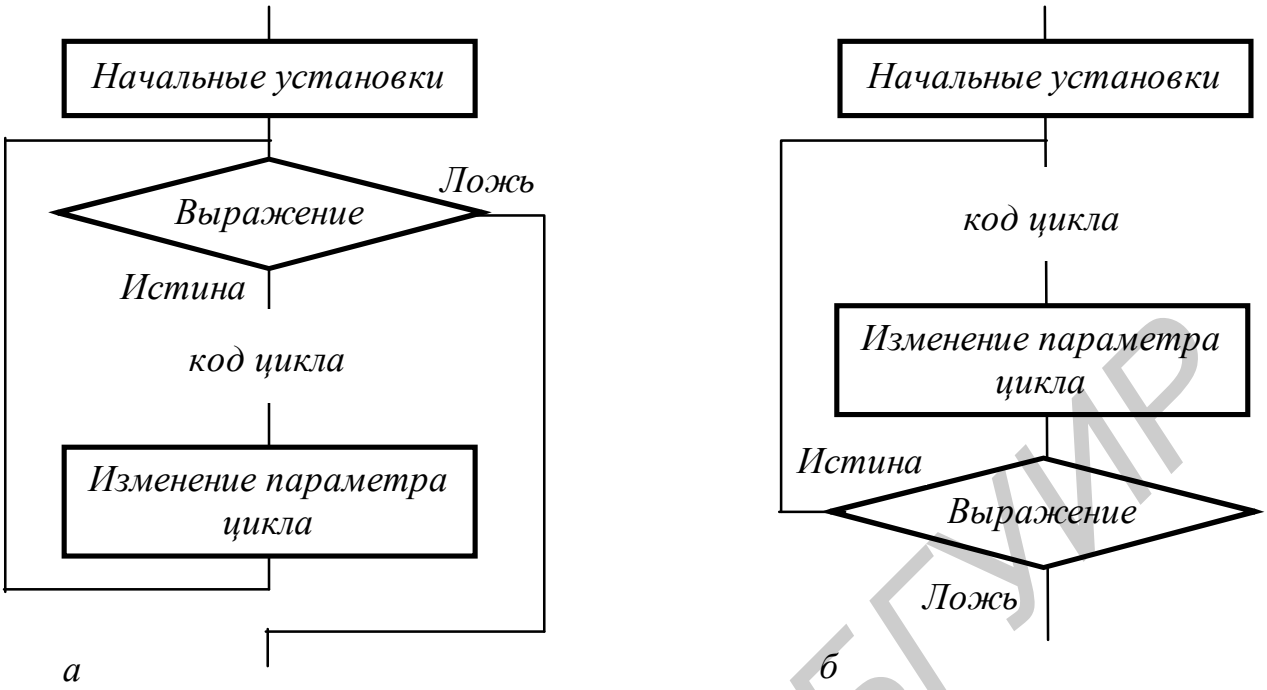



Рис. 7.1. Схемы операторов цикла:
*а* – цикл с предусловием; *б* – цикл с постусловием

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом.

Для этого используют оператор ***continue*** – переход к следующей итерации цикла и ***break*** – выход из цикла (см. разд. 9.2, 9.3).

Передавать управление извне внутрь цикла не рекомендуется, так как получите непредсказуемый результат.

Например, необходимо сосчитать количество символов в строке. Предполагается, что входной поток настроен на начало строки. Тогда подсчет символов выполняется следующим образом:

```
int count = 0;
char ch = getchar();
while ( ch != \n') {
      count++;
      ch = getchar();
 }
```

В языке Си в выражение, управляющее циклом, можно включить и оператор присваивания переменной *ch*, например:

```
char ch;
int count = 0;
while (( ch=getchar()) != '\n') count++;
```

Как видим, переменная *ch* применяется только в выражении, управляющем циклом, поэтому от ch можно отказаться:

```
int count = 0;
while ( getchar() !='\n') count ++;
```

***Полезные примеры***

1. Организация выхода из бесконечного цикла по нажатии клавиши *Esc*

```
while (1) {                          // Бесконечный цикл
        ...
    if (kbhit() && getch()==27 ) break;
        ...
}
```

Функция ***kbhit***() возвращает значение > 0, если нажата любая клавиша, а функция *getch*() возвращает код нажатой клавиши (код клавиши *Esc* равен 27). В результате выполнения оператора *if*, если будет нажата клавиша *Esc*, выполнится оператор *break* и произойдет выход из цикла.

Приведенный пример – распространенный прием программирования.

2. Организации паузы в работе программы с помощью цикла, выполняющегося до тех пор, пока не нажата любая клавиша

```
    ...
while (!kbhit());
    ...
```

### *7.3. Оператор цикла с постусловием do – while*

Цикл с постусловием реализует структурную схему, приведенную на рис. 7.1, *б*.

Общий вид записи такой конструкции

***do***
*код цикла;*
***while*** (*выражение*);

*Код цикла* будет выполняться до тех пор, пока *выражение* истинно. Все, что говорилось выше, справедливо и здесь, за исключением того, что данный цикл всегда выполняется хотя бы один раз, даже если изначально выражение ложно.

Здесь сначала выполняется код цикла, после чего проверяется, надо ли его выполнять еще раз.

Следующая программа будет «вас приветствовать» до тех пор, пока будем вводить символ *Y* или *y* (*Yes*). После введения любого другого символа цикл завершит свою работу.

```
#include <stdio.h>
void main(void)
{
```

```
        char answer;
        do {
                puts(" Hello! => ");
                scanf(" %c ", &answer);
        }
        while ((answer=='y')||(answer=='Y'));
}
```

Результат выполнения программы:

*Hello! => Y*
*Hello! => y*
*Hello! => d*

### *7.4. Оператор цикла с предусловием и коррекцией for*

Общий вид оператора:

***for*** (*выражение* 1; *выражение* 2; *выражение* 3)
*код цикла*;

где *выражение* 1 – инициализация счетчика (параметр цикла);
*выражение* 2 – условие продолжения счета;
*выражение* 3 – коррекция счетчика.

На рис. 7.2, *а* представлена схема работы цикла *for*, а на рис. 7.2, *б* – символ блок-схемы, использующийся для его обозначения.

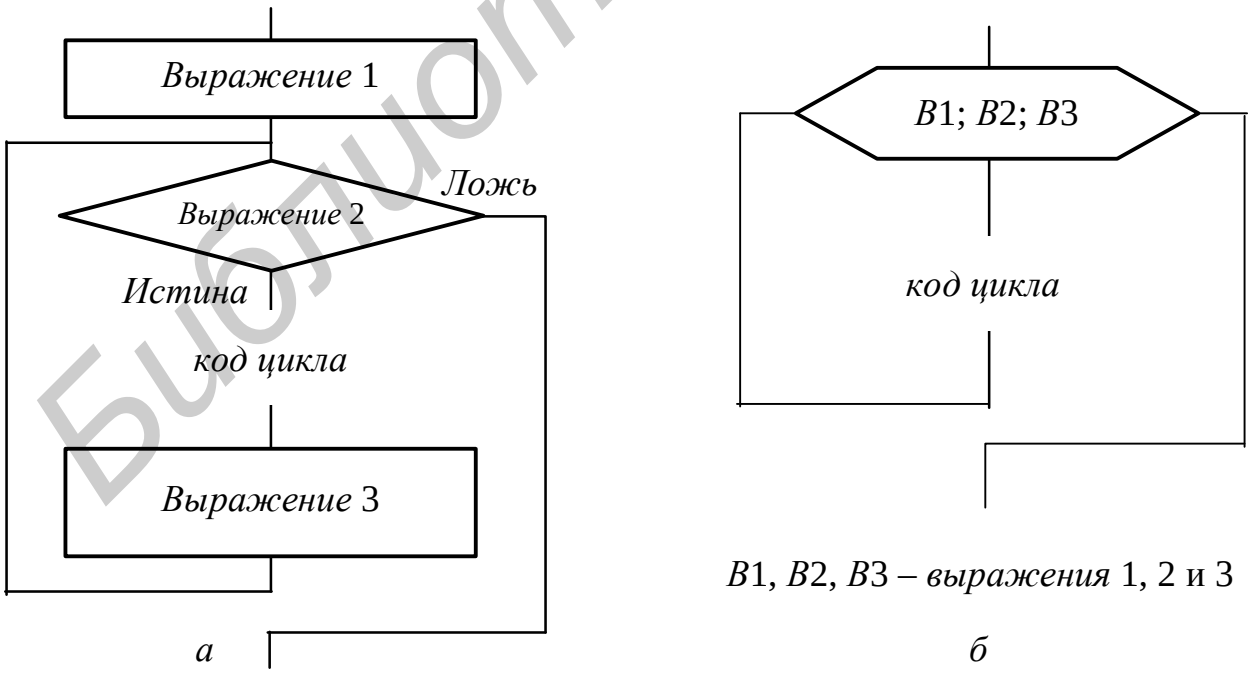



Рис. 7.2. Схемы оператора цикла *for:*
*а* – схема работы; *б* – блок-схема

Инициализация используется для присвоения счетчику (параметру цикла) начального значения.

*Выражение* 2 определяет условие выполнения цикла. Как и в предыдущих случаях, если его результат не нулевой («истина»), – то цикл выполняется, иначе – происходит выход из цикла.

Коррекция выполняется после каждой итерации цикла и служит для изменения параметра цикла.

*Выражения* 1, 2 и 3 могут отсутствовать (пустые выражения), но символы «;» опускать нельзя.

Например, для суммирования первых *N* натуральных чисел можно записать такой код:

```
sum = 0;
for ( i = 1; i<=N; i++)  sum+=i;
```

Заметим, что в *выражении* 1 переменную-счетчик можно декларировать. Например:

```
for (int i = 1; i<=N; i++)
```

Областью действия такой переменной будет код цикла.

Но в старых версиях компиляторов такие действия могут интерпретироваться иначе.

Цикл *for* эквивалентен последовательности инструкций:

```
выражение 1;
while (выражение 2) {
        ...
    выражение 3;
}
```

а оператор

```
for (; выражение 2; )
        код цикла;
```

эквивалентен оператору

```
while (выражение 2)
        код цикла;
```

Если пропущено *выражение* 2, то цикл будет выполняться бесконечно, поскольку пустое условие всегда остается истинным. Бесконечный оператор:

```
for ( ; ; ) код цикла;
```

эквивалентен оператору

```
while (1) код цикла;
```

В заголовке оператора *for* может использоваться операция «запятая». Она позволяет включать в его выражения несколько операторов. Тогда рассмотренный пример суммирования первых *N* натуральных чисел можно записать в следующем виде:

```
for ( sum = 0 , i = 1; i<=N; sum+= i , i++) ;
```

Оператор *for* имеет следующие возможности:

– можно вести подсчет с помощью символов, а не только чисел:

```
for (ch = 'a'; ch <= 'z'; ch++) ... ;
```

– можно проверить выполнение некоторого произвольного условия:

```
for (n = 0; s[i] >= '0' && s[i] < '9'; i++) ... ;
```

или

```
for (n = 1; n*n*n <= 216; n++) ... ;
```

Первое выражение необязательно должно инициализировать переменную. Необходимо только помнить, что первое выражение вычисляется только один раз, перед тем как остальные части начнут выполняться.

```
for (printf(" вводить числа по порядку! \n"); num!=6;)
      scanf("%d", & num);
printf(" последнее число – это то, что нужно. \n");
```

В этом фрагменте первое сообщение выводится на печать один раз, а затем осуществляется прием вводимых чисел, пока не поступит число 6.

Переменные, входящие в *выражения* 2 и 3, можно изменять при выполнении кода цикла, например, значения *k* и *delta*:

```
for (n = 1; n < 10*k; n += delta) ... ;
```

Использование условных выражений позволяет во многих случаях значительно упростить программу, например:

```
for (i = 0; i<n; i++)
      printf("%6d%c", a[i],( (i%10==0) || (i==n–1) ) ? '\n' : ′ ′);
```

В этом цикле печатаются *n* элементов массива *a* по 10 в строке, разделяя каждый столбец одним пробелом и заканчивая каждую строку (включая последнюю) одним символом перевода строки. Символ перевода строки записывается после каждого десятого и *n*-го элементов. За всеми остальными – пробел.

Наиболее часто встречающиеся ***ошибки*** при создании циклов – это использование в коде цикла неинициализированных переменных и неверная запись условия выхода из цикла.

Чтобы избежать ошибок, нужно стараться:

– проверить, всем ли переменным, встречающимся в правой части операторов присваивания в коде цикла, присвоены до этого начальные значения (а также возможно ли выполнение других операторов);

– проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;

– предусмотреть аварийный выход из цикла по достижении некоторого количества итераций;

– если в состав цикла входит не один, а несколько операторов, нужно заключать их в фигурные скобки.

# ГЛАВА 8. Операторы и функции передачи управления

Формально к операторам передачи управления относятся:
– оператор безусловного перехода ***goto***;
– оператор перехода к следующему шагу (итерации) цикла ***continue***;
– выход из цикла, либо оператора *switch* – ***break***;
– оператор возврата из функции ***return***.

## *8.1. Оператор безусловного перехода goto*

В языке Си предусмотрен оператор *goto*, общий вид которого

***goto*** *метка* ;

Он предназначен для передачи управления оператору, помеченному указанной *меткой*. Метка представляет собой идентификатор, оформленный по всем правилам идентификации переменных с символом «двоеточие» после него, например, пустой помеченный меткой *m*1 оператор:

*m*1: ;

Область действия метки – функция, где эта метка определена. В случае необходимости можно использовать блок.

Циклы и переключатели можно вкладывать вдруг в друга и наиболее характерный оправданный случай использования оператора *goto* – выполнение прерывания (организация выхода) во вложенной структуре. Например, при возникновении грубых неисправимых ошибок необходимо выйти из двух (или более) вложенных структур (где нельзя использовать непосредственно оператор *break*, т.к. он прерывает только самый внутренний цикл):

```
for (...)
        for (...) {
                ...
                if (ошибка)  goto error;
        }
                ...
      error:  операторы для устранения ошибки;
```

Второй оправданный случай: организация переходов из нескольких мест функции в одно, например, когда перед завершением работы функции необходимо сделать одну и ту же операцию.

## *8.2. Операторы continue, break и return*

Оператор ***continue*** может использоваться во всех типах циклов (но не в операторе-переключателе *switch*). Наличие оператора *continue* вызывает пропуск «оставшейся» части итерации и переход к началу следующей, т.е. досрочное завершение текущего шага и переход к следующему шагу.

В циклах *while* и *do-while* это означает непосредственный переход к проверочной части. В цикле *for* управление передается на шаг коррекции, т.е. модификации *выражения* 3.

Оператор *continue* часто используется, когда последующая часть цикла оказывается слишком сложной, так что рассмотрение условия, обратного проверяемому, приводит к слишком высокому уровню вложенности программы.

Оператор ***break*** производит досрочный выход из цикла или оператора-переключателя *switch*, к которому он принадлежит, и передает управление первому оператору, следующему за текущим оператором. То есть *break* обеспечивает переход в точку кода программы, находящуюся за оператором, внутри которого он (*break*) находится.

Оператор ***return*** производит досрочный выход из текущей функции. Он также возвращает значение результата функции:

***return*** *выражение*;

*Выражение* должно иметь скалярный тип. Правила использования данного оператора будут рассмотрены в гл. 12.

### *8.3. Функции exit и abort*

Функция ***exit*** выполняет прерывание программы и используется для нормального, корректного завершения работы программы при возникновении какой-либо внештатной ситуации, например, ошибка при открытии файла (гл. 14). При этом записываются все буферы в соответствующие файлы, закрываются все потоки и вызываются все зарегистрированные стандартные функции завершения.

Прототип этой функции приведен в заголовочном файле *stdlib.h* и выглядит так:

*void* ***exit*** **(** *int* ***exit_code*****);**

Параметр данной функции – ненулевое целое число, передаваемое системе программирования (служебное сообщение о возникшей внештатной ситуации).

Для завершения работы программы также может использоваться функция

*void* ***abort*** (*void*);

действия которой аналогичны функции *exit*(3).

### *Советы по программированию*

При выполнении вариантов заданий придерживайтесь следующих ключевых моментов.

1. Выражение, стоящее в круглых скобках операторов *if*, *while* и *do-while*, вычисляется по правилам стандартных приоритетов операций.

2. Если в какой-либо ветви вычислений условного оператора или в цикле требуется выполнить два (и более) оператора, то они при помощи фигурных скобок объединяются в блок.

3. Проверка вещественных величин на равенство, как правило, из-за ограниченной разрядности дает неверный результат.

4. Чтобы получить максимальную читаемость и простоту структуры программы, надо правильно выбирать способ реализации ветвлений (с помощью *if*, *switch*, или условных операций), а также наиболее подходящий оператор цикла.

5. Выражение в операторе *switch* и константные выражения в *case* должны быть целочисленного или символьного типов.

6. Рекомендуется использовать в операторе *switch* ветвь *default*.

7. После каждой ветви для передачи управления на точку кода за оператором *switch* используется оператор *break*.

8. При построении любого цикла надо не забывать тот факт, что в нем всегда явно или неявно присутствуют четыре основных элемента: начальные установки, код цикла, модификация параметра цикла и проверка условия на продолжение цикла.

9. Если количество повторений цикла заранее не известно (реализуется итерационный процесс), необходимо предусмотреть аварийное завершение цикла при получении достаточно большого количества итераций.

10. При использовании бесконечного цикла обязательно необходима организация выхода из цикла по условию.

## ЗАДАНИЕ 2. Разветвляющиеся алгоритмы

***Первый уровень сложности***

Составить программу нахождения требуемого значения с исходными данными *x*, *y*, *z*. Обозначение: *min* и *max* – нахождение минимального и максимального из перечисленных в скобках значений элементов.

1. $m = \dfrac{\max(x, y, z)}{\min(x, y)} + 5$ ;

2. $n = \dfrac{\min(x + y, y - z)}{\max(x, y)}$ ;

3. $p = \dfrac{|\min(x, y) - \max(y, z)|}{2}$ ;

4. $q = \dfrac{\max(x + y + z,\ x \cdot y \cdot z)}{\min(x + y + z,\ x \cdot y \cdot z)}$ ;

5. $r = \dfrac{\max[\min(x, y), z]}{3}$ ;

6. $s = \dfrac{\min[\max(x, y), \max(y, z)]}{\max(y, z)}$ ;

7. $t = \dfrac{\max[\min(x, 5), \max(y, 0)]}{5}$ ;

8. $v = \max[\min(x - y, y - x), 0]$ ;

9. $w=\max^2[\max(x\cdot y, x+y), 0]$; 10. $z=\dfrac{\min(0,x)\ -\min(0,y)}{\max^2(y,x)}$;

11. $u=\dfrac{\min(y,z)}{\max[\min(x,y),\min(y,z)]}$; 12. $q=\dfrac{\min(x+y+z,\ x\cdot y\cdot z)}{\min(x-y+z,\ x\cdot y/z)}$;

13. $q=\dfrac{\max(x+y+z,\ x\cdot y\cdot z)}{\max[x+y+z,\ x/(y\cdot z)]}$; 14. $u=\dfrac{\max(y,z)}{\min[\min(x,y),\min(y,z)]}$;

15. $u=\dfrac{\min(y,z)}{\max[\max(x,y),\max(y,z)]}$.

***Второй уровень сложности***

Вычислить значение $y$ в зависимости от выбранной функции $\varphi(x)$, аргумент которой определяется из поставленного условия. Возможные значения функции $\varphi(x)$: $2x$, $x^2$, $x/3$. Предусмотреть вывод сообщений, показывающих, при каком условии и с какой функцией производились вычисления $y$.

1. $y=a\ln(1+x^{1/5})+\cos^2[\varphi(x)+1]$, где $x=\begin{cases} z^2; & z<1; \\ z+1; & z\ge 1. \end{cases}$

2. $y=\dfrac{2a\varphi(x)+b\cos\sqrt{|x|}}{x^2+5}$, где $x=\begin{cases} 2+z; & z<1; \\ \sin^2 z; & z\ge 1. \end{cases}$

3. $y=-\pi\varphi(x)+a\cos^2 x^3+b\sin^3 x^2$, где $x=\begin{cases} z; & z<1; \\ \sqrt{z^3}; & z\ge 1. \end{cases}$

4. $y=2a\cos^3 x^2+\sin^2 x^3-b\varphi(x)$, где $x=\begin{cases} z^3+0{,}2; & z<1; \\ z+\ln z; & z\ge 1. \end{cases}$

5. $y=a\varphi(x)-\ln(x+2{,}5)+b(e^x-e^{-x})$, где $x=\begin{cases} -z/3; & z<-1; \\ |z|; & z\ge -1. \end{cases}$

6. $y=\dfrac{2}{3}a\sin^2 x-\dfrac{3b}{4}\cos^2\varphi(x)$, где $x=\begin{cases} z; & z<0; \\ \sin z; & z\ge 0. \end{cases}$

7. $y=\sin^3[c\varphi(x)+d^2+x^2]$, где $x=\begin{cases} z^2-z; & z<0; \\ z^3; & z\ge 0. \end{cases}$

8. $y=\sin^2\varphi(x)+a\cos^5 x^3+c\ln x^{2/5}$, где $x=\begin{cases} 2z+1; & z\ge 0; \\ \ln(z^2-z); & z<0. \end{cases}$

9. $y=\dfrac{b\varphi(x)}{\cos x}+a\ln\left|tg\dfrac{x}{2}\right|$, где $x=\begin{cases}z^2/2; & z\le 0;\\ \sqrt{z}; & z>0.\end{cases}$

10. $y=\dfrac{d\varphi(x)e^{\sin^3 x}+c\ln(x+1)}{\sqrt{x}}$, где $x=\begin{cases}z^2+1; & z<1;\\ z-1; & z\ge 1;\end{cases}$

11. $y=\dfrac{2{,}5a\cdot e^{-3x}-4bx^2}{\ln|x|+\varphi(x)}$, где $x=\begin{cases}\dfrac{1}{z^2+2z}; & z>0;\\ 1-z^3; & z\le 0.\end{cases}$

12. $y=a\sin^3[\varphi(x)^2-1]+c\ln|x|+e^x$, где $x=\begin{cases}z^2+1; & z\le 1;\\ 1/\sqrt{z-1}; & z>1.\end{cases}$

13. $y=\sin[n\varphi(x)]+\cos kx+\ln mx$, где $x=\begin{cases}z; & z>1;\\ z^2+1; & z\le 1.\end{cases}$

14. $y=b\cos[a\varphi(x)]+\sin\dfrac{x}{5}+ae^x$, где $x=\begin{cases}\sqrt{z}; & z>0;\\ 3z+1; & z\le 0.\end{cases}$

15. $y=2\varphi(x)[a\sin x+d\cdot e^{-(x+3)}]$, где $x=\begin{cases}-3z; & z>0;\\ z^2; & z\le 0.\end{cases}$

## ЗАДАНИЕ 3. Циклические алгоритмы

***Первый уровень сложности***

Составить программу для определения таблицы значений функции ***y*** в произвольном диапазоне [***a***, ***b***] изменения аргумента ***x*** с произвольным шагом ***h***. Значения ***a, b, h*** вводятся с клавиатуры. Таблица должна содержать следующие столбцы: порядковый номер, значение аргумента ***x***, значение функции, сообщение о возрастании или убывании функции.

Определить максимальное и минимальное значения функции.

1. $Y(x)=\dfrac{2\sin x}{(1-x)^2}$, $a = -\pi$; $b = \pi$; $h = 0{,}4$.

2. $Y(x)=-\ln\left|2\sin\dfrac{x}{2}\right|$, $a = 0{,}7$; $b = 1{,}8$; $h = 0{,}1$.

3. $Y(x)=\dfrac{x\sin(\frac{\pi}{4})}{1-2x\cos\frac{\pi}{4}+x^2}$, $a = -0{,}5$; $b = 2{,}5$; $h = 0{,}2$.

4. $Y(x)=(1-\dfrac{x^2}{4})\cos x-\dfrac{x}{2}\sin x$, $a = -0{,}9$; $b = 2{,}7$; $h = 0{,}3$.

5. $Y(x)=\dfrac{x\cos\dfrac{\pi}{4}-x^2}{1-2x\cos\dfrac{\pi}{4}+x^2}$, $a = -2$; $b = 0{,}8$; $h = 0{,}2$.

6. $Y(x)=(\dfrac{x^2}{4}+\dfrac{x}{2}-3)\cdot e^{\frac{x}{2}}$, $a = -1{,}9$; $b = 2{,}7$; $h = 0{,}3$.

7. $Y(x)=x^2\sqrt{15+10\sin(x+\pi)}$, $a = -0{,}4\pi$; $b = 0{,}4\pi$; $h = 0{,}5$.

8. $Y(x)=e^x\sin x$, $a = -0{,}3\pi$; $b = 1{,}3\pi$; $h = \pi/10$.

9. $Y(x)=x^2\cos x\sin x$, $a = -\pi/2$; $b = \pi/2$; $h = \pi/10$.

10. $Y(x)=x\log(|x-0{,}6|)$, $a = -3$; $b = 3$; $h = 0{,}5$.

11. $Y(x)=\dfrac{x}{2}\cos x-\sin x$, $a = -\pi$; $b = \pi$; $h = \pi/6$.

12. $Y(x)=e^x+\sqrt{1+e^{2x}}-2$, $a = -0{,}9$; $b = 1$, $h = 0{,}3$.

13. $Y(x)=(1-\dfrac{x^2}{4})\cos x-\dfrac{x}{2}\sin x$, $a = -0{,}9$; $b = 2{,}7$; $h = 0{,}3$.

14. $Y(x)=\dfrac{1}{x^2-x+1}$, $a = -0{,}1$; $b = 2$; $h = 0{,}1$.

15. $Y(x)=\dfrac{\sin x\cos x}{\sqrt{x}}$, $a = \pi$; $b = 2\pi$; $h = \pi/15$.

***Второй уровень сложности***

Значение аргумента ***x*** изменяется от ***a*** до ***b*** с шагом ***h***. Для каждого ***x*** найти значения функции $Y(x)$, суммы $S(x)$ и $|Y(x)–S(x)|$ и вывести в виде таблицы. Значения $a$, $b$, $h$ и $n$ вводятся с клавиатуры. Так как значение $S(x)$ является рядом разложения функции $Y(x)$, значения $S$ и $Y$ для заданного аргумента $x$ должны совпадать в целой части и в первых двух-четырех позициях после десятичной точки.

Работу программы проверить для $a = 0{,}1$; $b = 1{,}0$; $h = 0{,}1$; значение параметра $n$ выбрать в зависимости от задания.

1. $S(x)=\sum_{k=0}^{n}(-1)^k\dfrac{x^{2k+1}}{(2k+1)!}$, $Y(x)=\sin(x)$.

2. $S(x)=\sum_{k=1}^{n}(-1)^{k+1}\dfrac{\sin(kx)}{k}$, $Y(x)=\dfrac{x}{2}$.

3. $S(x)=\sum_{k=0}^{n}\dfrac{\cos(\frac{k\pi}{4})}{k!}x^k$, $Y(x)=e^{x\cos\frac{\pi}{4}}\cos(x\sin(\pi/4))$.

4. $S(x)=\sum_{k=0}^{n}(-1)^k\frac{x^{2k}}{(2k)!}$, $Y(x)=\cos(x)$.

5. $S(x)=\sum_{k=0}^{n}\frac{\cos(kx)}{k!}$, $Y(x)=e^{\cos x}\cos(\sin(x))$.

6. $S(x)=\sum_{k=0}^{n}\frac{2k+1}{k!}x^{2k}$, $Y(x)=(1+2x^2)e^{x^2}$.

7. $S(x)=\sum_{k=1}^{n}\frac{x^k\cos\frac{k\pi}{3}}{k}$, $Y(x)=-\frac{1}{2}\ln(1-2x\cos\frac{\pi}{3}+x^2)$.

8. $S(x)=\sum_{k=1}^{n}(-1)^k\frac{\cos(kx)}{k^2}$, $Y(x)=\frac{1}{4}(x^2-\pi^2/3)$.

9. $S(x)=\sum_{k=1}^{n}(-1)^{k+1}\frac{x^{2k+1}}{4k^2-1}$, $Y(x)=\frac{1+x^2}{2}arctg(x)-x/2$.

10. $S(x)=\sum_{k=0}^{n}\frac{x^{2k}}{(2k)!}$, $Y(x)=\frac{e^x+e^{-x}}{2}$.

11. $S(x)=\sum_{k=0}^{n}\frac{k^2+1}{k!}(x/2)^k$, $Y(x)=(x^2/4+x/2+1)e^{x/2}$.

12. $S(x)=\sum_{k=0}^{n}(-1)^k\frac{2k^2+1}{(2k)!}x^{2k}$, $Y(x)=(1-\frac{x^2}{2})\cos(x)-\frac{x}{2}\sin(x)$.

13. $S(x)=\sum_{k=1}^{n}(-1)^k\frac{(2x)^{2k}}{(2k)!}$, $Y(x)=2(\cos^2 x-1)$.

14. $S(x)=\sum_{k=0}^{n}\frac{x^{2k+1}}{(2k+1)!}$, $Y(x)=\frac{e^x-e^{-x}}{2}$.

15. $S(x)=\sum_{k=1}^{n}(-1)^{k+1}\frac{x^{2k}}{2k(2k-1)}$, $Y(x)=xarctg(x)-\ln\sqrt{1+x^2}$.



# ГЛАВА 9. Указатели

## *9.1. Определение указателей*

При обработке декларации любой переменной, например *double x*=1.5; компилятор выделяет для переменной участок памяти, размер которого определяется ее типом (*double* – 8 байт), и инициализирует его указанным значением (если таковое имеется). Далее все обращения в программе к

переменной по имени заменяются компилятором на адрес участка памяти, в котором будет храниться значение этой переменной. Разработчик программы на языке Си имеет возможность определить собственные переменные для хранения адресов участков оперативной памяти. Такие переменные называются указателями.

Итак, указатель – это переменная, которая может содержать адрес некоторого объекта. Простейшая декларация указателя имеет формат

***тип * ID_указателя*;**

Например: int *a; double *f; char *w;

Здесь *тип* может быть любым, кроме ссылки или битового поля, причем *тип* может быть к этому моменту только декларирован, но еще не определен (следовательно, в структуре, например, может присутствовать указатель на структуру того же типа).

Символ *«звездочка»* относится непосредственно к *ID* указателя, поэтому для того, чтобы декларировать несколько указателей, ее нужно записывать перед именем каждого из них.

Например, в декларации:

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

определены два указателя на участки памяти для целочисленных данных, а также обычная целочисленная переменная *c*.

Значение указателя равно первому байту участка памяти, на который он ссылается.

Указатели предназначены для хранения адресов областей памяти. В языке Си имеются три вида указателей – указатели на объект известного типа, указатель типа *void* и указатель на функцию. Эти три вида различаются как своими свойствами, так и набором допустимых операций. Указатель не является самостоятельным типом данных, так как всегда связан с каким-либо конкретным типом, т.е. указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа.

Указатель типа ***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 типа *void* можно присвоить значение указателя любого типа, а также сравнивать его с любыми другими указателями, но перед выполнением каких-либо действий с участком памяти, на которую он ссылается, требуется явно преобразовать его к конкретному типу.

Указатель может быть константой или переменной, а также указывать на константу или переменную.

С указателями-переменными связаны две унарные операции **&** и *****.

Операция **&** означает «***взять адрес***» операнда. Операция ***** имеет смысл – «***значение, расположенное по указанному адресу***» (операция разадресации).

Таким образом, обращение к объектам любого типа как операндам операций в языке Си может производиться:

– по имени (идентификатору);

– по указателю (операция косвенной адресации):

*ID*_указателя=&*ID*_объекта; – операция разыменования;

**ID*_указателя – операция косвенной адресации.

Говорят, что использование указателя означает отказ от именования адресуемого им объекта.

Операция разадресации, или разыменования, предназначена для доступа к величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины (если она не объявлена как константа).

Унарная операция получения адреса & применима к переменным, имеющим имя (*ID*), для которых выделены участки оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной (типа *register*).

Отказ от именования объектов при наличии возможности доступа по указателю приближает язык Си по гибкости отображения «объект – память» к языку ассемблера.

***Пример* 1**:

```
int x,      – переменная типа int ;
*y;         – указатель на объект типа int;
y = &x;     – y – адрес переменной x;
*y=1;       – косвенная адресация указателем поля x, т.е. по
              указанному адресу записать 1: x = 1.
```

***Пример* 2**:

```
int i, j = 8, k = 5, *y;
y=&i;
*y=2;       – i = 2
y=&j;
*y+=i;      – j += i  → j = j+i  → j = j + 2 = 10
y=&k;
k+=*y;      – k += k → k = k + k = 10
(*y)++;     – k++  → k = k + 1 = 10 + 1 = 11
```

Как видно из приведенных примеров, конструкцию **ID_указателя* можно использовать в левой части оператора присваивания, так как она является *L*-значением (см. разд. 4.3), т.е. определяет адрес участка памяти. Эту конструкцию часто считают именем переменной, на которую ссылается указатель. С ней допустимы все действия, определенные для величин соответствующего типа (если указатель инициализирован).

***Пример* 3**:

```
int i1;              – целая переменная;
const int i2=1;      – целая константа;
int * pi1;           – указатель на целую переменную;
```

const int * pi2; – указатель на целую константу;
int * const pi1=&i1; – указатель-константа на целую переменную;
const int * const pi2=&i2; – указатель-константа на целую константу.

Как видно из примеров, модификатор *const*, находящийся между *ID* указателя и символом «*звездочка*», относится к самому указателю и запрещает его изменение, а *const* слева от звездочки задает константное значение объекта, на который он указывает. Для инициализации указателей использована операция получения адреса &.

Указатель подчиняется общим правилам определения области действия, видимости и времени жизни.

### *9.2. Операция sizeof*

Формат записи:

***sizeof*** ( *параметр* );

*параметр* – тип или идентификатор объекта (но не *ID* функции).

Данная операция позволяет определить размер указанного параметра в байтах (тип результата *int*).

Если указан идентификатор сложного объекта (массив, структура, объединение), то результатом будет размер всего объекта. Например:

sizeof(int) – результат 2(4) байта;
double b[5];
sizeof(b) – результат 8 байт * 5 = 40 байт.

Динамическое распределение оперативной памяти (ОП) связано с операциями порождения и уничтожения объектов по запросу программы, при котором захват и освобождение памяти производится программно, т.е. в процессе работы программы. При этом в языке Си порождение объектов (захват памяти) и уничтожение объектов (освобождение памяти) выполняются при помощи библиотечных функций.

### *9.3. Инициализация указателей*

При декларации указателя желательно выполнить его инициализацию, т.е. присвоение начального значения. Наиболее распространенная из ошибок в программах – непреднамеренное использование неинициализированных указателей.

Инициализатор записывается после *ID* указателя либо в круглых скобках, либо после знака равенства.

Существуют следующие способы инициализации указателя:

1. Присваивание указателю адреса существующего объекта:

а) используя операцию получения адреса переменной:

int ***a*** = 5;
int *p = &*a*; – указателю ***p*** присвоили адрес объекта *a*;

```
int *p(&a);          – то же самое другим способом;
```

б) с помощью значения другого инициализированного указателя:

```
int *g = p;
```

Указателю-переменной можно присвоить значение другого указателя либо выражения типа указатель с использованием при необходимости операции приведения типа. Приведение типа необязательно, если один из указателей имеет тип *void* *, например

```
int i,*x;
char *y;
x = &i;                    // x – поле объекта int;
y = (char *)x;             // y – поле объекта char;
y = (char *)&i;            // y – поле объекта char;
```

в) с помощью идентификаторов массива или функции, которые трактуются как адрес начала участка памяти, в котором размещается указанный объект. Причем следует учитывать тот факт, что *ID* массивов и функций являются константными указателями. Такую константу можно присвоить переменной типа указатель, но нельзя подвергать преобразованиям, например:

```
int x[100], *y;
y = x;        – присваивание константы переменной;
x = y;        – ошибка, т.к. в левой части указатель-константа.
```

2. Присваивание пустого значения:

```
int *x1 = NULL;
int *x2 = 0;
```

В первой строке используется константа *NULL*, определенная как указатель, равный нулю. Рекомендуется использовать просто цифру 0, так как это значение типа *int* будет правильно преобразовано стандартными способами в соответствии с контекстом. А так как объекта с нулевым (фиктивным) адресом не существует, пустой указатель обычно используют для контроля, ссылается указатель на конкретный объект или нет.

3. Присваивание указателю адреса выделенного участка динамической памяти:

а) с помощью операции *new* (см. разд. 16.4):

```
int *n = new int;
int *m = new int (10);
```

б) с помощью функции *malloc* (см. разд. 10.9):

```
int *p = (int*)malloc(sizeof(int));
```

Присваивание без явного приведения типов допускается в двух случаях:

– указателям типа *void**;

– если тип указателей справа и слева от операции присваивания один и тот же.

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, на которую указывала сама динамическая переменная, не освобождается. Такая ситуация называется «замусоривание оперативной памяти». Еще одна причина появления «мусора» – когда инициализированному указателю присваивается значение другого указателя. При этом старое значение указателя теряется.

### *9.4. Операции над указателями*

Помимо уже рассмотренных операций, с указателями можно выполнять арифметические операции сложения, инкремента (++), вычитания, декремента (--) и операции сравнения.

Арифметические операции с указателями 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.

Указатель, таким образом, может использоваться в выражениях вида

$$\boldsymbol{p} \# \boldsymbol{iv}, \quad \#\# \boldsymbol{p}, \quad \boldsymbol{p} \#\#, \quad \boldsymbol{p} \# = \boldsymbol{iv},$$

*p* – указатель, *iv* – целочисленное выражение, # – символ операции '+' или '–'.

Результатом таких выражений является увеличенное или уменьшенное значение указателя на величину *iv* * *sizeof*(**p*), т.е. если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа.

Текущее значение указателя всегда ссылается на позицию некоторого объекта в памяти с учетом правил выравнивания для соответствующего типа данных. Таким образом, значение *p # iv* указывает на объект того же типа, расположенный в памяти со смещением на *iv* позиций.

При сравнении указателей могут использоваться отношения любого вида («>», «<» и т.д.), но наиболее важными видами проверок являются отношения равенства и неравенства («==», «!=»).

Отношения порядка имеют смысл только для указателей на последовательно размещенные объекты (элементы одного массива).

Разность двух указателей дает число объектов адресуемого ими типа в соответствующем диапазоне адресов, т.е. в применении к массивам разность указателей, например, на третий и шестой элементы равна 3.

Очевидно, что уменьшаемый и вычитаемый указатели должны принадлежать одному массиву, иначе результат операции не имеет практической ценности и может привести к непредсказуемому результату. То же можно сказать и о суммировании указателей.

Значение указателя можно вывести на экран с помощью функции *printf*, используя спецификацию %***p*** (*pointer*), результат выводится в шестнадцатеричном виде.

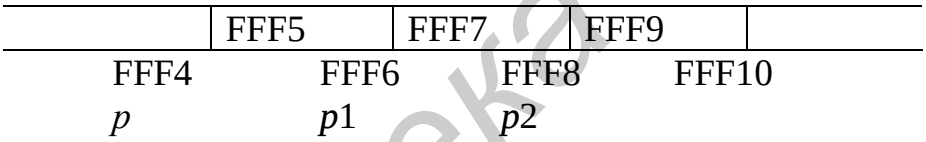
Рассмотрим фрагмент программы:

```
int a = 5, *p, *p1, *p2;
p = &a;
p2 = p1 = p;
++p1;
p2 += 2;
printf("a = %d , p = %d , p = %p , p1 = %p , p2 = %p .\n", a, *p, p, p1, p2);
```

Результат может быть следующим:

*a = 5 , *p = 5 , p = FFF4 , p1 = FFF6, p2 = FFF8 .*

Графически это выглядит следующим образом (в 16-разрядном процессоре на тип *int* отводится 2 байта):




*p* = FFF4,

*p*1 = FFF6 = ( FFF4 + 1*sizeof(*p)) → FFF4 + 2 (*int*)

*p*2 = FFF8 = ( FFF4 + 2*sizeof(*p)) → FFF4 + 2*2

На одну и ту же область памяти (как видно из приведенного примера), может ссылаться несколько указателей различного типа. Но примененная к ним операция разадресации даст разные результаты.

При смешивании в выражении указателей разных типов явное преобразование типов требуется для всех указателей, кроме *void**.

Явное приведение типов указателей позволяет получить адрес объекта любого типа:

```
type *p;
p = (type*) &object;
```

Значение указателя *p* позволяет работать с переменной *object* как объектом типа *type*.

# ГЛАВА 10. Массивы

## *10.1. Понятие массива*

В математике для удобства записи различных операций часто используют индексированные переменные: векторы, матрицы и т.п. Так, вектор $\vec{c}$ представляется набором чисел ($c_1$, $c_2$, ..., $c_n$), называемых его компонентами, причем каждая компонента имеет свой номер, который принято обозначать в виде индекса. Матрица $A$ – это таблица чисел ($a_{ij}$, $i$=1,..., $n$; $j$=1,..., $m$), $i$ – номер строки, $j$ – номер столбца. Операции над матрицами и векторами обычно имеют короткую запись, которая обозначает определенные, порой сложные действия над их индексными компонентами. Например, произведение двух векторов записывается как $\vec{c} \cdot \vec{b} = \sum_{i=1}^{n} c_i b_i$.

Произведение матрицы на вектор $\vec{b} = A \cdot \vec{c}, \quad b_i = \sum_{j=1}^{n} a_{ij} \cdot c_j$.

Таким образом, если с группой величин одинакового типа требуется выполнять однообразные действия, им дают одно имя, а различают по порядковому номеру.

Введение индексированных переменных в языках программирования также позволяет значительно облегчить реализацию многих сложных алгоритмов, связанных с обработкой массивов однотипных данных.

Например, использование массивов данных позволяет компактно записывать множество операций с помощью циклов.

В языке Си для этой цели используется сложный тип данных – ***массив***, представляющий собой упорядоченную конечную совокупность элементов одного типа. Число элементов массива называют его *размером*. Каждый элемент массива определяется идентификатором массива и своим порядковым номером – *индексом*. ***Индекс*** – целое число, по которому производится доступ к элементу массива. Индексов может быть несколько. В этом случае массив называют многомерным, а количество индексов одного элемента массива является его размерностью.

Описание массива в программе отличается от описания простой переменной наличием после имени квадратных скобок, в которых задается количество элементов массива. Например, *double a* [10]; – описание массива из 10 вещественных чисел.

При описании массивов квадратные скобки являются элементом синтаксиса, а не указанием на необязательность конструкции.

Размеры массивов предпочтительнее вводить с клавиатуры как значения целочисленных переменных или задавать с помощью именованных констант, поскольку при таком подходе для ее изменения достаточно скорректировать значение константы всего лишь в одном месте программы.

### *10.2. Одномерные массивы*

В программе одномерный массив объявляется следующим образом:

***тип ID_массива*** [***размер***] = {список начальных значений};

*тип* – базовый тип элементов массива (целый, вещественный, символьный); *размер* – количество элементов в массиве.

Список начальных значений используется при необходимости инициализировать данные при объявлении, он может отсутствовать.

При декларации массива можно использовать также атрибуты «класс памяти» и *const*.

Размер массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется на этапе компиляции, поэтому размер массива задается только константой или константным выражением. Нельзя задавать массив переменного размера, для этого существует отдельный механизм – динамическое выделение памяти.

Пример объявления массива целого типа: int ***a***[5];

***Индексы массивов в языке Си начинаются с 0***, т.е. в массиве *a* первый элемент: *a*[0], второй – *a*[1], … пятый – *a*[4].

Обращение к элементу массива в программе на языке Си осуществляется в традиционном для многих других языков стиле – записи операции обращения по индексу [] (квадратные скобки), например:

```
a[0]=1;
a[i]++;
a[3]=a[i]+a[i+1];
```

Пример объявления массива целого типа с инициализацией начальных значений:

```
int a[5]={2, 4, 6, 8, 10};
```

Если в группе {…} список значений короче, то оставшимся элементам присваивается 0.

***Внимание***. В языке Си с целью повышения быстродействия программы отсутствует механизм контроля выхода за границы индексов массивов. При необходимости такой механизм должен быть запрограммирован явно.

### *10.3. Связь указателей и массивов*

Идентификатор одномерного массива – это адрес памяти, начиная с которого он расположен, т.е. адрес его первого элемента. Таким образом, работа с массивами тесно взаимосвязана с применением указателей. Рассмотрим связь указателей с элементами одномерного массива на примере.

Пусть объявлены одномерный целочисленный массив ***a*** из 5 элементов и указатель ***p*** на целочисленные переменные:

```
int a[5]={1, 2, 3, 4, 5}, *p;
```

*ID* массива ***a*** является константным указателем на его начало, т.е. *a* = &*a*[0] – адрес начала массива. Расположение массива *a* в оперативной памяти, выделенной компилятором, может выглядеть следующим образом:

| *a*[0] | *a*[1] | *a*[2] | *a*[3] | *a*[4] | – элементы массива; |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | – значения элементов массива; |
| 4000 | 4002 | 4004 | 4006 | 4008 | – символические адреса. |

Указатель *a* содержит адрес начала массива и в нашем примере равен 4000 (*a* = = 4000).

Если установить указатель *p* на объект *a*, т.е. присвоить переменной-указателю адрес первого элемента массива:

*p* = *a*;

что эквивалентно выражению *p* = &*a*[0]; то получим, что и *p* = 4000. Тогда с учетом адресной арифметики обращение к *i*-му элементу массива ***a*** может быть записано следующими выражениями:

*a*[*i*] ~ *(*a*+*i*) ~ *(*p*+*i*) ~ *p*[*i*] ,

приводящими к одинаковому результату.

Идентификаторы *a* и *p* – указатели, очевидно, что выражения *a*[*i*] и *(*a*+*i*) эквивалентны. Отсюда следует, что операция обращения к элементу массива по индексу применима и при его именовании переменной-указателем. Таким образом, для любых указателей можно использовать две эквивалентные формы выражений для доступа к элементам массива: *p*[*i*] и *(*p*+*i*). Первая форма удобнее для читаемости текста, вторая – эффективнее по быстродействию программы.

Например, для получения значения 4-го элемента массива можно написать *a*[3] или *(*a*+3), результат будет один и тот же, а операции ***a***[3] = 8 и *(*p*+3) = 8 дадут тождественный результат, т.к. *p*+3 = 4000+3**sizeof*(*int*) = = 4000+3*2 = 4006, т.е. указатель *p* установлен на четвертый по счету элемент массива.

Очевидна и эквивалентность выражений:

– получение адреса начала массива в ОП:

&*a*[0] ↔ &(**a*) ↔ *a*

– обращение к первому элементу массива:

**a* ↔ *a*[0]

Последнее утверждение объясняет правильность выражения для получения количества элементов массива:

```
int x[]={1, 2, 3, 4, 5, 6, 7};
```

Размер ***n*** должен быть целочисленной константой:

```
int n = sizeof(x) / sizeof(*x);
```

### *10.4. Строки как одномерные массивы данных типа char*

В языке Си отдельного типа данных «строка символов» нет. Работа со строками реализована путем использования одномерных массивов типа *char*, т.е. строка символов – это одномерный массив символов, заканчивающийся нулевым байтом.

Нулевой байт – это байт, каждый бит которого равен нулю, при этом для нулевого байта определена символьная константа ´\0´ (признак окончания строки, или «*нуль-символ*»). Поэтому если строка должна содержать *k* символов, то в описании массива размер должен быть *k*+1. По положению нуль-символа определяется фактическая длина строки.

Например, char *s*[7]; – означает, что строка может содержать не более шести символов, а последний байт отводится под нуль-символ.

Отсутствие нуль-символа и выход указателя при просмотре строки за ее пределы – распространенная ошибка при работе со строками.

Строку можно инициализировать строковой константой (строковым литералом), которая представляет собой набор символов, заключенных в двойные кавычки. Например:

```
char S[ ] = “Работа со строками”;
```

для данной строки выделено и заполнено 19 байт – 18 на символы и 19-й на нуль-символ.

В конце строковой константы явно указывать символ ´\0´ не нужно. Компилятор добавит его автоматически.

Символ ´\0´ нужно использовать явно тогда, когда символьный массив при декларации инициализируется списком начальных значений, например, следующим образом:

```
char str[10] ={‘V’ , ‘a’, ‘s’, ‘j’ , ‘a’, ‘\0’};
```

или когда строка формируется посимвольно в коде программы. Пример такого формирования приведен в конце этого раздела.

При работе со строками можно пользоваться указателями, например:

```
char *x;
x = "БГУИР";
x = (i>0) ? "положительное" : (i<0) ? "отрицательное" : "нулевое";
```

Такая декларация строки – единственный случай, когда в коде программы можно использовать операцию присваивания явно.

Операция char *str = "БГУИР" создает не строковую переменную, а ***указатель на строковую константу, изменить которую невозможно***, причем это касается не только адреса ОП, но и его размера. Знак равенства перед строковым литералом означает инициализацию, а не присваивание.

Операция присваивания одной строки другой в языке Си не определена (поскольку строка является массивом) и может обрабатываться при помощи оператора цикла (с использованием стандартной библиотечной функций).

Процесс копирования строки *s*1 в строку *s*2 имеет вид

```
char s1[25], s2[25];
for (int i = 0; i <= strlen(s1); i++)
        s2[i] = s1[i];
```

Длина строки определяется с помощью стандартной функции ***strlen***, которая вычисляет длину, выполняя поиск нуль-символа (прототип функции приведен ниже). Таким образом, строка фактически просматривается дважды.

А вот следующие действия будут ошибочными:

```
char s1[51];
        s1 = ”Minsk”;
```

Это связано с тем, что *s*1 – константный указатель и не может использоваться в левой части операции присваивания.

Большинство действий со строковыми объектами в Си выполняются при помощи стандартных библиотечных функций, так, для правильного выполнения операции присваивания в последнем примере необходимо использовать стандартную функцию

```
        strcpy(s1, ”Minsk”);
```

Напомним, что для ввода строк, как и для других объектов программы, обычно используются две стандартные функции:

Функция ***scanf*** вводит значения для строковых переменных при помощи формата (спецификатора ввода) %*s* до появления первого символа “пробел” (символ «&» перед *ID* строковых данных указывать не надо);

Функция ***gets*** осуществляет ввод строки, которая может содержать пробелы. Завершается ввод нажатием клавиши ***Enter***.

Обе функции автоматически ставят в конец строки нулевой байт.

Вывод строк производится функциями ***printf*** или ***puts*** до нулевого байта.

Функция ***printf*** не переводит курсор после вывода на начало новой строки, а ***puts*** автоматически переводит курсор после вывода строковой информации в начало новой строки. Например:

```
char Str[30];
printf(“ Введите строку без пробелов : \n”);
scanf(“%s”, Str);
```

или

```
puts(“ Введите строку ”);
gets(Str);
```

Остальные операции над строками, как уже отмечалось ранее, выполняются с использованием стандартных библиотечных функций, декларация прототипов которых находятся в файле ***string.h***.

Приведем наиболее часто используемые стандартные строковые функции.

Функция ***strlen***(*S*) возвращает длину строки (количество символов в строке), при этом завершающий нулевой байт не учитывается, например:

```
char *S1 = ”Минск!\0”, S2[] = ”БГУИР–Ура!”;
printf(“ %d , %d .”, strlen(S1), strlen(S2));
```

Результат выполнения данного участка программы:

*6 , 10 .*

Функция ***strcpy***(*S*1, *S*2) – копирует содержимое строки *S*2 в строку *S*1.

Функция ***strcat***(*S*1, *S*2) – присоединяет строку *S*2 к строке *S*1 и помещает ее в массив, где находилась строка *S*1, при этом строка *S*2 не изменяется. Нулевой байт, который завершал строку *S*1, заменяется первым символом строки S2.

Функция *int* ***strcmp***(*S*1, *S*2) сравнивает строки *S*1 и *S*2 и возвращает значение <0, если *S*1<*S*2; >0, если *S*1>*S*2; =0, если строки равны, т.е. содержат одно и то же число одинаковых символов.

Функции преобразования строковых объектов в числовые описаны в библиотеке *stdlib.h*. Рассмотрим некоторые из них.

Преобразование строки *S* в число:

– целое: *int* ***atoi***(S);
– длинное целое: *long* ***atol***(S);
– действительное: *double* ***atof***(S);

при возникновении ошибки данные функции возвращают значение 0.

Функции преобразования числа *V* в строку *S*:

– целое: ***itoa***(V, S, kod);
– длинное целое: ***ltoa***(V, S, kod);

$2 \le$ *kod* $\le 36$, для десятичных чисел со знаком *kod* = 10.

Пример участка кода программы, в котором из строки *s* удаляется символ, значение которого содержится в переменной *c* каждый раз, когда он встречается

```
char s[81], c;
    ...
for( i = j = 0; s[i] != '\0'; i++)
    if( s[i] != c)  s[j++] = s[i];
s[j]='\0';
    ...
```

---

В режиме консольных приложений в среде *Visual C++* 6.0 вывод символов русского языка сопряжен с определенными неудобствами. Разрешение данной проблемы рассматривается в разд. 16.3.

---

### *10.5. Указатели на указатели*

Указатели, как и переменные любого другого типа, могут объединяться в массивы.

Объявление ***массива указателей*** на целые числа имеет вид

```
int *a[10], y;
```

Теперь каждому из элементов массива указателей *a* можно присвоить адрес целочисленной переменной *y*, например: *a*[1]=&*y*;

Чтобы теперь найти значение переменной *y* через данный элемент массива *a*, необходимо записать **a*[1].

В языке Си можно описать переменную типа «***указатель на указатель***». Это ячейка оперативной памяти (переменная), в которой будет храниться адрес указателя на некоторую переменную. Признак такого типа данных – повторение символа «*» перед идентификатором переменной. Количество символов «*» определяет уровень вложенности указателей друг в друга. При объявлении указателей на указатели возможна их одновременная инициализация. Например:

```
int a=5;
int *p1=&a;
int **pp1=&p1;
int ***ppp1=&pp1;
```

Если присвоить переменной *a* новое значение, например 10, то одинаковые результаты будут получены в следующих операциях:

```
a=10;    *p1=10;   **pp1=10;   ***ppp1=10;
```

Для доступа к области ОП, отведенной под переменную *a*, можно использовать и индексы. Эквивалентны следующие выражения:

```
*p1      ~    p1[0] ;
**pp1    ~    pp1[0][0] ;
***ppp1  ~    ppp1[0][0][0] .
```

Фактически, используя указатели на указатели, мы имеем дело с многомерными массивами.

### *10.6. Многомерные массивы*

Декларация многомерного массива имеет следующий формат:

*тип ID*[*размер*1][*размер*2]…[*размерN*] =
{ {*список начальных значений*},
{*список начальных значений*},
…
};

Списки начальных значений – атрибут необязательный.

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке Си размещаются в памяти компьютера построчно друг за другом (см. следующую тему «Адресная функция»).

Рассмотрим особенности работы с многомерными массивами на конкретном примере двухмерного массива.

Например, пусть приведена следующая декларация двухмерного массива:

*int m*[3][4];

Идентификатор двухмерного массива – это указатель на массив указателей (переменная типа указатель на указатель: *int* ***m*;).

Поэтому двухмерный массив *m*[3][4]; компилятор рассматривает как массив трех указателей, каждый из которых указывает на начало массива со значениями размером по четыре элемента каждый. В ОП данный массив будет расположен следующим образом:

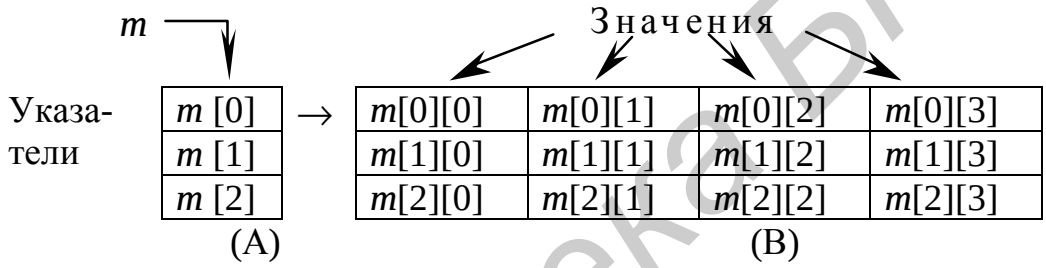



Рис. 10.1. Схема размещения элементов массива *m* размером 3×4

Причем в данном случае указатель *m*[1] будет иметь адрес *m*[0]+4**sizeof*(*int*), т.е. каждый первый элемент следующей строки располагается за последним элементом предыдущей строки.

Приведем пример программы конструирования массива массивов:

```
#include <stdio.h>
void main()
{
    int x0[4] = { 1, 2, 3,4};              //  Декларация и инициализация
    int x1[4] = {11,12,13,14};             //  одномерных массивов
    int x2[4] = {21,22,23,24};
    int *m[3] = {x0, x1, x2,};             // Создание массива указателей
    int i,j;
    for (i=0; i<3; i++) {
        printf("\n Строка %d) ", i+1);
        for (j=0; j<4; j++)
            printf("%3d", m[ i ] [ j ]);
        }
    }
```

Результаты работы программы:

*Строка 1) 1 2 3 4*
*Строка 2) 11 12 13 14*
*Строка 3) 21 22 23 24*

Такие же результаты будут получены и в следующей программе:

```
#include <stdio.h>
void main()
{
	int i, j;
	int m[3][4] = { { 1, 2, 3, 4}, {11,12,13,14}, {21,22,23,24} };
	for (i=0; i<3; i++) {
		printf("\n %2d)", i+1);
		for (j=0; j<4; j++)
			printf(" %3d",m[ i ] [ j ]);
	}
}
```

В последней программе массив указателей на соответствующие массивы элементов создается компилятором автоматически, т.е. данные массива располагаются в памяти последовательно по строкам, что является основанием для декларации массива *m* в виде

```
int m[3][4] = {1, 2, 3, 4, 11, 12, 13, 14, 21, 22, 23, 24};
```

Замена скобочного выражения *m*[3][4] на *m*[12] здесь не допускается, так как массив указателей не будет создан.

Таким образом, использование многомерных массивов в языке Си связано с расходами памяти на создание массивов указателей.

Очевидна и схема размещения такого массива в памяти – последовательное (друг за другом) размещение «строк» – одномерных массивов со значениями (векторная организация памяти).

Обращению к элементам массива при помощи операции индексации *m*[*i*][*j*] соответствует эквивалентное выражение, использующее адресную арифметику – *(*(*m*+*i*)+*j*).

Аналогичным образом можно установить соответствие между указателями и массивами с произвольным числом измерений.

### *10.7. Адресная функция*

Векторная память поддерживается почти всеми языками высокого уровня и предназначена для хранения массивов различной размерности и различных размеров. Каждому массиву выделяется непрерывный участок памяти указанного размера. При этом элементы, например, двухмерного массива *X* размерностью *n*1×*n*2 размещаются в ОП в следующей последовательности:

$X(0,0)$, $X(0,1)$, $X(0,2)$,... $X(0, n2–1)$, ..., $X(1,0)$, $X(1,1)$, $X(1,2)$,... $X(1, n2–1)$, ..., $X(n1–1,0)$, $X(n1–1,1)$, $X(n1–1,2)$, ..., $X(n1–1, n2–1)$.

Адресация элементов массива определяется некоторой адресной функцией, связывающей адрес и индексы элемента.

Пример адресной функции для массива $X$:

$$K(i, j) = n2*i + j;$$

где $i = 0,1,2,...,(n1–1)$; $j = 0,1,2,...,(n2–1)$; $j$ – изменяется в первую очередь.

Адресная функция двухмерного массива $A(n,m)$ будет выглядеть так:

$$N1 = K(i, j) = m*i + j,$$

$i$=0,1,..., $n$–1; $j$=0,1,... , $m$–1 .

Тогда справедливо следующее:

$$A(i, j) \leftrightarrow B(K(i, j)) = B(N1),$$

$B$ – одномерный массив с размером $N1 = n*m$.

Например, для двухмерного массива $A(2,3)$ имеем:

| (0,0) | (0,1) | (0,2) | (1,0) | (1,1) | (1,2) | – индексы массива A; |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | – индексы массива B. |

Проведем расчеты:

$i = 0, j = 0$ $N1 = 3*0+0 = 0$ $B(0)$
$i = 0, j = 1$ $N1 = 3*0+1 = 1$ $B(1)$
$i = 0, j = 2$ $N1 = 3*0+2 = 2$ $B(2)$
$i = 1, j = 0$ $N1 = 3*1+0 = 3$ $B(3)$
$i = 1, j = 1$ $N1 = 3*1+1 = 4$ $B(4)$
$i = 1, j = 2$ $N1 = 3*1+2 = 5$ $B(5)$

Аналогично получаем адресную функцию для трехмерного массива $X(n1, n2, n3)$:

$$K(i, j, k) = n3*n2*i + n2*j + k ,$$

где $i = 0,1,2,...,(n1–1)$; $j = 0,1,2,...,(n2–1)$; ); $k = 0,1,2,...,(n3–1)$; значение $k$ – изменяется в первую очередь.

Для размещения такого массива потребуется участок ОП размером $(n1*n2*n3)*sizeof(type)$. Рассматривая такую область как одномерный массив $Y(0,1,..., n1*n2*n3)$, можно установить соответствие между элементом трехмерного массива $X$ и элементом одномерного массива $Y$:

$$X(i, j, k) \leftrightarrow Y(K(i, j, k)) .$$

Необходимость введения адресных функций возникает лишь в случаях, когда требуется изменить способ отображения с учетом особенностей конкретной задачи.

### *10.8. Работа с динамической памятью*

Указатели чаще всего используют при работе с динамической памятью, которую иногда называют «***куча***» (перевод английского слова *heap*). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти производится только через указатели. Время жизни динамических объектов – от точки создания до конца программы или до явного освобождения памяти.

С другой стороны, некоторые задачи исключают использование структур данных фиксированного размера и требуют введения структур динамических, способных увеличивать или уменьшать свой размер уже в процессе работы программы. Основу таких структур составляют динамические переменные.

Динамическая переменная хранится в некоторой области ОП, не обозначенной именем, и обращение к ней производится через переменную-указатель.

Но вначале рассмотрим еще одну операцию языка Си, основное назначение которой – работа с участками оперативной памяти.

### *10.9. Библиотечные функции*

Функции для манипулирования динамической памятью в стандарте Си следующие:

*void* ****calloc***(*unsigned* ***n***, *unsigned* ***size***); – выделение памяти для размещения ***n*** объектов размером ***size*** байт и заполнение полученной области нулями; возвращает указатель на выделенную область памяти;

*void* ****malloc*** (*unsigned* ***n***) – выделение области памяти для размещения блока размером ***n*** байт; возвращает указатель на выделенную область памяти;

*void* ****realloc*** (*void* ****b***, *unsigned* ***n***) – изменение размера размещенного по адресу ***b*** блока на новое значение ***n*** и копирование (при необходимости) содержимого блока; возвращает указатель на перераспределенную область памяти; при возникновении ошибки, например, нехватке памяти, эти функции возвращают значение ***NULL***, что означает отсутствие адреса (нулевой адрес);

***coreleft*** (*void*) – получение размера свободной памяти в байтах только для *MS DOS* (используется в *Borland* C++), тип результата: *unsigned* – для моделей памяти *tiny*, *small* и *medium*; *unsigned long* – для моделей памяти *compact*, *large* и *huge*;

*void* ***free*** (*void* ****b***) – освобождение блока памяти, адресуемого указателем ***b***.

Для использования этих функций требуется подключить к программе в зависимости от среды программирования заголовочный файл ***alloc.h*** или ***malloc.h***.

В языке C++ введены операции захвата и освобождения памяти *new* и *delete*, рассматриваемые в разд. 16.4.

### *10.10. Пример создания одномерного динамического массива*

В языке Си размерность массива при объявлении должна задаваться константным выражением.

Если до выполнения программы неизвестно, сколько понадобится элементов массива, нужно использовать динамические массивы, т.е. при необходимости работы с массивами переменной размерности вместо массива достаточно объявить указатель требуемого типа и присвоить ему адрес свободной области памяти (захватить память).

Память под такие массивы выделяется с помощью функций ***malloc*** и ***calloc*** (операцией ***new*)** во время выполнения программы. Адрес начала массива хранится в переменной-указателе. Например:

```
int n = 10;
double *b = (double *) malloc(n * sizeof (double));
```

В примере значение переменной *n* задано, но может быть получено и программным путем.

Обнуления памяти при ее выделении не происходит. Инициализировать динамический массив при декларации нельзя.

Обращение к элементу динамического массива осуществляется так же, как и к элементу обычного – например *a*[3]. Можно обратиться к элементу массива и через косвенную адресацию – *(*a* + 3). В любом случае происходят те же действия, которые выполняются при обращении к элементу массива, декларированного обычным образом.

После работы захваченную под динамический массив память необходимо освободить, для нашего примера *free*(*b*);

Таким образом, время жизни динамического массива, как и любой динамической переменной – с момента выделения памяти до момента ее освобождения. Область действия элементов массива зависит от места декларации указателя, через который производится работа с его элементами. Область действия и время жизни указателей подчиняются общим правилам для остальных объектов программы.

Пример работы с динамическим массивом:

```
#include <alloc.h>
void main()
{
    double *x;
    int n;
    printf("\nВведите размер массива – ");
```

```
    scanf("%d", &n);
    if ((x = (double*)calloc(n, sizeof(*x)))==NULL)  {        // Захват памяти
        puts("Ошибка ");
        return;
    }
        ...
// Работа с элементами массива
        ...
    free(x);                                                  // Освобождение памяти
}
```

---

Примеры создания одномерного и двухмерного динамических массивов с использованием операций ***new*** и ***delete*** можно посмотреть в разд. 16.4.

---

### *10.11. Пример создания двухмерного динамического массива*

Напомним, что *ID* двухмерного массива – указатель на указатель. На рис. 10.1 приведена схема расположения элементов, причем в данном случае сначала выделяется память на указатели, расположенные последовательно друг за другом, а затем каждому из них выделяется соответствующий участок памяти под элементы.

```
        . . .
    int  **m, n1, n2, i, j;
    puts(" Введите размеры массива (строк, столбцов): ");
    scanf("%d%d", &n1, &n2);
// Захват памяти для указателей – A (n1=3)
    m = (int**)calloc(n1, sizeof(int*));
    for (i=0; i<n1; i++)                  // Захват памяти для элементов – B (n2=4)
        *(m+i) = (int*)calloc(n2, sizeof(int));
    for ( i=0; i<n1; i++)
        for ( j=0; j<n2; j++)
                    m[i] [j] = i+j;               //  *(*(m+i)+j) = i+j;
        . . .
    for(i=0; i<n; i++) free(m[i]);                // Освобождение памяти
    free(m);
        . . .
```

# ГЛАВА 11. Функции пользователя

С увеличением объема программы ее код становится все более сложным. Одним из способов борьбы со сложностью любой задачи является ее разбиение на части. В языке Си, как и в любом языке программирования высокого уровня, задача может быть разбита на более простые подзадачи при помощи подпрограмм-функций. После этого программу можно рассматривать в более укрупненном виде – на уровне взаимодействия созданных подпрограмм. Использование подпрограмм в коде программы и ведет к упрощению ее структуры.

Разделение программы на подзадачи позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Кроме того, упрощается процесс отладки программы, содержащей подпрограммы. Часто используемые функции можно помещать в отдельные библиотеки.

Следующим шагом в повышении уровня абстракции программы является группировка функций и связанных с ними данных в отдельные файлы (модули), компилируемые раздельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля, что позволяет отлаживать программу по частям.

Для того чтобы использовать модуль, достаточно знать только его интерфейс, а не все детали его реализации.

Разделение программы на максимально обособленные части (подпрограммы) является довольно сложной задачей, которая должна решаться на этапе проектирования программы.

В отличие от других языков программирования высокого уровня в языке Си нет разделения на подпрограммы-процедуры и подпрограммы-функции, здесь вся программа строится только из функций.

В языке Си любая подпрограмма является функцией, представляющей собой отдельный программный модуль, к которому можно обратиться, чтобы передать через параметры исходные данные и получить один или несколько результатов его работы.

Функция – это именованная последовательность инструкций, выполняющая какое-либо законченное действие.

Таким образом, любая программа на языке Си состоит из функций. Минимальная программа на Си содержит, как уже известно, единственную функцию *main* (основная, главная), с которой и начинается выполнение программы.

### *11.1. Декларация функции*

Как и любой объект программы на языке Си, пользовательские функции необходимо декларировать. Объявление функции пользователя, т.е. ее декларация, выполняется в двух формах – в форме описания (объявления) и в форме определения, т.е. любая пользовательская функция должна быть объявлена и определена.

Описанием функции является декларация ее прототипа, который сообщает компилятору о том, что далее будет приведено ее полное определение (текст), т.е. реализация.

Объявление функции (прототип, заголовок) задает ее свойства - идентификатор, тип возвращаемого значения (если такое имеется), количество и типы параметров.

В стандарте языка используется следующий формат декларации (объявления) функций:

***тип_результата ID_функции* (*список*);**

В *списке* перечисляются типы параметров данной функции, причем идентификаторы переменных в круглых скобках прототипа указывать необязательно, т.к. компилятор языка их не обрабатывает.

Описание прототипа дает возможность компилятору проверить соответствие типов и количества параметров при фактическом вызове этой функции.

Пример объявления функции *fun*, которая имеет три параметра типа *int*, один параметр типа *double* и возвращает результат типа *double*:

```
double fun(int, int, int, double);
```

Каждая функция, вызываемая в программе, должна быть определена (только один раз). ***Определение функции*** – это ее полный текст, включающий заголовок и код.

Полное определение (реализация) функции имеет следующий вид:

```
тип_результата ID_функции(список параметров)
{
        код функции
    return выражение;
}
```

Рассмотрим составные части определения пользовательской функции.

*Тип результата* определяет тип *выражения*, значение которого возвращается в точку ее вызова при помощи оператора ***return*** *выражение*; (возврат). *Выражение* преобразуется к *типу_результата*, указанному в заголовке функции и передается в точку вызова. Тип возвращаемого функцией значения может быть любым базовым типом, а также указателем на массив или функцию. Если функция не должна возвращать значение, указывается тип ***void***. В данном случае оператор *return* можно не ставить. Из функции, которая не описана как *void*, необходимо возвращать значение,

используя оператор *return*. Если тип функции не указан, то по умолчанию устанавливается тип *int*.

***Список параметров*** состоит из перечня типов и идентификаторов параметров, разделенных запятыми. Список параметров определяет объекты, которые требуется передать в функцию при ее вызове.

В определении и в объявлении одной и той же функции типы и порядок следования параметров должны совпадать. Тип возвращаемого значения и типы параметров совместно определяют тип функции.

Функция может не иметь параметров, но круглые скобки необходимы в любом случае. Если у функции отсутствует список параметров, то при декларации такой функции желательно в круглых скобках указать *void*. Например, *void main*(*void*){ ... }.

В функции может быть несколько операторов *return*, но может и не быть ни одного (тип *void* – это определяется потребностями алгоритма). В последнем случае возврат в вызывающую программу происходит после выполнения последнего оператора кода функции.

Пример функции, определяющей наименьшее значение из двух целочисленных переменных:

```
int  min (int x, int y)
{
        return (x<y) ? x : y;
}
```

Функции, возвращающие значение, желательно использовать в правой части выражений языка Си, иначе возвращаемый результат будет утерян.

В языке Си каждая функция – это отдельный блок программы, вход в который возможен только через вызов данной функции.

### *11.2. Вызов функции*

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечислить список передаваемых ей ***аргументов***. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция.

Простейший вызов функции имеет следующий формат:

***ID_функции*** (***список аргументов***);

где в качестве аргументов можно использовать константы, переменные, выражения (их значения перед вызовом функции будут определены компилятором).

Аргументы в списке вызова должны совпадать со списком параметров вызываемой функции по количеству и порядку следования, а типы аргументов при передаче в функцию будут преобразованы, если это возможно, к типу соответствующих им параметров.

Связь между функциями осуществляется через аргументы и возвращаемые функциями значения. Ее можно осуществить также через внешние, глобальные переменные (см. гл. 12).

Глобальные переменные доступны всем функциям, где они не описаны как локальные переменные. Использовать их для передачи данных между функциями довольно просто, но тем не менее этого делать не рекомендуется. Необходимо стремиться к тому, чтобы функции в программе были максимально независимыми и чтобы их интерфейс полностью определялся прототипами этих функций.

Функции могут располагаться в исходном файле в любом порядке, при этом исходная программа может размещаться в нескольких файлах.

Все величины, описанные внутри функции, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции, для того чтобы при выходе из нее можно было продолжить выполнение вызывающей функции. При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются. Если этого требуется избежать, при объявлении локальных переменных используется модификатор *static*, например:

```
#include <stdio.h>
        void f1(int);
void main(void)
{
        f1(5);
}
void f1(int i)
{
        int m=0;
        puts(" n  m  p ");
        while (i--) {
                static int n = 0;
                int p = 0;
                printf(" %d  %d  %d \n", n++ , m++ , p++);
        }
}
```

Статическая переменная *n* будет создана в сегменте данных ОП и проинициализируется нулем только один раз при первом выполнении оператора, содержащего ее определение, т.е. при первом вызове функции *f*1. Автоматическая переменная *m* инициализируется при каждом входе в функцию. Автоматическая переменная *p* инициализируется при каждом входе в блок цикла.

В результате выполнения программы получим

*n m p*
*0 0 0*
*1 1 0*
*2 2 0*
*3 3 0*
*4 4 0*

### *11.3. Передача аргументов в функцию*

В языке Си аргументы при стандартном вызове функции передаются по значению. Это означает, что в стеке, как и в случае локальных данных, выделяется место для формальных параметров функции. В выделенное место при вызове функции заносятся значения фактических аргументов, при этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение. Затем функция использует и может изменять эти значения в стеке.

При выходе из функции измененные значения теряются, т.к. время жизни и зона видимости локальных параметров определяется кодом функции. Вызванная функция не может изменить значения переменных, указанных как фактические аргументы при обращении к данной функции.

В случае необходимости функцию можно использовать для изменения передаваемых ей аргументов. В этом случае в качестве аргумента необходимо в вызываемую функцию передавать не значение переменной, а ее адрес.

При передаче по адресу в стек заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения аргументов. Для обращения к значению аргумента-оригинала используется операция «*».

Пример функции, в которой меняются местами значения *x* и *y*:

```
void  zam(int *x, int *y)
{
        int  t = *x;
        *x = *y;
        *y = t;
}
```

Участок программы с обращением к данной функции:

```
void zam (int*, int*);
void main (void)
{
        int a=2, b=3;
        printf(" a = %d , b = %d\n", a, b);
        zam (&a, &b);
```

```
    printf(" a = %d , b = %d\n", a, b);
}
```

При таком способе передачи данных в функцию их значения будут изменены, т.е. на экран монитора будет выведено

*a = 2 , b=3*
*a = 3 , b=2*

Если требуется запретить изменение значений какого-либо параметра внутри функции, то в его декларации используют атрибут *const*, например:

```
void f1(int, const double);
```

Рекомендуется указывать *const* перед всеми параметрами, изменение которых в функции не предусмотрено. Это облегчает, например, отладку программы, т.к. по заголовку функции видно, какие данные в функции изменяются, а какие нет.

### *11.4. Операция typedef*

Любому типу данных, как стандартному, так и определенному пользователем, можно задать новое имя с помощью операции *typedef*:

***typedef тип новое_имя ;***

Введенный таким образом новый тип используется аналогично стандартным типам, например, введя пользовательские типы:

typedef unsigned int ***UINT***; – здесь *UINT* – новое имя;
typedef char ***M_s*** [101]; – здесь *M_s* – тип пользователя, определяющий строки, длиной не более 100 символов.

Декларации объектов введенных типов будут иметь вид

**UINT** i, j; → две переменные типа *unsigned int* ;
**M_s** str[10]; → массив из 10 элементов, в каждом из которых можно хранить по 100 символов.

Рассмотренная операция упростит использование указателей на функции, которые рассматриваются в следующем разделе.

### *11.5. Указатели на функции*

В языке Си идентификатор функции является константным указателем на начало функции в оперативной памяти и не может быть значением переменной. Но имеется возможность декларировать указатели на функции, с которыми можно обращаться как с переменными (например, можно создать массив, элементами которого будут указатели на функции).

Рассмотрим методику работы с указателями на функции.

1. Как и любой объект языка Си, указатель на функции необходимо декларировать. Формат объявления указателя на функции следующий:

***тип*** (*****переменная-указатель***)(***список параметров***);

т.е. декларируется указатель, который можно устанавливать на функции, возвращающие результат указанного *типа* и которые имеют указанный *список параметров*. Наличие первых круглых скобок обязательно, так как без них – это декларация функции, которая возвращает указатель на результат.

Например, объявление вида

double (***p_f*** )(char, double);

говорит о том, что декларируется указатель *p_f*, который можно устанавливать на функции, возвращающие результат типа ***double*** и имеющие два параметра: первый – символьного типа, а второй – вещественного типа.

2. Идентификатор функции является константным указателем, поэтому для того чтобы установить переменную-указатель на конкретную функцию, достаточно ей присвоить ее идентификатор:

***переменная-указатель = ID_функции;***

Например, имеется функция с прототипом: double f1(char, double); тогда операция

p_f = f1;

установит указатель *p_f* на данную функцию.

3. Вызов функции после установки на нее указателя выглядит так:

(*переменная-указатель)(список аргументов);

или

переменная-указатель (список аргументов);

После таких действий кроме стандартного обращения к функции:

ID_функции(список аргументов);

появляется еще два способа вызова функции:

(*переменная-указатель)(список аргументов);

или

переменная-указатель (список аргументов);

Последняя запись справедлива, так как *p_f* также является адресом начала функции в оперативной памяти.

Для нашего примера к функции *f*1 можно обратиться следующими способами:

f1(‘z’, 1.5); – обращение к функции по имени (ID);
(* p_f)(‘z’, 1.5); – обращение к функции по указателю;
p_f(‘z’, 1.5); – обращение к функции по ID указателя.

Основное назначение указателей на функции – это обеспечение возможности передачи идентификаторов функций в качестве параметров в функцию, которая реализует некоторый вычислительный процесс, используя формальное имя вызываемой функции.

***Пример***: написать функцию вычисления суммы *sum*, обозначив слагаемое формальной функцией *fun*(*x*). При вызове функции суммирования передавать через параметр реальное имя функции, в которой задан явный вид слагаемого. Например, пусть требуется вычислить две суммы:

$$S_1 = \sum_{i=1}^{2n} \frac{x}{5} \qquad \text{и} \qquad S_2 = \sum_{i=1}^{n} \frac{x}{2} .$$

Поместим слагаемые этих сумм в пользовательские функции *f*1 и *f*2 соответственно. При этом воспользуемся операцией *typedef*, введя пользовательский тип данных: указатель на функции ***p_f***, который можно устанавливать на функции, возвращающие результат *double* и имеющие один параметр типа *double*.

Тогда в списке параметров функции суммирования достаточно будет указать фактические идентификаторы функций созданного типа ***p_f***.

Текст программы для решения данной задачи может быть следующим:

```
. . .
typedef double (*p_f)(double);
double sum(p_f, int, double);          // Декларации прототипов функций
double f1(double);
double f2(double);
void main(void)
{
    double x, s1, s2;
    int n;
    puts (" Введите кол-во слагаемых n и значение x: ");
    scanf (" %d %lf ", &n, &x);
    s1 = sum (f1, 2*n, x);
    s2 = sum (f2, n, x);
    printf("\n\t N = %d , X = %lf ", n, x);
    printf("\n\t Сумма 1 = %lf\n\t Сумма 2 = %lf ", s1, s2);
}
/*  Первый параметр функции суммирования – формальное имя функции, введенное с помощью typedef типа */
double sum(p_f  fun, int n, double x)  {
    double s=0;
    for(int i=1; i<=n; i++)
        s+=fun(x);
    return s;
}
//————————— Первое слагаемое ———————————
double f1(double r) {
    return r/5.;
}
//————————— Второе слагаемое ———————————
```

```
double f2(double r) {
        return r/2.;
}
```

В заключение рассмотрим оптимальную передачу в функции одномерных и двухмерных массивов.

***Передача в функцию одномерного массива***:

```
void main(void)
{
        int vect[20];
                …
        fun(vect);
                …
 }
void   fun( int v[ ]) {
        …
}
```

При использовании в качестве параметра одномерного массива в функцию передается указатель на его первый элемент, т.е. массив всегда передается по адресу и параметр v[ ] преобразуется в *v. Поэтому этой особенностью можно воспользоваться сразу:

```
void   fun( int *v) {
        …
}
```

При этом информация о количестве элементов массива теряется, так как размер одномерного массива недоступен вызываемой функции. Данную особенность можно обойти несколькими способами. Например, передавать его размер через отдельный параметр. Если же размер массива является константой, можно указать ее и при описании формального параметра, и в качестве границы циклов при обработке массива внутри функции:

```
 void   fun( int v[20]) {
        . . .
}
```

В случае передачи массива символов, т.е. строки, ее фактическую длину можно определить по положению признака окончания строки (нуль-символа) через стандартную функцию *strlen*.

***Передача в функцию двухмерного массива***:

Если размеры известны на этапе компиляции, то

```
void f1(int m[3][4])  {
        int i, j;
        for ( i = 0; i<3; i++)
                for ( j = 0; j<4; j++)
                        . . .                          // Обработка массива
        }
```

Двухмерный массив, как и одномерный, также передается как указатель, а указанные размеры используются просто для удобства записи. При этом первый размер массива не используется при поиске положения элемента массива в ОП, поэтому передать массив можно так:

```
void main(void)
{
        int mas [3][3]={{1,2,3}, {4,5,6}};
                …
        fun (mas);
                …
}
void fun( int m[ ][3]) {
        …
}
```

Если же размеры двухмерного массива, например, вводятся с клавиатуры (неизвестны на этапе компиляции), то их значения следует передавать через дополнительные параметры, например:

```
        …
void fun( int**, int, int);
void main()
{
        int **mas, n, m;
                ...
        fun (mas, n, m);
                …
}
void fun( int **m, int n, int m)  {
                . . .                                   // Обработка массива
}
```

### *11.6. Рекурсивные функции*

Рекурсивной (самовызываемой или самовызывающей) называют функцию, которая прямо или косвенно вызывает сама себя.

При каждом обращении к рекурсивной функции создается новый набор объектов автоматической памяти, локализованных в коде функции.

Возможность прямого или косвенного вызова позволяет различать прямую или косвенную рекурсии. Функция называется косвенно рекурсивной в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов первой функции. В этом случае по тексту определения функции ее рекурсивность (косвенная) может быть не видна. Если в функции используется вызов этой же функции, то имеет место прямая рекурсия, т.е. функция по определению рекурсивная.

Рекурсивные алгоритмы эффективны в задачах, где рекурсия использована в самом определении обрабатываемых данных. Поэтому изучение рекурсивных методов нужно проводить, вводя динамические структуры данных с рекурсивной структурой. Рассмотрим вначале только принципиальные возможности, которые предоставляет язык Си для организации рекурсивных алгоритмов.

В рекурсивных функциях необходимо выполнять следующие правила.

1. При каждом вызове в функцию передавать модифицированные данные.

2. На каком-то шаге должен быть прекращен дальнейший вызов этой функции, это значит, что рекурсивный процесс должен шаг за шагом упрощать задачу так, чтобы для нее появилось нерекурсивное решение, иначе функция будет вызывать себя бесконечно.

3. После завершения очередного обращения к рекурсивной функции в вызывающую функцию должен возвращаться некоторый результат для дальнейшего его использования.

***Пример* 1**. Заданы два числа *a* и *b*, большее из них разделить на меньшее, используя рекурсию.

Текст программы может быть следующим:

```
        . . .
double proc(double, double);
void main (void)
{
        double a,b;
        puts(“ Введи значения a, b : ”);
        scanf(“%lf %lf”, &a, &b);
        printf(“\n Результат деления : %lf”, proc(a,b));
}
//———————————— Функция ————————————————
double proc( double a, double b)  {
        if ( a< b ) return proc ( b, a );
                else  return a/b;
}
```

Если *a* больше *b*, условие, поставленное в функции, не выполняется и функция *proc* возвращает нерекурсивный результат.

Пусть теперь условие выполнилось, тогда функция *proc* обращается сама к себе, аргументы в вызове меняются местами и последующее обращение приводит к тому, что условие вновь не выполняется и функция возвращает нерекурсивный результат.

***Пример* 2**. Функция для вычисления факториала *неотрицательного* значения *k* (для возможных отрицательных значений необходимо добавить дополнительные условия).

```
double fact (int k) {
        if ( k < 1 ) return 1;
        else
                return  k * fact ( k – 1);
}
```

Для нулевого значения параметра функция возвращает 1 (0! = 1), в противном случае вызывается та же функция с уменьшенным на 1 значением

параметра и результат умножается на текущее значение параметра. Тем самым для значения параметра *k* организуется вычисление произведения

$$k * (k–1) * (k–2) * ... * 3 * 2 * 1 * 1$$

Последнее значение «1» – результат выполнения условия $k < 1$ при $k = 0$, т.е. последовательность рекурсивных обращений к функции *fact* прекращается при вызове *fact*(0). Именно этот вызов приводит к последнему значению «1» в произведении, так как последнее выражение, из которого вызывается функция, имеет вид: 1 * *fact*( 1 – 1).

***Пример* 3**. Рассмотрим функцию определения корня уравнения $f(x) = 0$ на отрезке $[a, b]$ с заданной точностью eps. Предположим, что исходные данные задаются без ошибок, т.е. $eps > 0$, $f(a)*f(b) < 0$, $b > a$, и вопрос о возможности существования нескольких корней на отрезке $[a,b]$ нас не интересует. Не очень эффективная рекурсивная функция для решения поставленной задачи приведена в следующей программе:

```
    . . .
int counter = 0;            // Счетчик обращений к тестовой функции
//———————— Нахождение корня методом деления отрезка пополам ——————————

double   Root(double f(double), double a, double b, double eps)  {
        double  fa = f(a), fb = f(b), c, fc;
        if ( fa * fb > 0) {
                printf("\n На интервале a,b НЕТ корня!");
                exit(1);
                }
        c = (a + b) / 2.0;
        fc = f(c);
        if (fc == 0.0 ||  fabs(b – a) < = eps) return c;
        return (fa * fc < 0.0) ? Root(f, a, c, eps) : Root(f, c, b, eps);
}
//————————————————————————————————————————————————————
void main()
{
        double x, a=0.1, b=3.5, eps=5e–5;
        double fun(double) ;                    // Прототип тестовой функции
        x = Root (fun, a, b, eps) ;
        printf ("\n Число обращений к функции = %d . ", counter);
        printf ("\n Корень = %lf  . ", x);
}
//———————————— Определение тестовой функции  fun ————————————————
double fun (double x) {
        counter++;          // Счетчик обращений – глобальная переменная
        return (2.0/x * cos(x/2.0));
}
```

Значения *a*, *b* и *eps* заданы постоянными только для тестового анализа полученных результатов, хотя лучше данные вводить с клавиатуры.

В результате выполнения программы с определенными в ней конкретными данными получим:

*Число обращений к функции = 54 .*

*Корень = 3.141601.*

Неэффективность предложенной программы связана, например, с излишним количеством обращений к программной реализации функции, для которой определяется корень. При каждом рекурсивном вызове функции *Root* повторно вычисляются значения *f*(*a*) и *f*(*b*), хотя они уже известны после предыдущего вызова.

В литературе по программированию рекурсиям уделено достаточно внимания как в теоретическом плане, так и в плане рассмотрения механизмов реализации рекурсивных алгоритмов. Сравнивая рекурсию с итерационными методами, отмечают, что рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно. В тех случаях, когда вычисляемые значения определяются с помощью простых рекуррентных соотношений, гораздо эффективнее применять итерационные методы.

Таким образом, рассмотренные выше примеры только иллюстрируют схемы организации рекурсивных функций, но не являются примерами эффективного применения рекурсивного подхода к вычислениям.

При обработке динамических информационных структур, которые включают рекурсивность в само определение обрабатываемых данных, применение рекурсивных алгоритмов не имеет конкуренции со стороны итерационных методов.

### *11.7. Параметры командной строки функции main*

В стандарте *ANSI* функция *main* возвращает целочисленный результат, т.е. используется следующим образом:

```
int main () {
        …
        return 0;
}
```

здесь оператор *return* возвращает операционной системе код завершения функции, причем значение 0 трактуется как нормальное завершение, остальные значения воспринимаются как ошибки.

Функция *main* может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию *main* используются два параметра, общепринятые (необязательные) идентификаторы которых *argc* и *argv*:

```
int main (int argc, char *argv[])  ...
```

Параметр *argc* имеет тип *int*, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой функции. Параметр *argv* – это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ пробел, то при записи его в командную строку оно должно быть заключено в кавычки.

Функция *main* может иметь и третий параметр *argp*, который служит для передачи параметров операционной системы (ОС), в которой выполняется программа, в этом случае ее заголовок имеет вид

```
int main (int argc, char *argv[], char *argp[])
```

Операционная система поддерживает передачу значений для параметров *argc*, *argv*, *argp*, а пользователь отвечает за передачу и использование фактических аргументов функции *main*.

Приведем пример программы печати фактических аргументов, передаваемых из ОС в функцию *main* и параметров оперативной системы.

```
int main ( int argc, char *argv[], char *argp[])
{
        int i;
        printf ("\n Program Name %s", argv[0]);
        for( i=1; i <=argc; i++)
            printf ("\n Argument %d  =  %s", i, argv[i]);
        printf ("\n ОС parametrs: ");
        while  (*argp) {
            printf ("\n %s", *argp);
            argp++;
        }
        return 0;
}
```

Очевидно, что оформленная таким образом функция *main*() может вызываться рекурсивно из любой функции.

# ГЛАВА 12. Классы памяти и область действия объектов

## *12.1. Классы памяти объектов в языке Си*

Напомним, что все объекты программы на Си перед их использованием должны быть декларированы. Операционные объекты (в частности переменные) при этом, кроме атрибута «*тип*», имеют необязательный атрибут «*класс памяти*», существенно влияющий на область и время их действия.

***Класс памяти*** программного объекта определяет время ее существования (время жизни) и область видимости (действия) и может принимать одно из значений: ***auto***, ***extern***, ***static*** и ***register***.

Класс памяти и область действия объектов по умолчанию зависят от места их размещения в коде программы.

Область действия объекта – это часть кода программы, в которой его можно использовать для доступа к связанному с ним участку памяти. В зависимости от области действия переменная может быть локальной (внутренней) или глобальной (внешней).

Имеется три основных участка программы, где можно декларировать переменные:

– внутри функции (блока);

– в заголовке функции при определении параметров;

– вне функции.

Эти переменные соответственно называются локальными (внутренними) переменными, формальными параметрами и глобальными (внешними) переменными.

Область действия локальных данных – от точки декларации до конца функции (блока), в которой произведена их декларация, включая все вложенные блоки.

Областью действия глобальных данных считается файл, в котором они определены, от точки описания до его окончания.

Если класс памяти у переменной не указан явным образом, он определяется компилятором исходя из контекста ее декларации.

Время жизни может быть постоянным – в течение выполнения программы, и временным – в течение выполнения функции (блока) программы.

### *12.2. Автоматические переменные*

Переменные, декларированные внутри функций, являются внутренними и называются локальными переменными. Никакая другая функция не имеет прямого доступа к ним. Такие объекты существуют временно на этапе активности функции.

Каждая локальная переменная существует только в блоке кода, в котором она объявлена, и уничтожается при выходе из него. Эти переменные называются автоматическими и располагаются в стековой области памяти.

При необходимости такая переменная инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти происходит при выходе из функции (блока), в которой декларирована переменная, т.е. время ее жизни – с момента описания до конца блока.

По умолчанию локальные объекты, объявленные в коде функции, имеют атрибут класса памяти ***auto***.

Принадлежность к этому классу можно также подчеркнуть явно, например:

```
void main(void)  {
        auto int max, lin;
                ...
}
```

так поступают, если хотят показать, что определение переменной не нужно искать вне функции.

Для глобальных переменных этот атрибут не используется.

***Регистровая память*** (атрибут ***register***) – объекты целого типа и символы рекомендуется размещать не в ОП, а в регистрах общего назначения (процессора), а при нехватке регистров – в стековой памяти (размер объекта не должен превышать разрядности регистра), для других типов компилятор может использовать другие способы размещения или просто проигнорировать данную рекомендацию.

Регистровая память позволяет увеличить быстродействие программы, но к размещаемым в ней объектам в языке Си (но не C++) не применима операция получения адреса «&».

### *12.3. Статические и внешние переменные*

Объекты, размещаемые в статической памяти, декларируются с атрибутом ***static*** и могут иметь любой атрибут области действия. В зависимости от расположения оператора описания статические переменные могут быть глобальными и локальными. Время жизни – постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. Глобальные объекты всегда являются статическими. Атрибут *static*, использованный при описании глобального объекта, предписывает ограничение его области применения только в пределах остатка текущего файла, а значения локальных статических объектов сохраняются до повторного вызова функции, т.е. в языке Си ключевое слово *static* имеет разный смысл для локальных и глобальных объектов.

Итак, переменная, описанная вне функции, является внешней (глобальной) переменной.

Так как внешние переменные доступны всюду, их можно использовать вместо списка аргументов для передачи значений между функциями.

Внешние переменные существуют постоянно. Они сохраняют свои значения и после того, как функции, присвоившие им эти значения, завершат свою работу.

При отсутствии явной инициализации для внешних и статических переменных гарантируется их обнуление. Автоматические и регистровые переменные имеют неопределенные начальные значения («*мусор*»).

Внешняя переменная должна быть определена вне всех функций. При этом ей выделяется фактическое место в памяти. Такая переменная должна быть описана в той функции, которая собирается ее использовать. Это можно сделать либо явным описанием *extern*, либо по контексту.

Описание *extern* может быть опущено, если определение внешней переменной находится в том же файле, но до ее использования в некоторой конкретной функции.

Ключевое слово *extern* позволяет функции использовать внешнюю переменную, даже в том случае, если она определяется позже в этом или другом файле.

Важно различать описание внешней переменной и ее определение. Описание указывает свойство переменной, ее размер, тип и т. д.; определение же вызывает еще и отведение ей участка оперативной памяти. Например, если вне какой-либо функции появляются инструкции

```
int sp;
double val[20];
```

то они определяют внешние переменные *sp* и *val*, вызывают отведение памяти для них и служат в качестве описания для остальной части этого исходного файла. В то же время строчки:

```
extern int sp;
extern double val [ ];
```

описывают в остальной части этого исходного файла переменную *sp* как *int*, а *val* как массив типа *double*, но не создают переменных и не отводят им места в памяти.

Во всех файлах, составляющих исходную программу, должно содержаться только одно определение внешней переменной. Другие файлы могут содержать описание *extern* для доступа к ней.

Любая инициализация внешней переменной проводится только в декларации. В декларации должны указываться размеры массивов, а в описании *extern* этого можно не делать.

Например, в основном файле проекта:

```
int sp = 50;
double val [20];
void main() {
    ...
```

а в дополнительном файле этого проекта:

```
extern int sp;
extern double val [ ];
    ...
```

В Си есть возможность с помощью директивы компилятору *#include* использовать во всей программе только одну копию описаний *extern* и присоединять ее к каждому файлу во время его препроцессорной обработки.

Если переменная с таким же идентификатором, как внешняя, декларирована в функции без указания *extern*, то тем самым она становится внутренней (локальной) для данной функции.

Не стоит злоупотреблять внешними переменными, так как такой стиль программирования приводит к программам, связи данных внутри которых не вполне очевидны. Переменные при этом могут изменяться неожиданным образом. Модификация таких программ вызывает затруднения.

Пример, иллюстрирующий использование внешних данных:

| Основной файл проекта | Дополнительный файл |
|---|---|
| …<br>int x, y;<br>char str[ ] = “Rezult = ”;<br>void fun1(void);<br>void fun2(void);<br>void fun3(void);<br>void main(void)<br>{<br>fun1();<br>fun2();<br>fun3();<br>}<br>void fun1(void) {<br>y = 15;<br>printf(“\n %s %d\n”, str, y);<br>} | …<br>extern int x, y;<br>extern char str[ ];<br>int r = 4;<br><br>void fun2(void) {<br>x = y / 5 + r;<br>printf(“ %s %d\n”, str, x);<br>}<br><br>void fun3(void) {<br>int z= x + y;<br>printf(“ %s %d\n”, str, z);<br>} |

В результате выполнения этого проекта, состоящего из двух различных файлов, будет получено следующее:

*Rezult = 15*
*Rezult = 7*
*Rezult = 22*

### *12.4. Область действия переменных*

В языке Си нет ключевого слова, указывающего область действия (видимости) объекта. Область действия определяется местоположением декларации объекта в файле исходного текста программы.

Напомним общую структуру исходного текста программы:

<директивы препроцессора>
<описание глобальных объектов>
<заголовок функции>
{
<описание локальных объектов>
<список инструкций>
}

Файл исходного текста может включать любое количество определений функций и/или глобальных данных.

Параметры функции являются локальными объектами и должны отличаться по идентификаторам от используемых в коде функции глобальных объектов. Локальные объекты, описанные в коде функции, имеют приоритет перед объектами, описанными вне функции, например:

```
#include<stdio.h>
        int f1(int);
        int f2(int);
        int f3(int);
        int n;                                  // Глобальная n
void main (void)
{
        int i=2;                                // Локальная i=2
        n=3;                                    // Глобальная n=3
        i = f1(i);                              // Обе переменные i – локальные
        printf(" 1: i=%d , n=%d\n",i,n);        // i=7, n=3
        n = f1(i);                              // n – глобальная, i – локальная
        printf(" 2: i=%d , n=%d\n",i,n);        // i=7, n=12
        i = f2(n);                              // i и n – локальные
        printf(" 3: i=%d , n=%d\n",i,n);        // i=15, n=12
        i = f3(i);                              // Обе переменные i – локальные
        printf(" 4: i=%d , n=%d\n",i,n);        // i=29, n=14
}
//-------------------------------------------------------------------------
int f1(int i)   {               // Параметр функции i – локальная
        int n = 5;              // n – локальная
        n+=i;
        return n;
}
//-------------------------------------------------------------------------
int f2(int n)  {                // Параметр функции n – локальная
        n+=3;
        return n;
}
//-------------------------------------------------------------------------
int f3(int i)  {
        n+=2;                   // n – глобальная
        return i+n;
}
```

Существуют следующие области действия: блок, файл, функция, прототип функции, область структурированных типов данных.

***Блок***. Идентификаторы, описанные внутри блока, являются локальными. Область действия идентификатора начинается в точке определения и заканчивается в конце блока, видимость – в пределах блока и внутренних блоков, время жизни – до выхода из блока. После выхода из блока память освобождается.

***Файл***. Идентификаторы, описанные вне любого блока, функции, класса или пространства имен, имеют глобальную видимость и постоянное время жизни и могут использоваться с момента их определения.

***Функция***. Единственными идентификаторами, имеющими такую область действия, являются метки операторов. В одной функции все метки должны различаться, но могут совпадать с метками других функций.

***Прототип функции***. Идентификаторы, указанные в списке параметров прототипа (декларации) функции, имеют областью действия только прототип функции.

***Структурированный тип данных***. Элементы структур и объединений являются видимыми лишь в их пределах. Они образуются при создании переменной указанного типа и разрушаются при ее уничтожении.

В языке C++ допускается в разных блоках программы использовать один и тот же идентификатор для разных объектов. Декларация такого идентификатора внутри блока скрывает доступ к ранее объявленному, например:

```
void main(void)
{
        int a = 3;
        printf("\n Block 1: %d ", a); {
            double a = 2.5;
            printf( "\n Block 2:  %lf ", a); {
                char a = 'A';
                printf( "\n Block 3:  %c ", a);
            }
            printf( "\n New Block 2:  %lf",  a+1.25);
        }
        printf( "\n New Block 1:  %d", ++a);
    }
```

Результат программы:

*Block 1:  3*
*Block 2:  2.5*
*Block 3:  A*
*New Block 2:  3.75*
*New Block 1:  4*

***Советы по программированию***

При выполнении вариантов заданий придерживайтесь следующих ключевых моментов.

1. Размеры нединамических массивов задаются константами или константными выражениями. Рекомендуется для этого использовать поименованные константы.

2. Элементы массивов нумеруются с нуля, максимальный номер (индекс) элемента всегда на единицу меньше указанного размера.

3. Автоматический контроль выхода индексов элементов за указанные границы массива отсутствует.

4. Указатель – это переменная, в которой хранится адрес участка оперативной памяти.

5. Имя массива является указателем на его нулевой элемент, т.е. на его начало в оперативной памяти.

6. Обнуления динамической памяти при ее выделении не происходит. Инициализировать динамический массив нельзя.

7. Освобождение памяти, выделенной посредством стандартных функций, выполняется при помощи функции *free* (при использовании операции *new* – операцией *delete*).

8. Если количество элементов массива известно заранее, определяйте массив в области декларации переменных (причем лучше как локальный объект). Если же количество элементов массива можно задать во время выполнения программы, лучше создать динамический массив.

9. При задании длины строки необходимо учитывать завершающий нуль-символ (признак окончания строки).

10. Операция присваивания строк выполняется с помощью функции стандартной библиотеки.

11. Для ввода строк, содержащих пробелы, используют функцию *gets*.

12. Программа, написанная с использованием функций, может получиться более быстродействующей, но менее безопасной.

13. Недостатком символьных массивов является отсутствие проверки выхода за пределы отведенной им памяти.

14. Двухмерный массив хранится по строкам в непрерывной области памяти.

15. Первый индекс двухмерного массива трактуется как номер строки таблицы, второй – как номер столбца. Каждый индекс может изменяться от 0 до значения соответствующего размера, уменьшенного на единицу.

16. Для выделения динамической памяти под массив, в котором все размеры переменные, используются циклы.

17. Функция – это именованная последовательность операторов, выполняющая законченное действие. Функции используют для упрощения структуры программы.

18. Вызов функции осуществляется путем указания ее идентификатора (имени) и в случае необходимости – набора аргументов.

19. Передача аргументов в функцию может выполняться по значению или по адресу.

20. Массивы всегда передаются в функцию по адресу. Количество элементов в массиве должно передаваться отдельным параметром.

21. Рекурсивная функция должна содержать хотя бы одну не рекурсивную ветвь. При использовании рекурсии следует учитывать возникающее при этом использование дополнительной памяти.

## ЗАДАНИЕ 4. Обработка массивов

***Первый уровень сложности***

Составить программу, решающую указанную ниже задачу.

В одномерном массиве, состоящем из $n$ (не более 10) вводимых с клавиатуры значений, вычислить заданное значение.

1. Произведение элементов массива, расположенных между максимальным и минимальным элементами.

2. Сумму элементов массива, расположенных между первым и последним нулевыми элементами.

3. Сумму элементов массива, расположенных до последнего положительного элемента.

4. Сумму элементов массива, расположенных между первым и последним положительными элементами.

5. Произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

6. Сумму элементов массива, расположенных между первым и вторым отрицательными элементами.

7. Сумму элементов массива, расположенных до минимального элемента.

8. Сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.

9. Сумму элементов массива, расположенных после последнего элемента, равного нулю.

10. Сумму модулей элементов массива, расположенных после минимального по модулю элемента.

11. Сумму элементов массива, расположенных после минимального элемента.

12. Сумму элементов массива, расположенных после первого положительного элемента.

13. Сумму модулей элементов массива, расположенных после первого отрицательного элемента.

14. Сумму модулей элементов массива, расположенных после первого элемента, равного нулю.

15. Сумму положительных элементов массива, расположенных до максимального элемента.

***Второй уровень сложности***

Написать программу по обработке двухмерного массива. Предусмотреть динамический захват и освобождение памяти. Размеры массива $n$, $m$ и значения элементов массива вводятся с клавиатуры.

1. Определить количество строк, не содержащих ни одного нулевого элемента.

2. Определить количество столбцов, не содержащих ни одного нулевого элемента.

3. Определить количество столбцов, содержащих хотя бы один нулевой элемент.

4. Определить произведение элементов в тех строках, которые не содержат отрицательных элементов.

5. Определить сумму элементов в тех столбцах, которые не содержат отрицательных элементов.

6. Определить сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

7. Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

8. Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.

9. Найти сумму модулей элементов, расположенных ниже главной диагонали.

10. Найти сумму модулей элементов, расположенных выше главной диагонали.

11. Найти количество строк, среднее арифметическое элементов которых меньше введенной с клавиатуры величины.

12. Найти номер первой из строк, содержащих хотя бы один положительный элемент.

13. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.

14. Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.

15. Найти номер первой из строк, не содержащих ни одного положительного элемента.

## ЗАДАНИЕ 5. Функции пользователя

***Первый уровень сложности***

Составить программу из задания 3 (второй уровень сложности), в которой для каждого ***x***, изменяющегося от ***a*** до ***b*** с шагом ***h***, вычисление значений $Y(x)$ и $S(x)$ оформить в виде функций пользователя.

В основной функции реализовать следующие действия:

– ввод исходных значений $a$, $b$, $h$ и $n$;

– обращение к функциям расчета $Y(x)$ и $S(x)$;

– вывод результатов в виде таблицы.

Если в задании используется значение факториала, его расчет также оформить функцией.

***Второй уровень сложности***

Решить поставленную задачу с использованием рекурсивной и обычной функций. Сравнить полученные результаты.

1. Для заданного целого десятичного числа $N$ получить его представление в $p$-ичной системе счисления ($p < 10$).

2. В упорядоченном массиве целых чисел $a_i$ ($i$=1, ..., $n$) найти номер находящегося в массиве элемента $c$, используя метод двоичного поиска.

3. Найти наибольший общий делитель чисел $M$ и $N$, используя теорему Эйлера: если $M$ делится на $N$, то $НОД\ (N, M) = N$, иначе $НОД\ (N, M) = (M\ mod\ N, N)$.

4. Числа Фибоначчи определяются следующим образом: $Fb(0) = 0$; $Fb(1) = 1$; $Fb(n) = Fb(n\text{-}1) + Fb(n\text{-}2)$. Определить $Fb(n)$.

5. Найти значение функции Аккермана $A(m, n)$, которая определяется для всех неотрицательных целых аргументов $m$ и $n$ следующим образом:

$A(0, n) = n + 1$;

$A(m, 0) = A(m\text{-}1, 1)$; при $m > 0$;

$A(m, n) = A(m\text{-}1, A(m, n\text{-}1))$; при $m > 0$ и $n > 0$.

6. Найти методом деления отрезка пополам минимум функции $f(x) = 7\sin^2(x)$ на отрезке [2, 6] с заданной точностью ε (например 0.01).

7. Вычислить значение $x = \sqrt{a}$, используя рекуррентную формулу $x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right)$, в качестве начального значения использовать $x_0 = 0{,}5(1 + a)$.

8. Найти максимальный элемент в массиве $a_i$ ($i$=1, …, $n$), используя очевидное соотношение $max(a_1, \ldots, a_n) = max[max(a_1, \ldots, a_{n\text{-}1}), a_n]$.

9. Вычислить значение $y(n) = \sqrt{1+\sqrt{2+\mathbf{K}+\sqrt{n}}}$.

10. Найти максимальный элемент в массиве $a_i$ ($i$=1, …, $n$), используя соотношение (деления пополам) $max(a_1,\ldots, a_n) = max[max(a_1,\ldots, a_{n/2}), max(a_{n/2+1}, \ldots, a_n)]$.

11. Вычислить значение $y(n) = \cfrac{1}{n+\cfrac{1}{(n-1)+\cfrac{1}{(n-2)+\cfrac{1}{\mathbf{L}\cfrac{}{\mathbf{L}+\cfrac{1}{1+\cfrac{1}{2}}}}}}}$.

12. Вычислить произведение четного количества $n$ ($n \geq 2$) сомножителей следующего вида $y = \left(\frac{2}{1}\cdot\frac{2}{3}\right)\cdot\left(\frac{4}{3}\cdot\frac{4}{5}\right)\cdot\left(\frac{6}{5}\cdot\frac{6}{7}\right)\ldots$.

13. Вычислить $y = x^n$ по следующему правилу: $y = (x^{n/2})^2$, если $n$ четное и $y = x \cdot y^{n-1}$, если $n$ нечетное.

14. Вычислить значение $C_n^k = \frac{n!}{k!(n-k)!}$ (значение 0! = 1).

15. Вычислить $y(n) = \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\mathbf{L}}}}}$, $n$ задает число ступеней.

# ГЛАВА 13. Структуры, объединения, перечисления

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе базовых типов данных, массивов и указателей. Языки высокого уровня позволяют программисту определять свои типы данных и правила работы с ними, т.е. типы, определяемые пользователем. В языке Си к ним относятся структуры, объединения и перечисления. Рассмотрим их более подробно.

## *13.1. Структуры*

Структура – это составной объект языка Си, представляющий собой совокупность логически связанных данных различных типов, объединенных в группу под одним идентификатором. Данные, входящие в эту группу, называют полями.

Термин «***структура***» в языке Си соответствует двум разным по смыслу понятиям:

– структура – это обозначение участка оперативной памяти, где располагаются конкретные значения данных; в дальнейшем – это структурная переменная, поля которой располагаются в смежных областях ОП;

– структура – это правила формирования структурной переменной, которыми руководствуется компилятор при выделении ей места в ОП и организации доступа к ее полям.

Определение объектов типа структуры производится за два шага:

– декларация структурного типа данных, не приводящая к выделению участка памяти;

– определение структурных переменных объявленного структурного типа с выделением для них памяти.

### *13.2. Декларация структурного типа данных*

Структурный тип данных задается в виде шаблона, общий формат описания которого следующий:

```
struct ID структурного типа {
        описание полей
};
```

Атрибут «*ID структурного типа*» является необязательным и может отсутствовать. Описание полей производится обычным способом: указываются типы переменных и их идентификаторы.

***Пример определения структурного типа***

Необходимо создать шаблон, описывающий информацию о студенте: номер группы, Ф.И.О. и средний балл. Один из возможных вариантов:

```
struct Stud_type  {
        char Number[10];
        char Fio[40];
        double S_b;
};
```

Поля одного типа при описании можно объединять в одну группу:

```
struct Stud_type  {
        char Number[10], Fio[40];
        double S_b;
};
```

Размещение данного объекта типа *Stud_type* в ОП схематически будет выглядеть следующим образом:

| *Number* | *Fio* | *S_b* |
|---|---|---|
| 10 байт | 40 байт | 8 байт |

Структурный тип данных удобно применять для групповой обработки логически связанных объектов. Параметрами таких операций являются адрес и размер структуры.

Примеры групповых операций:

– захват и освобождение памяти для объекта;

– запись и чтение данных, хранящихся на внешних носителях как физические и/или логические записи с известной структурой (при работе с файлами).

Так как одним из параметров групповой обработки структурных объектов является размер, не рекомендуется декларировать поле структуры указателем на объект переменной размерности, т.к. в данном случае многие операции со структурными данными будут некорректны, например,

```
struct Stud_type  {
        char *Number, *fio;
        double S_b;
};
```

В данном случае, вводя строки *Number* и *fio* различной длины, размеры объектов будут также различны.

### *13.3. Создание структурных переменных*

Как уже отмечалось, само описание структуры не приводит к выделению под нее места в ОП. Для работы со структурами необходимо создать нужное количество переменных приведенного структурного типа, сделать это можно двумя способами.

***Способ* 1**. В любом месте программы для декларации структурных переменных, массивов, функций и т.д. используется объявленный в шаблоне структурный тип, например:

```
struct Stud_type  student;        – структурная переменная;
Stud_type  Stud[100];             – массив структур
Stud_type  *p_stud;               – указатель на структуру
Stud_type* Fun(Stud_type);  – прототип функции с параметром структурного типа, возвращающей указатель на объект структурного типа.
```

***Способ* 2**. В шаблоне структуры между закрывающейся фигурной скобкой и символом «;» указывают через запятые идентификаторы структурных данных.

Для нашего примера можно записать:

```
struct Stud_type  {
        char Number[10], Fio[40];
        double S_b;
} student, Stud[100], *p_stud;
```

Если дальше в программе не понадобится вводить новые данные объявленного структурного типа, идентификатор *Stud_type* можно не указывать.

При декларации структурных переменных возможна их одновременная инициализация, например:

```
struct Stud_type  {
        char Number[10], Fio[40];
        double S_b;
} student = {"123456", "Иванов И.И.", 6.53 };
```

или:

```
        Stud_Type  stud1 = {"123456", "Иванов И.И." };
```

Если список инициализаций будет короче, то оставшиеся поля структурной переменной заполняются нулями.

***Некоторые особенности:***

1) поля не могут иметь атрибут, указывающий «класс памяти», данный атрибут можно определить только для всей структуры;

2) идентификаторы полей могут совпадать с идентификаторами других объектов программы, т.к. шаблон структуры обладает собственным пространством имен;

3) при наличии в программе функций пользователя шаблон структуры рекомендуется поместить глобально перед определениями всех функций и в этом случае он будет доступен всем функциям.

### *13.4. Обращение к полям структур*

Обращение к полям структур производится путем создания составных имен, которые образуются двумя способами:

1) при помощи *операции принадлежности* (**.**) общий вид которой

***ID_структуры . ID_поля***

или

(***указатель_структуры***) **. *ID_поля***

2) при помощи *операции косвенной адресации* (**–>**) в виде

***указатель_структуры –> ID_поля***

или

(***&ID_структуры***) **–> *ID_поля***

Если в программе созданы объекты объявленного ранее шаблона:

```
        Stud_Type  s1, *s2;
```

то к полям объекта *s*1 можно обратиться следующим образом:

```
s1. Number,          s1. Fio,          s1. S_b;
```

или

```
(&s1) –> Number,     (&s1) –> Fio,     (&s1) –> S_b;
```

а к полям объекта, адрес которого s2:

s2 –> Number, s2 –> Fio, s2 –> S_b;

или

(*s2) . Number, (*s2) . Fio, (*s2) . S_b;

### *13.5. Вложенные структуры*

Структуры могут быть вложенными, т.е. поле структуры может быть связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основному шаблону.

Например, в структуре *Person*, содержащей сведения – ФИО, дата рождения, сделать дату рождения внутренней структурой *date* по отношению к структуре *Person*. Тогда шаблон такой конструкции будет выглядеть так:

```
struct date {
        int day, month, year;
        };
 struct Person   {
        char fio[40];
        struct date f1;
        };
```

Объявляем переменную и указатель на переменные такой структуры:

```
        struct Person  a, *p;
```

Инициализируем указатель *p* адресом переменной *a*:

```
                p = &a;
```

Тогда обращение к полям структурной переменной *a* будет выглядеть следующим образом:

*a* . fio *a*. f1 . day *a* . f1 . month *a* . f1 . year

или

p–>fio p–>f1.day p–>f1.month p–>f1.year

Можно в качестве связи с вложенной структурой использовать указатель на нее:

```
struct date  {
        int day, month, year;
        };
 struct Person  {
        char fio[40];
        struct date *f1;
        };
```

Тогда обращение к полям будет следующим:

*a* .fio *a*.f1–>day *a*.f1–>month *a*.f1–>year

или

p–>fio p–>f1–>day p–>f1–>month p–>f1–>year

### *13.6. Массивы структур*

Структурный тип «*struct ID_структуры*», как правило, используют для декларации массивов, элементами которых являются структурные переменные. Это позволяет создавать программы, оперирующие с простейшими базами данных.

Например, массив структур, объявленного ранее типа:

```
struct Person spisok[100];
```

причем ключевое слово *struct* можно не писать. Декларацию массива можно выполнить и в описании шаблона следующим образом:

```
struct Person {
        char fio[40];
        int day, month, year;
} spisok[100];
```

В данном случае обращение к полю, например, *day* элемента массива с индексом *i* может быть выполнено одним из следующих способов:

spisok[i].day=22; *(spisok+i).day=22; (spisok+i)–>day=22;

***Пример***. Приведем часть программы, иллюстрирующей создание массива структур и передачу структурных данных в функции:

```
struct Spisok {
        char Fio[20];
        double S_Bal;
};
//  Описание прототипов функций пользователя
    void Out(int, Spisok);
    void In(int, Spisok *);
void main(void)
{
    Spisok Stud[50], *sved;
        . . .
    for(i=0;i<N;i++)
        Vvod(i, &Stud[i]);
    puts("\n Spisok Students");
    for(i=0;i<N;i++)
        Out(i+1, Stud[i]);
        . . .
}
// Функция вывода на экран данных одного элемента структуры
```

```
void Out(int nom, Spisok dan)   {
            printf("\n %3d – %20s  %4.2lf ",nom, dan.Fio, dan.S_Bal);
}
// Функция ввода данных одного элемента структуры
void In (int nom, Spisok *sved)  {
            printf("\n Введите сведения %d : ", nom+1);
            fflush(stdin);
            puts("\n ФИО      – ");
            gets(sved–>Fio);
            puts("\n Средний балл   – ");
            scanf(“%lf”, &sved–>S_Bal);
}
```

### *13.7. Размещение структурных переменных в памяти*

При анализе размеров структурных переменных иногда число байт, выделенных компилятором под структурную переменную, оказывается больше, чем сумма байт ее полей. Это связано с тем, что компилятор выделяет участок ОП для структурных переменных с учетом выравнивания границ, добавляя между полями пустые байты по следующим правилам:

– структурные переменные, являющиеся элементами массива, начинаются на границе слова, т.е. с четного адреса;

– любое поле структурной переменной начинается на границе слова, т.е. с четного адреса и имеет четное смещение по отношению к началу переменной;

– при необходимости в конец переменной добавляется пустой байт, чтобы общее число байт было четное.

### *13.8. Объединения*

Объединение – поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента.

Объединенный тип данных декларируется подобно структурному типу:

```
union ID_объединения {
            описание полей
};
```

Пример описания объединенного типа:

```
union word {
        int nom;
        char str[20];
};
```

Пример объявления объектов объединенного типа:

```
union word *p_w,  mas_w[100];
```

Объединения применяют для экономии памяти в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных.

Практически все вышесказанное для структур имеет место и для объединений. Декларация данных типа ***union***, создание переменных этого типа и обращение к полям объединений производится аналогично структурам.

Пример использования переменных типа ***union***:

```
        . . .
    typedef union q {
        int a;
        double b;
        char s[5];
    } W;
void main(void)
{
    W  s, *p = &s;
    s.a = 4;
    printf("\n Integer a = %d, Sizeof(s.a) = %d", s.a, sizeof(s.a));
    p –> b = 1.5;
    printf("\n Double b = %lf, Sizeof(s.b) = %d", s.b, sizeof(s.b));
    strcpy(p–>s, "Minsk");
    printf("\n String a = %s, Sizeof(s.s) = %d", s.s, sizeof(s.s));
    printf("\n Sizeof(s) = %d", sizeof(s));
}
```

Результат работы программы:

*Integer a = 4, Sizeof(s.a) = 2*
*Double b = 1.500000, Sizeof(s.b) = 4*
*String a = Minsk, Sizeof(s.s) = 5*
*Sizeof(s) = 5*

### *13.9. Перечисления*

*Перечисления* – средство создания типа данных посредством задания ограниченного множества значений.

Определение перечисляемого типа данных имеет вид

```
enum ID_перечисляемого_типа {
                список_значений
};
```

Значения данных перечисляемого типа указываются идентификаторами, например:

```
enum marks {
    zero, one, two, three, four, five
```

```
};
```

Компилятор последовательно присваивает идентификаторам списка значений целочисленные величины 0, 1, 2,... . При необходимости можно явно задать значение идентификатора, тогда очередные элементы списка будут получать последующие возрастающие значения. Например:

```
enum level {
    low=100, medium=500, high=1000, limit
};
```

Константа *limit* по умолчанию получит значение, равное 1001.

Примеры объявления переменных перечисляемого типа:

```
enum marks Est;
enum level state;
```

Переменная типа *marks* может принимать только значения из множества {zero, one, two, three, four, five}.

Основные операции с данными перечисляемого типа:

– присваивание переменных и констант одного типа;

– сравнение для выявления равенства либо неравенства.

Практическое назначение перечисления – определение множества различающихся символических констант целого типа.

Пример использования переменных перечисляемого типа:

```
                . . .
    typedef enum {
            mo=1, tu, we, th, fr, sa, su
    } days;
void main(void)
{
    days w_day;         // Переменная перечисляемого типа
    int t_day, end, start;
// Текущий день недели, начало и конец недели соответственно
    puts(“ Введите день недели (от 1 до 7) : ”);
    scanf(“%d”, &t_day);
    w_day = su;
    start = mo;
    end = w_day – t_day;
    printf(“\n Понедельник – %d день недели, \
сейчас %d – й день и \n\
до конца недели %d дн. ”, start, t_day, end );
}
```

Результат работы программы:

*Введите день недели (от 1 до 7) : 5*
*Понедельник – 1 день недели, сейчас 5-й день и*
*до конца недели 2 дн.*

### *13.10. Битовые поля*

Битовые поля – это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «*да*/*нет*». Минимальная адресуемая ячейка памяти – 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа), не превышающая разрядности поля типа *int*:

```
struct fields {
    unsigned int flag: 1;
    unsigned int mask: 10;
    unsigned int code: 5;
};
```

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом – по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры. Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры. В основном битовые поля размещаются последовательно в поле типа *int*, а при нехватке места для очередного битового поля происходит переход на следующее поле типа *int*. Возможно объявление безымянных битовых полей, а длина поля 0 означает необходимость перехода на очередное поле *int*:

```
struct areas {
    unsigned f1: 1;
              : 2;        – безымянное поле длиной 2 бита;
    unsigned f2: 5;
               : 0        – признак перехода на следующее поле int;
    unsigned f3:5;
     double data;
     char buffs[100];     – структура может содержать элементы любых
                            типов данных;
};
```

Битовые поля могут использоваться в выражениях как целые числа соответствующей длины поля разрядности в двоичной системе исчисления. Единственное отличие этих полей от обычных объектов – запрет операции определения адреса (&). Следует учитывать, что использование битовых полей снижает быстродействие программы по сравнению с представлением данных в полных полях из-за необходимости выделения битового поля.

# ГЛАВА 14. Файлы в языке Си

***Файл*** – это набор данных, размещенный на внешнем носителе и рассматриваемый в процессе обработки как единое целое. В файлах размещаются данные, предназначенные для длительного хранения.

Различают два вида файлов: *текстовые* и *бинарные*.

***Текстовые файлы*** представляют собой последовательность ASCII символов и могут быть просмотрены и отредактированы с помощью любого текстового редактора. Эта последовательность символов разбивается на строки символов, при этом каждая строка заканчивается двумя кодами «перевод строки», «возврат каретки»: 13 и 10 (0xD и 0xA).

***Бинарные (двоичные) файлы*** представляют собой последовательность данных, структура которых определяется программно.

В языке Си не предусмотрены никакие заранее определенные структуры файлов. Все файлы рассматриваются компилятором как последовательность (поток байт) информации.

Для файлов определен маркер или указатель чтения-записи данных, который определяет текущую позицию доступа к файлу. Напомним, что с началом работы любой программы автоматически открываются стандартные потоки *stdin* и *stdout*.

В языке Си имеется большой набор функций для работы с файлами, большинство которых находятся в библиотеках ***stdio.h*** и ***io.h***. При этом потоки данных, с которыми работают функции ввода-вывода данных по умолчанию, буферизированы. Это означает, что при открытии потока с ним автоматически связывается определенный участок ОП, который и называется буфером. Все операции чтения-записи ведутся через этот буфер. Его размер фиксирован специальной константой *BUFSIZ*, которая определена в файле *stdio.h* как 512 (хотя программно ее можно изменять).

## *14.1. Открытие файла*

Каждому файлу в программе присваивается внутреннее логическое имя, используемое в дальнейшем при обращении к нему. Логическое имя (идентификатор файла) – это указатель на файл, т.е. на область памяти, где содержится вся необходимая информация о файле.

Формат объявления указателя на файл следующий:

***FILE *ID_указателя_на_файл;***

*FILE* – идентификатор структурного типа, описанный в стандартной библиотеке *stdio.h* и содержащий следующую информацию:

```
struct FILE {
    short level;        – число оставшихся в буфере непрочитанных байт;
                          обычный размер буфера – 512 байт; как только
```

*level* = 0, в буфер из файла читается следующий блок данных;

unsigned flags; – флаг статуса файла – чтение, запись, дополнение;

char fd; – дескриптор файла, т.е. число, определяющее его номер;

unsigned char hold; – непереданный символ, т.е. *ungetc*-символ;

short bsize; – размер внутреннего промежуточного буфера;

unsigned char buffer; – значение указателя для доступа внутри буфера; задает начало буфера, начало строки или текущее значение указателя внутри буфера в зависимости от режима буферизации;

unsigned char *curp; – текущее значение указателя для доступа внутри буфера; задает текущую позицию в буфере для обмена с программой;

unsigned istemp; – флаг временного файла;

short token; – флаг при работе с файлом;

};

Прежде чем начать работать с файлом, т.е. получить возможность чтения или записи информации в файл, его нужно открыть для доступа.

Для этого обычно используется функция

FILE* ***fopen***(char * ID_файла, char *режим);

Данная функция берет внешнее представление – физическое имя файла на носителе (дискета, винчестер) и ставит ему в соответствие логическое имя (программное имя – указатель файла).

Физическое имя, т.е. *ID* файла и путь к нему задается первым параметром – строкой, например, “*a:Mas_dat.dat*” – файл с именем *Mas_dat* и расширением *dat*, находящийся на дискете, “*d:\\work\\Sved.txt*” – файл с именем *Sved* и расширением *txt*, находящийся на винчестере в каталоге *work*.

***Внимание***. Обратный слеш «\», как специальный символ в строке записывается дважды.

При успешном открытии функция *fopen* возвращает указатель на файл (в дальнейшем – указатель файла). При ошибке возвращается *NULL*. Данная ситуация обычно возникает, когда неверно указывается путь к открываемому файлу, например, если указать путь, запрещенный для записи.

Второй параметр – строка, в которой задается режим доступа к файлу.

Возможные значения данного параметра следующие:

*w* – файл открывается для записи (*write*); если файла с заданным именем нет, то он будет создан; если же такой файл уже существует, то перед открытием прежняя информация уничтожается;

*r* – файл открывается для чтения (*read*); если такого файла нет, то возникает ошибка;

*a* – файл открывается для добавления (*append*) новой информации в конец;

*r*+ (*w*+) – файл открывается для редактирования данных, т.е. возможны и запись, и чтение информации;

*a*+ – то же, что и для *a*, только запись можно выполнять в любое место файла (доступно и чтение файла);

*t* – файл открывается в текстовом режиме;

*b* – файл открывается в двоичном режиме;

Последние два режима используются совместно с рассмотренными выше. Возможны следующие комбинации режимов доступа: "*w*+*b*", "*wb*+", "*rw*+", "*w*+*t*", "*rt*+", а также некоторые другие комбинации.

По умолчанию файл открывается в текстовом режиме.

Текстовый режим отличается от двоичного тем, что при открытии файла как текстового пара символов «перевод строки» и «возврат каретки» заменяется на один символ «перевод строки» для всех функций записи данных в файл, а для всех функций вывода – наоборот – символ «перевод строки» заменяется на два символа – «перевод строки» и «возврат каретки».

***Пример*** открытия файла:

FILE *f; – объявляется указатель на файл *f*;

f = fopen (" d:\\work\\Dat_sp.dat ", "w"); – открывается для записи файл с логическим именем *f*, имеющий физическое имя *Dat_sp.dat* и находящийся на диске *d* в каталоге *work*, или более кратко:

FILE *f = fopen ("d:\\work\\Dat_sp.dat", "w");

### *14.2. Закрытие файла*

После работы с файлом доступ к нему необходимо закрыть с помощью функции

*int* ***fclose*** (*указатель файла*);

Например, для предыдущего примера файл закрывается так: *fclose* (*f*);

Для закрытия нескольких файлов введена функция:

*void* ***fcloseall*** (*void*);

Если требуется изменить режим доступа к открытому в настоящий момент файлу, то его необходимо сначала закрыть, а затем вновь открыть с другими правами доступа. Для этого используется функция

*FILE** ***freopen*** (char *ID_файла, char *режим, FILE *указатель_файла);

которая сначала закрывает файл, заданный в третьем параметре (указатель файла), как это выполняет функция *fclose*, а затем выполняет действия, аналогичные функции *fopen*, используя указанные первый и второй параметры (открывает файл с *ID_файла* и правами доступа *режим*).

В языке Си имеется возможность работы с временными файлами, которые нужны только в процессе работы программы и должны быть

удалены после выполнения некоторых вычислений. В этом случае используется функция

*FILE** ***tmpfile*** (*void*);

которая создает на диске временный файл с правами доступа *w+b*. После завершения работы программы или закрытия этого (временного) файла он автоматически удаляется.

### *14.3. Запись-чтение информации*

Все действия по чтению-записи данных в файл можно разделить на три группы:

– операции посимвольного ввода-вывода;

– операции построчного ввода-вывода;

– операции ввода-вывода по блокам.

Рассмотрим основные функции для записи-чтения данных из файлов.

Для работы с текстовыми файлами в библиотеке языка Си содержится достаточно много функций, самыми распространенными из которых являются функции

***fprintf***, ***fscanf***, ***fgets***, ***fputs***.

Формат параметров этих функций практически такой же, как и формат рассмотренных ранее (см. разд. 5.3, 5.4) функций *printf*, *scanf*, *gets* и *puts*. Так же практически совпадают и действия этих функций. Отличие состоит в том, что *printf* и другие функции работают по умолчанию с экраном монитора и клавиатурой, а функции *fprintf* и другие – с файлом, указатель которого является одним из параметров этих функций.

Рассмотрим общий пример создания текстового файла:

```
#include<stdio.h>
void main(void)
{
    FILE *f1;
    int a=2, b=3;
    if( ! (f1 = fopen(“d:\\work\\f_rez.txt”,”w+t”) ) ) {        // f1 = NULL
        puts(“Open File Error!”);
        return;                                                 // exit(1);
    }
    fprintf(f1,”\t Файл результатов \n”);
    fprintf(f1,” %d плюс %d = %d\n”, a, b, a+b);
    fclose(f1);
}
```

Просмотрев содержимое файла любым текстовым редактором, можно убедиться, что данные в нем располагаются точно так, как на экране, если воспользоваться функцией *printf* с такими же списками параметров.

Создание текстовых результирующих файлов обычно необходимо для оформления отчетов, различных документов, а также других текстовых материалов.

Бинарные (двоичные) файлы обычно используются для организации баз данных, состоящих, как правило, из объектов структурного типа. При чтении-записи бинарных файлов удобнее всего пользоваться функциями *fread* и *fwrite*, которые выполняют ввод-вывод данных блоками.

Такой способ обмена данными требует меньше времени.

Функция

*unsigned* ***fread*** (*void* ****p***, *unsigned* ***size***, *unsigned* ***n***, *FILE* ****f***);

выполняет считывание из файла *f* *n* блоков размером *size* байт каждый в область памяти, адрес которой *p*. В случае успеха функция возвращает количество считанных блоков. При возникновении ошибки или по достижении признака окончания файла – значение ***EOF*** (*End Of File* – признак окончания файла).

Обратное действие выполняет функция:

*unsigned* ***fwrite*** (*void* ****p***, *unsigned* ***size***, *unsigned* ***n***, *FILE* ****f***);

при вызове которой в файл *f* будет записано *n* блоков размером *size* байт каждый из области памяти, начиная с адреса *p*.

### *14.4. Позиционирование в файле*

Каждый открытый файл, как уже отмечалось, имеет скрытый указатель на текущую позицию в нем. При открытии файла этот указатель устанавливается на начало данных, и все операции в файле будут производиться с данными, начинающимися в этой позиции.

При каждом выполнении функции чтения или записи указатель смещается на количество прочитанных или записанных байт, т.е. устанавливается после прочитанного или записанного блока данных в файле – это ***последовательный доступ к данным***.

В языке Си/С++ можно установить указатель на некоторую заданную позицию в файле. Для этого используют стандартную функцию ***fseek***, которая позволяет выполнить чтение или запись данных в произвольном порядке.

Декларация функции позиционирования следующая:

*int* ***fseek***(*FILE* ****f***, *long* ***size***, *int* ***code***);

Значение параметра *size* задает количество байт, на которое необходимо сместить указатель в файле *f*, в направлении параметра *code*, который может принимать следующие значения:

– смещение от начала файла – 0 (*SEEK_SET*);
– смещение от текущей позиции – 1 (*SEEK_CUR*);
– смещение от конца файла – 2 (*SEEK_END*).

Таким образом, смещение может быть как положительным, так и отрицательным, но нельзя выходить за пределы файла.

В случае успеха функция возвращает нулевое значение, а в случае ошибки (например, попытка выхода за пределы файла) – единицу.

Доступ к файлу с использованием функции позиционирования (*fseek*) называют ***произвольным доступом***.

Иногда нужно определить текущее положение в файле. Для этого используют функцию со следующей декларацией:

*long* ***ftell***(*FILE* ***f*);

которая возвращает значение указателя на текущую позицию в файле или –1 в случае ошибки.

### *14.5. Дополнительные файловые функции*

В заключение рассмотрим наиболее распространенные функции, с помощью которых можно организовать работу с файлами:

*int* ***fileno*** (*FILE* ***f*) – определяет и возвращает значение дескриптора (*fd*) файла *f*, т.е. число, определяющее номер файла;

*long* ***filelength*** (*int* ***fd***) – возвращает длину файла, имеющего дескриптор *fd* , в байтах;

*int* ***chsize*** (*int* ***fd***, *long* ***pos***) – выполняет изменение размера файла, имеющего номер *fd*, признак конца файла устанавливается после байта с номером *pos*;

*int* ***feof*** (*FILE* ***f*) – возвращает ненулевое значение при правильной записи признака конца файла;

*int* ***fgetpos*** (*FILE* ***f*, *long* ****pos***) – определяет значение текущей позиции *pos* файла *f*.

#### *Пример программы работы с файлом структур*

Создать программу, в которой реализованы создание, добавление и просмотр файла, содержащего информацию о фамилии и среднем балле студентов. Процесс добавления информации заканчивается при нажатии точки.

```
#include <stdio.h>
#include <stdlib.h>
struct Sved {
        char Fam[30];
        double S_Bal;
        } zap,zapt;
char Spis[]="c:\\work\\Sp.dat";
FILE *F_zap;
FILE* Open_file(char*, char*);

void main (void)
{
```

```
        int i, j, kodR, size = sizeof(Sved), kod_read;
        while(1) {
                puts("Создать  – 1\n Добавить– 3\nПросмотреть– 2\nВыход – 0");
                scanf("%d",&kodR);
                switch(kodR) {
                        case 1: case 3:
                        if(kodR==1) F_zap = Open_file (Spis,"w+");
                                else F_zap = Open_file (Spis,"a+");
                        while(2) {
                                puts("\n Fam (. – end) ");
                                scanf("%s",zap.Fam);
                                if((zap.Fam[0])=='.')  break;
                                puts("\n Ball: ");
                                scanf("%lf",&zap.S_Bal);
                                fwrite(&zap,size,1,F_zap);
                        }
                        fclose(F_zap);
                        break;
                        case 2:   F_zap = Open_file (Spis,"r+"); int nom=1;
                        while(2) {
                                if(!fread(&zap,size, 1, F_zap)) break;
                                printf(" %2d: %20s %5.2lf\n",
                                        nom++, zap.Fam, zap.S_Bal);
                        }
                        fclose(F_zap);
                        break;
                        case 0:   return;               // exit(0);
                }                                       // Закрывает switch()
        }                                               // Закрывает while()
}
// Функция обработки ошибочной ситуации при открытии файла
FILE* Open_file(char *file, char *kod)
{
        FILE *f;
        if(!(f = fopen(file, kod))) {
                puts("Open File Error!");
                exit(1);
        }
        return f;
}
```

***Советы по программированию***

При выполнении вариантов заданий придерживайтесь следующих ключевых моментов.

1. Объекты типов структуры и объединения применяются для логически связанных между собой данных различных типов.

2. После описания шаблона структурного типа данных ставится точка с запятой.

3. Элементы данных, входящие в структуры и объединения, называются полями. Поля могут быть любого базового (стандартного) типа данных, массивом, указателем, объединением или структурой.

4. Для обращения к полю используется операция принадлежности (привязки, выбора) «.» (точка) при обращении через *ID* структуры, или «–>» (стрелка) при обращении через указатель.

5. Структуры одного типа можно присваивать друг другу с использованием стандартной функции *memcpy*.

6. Ввод-вывод структур выполняется поэлементно.

7. Структуры, память под которые выделяет компилятор, можно инициализировать значениями их полей.

8. Файл – это именованный объект, хранящий данные на каком-либо носителе, хотя может располагаться и на электронном диске в ОП.

9. Файл не имеет фиксированной длины, т.е. может увеличиваться или уменьшаться в процессе обработки.

10. Перед работой файл необходимо открыть (функция *fopen*), а после работы закрыть (функция *fclose*).

## ЗАДАНИЕ 6. Создание и обработка структур

***Первый уровень сложности***

Написать программу по обработке массива структур, содержащего следующую информацию о студентах:

– фамилия и инициалы;
– год рождения;
– номер группы;
– оценки за семестр: физика, математика, информатика, химия;
– средний балл.

Организовать ввод исходных данных, средний балл рассчитать по введенным оценкам.

1. Распечатать анкетные данные студентов, сдавших сессию на 8, 9 и 10.

2. Распечатать анкетные данные студентов-отличников, фамилии которых начинаются с интересующей вас буквы.

3. Распечатать анкетные данные студентов-отличников из интересующей вас группы.

4. Распечатать анкетные данные студентов, фамилии которых начинаются с буквы *A* и сдавших математику на 9 и 10.

5. Распечатать анкетные данные студентов интересующей вас группы, имеющих оценку 9 по физике и оценку 10 по высшей математике.

6. Распечатать анкетные данные студентов интересующей вас группы. Фамилии студентов начинаются с букв *В*, *Г* и *Д*.

7. Распечатать анкетные данные студентов, не имеющих оценок 4 и 5 по информатике и математике.

8. Вычислить общий средний балл всех студентов и распечатать список студентов со средним баллом выше общего среднего балла.

9. Вычислить общий средний балл всех студентов и распечатать список студентов интересующей вас группы, имеющих средний балл выше общего среднего балла.

10. Распечатать анкетные данные студентов интересующей вас группы, имеющих оценки 3 и 4.

11. Распечатать анкетные данные студентов интересующей вас группы, имеющих оценку 9 по информатике.

12. Распечатать анкетные данные студентов, имеющих оценку 8 по физике и оценку 9 по высшей математике.

13. Вычислить общий средний балл студентов интересующей вас группы и распечатать список студентов этой группы, имеющих средний балл выше общего среднего.

14. Распечатать анкетные данные студентов-отличников интересующей вас группы.

15. Распечатать анкетные данные студентов интересующей вас группы, имеющих средний балл выше введенного с клавиатуры.

***Второй уровень сложности***

Написать программу предыдущего варианта, создав из предложенных анкетных данных динамический массив введенной с клавиатуры размерности. Полученные данные упорядочить: для символьных данных – по алфавиту (выбрав нужное поле), для числовых данных – по возрастанию (убыванию).

## ЗАДАНИЕ 7. Создание и обработка файлов

***Первый уровень сложности***

Написать программу по обработке файла, состоящего из структур, содержащих информацию задания 6. Средний балл рассчитать программно по введенным оценкам. Массив структур не использовать.

В программе реализовать следующие действия по обработке файла:

– создание;

– просмотр;

– добавление нового элемента;

– удаление (редактирование);

– решение индивидуального задания (первый уровень сложности задания 6).

Результаты выполнения индивидуального задания записать в текстовый файл.

***Второй уровень сложности***

Задачи шифровки. Составить программу, которая вводит строку с клавиатуры; признак окончания ввода – нажатие клавиши *Enter*, шифрует введенный текст в файл на диске по определенному алгоритму. Программа должна считывать эту строку из файла и далее дешифровать текст, выводя его на экран и записывая в выходной файл.

В программе реализовать следующие действия:

– ввод с клавиатуры исходной строки текста и запись в файл *a.txt*;

– считывание строки из файла и вывод на экран;

– шифровка текста;

– расшифровка.

Алгоритмы шифровки:

1. Каждая буква от «а» до «ю» заменяется на следующую по алфавиту, а «я» заменяется на «а».

2. Первая буква «а» заменяется на 11-ю, вторая «б» – на 12-ю, третья – на 13-ю, ... , последняя «я» – на 10-ю.

3. После каждой согласной буквы вставляется буква «а».

4. После каждой согласной буквы вставляется слог «ла».

5. Каждая пара букв «ле» заменяется на «ю», «са» – на «щ», «ик» – на «ж».

6. Каждая из пары букв «си», «ли» и «ти» заменяются соответственно на «иис», «иил» и «иит».

7. После каждой гласной буквы вставляется буква «с».

8. После каждой гласной буквы вставляется слог «ла».

9. Каждая из букв «а», «о», «и» заменяется соответственно на «ц», «ш», «щ».

10. Каждая буква заменяется на следующую в алфавите по часовой стрелке.

11. Каждая буква заменяется на следующую в алфавите против часовой стрелки.

12. Каждая буква «а» заменяется на слог «си», а «и» – на «са».

13. Четные и нечетные символы меняются местами.

14. Символы, кратные двум по порядку следования, заменяются на единицы.

15. Символы, кратные двум по порядку следования, заменяются на свой порядковый номер.

# ГЛАВА 15. Динамические структуры данных

## *15.1. Линейные списки*

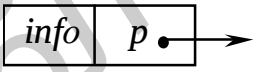
Некоторые задачи исключают использование структур данных фиксированного размера и требуют введения структур, способных увеличивать или уменьшать свой размер уже в процессе работы программы. Основу таких структур составляют динамические переменные.

Динамическая переменная хранится в некоторой области ОП, обращение к которой производится через переменную-указатель.

Как правило, динамические переменные организуются в списковые структуры данных, элементы которых имеют тип ***struct***. Для адресации элементов в структуру включается указатель (адресное поле) на область размещения следующего элемента.

Такой список называют ***однонаправленным*** *(односвязным).* Если добавить в каждый элемент ссылку на предыдущий, получится ***двунаправленный*** *список (двусвязный),* если последний элемент связать указателем с первым, получится ***кольцевой*** *список.*

Например, пусть необходимо создать линейный список, содержащий целые числа, тогда каждый элемент списка должен иметь информационную (*info*) часть, в которой будут находиться данные, и адресную часть (*p*), в которой будут размещаться указатели связей, т.е. элемент такого списка имеет вид




а шаблон структуры будет иметь вид

```
struct Spis {
        int info;
        Spis *p;
} ;
```

Каждый элемент списка содержит *ключ,* идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных.

Например, если создается линейный список из записей, содержащих фамилию, год рождения, стаж работы и пол, любая часть записи может выступать в качестве ключа. При упорядочивании такого списка по алфавиту

ключом будет являться фамилия, а при поиске, например, ветеранов труда – ключом будет стаж работы. Как правило, ключи должны быть уникальными, но могут и совпадать. В случае совпадения ключей лучше всего использовать схемы организации структур данных по принципам «хеширования».

Над списками можно выполнять следующие *операции:*

– начальное формирование списка (создание первого элемента);

– добавление элемента в список;

– обработка (чтение, удаление и т.п.) элемента с заданным ключом;

– вставка элемента в заданное место списка (до или после элемента с заданным ключом);

– упорядочивание списка по ключу.

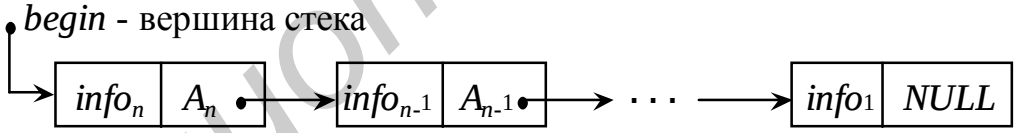
Если программа состоит из функций, решающих вышеперечисленные задачи, то необходимо соблюдать следующие требования:

– все параметры, не изменяемые внутри функций, должны передаваться с модификатором *const*;

– указатели, которые могут изменяться, передаются по адресу. Например, при удалении из списка последнего элемента, измененный указатель на конец списка требует корректировки, т.е. передачи в точку вызова.

### *15.2. Структура данных СТЕК*

Стек – упорядоченный набор данных, в котором размещение новых элементов и удаление существующих производится только с одного его конца, который называют вершиной стека, т.е. стек – это список с одной точкой доступа к его элементам. Графически это можно изобразить так:




Стек – структура типа ***LIFO*** (*Last In, First Out*) – последним вошел, первым выйдет. Стек получил свое название из-за схожести с оружейным магазином с патронами (обойма): когда в стек добавляется новый элемент, то прежний проталкивается вниз и временно становится недоступным. Когда же верхний элемент удаляется из стека, следующий за ним поднимается вверх и становится опять доступным.

Максимальное число элементов стека ограничивается, т.е. по мере вталкивания в стек новых элементов память под него должна динамически запрашиваться и освобождаться также динамически при удалении элемента из стека. Таким образом, стек – динамическая структура данных, состоящая из переменного числа элементов одинакового типа.

Состояние стека рассматривается только по отношению к его вершине, а не по отношению к количеству его элементов, т.е. только вершина стека характеризует его состояние.

Операции, выполняемые над стеком, имеют специальные названия:

*push* – добавление элемента в стек (вталкивание);

*pop* – выталкивание (извлечение) элемента из стека, верхний элемент стека удаляется (не может применяться к пустому стеку).

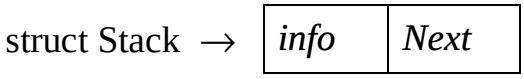
Кроме этих обязательных операций часто нужно прочитать значение элемента в вершине стека, не извлекая его оттуда. Такая операция получила название *peek*.

Рассмотрим основные алгоритмы работы со стеком, взяв для простоты в качестве информационной части целые числа, хотя информационная часть может состоять из любого количества объектов допустимого типа, за исключением файлов.

### *15.2.1. Алгоритм формирования стека*

Рассмотрим данный алгоритм для первых двух элементов.

1. Описание структуры переменной, содержащей информационное и адресное поля:




Шаблон структуры рекомендуется описывать глобально:

```
struct Stack {
            int info;
            Stack *Next;
} ;
```

2. Объявление указателей на структуру:

Stack *begin (вершина стека), *t (текущий элемент);

3. Так как первоначально стек пуст: begin = NULL;

4. Захват памяти под первый (текущий) элемент:

t = (Stack*) malloc (sizeof(Stack)); или t = new Stack;

формируется конкретный адрес ОП (обозначим его *A*1) для первого элемента, т.е. *t* равен *A*1.

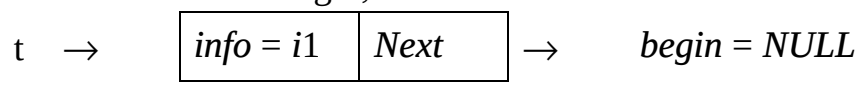
5. Ввод информации (например, *i*1);

а) формирование информационной части:

t -> info = i1;

б) формирование адресной части: значение адреса вершины стека записываем в адресную часть текущего элемента (там был *NULL*)

t -> Next = begin;




6. Вершина стека переносится на созданный первый элемент:

begin = t;

в результате получается следующее:

*begin* (*A*1) → | *info* = *i*1 | *NULL* |

7. Захват памяти под второй элемент:

t = (Stack*) malloc (sizeof(Stack)); или t = new Stack;

формируется конкретный адрес ОП (*A*2) для второго элемента.

8. Ввод информации для второго элемента (*i*2);

а) формирование информационной части:

t -> info = i2;

б) в адресную часть записываем значение адреса вершины, т.е. адрес первого (предыдущего) элемента (*A*1):

t -> Next = begin;

*t* (*A*2) → | *info* = *i*2 | *Next* = *A*1 |

9. Вершина стека снимается с первого и устанавливается на новый элемент (*A*2):

begin = t;

получается следующая цепочка:

*begin* (*A*2) → | *info* = *i*2 | *Next* = *A*1 | → | *info* = *i*1 | *Next* = *NULL* |

Обратите внимание, что действия 7, 8, 9 идентичны действиям 4, 5, 6, т.е. добавление новых элементов в стек можно выполнять в цикле, до тех пор, пока это необходимо.

Функция формирования элемента стека для объявленного ранее типа данных может выглядеть следующим образом:

```
Stack* Create(Stack *begin) {
        Stack *t = (Stack*)malloc(sizeof(Stack));
        printf("\n Input Info  ");
        scanf("%d", &t -> info);
        t -> Next = begin;
        return t;
        }
```

Участок программы с обращением к функции *Create* для добавление необходимого количества элементов в стек может иметь следующий вид:

```
        …
Stack *begin = NULL;
int repeat = 1;
while(repeat)   {                   // repeat=1 – продолжение ввода данных
        begin = Create(begin);
        printf(" Stop - 0 ");       // repeat=0 – конец ввода данных
        scanf("%d", &repeat);
}
        …
```

Если в функцию *Create* указатель на вершину передавать по адресу и использовать для захвата памяти операцию *new*, то она может иметь следующий вид:

```
void Create(Stack **pt) {
        Stack *t = new Stack;
        printf("\n Input Info  ");
        scanf("%d", &t -> info);
        t -> Next = *pt;
}
```

Обращение к ней в данном случае будет:  Create(&begin);

### *15.2.2. Алгоритм извлечения элемента из стека*

В данном алгоритме вершина *begin* не сдвигается.

1. Устанавливаем текущий указатель на вершину стека:  t = begin;

2. Обрабатываем информационную часть текущего элемента (*t ->info*), например, выводим на экран.

3. Переставляем указатель текущего элемента на следующий элемент, адрес которого находится в адресной части текущего:

```
t = t->Next;
```

### *15.2.3. Просмотр стека*

1. Устанавливаем текущий указатель на вершину t = begin.

2. Проверяем, если *begin* равен *NULL*, то стек пуст, выводим сообщение и либо завершаем работу, либо переходим на формирование стека.

3. Если стек не пуст, начинаем выполнять цикл до тех пор, пока текущий указатель *t* не равен *NULL*, т.е. пока не обработаем последний элемент, в адресной части которого находится значение *NULL*.

4. Выводим на экран информационную часть текущего элемента:

```
printf("\n Элемент:  %d", t -> info);    или     cout << t->info;
```

5. Переставляем текущий указатель на следующий элемент:

```
t = t -> Next;
```

6. Конец цикла.

Функция просмотра стека без сдвига его вершины может выглядеть следующим образом:

```
void View(Stack *begin)
{
        Stack *t = begin;
        if(begin == NULL) {
                puts(" Стек пуст! ");
                return;
```

```
        }
        while( t != NULL) {
                printf(“ %d \n”, t->info);
                t = t -> Next;
        }
}
```

Обращение к этой функции: View(begin);

### *15.2.4. Алгоритм освобождения памяти, занятой стеком*

1. Начинаем цикл, выполняющийся пока *begin* не станет равным *NULL*.
2. Устанавливаем текущий указатель на вершину стека: t = begin;
3. Вершину стека переставляем на следующий элемент: begin = t->Next;
4. Уничтожаем текущий (бывшую вершину) элемент, т.е. освобождаем занятую под него память *free*(*t*);

Функция освобождения памяти, занятой стеком, будет выглядеть следующим образом:

```
void Delete_Stack(Stack **begin) {
        Stack *t;
        while( *begin != NULL) {
                t = *begin;
                *begin = (*begin) -> Next;
                free(t);
        }
}
```

Параметром данной функции является указатель на указатель, так как значение вершины стека передается в функцию и должно быть возвращено из нее. Тогда обращение к функции *Delete_Stack* с контролем ее выполнения будет следующим:

```
        Delete_Stack(&begin);
        if(begin == NULL)
                puts(“ Free! ”);
        . . .
```

### *15.2.5. Алгоритм проверки правильности расстановки скобок*

Стек может использоваться для проверки правильности расстановки скобок в арифметическом выражении по следующему правилу: скобки расставлены верно, если число открывающихся и закрывающихся скобок совпадает и каждой открывающейся скобке соответствует закрывающаяся скобка.

При реализации алгоритма анализа исходное выражение просматривается слева направо.

1. Если в выражении обнаружена открывающаяся скобка, то анализируем содержимое стека:

а) если стек пуст или не пуст, но в вершине находится тоже открывающая скобка, то записываем ее в стек;

б) если стек не пуст и в вершине находится закрывающая скобка, то обе скобки выбрасываем из рассмотрения (находящуюся в стеке удаляем).

2. Если обнаружена закрывающая скобка и стек пуст, то выражение составлено неверно, выводим сообщение об этом и завершаем работу.

3. Просмотрев все выражение, проверяем стек и, если он не пуст, то баланс скобок нарушен и выражение составлено неверно, выводим сообщение об этом и завершаем работу.

По такому принципу работают все компиляторы, проверяя баланс круглых скобок в выражениях, баланс фигурных скобок во вложенных блоках, вложенные циклы и т.п.

### *15.3. Структура данных ОЧЕРЕДЬ*

***Очередь*** – упорядоченный набор данных (структура данных), в котором в отличие от стека извлечение данных происходит из начала цепочки, а добавление данных – в конец этой цепочки.

Очередь также называют структурой данных, организованной по принципу ***FIFO*** (*First In, First Out*) – первый вошел (первый созданный элемент очереди), первый вышел.

В языке Си работа с очередью, как и со стеком, реализуется при помощи структур, указателей на структуры и операций динамического выделения и освобождения памяти.

Пример очереди – некоторый механизм обслуживания, который может выполнять заказы только последовательно один за другим. Если при поступлении нового заказа данное устройство свободно, оно немедленно приступит к выполнению этого заказа, если же оно выполняет какой-то ранее полученный заказ, то новый заказ поступает в конец очереди из других ранее пришедших заказов. Когда устройство освобождается, оно приступает к выполнению заказа из начала очереди, т.е. этот заказ удаляется из очереди и первым в ней становится следующий за ним заказ.

Заказы, как правило, поступают нерегулярно и очередь то увеличивается, то укорачивается и даже может оказаться пустой.

При работе с очередью обычно помимо текущего указателя используют еще два указателя, первый указатель устанавливается на начало очереди, а второй – на ее конец.

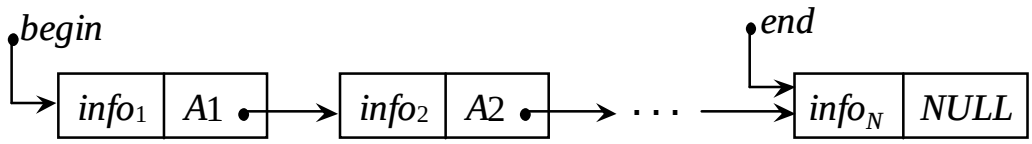
Шаблон элемента структуры, информационной частью которого является целое число, может иметь следующий вид:

```
struct Spis {
        int info;
        Spis *Next;
};
```

При организации очереди обычно используют два указателя

Spis *begin, *end;

где ***begin*** и ***end*** – указатели на начало и конец очереди соответственно, т.е. при создании очереди мы организуем структуру данных следующего вида:




каждый элемент которой имеет информационную *info* и адресную *Next* (*A*1, *A*2, ...) части.

Основные операции с очередью следующие:

– формирование очереди;

– добавление нового элемента в конец очереди;

– удаление элемента из начала очереди.

### *15.3.1. Формирование очереди*

Формирование очереди состоит из двух этапов: создание первого элемента, добавление нового элемента в конец очереди.

***Создание первого элемента очереди***

Этот этап заключается в создании первого элемента, для которого адресная часть должна быть нулевой (*NULL*). Для этого нужно:

1) ввести информацию для первого элемента (целое число *i*);

2) захватить память, используя текущий указатель:

t = (Spis*) malloc(sizeof(Spis)); или t = new Spis;

в результате формируется конкретный адрес (*A*1) для первого элемента;

3) сформировать информационную часть:

t -> info = i; (обозначим *i*1 )

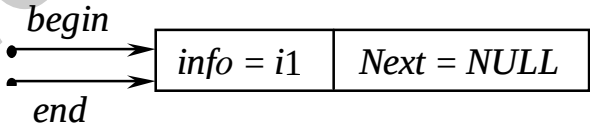
4) в адресную часть занести *NULL*:

t -> Next = NULL;

5) указателям на начало и конец очереди присвоить значение *t*:

begin = end = t;

На этом этапе получим следующее:




***Добавление элемента в очередь***

Рассмотрим алгоритм добавления только для второго элемента.

1. Ввод информации для текущего (второго) элемента – значение *i* .

2. Захватываем память под текущий элемент:

t = (Spis*) malloc (sizeof(Spis)); или t = new Spis;

3. Формируем информационную часть (обозначим *i*2):

t -> info = i;

4. В адресную часть созданного элемента (текущего) заносим *NULL*, т.к. этот элемент становится последним:

```
t -> Next = NULL;
```

5. Элемент добавляется в конец очереди, поэтому в адресную часть бывшего последнего элемента *end* заносим адрес созданного:

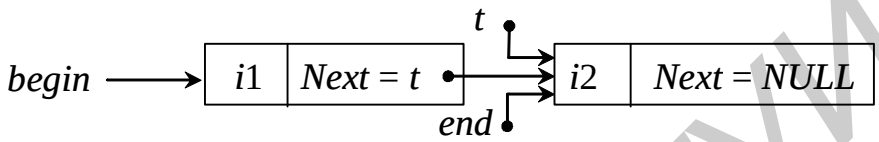
```
end -> Next = t;
```

бывший последний элемент становится предпоследним.

6. Переставляем указатель последнего элемента на добавленный:

```
end = t;
```

В результате получим




Для добавления в очередь любого количества элементов организуется цикл, включающий пункты 1– 6 рассмотренного алгоритма. Завершение цикла реализуется в зависимости от поставленной задачи.

Обобщим рассмотренные этапы, тогда функция формирования очереди из данных объявленного типа с добавлением новых элементов в конец может иметь следующий вид:

```
void Create(Spis **begin, Spis **end)  {
        Spis *t = (Spis*) malloc(sizeof(Spis));
        printf("\n Input Info  ");
        scanf("%d",  &t -> info);
        t -> Next = NULL;
        if(*begin == NULL)              // Формирование первого элемента
                *begin = *end = t;
        else {
                (*end) -> Next = t;     // Добавление в конец
                *end = t;
        }
}
```

Участок программы с обращением к функции *Create* для добавление необходимого количества элементов в очередь может иметь следующий вид:

```
        …
Spis *begin = NULL, *end;
int repeat = 1;
while(repeat)   {               // repeat=1 – продолжение ввода данных
        Create(&begin, &end);
        printf(" Stop - 0 ");   // repeat=0 – конец ввода данных
        scanf("%d", &repeat);
        }
        …
```

### *15.3.2. Алгоритм удаления первого элемента из очереди*

Предположим, что очередь создана, т.е. *begin* не равен *NULL* (рекомендуется организовать проверку на равенство *NULL* с соответствующей обработкой данной ситуации).

1. Устанавливаем текущий указатель на начало очереди: t = begin;
2. Обрабатываем информационную часть первого элемента очереди, например, выводим на экран.
3. Указатель на начало очереди переставляем на следующий (2-й) элемент

```
begin = begin->Next;
```

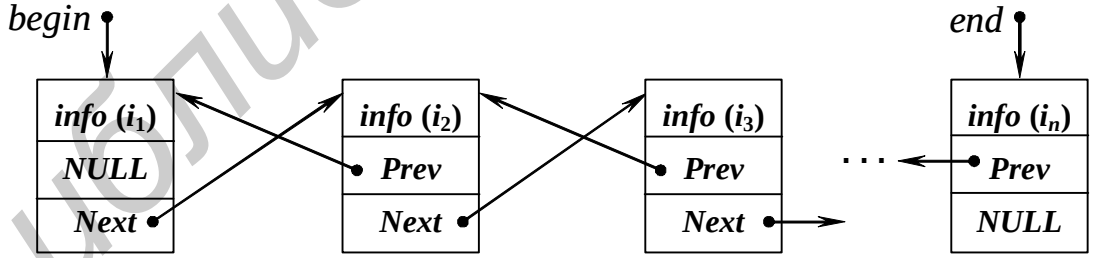
4. Освобождаем память, захваченную под 1-й элемент: free(t);
5. Выводим сообщение, например, «Элемент удален!».

***Алгоритмы просмотра очереди и освобождения памяти*** выполняются аналогично стеку (см. п. 15.2).

### *15.4. Двунаправленный линейный список*

Более универсальным является двунаправленный (двухсвязный) список, в каждый элемент которого кроме указателя на следующий элемент включен и указатель на предыдущий. Для обработки такого списка обычно аналогично очереди используются два указателя – на первый и последний элементы.

Графически такой список выглядит следующим образом:




Введем структуру, в которой (для простоты, как и раньше) информационной частью *info* будут целые числа, а адресная часть состоит из двух указателей на предыдущий (*Prev*) и следующий (*Next*) элементы:

```
struct Spis    {
                int info;
                Spis  *Prev, *Next;
        } ;
```

Для работы со списком декларируем Spis **begin, *end*; – указатели на начало и конец списка соответственно.

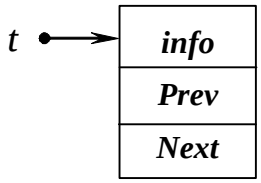
Формирование двунаправленного списка проводится в два этапа – формирование первого элемента и добавление нового. Причем добавление может выполняться как в начало, так и в конец списка.

### *15.4.1. Формирование первого элемента*

1. Захват памяти под текущий элемент:

```
Spis *t = (Spis*) malloc (sizeof(Spis));
```

На данном этапе имеем элемент:




2. Формируем первый элемент списка:

а) формируем информационную часть, например, вводя с клавиатуры:

```
scanf("%d",  &t -> info);        или    cin >> t -> info;
```

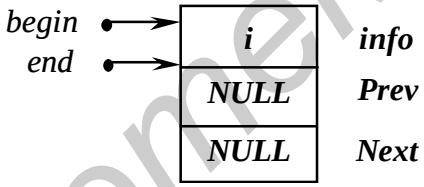
б) формируем адресные части (первоначально это *NULL*):

```
t -> Prev = t -> Next = NULL;
```

в) указатели начала и конца списка устанавливаем на этот элемент:

```
begin = end = t;
```

После выполнения указанных действий получили первый элемент списка:




### *15.4.2. Добавление элементов в конец списка*

1. Начало цикла.
2. Захват памяти под текущий элемент:

```
t = (Spis*) malloc(sizeof(Spis));
```

3. Формирование информационной части:

```
scanf("%d", &t -> info);
```

4. Формирование адресных частей текущего элемента:

а) указателю на следующий элемент (*Next*) присваиваем значение *NULL*, т.к. добавление выполняем в конец, следующего за *t* нет:

```
t -> Next = NULL;
```

б) указателю на предыдущий элемент (*Prev*), используя указатель конца (*end*), присваиваем адрес бывшего последнего элемента:

```
t -> Prev = end;
```

5. Последним элементом был *end*, а должен стать *t*, для этого указатель на следующий элемент последнего в списке (*end*) устанавливаем на созданный (текущий):

```
end -> Next = t;
```

6. Переставляем указатель конца списка на созданный элемент:

```
end = t;
```

7. Продолжаем цикл до тех пор, пока не обработаем признак его завершения.

### *15.4.3. Алгоритм просмотра списка*

| ***С начала*** | ***С конца*** |
|---|---|
| 1. Устанавливаем текущий указатель на: | |
| начало списка t = begin; | конец списка t = end; |
| 2. Начало цикла, работающего до тех пор, пока t != NULL. | |
| 3. Информационную часть текущего элемента t -> info – на печать. | |
| 4. Устанавливаем текущий указатель на: | |
| следующий элемент, адрес которого находится в поле *Next* текущего элемента<br>t = t -> Next; | предыдущий элемент, адрес которого находится в поле *Prev* текущего элемента<br>t = t -> Prev; |

5. Конец цикла.

### *15.4.4. Алгоритм поиска элемента в списке по ключу*

Ключом может быть любое интересующее значение (в зависимости от поставленной задачи). Поэтому уточним задачу: найдем конкретное значение *info* в списке и его порядковый номер.

1. Введем с клавиатуры ключ поиска, т.е. искомое значение *i_p*.

2. Установим текущий указатель на начало списка:

```
t = begin;
```

3. Счетчик элементов *k* = 1;

4. Начало цикла (выполнять пока t != NULL, т.е. не дойдем до конца).

5. Сравниваем информационную часть текущего элемента с искомым:

а) если они совпадают (t -> info = i_p), выводим на экран элемент, его номер *k* и завершаем поиск (*break*);

б) иначе, переставляем текущий указатель на следующий элемент и увеличиваем счетчик *k* на 1:

```
t = t -> Next;
k++;
```

6. Конец цикла.

### *15.4.5. Алгоритм удаления элемента в списке по ключу*

Удалить из списка элемент, информационная часть (ключ) которого совпадает со значением, введенным с клавиатуры.

Решение данной задачи проводим в два этапа – поиск и удаление.

Изменим алгоритм поиска, т.к. в дальнейшем понадобится дополнительный указатель для удаления и добавим контроль на случай отсутствия в списке искомого элемента.

***Первый этап – поиск***

1. Введем дополнительный указатель и присвоим ему значение *NULL*:

```
Spis *key = NULL;
```

2. Введем с клавиатуры искомое значение *i_p* (ключ поиска).

3. Установим текущий указатель на начало списка:

```
t = begin;
```

4. Начало цикла (выполнять пока t != NULL).

5. Сравниваем информационную часть текущего элемента с искомым.

5.1. Если они совпадают (t -> info = i_p), то (выводим на экран сообщение об успехе);

а) запоминаем адрес найденного элемента:

```
key = t;
```

б) завершаем поиск – досрочный выход из цикла (*break*);

5.2. Иначе, переставляем текущий указатель на следующий элемент:

```
t = t -> Next;
```

6. Конец цикла.

7. Контроль, если key = NULL , т.е. искомый элемент не найден, то сообщаем о неудаче и этап удаления не выполняем (*return* или *exit*) .

***Второй этап – удаление***

1. Если найден элемент для удаления, т.е. key != NULL, то удаляем элемент из списка в зависимости от его местонахождения.

2. Если удаляемый элемент находится в начале списка, т.е. key = begin, то создаем новый начальный элемент:

а) указатель начала списка переставляем на следующий (второй) элемент:

```
begin = begin -> Next;
```

б) указателю *Prev* элемента, который был вторым, а теперь стал первым присваиваем значение *NULL*, т.е. предыдущего нет:

```
begin -> Prev = NULL;
```

3. Если удаляемый элемент в конце списка, т.е. *key* равен *end*, то:

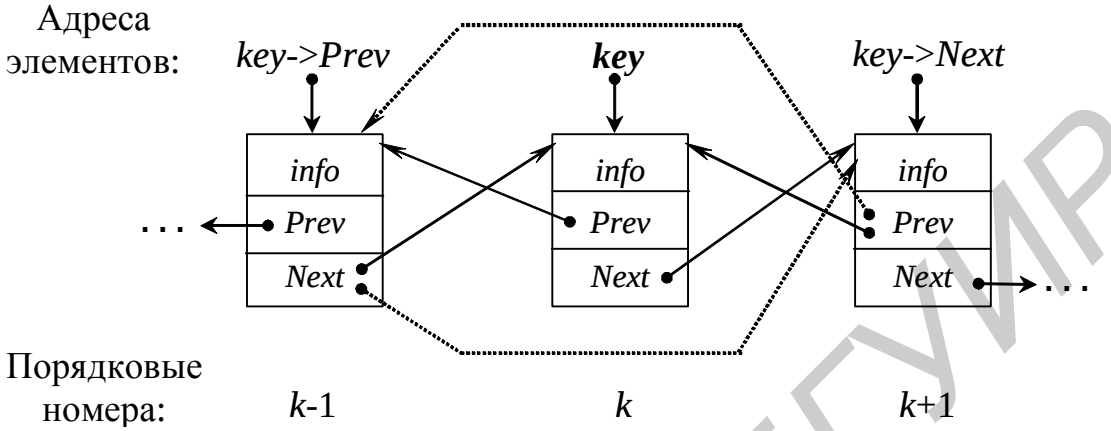
а) указатель конца списка переставляем на предыдущий элемент, адрес которого в поле *Prev* последнего (*end*):

```
end = end -> Prev;
```

б) обнуляем указатель на следующий (*Next*) элемент нового последнего элемента

end -> Next = NULL;

4. Если удаляемый элемент находится в середине списка, нужно обеспечить связь предыдущего и последующего элементов:




а) от *k*-го элемента с адресом *key* обратимся к предыдущему (*k*–1)-му элементу, адрес которого *key->Prev*, и в его поле *Next* [(*key->Prev*)->*Next*] запишем адрес (*k*+1)-го элемента, значение которого *key->Next*:

( key -> Prev ) -> Next = key -> Next;

б) аналогично в поле *Prev* (*k*+1)-го элемента с адресом *key->Next* запишем адрес (*k*-1)-го элемента:

( key -> Next ) -> Prev = key -> Prev;

5. Освобождаем память, занятую удаленным элементом *free*(*key*);

### *15.4.6. Алгоритм вставки элемента в список после элемента с указанным ключом*

Вставить в список элемент после элемента, значение информационной части (ключ) которого совпадает со значением, введенным с клавиатуры.

Решение данной задачи проводится в два этапа – поиск и вставка.

***Первый этап*** аналогичен рассмотренному в алгоритме удаления, а второй проводится только при условии, что искомый элемент найден, т.е. указатель на него *key* не равен *NULL*.

***Этап второй – вставка***

1. Захватываем память под новый элемент

t = (Spis*) malloc(sizeof(Spis));

2. Формируем информационную часть:

scanf(“%d”, &t -> info);

3. Связываем новый элемент с предыдущим

t -> Prev = key;

4. Связываем новый элемент со следующим

t -> Next = key -> Next;

5. Связываем предыдущий элемент с новым

key -> Next = t;

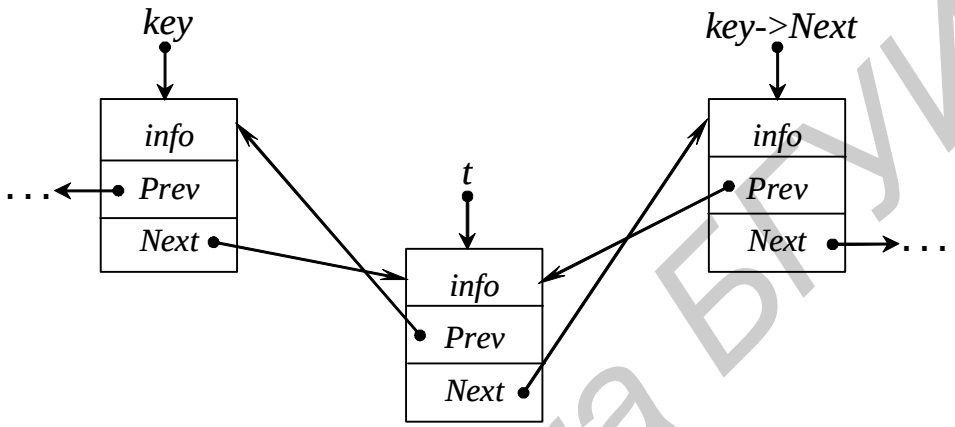
6. Если элемент добавляется не в конец списка (как показано на схеме ниже), т.е. key != end, то

( t -> Next ) -> Prev = t;

7. Иначе, если *key = end*, то указатель *key->Next* равен *NULL* (в п. 4 установлено окончание списка) и новым последним становится *t*

end = t;

Общая схема вставки элемента:

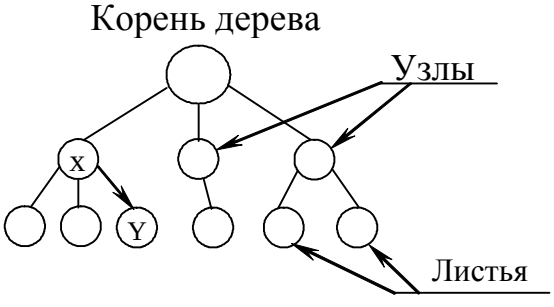



***Алгоритм освобождения памяти, занятой списком***, аналогичен рассмотренному алгоритму для стека (см. разд. 15.2).

### *15.5. Нелинейные структуры данных*

В предыдущих разделах мы рассмотрели линейные структуры динамических списковых данных.

Введение в динамическую переменную двух и более полей-указателей позволяет получить нелинейные структуры данных. Наиболее распространенными являются структуры с иерархическим представлением, которые хорошо изображаются следующим образом:




Такая конструкция данных получила название «***дерево***».

Дерево состоит из элементов, называемых ***узлами*** (вершинами). Узлы соединены между собой направленными дугами. В случае $X \rightarrow Y$ вершина $X$ называется *родителем*, а $Y$ – *сыном* (*дочерью*).

Дерево имеет единственный узел, не имеющий родителей (ссылок на этот узел), который называется ***корнем***. Любой другой узел имеет ровно одного родителя, т.е. на каждый узел дерева имеется ровно одна ссылка.

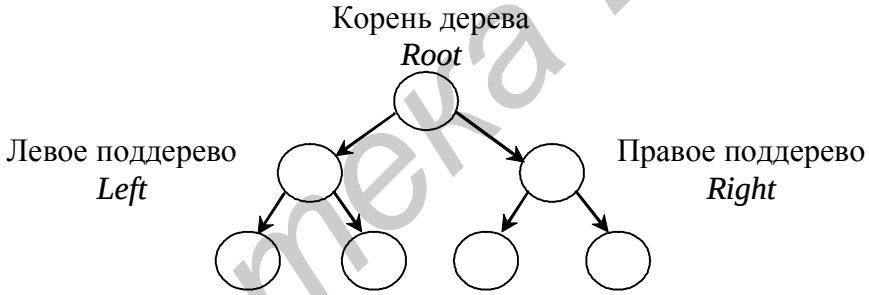
Узел, не имеющий сыновей, называется ***листом***.

***Внутренний*** узел – это узел, не являющийся ни листом, ни корнем. ***Порядок узла*** равен количеству его узлов-сыновей. ***Степень дерева*** – максимальный порядок его узлов. ***Высота (глубина) узла*** равна числу его родителей плюс один. ***Высота дерева*** – это наибольшая высота его узлов.

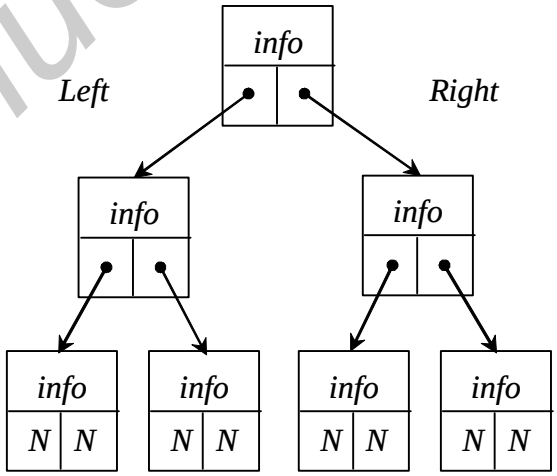
### *15.5.1. Бинарные деревья*

Бинарное дерево – это динамическая структура данных, в которой каждый узел-родитель содержит, кроме данных, не более двух сыновей (левый и правый).

На рисунке приведен пример бинарного дерева (корень обычно изображается сверху, хотя изображение можно и перевернуть).




Такая структура данных организуется следующим образом ($N$ – *NULL*):




Высота дерева, как и раньше, определяется количеством уровней, на которых располагаются его узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его

правого поддерева – больше, оно называется деревом поиска. Одинаковые ключи здесь не допускаются.

Представление динамических данных в виде древовидных структур оказывается довольно удобным и эффективным для решения задач быстрого поиска информации.

*Сбалансированными*, или *AVL*-деревьями, называются деревья, для каждого узла которых высóты его поддеревьев различаются не более чем на 1. Причем этот критерий обычно называют *AVL*-сбалансированностью в отличие от идеальной сбалансированности, когда для каждого узла дерева количество узлов в его левом и правом поддеревьях различаются не более чем на единицу [44].

Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. В связи с этим действия с такими структурами чаще всего описываются с помощью рекурсивных алгоритмов.

### *15.5.2. Основные алгоритмы работы с бинарным деревом*

В общем случае при работе с такими структурами необходимо уметь:

– построить (создать) дерево;

– вставить новый элемент;

– обойти все элементы дерева, например, для просмотра или выполнения некоторой операции;

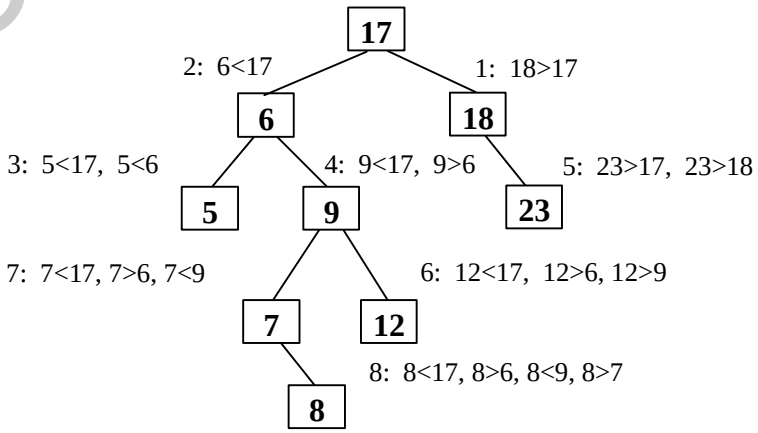
– выполнить поиск элемента с указанным значением в узле;

– удалить заданный элемент.

Обычно бинарное дерево строится сразу упорядоченным, т.е. узел левого сына имеет значение меньшее, чем значение родителя, а узел правого сына – большее.

### *15.5.3. Формирование дерева*

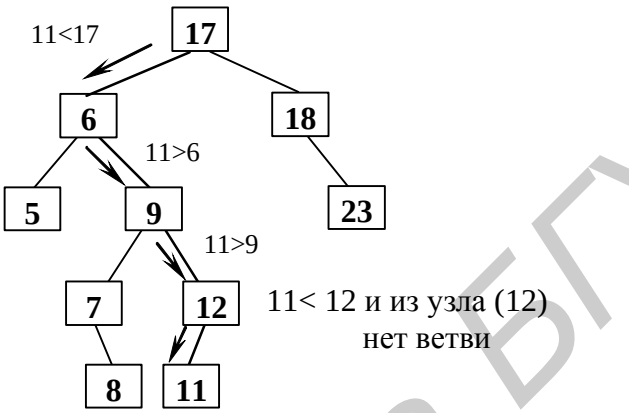
Например, имеется последовательность ключей 17, 18, 6, 5, 9, 23, 12, 7, 8, тогда построенное по ним дерево будет выглядеть следующим образом:



### *15.5.4. Вставка нового элемента*

Для того чтобы вставить новый элемент в дерево, необходимо найти для него место. Для этого, начиная с корня, сравниваем значения узлов (*Tree->info*) со значением нового элемента (*b*). Если *b* < *info*, то идем по левой ветви, в противном случае – по правой ветви. Когда дойдем до узла, из которого не выходит нужная ветвь для дальнейшего поиска, это означает, что место под новый элемент найдено.

Путь поиска места в построенном дереве для числа **11**:




Функция создания дерева, ключами которого являются ***целые положительные*** числа, может иметь следующий вид:

```
Tree* Make(Tree *Root)  {
        Tree *Prev, *t;      // Prev родитель (предыдущий) текущего элемента
        int b, find;
        if ( Root == NULL )    {        // Если дерево не создано
                printf("\n Input Root :  ");
                scanf("%d", &b);
                Root = List(b);              // Установили указатель на корень
        }
//================ Добавление элементов ================
        while(1) {
                printf("\n Input Info :  ");  scanf("%d", &b);
                if (b<0) break;                      // Признак выхода число < 0
                t = Root;                            // Текущий указатель на корень
                find = 0;                            // Признак поиска
                while ( t && ! find) {
                        Prev = t;
                        if( b == t->info)
                                find = 1;            // Ключи должны быть уникальны
                        else
                                if ( b < t -> info ) t = t -> Left;
                                        else   t = t -> Right;
                }
```

```
            if (!find) {                        // Нашли место с адресом Prev
                t = List(b);                    // Создаем новый узел
                if ( b < Prev -> info )         // и присоединяем его, либо
                    Prev -> Left = t;           // на левую ветвь,
                else    Prev -> Right = t;      // либо на правую ветвь
            }
        }       // Конец цикла
        return Root;
}
```

Функция *List* предназначена для создания нового элемента – листа:

```
Tree* List(int i)  {
        Tree *t = (Tree*) malloc (sizeof(Tree));
        t -> info = i;
        t -> Left = t -> Right = NULL;
        return t;
}
```

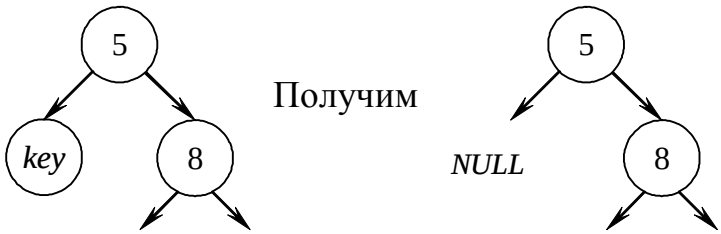
Участок кода с обращением к функции *Create* будет иметь следующий вид:

```
            …
        struct Tree {                   // Декларация шаблона
            int info;
            Tree *Left, *Right;
            };
void main()
{
        Tree *Root = NULL;              // Указатель корня
            …
        Root = Make(Root);
            …
```

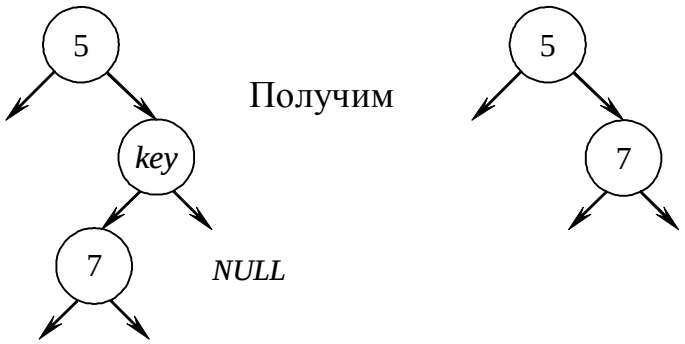
### *15.5.5. Удаление узла*

При удалении узла из дерева возможны три ситуации в зависимости от того, сколько сыновей (потомков) имеет удаляемый узел.

1. Удаляемый узел является листом – просто удаляем ссылку на него. Приведем пример схемы удаления листа с ключом *key*:

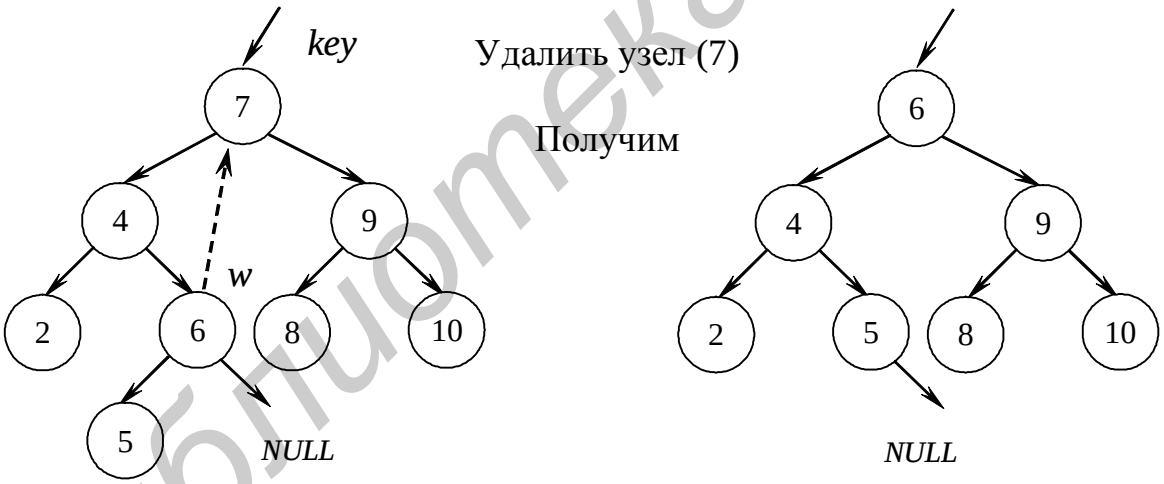


2. Удаляемый узел имеет только одного потомка, т.е. из удаляемого узла выходит ровно одна ветвь. Пример схемы удаления узла *key*, имеющего одного сына:

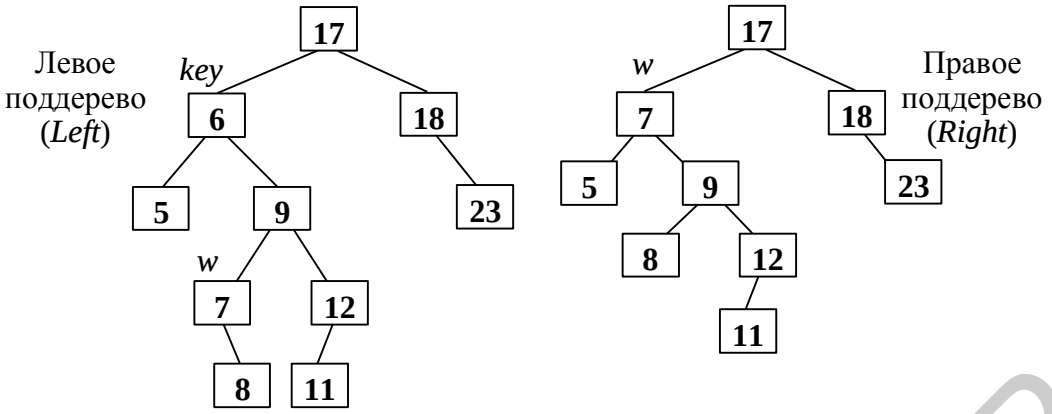



3. Удаление узла, имеющего двух сыновей, значительно сложнее рассмотренных выше. Если *key* – удаляемый узел, то его следует заменить узлом *w*, который содержит либо наибольший ключ (самый правый, у которого указатель *Right* равен *NULL*) в левом поддереве, либо наименьший ключ (самый левый, у которого указатель *Left* равен *NULL*) в правом поддереве.

Используя первое условие, находим узел *w*, который является самым правым узлом поддерева *key*, у него имеется только левый сын:




В построенном ранее дереве удалим узел *key* (6). Используем второе условие, т.е. ищем самый левый узел в правом поддереве – это узел *w* (указатель *Left* равен *NULL*):




Функция удаления узла по заданному ключу ***key*** может иметь вид

```
Tree* Del(Tree *Root, int key)  {
        Tree *Del, *Prev_Del, *R, *Prev_R;
// Del, Prev_Del – удаляемый элемент и его предыдущий (родитель);
// R, Prev_R – элемент, на который заменяется удаленный, и его родитель;
    Del = Root;
    Prev_Del = NULL;
// ===== Поиск удаляемого элемента и его родителя по ключу key =====
    while (Del != NULL && Del -> info != key) {
        Prev_Del = Del;
        if (Del->info > key)  Del = Del->Left;
            else Del = Del->Right;
        }
    if (Del == NULL) {              // Элемент не найден
        puts("\n NO Key!");
        return Root;
    }
// ============ Поиск элемента R для замены ================
    if (Del -> Right == NULL) R = Del->Left;
    else
        if (Del -> Left == NULL) R = Del->Right;
        else {
// Ищем самый правый узел в левом поддереве
            Prev_R = Del;
            R = Del->Left;
            while (R->Right != NULL) {
                Prev_R = R;
                R = R->Right;
            }
// Нашли элемент для замены R и его родителя Prev_R
            if( Prev_R == Del)
                        R->Right = Del->Right;
            else {
```

```
                    R->Right = Del->Right;
                    Prev_R->Right = R->Left;
                    R->Left = Prev_R;
                }
            }
        if (Del== Root) Root = R;        // Удаляя корень, заменяем его на R
        else
// Поддерево R присоединяем к родителю удаляемого узла
        if (Del->info < Prev_Del->info)  Prev_Del->Left = R;    // на левую ветвь
            else    Prev_Del->Right = R;                        // на правую ветвь
        printf("\n Delete element  %d \n", Del->info);
        free(Del);
        return Root;
}
```

Участок программы с обращением к данной функции будет иметь вид

```
        …
    printf("\n Input Del Info :  ");
    scanf(“%d”, &key);
    Root = Del(Root, key);
        …
```

### *15.5.6. Алгоритмы обхода дерева*

Существуют три алгоритма обхода деревьев, которые естественно следуют из самой структуры дерева.

1. Обход слева направо: *Left-Root-Right* (сначала посещаем левое поддерево, затем – корень и, наконец, правое поддерево).

2. Обход сверху вниз: *Root-Left-Right* (посещаем корень до поддеревьев).

3. Обход снизу вверх: *Left-Right-Root* (посещаем корень после поддеревьев).

Интересно проследить результаты этих трех обходов на примере записи формулы в виде дерева, так как они и позволяют получить различные формы записи арифметических выражений.

Пусть для операндов *A* и *B* выполняется операция сложения. Привычная форма записи в виде *A+B* называется ***инфиксной.*** Форма записи, в которой знак операции следует перед операндами *+AB*, называется ***префиксной***, если же операция записывается после операндов *AB+* – ***постфиксной***.

Рассмотрим небольшой пример, пусть задано выражение *A+B*C*. Так как умножение имеет более высокий приоритет, то данное выражение можно переписать в виде *A+*(*B*C*). Для записи выражения в постфиксной форме сначала преобразуем ту часть выражения, которая вычисляется первой, в результате получим: *A+*(*BC**).

Теперь запишем в постфиксной форме операцию сложения между операндами A и (*BC**): *ABC*+*.

Таким образом, выражение *A+B*C* в постфиксном виде *ABC**+, префиксная форма записи будет иметь вид *+*ABC*.

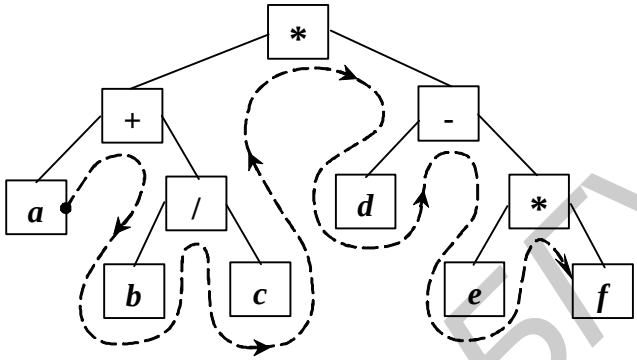
Рассмотрим различные обходы дерева на примере формулы: ((*a*+*b*/*c*)*(*d*–*e***f* )). Дерево формируется по принципу:

– в корне размещаем операцию, которая выполнится последней;

– далее узлы-операции, операнды – листья дерева.

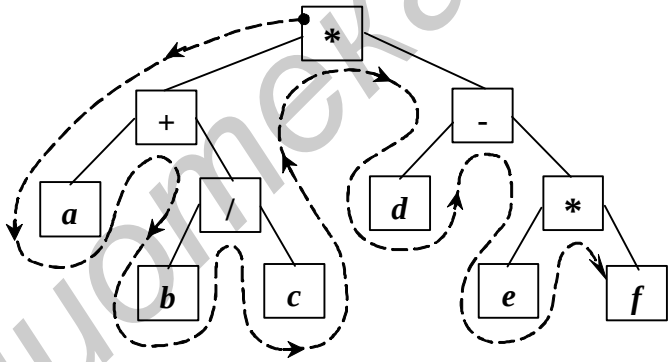
***Обход 1*** (*Left-Root-Right*) дает обычную инфиксную запись выражения (без скобок):

*a* + *b* / *c* * *d* – *e* * *f* .



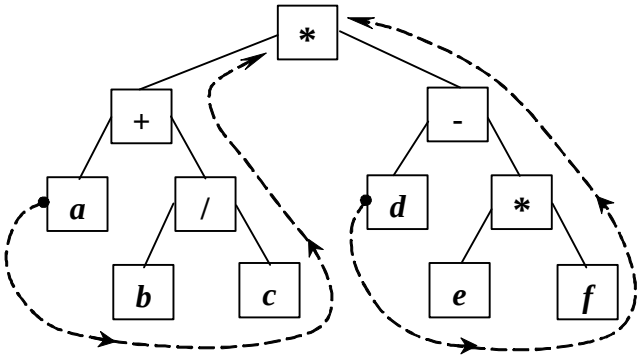

***Обход 2*** (*Root-Left-Right*) – префиксную запись выражения (без скобок):

* + *a* / *b c* – *d* * *e f* .




***Обход 3*** (*Left-Right-Root*) – постфиксную запись выражения:

*a b c* / + *d e f* * – * .




### *15.5.7. Функция просмотра*

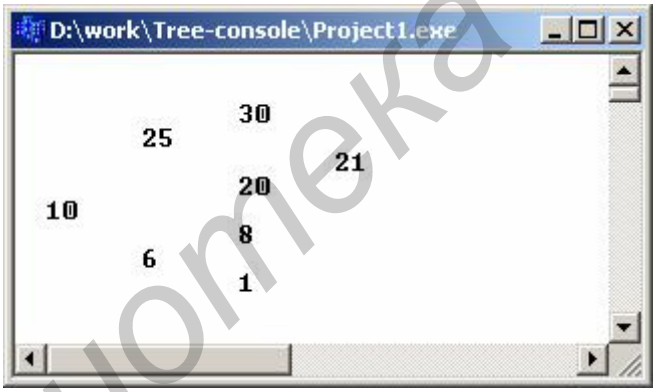
Приведем простой пример функции вывода элементов (ключей) дерева, использующий правила обхода 2.

```
void View ( Tree *t, int level )  {
        if ( t ) {
                View ( t -> Right , level+1);              // Вывод правого поддерева
                for ( int i=0; i<level; i++)  printf("    ");
                printf(“ %d\n”, t -> info);
                View( t -> Left , level+1);                // Вывод левого поддерева
        }
}
```

Обращение к функции *View* будет иметь вид View(Root, 0);

Функция *View* рекурсивная, вторым ее параметром является переменная, определяющая, на каком уровне (*level*) находится узел. Корень находится на уровне «0». Значения узлов выводятся по горизонтали так, что корень находится слева. Перед значением узла для имитации структуры дерева выводится количество пробелов, пропорциональное уровню узла. Если закомментировать цикл печати пробелов, значения ключей будут выведены просто в столбик.

Для последовательно введенных ключей 10 (корень), 25, 20, 6, 21, 8, 1, 30, будет построено дерево, вывод которого на экран с помощью функции *View* будет иметь следующий вид:




### *15.5.8. Освобождение памяти*

Функция освобождения памяти, занятой элементами дерева, может быть реализована аналогично рекурсивной функции *View*

```
void Del_All(Tree *t)  {
    if ( t != NULL)  {
            Del_All ( t -> Left);
            Del_All ( t -> Right);
            free(t);
    }
}
```

## *15.6. Построение обратной польской записи*

Сложные вычислительные задачи обычно требуют больших объемов вычислений, поэтому к разработчикам языков программирования было предъявлено требование: максимально приблизить форму записи математических выражений в коде программы к естественному языку математики.

Одну из первых областей системного программирования составили исследования способов трансляции математических выражений.

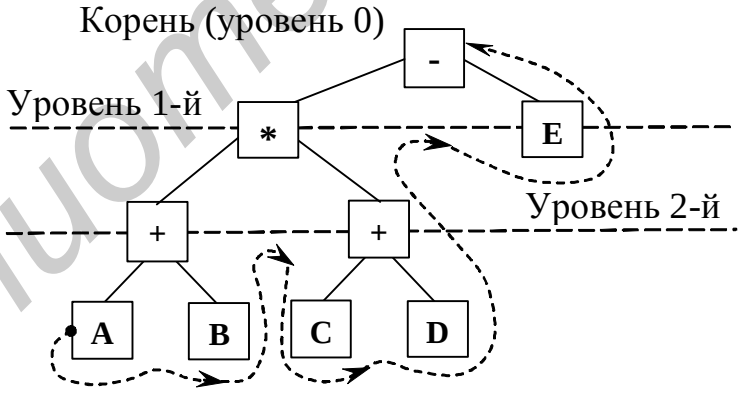
В результате наибольшее распространение получил метод трансляции при помощи обратной польской записи, которую предложил польский математик Я. Лукашевич.

Рассмотрим алгоритмы получения обратной польской записи с использованием структур в виде дерева и стека.

### *15.6.1. Алгоритм, использующий дерево*

Данный алгоритм основан на представлении математического выражения в виде дерева и использовании третьего способа его обхода (см. п. 15.5.6). Напомним его на примере арифметического выражения (A+B)*(C+D)–E.

Представим это выражение в виде дерева, в котором узлам соответствуют операции, а листьям – операнды. Построение начинается с корня, в качестве которого выбирается последняя выполняемая операция. Левой ветви соответствует левый операнд операции, а правой ветви – правый. Дерево выражения имеет вид:




Совершим обход дерева, под которым понимается формирование строки из символов узлов и ветвей дерева. Обход будем совершать от самой левой ветви вправо и узел переписывать в выходную строку только после рассмотрения всех его ветвей. Обход совершаем строго по уровням:

1) уровень 2: **AB+CD+**
2) поднялись на уровень 1: ***E**
3) и, наконец, корень: **–**

В результате такого обхода получили обратную польскую запись:

**AB+CD+*E–** (15.1)

### *15.6.2. Алгоритм, использующий стек*

Получение обратной польской записи с использованием стека может осуществляться весьма просто на основе алгоритма, предложенного Дейкстрой, который ввел понятие стекового приоритета операций:

| Операция | Приоритет |
|---|---|
| ( | 1 |
| + – | 2 |
| * / | 3 |

*Суть алгоритма в следующем*. Просматривается исходная строка символов *S* слева направо, операнды переписываются в выходную строку *B*, а знаки операций заносятся в стек, который первоначально пуст, на основе следующих правил:

1) если в строке *S* встретился операнд, то помещаем его в строку *B*;

2) если в *S* встретилась открывающая скобка, то помещаем ее в стек;

3) если в *S* встретилась закрывающая скобка, то выталкиваем из стека в строку *B* все операции до открывающей скобки, саму отрывающую скобку также извлекаем из стека; обе скобки (открывающая и закрывающая) игнорируются;

4) если в *S* встретилась операция *X*, то выталкиваем из стека все операции, приоритет которых не ниже *X*, после чего операцию *X* записываем в стек;

5) при достижении конца строки *S*, если стек не пуст, переписываем его элементы в выходную строку *B*.

Обратная польская запись обладает рядом замечательных свойств, которые превращают ее в идеальный промежуточный язык при трансляции.

***Во-первых***, вычисление выражения, записанного в обратной польской записи, может проводиться путем однократного просмотра, что является весьма удобным при генерации объектного кода программ.

Например, вычисление полученного выражения (15.1) может быть проведено следующим образом:

| Шаг | Анализируемая строка | Действие |
|---|---|---|
| 1 | AB+CD+*E– | *R*1=A+B |
| 2 | *R*1CD+*E– | *R*2=C+D |
| 3 | *R*1 *R*2*E– | *R*1=*R*1**R*2 |
| 4 | *R*1 E– | *R*1=*R*1–E |
| 5 | *R*1 | |

Здесь *R*1 и *R*2 – вспомогательные переменные.

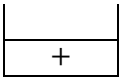
### *15.6.3. Пример реализации*

Пусть задано выражение *a*+*b***c*+(*d***e*+*f*)**g*. Необходимо записать это выражение в постфиксной форме. Правильным ответом будет выражение *abc**+*de***f*+*g**+. Решаем эту задачу, используя стек.

Пусть исходная информация хранится в строке *S*=”*a*+*b***c*+(*d***e*+*f*)**g*”. Результат будем получать в строке *B*.

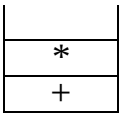
Начинаем последовательно просматривать символы исходной строки, причем стек пуст и *B* – пустая строка.

Букву «*a*» помещается в строку *B*, а операцию «+» помещаем в стек. Букву «*b*» помещаем в строку *B*. На этот момент стек и строка *B* выглядят следующим образом:




*B* = ”*ab*”

Операцию «*» помещаем в стек, т.к. элемент в вершине стека имеет более низкий приоритет. Букву «*c*» помещаем в строку *B*, после чего имеем

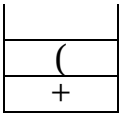



*B* = ”*abc*”

Следующий символ строки *S* «+». Анализируем стек и видим, что элемент в вершине стека «*» и следующий за ним «+» имеют приоритеты не ниже текущего, следовательно, обе операции извлекаем из стека и помещаем в строку *B*, а текущий элемент помещаем в стек. В итоге имеем

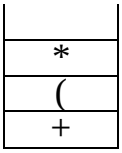

*B* = ”*abc**+”

Далее в строке *S* следует символ «(», его помещаем в стек, а букву «*d*» помещаем в строку *B*, в результате получается



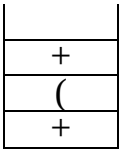

*B* = ”*abc**+*d*”

Следующий в строке *S* символ «*». Так как открывающую скобку нельзя извлечь из стека до тех пор, пока не встретилась закрывающая, то «*» помещаем в стек. Букву «*e*» помещаем в строку *B*:



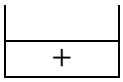

*B* = ”*abc**+*de*”

Следующий прочитанный символ «+», и т.к. элемент стека «*» имеет более высокий приоритет, то извлекаем его из стека и помещаем в строку *B*, а текущий символ «+» помещаем в стек. Символ «*f*» помещаем в строку *B*:



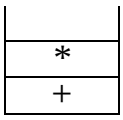

*B* = ”*abc**+*de***f*”

Далее в строке *S* идет закрывающая скобка, все элементы стека до символа «)» помещаем в строку *B* (это элемент «+»), а сам символ «(» извлекаем из стека. Обе скобки игнорируются:




*B = ”abc*+de*f+”*

Операцию «*» помещаем в стек, а букву «*g*» – в строку *B*:




B = ”abc*+de*f+g”

Все символы строки *S* рассмотрены, следовательно, анализируем состояние стека, если он не пуст, то переписываем все его элементы в строку *B*:

*B = ”abc*+de*f+g*+”*

Таким образом, просмотрев исходную информацию только один раз, мы решили поставленную задачу.

Текст программы, реализующий рассмотренный алгоритм, может иметь следующий вид:

```
        . . .
    struct Stack {
        char c;                     // Символ операции
        Stack *Next;
    } ;
int Prior (char);
Stack* InS( Stack*,char);
Stack* OutS( Stack*,char*);
void main ()
{
    Stack *t, *Op = NULL;           // Стек операций Op – пуст
    char a, In[50], Out[50];        // Входная In и выходная Out строки
    int k = 0, l = 0;               // Текущие индексы для строк
    puts(" Input formula : ");  gets(In);
    while ( In[k] != '\0') {        // Анализируем символы строки In
// Если символ «)», выталкиваем из стека в выходную строку все операции
        if ( In[k] == ')' ) {
            while ( (Op -> c) != '(' ) {    // до открывающей скобки
                Op = OutS(Op,&a);           // Считываем элемент из стека
                if ( !Op ) a = '\0';
                Out[l++] = a;               // и записываем в строку Out.
            }
            t = Op;             // Удаляем из стека открывающую скобку
            Op = Op -> Next;
            free(t);
        }
// Если символ строки In – буква, заносим ее в строку Out
```

```
            if ( In[k] >= 'a' && In[k] <= 'z' )   Out[l++] = In[k];
// Если символ – открывающая скобка, записываем ее в стек
            if ( In[k] == '(' )   Op = InS(Op,In[k]);
/* Если символ – знак операции, переписываем из стека в строку Out все операции с большим или равным приоритетом */
            if ( In[k] == '+' || In[k] == '–' || In[k] == '*'  || In[k] == '/' ) {
                  while ( Op != NULL && Prior (Op -> c) >= Prior (In[k]) ) {
                        Op = OutS(Op,&a);       // Извлекаем из стека символ
                        Out[l++] = a;           // и записываем в строку Out
                  }
                  Op = InS(Op,In[k]);           // Текущий символ – в стек
            }
            k++;
      }                                 // Конец цикла анализа входной строки
// Если стек не пуст, переписываем все операции в выходную строку
      while ( Op !=NULL) {
            Op = OutS(Op,&a);
            Out[l++] = a;
      }
      Out[l] = '\0';
      printf("\n Polish = %s", Out);          // Выводим на экран результат
}
//======= Функция реализации приоритета операций =========
int Prior ( char a )  {
      switch ( a ) {
            case '*':      case '/':      return 3;
            case '–':      case '+':      return 2;
            case '(':   return 1;
      }
      return 0;
}
// ============= Добавление элемента в стек =============
Stack* InS( Stack *t, char s)  {
      Stack *t1 =  (Stack*) malloc(sizeof(Stack));
      t1 -> c  = s;
      t1-> Next = t;
      return t1;
}
// ============= Извлечение элемента из стека ==============
Stack* OutS( Stack *t, char *s ) {
      Stack *t1 = t;
      *s = t -> c;
      t = t->Next;
      free(t1);
      return t;
```

}

## *15.7. Понятие хеширования*

Для решения задачи поиска необходимого элемента среди данных большого объема был предложен алгоритм ***хеширования*** (*hashing* – перемешивание), при котором создаются ключи, определяющие данные массива и на их основании данные записываются в таблицу, названную ***хеш-таблицей***. Ключи для записи определяются при помощи функции ***i = h(key)***, называемой ***хеш-функцией***. Алгоритм хеширования определяет положение искомого элемента в хеш-таблице по значению его ключа, полученного хеш-функцией.

Понятие ***хеширования –*** это разбиение общего (базового) набора уникальных ключей элементов данных на непересекающиеся наборы с определенным свойством.

Возьмем, например, словарь или энциклопедию. В этом случае буквы алфавита могут быть приняты за ключи поиска, т.е. основным элементом алгоритма хеширования является ***ключ*** (*key*). В большинстве приложений ключ обеспечивает косвенную ссылку на данные.

Фактически хеширование – это специальный метод адресации данных для быстрого поиска нужной информации ***по ключам***.

Если базовый набор содержит *N* элементов, то его можно разбить на $2^N$ различных подмножеств.

### *15.7.1. Хеш-таблица и хеш-функции*

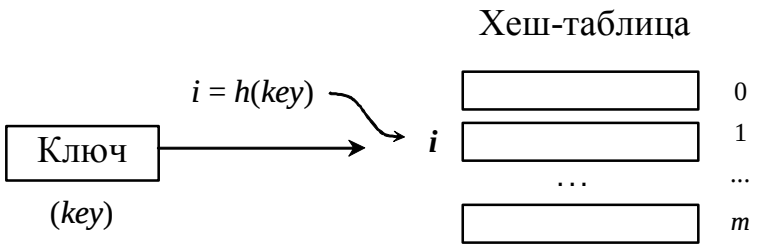
Функция, отображающая ключи элементов данных во множество целых чисел (индексы в таблице – ***хеш-таблица***), называется ***функцией хеширования***, или ***хеш-функцией***:

***i = h(key);***

где *key* – преобразуемый ключ, *i* – получаемый индекс таблицы, т.е. ключ отображается во множество целых чисел (***хеш-адреса***), которые впоследствии используются для доступа к данным.

Однако хеш-функция для нескольких значений ключа может давать одинаковое значение позиции *i* в таблице. Ситуация, при которой два или более ключа получают один и тот же индекс (хеш-адрес), называется ***коллизией*** при хешировании.

Хорошей хеш-функцией считается такая функция, которая минимизирует коллизии и распределяет данные равномерно по всей таблице, а совершенной хеш-функцией – функция, которая не порождает коллизий:




Разрешить коллизии при хешировании можно двумя методами:
– методом открытой адресации с линейным опробыванием;
– методом цепочек.

***Хеш-таблица***

Хеш-таблица представляет собой обычный массив с необычной адресацией, задаваемой хеш-функцией.

***Хеш-структуру*** считают обобщением массива, который обеспечивает быстрый прямой доступ к данным по индексу.

Имеется множество схем хеширования, различающихся как выбором удачной функции *h*(*key*), так и алгоритма разрешения конфликтов. Эффективность решения реальной практической задачи будет существенно зависеть от выбираемой стратегии.

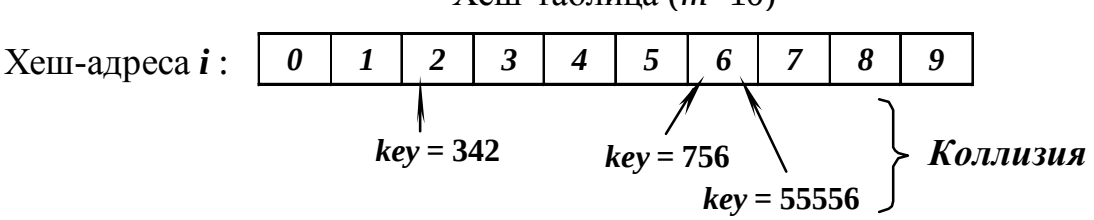
### *15.7.2. Примеры хеш-функций*

Выбираемая хеш-функция должна легко вычисляться и создавать как можно меньше коллизий, т.е. должна равномерно распределять ключи на имеющиеся индексы в таблице. Конечно, нельзя определить, будет ли некоторая конкретная хеш-функция распределять ключи правильно, если эти ключи заранее не известны. Однако, хотя до выбора хеш-функции редко известны сами ключи, некоторые свойства этих ключей, которые влияют на их распределение, обычно известны. Рассмотрим наиболее распространенные методы задания хеш-функции.

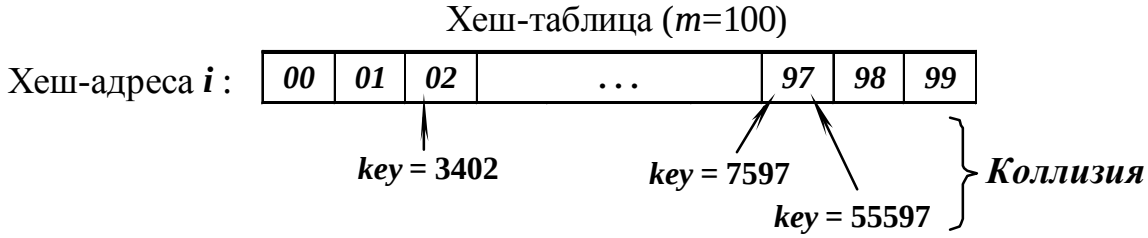
***Метод деления***. Исходными данными являются – некоторый целый ключ *key* и размер таблицы *m*. Результатом данной функции является остаток от деления этого ключа на размер таблицы. Общий вид функции:

```
int h(int key, int m) {
        return key % m;          // Значения
    }
```

Для *m* = 10 хеш-функция возвращает младшую цифру ключа.




Для *m* = 100 хеш-функция возвращает две младшие цифры ключа.




***Аддитивный метод***, в котором ключом является символьная строка. В хеш-функции строка преобразуется в целое суммированием всех символов и возвращается остаток от деления на *m* (обычно размер таблицы *m* = 256).

```
int h(char *key, int m) {
        int s = 0;
        while(*key)
                s += *key++;
        return s % m;
}
```

Коллизии возникают в строках, состоящих из одинакового набора символов, например, *abc* и *cab*.

Данный метод можно несколько модифицировать, получая результат, суммируя только первый и последний символы строки-ключа.

```
int h(char *key, int m) {
        int  len = strlen(key), s = 0;
        if(len < 2)             // Если длина ключа равна 0 или 1,
                s = key[0];  // возвратить key[0]
        else
                s = key[0] + key[len–1];
        return s % m;
}
```

В этом случае коллизии будут возникать только в строках, например, *abc* и *amc*.

***Метод середины квадрата***, в котором ключ возводится в квадрат (умножается сам на себя) и в качестве индекса используются несколько средних цифр полученного значения.

Например, ключом является целое 32-битное число, а хеш-функция возвращает средние 10 бит его квадрата:

```
int h(int key) {
        key *= key;
        key >>= 11;                     // Отбрасываем 11 младших бит
        return key % 1024;              // Возвращаем 10 младших бит
}
```

***Метод исключающего ИЛИ*** для ключей-строк (обычно размер таблицы *m*=256). Этот метод аналогичен аддитивному, но в нем различаются

схожие слова. Метод заключается в том, что к элементам строки последовательно применяется операция «исключающее ИЛИ».

В ***мультипликативном методе*** дополнительно используется случайное действительное число *r* из интервала [0,1), тогда дробная часть произведения *r*key* будет находиться в интервале [0,1]. Если это произведение умножить на размер таблицы *m*, то целая часть полученного произведения даст значение в диапазоне от 0 до *m*–1.

```
int h(int key, int m) {
        double r = key * rnd();
        r = r – (int)r;                  // Выделили дробную часть
        return r * m;
    }
```

В общем случае при больших значениях *m* индексы, формируемые хеш-функцией, имеют большой разброс. Более того, математическая теория утверждает, что распределение получается более равномерным, если *m* является простым числом.

В рассмотренных примерах хеш-функция *i = h*(*key*) только определяет позицию, начиная с которой нужно искать (или первоначально – поместить в таблицу) запись с ключом *key*. Поэтому схема хеширования должна включать ***алгоритм решения конфликтов***, определяющий порядок действий, если позиция *i = h*(*key*) оказывается уже занятой записью с другим ключом.

### *15.7.3. Схемы хеширования*

В большинстве задач два и более ключей хешируются одинаково, но они не могут занимать в хеш-таблице одну и ту же ячейку. Существуют два возможных варианта: либо найти для нового ключа другую позицию, либо создать для каждого индекса хеш-таблицы отдельный список, в который помещаются все ключи, преобразованные в этот индекс.

Эти варианты и представляют собой две классические схемы:

– хеширование методом цепочек (со списками), или так называемое многомерное хеширование – *chaining with separate lists*;

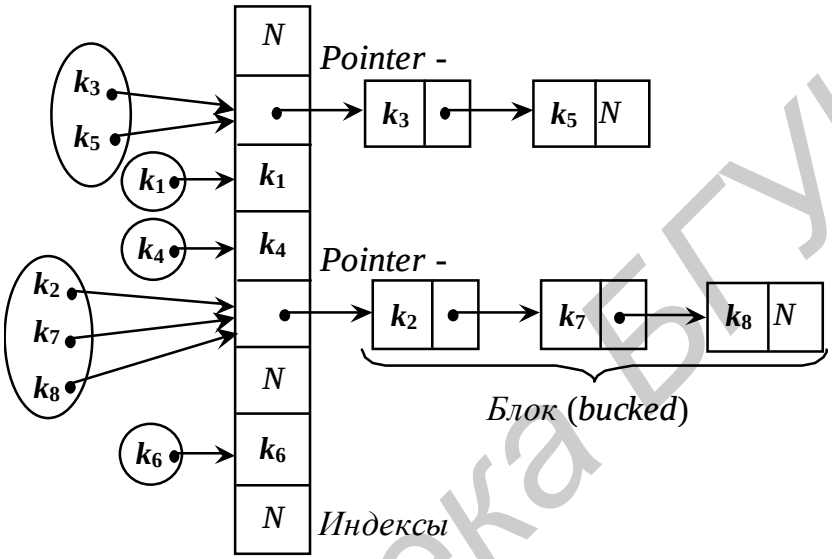
– хеширование методом открытой адресации с линейным опробыванием – *linear probe open addressing*.

***Метод открытой адресации с линейным опробыванием***. Изначально все ячейки хеш-таблицы, которая является обычным одномерным массивом, помечены как не занятые. Поэтому при добавлении нового ключа проверяется, занята ли данная ячейка. Если ячейка занята, то алгоритм осуществляет осмотр по кругу до тех пор, пока не найдется свободное место («открытый адрес»), т.е. либо элементы с однородными ключами размещают вблизи полученного индекса, либо осуществляют двойное хеширование, используя для этого разные, но взаимосвязанные хеш-функции.

В дальнейшем, осуществляя поиск, сначала находят по ключу позицию *i* в таблице, и, если ключ не совпадает, то последующий поиск

осуществляется в соответствии с алгоритмом разрешения конфликтов, начиная с позиции *i* по списку.

***Метод цепочек*** используется чаще предыдущего. В этом случае полученный хеш-функцией индекс *i* трактуется как индекс в хеш-таблице списков, т.е. ключ *key* очередной записи отображается на позицию *i* = *h*(*key*) таблицы. Если позиция свободна, то в нее помещается элемент с ключом *key*, если же она занята, то отрабатывается алгоритм разрешения конфликтов, в результате которого такие ключи добавляются в список, начинающийся в *i*-й ячейке хеш-таблицы. Например, обозачив *N* –*NULL*:




В итоге имеем таблицу массива связных списков или деревьев.

Процесс заполнения (считывания) хеш-таблицы прост, но доступ к элементам требует выполнения следующих операций:

– вычисление индекса *i*;

– поиск в соответствующей цепочке.

Для улучшения поиска при добавлении нового элемента можно использовать алгоритма вставки не в конец списка, а – с упорядочиванием, т.е. добавлять элемент в нужное место.

При решении задач на практике необходимо подобрать хеш-функцию *i* = *h*(*key*), которая по возможности равномерно отображает значения ключа *key* на интервал [0, *m*–1], *m* – размер хеш-таблицы. И чаще всего, если нет информации о вероятности распределения ключей по записям, используя метод деления, берут хеш-функцию *i* = *h*(*key*) = *key*%*m*.

При решении обратной задачи – доступ (поиск) к определенному подмножеству возможен из хеш-таблицы (хеш-структуры), которая обеспечивает по хеш-адресу (индексу) быстрый доступ к нужному элементу.

### *15.7.4. Примеры реализации схем хеширования*

***Пример реализации метода прямой адресации с линейным опробыванием***. Исходными данными являются 7 записей (для простоты

информационная часть состоит из целых чисел) объявленного структурного типа:

```
struct zap {
        int key;            // Ключ
        int info;           // Информация
} data;
```

{59,1}, {70,3}, {96,5}, {81,7}, {13,8}, {41,2}, {79,9}; размер хеш-таблицы m = 10. Выберем хеш-функцию $i = h(data) = data.key\%10$; т.е. остаток от деления на 10 – $i \in [0,9]$.

На основании исходных данных последовательно заполняем хеш-таблицу.

Хеш-таблица ($m$=10)

| Хеш-адреса ***i*** : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***key*** : | ***70*** | ***81*** | ***41*** | ***13*** | ***79*** | | ***96*** | | | ***59*** |
| ***info*** : | ***3*** | ***7*** | ***2*** | ***8*** | ***9*** | | ***5*** | | | ***1*** |
| проба : | 1 | 1 | 2 | 1 | 6 | | 1 | | | 1 |

Хеширование первых пяти ключей дает различные индексы (хеш-адреса):

```
i = 59 % 10 = 9;        i = 70 % 10 = 0;
i = 96 % 10 = 6;        i = 81 % 10 = 1;
i = 13 % 10 = 3.
```

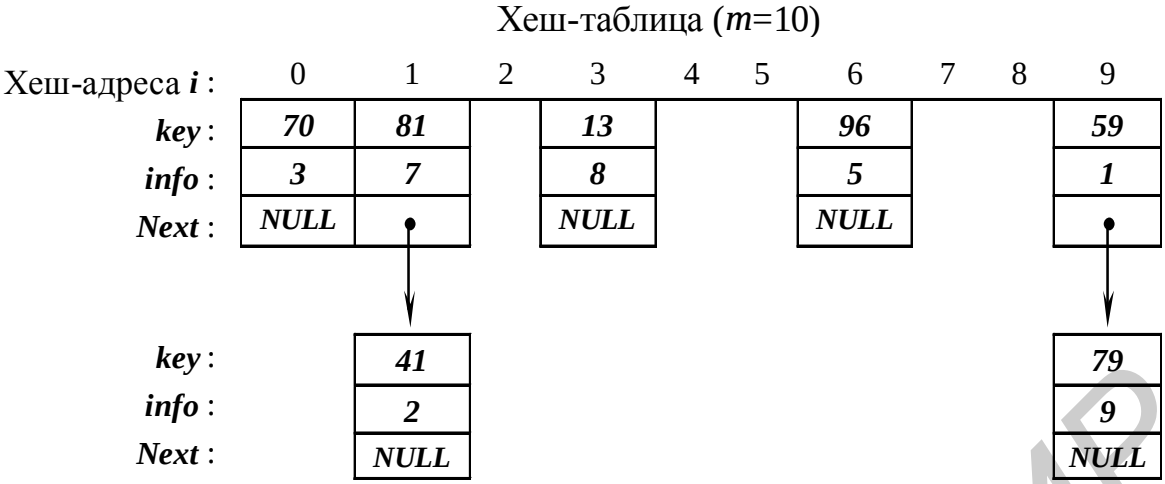
Первая коллизия возникает между ключами 81 и 41 – место с индексом 1 занято. Поэтому просматриваем хеш-таблицу с целью поиска ближайшего свободного места, в данном случае – это $i$ = 2.

Следующий ключ 79 также порождает коллизию: позиция 9 уже занята. Эффективность алгоритма резко падает, т.к. для поиска свободного места понадобилось 6 проб (сравнений), свободным оказался индекс $i$ = 4. Общее число проб – 1–9 проб на элемент.

***Реализация метода цепочек*** для предыдущего примера. Объявляем структурный тип для элемента однонаправленного списка:

```
struct zap {
        int key;        // Ключ
        int info;       // Информация
        zap *Next;      // Указатель на следующий элемент в списке
} data;
```

На основании исходных данных последовательно заполняем хеш-таблицу, добавляя новый элемент в конец списка, если место уже занято.




Хеширование первых пяти ключей, как и в предыдущем случае, дает различные индексы (хеш-адреса): 9, 0, 6, 1, и 3.

При возникновении коллизии новый элемент добавляется в конец списка. Поэтому элемент с ключом 41 помещается после элемента с ключом 81, а элемент с ключом 79 – после элемента с ключом 59.

## ЗАДАНИЕ 8. Обработка списков

***Вариант 1. Однонаправленные списки***

Написать программу по созданию, просмотру, добавлению и решению поставленной задачи для однонаправленного линейного списка (стек и/или очередь).

1. Создать список из случайных целых чисел, лежащих в диапазоне от –50 до +50 и преобразовать его в два списка. Первый должен содержать только положительные числа, а второй – только отрицательные. Порядок следования чисел должен быть сохранен.

2. Создать список из случайных целых чисел и удалить из него записи с четными числами.

3. Создать список из случайных положительных и отрицательных целых чисел (от –10 до 10) и удалить из него отрицательные элементы.

4. Создать список из случайных целых чисел и поменять местами крайние элементы.

5. Создать список из случайных целых чисел и удалить элементы, заканчивающиеся на цифру 5.

6. Создать список из случайных целых чисел и поменять местами элементы, содержащие максимальное и минимальное значения.

7. Создать список из случайных целых чисел. Перенести в другой список все элементы, находящиеся между вершиной и элементом с максимальным значением.

8. Создать список из случайных целых чисел. Перенести в другой список все элементы, находящиеся между вершиной и элементом с минимальным значением.

9. Создать список из случайных чисел, определить количество элементов, находящихся между минимальным и максимальным элементами, и удалить их.

10. Создать список из случайных чисел и определить количество элементов, имеющих значения, меньше среднего значения от всех элементов, и удалить эти элементы.

11. Создать список из случайных чисел, вычислить среднее арифметическое и заменить им первый элемент.

12. Создать список из случайных целых чисел, разделить его на два: в первый поместить все четные, а во второй – нечетные числа.

13. Создать список из случайных целых чисел в диапазоне от 1 до 10, определить наиболее часто встречающееся число и удалить его.

14. Создать список из случайных целых чисел и удалить из него каждый второй элемент.

15. Создать список из случайных целых чисел и удалить из него каждый нечетный элемент.

***Вариант 2. Двунаправленные списки***

Написать программу по созданию, просмотру, добавлению и решению поставленной задачи для двунаправленного линейного списка.

1. Создать список из случайных целых чисел. Найти минимальный элемент и сделать его первым.

2. Создать два списка из случайных целых чисел. В первом найти максимальный элемент и за ним вставить элементы второго.

3. Создать список из случайных целых чисел. Удалить из списка все элементы, находящиеся между максимальным и минимальным элементами.

4. Упорядочить элементы списка случайных целых чисел в порядке возрастания.

5. Создать список из случайных целых чисел. Удалить из списка все элементы, находящиеся до максимального элемента.

6. Создать список из случайных целых чисел. Удалить из списка все элементы, находящиеся после минимального элемента.

7. Создать список из случайных целых чисел. Из элементов, расположенных между максимальным и минимальным элементами, создать второй список, а из остальных – третий.

8. Создать список из случайных положительных и отрицательных целых чисел. Образовать из него два списка, первый должен содержать отрицательные числа, а второй – положительные.

9. Создать список из случайных целых чисел. Удалить из списка все элементы, находящиеся после максимального элемента.

10. Создать два списка из случайных целых чисел. Вместо элементов первого списка, заключенных между максимальным и минимальным элементами, вставить второй список.

11. Создать список из случайных целых чисел. Удалить из списка элементы с повторяющимися более одного раза значениями.

12. Создать список из случайных целых чисел и удалить все элементы, кратные 5.

13. Создать список из случайных целых чисел. Удалить из списка все элементы, большие среднего арифметического.

14. Создать список из случайных чисел. Преобразовать его в кольцо. Предусмотреть возможность движения по кольцу в обе стороны с отображением места положения текущего элемента.

15. Создать список из случайных целых чисел. Удалить из списка все элементы, находящиеся между максимальным и минимальным элементами.

## ЗАДАНИЕ 9. Деревья и польская запись

***Вариант 1. Создание и обработка структур типа «дерево»***

Разработать проект для обработки дерева поиска, каждый элемент которого содержит целочисленный ключ и строку текста, содержащую, например, ФИО и номер паспорта (ввод исходной информации рекомендуется записать в файл). В программе должны быть реализованы следующие возможности:

- создание дерева;
- добавление новой записи;
- поиск информации по заданному ключу;
- удаление информации с заданным ключом;
- вывод информации;
- решение индивидуального задания;
- освобождение памяти при выходе из программы.

1. Поменять местами информацию, содержащую максимальный и минимальный ключи.

2. Подсчитать число листьев в дереве.

3. Удалить из дерева ветвь с вершиной, имеющей заданный ключ.

4. Определить глубину дерева.

5. Определить число узлов на каждом уровне дерева.

6. Удалить из левой ветви дерева узел с максимальным значением ключа и все связанные с ним узлы.

7. Определить количество узлов с четными ключами.

8. Определить число листьев на каждом уровне дерева.

9. Определить число узлов в дереве, имеющих только одного потомка.

10. Определить количество узлов правой ветви дерева.

11. Определить количество записей в дереве, начинающихся с введенной с клавиатуры буквы.

12. Найти среднее значение всех ключей дерева и найти строку, имеющую ближайший к этому значению ключ.

13. Определить количество узлов левой ветви дерева.

14. Определить число узлов в дереве, имеющих двух потомков.

15. Найти запись с ключом, ближайшим к среднему значению между максимальным и минимальным значениями ключей.

***Вариант 2. Создание и использование польской записи***

Написать программу формирования обратной польской записи и расчета полученного выражения. Предусмотреть возможности того, что идентификаторы могут состоять более чем из одного символа и могут быть использованы операции % и возведение в степень. Результат работы программы проверить на конкретном примере (табл. 15.1).

Например, если ввести выражение $(a + b)*(c - d)/e$ и значения переменных $a = 3$, $b = 5$, $c = 6$, $d = 9$, $e = 7$, должны получиться следующие результаты:

*Постфиксная форма* *ab+cd– *e/*
*Результат расчета* *– 3.42857*

Таблица 15.1

| № | Выражение | *a* | *b* | *c* | *d* | *e* | Результат |
|---|---|---|---|---|---|---|---|
| 1 | *a*/(*b*– *c*)*(*d*+*e*) | 8.6 | 2.4 | 5.1 | 0.3 | 7.9 | – 26.12 |
| 2 | (*a*+*b*)*(*c*– *d*)/*e* | 7.4 | 3.6 | 2.8 | 9.5 | 0.9 | – 81.89 |
| 3 | *a*– (*b*+*c**d*)/*e* | 3.1 | 5.4 | 0.2 | 9.6 | 7.8 | 2.16 |
| 4 | *a*/*b*– ((*c*+*d*)**e*) | 1.2 | 0.7 | 9.3 | 6.5 | 8.4 | – 131.006 |
| 5 | *a**(*b*– *c*+*d*)/*e* | 9.7 | 8.2 | 3.6 | 4.1 | 0.5 | 168.78 |
| 6 | (*a*+*b*)*(*c*– *d*)/*e* | 0.8 | 4.1 | 7.9 | 6.2 | 3.5 | 2.38 |
| 7 | *a**(*b*– *c*)/(*d*+*e*) | 1.6 | 4.9 | 5.7 | 0.8 | 2.3 | – 0.413 |
| 8 | *a*/(*b**(*c*+*d*))– *e* | 8.5 | 0.3 | 2.4 | 7.9 | 1.6 | 1.151 |
| 9 | (*a*+(*b*/*c*– *d*))**e* | 5.6 | 7.4 | 8.9 | 3.1 | 0.2 | 0.666 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | $a*(b+c)/(d-e)$ | 0.4 | 2.3 | 6.7 | 5.8 | 9.1 | – 1.091 |
| 11 | $a-(b/c*(d+e))$ | 5.6 | 3.2 | 0.9 | 1.7 | 4.8 | – 17.51 |
| 12 | $(a-b)/(c+d)*e$ | 0.3 | 6.7 | 8.4 | 9.5 | 1.2 | – 0.429 |
| 13 | $a/(b+c-d*e)$ | 7.6 | 4.8 | 3.5 | 9.1 | 0.2 | 1.173 |
| 14 | $a*(b-c)/(d+e)$ | 0.5 | 6.1 | 8.9 | 2.4 | 7.3 | – 0.144 |
| 15 | $(a+b*c)/(d-e)$ | 9.1 | 0.6 | 2.4 | 3.7 | 8.5 | – 2.196 |

# ГЛАВА 16. Переход к ООП

При переходе от языка Си к языку C++ в стандарт *ANSI* были введены дополнительные механизмы, которые позволили в конечном итоге создать среду для разработки программ в объектно-ориентированном стиле.

Рассмотрим некоторые из них.

## *16.1. Потоковый ввод-вывод*

Поток – это абстрактное понятие, которое относится к любому переносу данных от источника к приемнику. Потоки C++ обеспечивают надежную работу как со стандартными (*stdin, stdout*), так и с определенными пользователями типами данных. Поток определяется как последовательность байт, не зависящая от конкретного устройства.

Для ввода-вывода в языке C++ используются два объекта класса *iostream*: *cin* (класс *istream*), *cout* (класс *ostream*) и две переопределенные операции побитового сдвига. Для их работы необходимо подключить заголовочный файл *iostream.h.*

Формат записи операций *помещения в поток* << (вывод на экран) и *извлечения из потока* >> (ввод с клавиатуры) следующий:

***cout*** << *ID переменной* ;
***cin*** >> *ID переменной* ;

Стандартный поток вывода *cout* по умолчанию связан со стандартным устройством вывода ***stdout*** (дисплей монитора), а ввода *cin* – со стандартным устройством ввода ***stdin***, т.е. клавиатурой. Приведем пример:

```
#include<iostream.h>
void  main (void)
{
        int  i, j, k;
        cout << " Hello! " << endl;    //  «end line» – переход на новую строку
        cout << " Input  i, j ";
        cin  >> i >> j ;
        k = i + j ;
        cout << " Sum  i , j = " << k << endl;
}
```

## *16.2. Управление выводом*

В стандарте языка Си ANSI ввод-вывод данных осуществляется при помощи стандартных библиотечных функций. Управление выводом осуществляется при помощи использования форматов и управляющих символов.

Для форматирования и управления выводом данных в потоке введен механизм ***манипуляторов*** – специальных функций для модификации работы

потока, предназначенных для форматирования данных, как при выводе, так и в оперативной памяти.

***Использование манипуляторов***

Манипуляторы – специальные функции, возвращающие модифицированные данные потока. В большинстве случаев их использование позволяет форматировать данные, как при выводе, так и в оперативной памяти.

Для их использования необходимо вместо файла *iostream.h* подключить заголовочный файл ***iomanip.h*** (манипуляторы для вывода потоками).

Рассмотрим работу некоторых манипуляторов на конкретном примере.

```
#include<iomanip.h>
    main()
    {
        int a = 157;
        double b = 1.55555;
        cout << setw(10) << a << endl;
```

/* Манипулятор ***setw***(***n***) – устанавливает ширину поля, т.е. *n* позиций, для вывода объекта. На экране объект *a* будет выводиться с 8-й позиции, первые 7 позиций – пустые: 157 (заполнение пробелами неиспользуемой части). Действует только для следующего за ним объекта. */

```
        cout << setw(10) << setfill('z') << a << endl;
```

/* Манипулятор ***setwfill***(***kod***) – устанавливает заполнитель пробелов, заданный символом или его кодом (можно было указать 122 – код символа 'z'). На экране: zzzzzzz157. Действует до изменения или отмены ***setwfill***(**0**) .*/

```
        cout << oct << a << endl;
```

/* Манипулятор ***oct*** – выполняет преобразование объекта в 8-ричную форму представления. На экране: 235 */

```
        cout << hex << a << endl;
```

// ***hex*** – преобразует объект в 16-ричную форму. На экране: 9d

```
        cout << dec << a << endl;
```

// ***dec*** – преобразует обратно в 10-тичную. На экране: 157

```
        cout << b << endl;                          // На экране: 1.55555
        cout << setprecision(3) << b << endl;
```

/* ***setprecision***(***n***) – устанавливает *n* значащих цифр после запятой с учетом точки или без нее, в зависимости от системы программирования. На экране:
1.56 или 1.556 */

```
        return 0;
}
```

***Флажки***

Помимо манипуляторов для управления выводом данных используются специальные флажки, принадлежащие классу *ios*, которые также позволяют формировать потоки вывода.

Установить флажок позволяет функция ***setiosflags***(*ios* :: *flag*);

Снять флажок позволяет функция ***resetiosflags***(*ios* :: *flag*);

Причем можно установить сразу несколько флажков, используя для этого побитовую операцию «|» (поразрядное ИЛИ) для их объединения в одну группу.

Следующий пример показывает приемы работы с некоторыми флажками механизма вывода потоками.

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
void main(void) {
        int a = 157;
    cout<<setiosflags(ios :: showbase)<<a<<“  “<<oct<<a<< “  “
                                                <<hex<<a<< endl;
```

/* ***showbase*** – показать, в какой системе счисления выводится число. На экране: 157 0235 0x9d */

```
                double a1 = 12.99, a2 = 15;
                cout << setiosflags(ios :: showpoint | ios :: fixed)
```

/* ***showpoint*** – печатать десятичную точку, ***fixed*** – выводить в форме с фиксированной десятичной точкой */

```
        << setprecision(2) << setfill(‘*’) << setiosflags(ios :: right)
```

// ***right*** – выравнивать вывод по правому краю (по левому – ***left***)

```
                << “ a1  “ << setw(10) << a1
                << “ a2  “ << setw(10) << a2 << endl;
```

// На экране: a1 *****12.99 a2 *****15.00

```
                double pi = 3.14159;
                cout << “ Pi “ << setw(15) << setfill(‘_’)
```

// Символ заполнения ‘_’ – знак подчеркивания

```
        << setiosflags(ios :: showpos | ios :: scientific)
        << setprecision(5) << pi << endl;
```

/* ***showpos*** – явно показать знак «+», ***scientific*** – вывод в форме с плавающей десятичной точкой. На экране: Pi _ _ _ +3.14159e+00 */

```
}
```

В заключение отметим, что можно создавать свои собственные манипуляторы, которые будут выполнять запрограммированные действия.

### *16.3. Проблема ввода-вывода кириллицы в среде Visual C++*

Работа в среде *Visual* C++ 6.0 (в режиме консольных приложений) сопряжена с определенными неудобствами. Например, попытка вывести фразу на русском языке, как стандартными функциями вывода, так и с помощью ввода-вывода потоками, терпит неудачу. Создадим в среде *Visual* C++ 6.0 консольное приложение и наберем следующий текст:

```
#include <iostream.h>
int main()
{
	cout << "Welcome to C++ !" << endl;
	cout << "Добро пожаловать в C++ !" << endl;
	return 0;
}
```

В результате на экране получим нечто следующее:

*Welcome to C++ !*
*─юсЕю яюцрыютрЄ№ т C++ !*
*Press any key to continue*

То есть вместо фразы на русском языке получается бессмысленный набор символов. Это вызвано различными стандартами кодировки символов кириллицы в операционных системах *MS DOS* и *Windows*.

Весь ввод-вывод в консольном окне идет в кодировке стандарта *ASCII*. Данный стандарт является международным только в первой половине кодов, т.е. для кодов от 0 до 127, а вторая половина кодов от 128 до 255 предназначена для национальных шрифтов. Так, например, в бывшем СССР помимо альтернативной кодировки ГОСТа (*Alt*), использовались – основная кодировка ГОСТа (*Mai*), болгарская кодировка (*MIC*), кодировка КОИ-8 (*KOI*), у которых символы кириллицы имеют разные коды. Сейчас в России – альтернативная кодировка *ASCII*.

Текст же в исходных файлах, набираемый в текстовом редакторе *Visual* C++, имеет кодировку в стандарте *ANSI*. Данный стандарт в первой половине совпадает с альтернативной кодировкой *ASCII*, а во второй – отличается, так как разработчики *Visual* решили, что консольное приложение должно имитировать работу в среде *MS DOS* и оставили альтернативную кодировку *ASCII*.

Для нормального вывода строки, содержащей буквы русского алфавита, надо использовать функцию ***CharToOem***, предназначенную для преобразования символов с кодировкой *ANSI* в кодировку *ASCII*. Аналогично, если в программе есть консольный ввод текста и этот текст в дальнейшем надо сохранять в документах (файлах) с кодировкой *ANSI*, то перед сохранением нужно воспользоваться функцией обратного преобразования – ***OemToChar***. Эти функции декларированы в заголовочном файле *windows.h*.

С учетом сказанного выше можно предложить следующую программу корректного вывода информации на русском языке:

```
#include <iostream.h>
#include <windows.h>
        char* Rus(const char* text);
        char bufRus[255];
int main()
{
        char s[] = "Минск!", ss[100];
        cout << Rus("Город  ") << Rus(s) <<endl;
        cout << Rus("Введи строку:");
        cin >> ss;
        cout << Rus(" Строка: ") << ss << endl;
        return 0;
}
char* Rus (const char* text)
{
        CharToOem(text, bufRus);
        return bufRus;
}
```

Результат программы может быть следующим:

*Город Минск!*
*Введи строку: Москва!*
*Строка: Москва!*

Таким образом, для решения проблемы с русским языком в консольном выводе *Visual* C++ 6.0 создана небольшая функция ***Rus***, которая обращается к функции ***CharToOem***, передает ей для преобразования полученный через свой параметр текст на русском языке и возвращает указатель на преобразованную строку. В качестве временного хранилища используется глобальный символьный массив ***bufRus***. Использовать функцию просто: везде вместо строковых объектов (строковых констант и переменных) в программах нужно писать ***Rus***(строковый объект).

Непосредственное использование функции ***CharToOem***, например, в стандартных функциях вывода данных недопустимо, так как возвращает результат типа ***BOOL***, а результат преобразования размещает по адресу своего второго аргумента. Поэтому и была создана эта небольшая пользовательская функция, которая имеет единственное ограничение: функцию ***Rus*** нельзя использовать в цепочке операций << более одного раза, так как для различных компиляторов и режимов оптимизации может быть получен неверный результат.

### *16.4. Операции new и delete*

В языке С++ для захвата и освобождения памяти используется более простой механизм – операции ***new*** и ***delete***. Рассмотрим эти операции на простых примерах:

1) *type *p* = ***new*** *type* (*значение*); – захват участка памяти размером *sizeof*(*type*), путем установки на него указателя, и запись в эту область указанного значения;

. . .

***delete*** *p*; – освобождение захваченной памяти.

2) *type *p* = ***new*** *type*[*n*]; – захват памяти на *n* последовательно размещенных объектов, возвращает указатель на начало участка ОП размером *n*sizeof*(*type*); используется для создания массива;

. . .

***delete*** []*p*; – освобождение всей захваченной памяти.

Следует заметить, что операция *delete* не уничтожает значения, находящиеся по указанным адресам, а дает компилятору разрешение использовать ранее занятую память в дальнейшем.

Квадратные скобки в операции *delete* [ ] при освобождении памяти, занятой массивом, обязательны. Их отсутствие может привести к непредсказуемым результатам.

***Пример создания одномерного динамического массива***

Для примера приведем участок кода программы для одномерного динамического массива с использованием операций *new* и *delete*.

Напомним, что результатом операции *new* является адрес начала области памяти для размещения данных, указанного количества и типа. При нехватке памяти результат равен *NULL*.

```
        …
double *x;
int   i, n;
puts(" Введите размер массива: ");
scanf(“%d”, &n);
x = new double [n] ;
if (x = = NULL) {
        puts(" Ошибка ! ");
        return;
}
for (i=0; i<n; i++)                  // Ввод элементов массива
        scanf(“%lf”, &x[i]);
        …                            // Обработка массива
delete [ ]x;                         // Освобождение памяти
        …
```

***Пример создания двухмерного динамического массива***

Напомним, при создании двухмерного динамического массива сначала выделяется память на указатели, расположенные последовательно друг за другом, а затем каждому из них выделяется соответствующий участок памяти под элементы.

```
        . . .
int  **m, n1, n2, i, j;
puts(" Введите размеры массива (строк, столбцов): ");
scanf("%d%d", &n1, &n2);
m = new int*[n1];                  // Захват памяти для указателей – A (n1=3)
for (i=0; i<n1; i++)               // Захват памяти для элементов
        *(m+i) = new int[n2];
for ( i=0; i<n1; i++)
        for ( j=0; j<n2; j++)
                        m[i] [j] = i+j;              //  *(*(m+i)+j) = i+j;
        . . .
for ( i=0; i<n1; i++)              // Освобождение памяти
        delete []m[i];
delete []m;
        . . .
```

## *16.5. Дополнительные возможности при работе с пользовательскими функциями*

***Параметры со значениями по умолчанию***

Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию. Эти параметры должны быть последними в списке и при вызове функции аргументы для них могут опускаться. Если при вызове аргумент опущен, то должны отсутствовать и все аргументы, стоящие за ним, т.к. задавать значения по умолчанию можно только для последних параметров в списке функции.

В качестве значений параметров по умолчанию могут использоваться константы или константные выражения.

Параметр по умолчанию проходит проверку типа во время описания функции и вычисляется во время ее вызова.

Пример участка кода функции, определяющей сумму переменных отношений от 2-х до 5-ти:

```
        . . .
int sum(int a, int b, int c=0, int d=0, int e=0)  {      // 0 – умалчиваемые значения
        return (a+b+c+d+e);
}
```

```
int main ()
{
        int  x1=1, x2=2, x3=3, x4=4, x5=5;
        int  y2, y3, y4, y5;
        y2= Sum (x1, x2);                  // Работают все умалчиваемые значения;
        y3= Sum (x1, x2, x3);              // – два последних значения;
        y4= Sum (x1, x2, x3, x4);          // – одно последнее значение;
        y5= Sum (x1, x2, x3, x4, x5)
        . . .
        return 0;
}
```

Таким образом:

1. Умалчиваемое значение аргумента функции задается при его объявлении в заголовке функции.

2. В начале списка указывают параметры, значения которых будут передаваться всегда.

3. При обращении пропуск умалчиваемых параметров в списке недопустим, т.е. для получения значения $x1 + x2 + x3 + x5$ вызов функции *Sum* ($x1$, $x2$, $x3$, $x5$); приведет к ошибочному результату.

Правильным будет обращение *Sum*($x1$, $x2$, $x3$, 0, $x5$);

### *Перегрузка функций*

В языке C++ реализована возможность использования одного идентификатора для функций, выполняющих различные действия над различными типами данных, в результате чего можно использовать несколько функций с одним и тем же именем, но с разными списками параметров, как по количеству, так и по типу.

Такие функции называют *перегруженными*, а сам механизм – *перегрузка функций*.

Компилятор определяет, к какой из функций с одним и тем же именем следует обратиться путем сравнения типов фактических аргументов с типами формальных параметров в заголовках всех этих функций, т.е. компилятор в зависимости от типа и количества аргументов будет формировать необходимое обращение к соответствующей функции.

Поиск функции, которую надо вызвать, осуществляется за три отдельных шага:

1. Поиск функции с точным соответствием параметров и ее использование, если она найдена.

2. Поиск соответствующей функции, используя встроенные преобразования типов данных.

3. Поиск соответствующей функции, используя преобразования, определенные пользователем.

***Пример перегрузки функций***

Приведем пример функции *S*1 с двумя параметрами *x*, *y*, работающая в зависимости от типа передаваемых аргументов, следующим образом:

– если тип параметров целочисленный, функция *S*1 складывает их значения и возвращает полученную сумму;

– если тип параметров *long*, функция *S*1 перемножает их значения и возвращает полученное произведение;

– если тип параметров вещественный, функция *S*1 делит их значения и возвращает частное от деления.

```
# include <stdio.h>
int S1 (int x, int y) {
        return x+y;
}
long S1 (long x, long y) {
        return x*y;
}
double S1 (double x, double y) {
        return x/y;
}
int main ()
{
        int a = 1, b = 2, c;
        long i = 3, j = 4, k;
        double x = 10, y = 2, z;
        c=S1(a, b);
        k=S1(i, j);
        z=S1(x, y),
        printf("\n c = %d; k = %ld; z = %lf . \n", c, k, z);
        return 0;
}
```

В результате получим:

*c = 3; k = 12; z = 5.000000* .

***Функции с переменным числом параметров***

Многоточие в списке параметров пользовательской функции используется тогда, когда число аргументов заранее неизвестно. При этом неопределенное количество параметров можно указать в ее прототипе следующим образом:

***void f1*(*int a, double b, …*);**

Такая запись указывает компилятору на то, что за обязательными фактическими аргументами для параметров *a* и *b* могут следовать, а могут и не следовать другие аргументы при вызове этой функции.

Перечислим основные особенности использования данного механизма.

1. Используется несколько макрокоманд для доступа к параметрам таких функций, это:

***va_list*** и ***va_start*** – макрокоманды подготовки доступа к параметрам;

***va_arg*** – использование параметров;

***va_end*** – отчистка перед выходом.

Они объявлены в заголовочном файле ***stdarg.h***.

2. Такая функция должна иметь минимум один параметр (именованный) для передачи ей количества передаваемых аргументов.

3. Для макроса *va_start* необходимо передать два аргумента – имя списка параметров, который задает *va_list* и их количество.

4. Нарушать указанный порядок макрокоманд нельзя. Иначе можно получить непредсказуемые последствия.

5. Для макроса *va_arg* нужно помимо имени списка параметров передать и предполагаемый тип. При несоответствии типов – ошибка.

Использование многоточий полностью выключает проверку типов параметров. Многоточие необходимо, только если изменяются и число параметров, и их тип.

Следующий пример иллюстрирует эту возможность.

```
#include <stdarg.h>
#include <stdio.h>
void f1(double s, int n ...)  {
        int v;
        va_list p;
        va_start(p, n);
        printf(" \n Double S = %lf ", s);
        for(int i=1; i<=n; i++) {
                v = va_arg(p, int);
                printf("\n Argument %d = %d ", i, v);
        }
        va_end(p);
}
void main(void) {
        f1(1.5, 3, 4, 5, 6);
}
```

В результате получим:

*Double S = 1.500000*
*Argument 1 = 4*
*Argument 2 = 5*
*Argument 3 = 6*
*Press any key to continue*

## *16.6. Шаблоны функций*

### *Понятие шаблона функции*

Параметризация определений функций позволяет сократить трудоемкость программирования и повысить лаконичность текста.

Механизм шаблонов – средство построения обобщенных определений функций, независимых от используемых типов данных. Их использование избавляет от необходимости дублировать код функции для различных типов данных, составляющих их параметры и возвращаемые результаты. Это позволяет снять достаточно жесткие ограничения, накладываемые механизмом взаимосвязи между формальными параметрами и фактическими аргументами при вызове функции.

Компилятор по заданному в качестве аргумента конкретному типу данных на основе определения шаблона автоматически порождает соответствующие экземпляры или «представители» функций.

Рассмотрим необходимость создания шаблона функции на конкретном примере.

Пусть нужно написать код для определения максимального из пар значений *x* и *y* для разных типов значений для этой пары: например, целочисленные данные и пара вещественных данных.

Существуют следующие варианты решения данной задачи.

1. Для ускорения работы программы воспользоваться перегрузкой функций:

```
int Max(int x, int y) {
        return (x>y) ? x : y;
}
double Max(double x, double y) {
        return(x>y)? x : y;
}
        . . .
int i, j, k;
k = Max(i, j);                  // Первая функция
double a, b, d;
d = Max(a, b);                  // Вторая функция
        . . .
```

Идентификаторы функций и их код дублируются, различие только в типах параметров и возвращаемого результата.

2. Вторая возможность – воспользоваться макросом:

```
#define Max(x, y)  ((x>y) ? x : y )
```

В этом случае недостатки:

– снижается надежность, т.к. при использовании макросов не работает проверка типов аргументов;

– данная подстановка будет выполнена во всех местах, где встретится *Max*;

– в данной ситуации лучше воспользоваться механизмом шаблонов, основанном на параметризации. Данная параметризация осуществляется при помощи обобщенного определения этой функции шаблоном (***template***), в котором тип параметров определяют символическим идентификатором (например буквой).

Компилятор, проанализировав передаваемые в функцию аргументы, а именно – их типы, сгенерирует код нужной функции, заменив символическое имя типа параметров на конкретный тип.

Формат шаблона функции:

***template*** *список параметров шаблона*
*декларация функции*;

Набор параметризуемых типов определяется списком параметров шаблона. Каждый тип определяется ключевым словом ***class*** и локальным в рамках функции идентификатором типа. Список параметров не может быть пустым, а его элементы разделяются запятыми.

Элемент *декларация функции* – обычное определение (или описание – прототип) функции. В списке ее параметров необходимо упомянуть хотя бы один раз типы параметров из списка шаблона.

Примеры отношений между параметрами шаблона и функции:

1. Шаблон может иметь единственный параметр:

```
template <class T>
void f1(T par) {
        код функции f1
}
```

При этом символ типа *T* можно использовать и для спецификации возвращаемого значения, а также для любых других объектов в коде функции. Обязательный пункт – использование *T* в списке параметров функции выполнен.

Функция может иметь любые типы параметров как параметризованные, так и стандартно декларированные.

2. Шаблон функции с частично параметризованными параметрами:

```
template <class T>
void f2(T par, int x, int y) {
        код функции f2
}
```

3. Шаблон может иметь несколько параметризованных параметров с разными символическими идентификаторами:

```
template <class T2, class T1>
void f3(T1 par1, T2 par2) {
```

```
        код функции f3
    }
```

Порядок следования *ID* параметров в заголовке функции может отличаться от их декларации в шаблоне.

Использование шаблона функций не требует специальных действий от программиста, т.к. компилятор автоматически формирует требуемый операцией вызова экземпляр кода по набору типов аргументов, которые передаются в функцию. То есть шаблон функции не является ее экземпляром. Только при обращении к функции с аргументами конкретного типа происходит генерация конкретной функции.

Пример нахождения максимума для разных типов аргументов:

```
#include <iostream.h>
    template <class T>
    T max(T x, T y) {
        return(x>y)? x : y;
     }
     void main()
     {
        int i = 2, j = 3;
        double x = 1.5, y = 2.5;
        cout <<" I = "<<i<<"  J = "<<j<< "  Max = "<<max(i, j)<<endl;
        cout <<" X = "<<x<<"  Y = "<<y<<" Max = "<<max(x,y)<<endl;
     }
```

Результат выполнения программы:

*i = 2  j = 3  Max = 3*

*X = 1.5  y = 2.5  Max = 2.5*

Таким образом, шаблон функции избавил копирование ее определения для различных типов данных.

***Перегрузка шаблонов функций***

Можно перегружать функции-шаблоны для неподходящих под данный код шаблона данных, т.к. их использование базируется, как и у обычных функций, на распознавании компилятором различий в списках их параметров.

Кроме того, т.к. код шаблона не меняется, лучше его оформлять отдельным заголовочным файлом, а в функции, которая его использует, подключать его на этапе препроцессорной обработки.

Приведем пример перегрузки шаблонов, состоящей из ограничения использования функции *max* для некоторых типов аргументов.

Перегрузка шаблона создается путем явного его определения (специализированная функция) для конкретного типа данных.

Например, для строк шаблон-функция *max* не подходит, так как все операции над строками производятся только при помощи библиотечных функций, декларированных в заголовочном файле *string.h.*

Шаблон функции поиска максимума будет иметь следующий вид:

```
        template <class T>
T max(T x, T y) {
                return (x>y)? x : y;
}
```

Специализированная функция выбора максимальной из двух строк:

```
char* max(char* x, char* y) {
                return (strcmp(x, y)>0) ? x : y;
}
```

Осуществляется следующая последовательность этапов выбора «экземпляра» перегруженных функций:

1) поиск специализированной (не шаблон) функции с совпадающим списком параметров, и в случае неудачи:

– поиск шаблона функции с точным соответствием списка параметров;

– поиск специальной функции по условию совпадения списка параметров после возможных преобразований типов;

2) поиск завершается выбором нужной функции при совпадении параметров экземпляра функции со списком передаваемых аргументов.

Программа, использующая преобразования шаблонов и специализированных функций, будет иметь следующий вид:

```
#include <stdio.h>
#include <string.h>
        template <class T>
        T max(T x, T y) {
                return (x>y)? x : y;
        }
        char* max(char* x, char* y) {
                return (strcmp(x, y)>0) ? x : y;
        }
        void main()
        {
                int a=7, b=5;
                char c1=’a’, c2=’z’;
                char s1[]=”one”, s2[]=”two”;
                        printf(“\n a=%d, b=%d, max=%d \n”, a, b, max(a, b));
                        printf(“\n c1=%c, c2=%c, max=%c \n”, c1, c2, max(c1, c2));
                        printf(“\n s1=%s, s2=%s, max=%s \n”, s1, s2, max(s1, s2));
        }
```

При построении шаблонов функций, они, как правило, декларируются прототипами, а в программе определяются в произвольном порядке.

Рассмотрим пример нахождения максимального значения в массивах с данными различных типов.

```
        . . .
template <class T> Maxa(T *z, int n);                    // прототип шаблона
void main(void) {
        int a[100], n;
        printf("\n Input n < 100 : ");
        scanf("%d", &n);
        for( int i=0; i<n; i++) {
                printf("\n a[%d] = ", i+1);
                scanf("%d", &a[i]);
        }
        printf("\n Max int = %d ", Maxa(a, n));
// Вызывается  функция Maxa для типа int
        double b[ ] = {0.22, 1.5, 0.88};
        printf("\n Max double = %lf ", Maxa(b,3));
// Вызывается функция Maxa для типа double
        char Str[ ] = "ABC123abc";
        printf("\n Max char = %c ", Maxa(Str, (int)strlen(Str)));
// Вызывается Maxa для типа char
}
// Код шаблона
template <class T>
T Maxa(T *z, int n) {
        int imax = 0;
        for(int i=1; i<n; i++)
                if(z[i]>z[imax]) imax = i;
        return (z[imax]);
}
```

***Советы по программированию***

При выполнении вариантов заданий придерживайтесь следующих ключевых моментов.

1. Шаблоны функций применяются для записи идентичных действий над данными различных типов.

2. Вызов шаблона – это создание шаблоном конкретного варианта функции.

3. Шаблоны можно перегружать как обычными функциями, так и другими шаблонами.

4. Перегрузкой функций называется использование нескольких функций с одним именем и различными типами параметров.

5. Перегрузка применяется, когда одно и то же по смыслу действие реализуется по-разному для различных типов или структур данных.

6. При написании перегруженных функций необходимо, чтобы в процессе поиска нужного варианта функции по ее вызову не возникало неоднозначности. Неоднозначность может возникнуть из-за преобразований типов, параметров по умолчанию и ссылок.

7. Перегруженные функции должны находиться *в одной области видимости*, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.

8. Перегруженные функции могут иметь *параметры по умолчанию*, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.

9. Функции не могут быть перегружены, если описание их параметров отличается только модификатором *const* или использованием ссылки.

10. Многоточие в заголовке функции используется тогда, когда число аргументов неизвестно.

11. Многоточие в заголовке функции требует использования макрокоманд для доступа к параметрам таких функций.

12. Нарушать указанный порядок макрокоманд нельзя.

## ЗАДАНИЕ 10. Перегрузка функций

***Первый уровень сложности***

Написать программу для работы с двумя одномерными динамическими массивами размером *n*. Для формирования массивов и вывода их на печать использовать стандартные потоки. Оформить решение задачи в виде шаблона функций. Глобальные данные в программе не допускаются. Все необходимые данные должны передаваться функциям через параметры (для параметров, соответствующих размерам массивов, используйте умалчиваемые значения). Программа должна демонстрировать использование умалчиваемых значений параметров, а также использование шаблона функций для типов данных ***int*** и ***double***.

1. Первым на печать вывести массив, сумма значений которого окажется наименьшей.

2. Первым на печать вывести массив, произведение значений которого окажется наименьшим.

3. В каждом из массивов найти наименьшее значение и прибавить его ко всем элементам массивов. На печать вывести исходные и преобразованные массивы.

4. В каждом из массивов найти наибольшее значение и вычесть его из всех элементов массивов. На печать вывести исходные и преобразованные массивы.

5. В каждом из массивов найти среднее арифметическое всех элементов массивов. На печать вывести исходные массивы и найденные значения.

6. Первым на печать вывести массив, содержащий наименьшее значение. Напечатать также это значение и его порядковый номер.

7. Подсчитать в массивах количество отрицательных элементов и первым на печать вывести массив, имеющий наименьшее их количество.

8. Подсчитать в массивах количество положительных элементов и первым на печать вывести массив, имеющий наименьшее их количество.

9. Подсчитать в массивах количество отрицательных элементов и первым на печать вывести массив, имеющий наибольшее их количество.

10. Подсчитать в массивах количество положительных элементов и первым на печать вывести массив, имеющий наибольшее их количество.

11. Подсчитать в массивах количество элементов, больших значения *t*, и первым на печать вывести массив, имеющий наименьшее их количество.

12. Подсчитать в массивах количество элементов, меньших значения *t*, и первым на печать вывести массив, имеющий наименьшее их количество.

13. Подсчитать в массивах количество элементов, больших значения *t*, и первым на печать вывести массив, имеющий наибольшее их количество.

14. В каждом из массивов найти наименьшее значение и умножить на него все элементы массивов. На печать вывести исходные и преобразованные массивы.

15. В каждом из массивов найти наибольшее значение и умножить на него все элементы массивов. На печать вывести исходные и преобразованные массивы.

***Второй уровень сложности***

Написать программу для работы с двухмерным динамическим массивом размером $n \times m$. Для формирования массива и вывода на печать использовать стандартные потоки. Оформить решение задачи в виде шаблона функций. Глобальные данные в программе не допускаются. Все необходимые данные должны передаваться функциям через параметры (для параметров, соответствующих размерам массивов, используйте умалчиваемые значения). Программа должна демонстрировать использование умалчиваемых значений параметров, а также использование шаблона функций для типов данных ***int*** и ***double***.

1. В заданном двухмерном массиве определить максимальное из чисел, встречающихся в массиве более одного раза.

2. В заданном двухмерном массиве определить сумму элементов в тех строках, которые не содержат отрицательных элементов.

3. В заданном двухмерном массиве определить количество строк, содержащих хотя бы один нулевой элемент.

4. В заданном двухмерном массиве определить номер первой из строк, не содержащих ни одного положительного элемента.

5. В заданном двухмерном массиве определить номер первого из столбцов, не содержащих ни одного отрицательного элемента.

6. В заданном двухмерном массиве определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.

7. В заданном двухмерном массиве определить номер первой из строк, содержащих хотя бы один положительный элемент.

8. В заданном двухмерном массиве определить количество строк, среднее арифметическое элементов которых меньше заданной величины.

9. В заданном двухмерном массиве определить сумму модулей его отрицательных нечетных элементов.

10. В заданном двухмерном массиве определить минимум среди сумм элементов диагоналей, параллельных главной диагонали.

11. В заданном двухмерном массиве определить количество элементов массива, лежащих в диапазоне от $A$ до $B$.

12. В заданном двухмерном массиве определить произведение элементов массива, расположенных после максимального по модулю элемента.

13. В заданном двухмерном массиве определить сумму модулей элементов массива, расположенных после минимального по модулю элемента.

14. В заданном двухмерном массиве определить сумму положительных элементов массива, расположенных до максимального элемента.

15. В заданном двухмерном массиве определить сумму элементов массива, расположенных до минимального элемента.

# Таблицы символов ASCII

## *Стандартная часть таблицы символов ASCII*

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

Некоторые из вышеперечисленных символов имеют особый смысл. Так, например, символ с кодом 9 обозначает символ горизонтальной табуляции, символ с кодом 10 – символ перевода строки, символ с кодом 13 – символ возврата каретки.

## *Дополнительная часть таблицы символов*

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

В таблицах обозначение **КС** означает «код символа», а **С** – «символ».

# Операции языка Си

Операции приведены в порядке убывания приоритета, операции с разными приоритетами разделены чертой.

| Опера-ция | Краткое описание | Использование | Порядок выполнения |
|---|---|---|---|
| ***Первичные (унарные) операции*** | | | |
| **.** | Доступ к члену | *объект* **.** *член* | Слева направо |
| **->** | Доступ по указателю | *указатель* -> *член* | |
| **[ ]** | Индексирование | *переменная* [*выражение*] | |
| **( )** | Вызов функции | ID_*функции*(*список*) | |
| ***Унарные операции*** | | | |
| **++** | Постфиксный инкремент | *lvalue*++ | Справа налево |
| **--** | Постфиксный декремент | *lvalue*-- | |
| **sizeof** | Размер объекта (типа) | sizeof(ID *или тип*) | |
| **++** | Префиксный инкремент | ++*lvalue* | |
| **--** | Префиксный декремент | --*lvalue* | |
| **~** | Побитовое НЕ | ~*выражение* | |
| **!** | Логическое НЕ | !*выражение* | |
| **– (+)** | Унарный минус (плюс) | – (+)*выражение* | |
| ***** | Разадресация | **выражение* | |
| **&** | Адрес | &*выражение* | |
| **()** | Приведение типа | (*тип*)*выражение* | |
| ***Бинарные и тернарная операции*** | | | |
| ***** | Умножение | *выражение* * *выражение* | Слева направо |
| **/** | Деление | *выражение* / *выражение* | |
| **%** | Получение остатка | *выражение* % *выражение* | |
| **+** | Сложение | *выражение* + *выражение* | |
| **–** | Вычитание | *выражение* – *выражение* | |
| **<<** | Сдвиг влево | *выражение* << *выражение* | |
| **>>** | Сдвиг вправо | *выражение* >> *выражение* | |
| **<** | Меньше | *выражение* < *выражение* | |
| **<=** | Меньше или равно | *выражение* <= *выражение* | |
| **>** | Больше | *выражение* > *выражение* | |
| **>=** | Больше или равно | *выражение* >= *выражение* | |
| **==** | Равно | *выражение* == *выражение* | |
| **!=** | Не равно | *выражение* != *выражение* | |
| **&** | Побитовое И | *выражение* & *выражение* | |
| **^** | Побитовое исключ. ИЛИ | *выражение* ^ *выражение* | |

| Операция | Краткое описание | Использование | Порядок выполнения |
|---|---|---|---|
| \| | Побитовое ИЛИ | *выражение* \| *выражение* | Слева направо |
| **&&** | Логическое И | *выражение* && *выражение* | |
| \|\| | Логическое ИЛИ | *выражение* \|\| *выражение* | |
| **?:** | Условная операция (***тернарная***) | *выражение* ? *выражение* : *выражение* | Справа налево |
| = | Присваивание | *lvalue* = *выражение* | |
| *= | Умножение с присваиванием | *lvalue* *= *выражение* | |
| /= | Деление с присваиванием | *lvalue* /= *выражение* | |
| %= | Остаток от деления с присваиванием | *lvalue* %= *выражение* | |
| += | Сложение с присваиванием | *lvalue* += *выражение* | |
| - = | Вычитание с присваиванием | *lvalue* -= *выражение* | |
| <<= | Сдвиг влево с присваиванием | *lvalue* <<= *выражение* | |
| >>= | Сдвиг вправо с присваиванием | *lvalue* >>= *выражение* | |
| &= | Поразрядное И с присваиванием | *lvalue* &= *выражение* | |
| \|= | Поразрядное ИЛИ с присваиванием | *lvalue* \|= *выражение* | |
| ^= | Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием | *lvalue* ^= *выражение* | |
| , | Последовательное вычисление | *выражение*, *выражение* | Слева направо |

# Возможности препроцессора

Препроцессор, как мы уже знаем, это программа предварительной обработки исходного текста программы перед этапом компиляции. Чаще всего препроцессор автоматически вызывается на этапе компиляции, если в исходном тексте обнаружена хотя бы одна его директива.

Признаком директивы препроцессора является символ #. При необходимости продолжения директивы в следующей строке текущую строку должен завершать символ '\'.

Возможности препроцессора языка Си:

– лексемное замещение идентификаторов;
– макрозамещение;
– включение файлов исходного текста;
– условная компиляция;
– изменение нумерации строк и текущего имени файла.

### *Директивы лексемного замещения идентификаторов*

Директива определения значения идентификатора (*ID*):

```
#define  ID  строка
```

В результате каждое вхождение в исходный текст элемента *ID* заменяется на значение элемента *строка*:

```
#define L_bufs 2048
#define binary int
#define WAIT fflush(stdin); getch()
#define BEEP sound(800);\
             delay(100);\
             nosound()
```

Лексемное замещение весьма удобно для сокращения записи повторяющихся фрагментов теста и определения символических констант:

```
#define YES 1
#define NO  2
#define ESC 27
#define Enter 30
```

которые могут быть в дальнейшем использованы:

```
if (x==ESC) break;
BEEP;
return(YES);
```

### *Директива отмены*

```
#undef  ID
```

Далее по исходному тексту можно назначить новое значение такого идентификатора.

***Макрозамещение***

Макрозамещение – обобщение лексемного замещения посредством параметризации строки директивы define в виде:

#define ID(*параметр*1,... ) *строка*

между элементом *ID* и открывающей скобкой пробелы не допускаются.

Такой вариант директивы define иногда называют макроопределением. Элемент *строка* обычно содержит параметры, которые препроцессором будут заменены на фактические аргументы так называемой макрокоманды, записываемой в формате

ID(*аргумент*1,... )

Пример макроопределения и макрокоманд:

```
#define P(X) printf("\n%s",X)
        . . .
        char *x;
        P(x);                       // Использование макроопределения P(X)
        P(" НАЧАЛО ОПТИМИЗАЦИИ");
        printf("\n%s",x);           // Эквивалентные операторы
        printf("\n%s"," НАЧАЛО ОПТИМИЗАЦИИ");
```

В строке макроопределений идентификаторы параметров сложных выражений рекомендуется заключать в круглые скобки:

```
#define MAX(A,B) ((A)>(B)? (A):(B))
#define ABS(X)    ((X)<0? –(X):(X))
```

Потребность в круглых скобках возникает при опасности искажения смысла вложенных выражений из-за действия правил приоритета операций. Пример искажения смысла операций:

```
#define BP(X) X*X
        . . .
int x,y,z;
x=BP(y+z);        ↔    x=y+z*y+z; ↔    x=y+(z*y)+z;
```

Очевидно, что ошибки будут и при следующих вариантах:

```
#define BP(X) (X*X)
#define BP(X) (X)*(X)
```

Безопасный вариант:

```
#define BP(X) ((X)*(X))
```

Иногда источником ошибок может быть символ «точка с запятой» в конце строки макроопределения:

```
#define BP(X) ((X)*(X));
        . . .
```

```
int x,y,z;
x=BP(z)–BP(y);   ↔     y=((z)*(z));          –((y)*(y));
```

Макроопределение отменяется директивой ***undef***.

Идентификаторы макроопределений обычно составляют из прописных букв латинского алфавита. Это позволяет отличать макрокоманды от вызова функций.

Макрокоманда внешне синтаксически эквивалентна операции вызова функции, но смысл их различен. Функция в программе имеется в одном экземпляре, но на ее вызов тратится время для подготовки параметров и передачи управления. Каждая макрокоманда замещается соответствующей частью макроопределения, но потерь на передачу управления нет.

### *Подключение файлов исходного текста*

Напомним, что имеются два варианта запроса включения в текущий файл содержимого другого файла. Директива

#include < ID_*файла*>

вводит содержимое файла из стандартного каталога (обычно – *include*), а директива

#include " ID_*файла*"

организует последовательный поиск в текущем, системном и стандартном каталогах. Например:

```
#include <alloc.h>              // Средства распределения памяти
#include <dos.h>                // Обращения к функциям ОС
#include "a:\prs\head.h"        // Включение файла пользователя
```

Рекомендуется описания системных объектов включать из стандартных каталогов и размещать их в начале файла исходного текста программы. Системные объекты в результате получают атрибут области действия «глобальный», что устраняет неоднозначность их описания.

### *Условная компиляция*

Директивы условной компиляции и реализуемые правила включения исходного текста:

а) условное включение (аналог работы оператора *if*):

```
#if<предикат_условия>
        ТЕКСТ_1
#endif
```

б) альтернативное включение (аналог *if-else*):

```
#if<предикат_условия>
        ТЕКСТ_1
#else
        ТЕКСТ_2
#endif
```

***Виды предикатов условий:***

константное_выражение → *истина*, если его значение ≠0;

*def* ID → *истина*, если *ID* был определен ранее оператором #*define*;

*ndef* ID → *истина*, если *ID* не был определен оператором #*define*.

Константное_выражение отделяется от ключевого слова *if* разделителем, а *def* и *ndef* – нет.

Пример:

```
#ifdef DEBUG
    print_state();
#endif
```

Элементы исходного текста «ТЕКСТ_1» или «ТЕКСТ_2» могут содержать любые директивы препроцессора.

Примеры:

```
#ifndef EOF
#define EOF –1
#endif
#if UNIT==CON
#include "conproc.c"
#else
#include "outproc.c"
#endif
```

***Изменение нумерации строк и идентификатора файла***

По умолчанию диагностические сообщения компилятора привязываются к номеру строки и *ID* файла исходного текста.

Директива

*#line номер_строки* ***ID****_файла*

позволяет с целью более приметной привязки к фрагментам текста изменить номер текущей строки и *ID* файла на новые значения («*ID_файла*» можно опустить).

# Интегрированная среда программирования *Visual* C++

Интегрированная среда разработки (*Integrated Development Environment*, или сокращенно *IDE*) – это программный продукт, объединяющий текстовый редактор, компилятор, отладчик и справочную систему.

Любая программа в среде *Visual* C++ всегда создается в виде отдельного проекта. Проект (*project*) – это набор взаимосвязанных исходных файлов и, возможно, включаемых заголовочных файлов, компиляция и компоновка которых позволяет создать исполняемую программу. Основу *Visual* C++ составляет рабочая область (*project workspace*). Она может содержать любое количество различных проектов, сгруппированных вместе для согласованной разработки: от отдельного приложения до библиотеки функций или целого программного пакета. Решение же простых (учебных) задач сводится к оформлению каждой программы в виде одного проекта, т.е. рабочая область проекта будет содержать ровно один проект.

### *Вид рабочего стола консольного приложения Visual C++*

После запуска *Visual* C++ появляется главное окно программы, вид которого приведен на рис. П 4.1. (В зависимости от настроек *Visual* C++ его вид может быть несколько иным.)

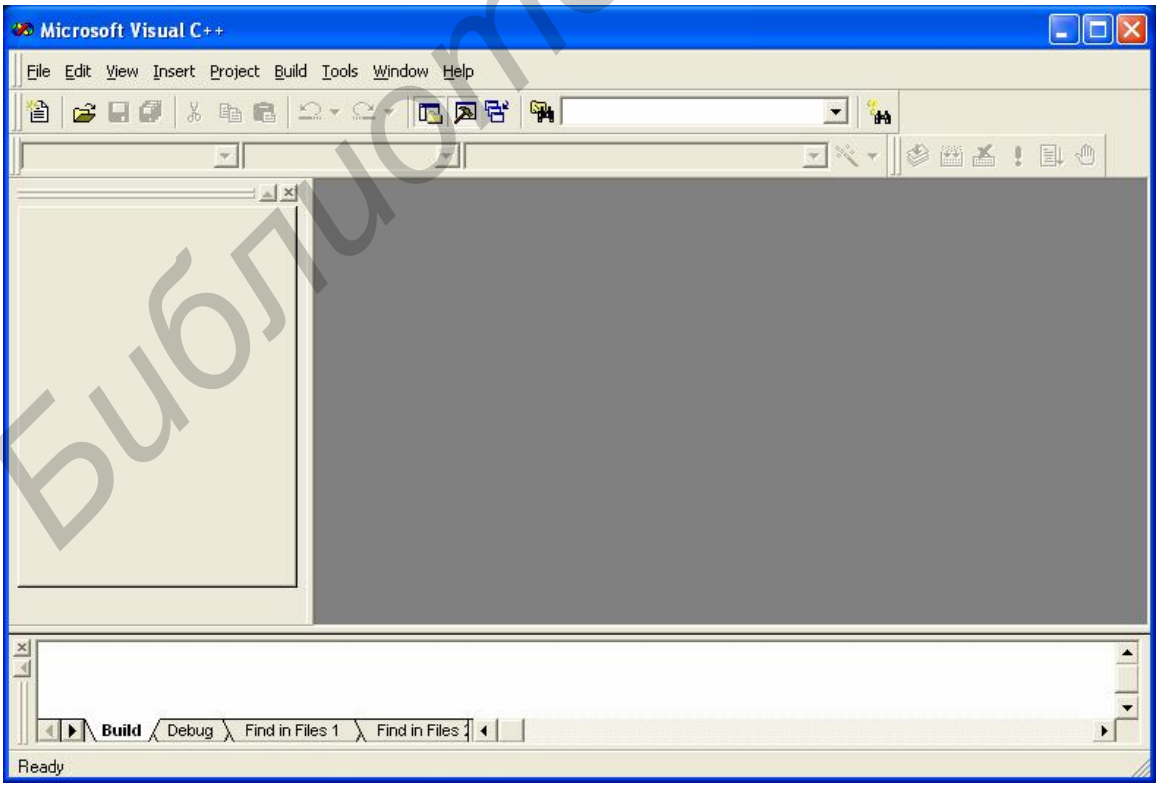



Рис. П 4.1

Экран *Visual* C++ разделен на четыре основные зоны.

Сверху расположены меню и панели инструментов. Кроме них рабочий стол *Visual* C++ включает в себя три окна:

Окно ***Project Workspace*** (окно рабочей области) – расположено в левой части. Первоначально окно закрыто, но после создания нового проекта или загрузки существующего проекта это окно будет содержать несколько вкладок.

Справа расположено окно ***Editor*** (окно редактирования). Его используют для ввода, проверки и редактирования исходного кода программы.

Окно ***Output*** (окно вывода) служит для вывода сообщений о ходе компиляции, сборки и выполнения программы и сообщений о возникающих ошибках.

Для кнопок панелей инструментов предусмотрена удобная контекстная помощь: если навести курсор мыши на кнопку и задержать на пару секунд, то всплывет подсказка с назначением данной кнопки.

***Developer Studio*** позволяет создавать проекты различных типов, которые ориентированы на различные сферы применения. Большинство типов проектов являются оконными *Windows*-приложениями с соответствующим графическим интерфейсом. Но также предусмотрена работа и с консольными приложениями. При запуске консольного приложения операционная система создает консольное окно, через которое идет весь ввод-вывод данных программы. Такая работа и представляет имитацию работы в операционной системе *MS DOS* или других операционных системах в режиме командной строки. Этот тип приложений больше всего подходит для целей изучения языка C/C++, т.к. не требует создания *Windows*-кода для пользовательского интерфейса. Рассмотрим приемы работы с консольными приложениями более подробно.

***Создание нового проекта***

Каждый новый проект рекомендуется помещать в созданную рабочую папку.

Для создания нового проекта типа «консольное приложение» выполните следующие действия:

1. Выберите в строке меню главного окна команду ***File/New***...

2. В открывшемся диалоговом окне *New* выберите вкладку ***Projects***:

– выберите тип проекта: ***Win32 Console Application***;

– введите имя проекта в текстовом поле ***Project Name***, например ***lr*1**;

– в текстовом поле ***Location*** введите имя каталога (полный путь к нему) для размещения будущих файлов проекта (если указанный вами каталог отсутствует, то он будет создан автоматически); путь к будущему проекту можно выбрать, щелкнув на кнопке, расположенной справа от текстового поля *Location*;

– выберите щелчком левой кнопкой мыши ***OK***.

3. Щелчок запустит встроенный мастер приложений: *Application Wizard*, который откроет диалоговое окно ***Win32 Console Application*** – *Step* 1 *of* 1 с предложением определиться, какой подтип консольного приложения желаете создать:

– выберите тип: ***An empty project*** (пустой проект);

– выберите щелчком ***Finish***.

4. После щелчка появится окно: ***New Project Information*** (информация о новом проекте) со спецификациями проекта и информацией о каталоге, в котором будет размещен создаваемый проект:

5. Подтвердите щелчком ***OK***.

После выполненных шагов рабочий стол примет вид, показанный на рис. П 4.2, а в папке *lr*1, созданной мастером приложений, будут находиться файлы ***First.dsw, First.dsp, First.opt, First.ncb*** и папка ***Debug*** (или *Release* – в зависимости от конфигурации проекта).

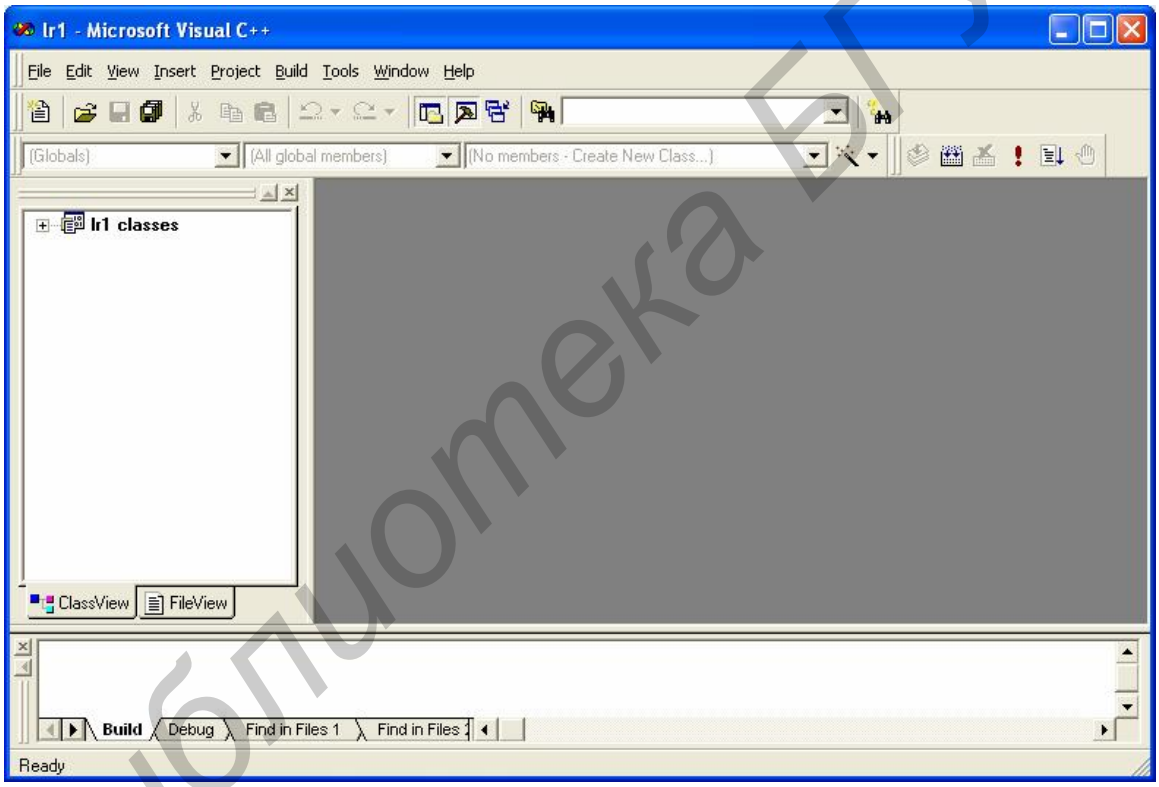



Рис. П 4.2

***Краткое описание каждого из файлов:***

***First.dsw*** – файл рабочей области проекта, используемый внутри интегрированной среды разработки. Он объединяет всю информацию о проектах, входящих в данную рабочую область.

***First.dsp*** – проектный файл, используемый для построения (*building*) отдельного проекта или подпроекта).

***First.opt*** – файл, содержащий опции рабочей области проекта. Благодаря этому файлу при каждом открытии рабочей области проекта все парамет-

ры *Developer Studio*, выбранные во время последнего сеанса работы с данной рабочей областью, будут восстановлены.

***First.ncb*** – служебный файл. Он создается компилятором и содержит информацию, которая используется в инструменте интегрированной среды под названием *ClassView*. Панель *ClassView* находится в окне *Project Workspace* и показывает все классы C++, для которых доступны определения в рамках данного проекта, а также все элементы этих классов. Если классы C++ не используются, данная панель будет пустой,

***Debug*** – папка, в которую будут помещаться файлы, формируемые компилятором и сборщиком. Из этих файлов нас будет интересовать только один – исполняемый файл, имеющий расширение **.exe*.

Как видно из рис. П 4.2, окно *Project Workspace* теперь открыто. В нем отображены две вкладки: ***Class View*** и ***File View***. Вкладка *Class View* пустая. Щелчком мыши переключаемся на вкладку *File View*. Она предназначена для просмотра списка файлов проекта. Выбрав щелчком мыши значок «+», откроем список *First files*. Появится дерево списка файлов, содержащее пиктограммы трех папок: ***Source Files*** (исходные коды), ***Header Files*** (заголовочные файлы), ***Resource Files*** (файлы ресурсов). Так как в консольных приложениях файлы ресурсов не используются, последняя папка всегда будет пустой. Первоначально и первые две папки пустые, т.к. в качестве подтипа консольного приложения был выбран пустой проект (опция *An empty project*).

***Добавление к проекту существующего файла***

Скопируйте исходный файл (*lr*1.*cpp*) в папку рабочей области проекта (в данном случае – *lr*1).

Вернитесь к списку *lr*1 *files* в окне *Project Workspace* проекта и щелкните правой кнопкой мыши на папке *Source Files*.

В появившемся контекстном меню щелчком мыши выберите команду добавления файлов ***Add Files to Folder***....

В открывшемся диалоговом окне ***Insert Files***... выберите нужный файл (*lr*1.*cpp*) и подтвердите выбор щелчком ***OK***.

***Создание и добавление к проекту нового файла***

В этом случае необходимо выполнить следующие действия:

Выберите в строке меню главного окна команду ***File/New***.... В результате откроется диалоговое окно ***New***.

На вкладке ***Files***:

– выберите тип файла (в данном случае: **C++ *Source File***);

– в текстовом поле ***File Name*** введите нужное имя файла (в данном случае: *lr*1.*cpp*);флажок ***Add to project*** должен быть включен;

– щелкните на кнопке ***OK***.

После этого получим следующие результаты:

– в окне *Project Workspace* раскроется папка *Source Files* списка файлов проекта и в нее будет помещен файл *lr*1.*cpp*;

– окно редактора *Editor* станет белым, а в его левом верхнем углу будет мерцать черный текстовый курсор, предлагая ввести какой-нибудь текст.

Введем, например, такой текст программы:

```
#include <stdio.h>
void main(void)
{
	printf(“\n Hello World! \n”);
}
```

В случае необходимости переключитесь на вкладку *File View*. Открываем список *rl*1 *files*, папку *Source Files* и убеждаемся, что в проекте создан файл *rl*1.*cpp* с только что набранным кодом.

Рис. П 4.3 иллюстрирует вид главного окна *Visual* C++ после проделанной работы:

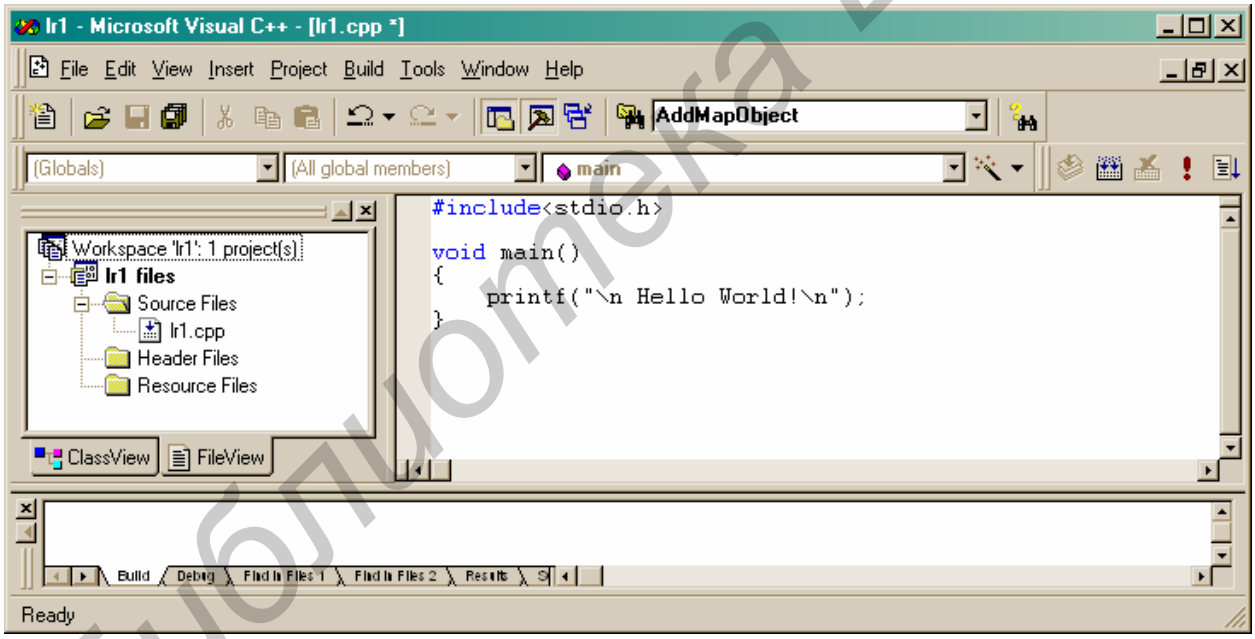



Рис. П 4.3

***Компиляция, компоновка и выполнение проекта***

Эти операции можно выполнить или через подменю *Build* главного окна, или при помощи кнопок панели инструментов, или при помощи комбинации горячих клавиш.

Данное подменю объединяет команды для компиляции, сборки и отладки программ. Рассмотрим основные команды меню ***Build***.

***Compile*** – компиляция выбранного файла, результаты которой выводятся в окно *Output*.

***Build*** – компоновка проекта. Компилируются все файлы, в которых произошли изменения с момента последней компоновки. После компиляции происходит сборка (*link*) всех объектных модулей, включая библиотечные, в результирующий исполняемый файл. Сообщения об ошибках компоновки выводятся в окно *Output*. Если обе фазы компоновки завершились без ошибок, среда программирования создаст исполняемый файл с расширением *.exe* (для данного примера: *lr*1.*exe*), который можно запустить на выполнение.

***Rebuild All*** – то же, что и *Build*, но компилируются все файлы проекта независимо от того, были ли в них произведены изменения или нет.

***Execute*** – запуск исполняемого файла, созданного в результате компоновки проекта. Если в исходный текст были внесены изменения, то – перекомпилирование, перекомпоновка и запуск на выполнение.

Операциям *Compile*, *Build* и *Execute* соответствуют первая, вторая и четвертая кнопки панели инструментов *Build MiniBar*, которая расположена на рабочем столе (см. рис. П 4.3) справа вверху рядом с системными кнопками. Перечислим их с указанием комбинаций горячих клавиш:

Compile = Ctrl+F7
Build = F7
Execute Program = Ctrl+F5

Откомпилируйте проект, например, выбрав щелчком *Build*. В процессе компиляции в окне вывода *Output* отображаются диагностические сообщения компилятора и сборщика. И если все в порядке, в последней строке окне вывода будет сообщение

lr1.exe – 0 error(s), 0 warning(s)

Теперь запускаем приложение на выполнение щелчком, например на кнопке *Execute Program* (*Ctrl*+*F*5). Появится окно приложения *rl*1, изображенное на рис. П 4.4.

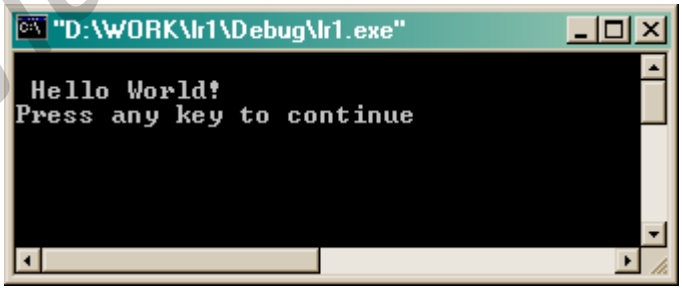



Рис. П 4.4

***Конфигурация проекта***

*Visual* C++ позволяет строить проект либо в отладочной конфигурации (***Win32 Debug***), либо в итоговой конфигурации (***Win32 Release***). Начинать работу нужно в отладочной конфигурации, которая обычно установлена по умолчанию. Для того чтобы проверить, какова текущая конфигурация в проекте, нужно выбрать в подменю ***Project*** пункт ***Settings...*** Откроется

диалоговое окно ***Project Settings***. Смотрим, какое значение установлено в окне комбинированного списка ***Settings For***:.. Если это не *Win32 Debug*, то переключитесь на нужное значение через команду меню *Build*/*Set Active Configuration*...

Но в отладочной конфигурации даже минимальный проект имеет очень большие размеры. Так, только что созданный проект *rl*1 имеет размер на диске более одного мегабайта. Поэтому после получения финальной версии проект нужно перекомпилировать в итоговой конфигурации (*Win32 Release*).

***Окончание работы над проектом***

Для окончания работы над проектом можно выбрать в меню ***File*** пункт ***Close Workspace***, а можно просто закрыть приложение *Visual* C++.

***Открытие существующего проекта***

Перечислим последовательность действий, необходимых для открытия существующего проекта.

1. Запустить на выполнение среду программирования *Visual* C++.
2. Выбрать в меню ***File*** пункт ***Open Workspace***...
3. В открывшемся диалоговом окне найти папку с нужным проектом, в ней – файл *ProjectName.dsw* и открыть найденный файл, щелкнув по нему мышью.

Если нужный проект был в работе не так давно, то:

1. Запустить на выполнение *Visual* C++.
2. Выбрать подменю ***File***, навести курсор мыши на пункт ***Recent Workspaces***.
3. Если в появившемся меню со списком последних файлов, с которыми шла работа, есть файл *ProjectName.dsw*, щелкнуть по нему мышью.

Или следующим образом:

– не вызывая *Visual* C++, найти папку с нужным проектом;

– выбрать щелчком мыши файл *ProjectName.dsw*, – ОС запустит на выполнение среду *Visual* C++, открыв при этом указанный проект.

# Некоторые возможности отладчика *Visual* C++

При создании проектов в любой среде программирования одним из важнейших этапов работы являются действия по отладке создаваемых программ.

Приведем начальные возможности отладчика *Visual* C++.

Проще всего это можно сделать следующим образом: написать программу, содержащую несколько ошибок, и после этого при помощи отладчика показать, как найти и исправить эти ошибки.

Напомним, что синтаксические ошибки – это результат нарушения формальных правил написания программы на конкретном языке программирования.

Логические ошибки делят на ошибки алгоритма и семантические ошибки. Причина ошибки алгоритма – несоответствие построенного алгоритма ходу получения конечного результата сформулированной задачи. Причина семантической ошибки – неправильное понимание смысла (семантики) операторов выбранного алгоритмического языка.

Создадим консольное приложение под названием *Test*1. В программе вычислим сумму первых пяти натуральных чисел (от 1 до 5). В результате должно получиться 15, а текст программы запишем следующий:

```
#include <stdio.h>
void main()
{
        int i, sum, n = 5, a[] = {1, 2, 3, 4, 5};
        for(i=1; i<n; i++)
                        sum += a[i];
        printf("\n Summa = %d\n", sum);
}
```

Как видно из текста, синтаксис программы правильный. Учимся отлавливать ошибки. Запускаем программу на выполнение и получаем на экране нечто такое:

*Summa* = – 858993446

на вашем компьютере может быть и другое число, но мало похожее на 15.

Начинаем отладку программы.

### *Установка точки прерывания*

Точка прерывания позволяет остановить выполнение программы перед любой *выполняемой* инструкцией для того, чтобы или продолжать дальнейшее выполнение программы, или продолжать программу в пошаговом режиме с целью отладки ее последующих участков, или продолжать выполнение программы в непрерывном режиме до конца программы или следующей точки прерывания.

Для того чтобы установить точку прерывания перед оператором, необходимо установить перед ним текстовый курсор и/или нажать клавишу ***F*9** или выбрать щелчком мыши ***Insert/Remove Breakpoint*** на панели инструментов ***Build MiniBar*** (крайняя правая кнопка). Точка прерывания обозначится в виде красного кружочка на левом поле окна редактирования перед выбранным оператором. Повторный щелчок на указанной кнопке снимает точку прерывания. В программе точек прерывания можно устанавливать столько, сколько нужно.

***Выполнение программы до точки прерывания***

Программа запускается в отладочном режиме с помощью команды подменю ***Build*: *Start Debug* > *Go*** (или нажатием клавиши ***F*5**).

В результате код программы выполняется до той строки, на которой установлена точка прерывания. После чего программа останавливается и отображает в окне редактора ту часть кода, где находится точка прерывания. При этом появляется желтая стрелка на левом поле, указывающая на ту строку, которая будет выполнена на следующем шаге отладки.

Далее, установив точку прерывания перед оператором *for*, запустим программу в отладочном режиме, нажав клавишу ***F*5**. Программа выполнится до данного оператора и остановится. При этом подменю ***Build*** заменилось на меню ***Debug***. Вид экрана изображен на рис. П 5.1.

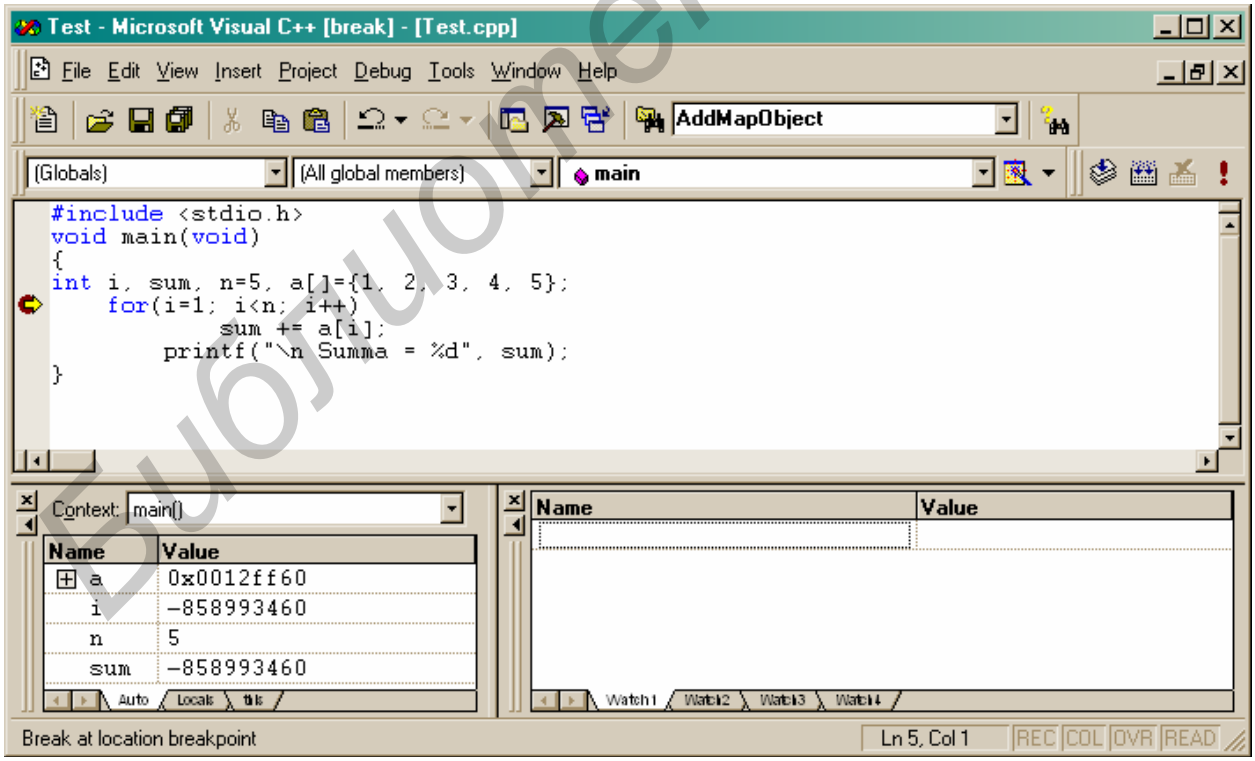



Рис. П 5.1

### *Пошаговое выполнение программы*

Теперь можно выполнять один за другим операторы оставшейся части программы, нажимая клавишу ***F*10**. При этом есть следующие дополнительные возможности.

1. Предположим, что код программы содержит функцию пользователя *f*1() и что при пошаговом выполнении программы вы дошли до строки, содержащей вызов этой функции.

Возможны две ситуации:

– если необходимо просмотреть код вызываемой функции, то надо нажать клавишу ***F*11**;

– если нужен только результат ее выполнения, то надо нажать клавишу ***F*10**.

2. Далее, пусть мы проходим код функции *f*1(), нажав клавишу *F*11. Нажатие клавиш ***Shift*+*F*11** обеспечит досрочный выход из нее в точку вызова.

3. Существует возможность пропустить пошаговое выполнение некоторого участка программы. Для этого нужно установить курсор в нужное место программы и нажать клавиши ***Ctrl*+*F*10**.

Продолжаем отладку, нажав клавишу ***F*10**, после чего указатель переместится на оператор:

*Summa += a[i];*

#### *Проверка значений переменных во время выполнения программы*

В любой момент можно узнать текущее значение любой переменной. Чтобы узнать, например, значение переменной ***summa***, в которой будет накапливаться сумма элементов массива *a*, установите и задержите над ней указатель мыши. Рядом с именем переменной на экране появляется подсказка со значением этой переменной:

*Summa = – 858993446*

(или с другим произвольным значением).

Переменная ***summa*** уже ненулевая, хотя еще не было никакого суммирования, т.к. допущена ***ошибка 1*** – не обнулена переменная накопления ***summa*** до входа в цикл по организации суммирования.

Нажимаем комбинацию клавиш ***Shift*+*F*5**, чтобы выйти из отладчика, и исправляем найденную ошибку:

*int i, Summa=0, n=5;*

снимаем точку прерывания, компилируем – *Build* (*F*7), запускаем на выполнение – *Execute Program* (*Ctrl*+*F*5) и получаем новый результат:

*Summa = 14*

Все равно неправильно, необходимо продолжить отладку.

Вновь устанавливаем точку прерывания перед оператором *for*, запускаем программу в отладочном режиме – клавиша *F*5, нажимаем клавишу *F*10 – указатель следующей выполняемой команды переместился на оператор

*Summa += a*[*i*];

нажимаем клавишу *F*10 – указатель следующей выполняемой команды переместится на оператор

for(i = 1; i<n; i++)

Опять указатель мыши – над переменной ***summa***, рядом с именем переменной на экране появится подсказка со значением этой переменной

*Summa = 2*

Сделана только 1-я итерация цикла, переменная ***Summa*** должна быть равна значению первого элемента массива 1, а у нас число 2, т.е. значение второго элемента массива.

***Ошибка 2*** находится в заголовке цикла for(i = 1; i < n; i++) – нумерация элементов массива в языке Си начинается с нуля, а у нас $i = 1$ – первоначально из цикла выдергивается второй по счету элемент, а не первый.

Выйдя из отладчика, снимаем точку прерывания, исправляем текст программы в операторе *for*:

for(i = 0; i < n; i++)

Вновь компилируем, запускаем на выполнение и получаем требуемый результат:

*Summa = 15*

## *Окна Auto и Watch 1*

Помимо экранной подсказки, переменная sum со своим значением отображается в окне *Auto*, расположенном в левом нижнем углу экрана (см. рис. П 5.1). В этом окне приведены значения последних переменных, с которыми работал *Visual* C++.

Кроме этого, в окне *Watch*, которое находится в правом нижнем углу, можно задать имя любой переменной, за значениями которой вы хотите понаблюдать.

В заключение напомним, что после отладки проекта необходимо построить его финальную версию, т.к. в ***EXE*-файле** есть дополнительный отладочный код, включенный через директивы компилятора ***_DEBUG.*** В папке ***Debug*** его размер приблизительно равен 1 Мб. Для того чтобы получить нормальную версию, необходимо переключиться на проект типа ***Release.*** Это делается в меню ***Build*:** далее ***Set Active Configuration,*** дальше надо выбрать ***Release*** и запустить сборку: ***Rebuild all***. Весь отладочный код пропускается, в каталоге проекта появится папка ***Release***. Теперь размер ***EXE*-файла** около 100 Кб.

Размер проекта еще зависит от того, как используется библиотека ***MFC***. Как вы уже знаете – есть два варианта: ***Static Library*** и ***Dinamic Library***. Если использовать первый вариант, то код ***MFC*** будет встроен в проект и размер возрастет. Во втором варианте программа будет пользоваться файлами **.DLL* и размер будет меньше.

### *Программные средства отладки*

При создании приложения на базе *MFC* в него можно включить определенные инструкции, используемые только во время его отладки. Их использование замедляет выполнение программы. Цель использования: получение дополнительных сообщений, которые облегчают процесс отладки. При создании окончательной версии приложения эти операторы следует удалить.

Режим компиляции приложения должен быть *Debug* (отладочный), т.к. инструкции – специальные функции и макросы библиотеки *MFC*, работающие только в отладочном режиме. Кроме того, в отладочной версии приложения *Visual* C++ с помощью директивы ***#define*** определяется константа ***_DEBUG***, используемая в директивах условной компиляции и в макросах отладки.

***Макрос ASSERT (BOOLexpression).*** Данный макрос используется для проверки некоторых логических условий, которые должны выполняться в данной точке программы. Его работа заключается в следующем: если логическое выражение *BOOLexpression*, передаваемое ему в качестве аргумента, имеет значение ***FALSE***, выполнение приложения прерывается и на экран выводится окно сообщения, показанное на рис. П 5.2. В данном окне указывается имя файла и номер строки, в которой произошла ошибка.

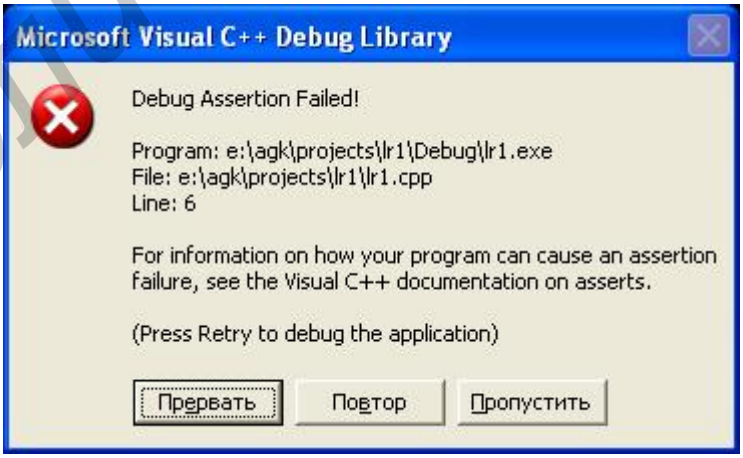



Рис. П 5.2

Нажатие кнопки «*Повтор*» позволяет перейти в текст программы для ее дальнейшей отладки, причем текущая точка останова устанавливается на

строку соответствующего макроса ***ASSERT***. В противном же случае ничего не происходит, и программа выполняется дальше.

***Макрос TRACE (exp)*** служит для вывода диагностических сообщений. Синтаксис макроса *TRACE* аналогичен синтаксису функции *printf*. Здесь ***exp –*** переменное число параметрических аргументов, т.е. макрос позволяет выводить сообщения с любым числом параметрических аргументов. Под параметрическим аргументом понимается идентификатор переменной, значение которой должно быть преобразовано в текстовую строку в соответствии с указанным форматом. Помимо макроса *TRACE* существуют *TRACE*1, *TRACE*2 и *TRACE*3. Число в имени макроса указывает на количество параметрических аргументов в нем. Макросы *TRACE*0, *TRACE*1, *TRACE*2 и *TRACE*3 созданы исключительно с целью экономии места в сегменте данных. Все макросы *TRACE* посылают свои сообщения в поток *afxDump*.

Для посылки сообщения могут быть использованы и обычные функции вывода, однако они будут работать и в окончательной версии, что бывает нежелательно.

Пример вывода сообщения о возникновении ошибки в файле:

```
TRACE2("\n Ошибка номер: %d в файле %s \n", nError, szFileName);
```

Следующий пример иллюстрирует работу данного макроса:

```
        . . .
int a = 5;
char s[] = ”Minsk”;
TRACE(“\n a = %d, s = %s \n”, a, s);
        . . .
```

В поле среды *OutPut* получим

*a=5, s=Minsk*

В окончательной версии приложения *Release*, в которой константа *_DEBUG* не определена, макросы *ASSERT* и *TRACE* не выполняют никаких действий. Это позволяет оставлять их в тексте программы. В случае же необходимости контроля некоторых логических условий и в рабочей области вместо *ASSERT* необходимо использовать макрос *VERIFY* (*BOOLexpression*), который работает точно так же, но в рабочей версии проекта.

# Некоторые возможности графической подсистемы

## *6.1. Основные понятия*

В операционной системе *Windows* для создания программ с использованием графики существует интерфейс программирования приложений *API* (*Application programming interface*).

Графическая информация в *Windows* обрабатывается в основном функциями *GDI (Graphics Device interface)*, представляющими собой единый унифицированный интерфейс устройств (средств) отображения.

Поскольку к ЭВМ может быть подключено множество различных устройств отображения, одной из основных задач *GDI* является поддержка аппаратно-независимой графики.

Все графические устройства отображения делятся на растровые и векторные устройства. Большинство устройств, подключаемых к ЭВМ – растровые, т.е. представляют графические образы как шаблон точек (видеоадаптеры, матричные и лазерные принтеры). Группа векторных устройств, отображающих графические образы с использованием линий, в основном состоит из плоттеров.

## *6.2. Контекст устройства*

Работа *GDI* базируется на понятии *контекст устройства* (*DC – device context*), абстрагирующего свойства реальных устройств, к которым в первую очередь относятся экран, принтер и битовый образ в памяти. Контекст является внутренним объектом *Windows*, и доступ к нему осуществляется с помощью функций *API*. Контекст идентифицируется его описателем, имеющим тип *HDC* (*handle DC*). Практически каждой функции *GDI* необходим этот параметр.

Ввиду того, что прикладная программа *никогда* не манипулирует контекстом непосредственно, используя для доступа к контексту его описатель, в дальнейшем термин *контекст* будем употреблять как к системной структуре данных, так и к ее описателю.

Контекст сопоставляется системой с каждым изображаемым элементом (чаще всего – окном) и может быть получен прикладной программой, после чего можно обращаться к нему с единым набором функций, причем поведение контекста будет одинаковым независимо от того, с каким устройством он связан.

## *6.3. Примитивы GDI*

Основные типы графических объектов часто называют «*примитивами*». К ним относятся:

– ***прямые (отрезки) и кривые***; *GDI* поддерживает прямые линии, прямоугольники, эллипсы (включая окружности), дуги и сплайны Безье; более
сложные кривые могут быть изображены как ломаные линии, состоящие из коротких прямых, определяющих кривые; линии рисуются с использованием графического объекта ***пера***, выбранного в контексте устройства;

– ***закрашенные области*;** если набор прямых и кривых линий ограничивает со всех сторон некоторую область, то она может быть закрашена с использованием ***кисти***, выбранной в контексте устройства;

– ***битовые шаблоны*** (растровые шаблоны, растровые образы) – это двухмерный массив бит, соответствующий пикселам устройства отображения (базовый инструмент растровой графики); битовые образы используются для отображения сложных изображений (значки, курсоры мыши, кнопки панели инструментов);

– ***текст***, отличающийся от других объектов графики. Так как типов текста много и структуры данных, используемые для описания шрифтов (объектов *GDI*), а также получения информации о них – самые большие среди других структур данных в *Windows*, поддержка текста – часто наиболее сложная часть в системах компьютерной графики.

### *6.4. Пример вывода текста*

Чтобы легче было разобраться в некоторых аспектах работы с графикой в среде *Windows*, рассмотрим программу, создающую окно, в котором выводится текст «*Hello, Windows* !» [34].

В тексте программы свыше 80 строк, большая часть из которых является надстройкой и будет повторяться почти в каждой программе для *Windows*.

```
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Hello" ;
     HWND   hwnd ;
     MSG      msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize         = sizeof (wndclass) ;
     wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION);
```

```
        wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW);
        wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
        wndclass.lpszMenuName = NULL ;

        wndclass.lpszClassName = szAppName ;
        wndclass.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;

            RegisterClassEx (&wndclass) ;

            hwnd = CreateWindow (szAppName,
                        "First Example",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL,       NULL,
                        hInstance,    NULL) ;
            ShowWindow (hwnd, iCmdShow) ;
            UpdateWindow (hwnd) ;
            while (GetMessage (&msg, NULL, 0, 0)) {
                        TranslateMessage (&msg) ;
                        DispatchMessage (&msg) ;
            }
            return msg.wParam ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM
wParam, LPARAM lParam)
{
          HDC     hdc ;
          PAINTSTRUCT ps ;
          RECT   rect ;
          switch (iMsg) {
           case WM_PAINT :
                    hdc = BeginPaint (hwnd, &ps) ;
                    GetClientRect (hwnd, &rect) ;
                    DrawText (hdc, "Hello, Windows !", -1, &rect,
                              DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
                    EndPaint (hwnd, &ps) ;
            return 0 ;
            case WM_DESTROY :
                     PostQuitMessage (0) ;
            return 0 ;
          }
           return DefWindowProc (hwnd, iMsg, wParam, lParam);
}
```

В программе создается стандартное окно (рис. П 6.1), в центре рабочей области которого выводится текст «*Hello, Windows* !».

Это окно имеет все свойства *Windows*: можно захватить указателем мыши заголовок окна и перемещать его по всему экрану, можно изменить размеры окна, можно развернуть и увеличить его до размеров всего экрана, можно свернуть или завершить программу кнопкой закрытия окна.

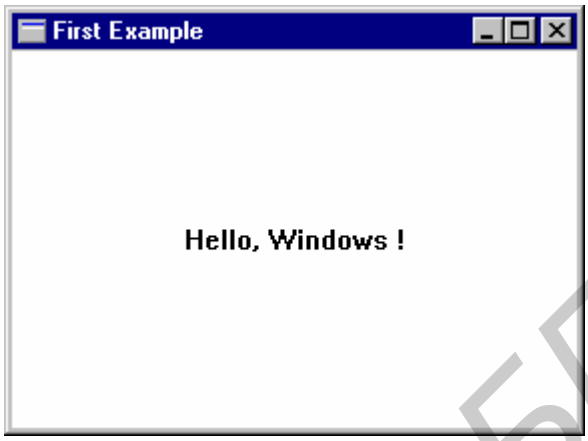



Рис. П 6.1. Результат программы *Hello*

Рассмотрим действия данной программы.

***Стандартные функции Windows***

В программе используются функции, описанные в различных заголовочных файлах из стандартной библиотеки ***windows.h***:

*LoadIcon* – загружает значок (*Icon*) для использования в программе;

*LoadCursor* – загружает курсор (*Cursor*) мыши;

*GetStockObject* – получает графический объект – кисть (*Stock*);

*RegisterClassEx* – регистрирует класс окна;

*CreateWindow* – создает окно на основе класса окна;

*ShowWindow* – выводит окно на экран;

*UpdateWindow* – заставляет окно перерисовать свое содержимое;

*GetMessage* – получает сообщение из очереди сообщений;

*TranslateMessage* – преобразует некоторые сообщения, полученные с помощью клавиатуры;

*DispatchMessage* – отправляет сообщение оконной процедуре;

*BeginPaint* – инициирует начало процесса рисования окна;

*GetClientRect* – получает размер рабочей области окна;

*DrawText* – выводит на экран строку текста;

*EndPaint* – прекращает рисование окна;

*PostQuitMessage* – вставляет сообщение «Завершить» в очередь сообщений;

*DefWindowProc* – выполняет обработку сообщений по умолчанию.

***Идентификаторы и типы данных***

При программировании в среде *Windows* используют соглашения по именованию переменных – идентификатор переменной составляется из букв или частей слов, отражающих смысл этой переменной.

Префиксы некоторых переменных, использующихся в дальнейшем: ***c*** – символ; ***by*** – *BYTE* (беззнаковый символ); ***n*** – короткое целое; ***i*** – целое; ***cx, cy*** – целое (длины *x* и *y*; ***c*** означает счет – *count*); ***b*** или ***f*** – *BOOL* (булево целое, ***f*** – флаг – *flag*); ***w*** – *WORD* (беззнаковое короткое целое); ***l*** – *LONG* (длинное целое); ***dw*** – *DWORD* (беззнаковое длинное целое); ***fn*** – функция; ***s*** – строка; ***sz*** – строка, завершаемая нулем (*string terminated by zero*); ***h*** – описатель (*handle*); ***p*** – указатель (*pointer*).

Идентификаторы, написанные прописными буквами, задаются в заголовочных файлах *Windows*. Двух- или трехбуквенный префикс, за которым следует символ подчеркивания, показывает основную категорию ее принадлежности, например: *CS* – опция стиля класса (*Class Style*); *IDI* – идентификационный номер иконки (*ID Icon*); *IDC* – идентификационный номер курсора; *WS* – стиль окна (*windows style*); *WM* – сообщение окна.

Аналогичен смысл новых типов данных, например, тип ***UINT*** – 32-разрядное беззнаковое целое (*unsigned int*), ***PSTR*** – указатель на строку символов (*pointer string*), т.е. *char**; ***LONG*** – длинное целое.

*WndProc* возвращает значение типа *LRESULT – Long RESULT*. Функция *WinMain* получает тип *WINAPI* (как и любая другая функция *Windows*), а функция *WndProc* получает тип *CALLBACK* – эти идентификаторы являются ссылкой на особую последовательность вызовов функций, которая имеет место между ОС *Windows* и ее приложением.

В программе использованы структуры данных: ***MSG*** – структура сообщения (*message*); ***WNDCLASSEX*** – структура класса окна; ***PAINTSTRUCT*** – структура рисования; ***RECT*** – структура прямоугольника.

При обозначении переменных структуры пользуются именем самой структуры и строчными буквами, например, переменная *msg* – структура типа *MSG*; *wndclass* – структура типа *WNDCLASSEX*.

В программе используются идентификаторы, предназначенные для разных типов описателей (*handles*): ***HINSTANCE*** – описатель экземпляра (*instance*) самой программы; ***HWND*** – описатель окна (*handle to a window*); ***HDC*** – описатель контекста устройства.

***Основная программа***

Для работы программы подключен файл ***windows.h***, содержащий заголовочные файлы с объявлениями функций, структур и числовых констант. Далее следует декларация прототипа функции *WndProc* – «оконная процедура» для окна программы *Hello*.

Основной функцией (точкой входа программы), аналогом стандартной функции *main* языка Си для *Windows* является функция *WinMain*:

int WINAPI ***WinMain*** (HINSTANCE hInstance, HINSTANCE hPrevInstance,
PSTR szCmdLine, int iCmdShow);

использующая последовательность вызовов *WINAPI* и возвращающая ОС *Windows* целое значение. В ней используются четыре параметра:

*hInstance* – описатель экземпляра – уникальное число, идентифицирующее программу;

*hPrevInstance* – предыдущий (*previous*) экземпляр; если в данный момент не было загружено копий программы, то *hPrevInstance* = 0 или *NULL*;

*szCmdLine* – указатель на строку, в которой содержатся любые параметры, переданные в программу из командной строки;

*iCmdShow* – число, показывающее, каким должно быть выведено на экран окно в начальный момент; обычно: *SW_SHOWNORMAL* (1) – вывод окна нормального размера, *SW_SHOWMINNOACTIVE* (7) – окно должно быть изначально свернутым; префикс *SW* означает «показать окно» (*show window*).

***Регистрация класса окна***

Окно всегда создается на основе класса окна. Класс окна идентифицирует оконную процедуру, которая выполняет процесс обработки поступающих сообщений.

Перед созданием окна необходимо зарегистрировать класс окна, вызвав функцию ***RegisterClassEx***, это расширенная версия функции *Register-Class* (*Ex – extended* – расширенный). Параметр функции – указатель на структуру типа *WNDCLASSEX*, определенную в заголовочных файлах:

```
typedef struct tagWNDCLASSEX  {
    UINT        cbSize;
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCSTR      lpszMenuName;
    LPCSTR      lpszClassName;
    HICON       hIconSm;
}  WNDCLASSEX;
```

Префиксы *LP* и *lp* означают «длинный указатель» (*long pointer*), приставка *lpfn* – «длинный указатель на функцию» (*long pointer to a function*);

приставка *cb* – «счетчик байт» (*counter of bytes*); префикс *hbr* – это «описатель кисти» (*handle to a brush*).

Для работы *WinMain* необходимо определить эту структуру:

WNDCLASSEX ***wndclass***;

наиболее важными в ней являются второе и третье от конца поля. Второе поле – имя класса окна (обычно совпадает с именем программы), третье поле – адрес оконной процедуры, использующейся для всех окон, созданных на основе этого класса. Другие поля описывают характеристики окон, создаваемых на основе этого класса окна.

Поле *cbSize* равно длине структуры. Инструкция:

wndclass.style = CS_HREDRAW | CS_VREDRAW;

осуществляет объединение двух идентификаторов «стиля класса» с помощью поразрядной операции «ИЛИ» (|). В заголовочных файлах *Windows*, идентификаторы, начинающиеся с префикса *CS*, задаются в виде 32-разрядной константы, только один из разрядов которой установлен в 1. Например, *CS_VREDRAW* (горизонтальный) – 0x0001, а *CS_HREDRAW* (вертикальный) – 0x0002 показывают, что все окна должны целиком перерисовываться при изменении размеров окна. Эти идентификаторы называют «поразрядными флагами» (*bit flags*).

Третье поле *wndclass.lpfnWndProc = WndProc*; устанавливает оконную процедуру данного окна.

Следующие две инструкции:

wndclass.cbClsExtra = 0; wndclass.cbWndExtra = 0;

резервируют дополнительное пространство, которое может быть использовано программой, в данной программе эта возможность не используется – значения равны 0, иначе в этом поле было бы установлено «число байт» резервируемой памяти.

В следующем поле – описатель экземпляра программы, который является одним из параметров *WinMain*:

wndclass.hInstance = *hInstance*;

Инструкции

wndclass.hIcon = ***LoadIcon*** (NULL, IDI_APPLICATION);

и

wndclass.hIconSm = ***LoadIcon*** (NULL, IDI_APPLICATION);

устанавливают значки (иконки), которые появляется на панели задач и в заголовке окна. Для получения описателя стандартного значка первый параметр устанавливается в *NULL*; при загрузке пользовательского значка этот параметр должен быть равен описателю экземпляра программы. Второй параметр определяется в заголовочных файлах, значок *IDI_APPLICATION* – это маленькое изображение окна. Функция *LoadIcon* возвращает описатель этого значка для полей *wndclass.hIcon* и *wndclass.hIconSm*, имеющих тип *HICON* («описатель значка» – *handle to an icon*).

Функция *LoadCursor* загружает стандартный курсор *IDC_ARROW* и возвращает его описатель полю *hCursor*:

wndclass.hCursor = ***LoadCursor*** (NULL, IDC_ARROW);

Поле *hbrBackground* задает цвет фона рабочей области окон (префикс *hbr* означает «описатель кисти» – *handle to a brush*). ***Кисть*** – графический объект, используемый для закрашивания области. В *Windows* имеется несколько стандартных кистей. В инструкции

wndclass.hbrBackground = ***GetStockObject*** (WHITE_BRUSH);

функция *GetStockObject* возвращает описатель белой кисти.

Следующее поле задает меню класса окна, т.к. в нашем случае оно отсутствует – поле установлено в *NULL*:

wndclass.lpszMenuName = NULL;

На последнем этапе классу должно быть присвоено имя, которое может совпадать с именем программы, хранящемся в переменной *szAppName*:

wndclass.lpszClassName = szAppName;

После того как инициализированы все поля структуры, регистрируем класс окна, вызывая функцию: ***RegisterClassEx*** (&*wndclass*);

***Создание окна***

Класс окна определяет основные характеристики окна, что позволяет использовать один и тот же класс для создания множества различных окон.

Функция *CreateWindow* создает окно, детализируя информацию о нем, которая должна передаваться функции в качестве параметров:

```
hwnd = CreateWindow (szAppName,          – имя класса окна;
       "First Example",                  – заголовок окна;
       WS_OVERLAPPEDWINDOW,              – стиль окна;
       CW_USEDEFAULT,                    – начальное положение по x
       CW_USEDEFAULT,                      и по y;
       CW_USEDEFAULT,                    – начальные размеры по x
       CW_USEDEFAULT,                      и по y;
       NULL,                             – описатель родительского окна;
       NULL,                             – описатель меню окна;
       hInstance,                        – описатель экземпляра программы;
       NULL);                            – параметры создания.
```

Параметр «имя класса окна» – *szAppName* содержит строку «*Hello*», являющуюся именем зарегистрированного класса окна, этот параметр связывает окно с классом окна.

Окно, созданное программой, является обычным перекрывающимся окном с заголовком, системным меню слева на строке заголовка, иконками для сворачивания, разворачивания и закрытия окна.

*WS_OVERLAPPEDWINDOW* – стандартный «стиль окна»; «заголовок окна» – текст, который выводится в строке заголовка.

Параметры «начальное положение по *x* и по *y*» задают начальные координаты верхнего левого угла окна относительно левого верхнего угла экрана; идентификатор *CW_USEDEFAULT* задает по умолчанию начальное положение для перекрывающегося окна; аналогично задаются значения параметров «начальные размеры по *x* и по *y*».

Поскольку у нашего окна отсутствует родительское окно, параметр «описатель родительского окна» – *NULL*; аналогично «описатель меню окна» – *NULL* (у окна нет меню).

В параметр «описатель экземпляра программы» помещается описатель экземпляра, переданный программе в качестве параметра функции *WinMain*.

Значение «параметры создания» установлено в *NULL*, при необходимости этот параметр используется в качестве указателя на данные, к которым программа в дальнейшем могла бы обратиться.

Функция *CreateWindow* возвращает описатель созданного окна, который хранится в переменной *hwnd* типа. У каждого окна в *Windows* имеется его описатель, который используется для ссылки на это окно. При создании нескольких окон каждое из них имеет свой описатель, являющийся одним из важнейших, которыми оперирует программа для *Windows*.

Для многих функций *Windows* в качестве параметра требуется *hwnd*, благодаря чему *Windows* знает, к какому окну применить эту функцию.

### *Отображение окна*

К тому времени, когда функция *CreateWindow* возвращает управление программе, окно уже создано внутри *Windows*, однако на экране монитора оно еще не появилось. Выводит окно на экран функция

***ShowWindow*** (hwnd, iCmdShow);

параметр *hwnd* – описатель созданного окна; параметр *iCmdShow* – задает начальный вид окна на экране: *SW_SHOWNORMAL* (1) – на экран выводится обычное окно, фон рабочей области закрашивается заданной в классе окна кистью; *SW_SHOWMINNOACTIVE* (7) – окно не выводится, а на панели задач появляются его имя и иконка.

Функция ***UpdateWindow*** (*hwnd*); выполняет перерисовку рабочей области окна, посылая сообщение *WM_PAINT* в оконную процедуру (*WndProc*).

### *Цикл обработки сообщений*

Программа получает информацию от пользователя через клавиатуру и мышь. Для каждой программы, работающей в данный момент, *Windows* поддерживает «очередь сообщений» (*message queue*). Когда происходит ввод информации, она преобразуется в «сообщение», которое помещается в эту очередь.

Программа извлекает сообщения из очереди, выполняя блок соответствующих команд, известный как «цикл обработки сообщений» (*message loop*):

```
while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
}
return msg.wParam;
```

Переменная *msg* – структура типа *MSG*, определенная следующим образом:

```
typedef struct tagMSG {
        HWND    hwnd;
        UINT    message;
        WPARAM  wParam;
        LPARAM  lParam;
        DWORD   time;
        POINT   pt;
} MSG;
```

Тип данных *POINT* определяется как

```
typedef struct tagPOINT {
        LONG x;
        LONG y;
} POINT;
```

Вызов функции

```
GetMessage (&msg, NULL, 0, 0);
```

извлекает сообщение из очереди и в *Windows* передается указатель на структуру *msg*; второй, третий и четвертый параметры (*NULL* или 0) показывают, что программа получает все сообщения от всех окон. Система *Windows* заполняет поля структуры сообщений информацией об очередном сообщении из очереди. Поля этой структуры следующие:

*hwnd* – описатель окна, для которого предназначено сообщение;

*message* – идентификатор сообщения, который задается в заголовочных файлах и начинается с префикса *WM* (оконное сообщение – *window message*);

*wParam* – 32-разрядный параметр сообщения, смысл и значение которого зависят от особенностей сообщения;

*lParam* – 32-разрядный параметр, зависящий от сообщения;

*time* – время, когда сообщение было помещено в очередь;

*pt* – координаты курсора в момент помещения сообщения в очередь.

Если поле *message* сообщения, извлеченного из очереди, равно любому значению, кроме *WM_QUIT* (0x0012), то функция *GetMessage* возвращает ненулевое значение; сообщение *WM_QUIT* прерывает цикл обработки сообщений – программа заканчивается, возвращая число *wParam* структуры *msg*.

Функция ***TranslateMessage*** (&msg); передает структуру *msg* обратно в *Windows* для преобразования сообщения.

Функция ***DispatchMessage*** (&*msg*); передает структуру *msg* для ее обработки соответствующей оконной процедурой (*WndProc*). После того как *WndProc* обработает сообщение, оно возвращается в *Windows*, которая все еще обслуживает вызов функции *DispatchMessage*. Когда *Windows* возвращает управление в программу к следующему за вызовом *DispatchMessage* коду, цикл обработки сообщений в очередной раз возобновляет работу, вызывая *GetMessage*.

### *Оконная процедура*

Оконная процедура определяет, что выводится в рабочую область окна и как окну реагировать на пользовательский ввод.

Оконной процедуре (*WndProc*) можно назначить любое имя. В программе может содержаться несколько оконных процедур. Оконная процедура всегда связана с определенным классом окна, который регистрируется функцией *RegisterClassEx*. Функция *CreateWindow* создает окно на основе определенного класса окна. На основе одного и того же класса можно создавать несколько окон.

Оконная процедура определяется следующим образом:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg,
                          WPARAM wParam, LPARAM lParam);
```

ее параметры идентичны первым четырем полям структуры *MSG*.

### *Обработка сообщений*

Каждое получаемое окном сообщение идентифицируется номером, содержащемся в параметре *iMsg* оконной процедуры.

Для определения того, какое сообщение получила оконная процедура и как его обрабатывать, обычно используют оператор *switch*. Если оконная процедура обрабатывает сообщение, то ее возвращаемым значением должен быть 0. Все сообщения, не обрабатываемые оконной процедурой, передаются функции *DefWindowProc*, результат которой должен быть возвращаемым значением оконной процедуры.

Функция *WndProc* обрабатывает два сообщения ***WM_PAINT*** и ***WM_DESTROY***, а функция ***DefWindowProc*** обрабатывает (по умолчанию) все сообщения, не обработанные оконной процедурой.

### *Сообщение WM_PAINT*

Сообщение *WM_PAINT* – передает программе, что часть или вся рабочая область окна недействительна (*invalid*) и ее следует перерисовать.

При первом создании окна недействительна вся рабочая зона и сообщение *WM_PAINT* заставляет оконную процедуру рисовать в рабочей области, что происходит также и при изменении размера окна.

Обработка сообщения *WM_PAINT* почти всегда начинается вызовом:

hdc = ***BeginPaint*** (hwnd, &ps);

и заканчивается функцией

***EndPaint*** (hwnd, &ps);

первый параметр в обеих функциях – описатель окна, второй – указатель на структуру типа *PAINTSTRUCT*, в которой содержится информация, необходимая оконной процедуре для рисования в рабочей области.

При обработке вызова *BeginPaint* обновляется фон рабочей области, заданной кистью; вся рабочая область становится действительной и возвращается ***описатель контекста устройства***, описывающий физическое устройство вывода информации (дисплей) и его драйвер.

Функция ***EndPaint*** освобождает описатель контекста устройства, после чего его значение нельзя использовать.

Для определения размера рабочей области окна вызывается функция

***GetClientRect*** (hwnd, &rect);

параметр *rect* – структура типа *RECT* (*rectangle* – прямоугольник), содержащая четыре поля типа *LONG* (*left* – левое, *top* – верх, *right* – правое, *bottom* – низ), в которые помещаются размеры рабочей области окна; поля *left, top* всегда устанавливаются в 0, поля *right, bottom* – ширина и высота рабочей области в пикселях.

Структура *rect* используется в качестве четвертого параметра функции отображения текста:

```
DrawText (hdc, "Hello, Windows !", –1, &rect,
        DT_SINGLELINE | DT_CENTER | DT_VCENTER);
```

второй параметр – отображаемый текст; третий параметр (–1) указывает, что строка текста заканчивается нулевым символом; последний параметр – набор флагов, значения которых указывают, что текст следует выводить в одну строку, по центру прямоугольной области размером, заданным в *rect*. Вызов этой функции отображает строку «*Hello, Windows*!» в центре рабочей области созданного окна.

### *Сообщение WM_DESTROY*

Пользователь вызывает это сообщение, если щелкнет кнопку закрытия окна (выберет *Close* из системного меню программы, или нажмет <*Alt*> + <*F*4>).

Программа на это сообщение вызывает функцию

***PostQuitMessage***(0);

которая ставит сообщение *WM_QUIT* в очередь. Функция *GetMessage* возвращает ненулевое значение при любом сообщении, полученном из очереди, кроме *WM_QUIT*, при котором она возвращает значение 0, что

заставляет *WinMain* прервать цикл обработки сообщений и выйти в систему, закончив программу.

### *6.5. Получение описателя контекста устройства*

В простейшем случае описатель контекста (контекст) может быть получен с помощью функций:

HDC ***GetDC***(HWND *hwnd*);

HDC ***GetWindowDC***(HWND *hwnd*);

HDC ***GetDCEx***(HWND *hwnd*, HRGN *hrgnClip*, DWORD *dwFlags*);

Функции ***GetDC*** и ***GetDCEx*** возвращают контекст клиентской области окна (без заголовка, рамки и пр.). Контекст всего окна может быть получен с помощью функций ***GetWindowDC*** или ***GetDCEx*** с соответствующим флагом:

*DCX_WINDOW* – дать контекст всего окна, включая системную область;

*DCX_PARENTCLIP* – использовать видимую часть родительского окна;

*DCX_CLIPCHILDREN* – исключить подчиненные окна;

*DCX_NORESETATTRS* – не сбрасывать состояние контекста при его освобождении и т.д.

При этом поведение функций зависит также от некоторых установок стиля оконного класса:

*CS_CLASSDC* – использовать единственный разделяемый контекст для всех окон данного класса;

*CS_OWNDC* – использовать собственный контекст для каждого экземпляра окна;

*CS_PARENTDC* – использовать контекст и регион отсечения родительского окна и т.д.

Кроме получения контекста существующего окна можно создавать новые контексты, связанные с определенным устройством.

Функции

HDC ***CreateDC***(LPCTSTR *lpszDriver*, LPCTSTR *lpszDevice*,
LPCTSTR *lpszOutput*, CONST DEVMODE* *lpInitData* );

HDC ***CreateCompatibleDC***(HDC *hPrimDC*);

создают новый контекст, связанный с указанным устройством (*CreateDC*) или совместимым с известным контекстом (*CreateCompatibleDC*). Во втором случае если «образцовый» контекст не задан, то создается *контекст в памяти* (*memory context*), совместимый с текущими установками экрана. Возвращаемое значение – описатель контекста или *NULL* – в случае ошибки. Параметры:

*lpszDriver* – может быть *DISPLAY* для контекстов, связанных с экраном, и *NULL* для всех других устройств;

*lpszDevice* – логическое имя устройства в системе;

*lpszOutput* – имя устройства в файловой системе (в *Win*32 – *NULL*);

*lpInitData* – указатель на структуру *DEVMODE* с инициирующими данными для устройства, *NULL* – настройки по умолчанию;

*hPrimDC* – «образцовый» контекст, с которым будет совместим вновь создаваемый, если *NULL* – экран с текущими настройками.

По окончании работы с контекстом он должен быть освобожден (закрыт). Для контекстов, полученных с помощью функций *Get...*, используется функция

int ***ReleaseDC***(HWND hwnd, HDC hdc);

которая освобождает «общие» и оконные контексты.

Для контекстов, созданных с помощью функций *Create...*, используется функция

int ***DeleteDC***(HDC hdc);

возвращает 1 – при успешном завершении, 0 – при ошибке.

Многие характеристики действующего контекста могут быть получены с помощью функции ***GetDeviceCaps***(), принимающей в качестве аргумента описатель контекста и индекс (номер) интересующего параметра (описываются системными константами) и возвращающей значение этого параметра.

### *6.6. Основные инструменты графической подсистемы*

В среде *Windows* за формирование изображения отвечают так называемые *инструменты*, а функции рисования лишь задают их поведение. Инструменты являются системными объектами, но прикладная программа может произвольно создавать их и управлять ими.

К основным инструментам относятся:

– ***перо*** (*Pen*) – отображение контурных примитивов («карандаш»);

– ***кисть*** (*Brush*) – заполнение внутренних областей примитивов;

– ***шрифт*** (*Font*) – отображение символов и строк;

– ***битовая карта*** (*Bitmap*) – «готовые» растровые изображения.

Инструмент идентифицируется его описателем и создается соответствующей функцией вида ***Create...***, которая возвращает этот описатель (*NULL* – признак ошибки). Количество создаваемых инструментов искусственно не ограничивается, но в любом контексте одновременно может быть активным только один инструмент каждого типа. Выбор инструмента отменить нельзя, но можно повторять его сколько угодно раз с любыми другими инструментами того же типа. Инструменты различного типа между собой взаимно независимы. Перед удалением инструмент следует дезактивировать, выбрав активным другой инструмент того же типа, например, сохраненный предыдущий.

***Инструмент Pen***

Для отображения контурных примитивов используется перо, выбранное в контексте, определяющее цвет, ширину и стиль линии, который может быть сплошным (*solid*), точечным (*dotted*) или пунктирным (*dashed*).

По умолчанию устанавливается одно из трех ***стандартных*** перьев, рисующих сплошные линии толщиной в единицу, выбранного цвета: *BLACK_PEN* – черное перо, *WHITE_PEN* – белое перо и *NULL_PEN* – пустое перо, которое ничего не рисует.

Определив переменную, например, *hPen* типа *HPEN* (описатель пера – *handle to a pen*): ***HPEN hPen***; получить описатель одного из стандартных перьев, например белого пера, можно, вызывая функцию

hPen = ***GetStockObject*** (WHITE_PEN);

сделаем это перо текущим, вызвав функцию

***SelectObject*** (hdc, hPen);

После этого все линии будут использовать белое перо до тех пор, пока не выберем другое перо в контекст устройства или пока не освободим контекст устройства.

Все вышесказанное можно совместить в одной инструкции:

hPen = ***SelectObject*** (hdc, ***GetStockObject*** (WHITE_PEN));

Если это первый вызов, функция *SelectObject* возвращает описатель того пера, которое уже было выбрано в контексте устройства; текущим пером становится белое, а переменная *hPen* получает описатель предыдущего (по умолчанию – черного) пера, вернуться к которому можно, используя вызов

SelectObject (hdc, hPen);

Для ***создания пера*** используются функции *CreatePen* или *CreatePenIndirect* (*ExtCreatePen*).

Функция *CreatePen*:

hPen = ***CreatePen*** (iPenStyle, iWidth, rgbColor);

параметр *iPenStyle* определяет стиль линии и может принимать одно из семи значений (семь стилей пера): *PS_SOLID* – сплошное, *PS_DASH*, *PS_DOT*, *PS_DASHDOT*, *PS_DASHDOTDOT* – штриховое, пунктирное и штрихпунктирное, *PS_NULL* – нерисующее (пустое) перо, *PS_INSIDEFRAME* – внутренняя обводка, в замкнутом контуре автоматически отступает внутрь в соответствии с толщиной линии.

Для стилей *PS_SOLID*, *PS_NULL* и *PS_INSIDEFRAME* параметр *iWidth* задает ширину пера, при *iWidth* = 0 – перо шириной в один пиксел.

Параметр *rgbColor* – цвет пера; для перьев всех стилей, кроме *PS_INSIDEFRAME*, преобразуется в ближайший чистый цвет, стиль *PS_INSIDEFRAME* позволяет использовать полутона при ширине больше 1.

Можно создать перо, определив структуру ***LOGPEN logpen*** («логическое перо» – *logical pen*), содержащую поля:

*lopnStyle* (*UINT*) – стиль пера;
*lopnWidth* (*POINT*) – ширина пера в логических единицах измерения;
*lopnColor* (*COLORREF*) – цвет пера.

Затем создаем перо, передавая адрес структуры в функцию *CreatePenIndirect*:

```
hPen = CreatePenIndirect (&logpen);
```

Получить информацию об уже существующем пере можно, используя функцию

```
GetObject (hPen, sizeof(LOGPEN), &logpen);
```

Функции *CreatePen* и *CreatePenIndirect* не требуют описателя контекста устройства. Эти функции создают логические перья, которые никак не связаны с контекстом устройства до тех пор, пока не вызвать функцию *SelectObject*.

Рассмотрим метод создания, выбора и удаления перьев. Предположим, в программе используются два пера – красное шириной 3 и черное точечное. Определим переменные для хранения описателей этих перьев:

```
static HPEN hPen1, hPen2;
```

В процессе обработки сообщения ***WM_CREATE*** создадим перья:

```
hPen1 = CreatePen (PS_SOLID, 3, RGB (255, 0, 0));
hPen2 = CreatePen (PS_DOT,  0, 0);
```

В процессе обработки сообщения ***WM_PAINT*** или когда есть действительный контекст устройства, выберем одно из этих перьев в контекст устройства и можем рисовать, используя его:

```
SelectObject (hdc, hPen2);
    [функции рисования линий]
SelectObject (hdc, hPen1);
    [другие функции рисования линий]
```

В процессе обработки сообщения ***WM_DESTROY*** удалить их:

```
DeleteObject (hPen1);
DeleteObject (hPen2);
```

Можно создать перья в процессе обработки сообщения *WM_PAINT* и удалить их после вызова *EndPaint*.

Можно объединить вызовы функций *CreatePen* и *SelectObject* в одну инструкцию:

```
SelectObject (hdc, CreatePen (PS_DASH, 0, RGB (255, 0, 0)));
```

Удалить перо можно, выбрав стандартное перо *BLACK_PEN* в контекст устройства и удаления значения, возвращаемого функцией *SelectObject*:

```
DeleteObject (SelectObject (hdc, GetStockObject (BLACK_PEN)));
```

Выбирая только что созданное перо в контекст устройства, сохраним описатель, возвращаемый функцией *SelectObject*:

hPen = SelectObject (hdc, CreatePen (PS_DASH, 0, RGB (255, 0, 0)));

Если это первый вызов, *hPen* – описатель стандартного черного пера, мы можем теперь выбрать его в контекст устройства и удалить созданное перо (описатель, возвращаемый вызовом функции *SelectObject*) в одной инструкции:

DeleteObject (SelectObject (hdc, hPen));

***Инструмент Brush***

Объект кисть – это битовый образ, свойства которого распространяются в горизонтальном и вертикальном направлениях при закрашивании области.

Имеется шесть ***стандартных*** (*Stock*) кистей: *WHITE_BRUSH* – белая, *LTGRAY_ BRUSH* – светло-серая, *GRAY_ BRUSH* – серая, *DKGRAY_ BRUSH* – темно-серая, *BLACK_ BRUSH* – черная и *NULL_ BRUSH* (*HOLLOW*) – пустая кисть.

Выбрать одну из стандартных кистей в контекст устройства можно аналогично выбору пера, определив переменную типа описателя кисти ***HBRUSH hBrush***; получаем описатель кисти, например, *GRAY_BRUSH*:

hBrush = GetStockObject (GRAY_BRUSH);

Выберем эту кисть в контекст устройства:

SelectObject (hdc, hBrush);

Теперь внутренняя область рисуемых фигур будет закрашиваться серым.

Если необходимо нарисовать фигуру без рамки, выберем перо *NULL_PEN*:

SelectObject (hdc, GetStockObject (NULL_PEN));

А если только контур фигуры без закрашивания внутренней области, выберем кисть *NULL_BRUSH*:

SelectObject (hdc, GetStockObject (NULL_BRUSH));

Для создания сплошной (*Solid*) логической кисти:

hBrush = ***CreateSolidBrush*** (rgbColor);

Для создания штриховой (*Hatch*) кисти, состоящей из горизонтальных, вертикальных или диагональных линий:

hBrush = ***CreateHatchBrush*** (iHatchStyle, rgbColor);

параметр *iHatchStyle* – стиль штриховки: *HS_HORIZONTAL*, *HS_VERTICAL*, *HS_BDIAGONAL* – диагональная слева направо вверх; *HS_FDIAGONAL* – диагональная слева направо вниз; *HS_CROSS* – прямая сетка; *HS_DIAG-*

*CROSS* – диагональная сетка; в обеих функциях *rgbColor* – цвет штриховых линий.

Промежутки между штриховыми линиями закрашиваются в соответствии с режимом и цветом фона, если режим фона – *OPAQUE*, то цвет фона используется для закрашивания промежутков между штриховыми линиями, если режим фона – *TRANSPARENT*, то промежутки между штриховыми линиями не зарисовываются.

Можно ***создавать кисти***, основанные на битовых шаблонах, используя функцию *CreatePatternBrush*:

hBrush = ***CreatePatternBrush*** (hBitmap);

Функция, включающая три рассмотренные ранее функции, строящие кисти:

hBrush = ***CreateBrushIndirect*** (&logbrush);

переменная *logbrush* – структура типа *LOGBRUSH* («логическая кисть» – *logical brush*), содержащая поля:

UINT *lbStyle* – стиль кисти: *BS_SOLID* – сплошная; *BS_HOLLOW*, *BS_NULL* – «пустая» (невидимая); *BS_HATCHED* – штрихованная; *BS_PATTERN*, *BS_PATTERN8X8* – задаются битовой картой; *BS_DIB-PATTERN*, *BS_DIBPATTERN8X8*, *BS_DIBPATTERNPT* – задаются битовой картой *DIB* (в *Windows* 95 размер шаблона ограничен 8×8 точек);

COLORREF *lbColor* – цвет кисти, для пустой или «шаблонной» кисти игнорируется, для кистей с *DIB*-шаблоном младшее слово определяет, следует ли интерпретировать его цвета как заданные цветовыми компонентами (*DIB_RGB_COLORS*) или как палитровые (*DIB_PAL_COLORS*).

Значение поля *lbStyle* определяет, как интерпретируются другие поля:

| *lbStyle* | *lbColor* | *lbHatch* |
|---|---|---|
| *BS_SOLID* | Цвет кисти | Игнорируется |
| *BS_HOLLOW* | Игнорируется | Игнорируется |
| *BS_HATCHED* | Цвет штриховых линий | Стиль штриховки |
| *BS_PATTERN* | Игнорируется | Описатель битового шаблона |

Получить описатель логической кисти: ***SelectObject*** (hdc, hBrush);
Удалить созданную кисть: ***DeleteObject*** (*hBrush*);
Получить информацию о кисти:

***GetObject*** (hBrush, sizeof (LOGBRUSH), &logbrush);

### *Инструмент Font*

Более сложным инструментом является шрифт. Все символы в оконном интерфейсе формируются в соответствии с одним из зарегистрированных в системе шрифтов.

Физический шрифт – файл (образ в памяти) с описанием начертаний всех известных в данном шрифте символов. Логический шрифт – объект

*GDI*, характеризуемый как физическим шрифтом, так и его конкретными характеристиками. Он же является и инструментом, отвечающим за формирование символов.

Для ***создания*** логического шрифта используется функция

HFONT ***CreateFont*** (int nHeight, int nWidth, int nEscapement,
int nOrientation, int fnWeight, DWORD fdwItalic, DWORD fdwUnderline,
DWORD fdwStrikeOut, DWORD fdwCharSet,
DWORD fdwOutputPrecision, DWORD fdwClipPrecision,
DWORD fdwQuality, DWORD fdwPitchAndFamily,
LPCTSTR lpszFace);

возвращающая описатель созданного инструмента (*NULL* – ошибка), параметры:

*nHeight* – основной размер (высота) шрифта в логических единицах: положительное значение определяет высоту знакоместа, отрицательное – высоту шрифта (после смены знака), нулевое – размер по умолчанию; при неточном соответствии требуемого размера выбирается наибольший, не превышающий требуемый;

*nWidth* – «приблизительная» ширина шрифта, если 0 – стандартная для выбранного основного размера;

*nEscapement* – направление вывода строки символов (угол между базовой линией и горизонтальной осью по часовой стрелке в десятых долях градуса);

*nOrientation* – ориентация отдельного символа (исчисление аналогично предыдущему параметру, в *Windows* 9x ориентация символов и направление вывода должны совпадать);

*fnWeight* – толщина символов в условных единицах от 0 до 1000, например: 0 (*FW_DONTCARE*) – стандартный (по умолчанию), 400 (*FW_NORMAL*, *FW_REGULAR*) – стандартная толщина, обычный шрифт, 700 (*FW_BOLD*) – жирный, выделенный, и т.д.;

*fdwItalic*, *fdwUnderline*, *fdwStrikeOut* – флаги, указывающие, является ли шрифт наклонным, подчеркнутым или перечеркнутым;

*fdwCharSet* – тип символьного набора: *DEFAULT_CHARSET*, *ANSI_CHARSET*, *OEM_CHARSET* и т.д. (национальные наборы символов);

*fdwOutputPrecision* – «точность вывода» символов, фактически предпочтение типа шрифта при наличии альтернативного выбора: *OUT_DEFAULT_PRECIS*, *OUT_DEVICE_PRECIS* и т.д.;

*fdwClipPrecision* – точность отсечения;

*fdwQuality* – качество вывода;

*fdwPitchAndFamily* – комбинация по «ИЛИ» двух групп параметров, действующих при неопределенном имени шрифта:

– питч (шаг символов) – 2 младших бита: *DEFAULT_PITCH*, *FIXED_PITCH*, *VARIABLE_PITCH*;

– «семейство» шрифта – 4 старших бита: *FF_DONTCARE* – по умолчанию, *FF_MODERN* – моноширинные шрифты, *FF_ROMAN* –

«книжные» с засечками (переменный шаг), *FF_SWISS* – без засечек (переменный шаг), *FF_SCRIPT* – «рукописные» и курсивные;

*lpszFace* – имя шрифта, обычно совпадает с именем его дискового файла; если *NULL* – система подбирает шрифт, наиболее отвечающий заданным требованиям, иначе явно указанный шрифт перекрывает их.

Функция ***CreateFontIndirect*** (const LOGFONT* lplgFont); использует в качестве аргумента структуру с полями аналогичного назначения.

Параметры шрифта не включают цвет отображающего инструмента. Управление цветом выводимого текста осуществляется функциями

COLORREF ***SetTextColor*** (hdc, crColor);

COLORREF ***GetTextColor*** (hdc);

Базовой функцией вывода символа является

BOOL ***TextOut*** (hdc, nXStart, nYStart, lpString, cbString);

Позицией символа считается верхний левый угол его знакоместа.

### *6.7. Закрашивание пустот*

Если выбранные перо или кисть не сплошные, то они не воздействуют на фоновые промежутки, например, между штрихами, однако эти промежутки заполняются фоновым цветом, для работы с которым служат функции

COLORREF ***SetBkColor***(hdc, crColor);

COLORREF ***GetBkColor***(hdc);

первая из которых устанавливает новый, а вторая – получает текущий фоновые цвета; признак ошибки – значение *CLR_INVALID*. Функция *SetBkColor* возвращает значение предыдущего цвета.

Функции

int ***SetBkMode*** (hdc, iBkMode);

int ***GetBkMode*** (hdc);

соответственно устанавливают новый или определяют текущий режим фона, 0 – признак ошибки; параметр int *iBkMode* – режим фона, по умолчанию белый (*OPAQUE*) – сначала рисуется фон, затем передний план; режим *TRANSPARENT* – отменяет заполнение пустот, в этом случае цвет фона игнорируется и пустоты заполняться не будут.

### *6.8. Рисование линий и кривых*

Теоретически все, что необходимо драйверу устройства для рисования, это функции *SetPixel* и *GetPixel*.

Значительно более эффективным является использование стандартных функций рисования отрезков и других сложных графических операций.

На представление линий, созданных с использованием функций, влияют атрибуты контекста устройства: текущая позиция, перо, режим фона (для несплошных перьев), цвет фона (для режима фона *OPAQUE*) и режим рисования.

Функция

***LineTo*** (hdc, xEnd, yEnd);

рисует отрезок прямой из текущего положения пера, определенного в контексте устройства, до точки (*xEnd*, *yEnd*), которая не включается в отрезок. В контексте текущее положение пера по умолчанию устанавливается в точку (0,0) и функция *LineTo* без предварительной установки текущей позиции нарисует отрезок, начинающийся в левом верхнем углу рабочей области окна.

Для рисования отрезка из точки (*xStart*, *yStart*) в точку (*xEnd*, *yEnd*) необходимо сначала для установки (изменения) текущего положения пера использовать функцию

***MoveToEx*** (hdc, xStart, yStart, &pt);

*pt* – структура типа *POINT*, определяющая предыдущую позицию. После чего, используя *LineTo* (*hdc*, *xEnd*, *yEnd*); будет нарисован отрезок до точки (*xEnd*, *yEnd*), не включая ее в отрезок, и текущее положение пера установится в точку (*xEnd*, *yEnd*).

Узнать текущее положение пера можно с помощью функции

***GetCurrentPositionEx*** (hdc, &pt);

Следующий фрагмент программы отображает в рабочей области окна сетку с интервалом в 100 пикселей, начиная от левого верхнего угла:

```
GetClientRect (hwnd, &rect);   (см. Пример 1, Сообщение WM_PAINT);
for (x = 0; x < rect.right; x += 100)    {
        MoveToEx (hdc, x, 0, NULL);
        LineTo (hdc, x, rect.bottom);
}
for (y = 0; y < rect.bottom; y += 100) {
        MoveToEx (hdc, 0, y, NULL);
        LineTo (hdc, rect.right, y);
}
```

Когда необходимо соединить отрезками массив точек *pt* размером *cPoint*, можно использовать функцию

***Polyline*** (hdc, pt, cPoint);

Например, определим массив из 5 точек (10 значений), описывающих контур прямоугольника:

```
POINT pt [5] = { 100, 100, 200, 100, 200, 200, 100, 200, 100, 100 };
```

используем функцию *Polyline* (hdc, pt, 5). Следует обратить внимание, что первая и последняя точки совпадают.

Функция *Polyline* не учитывает и не изменяет текущее положение пера. Функция ***PolylineTo*** использует текущее положение для начальной точки и устанавливает текущее положение в конец последнего нарисованного отрезка. Предыдущий пример будет выглядеть

```
MoveToEx (hdc, pt[0].x, pt[0].y, NULL);
PolylineTo (hdc, pt + 1, 4);
```

Для рисования дуги эллипса (рис. П 6.2) используется функция

***Arc*** (hdc, x1, y1, x2, y2, xStart, yStart, xEnd, yEnd);

в которой значения (*x*1, *y*1) задают левый верхний угол, (*x*2, *y*2) – правый нижний; (*xStart*, *yStart*) – начало дуги; (*xEnd*, *yEnd*) – конец дуги.

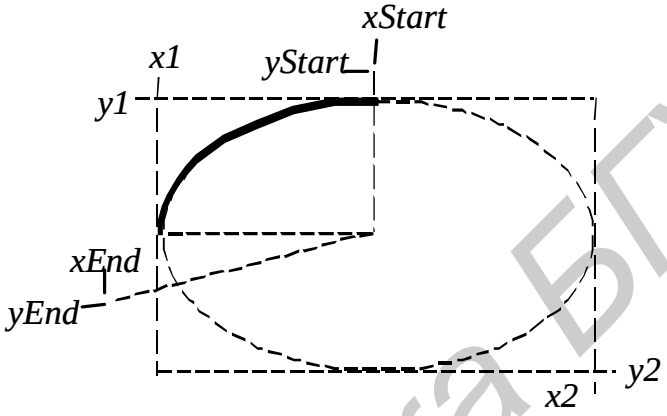



Рис. П 6.2. Фигура, нарисованная с использованием функции *Arc*

Чтобы нарисовать одну или более связанных кривых Безье, используются функции

***PolyBezier*** (hdc, pt, iCount);
***PolyBezierTo*** (hdc, pt, iCount);

*pt* – массив структур типа *POINT*. В функции *PolyBezier* первые четыре точки идут в таком порядке: начальная точка, первая контрольная точка, вторая контрольная точка, конечная точка кривой Безье. Каждая следующая кривая Безье требует три новых точки, поскольку начальная точка следующей кривой есть конечная точка предыдущей и т.д. Параметр *iCount* = 1+3**n* – равен единице плюс три, умноженное на число отображаемых кривых.

### *6.9. Пример изображения графика функции sin*

Программа содержит массив из 1000 структур *POINT*. В цикле от 0 до 999 член x структуры растет от 0 до *cxClient*. В каждом цикле член структуры определяет значение синуса и масштабируется до размеров клиентской области окна. Вся кривая целиком отображается с использованием одного вызова функции *Polyline* (рис. П 6.3).

Текст программы может быть следующим:

```
#include <windows.h>
#include <math.h>
```

```
#define NUM  1000
#define TWOPI  (2 * 3.14159)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)

{
     static char szAppName[] = "Sin" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;
     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground=(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Second Example",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
     while (GetMessage (&msg, NULL, 0, 0))  {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}
     LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg,
     WPARAM wParam, LPARAM lParam)
     {
         static int  cxClient, cyClient ;
         HDC         hdc ;
         int         i ;
```

```
        PAINTSTRUCT ps ;
        POINT       pt [NUM] ;
        switch (iMsg)  {
            case WM_SIZE:
                cxClient = LOWORD (lParam) ;
                cyClient = HIWORD (lParam) ;

                return 0 ;
            case WM_PAINT:
                hdc = BeginPaint (hwnd, &ps) ;
                MoveToEx (hdc, 0,        cyClient / 2, NULL) ;
                LineTo   (hdc, cxClient, cyClient / 2) ;
                for (i = 0 ; i < NUM ; i++) {
                    pt[i].x = i * cxClient / NUM ;
                    pt[i].y = (int) (cyClient / 2 * (1 - sin (TWOPI * i / NUM))) ;
                }
                Polyline (hdc, pt, NUM) ;
            return 0 ;
            case WM_DESTROY:
                PostQuitMessage (0) ;
                return 0 ;
            }
        return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Результат работы программы:

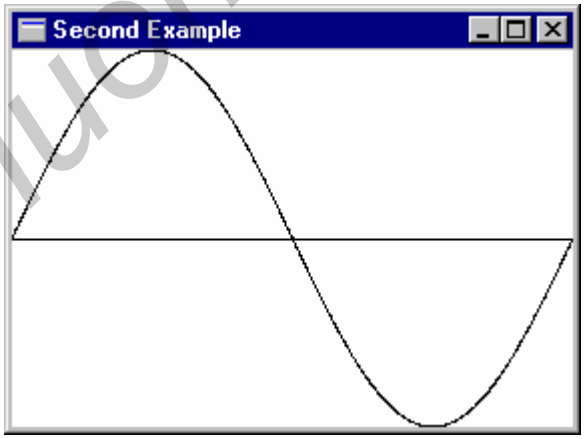



Рис. П 6.3

### *6.10. Рисование замкнутых фигур*

Рассмотрим функции для рисования замкнутых фигур:

*Rectangle* – прямоугольник; *Ellipse* – эллипс; *RoundRect* – прямоугольник со скругленными углами; *Chord* – дуга кривой эллипса,

концы которой соединены хордой; *Pie* – кусок, вырезанный из эллипса; *Polygon* – многоугольник; *PolyPolygon* – множество многоугольников.

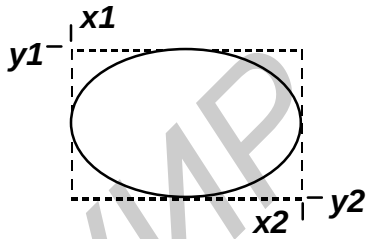
Контур фигуры рисуется текущим пером, а фигура закрашивается текущей кистью. По умолчанию это стандартная кисть *WHITE_BRUSH*.

Простейшей является функция рисования прямоугольника:

***Rectangle*** (hdc, x1, y1, x2, y2);

(*x*1, *y*1) – координаты левого верхнего угла, (*x*2, *y*2) – правого нижнего угла.

Для рисования эллипса используется функция, имеющая те же параметры:



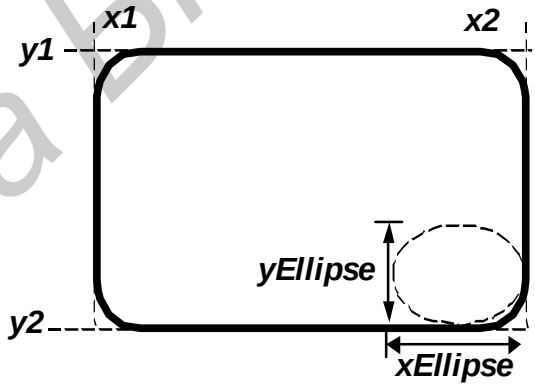

***Ellipse*** (hdc, x1, y1, x2, y2);

Фигура, отображаемая функцией *Ellipse* (вместе с ограничивающим прямоугольником).

Функция для рисования прямоугольника со скругленными углами:

***RoundRect*** (hdc, x1, y1, x2, y2, xEllipse, yEllipse);

имеет два дополнительных параметра: для рисования скругленных углов используется маленький эллипс, шириной *xEllipse*, высотой *yEllipse*. Фигура, отображаемая этой функцией, приведена на рисунке. Скругленные углы были нарисованы с использованием размеров эллипса, вычисленных по формулам




xEllipse = (x2–x1)/4; yEllipse = (y2–y1)/4;

Это простое приближение, но результаты скорее всего будут выглядеть не совсем правильно, потому что округлость углов более заметна при больших размерах прямоугольника.

Функции *Chord* (сегмент эллипса) и *Pie* (сектор эллипса) имеют одинаковые параметры:

***Chord*** (hdc, x1, y1, x2, y2, xStart, yStart, xEnd, yEnd);

***Pie*** (hdc, x1, y1, x2, y2, xStart, yStart, xEnd, yEnd);

При рисовании используется воображаемая линия для соединения точки (*xStart*, *yStart* – начало дуги) с центром эллипса. В точке, где эта линия пересекается с ограничивающим прямоугольником, начинается рисование дуги эллипса в направлении против часовой стрелки; аналогично используется воображаемая линия для соединения точки (*xEnd*, *yEnd* – конец дуги) с центром эллипса. В точке, где эта линия пересекается с ограничивающим прямоугольником, завершается рисование дуги.

В функции *Chord* соединяются конечные точки дуги, а в функции *Pie* соединяются начальная и конечная точки дуги с центром эллипса.

Фигуры, отображаемые функциями *Chord* и *Pie*, приведены на рис. П 6.3.

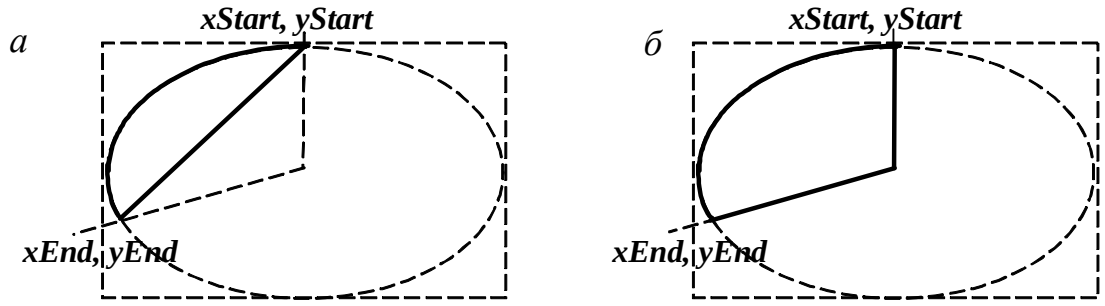



Рис. П 6.3. Фигуры, нарисованные с использованием:
*а* – функции *Chord*; *б* – функции *Pie*

### *6.11. Функция Polygon и режим закрашивания многоугольника*

Функция рисования многоугольника:

***Polygon*** (hdc, pt, iCount);

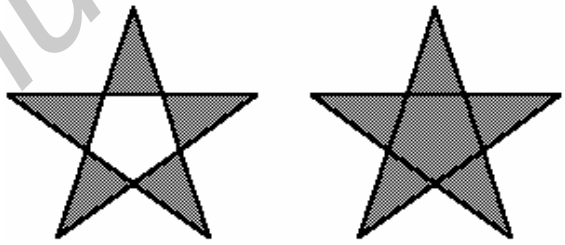
*pt* – это массив структур типа *POINT*, *iCount* – число точек; если последняя точка в массиве не совпадает с первой, то добавляется линия, их соединяющая.

Внутренняя область фигуры закрашивается текущей кистью. По умолчанию режим закрашивания равен *ALTERNATE* (попеременный), т.е. закрашиваются только те фрагменты внутренней области многоугольника, которые получаются путем соединения линий с нечетными номерами (1, 3, 5...), другие фрагменты внутренней области не закрашиваются.

Режим закрашивания устанавливается функцией:

***SetPolyFillMode*** (hdc, iMode);

*iMode* – режим закрашивания, *WINDING* (сквозной) – закрашиваются все внутренние области. Для иллюстрации приведем пример:



звезда, находящаяся слева, нарисована в режиме *ALTERNATE*, а звезда, находящаяся справа – в режиме *WINDING*.

### *6.12. Пример отображения линий*

Если в предыдущей программе (*Second Example*) заменить *case WM_PAINT...* и убрать лишние переменные, получим программу (*Third Example*), в которой рисуются прямоугольник, эллипс, прямоугольник со скругленными углами и два отрезка. Программа показывает, что функции,

определяющие области, закрашивают их, поэтому отрезки не видны там, где нарисован эллипс. Результат работы программы приведен на рис. П 6.4.

```
        . . .
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;
    Rectangle (hdc, cxClient /8, cyClient /8, 7*cxClient /8, 7*cyClient /8) ;
    MoveToEx (hdc, 0, 0, NULL) ;
    LineTo (hdc, cxClient, cyClient) ;
    MoveToEx (hdc, 0, cyClient, NULL) ;
    LineTo (hdc, cxClient, 0) ;
    Ellipse (hdc, cxClient /8, cyClient /8, 7*cxClient /8, 7*cyClient /8) ;
    RoundRect (hdc, cxClient /4, cyClient /4, 3*cxClient /4, 3*cyClient /4,
                    cxClient /4, cyClient /4) ;
    EndPaint (hwnd, &ps) ;
    return 0 ;          . . .
```

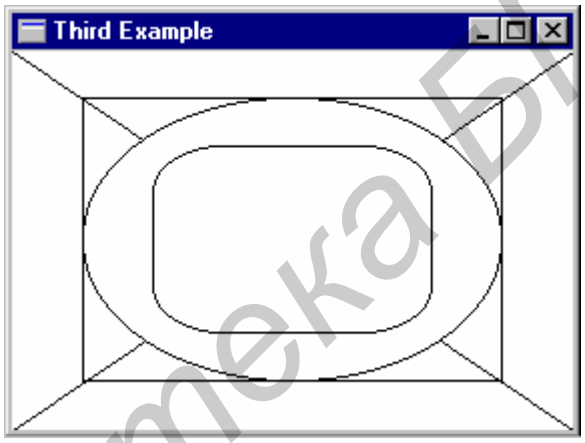



Рис. П 6.4. Результат программы *Third Example*

### *6.13. Управление областями вывода и отсечением*

Стандартно графический вывод отсекается по границам окна, с которым связан контекст, кроме того, из области вывода удаляются части, перекрытые другими окнами. Таким образом, границы области вывода могут иметь достаточно сложную форму.

В среде *Windows* используются функции, работающие с прямоугольными областями, использующими структуры типа *RECT* (прямоугольник) и произвольными областями – регионами (*regions*).

#### *Работа с прямоугольниками*

Простейшим средством, задающим границу области вывода, является прямоугольник. Функции отображения прямоугольных областей используют указатель на структуру *rect* типа *RECT* (прямоугольник), имеющую поля: *left*, *top*, *right*, *bottom*, заданные в логических единицах.

Функция ***FillRect*** (hdc, &rect, hBrush); закрашивает прямоугольник (не включая правую и нижнюю координаты) заданной кистью.

Функция ***FrameRect*** (hdc, &rect, hBrush); использует кисть для рисования прямоугольной рамки, но не закрашивает внутреннюю область.

Функция ***InvertRect*** (hdc, &rect); инвертирует все пиксели в прямоугольнике, устанавливая единичные биты в ноль, а нулевые – в единицу, т.е. переводит белую область в черную, черную – в белую (зеленую – в фиолетовую).

*Windows* содержит функции, позволяющие легко манипулировать со структурами типа *RECT*:

– установка всех полей структуры *RECT* в заданные значения:
***SetRect*** (&rect, xLeft, yTop, xRight, yBottom);

– перемещение на заданное число координат вдоль осей *x* и *y*:
***OffsetRect*** (&rect, x, y);

– увеличение или уменьшение размеров прямоугольника:
***InflateRect*** (&rect, x, y);

– установка полей структуры прямоугольника в ноль:
***SetRectEmpty*** (&rect);

– копирование одного прямоугольника в другой:
***CopyRect*** (&DestRect, &SrcRect);

– пересечение двух прямоугольников:
***IntersectRect*** (&DestRect, &SrcRect1, &SrcRect2);

– объединение двух прямоугольников:
***UnionRect*** (&DestRect, &SrcRect1, &SrcRect2);

– определение, является ли прямоугольник пустым:
bEmpty = ***IsRectEmpty*** (&rect);

– определение, содержится ли точка внутри прямоугольника:
bInRect = ***PtInRect*** (&rect, point);

***Создание и рисование регионов***

Унифицированным средством, задающим границу области вывода, является ***регион***, который может иметь прямоугольную, многоугольную, эллиптическую формы или их сочетание. Регион является объектом, идентифицируемым его описателем *HRGN*.

Произвольный регион создается универсальной функцией:

***ExtCreateRegion*** (const XFORM *lpXform, DWORD nDataSize,
const RGNDATA lpRgnData);

структура *XFORM* описывает преобразование региона в экранные координаты, при его равенстве *NULL* координаты считаются идентичными; регион описывается структурой *RGNDATA*, содержащей поля:

RGNDATAHEADER *rdh* – структура данных заголовка;

char *Buffer*[1] – буфер структур *RECT*, образующих регион.

В свою очередь, структура *RGNDATAHEADER* содержит поля:

DWORD *dwSize* – размер заголовка в байтах;

DWORD *iType* – тип региона (*RGN_RECTANGLES*);

DWORD *nCount* – количество прямоугольников в буфере;

DWORD *nRgnSize* – требуемый размер буфера (может быть нулевым);

RECT *rcBound* – вмещающий прямоугольник для региона.

Простейший тип региона – прямоугольник может быть создан с помощью функций

hRgn = ***CreateRectRgn*** (xLeft, yTop, xRight, yBottom);

hRgn = ***CreateRectRgnIndirect*** (&rect);

Для создания эллиптических регионов:

hRgn = ***CreateEllipticRgn*** (xLeft, yTop, xRight, yBottom);

hRgn = ***CreateEllipticRgnIndirect*** (&rect);

Функция ***CreateRoundRectRgn*** строит прямоугольный регион со скругленными углами.

Создание многоугольного региона (похоже на функцию *Polygon*):

hRgn = ***CreatePolygonRgn*** (&point, iCount, iPolyFillMode);

параметр *point* – массив структур типа *POINT*, ***iCount*** – число точек, *iPolyFillMode* – *ALTERNATE* (*WINDING*). Регион из множества многоугольников – ***CreatePolyPolygonRgn***.

Два региона могут быть объединены в один функцией:

iRgnType = ***CombineRgn*** (hDestRgn, hSrcRgn1, hSrcRgn2, iCombine);

которая комбинирует два исходных региона (*hSrcRgn1* и *hSrcRgn2*) и строит третий, на который ссылается *hDestRgn*. Все три описателя регионов еще до вызова функции должны быть действительными, однако дополнительный регион, описываемый ранее *hDestRgn*, уничтожается.

Параметр *iCombine* задает правило объединения регионов:

RGN_AND – область пересечения двух исходных регионов;

RGN_OR – объединение двух исходных регионов;

RGN_XOR – объединение двух исходных регионов за исключением области пересечения;

RGN_DIFF – часть региона *hSrcRgn1*, не входящая в регион *hSrcRg2*;

RGN_COPY– регион hSrcRgn1.

Значение *iRgnType*, возвращаемое функцией, означает: *NULLREGION* – регион пуст; *SIMPLEREGION* – регион представляет собой простой прямоугольник, эллипс или многоугольник; *COMPLEXREGION* – комбинация прямоугольников, эллипсов или многоугольников; *ERROR* – произошла ошибка.

Полученный описатель региона можно использовать в функциях

***FillRgn*** (hdc,hRgn,hBrush); ***FrameRgn*** (hdc,hRgn,hBrush,xFrame,yFrame);

***InvertRgn*** (hdc, hRgn); ***PaintRgn*** (hdc, hRgn);

аналогичных функциям *FillRect* (закрашивание прямоугольной области), *FrameRect* (закрашивание границы прямоугольной области) и *InvertRect*

(инвертирование пикселей в прямоугольнике); параметры *xFrame* и *yFrame* – ширина и высота рамки, которая будет нарисована вокруг региона.

Функция ***PaintRgn*** закрашивает внутреннюю область региона текущей выбранной в контекст устройства кистью. Во всех функциях предполагается, что регион определен в логических координатах.

Для удаления региона используется функция удаления объектов *GDI*:

***DeleteObject*** (hRgn);

Функция ***GetRgnData*** позволяет получить доступ к данным региона по его описателю. Вмещающий прямоугольник для региона может быть получен функцией ***GetRgnBox***.

Функции ***PtInRegion*** и ***RectInRegion*** проверяют соответственно точку и прямоугольник на вхождение их в регион.

Регион может быть обведен или закрашен с использованием текущих инструментов данного контекста.

***Прямоугольники и регионы отсечения***

Прямоугольники и регионы могут принимать участие в отсечении. Функция *InvalidateRect* делает недействительным прямоугольную область дисплея и генерирует сообщение *WM_PAINT*. Ее можно использовать, например, для обновления рабочей области:

***InvalidateRect*** (hwnd, NULL, TRUE);

Получить координаты недействительного прямоугольника можно с помощью функции ***GetUpdateRect***, а сделать действительным прямоугольник в рабочей области – ***ValidateRec***t.

Получая сообщение *WM_PAINT*, координаты недействительного прямоугольника доступны из полей структуры *PAINTSTRUCT*, заполняемой при вызове функции *BeginPaint*. Этот недействительный прямоугольник также определяет регион отсечения, за пределами которого нельзя рисовать.

Для создания региона отсечения (выбрав регион в контекст устройства) используются функции

***SelectObject*** (hdc, hRgn); ***SelectClipRgn*** (hdc, hRgn);

регион отсечения задается в координатах устройства.

Среда *Windows* содержит несколько функций для манипуляции с регионом отсечения, таких как ***ExcludeClipRect*** – исключение прямоугольника из региона отсечения; ***IntersectClipRect*** – создание нового региона отсечения, который представляет собой пересечение предыдущего региона отсечения и прямоугольника; ***OffsetClipRgn*** – перемещение региона отсечения в другую часть рабочей области.

## *6.14. Растровая графика*

Все рассмотренные выше функции базировались на вычерчивании графических примитивов определенными инструментами по заданным

командам, т.е. по векторному принципу. Растровая графика предусматривает доступ к изображению на уровне образующих его точек.

Для большинства устройств отображения первичен растровый принцип формирования изображения. Некоторые контексты поддерживают не все функции растровой графики. Информацию о совместимости может предоставить функция ***GetDeviceCaps***().

Простейшим и наиболее универсальным способом получения произвольных изображений является доступ к отдельным его точкам. Функции

COLORREF ***SetPixel*** (hdc, nX, nY, crColor);
BOOL ***SetPixelV*** (hdc, nX, nY, crColor);
COLORREF ***GetPixel*** (hdc, nX, nY);

выполняют соответственно изменение состояния (цвета) одной логической точки и получение текущего состояния. Функция ***SetPixelV***() приводит значение цвета к ближайшему представимому в данном контексте цвету; возвращаемое значение – состояние точки на момент вызова функции (*COLORREF*), либо признак успешности выполнения (*BOOL*). Параметры

*nX*, *nY* – логические координаты точки (*int*);

*crColor* – новое значение цвета точки (*COLORREF*).

Более сложные и эффективные функции манипулируют не отдельными точками, а массивами точек – фрагментами изображений и битовыми образами.

***Битовый образ*** (*bitmap*) – двухмерный массив числовых значений, характеризующий состояние точек некоторой области, обычно прямоугольной.

В простейшем случае битовый образ описывается структурой *BITMAP*, содержащей поля:

LONG *bmType* – тип образа, должен быть равен 0;

LONG *bmWidth*, LONG *bmHeight* – положительные значения ширины и высоты прямоугольной области в пикселах;

LONG *bmWidthBytes* – размер в байтах образа одной строки изображения, в среде *Windows* должен быть кратен 2, т.к. система предполагает, что массив состоит из слов;

WORD *bmPlanes* – количество цветовых планов (плоскостей), т.е. компонент, задающих цвет;

WORD *bmBitsPixel* – количество бит для кодирования цвета точки;

LPVOID *bmBits* – указатель на двухмерный массив данных, каждая строка которого соответствует одной строке изображения.

Используются монохромный и цветной типы образов. В случае монохромного – одноцветовой план и один бит на точку, единичное значение этого бита задает для точки цвет переднего плана (*foreground*), нулевое – заднего (*backgroung*).

Битовые образы – объекты, идентифицирующиеся их описателями – *HBITMAP*. Различают совместимые и контекстно-независимые объекты *BITMAP*.

Функции

HBITMAP ***CreateBitmap*** ( int nWidth, int nHeight, UINT cPlanes, UINT cBitsPerPel, const void* lpvBits );

HBITMAP ***CreateBitmapIndirect*** (const BITMAP* lpBitmap);

создают объект *BITMAP* с указанными характеристиками, возвращаемое значение – описатель объекта или *NULL* в случае ошибки; параметры:

*nWidth*, *nHeight* – размеры образа в точках изображения;

*cPlanes* – количество цветовых планов;

*cBitsPerPel* – «глубина» цвета;

*lpvBits* – массив данных образа;

*lpBitmap* – структура *BITMAP*, содержащая перечисленные параметры.

Функция

HBITMAP ***CreateCompatibleBitmap*** (hdc, int nWidth, int nHeight);

создает объект *BITMAP* совместимого типа для заданного контекста с заданными размерами; в зависимости от контекста он может быть создан цветным или монохромным (если в контексте заданы данные раздела *DIB* – контекстно-независимым); возвращаемое значение – описатель объекта или *NULL*; *nWidth* и *nHeight* – размеры образа.

Для доступа к содержимому битового образа предусмотрены функции ***SetDlBits***() и ***GetDlBits***(), которые работают построчно, однако имеется возможность воздействовать на него всеми доступными инструментами. Для этого объект *BITMAP* связывается с некоторым контекстом с помощью универсальной функции *SelectObject*(), после чего все изменения в контексте будут отображаться и в битовом образе.

Функции:

BOOL ***BitBlt*** (HDC hDstDC, int nDstX, int nDstY, int nDstWidth, int nDstHeight,HDC hSrcDC, int nSrcX, int nSrcY, DWORD dwRop);

BOOL ***StretchBlt*** (HDC hDstDC, int nDstX, int nDstY, int nDstWidth, int nDstHeight, HDC hSrcDC, int nSrcX, int nSrcY, int nSrcWidth, int nSrcHeight, DWORD dwRop);

BOOL ***MaskBlt*** (HDC hDstDC, int nDstX, int nDstY, int nDstWidth, int nDstHeight, HDC hSrcDC, int nSrcX, int nSrcY, HBITMAP hbmMask, int nMaskX, int nMaskY, DWORD dwRop);

BOOL ***PlgBlt*** (HDC hDstDC, const POINT* lpDstVertices, HDC hSrcDC, int nSrcX, int nSrcY, HBITMAP hbmMask, int nMaskX, int nMaskY, DWORD dwRop);

выполняют перенос прямоугольного фрагмента изображения из контекста-источника в контекст-приемник (с трансформацией и дополнительными операциями). Функция *StretchBlt* может изменять масштаб изображения фрагмента; *MaskBlt* позволяет маскировать часть изображения; *PlgBlt* осуществляет перенос в непрямоугольную область приемника с

соответствующим искажением; возвращаемое значение – признак успешности выполнения; параметры:

*hSrcDC*, *hDstDC* – контексты источника и приемника данных;

*nSrcX*, *nSrcY*, *nDstX*, *nDstY* – координаты фрагмента в обоих контекстах;

*nSrcWidth*, *nSrcHeight*, *nDstWidth*, *nDstHeight* – размеры фрагментов;

*hbmMask* – битовый образ маски, монохромного типа, нулевые точки маски указывают на применение к данной точке изображения операции «заднего плана», единичные – «переднего плана»;

*nMaskX*, *nMaskY* – точка привязки в образе маски;

*lpDstVertices* – массив структур, задающих вершины параллелограмма, образующего фрагмент-приемник;

*dwRop* – дополнительная операция, применяемая к фрагменту при переносе: *SRCCOPY* – простое копирование, *SRCAND* – комбинация цветов источника и приемника по «И», *SRCPAINT* – комбинация по «ИЛИ», *SRCINVERT* – комбинация по «исключающему ИЛИ», *SRCERASE* – комбинация по «И» цвета источника и инверсии цвета приемника, *NOTSRCCOPY*, *NOTSRCERASE* – соответствует одноименным, но результирующий цвет инвертируется, *DSTINVERT* – инверсия фрагмента-приемника, *BLACKNESS*, *WHITENESS* – заполнение фрагмента-получателя цветом соответственно 0 и 1 физической палитры и другие. Для *MaskBlt* параметр включает операции для переднего и заднего фонов, формируется с помощью макроса *MAKEROP*4.

Для успешного применения этих функций требуется, чтобы оба контекста относились к одному устройству или идентичным устройствам.

При использовании функций следует учитывать, что в логических координатных системах, связанных с обоими контекстами, отсчитываются только координаты опорных точек и размеры границ фрагмента, содержимое же его всегда ориентировано одинаково.

Эффекты, возникающие при деформации битового образа, дополнительно управляются функцией ***SetStretchBltMode***, текущая настойка – ***GetStretchBltMode***.

## ЗАДАНИЕ 11. Создание графических изображений

1. Написать программу, которая выводит на экран флаг олимпийских игр (круги разных цветов).

2. Написать программу, которая, используя метод базовой точки, выводит на экран изображение кораблика.

3. Написать программу, которая вычерчивает на экране узор из 100 окружностей случайного диаметра и цвета.

4. Написать программу, которая вычерчивает на экране ломаную линию, состоящую из 200 звеньев, окрашенных в разные цвета, выбираемые случайным образом, причем координаты звеньев тоже выбираются случайно.

5. Написать программу, которая выводит на экран контур пятиконечной звезды.

6. Написать программу, которая рисует флаг Республики Беларусь.

7. Написать программу, которая выводит на экран изображение шахматной доски.

8. Написать программу, которая рисует на экране раскрытую книгу.

9. Написать программу, которая выводит на экран оцифрованную координатную сетку.

10. Написать программу, которая выводит на экран точечный график функции $y = 0{,}5x^2 + 4x - 3$. Диапазон изменения аргумента от -15 до 5 с шагом 0,1. График вывести на фоне координатных осей, точка пересечения которых лежит в центре экрана.

# ЛИТЕРАТУРА


1. Аксенкин, М. А. Язык С / М. А. Аксенкин, О. Н. Целобенок. – Минск : Універсітэцкае, 1995.

2. Архангельский, А. Я. Программирование в С++ Builder 6 / А. Я. Архангельский. – М. : ЗАО «Издательство БИНОМ», 2002.

3. Березин, Б. И. Начальный курс С и С++ / Б. И. Березин, С. Б. Березин. – М. : Диалог– МРТИ, 1999.

4. Берри, В. Язык Си: введение для программистов / В. Берри, Б. Микинз. – М. : Финансы и статистика, 1988.

5. Больски, М. Н. Язык программирования Си: справочник / М. Н. Больски. – М. : Радио и связь, 1988.

6. Программирование: лаб. практикум для студ. 1–2-го курсов всех спец. БГУИР и всех форм обуч. Ч.2. Основы программирования на алгоритмическом языке С / В. Л. Бусько [и др.]. – Минск : БГУИР, 2005.

7. Основы алгоритмизации и программирования: конспект лекций для студ. всех спец. и всех форм обуч. БГУИР / В. Л. Бусько [и др.]. – Минск : БГУИР, 2004.

8. Бухтияров, А. М. Сборник задач по программированию на алгоритмических языках / А. М. Бухтияров, Г. Д. Фролов. – М. : Статистика, 1978.

9. Бабэ, Б. Просто и ясно о *Borland* С++ / Б. Бабэ. – М. : БИНОМ, 1995.

10. Бадд, Т. Объектно-ориентированное программирование в действии / Т. Бадд. – СПб. : Питер, 1997.

11. Вайнер, Р. С++ изнутри / Р. Вайнер, Л. Пинсон – Киев : ДиаСофт, 1993.

12. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – СПб. : Невский диалект, 2001.

13. Вирт, Н. Алгоритмы + структуры данных = программы / Н. Вирт. – М. : Мир, 1985.

14. Дейтел, Х. М. Как программировать на С / Х. М. Дейтел, П. Дж. Дейтел. – М. : БИНОМ, 2006.

15. Демидович, Е. М. Основы алгоритмизации и программирования. Язык Си / Е. М. Демидович. – Минск : Бестпринт, 2001.

16. Джехани, Н. Программирование на языке Си / Н. Джехани // Радио и связь, 1988.

17. Дэвис, С. С++ для «чайников» / С. Дэвис. – Киев : Диалектика, 1996.

18. Касаткин, А. И. Профессиональное программирование на языке Си: От Turbo-C к Borland C++: справ.пособие / А. И. Касаткин, А. Н. Вольвачев. – Минск : Выш. шк., 1992.

19. Касаткин, А. И. Профессиональное программирование на языке Си. Управление ресурсами: справ.пособие / А. И. Касаткин. – Минск : Выш.шк., 1992.

20. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ритчи. – М. : Финансы и статистика, 1992.

21. Керниган, Б. Язык программирования Си. Задачи по языку Си / Б. Керниган, Д. Ритчи, А. Фьюэр. – М. : Финансы и статистика, 1985.

22. Керниган, Б. Универсальная среда программирования UNIX / Б. Керниган, Р. Пайк. – М. : Финансы и статистика, 1992.

23. Климова, Л. И. С++. Практическое программирование / Л. И. Климова. – М. : Кудиц-Образ, 2001.

24. Кнут, Д. Искусство программирования: т. 1–3. Основные алгоритмы / Д. Кнут. – М. : Издательский дом «Вильямс», 2004.

25. Котлинская, Г. П. Программирование на языке Си / Г. П. Котлинская, О. И. Галиновский. – Минск : Выш.шк., 1991.

26. Методы алгоритмизации / В. М. Котов [и др.]. – Минск : Народная асвета, 1996.

27. Котов, В. М. Методы алгоритмизации: учеб. пособие / В. М. Котов, О. И. Мельников. – Минск : Народная асвета, 2000.

28. Котов, В. М. Структуры данных и алгоритмы: учеб. пособие / В. М. Котов, Е. П. Соболевская. – Минск : БГУ, 1996.

29. Программирование на С и С++ / А. В. Крячков [и др.]. – М. : Горячая линия – Телеком, 2000.

30. Морозов, А. А. Структуры данных и алгоритмы: учеб. пособие : в 2 ч. / А. А. Морозов. – Минск : БГПУ им. М. Танка. Ч. 1. – 2000, Ч. 2. – 2001.

31. Павловская, Т. А. С/С++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2004.

32. Павловская, Т. А. С/С++. Структурное программирование : Практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2002.

33. Павловская, Т. А. С++. Объектно-ориентированное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2004.

34. Петзольд, Ч. Программирование для Windows 95 / Ч. Петзольд. – BHV.: СПб. : БИНОМ, 1997.

35. Подбельский, В. В. Программирование на языке Си / В. В. Подбельский, С. С. Фомин. – М. : Финансы и статистика, 2001.

36. Подбельский, В. В. Язык С++ / В. В. Подбельский. – М. : ФиС, 2001.

37. Пол, И. Объектно-ориентированное программирование с использованием С++ / И. Пол. – Киев : ДиаСофт, 1995.

38. Романовская, Л. М. Программирование в среде СИ для ПЭВМ ЕС / Л. М. Романовская, Т. В. Русс, С. Г. Свитковский. – М. : Финансы и статистика, 1992.

39. Практикум по программированию на алгоритмических языках / Г. И. Светозарова [и др.]. – М. : Наука, 1980.

40. Синицын, А. К. Конспект лекций по курсу «Программирование» для студ. 1–2-го курсов всех спец. БГУИР / А. К. Синицын. – Минск : БГУИР, 2001.

41. Синицын, А. К. Программирование алгориттмов в среде *Builder* C++: лаб.практикум по курсам «Программирование» и «Основы алгоритмизации и программирование» для студ. 1-2 курсов всех спец. БГУИР дневн.
и веч. форм обуч.: в 2 ч. / А. К. Синицын. – Минск : БГУИР. Ч. 1. – 2004, Ч. 2. – 2005.

42. Страуструп, Б. Язык программирования C++ / Б. Страуструп. – СПб. : БИНОМ, 1999.

43. Тимофеев, В. В. Программирование в среде C++ *Builder* 5 / В. В. Тимофеев. – М. : БИНОМ, 2000.

44. Топп, У. Структуры данных в C++: пер. с англ. / У. Топп, У. Форд. – М. : ЗАО «Издательство БИНОМ», 2000.

45. Уингер, Р. Язык Турбо Си / Р. Уингер. – М. : Мир, 1991.

46. Успенский, В. А. Теория алгоритмов : основные открытия и приложения / В. А. Успенский, А. А. Семенов. – М., 1987.

47. Уэйт, М. Язык Си. Руководство для начинающих / М. Уэйт, С. Прама, Д. Мартин. – М. : Мир, 1988.

48. Фьюэр, А. Задачи по языку Си / А. Фьюэр. – М. : Финансы и статистика, 1985.

49. Холзнер, С. *Visual* C++ 6. Учебный курс / С. Холзнер. – СПб. : Питер, 1999.

50. Хэнкок, Л. Введение в программирование на языке Си / Л. Хэнкок, М. Кригер. – М. : Радио и связь,1986.

51. Шилд, Г. Программирование на *Borland* C++ / Г. Шилд. – Минск : ПОПУРРИ, 1999.

52. Юлин, В. А. Приглашение к Си / В. А. Юлин, И. Р. Булатова. – Минск : Выш.шк., 1990.

Учебное издание

**Батура** Михаил Павлович
**Бусько** Виталий Леонидович
**Корбит** Анатолий Григорьевич
**Кривоносова** Татьяна Михайловна

# Основы алгоритмизации и программирования. Язык Си

Учебное пособие



Редактор *Т. Н. Крюкова*
Корректор *Е. Н. Батурчик*
Компьютерная верстка *Е. Н. Мирошниченко*

Подписано в печать 21.05.2007. Формат 60х84 1/16. Бумага офсетная. Гарнитура «Таймс».
Печать ризографическая. Усл. печ. л. 14,2. Уч.-изд. л. 14,2. Тираж 500 экз. Заказ 599.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ № 02330/0056964 от 01.04.2004. ЛП № 02330/0131666 от 30.04.2004.
220013, Минск, П.Бровки, 6