Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

Н. В. Пермякова

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ

Учебное пособие

Томск
2016

УДК 004.438 Си (075.8)
ББК 32.973.2-018.1я73
П 275

Рецензенты:
В. В. Герасименко, канд. техн. наук, заместитель начальника отдела информационных технологий КД «Восток»,
П. В. Сенченко, канд. техн. наук, доцент кафедры АОИ ТУСУР

Пермякова Н. В.
П 275 Информатика и программирование : учебное пособие / Н. В. Пермякова. – Томск : ФДО, ТУСУР, 2016. – 188 с.

Учебное пособие предназначено для студентов направлений 231000.62 «Программная инженерия», 080500.62 «Бизнес-информатика», а также всех, начинающих изучать основы структурного программирования на языке Си.

Рассмотрены основные аспекты алгоритмизации, описаны синтаксис и алфавит языка Си, изложены основы структурного программирования в примерах на языке Си.

Для закрепления полученных знаний каждый раздел пособия содержит вопросы для самоконтроля и задания для выполнения.

Оглавление

Введение

Данное учебное пособие должно дать студентам, обучающимся по направлениям 231000.62 «Программная инженерия» и 080500.62 «Бизнес-информатика», основные понятия алгоритмизации и структурного программирования. Пособие будет полезно начинающим программистам и тем, кто хочет самостоятельно научиться программировать и освоить язык программирования Си.

Содержание разделов учебника основано на лекционном курсе, который автор читает для студентов дневной формы обучения кафедры автоматизированной обработки информации Томского государственного университета систем управления и радиоэлектроники.

В пособии рассмотрены основные приемы алгоритмизации, способы представления алгоритмов. В качестве инструментального средства для реализации полученных знаний в пособии рассматривается язык программирования Си.

Задания, предложенные для самостоятельного выполнения, фактически являются заданиями на практические занятия и лабораторные работы, выполняемые студентами дневной формы обучения. Выполнение этих заданий поможет подготовиться к решению компьютерной контрольной работы, выполнению лабораторных работ и последующей сдаче экзамена.

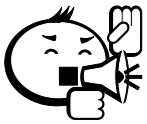
Можно выделить следующие основные цели, преследуемые учебником:

- познакомить с основами алгоритмизации и структурного программирования, это необходимо для дальнейшей профессиональной деятельности студентов направлений «Программная инженерия» и «Бизнес-информатика»;
- познакомить с базовыми алгоритмами. Знание базовых алгоритмов необходимо будущему программисту, поскольку существенно помогает в навыках разработки алгоритмов и избавляет от необходимости «изобретать велосипед»;
- научить разрабатывать алгоритмы поставленных задач;
- научить представлять алгоритмы на языке Си.

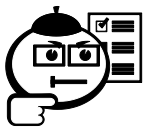
Соглашения, принятые в книге

Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.

Эта пиктограмма означает определение или новое понятие.

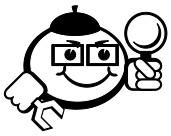


Эта пиктограмма означает «Внимание!». Здесь выделена важная информация, требующая акцента на ней. Автор может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.



Эта пиктограмма означает задание.

Эта пиктограмма означает совет. В данном блоке можно указать более простые или иные способы выполнения определенной задачи. Совет может касаться практического применения только что изученного или содержать указания на то, как немного повысить эффективность и значительно упростить выполнение некоторых задач.



Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Эта пиктограмма означает выводы. Здесь автор подводит итоги, обобщает изложенный материал или проводит анализ.

Контрольные вопросы по главе

1 Основы алгоритмизации

1.1 Основные понятия и определения

Начнем курс с некоторых определений, без которых не возможна дальнейшая работа. Основным понятием данного раздела является понятие алгоритма. Само слово «алгоритм» произошло от латинского слова *algorism* – правило выполнения арифметических действий с использованием арабских цифр (около 825 г.). Постепенно слово изменилось, и первоначальный вид и значение забылись. В XVII в. в словарях встречается слово *algorithmus* – объединение понятий о четырех типах арифметических действий. К 1950 г. под словом «алгоритм» чаще всего подразумевали так называемый алгоритм Евклида – нахождение наибольшего общего делителя (НОД). Приведем этот алгоритм:

Условие – даны два положительных числа m и n. Найти их наибольший общий делитель.

1. Разделить m на n. Пусть остаток равен r.
2. Если $r = 0$, то алгоритм закончен; n – наибольший общий делитель.
3. Заменим: $m = n$, $n = r$. Перейти к шагу 1.

Эти же самые действия могут быть записаны в виде блок-схемы, представленной на рисунке 1.1.

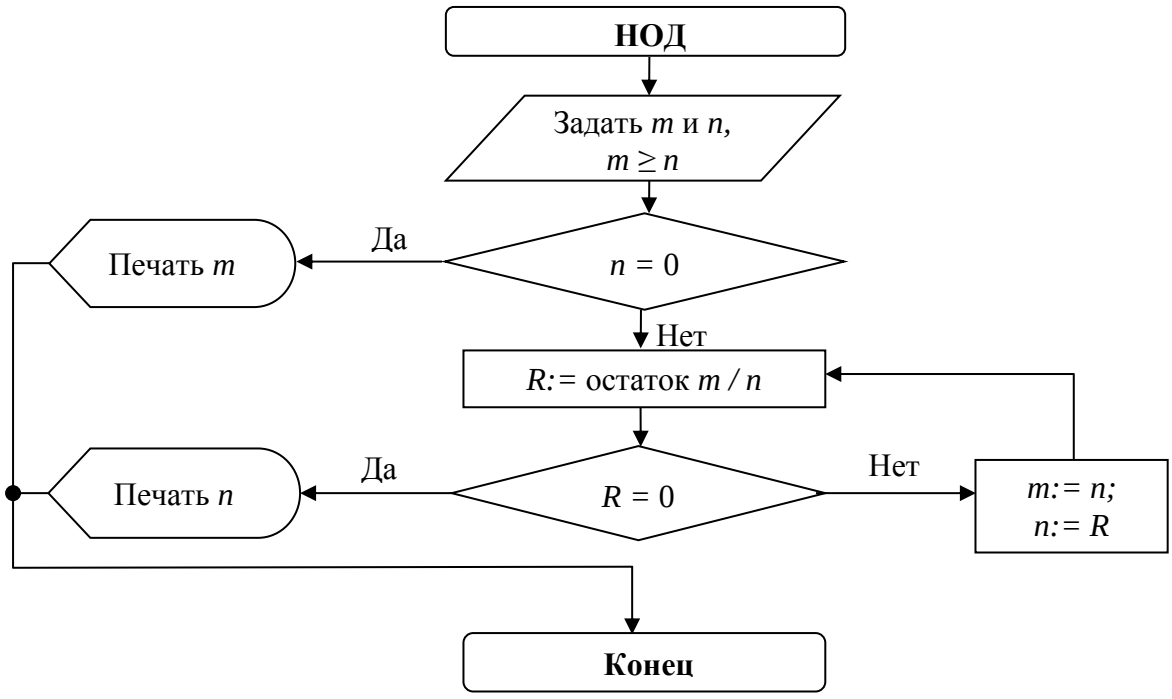


Рис. 1.1 – Блок-схема алгоритма нахождения наибольшего общего делителя

Существует множество различных определений алгоритма. Приведем наиболее часто используемые.

Интуитивное определение

Алгоритм *– последовательность действий, которую необходимо выполнить для достижения цели.*

Математическое определение

Алгоритм *– процесс построения величин, идущий в дискретном времени, который позволяет из системы величин в предыдущий момент времени получить систему величин в последующий момент, для которого задается начальная система и сформулировано правило окончания процесса.*

Из второго определения видно, что первая часть алгоритма обязательно должна описывать данные, названные начальными системами. Вторая часть алгоритма – это действия над этими данными. Таким образом, можно дать третье определение алгоритма, *общее*:

Алгоритм *– это описание данных и действий, производимых над ними для получения нужного результата.*

Однако необходимо отметить, что различные определения алгоритма требуют выполнения следующих условий [1]:

- алгоритм должен содержать конечное количество шагов;
- алгоритм должен выполнять конечное количество шагов при решении задачи;
- алгоритм должен быть единым для всех допустимых исходных данных;
- алгоритм должен приводить к правильному по отношению к поставленной задаче решению.

Таким образом, любой алгоритм должен удовлетворять:

- требованию конечности записи;
- требованию конечности действий;
- требованию универсальности;
- требованию правильности.

1.2 Типы данных

Данные *– это символьная или числовая информация, которая обрабатывается или используется в процессе выполнения алгоритма.*

Можно привести несколько оснований для классификации данных.

Во-первых, все данные делятся на входные, выходные и внутренние.

Входными *называются данные, значения которых вводятся в алгоритм извне.*

Выходными *называют данные, значения которых формируются в результате работы алгоритма.*

Внутренними *называют данные, которые используются внутри алгоритма.*

Такие данные часто называются локальными данными.

Во-вторых, данные делятся на константы и переменные.

Константы *– данные, не изменяющие свое значение на протяжении работы всего алгоритма.*

Переменные *– данные, значения которых могут изменяться в процессе работы алгоритма.*

И константы, и переменные имеют имена. Имена данных могут состоять из символов алфавита и цифр. Например, `A = 5`. «A» – имя константы, значение константы `A` равно `5`. К данным-переменным операция присваивания := может применяться один или несколько раз. Например: `k:=A, k:=A+10`.

Переменные алгоритма могут быть счетчиками (специальные переменные, по которым ведется счет каких-либо действий, выполняемых алгоритмом), флагами (еще один вид специальных переменных, которые принимают фиксированные значения в зависимости от выполнения или невыполнения условия).

В-третьих, данные могут быть ***простыми*** – целое число, вещественное число, символ, логическая переменная; и ***сложными*** – массивы простых данных, матрицы простых данных.

Массив – это вектор, который имеет конечное число элементов. Например, массив `x={1,3,5,7}` состоит из четырех элементов. Каждый элемент массива имеет свой ***индекс*** – целое число, определяющее положение элемента в массиве. Существует несколько видов записей индексов, например: $x_1=1$ или `x[2]=3`. Читается икс первое равно единице, икс второе равно трем.

Строка – это пример массива из символов. Например, `z = "предмет"; z[1]= "п"; z[2]= "р"; z[3]= "е"`.

Если массив подобен строке, то данные, собранные в таблицы, называются матрицами. Например: $S = \begin{bmatrix} 1 & 6 & 7 \\ 4 & 2 & 3 \\ 1 & 6 & 9 \end{bmatrix}$. Аналогично массиву каждый элемент матрицы имеет свое определенное место в таблице, которое определяется двумя индексами: первый определяет номер строки, в которой находится элемент, второй определяет номер столбца, в котором находится элемент. Например, $S_{2,3}=3$ или `S[3,1]=1`.

1.3 Структурное программирование

Структурное программирование *– методология разработки программного обеспечения, основанная на представлении программы в виде иерархической схемы блоков.*

В структурном программировании используются только три основные конструкции. Эти конструкции допускают последовательную, условную и итеративную передачу управления. Как следствие применения этих конструкций каждая сложная команда в программе имеет ровно одну точку входа и ровно одну точку выхода. А это, в свою очередь, облегчает восприятие программы.

Очень часто говорят, что структурное программирование – это программирование без `goto`.

Итальянские математики Коррадо Бём (*Corrado Böhm*) и Джузеппе Якопини (*Giuseppe Jacopini*) сформулировали и доказали теорему о структурном программировании в 1965 г. В 1966 г. теорема была опубликована в [2].

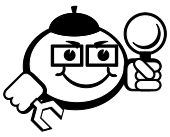
При использовании современной терминологии и обозначений, теорема может быть записана следующим образом:

Любая программа, заданная в виде блок-схемы, может быть представлена с помощью трех управляющих структур:

- последовательность, обозначается: `f THEN g`;
- ветвление, обозначается: `IF p THEN f ELSE g`;
- цикл, обозначается: `WHILE p DO f`, где `f`, `g` – блок-схемы с одним входом и одним выходом, `p` – условие, `THEN`, `IF`, `ELSE`, `WHILE`, `DO` – ключевые слова [3].

Пояснение. Формула `f THEN g` означает следующее: сначала выполняется программа `f`, затем выполняется программа `g`.

Приведем несколько примеров алгоритмов.

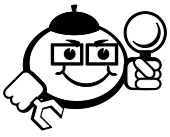


Пример 1.1

Последовательность:

```
k:=12;
z:=3;
p:=z*sin(π/12);
```

Действия выполняются последовательно, одно за другим.

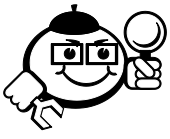


Пример 1.2

Ветвление:

```
k:=13;
z:=4;
IF (k>z) THEN f:=1 ELSE f:=0;
```

Если значение переменной `k` больше значения переменной `z`, то переменной `f` присвоить значение `1`, в противном случае переменной `f` присвоить значение `0`.



Пример 1.3

Цикл

```
p:=12;
WHILE (p>1) p := p – 3;
```

Пока значение переменной `p` больше единицы переменную `p` уменьшать на три.

В своей работе [4] Э. Дейкстра сформулировал принципы структурного программирования:

1. Следует отказаться от использования оператора безусловного перехода `goto`.
2. Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.
3. В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом. Никаких других средств управления последовательностью выполнения операций не предусматривается.
4. Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций). Таким же образом (в виде подпрограмм) можно оформить логически целостные фрагменты программы, даже если они не повторяются.
5. Каждую логически законченную группу инструкций следует оформить как блок. Блоки являются основой структурного программирования. Понятие «блок» означает, что к блоку инструкций следует обращаться как к единой инструкции.
6. Все перечисленные конструкции должны иметь один вход и один выход.
7. Разработка программы ведётся пошагово, методом «сверху вниз».

1.4 Системы кодирования алгоритмов

1.4.1 Система «псевдокод»

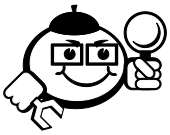
Для описания алгоритмов существует множество способов, алгоритмы могут быть описаны естественным языком, формальным языком (языком программирования) или схемой.

Псевдокод представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов. Псевдокод занимает промежуточное место между естественным и формальным языками. Примером псевдокода является язык, описанный в [5]. Основные служебные слова этого языка приведены в таблице 1.1.

Таблица 1.1 – Служебные слова алгоритмического языка (псевдокода)

алг (алгоритм)	сим (символьный)	дано	для	да
арг (аргумент)	лит (литерный)	надо	от	нет
рез (результат)	лог (логический)	если	до	при
нач (начало)	таб (таблица)	то	знач	выбор
кон (конец)	нц (начало цикла)	иначе	и	ввод
цел (целый)	кц (конец цикла)	все	или	вывод
вещ (вещественный)	длин (длина)	пока	не	утв

Запишем несколько алгоритмов на псевдокоде.

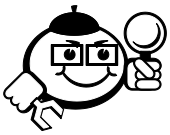


Пример 1.4

Даны два катета прямоугольного треугольника k1=3, k2=4. Найти гипотенузу. Записать решение задачи на псевдокоде.

```
алг Вычисление гипотенузы нач
дано k1=3, k2=4.
z:=k1^2     // Вычислить квадрат k1 и запомнить в переменной z
z2:=k2^2    // Вычислить квадрат k2 и запомнить в переменной z2
g=√(z+z2)  // Вычислить гипотенузу и запомнить в переменной g
рез g.
кон
```

Текст, выделенный курсивом, называется комментарием, пояснением к выполняемым действиям. Если в решаемой задаче требуется вычислить какое-либо значение (алгоритмы подобного рода называются вычислительными алгоритмами), то последним, завершающим действием алгоритма будет возвращение полученного результата.



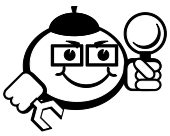
Пример 1.5

Даны два произвольных числа. Найти максимальное число.

```
алг Поиск максимального нач
вещ a,b
ввод a,b
если a≠b то
если a<b то рез «b – максимальное число»
иначе рез «a – максимальное число»
```

```
иначе рез «числа равны»
кон
```

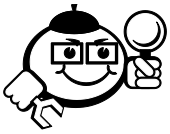
Следующий пример демонстрирует алгоритм с циклом с заданным количеством повторов.



Пример 1.6

Дана строка символов x длины n. Найти количество вхождений символа *y* в строку x.

```
алг Количество заданных символов нач
цел n
сим x[n], y
ввод x,y.
цел i:=0 // в строке x не найдено еще ни одного символа.
цикл для j от 1 до n /* если в цикле не указан шаг изменения переменной, то по умолчанию он считается равным 1. Таким образом, переменная j принимает значения 1,2,3, … n.*/
нц
если x[j]=y то i:=i+1    /* если текущий j-й символ строки равен y, то увеличить текущее значение переменной i на 1.*/
кц
рез i.
кон
```



Пример 1.7

Найти количество символов произвольной строки x до первой встреченной точки.

```
алг Количество символов до точки нач
сим x[255]
ввод x;
цел j:=1; // начинаем просмотр строки с первого символа
цел i:=0; //считаем количество символов равным нулю
пока x[j]≠'.' и j ≤ длин(x) /*пока не встречена точка и не найден конец строки*/
нц
i:=i+1;        // увеличиваем счетчик найденных символов i
j:=j+1;   // увеличиваем счетчик длины строки j
кц
```

```
если i=длин(x) то рез «Точки в данной строке нет»
иначе рез «Длина строки до точки равна i»
кон
```

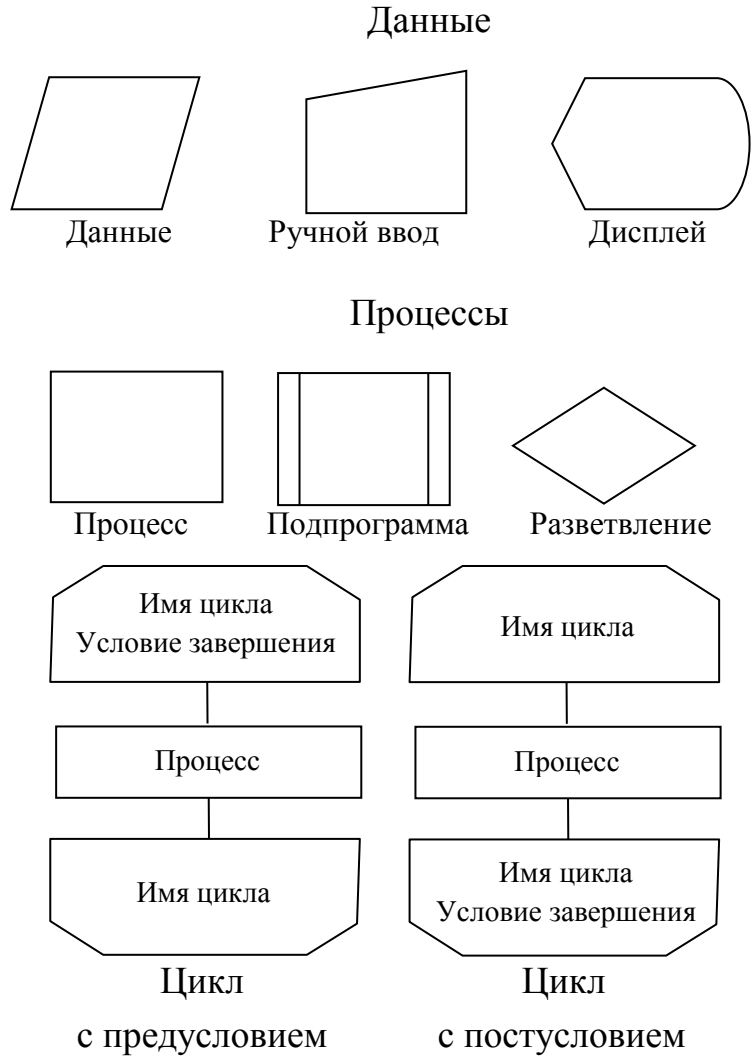
. . . . . .

В примере 1.7 в цикле использовалось сложное условие – два условия, связанных союзом **И**. Такое условие считается истинным тогда и только тогда, когда истинны оба условия. В сложных условиях может также использоваться союз **ИЛИ** – такое условие считается ложным тогда и только тогда, когда ложны оба условия.

1.4.2 Блок-диаграммы

Алгоритм можно задать, используя схему, которая носит название блок-диаграммы. Правила начертания элементов, соответствующих основным конструкциям структурного программирования, определены ГОСТ 19.701–90.

Приведем основные обозначения, соответствующие конструкциям структурного программирования и основным элементам программ:

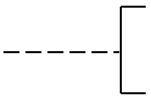


Специальные символы

Линия (для обозначения потоков данных или управления)

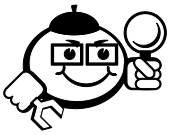
Соединитель (для обрыва линии и ее продолжения в другом месте)

Терминатор (для указания входа и выхода во внешнюю среду)



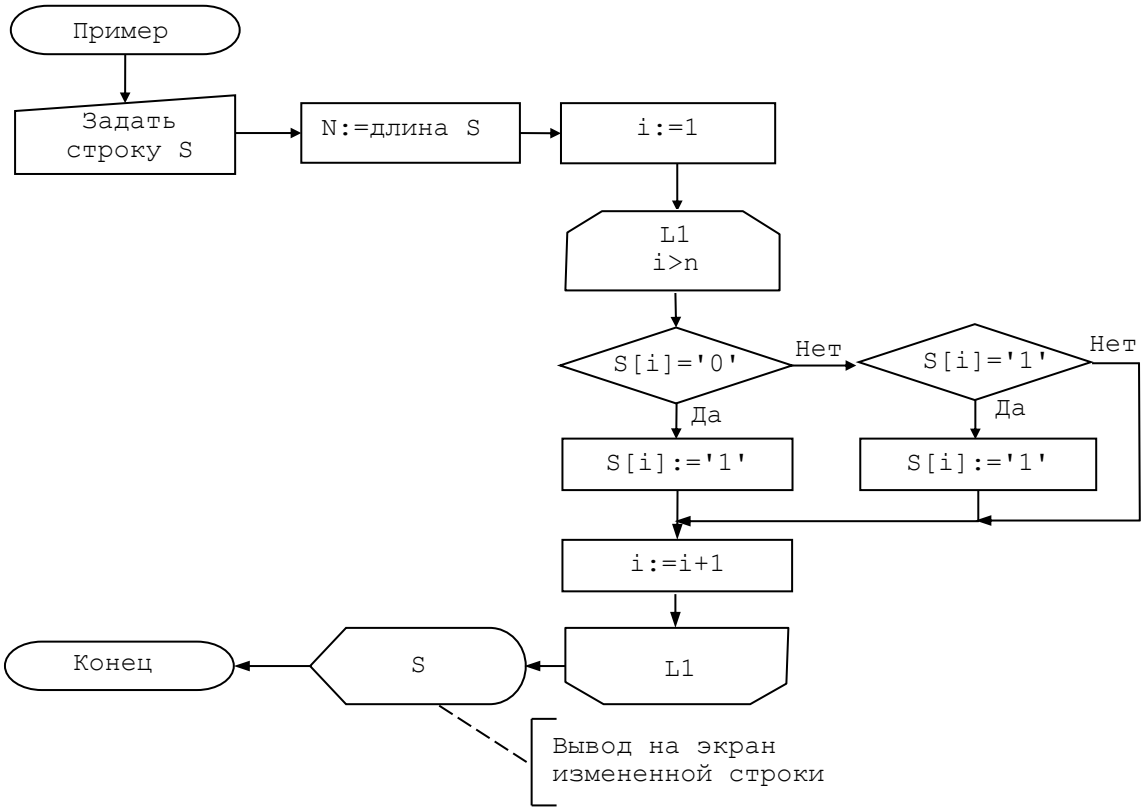
Комментарий

Построим блок-диаграмму для примера 1.8.



Пример 1.8

Дана строка символов `S`. Заменить все вхождения символа «`1`» на «`0`», а «`0`» на «`1`».



1.4.3 Диаграммы Насси – Шнейдермана

Основные конструкции структурного программирования в диаграмме Насси – Шнейдермана обозначаются следующим образом:

Следование –

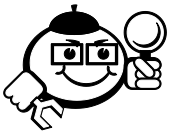
Действие 1
Действие 2

Развилка –

Условие	
Действия при истинном условии	Действия при ложном условии

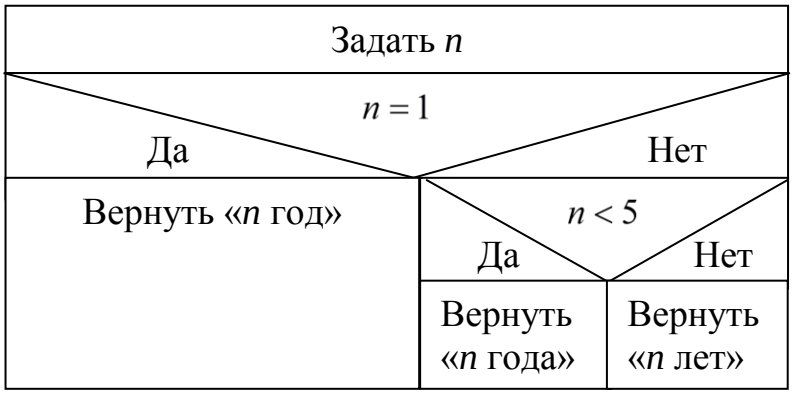
Цикл –

Условие выполнения цикла	
	Действия в цикле
	Действия в цикле
	Действия в цикле



Пример 1.9

Дано натуральное число $n < 10$. Вывести на экран грамматически верную фразу (n лет, 1 год, 2 года, 10 лет и т. д.).



1.5 Основные алгоритмы

1.5.1 Алгоритмы суммы и произведения

Условие: найти сумму чисел от заданного числа `N1` до заданного числа `N2` с шагом 1.

Запишем алгоритм на псевдокоде:

```
алг Сумма нач
```

```
цел S=0;
цикл для i от N1 до N2
нц
S:=S+i
кц
рез S
кон
```

Переменная S в данном случае сыграла роль накопителя, в ней постепенно накапливалась вся сумма.

Условие: найти произведение чисел от заданного N1 до заданного числа N2 с шагом 1.

Запишем алгоритм на псевдокоде:

```
алг Произведение нач
цел P=1;
цикл для i от N1 до N2
нц
P:=P*i
кц
рез P
кон
```

В данном примере роль накопителя сыграла переменная P. В ней на каждом шаге цикла накапливалось произведение.

1.5.2 Алгоритмы поиска

Алгоритм поиска элемента с заданным значением

Условие: найти элемент с заданным значением a в массиве X размерности n.

Определим последовательность действий для достижения поставленной цели: просмотрим все элементы массива X[i] и сравним их значения с заданным значением a. Если текущее значение X[i] равно a, то вернем в качестве результата индекс найденного элемента. Предусмотрим ситуацию промаха при поиске – в массиве не найден ни один элемент с заданным значением. Для разрешения этой проблемы введем в алгоритм *флаговую* переменную f, которой в начале решения присвоим нулевое значение. Если при просмотре массива найдено хотя бы одно заданное значение, то присвоим переменной f любое ненулевое значение. После выполнения всех шагов цикла просмотра массива останется только проверить значение флаговой переменной.

```
алг Поиск элемента с заданным значением нач
цел f, n, i
вещ X[n], a
ввод массива X из n элементов;
ввод a;
f:=0;
цикл для i от 1 до n нц
если x[i]=a то рез i, f:=1;
кц
если f=0 то рез «В массиве нет элементов с заданным значением»
кон
```

Алгоритмы поиска максимума и минимума в массиве

Условие: найти минимальный элемент в массиве.

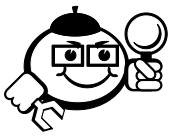
Схема поиска элемента с минимальным значением может быть следующей: определим как минимальный элемент первый элемент массива. Начнем просматривать массив со второго элемента и до последнего. Если текущее значение минимума стало больше просматриваемого элемента массива, то заменим текущее значение минимума на значение просматриваемого элемента массива.

```
алг Поиск минимального элемента в массиве нач
цел i, n
вещ X[n], min
ввод массив X из n элементов;
min=X[1];
цикл для i от 2 до n нц
если X[i]<min то min:=X[i]
кц
рез min;
кон
```

Используя эту же схему, попытайтесь самостоятельно записать алгоритм нахождения максимального элемента в массиве.

1.6 Примеры решения задач

Запишите алгоритм, используя псевдокод.



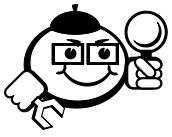
Пример 1.10

Дана вещественная матрица A[n×m], найти количество строк, содержащих нулевые элементы.

Решение

Алгоритм решения задачи выглядит следующим образом: определяется переменная S и ей присваивается начальное значение S = 0. Далее организуется просмотр матрицы по строкам от первой и до последней. Если в текущей строке найден элемент с нулевым значением, то значение переменной S увеличивается на единицу и просмотр текущей строки завершается. После того как будет завершен просмотр матрицы, в переменной S будет храниться количество строк матрицы, содержащих элементы с нулевым значением.

```
алг Задание 1 нач
цел i, j, n, m, S := 0
вещ A[n×m]
ввод A из n строк и m столбцов
цикл для i от 1 до n нц
     цикл для j от 1 до m нц
          если A[i][j] = 0 то S:=S+1 j := m+1;
     кц
кц
рез S
кон
```



Пример 1.11

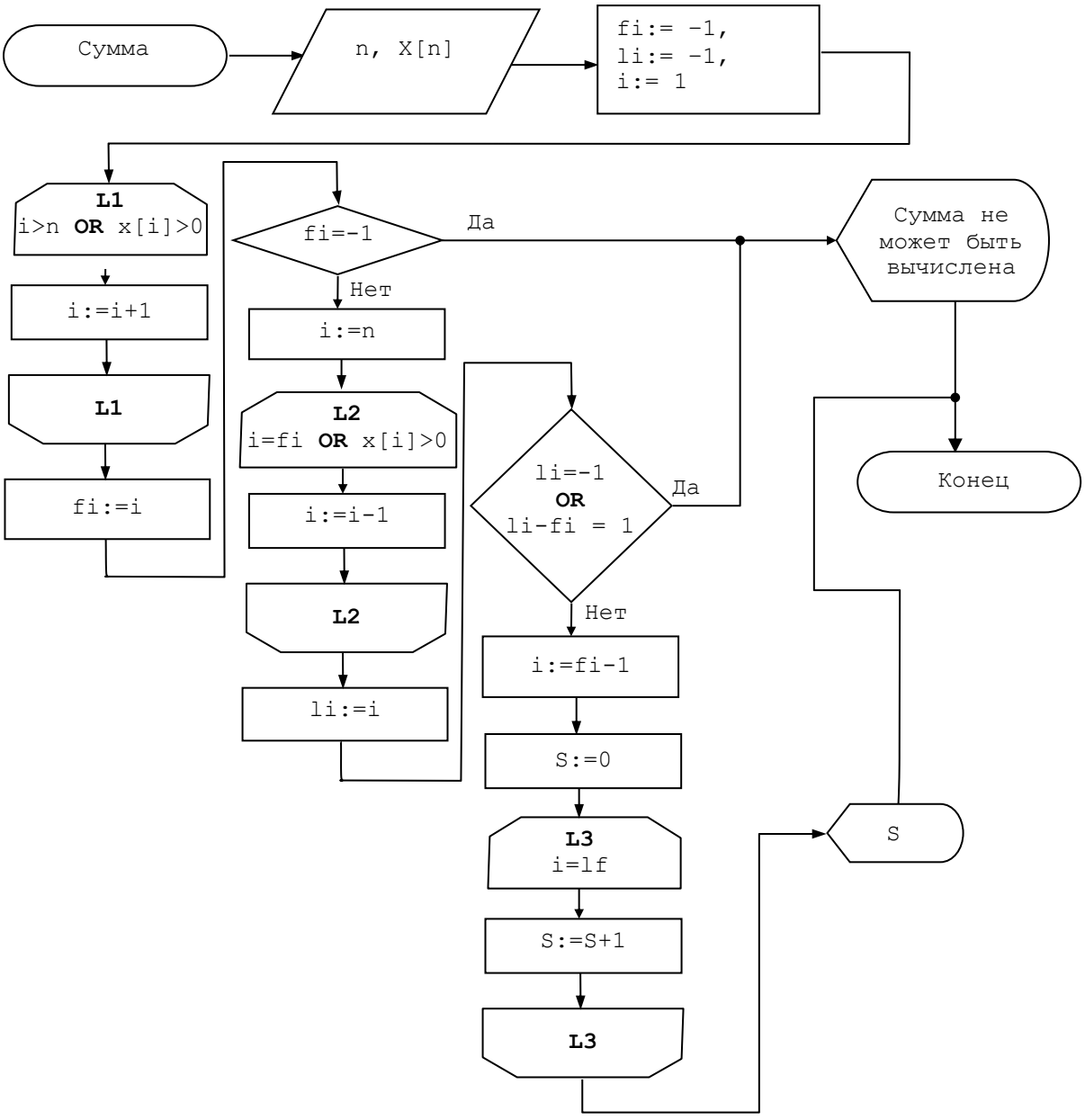
Запишите алгоритм, используя блок-диаграмму.

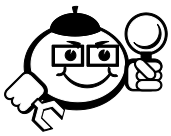
Дан массив целых чисел. Найти сумму элементов массива, расположенных между первым и последним положительными элементами.

Решение

Алгоритм решения задачи выглядит следующим образом: определим две переменные i_f и i_l, для хранения значений индекса первого и последнего положительных элементов. Значения этих переменных установим в −1 (если в массиве будет менее двух положительных элементов, значение −1 будет информировать о том, что сумма элементов не может быть найдена). Для нахождения элементов индекса первого положительного элемента просмотрим мас-

сив с начала, пока не найдем положительный элемент либо пока не просмотрим весь массив. Если положительный элемент не найден, закончим работу алгоритма с сообщением о невозможности вычислить сумму. В противном случае, для поиска последнего положительного элемента просмотрим весь массив, начиная с последнего элемента, пока не найдем положительный элемент или не достигнем элемента, стоящего перед первым положительным. Если второй положительный элемент не найден, закончим работу алгоритма с соответствующим сообщением. В противном случае просуммируем все элементы, лежащие между первым и последним положительными элементами. Блок-диаграмма алгоритма может быть записана следующим образом:





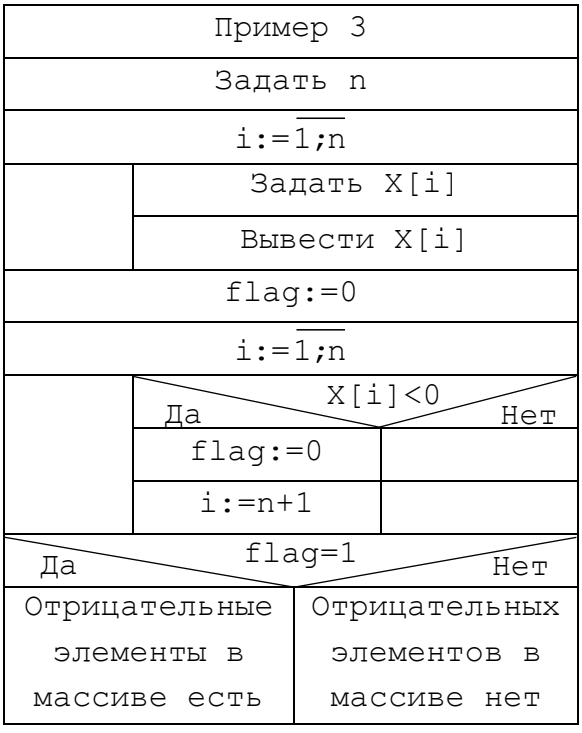
Пример 1.12

Запишите алгоритм, используя диаграмму Насси – Шнейдермана.

Дан массив целых чисел. Необходимо проверить, существуют ли в массиве отрицательные элементы.

Решение

Зададим количество элементов массива и значения элементов массива. Введем переменную-флаг и присвоим ей значение `0` (отрицательные элементы не найдены). Организуем цикл для просмотра элементов массива. На каждом шаге цикла проверим текущий элемент массива. Если он отрицательный, то значение флага установим в `1` (отрицательный элемент найден) и закончим выполнение цикла. Если текущий элемент не отрицательный, то продолжим выполнение цикла. После завершения цикла выполним проверку значения флага: если флаг остался равным `0`, то закончим выполнение алгоритма с сообщением об отсутствии отрицательных элементов, в противном случае сообщим о существовании отрицательных элементов и также закончим программу.



Контрольные вопросы по главе 1

1. Дайте определение алгоритма.
2. Перечислите основные конструкции структурного программирования.
3. Приведите пример алгоритма, который использует конструкцию следования.
4. Приведите пример алгоритма, который использует конструкцию развилки.
5. Приведите пример алгоритма, в котором используется циклическая конструкция.
6. Для чего может быть использована флаговая переменная?
7. Запишите алгоритм поиска максимального элемента в заданном массиве. Используйте любую систему кодирования.
8. Запишите алгоритм поиска индекса минимального значения в заданном массиве.
9. Запишите словесную постановку задачи, которая может быть решена следующим алгоритмом:

```
алг Вопрос 9 нач
ввод вещ x,y,z;
если x=y=z
то рез «Все числа одинаковы»
иначе если x>y то max=x
иначе max=y
если max<z то max=z
рез max
кон
```

10. Какие значения примут переменные x, y, z при выполнении следующего алгоритма:

```
алг Вопрос 10 нач
вещ x:=1, y:=7, z:=-5;
если x>0 и z>0
то x:=x/2, y:=y/2, z:=z/2
иначе x:=10*x, y:=10*y, z:=10*z
рез x,y,z
кон
```

11. Перечислите системы кодирования алгоритмов.

12. Запишите алгоритм вычисления значений функции, заданной следующим образом:

$$f(x)=\begin{cases} x^2, -\infty < x < -5 \\ x/2, -5 \le x < 0 \\ x^2+1, 0 \le x < \infty \end{cases}$$

13. Запишите алгоритм решения произвольного квадратного уравнения $ax^2+bx+c=0$. Какие конструкции структурного программирования использовались в алгоритме?
14. Запишите алгоритм, подсчитывающий количество слов заданной строки, начинающихся с заданного символа.

2 Интегрированная среда программирования *DEV-CPP*

2.1 Подготовка программного обеспечения для работы в среде *DEV-CPP*

DEV-CPP – свободно распространяемая среда разработки, небольшая по размерам, может быть использована новичками в программировании. Вместе со средой разработки поставляется компилятор *GNU CC.*

Перепишите с компакт-диска архив-инсталлятор *Dev-Cpp5.8.2DM-GCC4.8.1Setup.exe* на Ваш компьютер. Запустите инсталлятор. На рисунках 2.1–2.9 показан процесс установки.

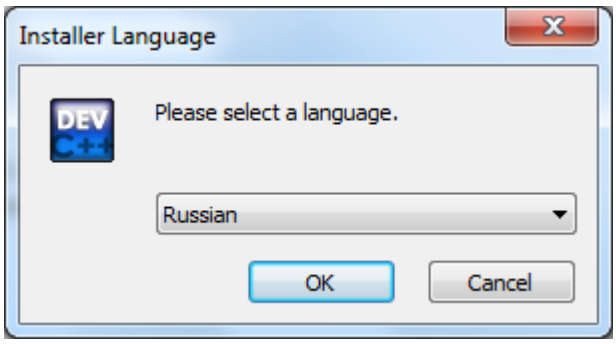


Рис. 2.1 – Выбор языка среды IDE

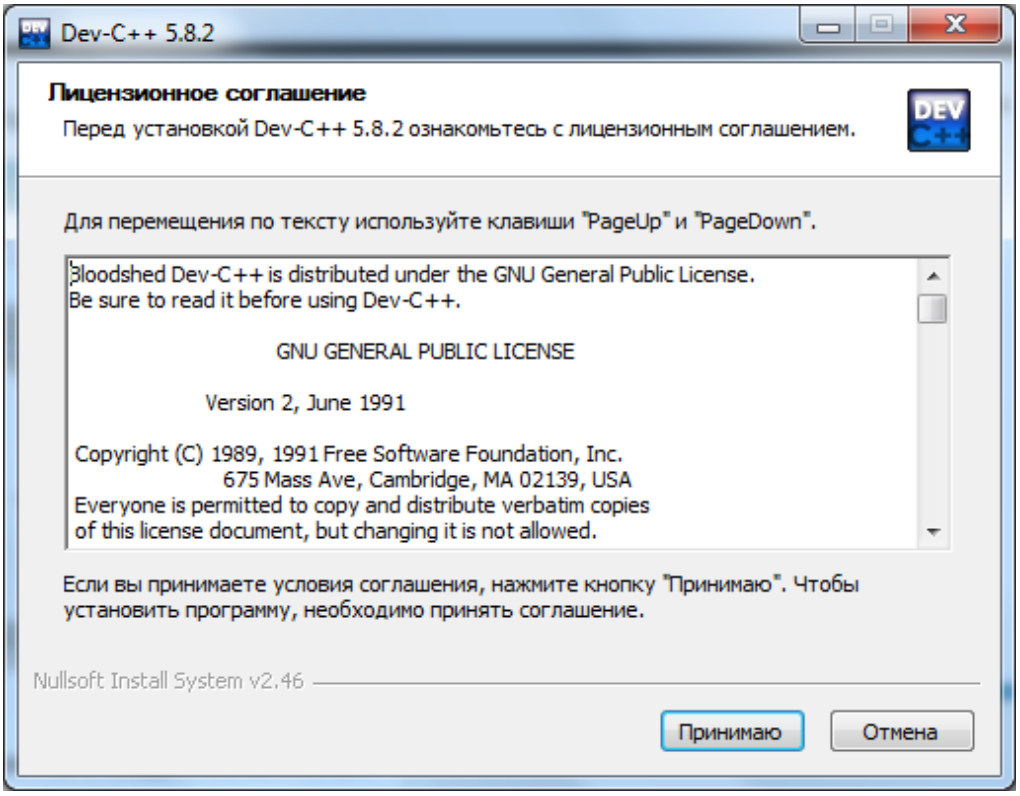


Рис. 2.2 – Лицензионное соглашение

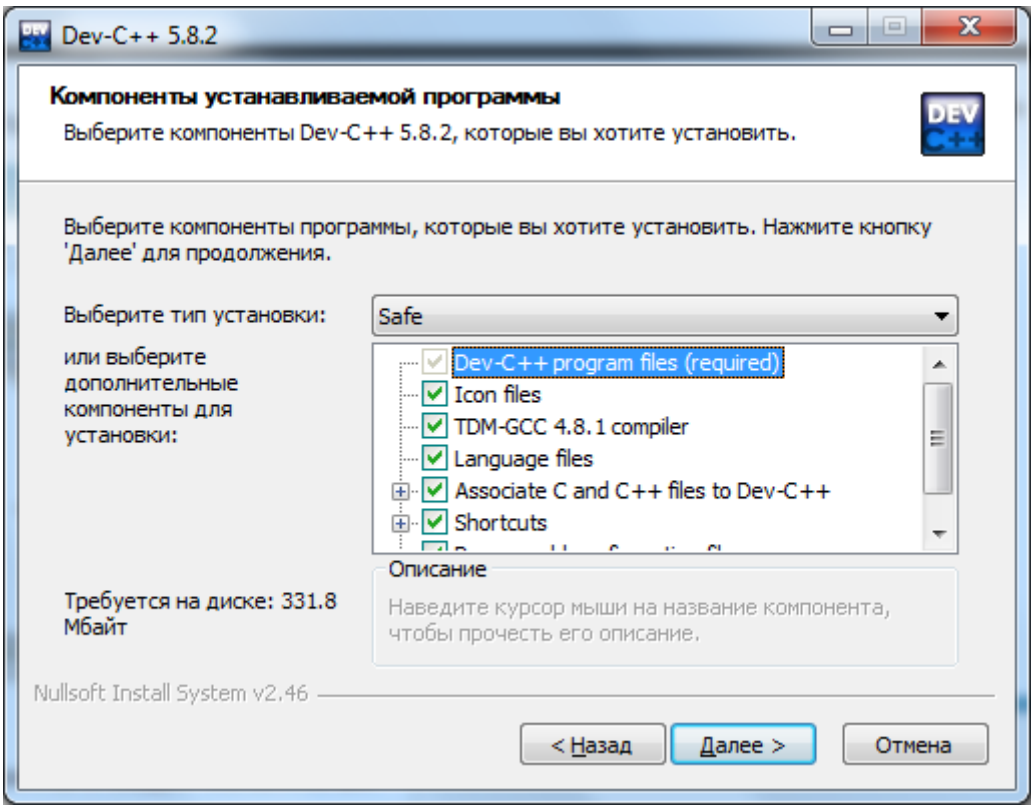


Рис. 2.3 – Выбор устанавливаемых компонентов

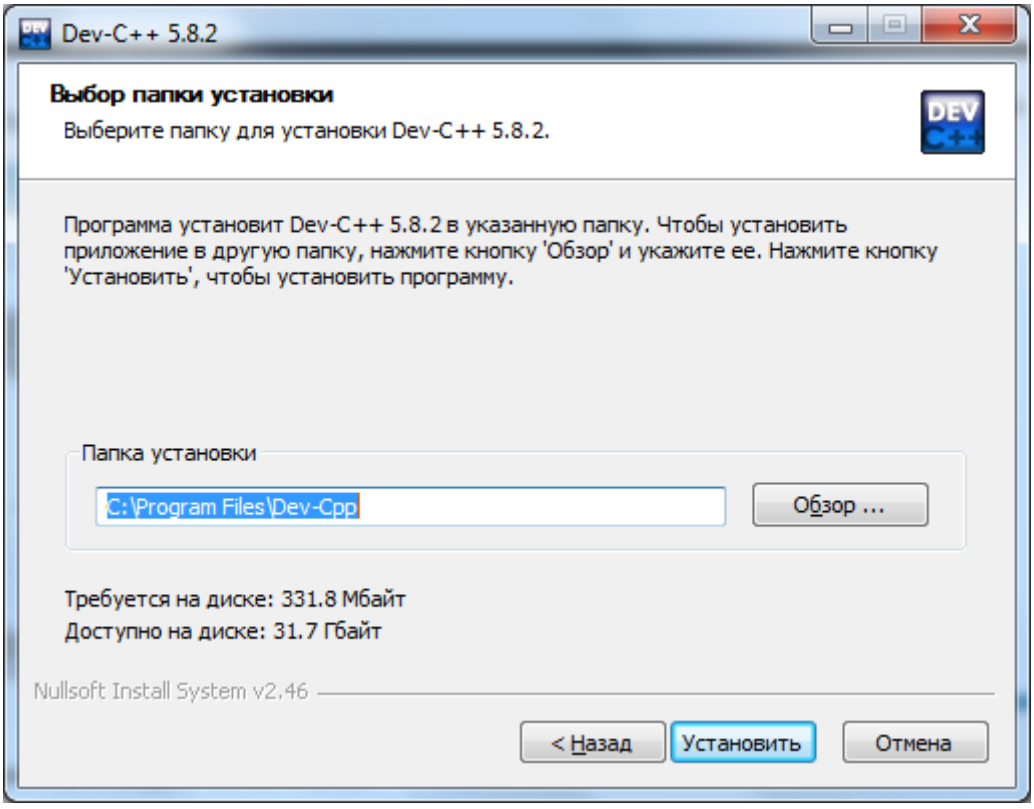


Рис. 2.4 – Выбор папки установки

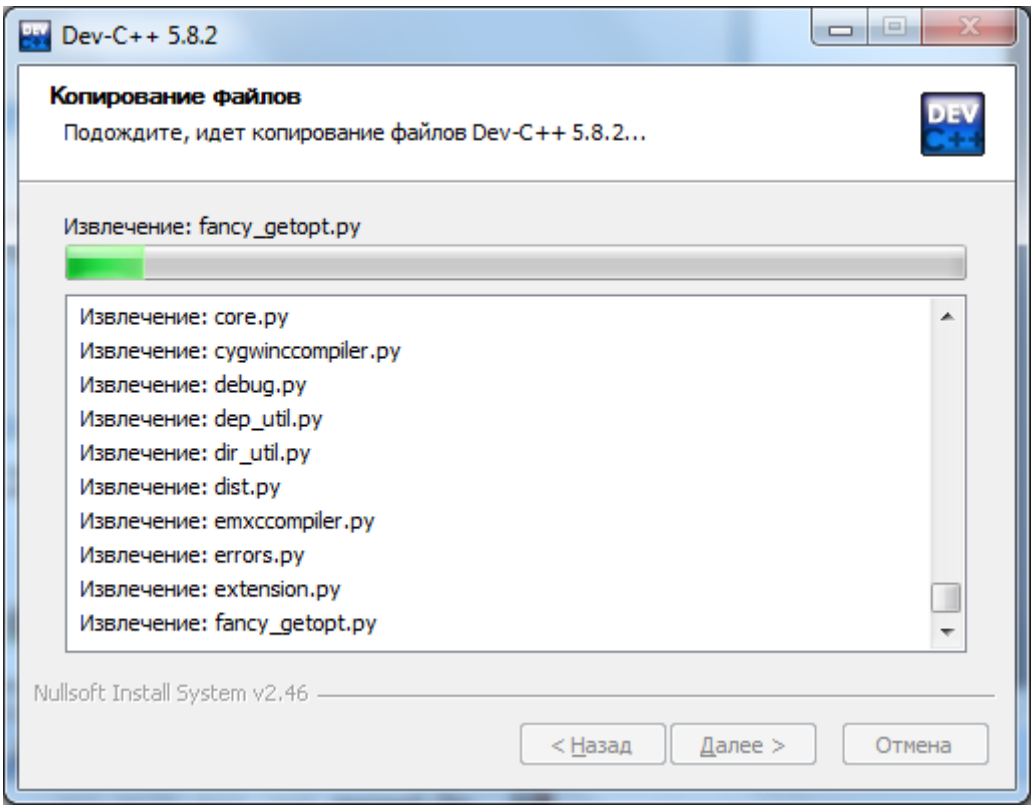


Рис. 2.5 – Процесс извлечения файлов

После успешного извлечения файлов откроется окно начальной конфигурации (рис. 2.6).

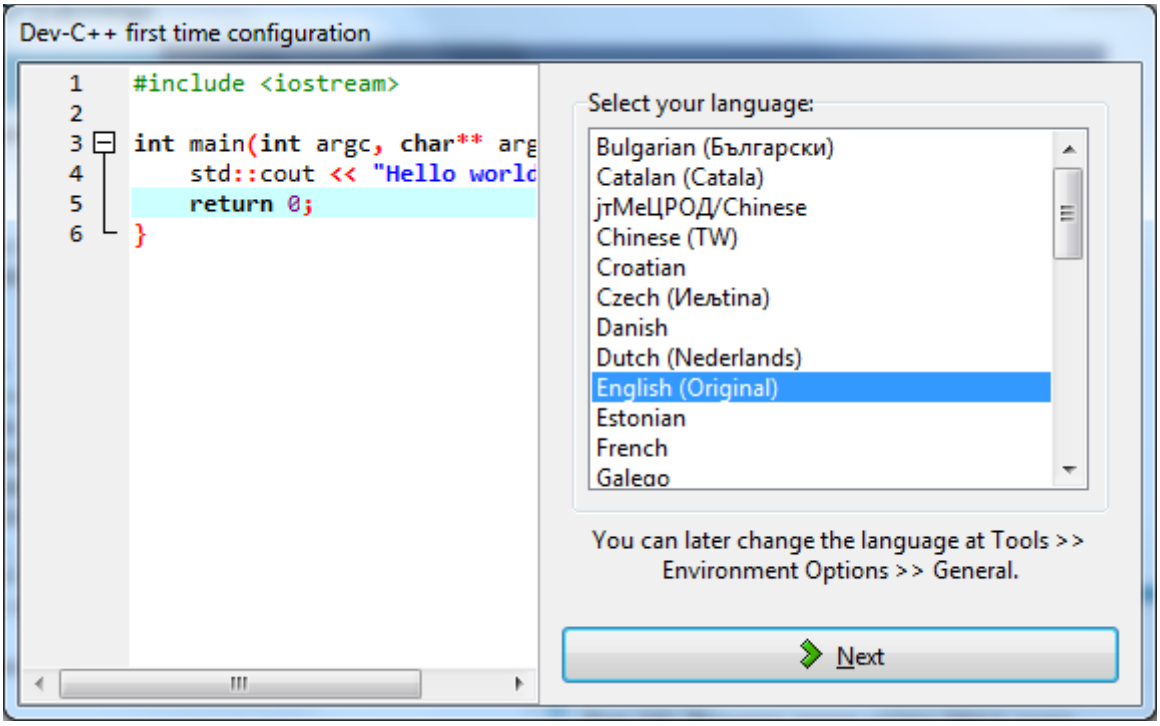


Рис. 2.6 – Выбор конфигурации среды

Выберите язык и оформление среды:

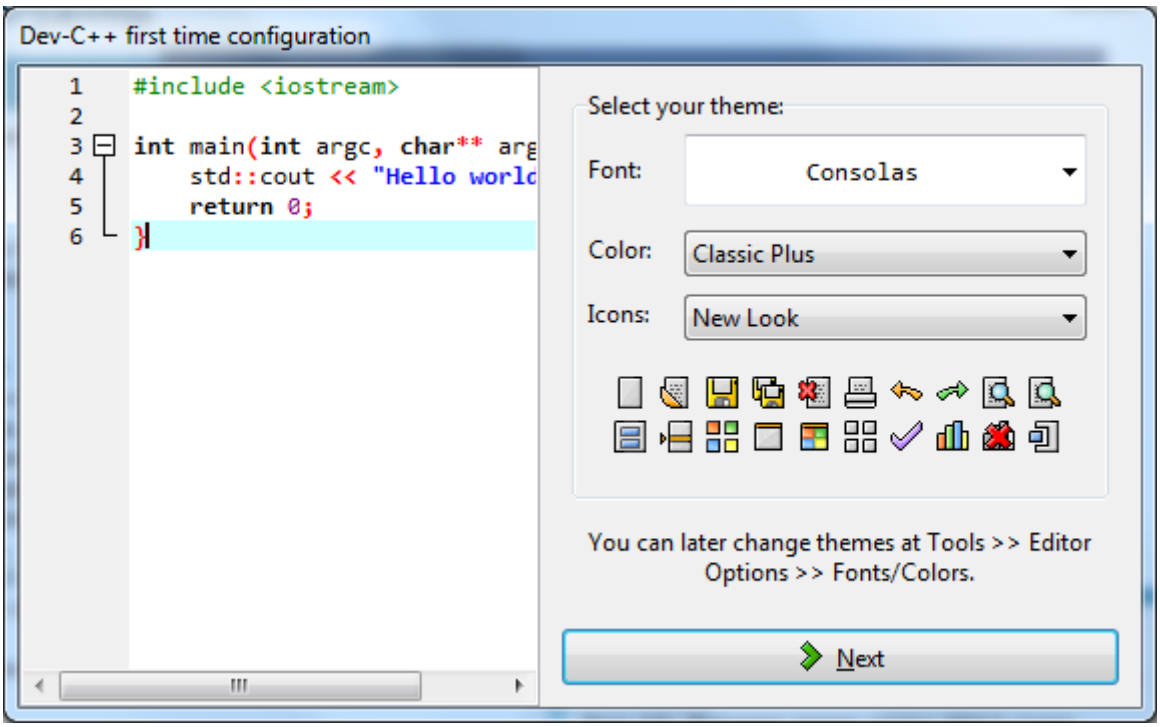


Рис. 2.7 – Выбор оформления

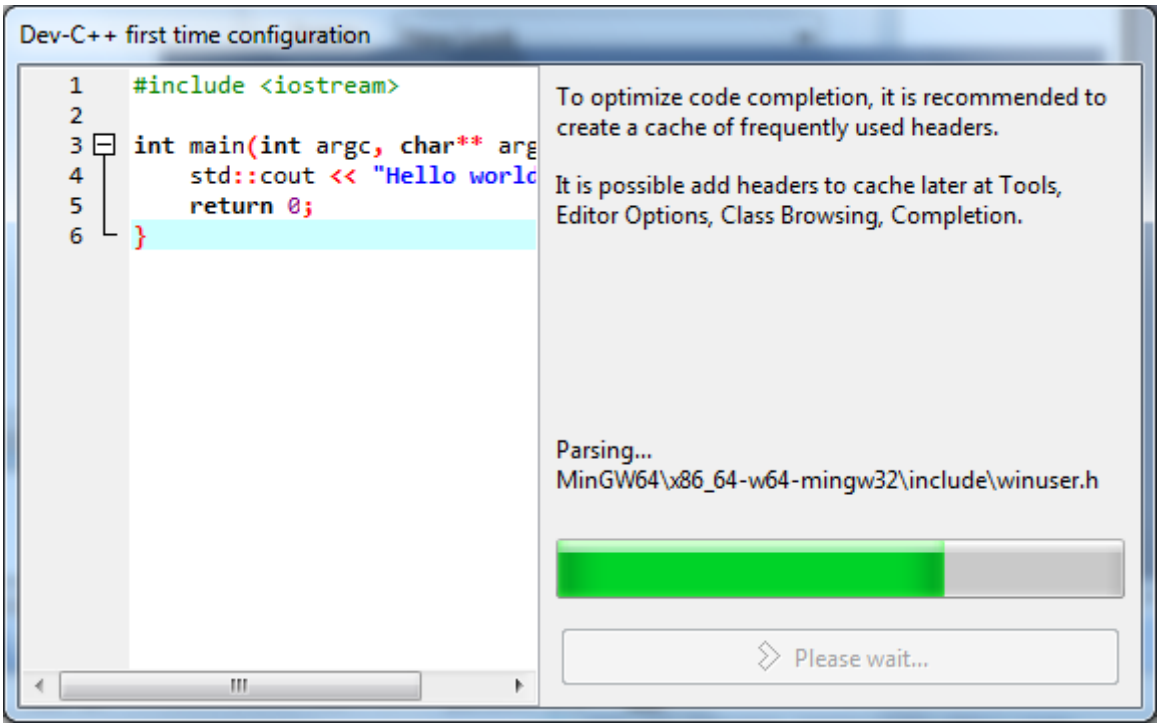


Рис. 2.8 – Завершающий этап установки

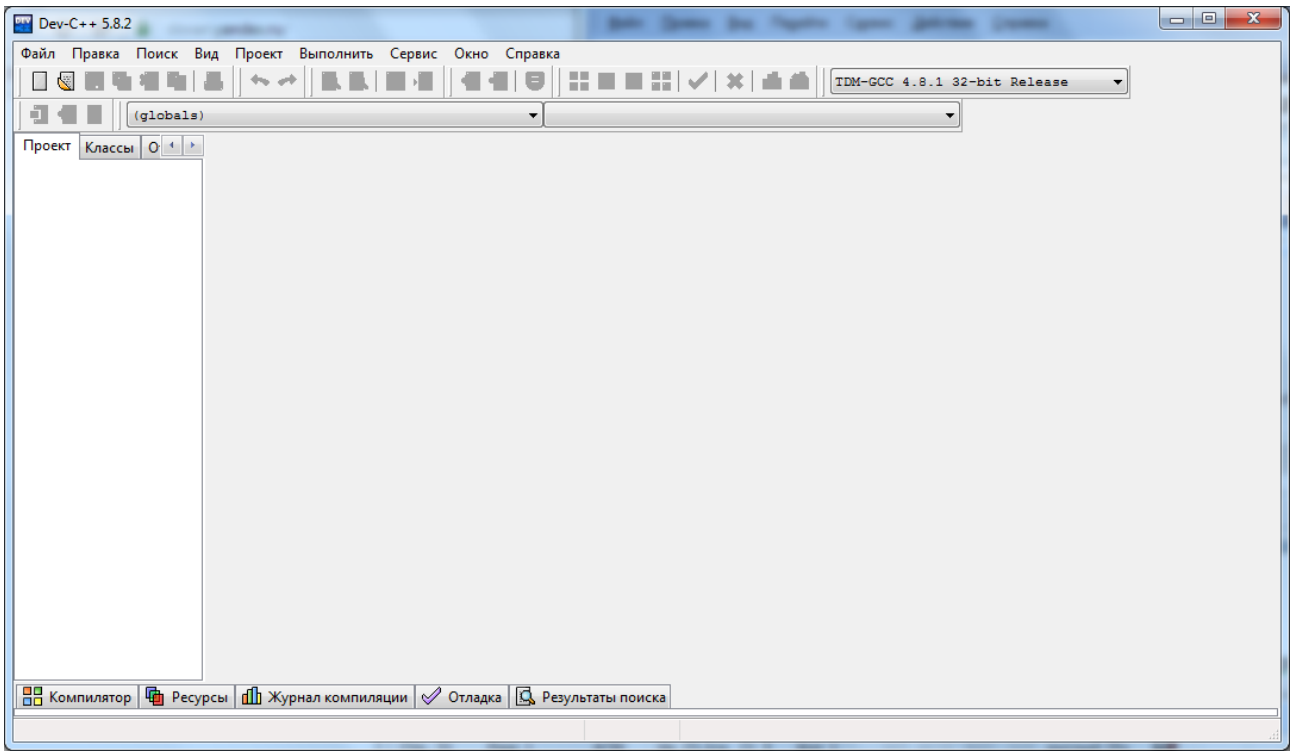


Рис. 2.9 – Вид окна приложения

2.2 Настройка параметров среды

После инсталляции среда полностью готова к работе, однако рассмотрим основные моменты настройки среды, такие как выбор режимов работы компилятора, определение путей доступа к библиотекам, определение параметров среды и редактора.

Меню *Сервис* среды позволяет изменить:

- параметры компилятора;
- параметры среды;
- параметры редактора.

Настройки компилятора

Окно изменения настроек компилятора позволяет выбрать набор настроек компилятора для соответствующей архитектуры компьютера из предложенного списка (рис. 2.10).

Во вкладке *Компилятор* Вы можете добавить команды в запуск компилятора и компоновщика программ.

Во вкладке *Настройки* Вы можете изменить настройки компилятора Си, настройки генерации кода, настройки вывода предупреждений, настройки компоновщика, настройки вывода.

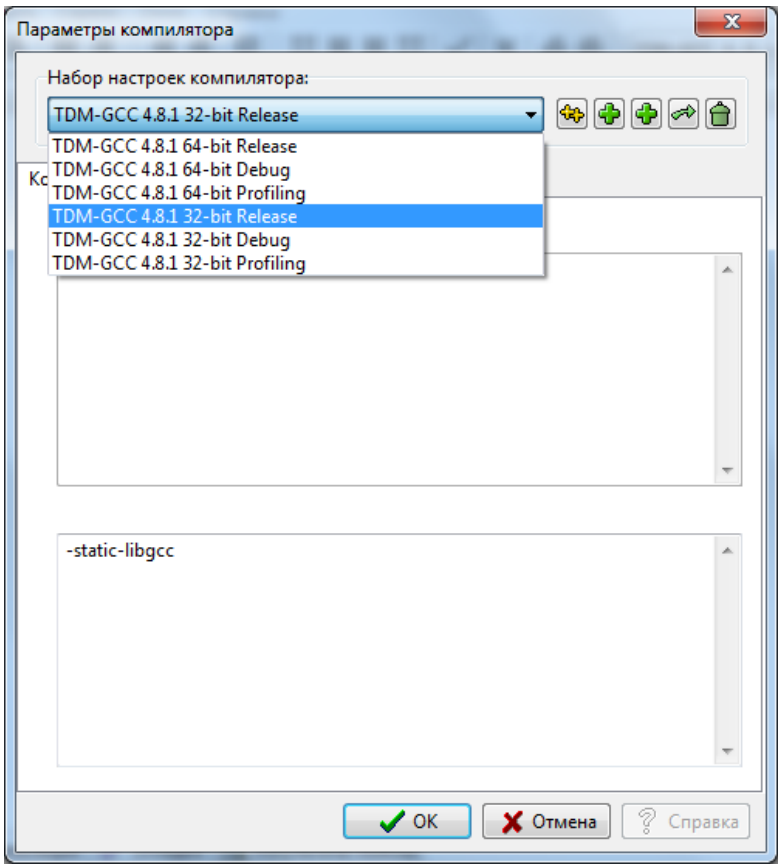


Рис. 2.10 – Изменение набора настроек компилятора

Во вкладке *Каталоги* Вы можете изменить пути доступа к программам, использующимся средой, библиотекам, включаемым файлам языка Си и включаемым файлам языка C++. По умолчанию установлены стандартные пути к указанным файлам.

Во вкладке *Программы* Вы можете изменить программу-компилятор, программу-компоновщик, программу-отладчик, которые использует интегрированная среда.

2.3 Создание проекта

Для написания программ в среде *DEV-CPP* требуется создать проект. Создать проект можно с помощью меню *Файл – Создать – Проект* либо с помощью иконки на панели инструментов  .

При создании проекта Вам будет предложено определить тип приложения (следует выбрать консольное приложение) и параметры проекта – имя проекта (произвольное имя) и тип проекта (C). Пример определений показан на рисунке 2.11.

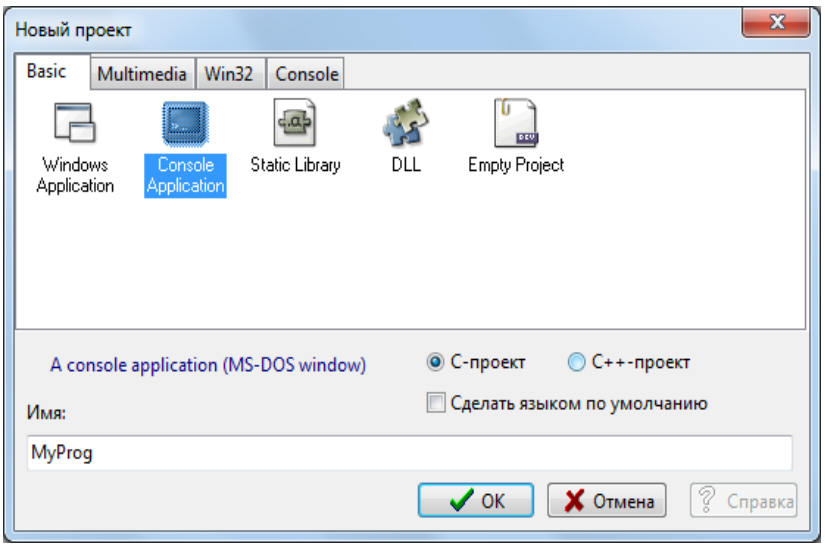


Рис. 2.11 – Определение параметров проекта

Рекомендуется называть проекты и папки, содержащие файлы проекта, не используя символы кириллицы.

После определения типа проекта среда предложит выбрать место расположения проекта. Окно выбора места расположения показано на рисунке 2.12.

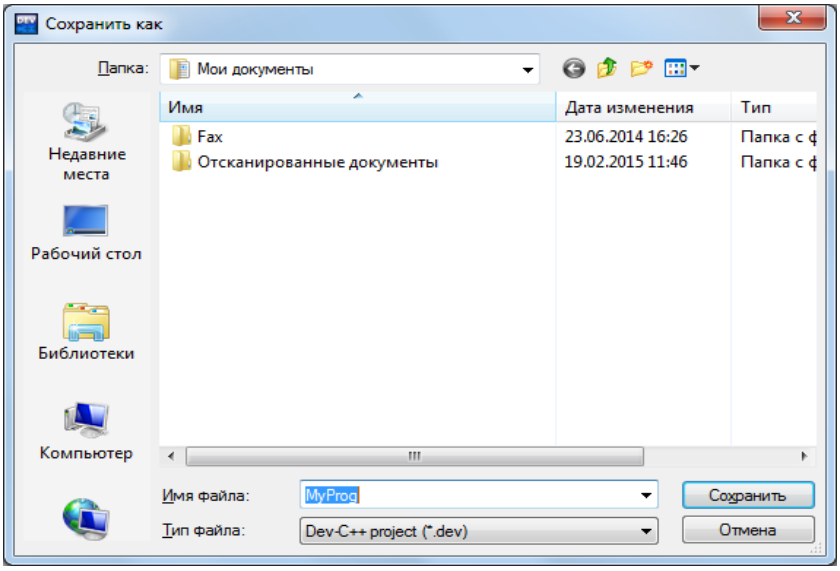


Рис. 2.12 – Определение места расположения проекта

Рекомендуется хранить проекты в отдельных папках.

После успешного создания проекта среда создаст шаблон программы, показанный на рисунке 2.13.

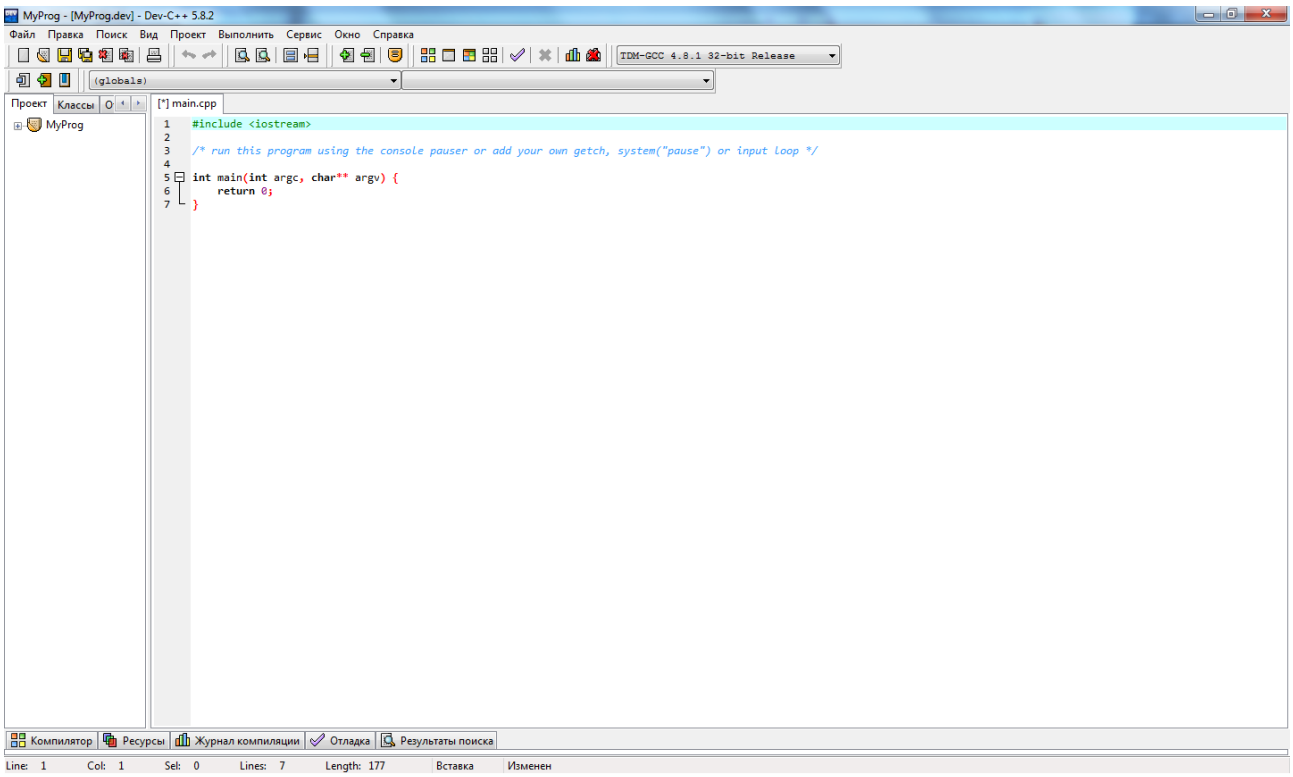


Рис. 2.13 – Шаблон программы

На основе предложенного шаблона напишем простую программу, выводящую на экран некоторую информацию:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
 system("chcp 1251");
 int num;
 num = 2;
 printf ("Моя первая программа\n");
 printf("Во %d-м семестре я изучаю язык Си\n",num);
 system("PAUSE");
 return 0;}
```

1. `#include <stdio.h>` – подключить заголовочный файл `stdio.h`.
2. `#include <stdlib.h>` – подключить заголовочный файл `stdlib.h`.
3. `int main(int argc, char **argv)` – имя функции. Любая программа на языке Си состоит из одной или нескольких функций. В приведенном примере функция одна. Функция с именем `main` обязательно должна быть в ***любой исполняемой программе***.

4. `/* простая программа */` – пример оформления комментария.
5. `{ }` – операторные скобки (`{` – начало функции; `}` – конец функции).
6. `system("chcp 1251");` – вызов функции `system` для смены кодировки страницы консоли; используется для корректного вывода символов кириллицы.
7. `int num;` – переменная `num` описана как переменная целого типа.
8. `;` – указывает, что в этой строке есть оператор языка Си (в отличие от других языков программирования является частью оператора, а не разделителем операторов).
9. `num = 2;` – оператор присваивания. Переменной `num` присвоено значение `2`.
10. `printf ("Моя первая программа\n");` – выводит на экран строку и переводит курсор в начало следующей строки, на что указывает `\n` – управляющий символ.
11. `printf("Во %d-м семестре я учу язык Си",num);` – на экран выведется строчка – «Во 2-м семестре я учу язык Си.» Спецификатор `%d` показывает, что в этом месте должно быть выведено целое число, после строки указывается имя переменной, значение которой должно быть выведено, в данном случае используется переменная `num`.
12. `system("PAUSE");` – вызов функции `system` для организации паузы; на экране появится надпись *Press any key to continue*….

2.4 Компиляция и выполнение

Для компиляции и последующего выполнения проекта среда предлагает следующие инструменты в меню *Выполнить*:

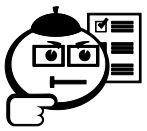
- перестроить проект  ;
- скомпилировать  ;
- скомпилировать и выполнить  ;
- выполнить  .

При первой компиляции проекта среда предложит сохранить файл с текстом программы в файле с именем `main.c`.

Переименуйте файл, например, дайте ему имя проекта.

После успешной компиляции в файле проекта создастся исполняемый файл `<имя проекта>.exe`. Помимо файла с текстом проекта и исполняемого файла в папке проекта будут находиться следующие файлы, созданные средой:

- `Makefile.win` – файл, содержащий инструкции для программы `make`;
- `<имя проекта>.dev`, `<имя проекта>.o` – информация о проекте.



Попробуйте набрать приведенную выше программу, сохранить текст программы, откомпилировать ее и отправить на выполнение.

2.5 Отладка программы

Самые первые ошибки, которые приходится исправлять программисту, – синтаксические ошибки – несоответствия с синтаксисом языка. Ниже приведен список наиболее часто встречающихся ошибок, примеры подобных ошибок и способы их исправления.

Как правило, большинство смысловых или алгоритмических ошибок в программе можно обнаружить, используя встроенный отладчик. Отладчик интегрированной среды позволяет выполнять программу по шагам; строка за строкой; выполнять программу до заданной строки, которую в дальнейшем будем называть точкой останова; просматривать текущие значения используемых переменных.

Для выполнения программы по шагам необходимо предварительно настроить отладчик, выполнив следующие шаги:

- Выберите меню *Сервис – Параметры компилятора* – вкладка *Настройки* – вкладка *Компоновщик*.
- Измените флаг напротив строки *Генерировать отладочную информацию* на *Yes* (рис. 2.14).

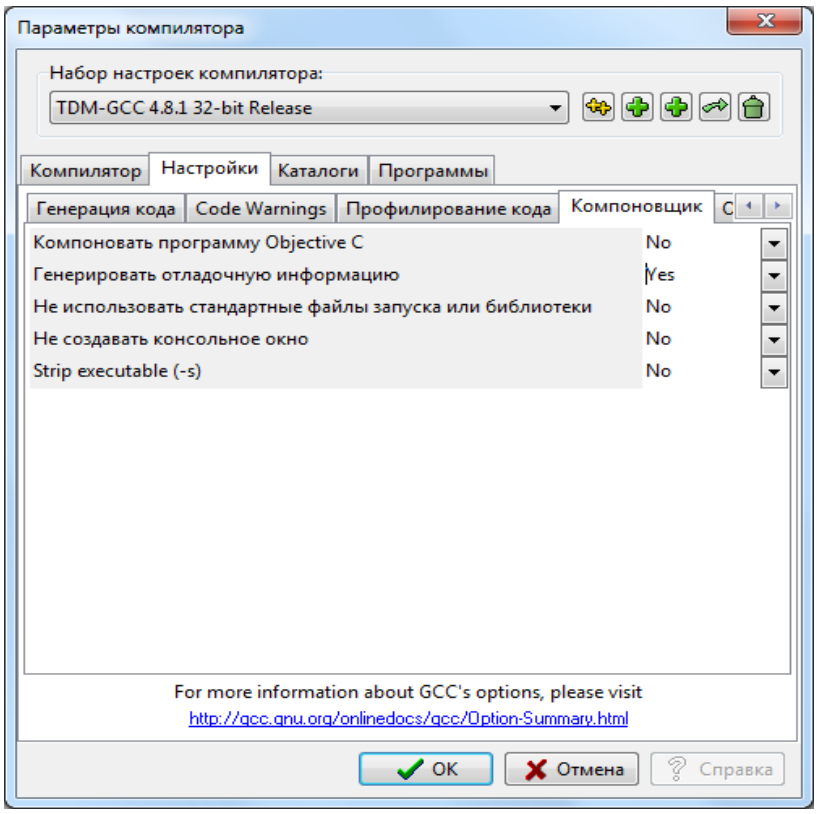


Рис. 2.14 – Изменение настроек компоновщика

- Выберите меню *Проект – Параметры проекта* – вкладка *Компилятор* – вкладка *Компоновщик.*
- Измените флаг напротив строки *Генерировать отладочную информацию* на *Yes* (рис. 2.15).

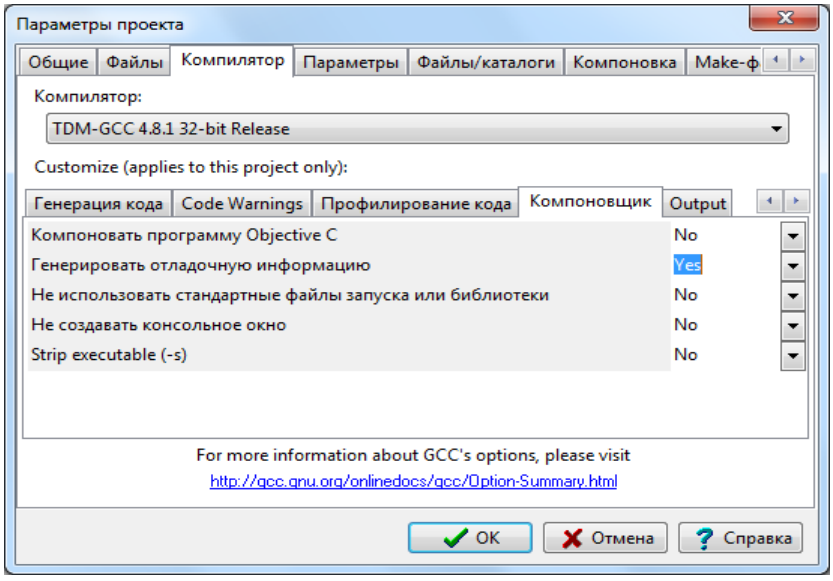


Рис. 2.15 – Изменение параметров проекта

Наведите курсор мыши на номер строки, в которой бы Вы хотели остановить программу (рис. 2.16), и, кликнув на выбранной строке, установите точку

останова. При отладке больших программ можно заранее выбрать одну или несколько точек останова. Удалить точки останова можно также при помощи мыши.

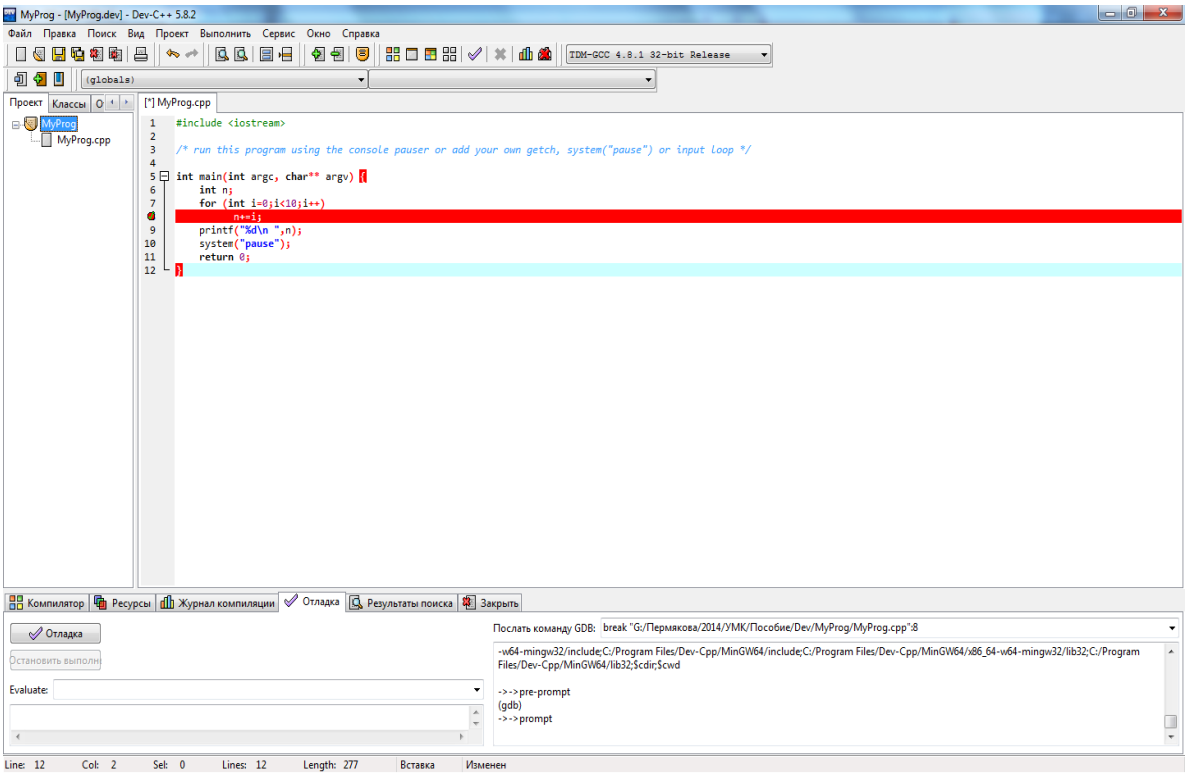


Рис. 2.16 – Определение точки останова программы

Для запуска процесса отладки используйте пункт меню *Выполнить – Отладка*, либо горячую клавишу <*F5*>, либо иконку панели инструментов  .

Отладчик будет прерывать выполнение программы в каждой отмеченной строке, отмечая строку, которая на данный момент является текущей, синим цветом (рис. 2.17).

Для просмотра значений переменных в процессе пошаговой отладки программы наведите курсор мыши на окно отладки. По нажатию правой кнопки мыши станет доступным меню, с помощью пунктов которого можно управлять просмотром переменных программы.

Выберите пункт *Добавить в наблюдаемые* и в появившемся диалоговом окне напишите имя переменной, значение которой Вы хотите просмотреть во время выполнения программы. На рисунке 2.18 при отладке программы просматриваются значения переменных `n` и `i`.

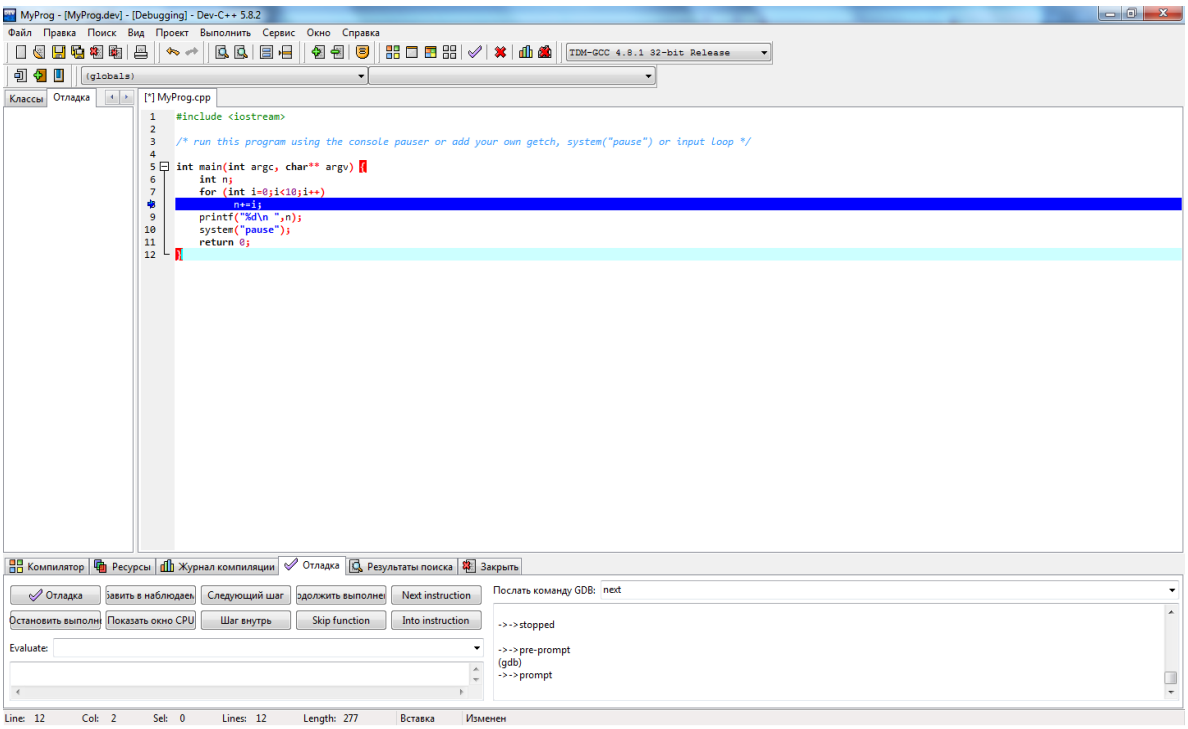


Рис. 2.17 – Процесс отладки

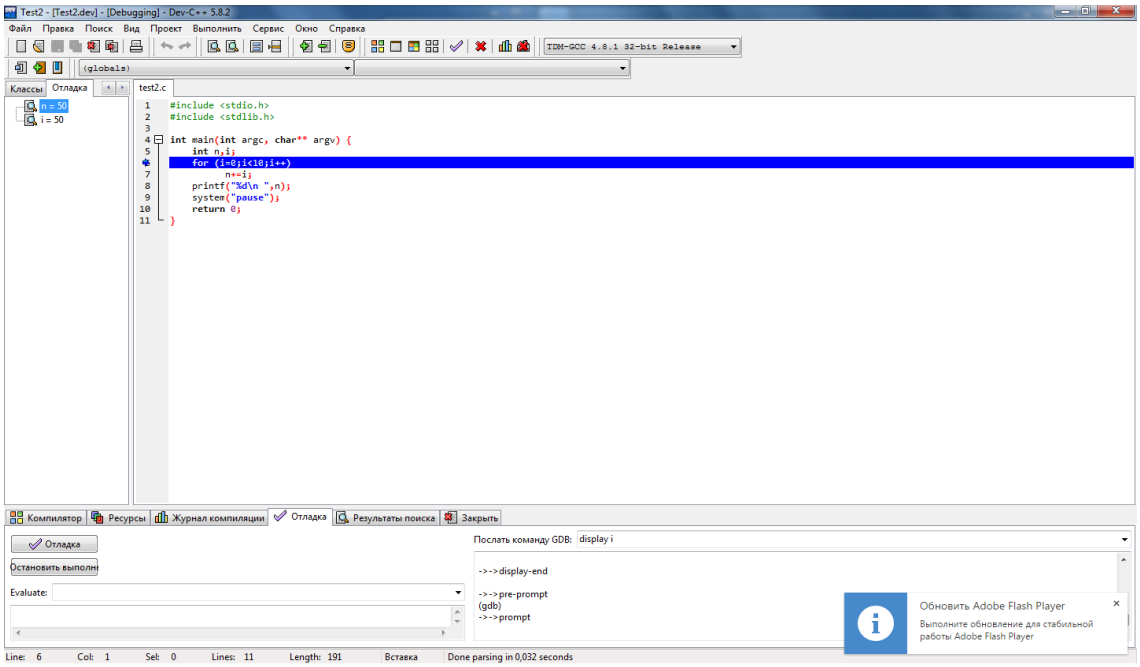
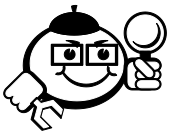


Рис. 2.18 – Наблюдаемые переменные

Рассмотрим несколько примеров отладки простейших программ.



Пример 2.1

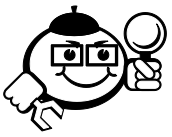
Пусть в программе необходимо найти сумму чисел от 0 до 9 и вывести ее на экран. При программировании этого задания была допущена ошибка – зна-

чение переменной n не было проинициализировано. Программа при выполнении дает неверный ответ, так как в переменной n при таком написании программы может храниться произвольное значение. Найти такую ошибку поможет отладчик. Выполним программу по шагам, просматривая в окне отладки значение переменной n. При первом выполнении цикла значение переменной не равно 0.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 int n,i;
 for (i=0;i<10;i++)
  n+=i;
 printf("%d\n ",n);
system("PAUSE");
return 0; }
```

Проинициализируем переменную n при описании – `int n = 0;`

В следующем примере ошибка допущена в теле цикла `while()`. При таком написании цикл становится бесконечным. Для поиска подобных ошибок можно использовать точку останова в теле цикла.



Пример 2.2

В окне отладки просматривается переменная, от которой зависит истинность/ложность условия цикла, в данном примере это переменная n. При пошаговом выполнении становится понятно, что переменная n никогда не примет значение, большее 20.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 int n=12;
 while (n<20)
 { n--; }
 printf("%d\n ",n);
 system("PAUSE");
 return 0; }
```

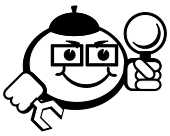
2.6 Многофайловая компиляция

При работе с большими программами наиболее удобен следующий подход:

- создаются один или несколько заголовочных файлов (файл с расширением `.h`), в которых хранятся описания (прототипы) используемых функций;
- файлы, содержащие тексты функций (способы разделения функций по отдельным файлам могут быть различными, например в одном файле разместить все функции, работающие с матрицами, в другом – все функции, отвечающие за вывод информации на экран и т. п.);
- файл с функцией `main()`.

Объединяет в одно целое выбранные файлы *проект*. Процедура создания проекта была описана ранее.

Для добавления файлов в проект используйте меню *Проект/Добавить в проект* или иконку на панели инструментов  .



Пример 2.3

Функции создания и печати матрицы сохранены в файле `func.cpp`, выполняемая функция сохранена в файле `matr.cpp`. На рисунке 2.19 показан вид окна проекта для совместной компиляции этих файлов.

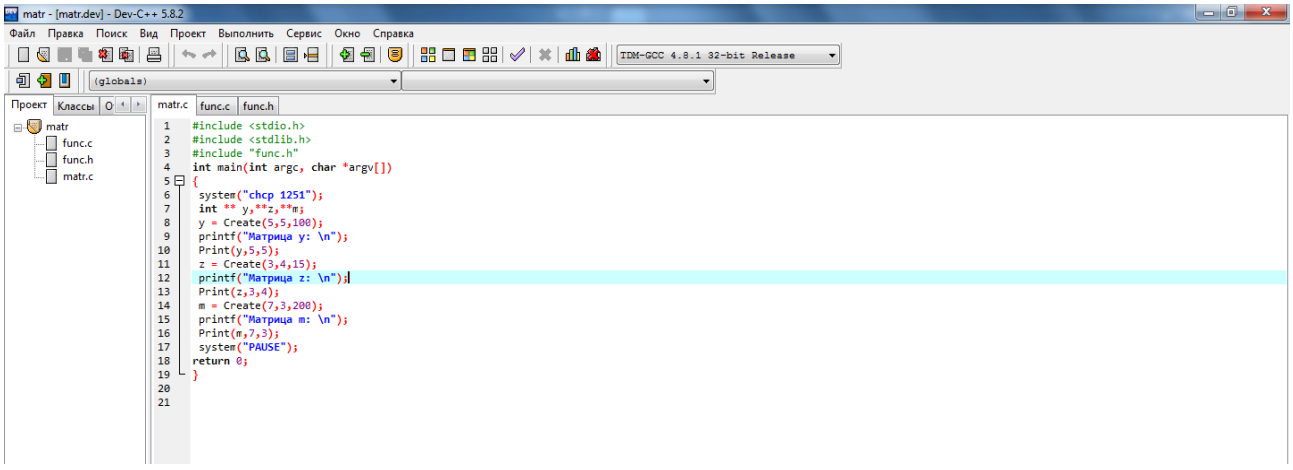


Рис. 2.19 – Пример проекта

```
// файл func.cpp
int ** Create(int n, int m,
int k)
```

```
void Print(int ** x, int n,
int m)
      {
```

```
      {
       int i,j;
       int** x =
(int**)malloc(sizeof( int*)*n);
       for (i=0;i<n;i++)
           x[i] =
(int*)malloc(sizeof(int)*m);
       for(i=0;i<n;i++)
           for ( j=0;j<m;j++)
            x[i][j] = rand()%k;
       return x; }
```

```
       int i,j;
       for(i=0;i<n;i++)
           {
           for (j=0;j<m;j++)
       printf("%4d",x[i][j]);
           printf("\n");
           }
      }
```

```
// файл func.h
int ** Create(int, int , int );
void Print(int ** , int , int );
```

```
// файл matr.cpp
#include <stdio.h>
#include <stdlib.h>
#include "func.h"
int main(int argc,
char *argv[])
{
int ** y,**z,**m;
y = Create(5,5,100);
printf("Матрица y:
\n");
 Print(y,5,5);
```

```
z = Create(3,4,15);
 printf("Матрица z: \n");
 Print(z,3,4);
 m = Create(7,3,200);
 printf("Матрица m: \n");
 Print(m,7,3);
 system("PAUSE");
return 0;
}
```

.....

Для компиляции отдельных файлов проекта используйте *Компилировать текущий файл*  , для компиляции всего проекта используйте *Перестроить проект*  .

2.7 Сообщения об ошибках

Компилятор GCC может генерировать четыре типа ошибок:

- ошибки фазы препроцессорной обработки;
- ошибки фазы компиляции;
- ошибки фазы построения связей;

- ошибки фазы выполнения.

Полную информацию об ошибках можно получить, изучая документацию, сопровождающую компилятор [6].

Рассмотрим наиболее часто встречающиеся ошибки фазы компиляции и исполнения.

Сообщения об ошибках компилятора

Диагностические сообщения компилятора *GCC* можно разделить на два класса:

- ошибки;
- предупреждения.

Ошибки указывают на синтаксические ошибки в программе, ошибки доступа к диску или памяти и ошибки командной строки.

В процессе работы компилятор пытается найти как можно больше действительных ошибок в исходной программе во время текущей фазы компиляции (препроцессорная обработка, лексический разбор, оптимизация и генерация кода).

Предупреждения не мешают окончанию компиляции. Они указывают на условия, которые могут быть подозрительными, но лексически правильны для языка.

В окне сообщений печатается класс сообщения, затем имя исходного файла и номер строки, в которой компилятор обнаружил ошибку или подозрительное условие, далее сам текст сообщения или ошибки.

Компилятор генерирует сообщения по мере их обнаружения. Поскольку язык не устанавливает ограничений на размещение операторов в строке, действительная причина ошибки может находиться на одну или более строк выше указываемой строки.

Ошибки

`'имя переменной' undeclared (first use in this function)` – отсутствует описание используемой переменной или функции.

`parse error before '...' syntax error` – компилятор встретил последовательность символов, которые не отвечают условиям синтаксиса языка. Сообщение может быть вызвано отсутствием закрывающей скобки,

скобки или точки с запятой, предшествующих указанной строке, либо использованем недопустимого ключевого слова.

`parse error at end of input` – компилятор неожиданно обнаружил конец файла, например, при анализе обнаружено несбалансированное количество открывающих и закрывающих скобок.

`unterminated string or character constant` – используется строка (символ), которая не имеет соответствующую закрывающую кавычку. Все символьные константы должны быть заключены в парные кавычки `("пример строки", 'm')`.

`character constant too long` – в одинарные кавычки заключен более чем один символ. Строки должны быть заключены в двойные кавычки.

`dereferencing pointer to incomplete type` – программа пытается получить доступ к элементам структуры через указатель без предварительного описания структуры.

`initializer element is not a constant` – глобальные переменные могут быть инициализированы только константами (числовыми значениями, нулем или фиксированными строками).

Предупреждения

`warning: implicit declaration of function '...'` – функция используется без прототипа. Возможно, не включен заголовочный файл, содержащий описание функции, или не представлен прототип функции.

`warning: initialization makes integer from pointer without a cast` – предупреждение о неправомерном использовании указателя как целого числа в контексте. Чаще всего это предупреждение является результатом использования указателя без разыменования (например, написание `int z = p` вместо `int z = *p`).

`warning: unknown escape sequence '...'` – неправильное использование символа в *Escape*-последовательности.

`warning: suggest parentheses around assignment used as truth value` – это предупреждение выделяет серьезную ошибку, возможно, использован оператор присваивания `'='` вместо оператора сравнения `'=='` в условном операторе или иных логических выражениях.

`warning: control reaches end of non-void function` – в функции, возвращающей значение, отсутствует оператор возврата значения `return`.

`warning: unused variable '...', warning: unused parameter '...'` – эти предупреждения указывают на то, что объявленная локальная переменная или параметр функции не были использованы.

`warning: passing arg of ... as ... due to prototype` – это предупреждение возникает, когда функция вызывается с параметром `Тип`, отличным от типа, указанного в описании функции.

`warning: assignment of read-only location, warning: cast discards qualifiers from pointer target type, warning: assignment discards qualifiers ..., warning: initialization discards qualifiers ..., warning: return discards qualifiers ...` – такие предупреждения компилятор генерирует в том случае, когда указатель используется неверно.

Ошибки времени выполнения

Такие ошибки возникают после того, как программа была успешно откомпилирована и выполняется.

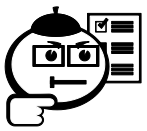
`Error while loading shared libraries:, cannot open shared object file: No such file or directory` – программа использует разделяемые библиотеки, но необходимые файлы общей библиотеки не удается найти.

`Segmentation fault, Bus error` – произошла ошибка доступа к памяти. Возможны следующие общие причины:

- разыменование нулевого указателя или неинициализированный указатель;
- обращение к несуществующим элементам массива;
- неправильное использование функций `malloc`, `free` и функций, связанных с выделением памяти;
- использование в функции `scanf` недопустимых аргументов.

`Floating point exception` – при выполнении программы произошло арифметическое исключение (деление на ноль, переполнение, потеря значимости), или выполнение недопустимой операции (например, извлечения квадратного корня из –1).

`Illegal instruction` – операционная система при выполнении программы встретила незнакомую инструкцию. Эта ошибка может произойти при выполнении кода, скомпилированного для другой архитектуры.



1. Создайте новый файл в интегрированной среде.
2. Наберите любой пример, рассмотренный в данном пособии.
3. Попробуйте получить контекстную справку по любой функции, используемой в программе.
4. Сохраните набранный пример.
5. Выполните компиляцию.
6. Отправьте программу на выполнение.
7. Попробуйте пройти набранный пример, используя трассировку по шагам.
8. Закончите работу в редакторе.

3 Синтаксис и алфавит языка Си

3.1 Алфавит языка Си

Для образования лексических частей языка (лексем) и связей между ними используются все символы латинского алфавита, цифры, специальные знаки ! @ % $ & * ( ) - + \/ | {} [ ] . ,_ ~ " ` #.

3.2 Синтаксис

3.2.1 Лексемы языка

Язык Си распознает шесть типов лексем: ключевые слова, идентификаторы, константы, литеральные строки, операторы, разделители (знаки пунктуации). Лексемы выделяются на фазе лексического анализа компиляции. Исходный код программы разбивается на лексемы и разделители. Разделителями считаются пробелы, горизонтальная и вертикальная табуляция, символы начала новой строки и комментариев. Разделители указывают, где начинаются и кончаются лексемы, но кроме этой функции любое присутствие разделителей игнорируется, например, последовательности

```
int z,k; float b;
```

и

```
int z,k;
float b;
```

лексически эквивалентны. При анализе дают 8 лексем:

1. `int`
2. `z`
3. `,`
4. `k`
5. `;`
6. `float`
7. `b`
8. `;`

3.2.2 Ключевые слова

Ключевые слова *– это слова, имеющие специальное назначение, их нельзя использовать как имена идентификаторов.*

В таблице 3.1 приведен список ключевых слов Си.

Таблица 3.1 – Ключевые слова

asm	_ds	interrupt	short
auto	else	_loadds	signed
break	enum	long	sizeof
case	_es	_near	_ss
catch	_export	near	static
_cdecl	extern	new	struct
cdecl	_far	operator	switch
char	far	_pascal	template
class	float	pascal	this
const	for	private	typedef
continue	friend	protected	union
_cs	goto	public	unsigned
default	huge	register	virtual
delete	if	return	void
do	inline	_saveregs	volatile
double	int	_seg	while

3.2.3 Идентификаторы

Идентификаторы *– это произвольные имена любой длины для классов, объектов, функций, переменных, типов данных, определенных пользователем и т. д.*

Идентификаторы могут содержать буквы от `A` до `Z` и от `a` до `z`, символы `_`, `$` и цифры от `0` до `9`. Существует только одно ограничение:

- Первый символ должен быть буквой, или `_`, или `$`.

Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на нее ограничения.

Си различает строчные и прописные буквы в идентификаторах так, что `Sum`, `sum` и `suM` – это разные идентификаторы.

Недопустимо использовать один и тот же идентификатор в одной сфере действия два и/или более раз.

3.2.4 Константы

Лексемы «константа» представляют числовые или символьные значения. Си поддерживает 4 класса констант:

- с плавающей точкой;

- целые;
- перечислимые;
- символьные.

Целые константы могут быть десятичными (основание 10), восьмеричными (основание 8) или шестнадцатиричными (основание 16).

Десятичные константы могут принимать значения от 0 до 4 294 967 295. Константы, превышающие этот предел, будут приводить к ошибке.

При записи восьмеричной константы записывается лидирующий 0. Если восьмеричная константа содержит цифры 8 или 9, генерируется ошибка. Восьмеричные константы, превышающие 037777777777, приводят к ошибке.

Все константы, начинающиеся с 0x (0X) интерпретируются как шестнадцатеричные. Шестнадцатеричные константы, превышающие 0xFFFFFFFF, приводят к ошибке.

К константам возможно добавлять специальные завершающие символы – суффиксы. Суффикс `L` (или `l`), присоединенный к любой константе, заставляет интерпретировать ее как `long`. Аналогично, суффикс `U` или `u` заставляет интерпретировать константу как `unsigned`. Можно использовать оба суффикса `U` и `L` в одной константе и в любом порядке: `lu`, `ul`, `UL` и т. д.

Приведем пример констант:

Константы в инициализации переменных:

```
int z = 10 /* десятичное 10 */
int x = 010 /* десятичное 8 */
int p = 0XF /* десятичное 15 */
```

Константы в выражениях:

```
long m = 2*100L /* умножение на десятичную константу типа long int */
unsigned n = 2*100L /* умножение на десятичную константу типа unsigned int */
```

Символьная константа – это один или более символов, заключенные в апострофы, как например `'A', '=', '\n'`.

Одиночная символьная константа имеет тип данных `char`, константа из нескольких символов имеет тип данных `int`.

Символ \ используется для обозначения *ESC*-последовательности, позволяя визуально отобразить некоторые графические символы. Например, константа `\n` используется для символа `"новая строка"`. Символ \ используется с восьмеричными или шестнадцатеричными числами для представления

ASCII символа или управляющего кода, соответствующего этому значению; например, `'\03'` для *Ctrl-C*.

Таблица 3.2 показывает допустимые *ESC*-последовательности.

Таблица 3.2 – *ESC*-последовательности в Си

Последо-ватель-ность	**Числовое обозна-чение**	**Символь-ное обо-значение**	**Что выполняет**
`\a`	`0x07`	`BEL`	Сигнал
`\b`	`0x08`	`BS`	Удаление предыдущего символа
`\f`	`0x0C`	`FF`	Перевод страницы
`\n`	`0x0A`	`LF`	Новая строка (пустая строка)
`\r`	`0x0D`	`CR`	Возврат каретки
`\t`	`0x09`	`HT`	Табуляция (горизонтальная)
`\v`	`0x0B`	`VT`	Табуляция (вертикальная)
`\\`	`0x5c`	`\`	Обратный \
`\'`	`0x27`	`'`	Одиночная кавычка (апостроф)
`\"`	`0x22`	`"`	Двойная кавычка
`\?`	`0x3F`	`?`	Знак вопроса
`\`*O*	Любое	O	Строка до трех восьмеричных цифр
`\x`*H*	Любое	H	Строка шестнадцатеричных цифр
`\X`*H*	Любое	H	Строка шестнадцатеричных цифр

Константа с плавающей точкой состоит из 6 частей:

- десятичная целая;
- десятичная точка;
- десятичная дробная;
- `e` или `E` и знаковая целая экспонента (необязательная);
- суффикс типа `f`, `F`, `l`, `L` (необязательный).

Примеры вещественных констант:

`3.25e4`	$3.25 \cdot 10^4$
`.05`	`0.05`
`1.`	`1.0`
`-1.25`	`1.25`
`3e-8`	$3.0 \cdot 10^{-8}$
`2.5E+6`	$2.5 \cdot 10^6$

3.2.5 Литеральные строки

Литеральные строки, или строковые константы, используются для обработки фиксированных последовательностей символов. Литеральная строка – это массив символов, записанный как последовательность любого числа символов внутри кавычек: `"это пример литеральной строки"`.

Символы внутри кавычек могут включать *ESC*-последовательности. Так, строка `"\n\"Hello !\"\n"` выведется на экран следующим образом:

```
"Hello!"
```

Перед `"Hello!"` стоит знак пустой строки, далее `\"` выводит кавычки. После строки вновь выводятся кавычки. Вывод завершается символом пустой строки.

Литеральная строка хранится как заданная последовательность символов и завершается нулевым символом (`'\0'`). Нулевая (пустая) строка хранится как символ `'\0'`. Пустая строка обозначается `""`.

3.2.6 Операторы

Оператор *– это лексема, которая переключает некоторые вычисления, когда применяется к переменной или к другому объекту в выражении.*

Язык Си представляет большой набор операторов не только арифметических и логических, но и операторов для побитовых действий, доступа к компонентам структур и объединений и операций с указателями. Операторы языка Си делятся на унарные, бинарные и существует единственный тернарный оператор. В таблицах 3.3–3.4 представлены операторы языка Си.

Отдельно рассмотрим условный оператор `A ? X : Y`, который является тернарным оператором. Если истинно отношение `A`, то выполняются действия `X`; иначе выполняются действия `Y`. Например, условный оператор может быть записан следующим образом:

`z = (x<y)?x+15:y-25;` – если `x` меньше, чем `y`, то переменной `z` присвоить значение `x+15`, в противном случае переменной `z` присвоить значение `y-25`.

Таблица 3.3 – Унарные операторы языка Си

Код оператора	Название	Результат операции
`&`	Адресный оператор	Выражение `&x` – адрес переменной `x`
`*`	Оператор косвенной адресации	`*x` – значение, расположенное по адресу `x`
`+`	Унарный плюс	`+5` – положительная константа
`-`	Унарный минус	`-4` – отрицательная константа, `-x` – значение переменной `x` с обратным знаком
`~`	Побитовое отрицание	`~x` – побитовое отрицание переменной `x`
`!`	Логическое отрицание	`!x` принимает значение `0` (лжи), если `x` имеет ненулевое (истинное) значение и наоборот
`++`	Префиксное/ постфиксное увеличение	`int x=5; ++x;` увеличит `x` на единицу; `int x=5; x++;` увеличит `x` на единицу
`--`	Префиксное/ постфиксное уменьшение	`int x=5; --x;` уменьшит `x` на единицу; `int x=5; x--;` уменьшит `x` на единицу

Таблица 3.4 – Бинарные операторы языка Си

Код оператора	Название	Результат операции
Аддитивные операторы		

`+`	Бинарный плюс	Вычисление суммы, например:
		`int x=2, y=1, z;`
		`z=x+y;`

`-`	Бинарный минус	Вычисление разности, например:
		`int x=2, y=1, z;`
		`z=x-y;`

Мультипликативные операторы		

`*`	Умножение	Вычисление произведения, например:
		`int x=2, y=1, z;`
		`z=x*y;`

`/`	Деление	Вычисление частного, например:
		`int x=12, y=2, z;`
		`z=x/y;`

Код оператора	Название	Результат операции

%	Остаток	Вычисление остатка от деления, например:
		`int x=12, y=7, z;`
		`z=x%y;`

Операторы сдвига		

<<	Сдвиг влево	Вычисление побитового сдвига влево
		`int x=12, y=2, z;`
		`z=x<<y;`

>>	Сдвиг вправо	Вычисление побитового сдвига вправо
		`int x=12, y=2, z;`
		`z=x>>y;`

&	Побитовое *AND* (И)	Вычисление конъюнкции
		`int x=12, y=2`
		`z=x&y;`

^	Побитовое *XOR* (исключающее ИЛИ)	Вычисление сложения по модулю 2
		`int x=12, y=2`
		`z=x^y;`

|	Побитовое *OR* (ИЛИ)	Вычисление дизъюнкции
		`int x=12, y=2`
		`z=x|y;`

Логические операторы		
&&	Логическое *AND* (И)	Проверка условий, связанных логическим И
||	Логическое *OR* (ИЛИ)	Проверка условий, связанных логическим ИЛИ
Операторы присваивания		
=	Присваивание	Присвоить переменной заданное значение или значение другой переменной
*=	Присвоить произведение	Выражение `x*=5` эквивалентно выражению `x=x*5`
/=	Присвоить частное	Выражение `x/=5` эквивалентно выражению `x=x/5`
%=	Присвоить остаток	Выражение `x%=5` эквивалентно выражению `x=x%5`
-=	Присвоить разность	Выражение `x-=5` эквивалентно выражению `x=x-5`
<<=	Присвоить левый сдвиг	Выражение `x<<=5` эквивалентно выражению `x=x<<5`
>>=	Присвоить правый сдвиг	Выражение `x>>=5` эквивалентно выражению `x=x>>5`

Код оператора	Название	Результат операции
`&=`	Присвоить побитовое AND	Выражение `x&=5` эквивалентно выражению `x=x&5`
`^=`	Присвоить побитовое XOR	Выражение `x^=5` эквивалентно выражению `x=x^5`
`|=`	Присвоить побитовое OR	Выражение `x|=5` эквивалентно выражению `x=x|5`
`+=`	Присвоить сумму	Выражение `x+=5` эквивалентно выражению `x=x+5`
Оператор отношения – выражение, использующее операторы отношения эквивалентности, в результате работы принимает значение `true`, если отношение истинно, если же отношение ложное, выражение принимает значение `false`.		
`<`	Меньше чем	`x<y`, x меньше y
`>`	Больше чем	`x>y`, x больше y
`<=`	Меньше чем или равно	`x<=y`, x меньше или равно y
`>=`	Больше чем или равно	`x>=y`, x больше или равно y
Операторы эквивалентности		
`= =`	Равно	`x==y`, x равно y
`!=`	Не равно	`x!=y`, x не равно y
Операторы выбора компонент		
`.`	Прямой селектор компоненты	`x.k` – компонента `k` переменной `x` (применяется при работе со структурами)
`->`	Непрямой селектор компоненты	`x->k` – компонента `k` указателя `x` (применяется при работе с указателями на структуры)
`,`	Оператор перечисления	Выполнить разделенные оператором действия слева направо, например `y+=5, x-=4, y+=x;`

3.2.7 Знаки пунктуации

В языке Си определены следующие знаки пунктуации: `[ ] ( ) { } , ; : ... * = #`.

Квадратные скобки указывают список индексов одномерного или многомерного массива:

```
char word[] = "Пример строки"; /* строка символов.*/
float mat[3][4]; /* матрица вещественных символов, имеющая три строки и четыре столбца. */
```

```
int x[3]; /* целочисленный массив из трех элементов. */
```

Круглые скобки выделяют групповое выражение, условное выражение и указывают на вызов функции и параметры функции:

```
d = (a+b)*x; /* указывают на порядок действий */
if (x= =z) x+=z; /* используются в условных выражениях */
matrix(); /* вызов функции matrix без аргументов */
int change(int x,int y); /*объявление функции с аргументами*/
```

Круглые скобки также используются для изменения обычного порядка вычисления операторов.

Фигурные скобки указывают на начало и конец составного оператора:

```
for(int i =0; i<10;i++)
     {
     x ++;
     y--;
     }
```

Закрывающая фигурная скобка используется как конец составного оператора, после нее не требуется разделитель «;» за исключением объявления структуры.

Запятая разделяет элементы списка аргументов функции –

```
void func(int n, float f, char ch);
```

Также запятая может использоваться и для перечисления действий, пример использования фигурных скобок может быть записан следующим образом:

```
for(int i =0; i<10;i++)
x ++, y--;
```

Точка с запятой указывает на конец оператора. Любое правильное выражение (включая пустое выражение) должно заканчиваться знаком «;».

Двоеточие «:» указывает помеченный оператор.

```
   start:   x=0;
   ...
   goto start;   ...     // пример использования метки
   switch (a)
{// пример использования множественного выбора
    case 1: puts("One");
        break;
    case 2: puts("Two");
        break;
   ...
   default: puts("None of the above!");
        break;
}
```

Знак «*» в объявлении переменной указывает на создание указателя на тип:

```
char * str; /* указатель на символ */
```

Указатель с несколькими уровнями ссылок может быть объявлен заданием соответствующего числа символа «*»:

```
int ** x;   /* указатель на указатель на int */
```

Если знак «#» встречается как первый символ, он указывает на директиву препроцессора. Знаки «#» и «##» также используются как операторы для замещения и объединения лексем во время фазы препроцессора.

Контрольные вопросы по главе 3

1. На какой фазе компиляции выделяются лексемы?
2. К какому типу лексем относится зарезервированное слово `auto`?
3. Сколько типов операторов (по количеству операндов) можно выделить в языке Си? Перечислите эти типы.
4. Чему будет равно значение переменной `y` после выполнения следующего фрагмента программы?

```
int k = 12;
int z = 5;
int y=(k<z)25;z+10;
```

5. Сколько лексем содержит фрагмент программы?

```
int k = 12;
int z = 5;
int z = z+k;
```

6. Сколько разделителей содержит фрагмент программы?

```
int x, y, z;
float k = 0;
x = k+12;
```

7. Выберите верно составленные идентификаторы:
   a. `asm`
   b. `_3t`
   c. `2_i`
   d. `ii`
   e. `Пj`

8. Чему будет равно значение переменной `x` после выполнения следующего фрагмента программы?

```
int x=12;
x+=17;
```

9. Могут ли литеральные строки содержать *ESC*-последовательности?
10. Что будет выведено на экран при выводе следующей литеральной строки: «`/n/n // Вопрос 10 // /n/n`»?
11. Выберите верно записанные целые константы:

    1) `081`;
    2) `045`;
    3) `0xAA`;
    4) `AA`;
    5) `15`;
    6) `125`.

12. Выберите верно записанные вещественные константы:

    1) `0.0175`;
    2) `.123`;
    3) `-.123`;
    4) `15e-12`;
    5) `0.23e5`;
    6) `11e+03`.

13. В каких случаях используется разделитель «`,`»?
14. Для чего используется разделитель «`;`»?
15. Ограничена ли длина идентификатора?

4 Типы данных языка Си

4.1 Основные типы данных

4.1.1 Простые типы

Программа, написанная на языке программирования Си, оперирует данными (переменными) различных типов. Все данные имеют тип и имя (идентификатор).

Тип данных – это описание диапазона значений, которые может принимать переменная указанного типа. Каждый тип данных характеризуется своим размером (количеством байт, необходимых для хранения переменной указанного типа) и диапазоном значений, которые может принимать переменная данного типа.

Все типы данных языка Си можно разделить на простые (скалярные) и сложные (векторные) типы и базовые (системные) и пользовательские (которые определил пользователь).

В языке Си систему базовых типов образуют четыре типа данных:

Тип `char`. Занимает в памяти 1 байт. Используется для представления символов и целых чисел от –128 до 127.

Тип `int`**.** Занимает в памяти 4 байта. Используется для представления целых чисел в диапазоне от –2 147 483 648 до 2 147 483 647.

Тип `float`. Занимает в памяти 4 байта. Используется для представления чисел с плавающей точкой. Диапазон значений типа ±3.4*E*±38. При этом 23 бита занимает мантисса числа, 8 бит – порядок, 1 бит знак. Точность вычислений до 7 знаков после запятой. Тип `float` называют вещественным типом одинарной точности.

Тип `double`**.** Занимает в памяти 8 байт. Используется для представления чисел с плавающей точкой в диапазоне ±1.7E±308. Точность вычислений до 15 знаков после запятой. Тип `double` называют вещественным типом двойной точности.

В стандарте языка Си C99 добавлен логический тип данных `_Bool`, переменные этого типа могут принимать значения `true`, `false`. До введения этого стандарта логическим типом считался тип `int`, ложью при этом считалось значение 0, а истиной любое ненулевое значение.

Также в языке Си объявлен специальный тип `void` (пустой). Используется этот тип для описания функций.

4.1.2 Приставки к типам данных

В языке СИ предусмотрены модификаторы типов данных двух видов: модификаторы знака: `signed` и `unsigned` и модификаторы размера: `short` и `long`.

С помощью этих модификаторов-приставок возможно формирование новых типов.

Приставка `signed` осуществляет преобразование целочисленных значений к знаковым числам. Используется с типами `int` и `char` по умолчанию.

Приставка `unsigned` преобразует целочисленные значения (`int`, `char`) к беззнаковым числам. Например, применение приставки к типу `char` изменяет диапазон значений с –128…127 на диапазон 0…255.

Понятие «без знака» говорит о том, что в машинном представлении числа знаковый бит используется как и все другие биты для хранения числа. То есть переменная типа `int` занимает в памяти 32 бита, из которых один бит используется для хранения знака, переменная типа `unsigned int` занимает в памяти 32 бита и все они используются для хранения числа.

Приставка `short` используется только с типом `int` и уменьшает размер типа в два раза. Диапазон значений переменных типа `short int` от –32 768 до 32 767.

Приставка `long` используется совместно с типами `int` и `double`. При этом размер переменных типа `int` остается прежним. А размер переменных типа `double` становится равным 12 байт (по стандарту *IEEE* – 10 байт).

4.1.3 Преобразование типов

В языке Си существует понятие неявного преобразования типов. Гибкий язык программирования позволяет делить целое число, присваивать целому числу вещественное значение без применения специальных функций. Например, при выполнении следующего фрагмента программы будут получены следующие значения переменных:

```
int x = 5,y = 2,z;
z=x/y;
x = 2.25;
```

Переменная z примет значение 2. При делении полученное вещественное число будет преобразовано к типу int посредством отбрасывания дробной части. Переменная x примет значение 2, т. е. будет использоваться тот же самый механизм преобразования.

В языке установлен следующий приоритет типов: double, float, long (все, что идет с приставками long), int, short (все, что идет с приставками short), char. При программировании на языке Си необходимо знать и помнить три основных правила неявного преобразования типов:

- Если два операнда выполняемой операции имеют тип A, а результат имеет тип B, то результат в процессе выполнения операции будет приведен к типу A.

```
float z;
z = 1/25; // переменная z будет равна 0
```

- Если два операнда одной операции имеют тип A и B, а результат имеет тип B, то результат будет приведен к типу B.

```
int z;
z = 42/2.5; // переменная z примет значение 16;
…
float z;
z = 42/2.5; // переменная z примет 16.7666;
```

- Если операция выполняется с двумя операндами разных типов, то обе величины приводятся к высшему (по рангу) из типов.

```
int z = 5;
float y= 2.11
z = z/y; // переменная z примет значение 1.848
```

Язык Си позволяет явно преобразовать тип выражения или переменной, указывая желаемый тип в круглых скобках перед выражением, например: `(char)(120 + 0.5) /* значение выражения будет приведено к символу 'x'. */`

Этот результат получен следующим образом: при сложении получено число 120.5, при приведении типов значение преобразуется в 120, т. к. тип char относится к целым типам. Число 120 – код символа «x».

4.2 Производные типы данных

4.2.1 Указатели

Указатели делятся на две большие группы:

- указатели на объекты;
- указатели на функции.

Оба типа указателей – это специальные переменные для хранения адресов памяти.

В этом разделе подробно рассмотрены указатели на переменные, указатели на функции будут обсуждаться далее.

Указатель на переменную заданного типа содержит адрес переменной указанного типа. Например, `int * x;` – адрес переменной целого типа. Так как указатель – это тоже переменная, возможно описание указателя на указатель `(int **y)` и т. д. Размер указателя на переменную 4 байта. Рекомендуется обнулять указатель до его использования. Например, это можно выполнить при объявлении указателя:

```
int *k = NULL; // пустой (нулевой) указатель
```

Если указатели используются для работы с динамической памятью (например, для значений сохранения переменных в динамической памяти), то для инициализации значения указателя (т. е., для того, чтобы в указателе сохранить адрес памяти) необходимо выполнить выделение памяти. Выделение памяти в языке Си позволяют выполнить 2 функции – `malloc()` и `calloc()`.

Прототип функции `malloc` выглядит следующим образом:

```
void malloc(size_t sizem);
```

Функция выделяет блок памяти размером `sizemem` байт и возвращает указатель на начало блока. Содержание выделенного блока памяти не инициализируется, оно остается с неопределенными значениями.

Параметры функции `sizemem` – размер выделяемого блока памяти в байтах.

Функция возвращает адрес, с которого начинается выделенный блок памяти. Обратите внимание, тип возвращаемого значения определен как `void*`, поэтому этот тип данных может быть приведен к желаемому типу данных.

Если функции не удалось выделить требуемый блок памяти, возвращается нулевой указатель.

Прототип функции `calloc`:

```
void* calloc(size_t num, size_t size);
```

В отличие от `malloc` функция инициализирует выделенную область памяти нулевыми значениями.

В приведенном примере выделяется память под переменные типа `int` и `float`.

```
int *x;
float *k;
x = (int)malloc(sizeof(int)); /* выделить память для хранения целочисленной переменной */
y = (float)calloc(sizeof(float)); /* выделить память для хранения вещественной переменной */
```

Наиболее часто указатели используются для работы с массивами. В таких случаях функция `malloc` (или `calloc`) выделяет память сразу под группу элементов заданного типа.

4.2.2 Ссылки

Ссылка *– это адрес переменной, описанной в программе.*

Для получения ссылки к имени переменной слева дописывается знак `&`. Например:

```
int z = 12; //объявлена и задана целая переменная
int *k = &z;/* указателю k присваивается значение адреса переменной z. */
```

Ссылка, в отличие от указателя, является неизменяемой величиной. Это вытекает из определения понятия ссылки – невозможно изменить адрес уже существующей переменной.

4.2.3 Разыменование указателей

Для получения или инициализации значения, хранящегося по заданному адресу, используют операцию разыменования указателя. В следующем фрагменте программы в память, выделенную под указатель, записывают значение переменной `z`:

```
int z = 12;
int *k = (int*)malloc(sizeof(int));
*k = z;
```

Рассмотрим подробно состояние памяти при выполнении этого фрагмента программы. При выполнении первой строки программы выделяется 4 байта памяти для хранения целочисленной переменной. В эту область памяти записывается значение переменной `z`. При выполнении второй строки программы в динамической области памяти выделяется 4 байта для хранения целого числа. Адрес выделенной памяти сохраняется в переменной `k`. После выполнения этой операции память остается неиспользованной. И только после выполнения третьей строки в эту область памяти записывается значение переменной `z`. При выполнении этого фрагмента программы значение `12` хранится в памяти дважды – в переменной `z` и по адресу `k`.

4.3 Сложные типы данных

4.3.1 Массивы

Язык программирования Си позволяет создавать переменные сложных типов данных, основанных на базовых типах или на типах, определенных пользователем.

Одномерным массивом называется набор данных одного типа. В языке Си различают статические и динамические массивы. Статический массив описывается следующим образом:

```
int x[15]; // описание массива из 15 целочисленных элементов.
```

Память для хранения статического массива выделяется автоматически непосредственно после описания. Таким образом, для хранения массива `x` выделено 60 байт.

Нумерация элементов массива начинается с `0`. Для обращения к заданному элементу массива используются `[]`. Например, фрагмент программы:

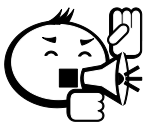
```
int x[5];
x[0]=10;
```

задает значение элемента с индексом `0`. Таким образом, могут быть заданы значения всех элементов массива.

Инициализация статического массива может быть проведена и при описании:

```
float y[5] = {1.1,2.2,3.3,4.4,0.05};
```

В этом случае `y[0] = 1.1, y[1] = 2.2, y[2] = 3.3, y[3] = 4.4, y[5] = 0.05`.



Основные ошибки при работе со статическими массивами:

- обращение к несуществующему элементу массива, например, в программе объявлен массив из 10 переменных, а обращаются к элементу с индексом 10, такая ошибка относится к разряду логических ошибок, не определяющихся при компиляции;
- использование еще не проинициализированных элементов массива также относится к логическим ошибкам, например, в следующем фрагменте программы массив `x` описан, но не проинициализирован, тем не менее, элемент массива с индексом `0` участвует в вычислении значения переменной `y`:

```
int x[10];
int y =x[0]+15;
```

Значение переменной `y` в этом случае не может быть определено однозначно.

Если программа должна уметь обрабатывать массивы произвольной размерности, целесообразней использовать динамические массивы. При описании указателей Вы познакомились с функциями выделения памяти `malloc()` и `calloc()`. В рассмотренном примере память выделялась под один элемент. Язык Си позволяет выделить память под несколько элементов:

```
int* x = (int*)malloc(sizeof(int) *15); /* выделение памяти под массив из 15 элементов */
```

Таким образом, понятия «указатель» и «массив» очень тесно связаны между собой. *Имя массива в Си – это указатель на первый элемент массива.* Обращение к элементу динамического массива с заданным индексом не отличается от обращения к элементу статического массива. Инициализировать элементы динамического массива возможно только поэлементно, после выделения необходимой памяти. Следующий фрагмент программы описывает и инициализирует элементы динамического массива `x`. Принцип работы цикла `for` подробно описан в следующей главе.

```
int *x,i;
int n = 5;
x = (int*) malloc(sizeof(int)*n);
for(i=0;i<n;i++)
```

```
x[i]=i;
```

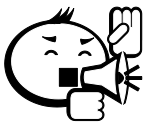
После выполнения этого фрагмента программы элементы массива *x* определены следующим образом:

```
x[0]=0,x[1]=1,x[2]=2,x[3]=3,x[4]=4.
```

Освобождение памяти после работы с динамическими массивами нужно выполнять следующим образом:

```
free(x);
```

– функция стандартной библиотеки языка Си, предназначенная для освобождения ранее выделенной динамической памяти. Функция принимает указатель на область памяти, подлежащую освобождению, или `NULL`.



Основная ошибка при работе с динамическими массивами – невыделение памяти под массив. Эта логическая ошибка приводит к совершенно непредсказуемым результатам работы программы.

Многомерные массивы принято называть матрицами. Матрица – это таблица однотипных данных, имеющая заданное количество строк и столбцов. По определению, каждый элемент характеризуется двумя индексами – номером строки и номером столбца. Статические матрицы в Си описываются следующим образом:

```
float y[5][4]; /* вещественная матрица из 5 строк и 4-х столбцов. */
```

Обращение к элементу матрицы, находящемуся в `i`-й строке и `j`-м столбце – `y[i][j]`. *Всегда* первый индекс – номер строки, второй индекс – номер столбца. Инициализация элементов статической матрицы может выполняться при описании:

```
int z[2][2] = {1,2,3,4};
```

Элементы при такой инициализации записываются построчно, то есть, элементы строки с номером `0` – `{1,2}`, строки с номером `1` – `{3,4}`.

При работе с матрицами разной размерности возможно использование динамических матриц. Если одномерный массив описывается в Си как указатель, матрица описывается как указатель на указатель. Обязательно выделение памяти, которое проводится по следующему алгоритму:

1) под указатель на указатель выделяется память под массив указателей (каждый элемент этого массива сам будет массивом);
2) под каждый элемент полученного массива указателей выделяется память под одномерный массив.

Ниже приведен фрагмент программы, описывающий динамическую матрицу и выделяющий память для элементов такой матрицы.

```
int **M;
int n = 10,i; // количество строк
int m = 5; // количество столбцов
M = (int**)malloc(sizeof(int*)*n);
for(i=0;i<n;i++)
 M[i] = (int*)malloc(sizeof(int)*m);
```

Освобождение занятой памяти происходит в обратном порядке:

```
for(i=n-1;i>=0;i--)
 free(M[i]);
free(M);
```

4.3.2 Структуры

Для создания сложных типов данных в языке Си используется тип данных – структура. Синтаксис описания структуры:

```
struct [имя]
{ тип поле1;
тип поле2;
…
}
```

Такое описание называется шаблоном структуры. Имя структуры необязательный элемент. Опишем структуру, содержащую информацию о студенте (Фамилия Имя Отчество, Номер группы, Количество баллов):

```
struct student {
char fio[31];
char group[6];
float ball;
}
```

Структура с именем `student` содержит три поля – массив символов (строка) для хранения фамилии (`fio`), строка для хранения номера группы (`group`), вещественная переменная для хранения количества баллов (`ball`).

Опишем переменную типа `struct student` –

```
struct student st1; // описана переменная st1
```

Опишем массив структур:

```
struct student m_st1[22]; // описан массив m_st1
```

Для обращения к полям структуры используется следующий синтаксис:

<имя переменной>. <имя поля>

`st1.fio` – строка в поле `fio` переменной `st1`.

`st1.fio[0]` – первый символу поля `fio` переменной `st1`.

`m_st[20].fio` – строка поле `fio` двадцатого элемента массива.

Полем структуры может быть структура:

```
struct  {
struct
     {
char fam[21];
char name[15];
char fname[20];
} fio;
char group[6];
float ball;
} st1;
```

В этом же примере показан другой способ описания переменной структурного типа. Обращение к полю `name`, переменной `st1` выглядит следующим образом: `st1.fio.name`.

При использовании указателя на структуру обращение к полям выглядит следующим образом:

```
struct Coord {
int x;
int y;
}
struct Coord *z;
…
z->y; // знак «-» и «>»
…
```

В языке Си существует механизм определения собственных, пользовательских типов – `typedef`. В следующем примере для структуры `Coord` определен тип `Pixel`.

```
typedef struct
{
int x;
int y;
} Pixel;
```

В таком случае описание переменных будет выглядеть следующим образом:

```
Pixel z,m. // описаны переменные m и z.
```

4.3.3 Объединения

В Си существует тип данных, который позволяет, во-первых, хранить разнотипные данные в одной области памяти, во-вторых, обращаться к частям одного целого. Например, тип `int` занимает в памяти 2 байта, а тип `char` 1 байт, используя тип `union` (объединение) можно обратиться к младшему или старшему байту типа `int`. Размер переменной такого типа определяется по полю с максимальной длиной.

Синтаксис:

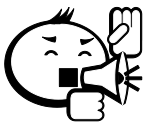
```
union [имя]
{
тип поле1;
тип поле2;
…
}
```

Например:

```
 union massiv
{
float f; int i; char ch[2];
} elem;
```

Переменная `elem` занимает в памяти 4 байта, т. к. это необходимо для хранения максимального по размеру поля типа `float`. Необходимо помнить, что в определенный момент времени в объединении храниться только одно из указанных полей (в отличие от структуры).

```
Elem.f = 23.0;
Elem.i = 11; // 23.0 стирается, записывается 11
Elem.ch[0] = 'c'; // 11 стирается, записывается «c»
Elem.ch[1] = 'd' // записывается «d»
```



Анализ содержимого объединения полностью ложится на программиста.

4.3.4 Перечисления

Перечисления *– это способ задания смысловым константам соответствующих целых числовых значений.*

Синтаксис:

```
enum [имя перечисления] {переменная 1 = [числовое значение], переменная 2 = [числовое значение], …}
```

Приведем пример перечисления:

```
typedef enum {min = 0,max = 1,eqv = 2} boolean;
```

Заданы смысловые константы `min`, `max`, `eqv`, принимающие значения 0,1,2.

Если значения констант не задаются – то перечисление начинается с 0. Возможен и следующий способ задания перечислений – `typedef enum {min = 1,max ,eqv } boolean;` – в этом случае следующей константе (`max`) присвоится значение 2, и т. д.

Переменной-перечислению нельзя присваивать значения констант, даже если эта константа входит в перечисление.

Например:

```
boolean flag;
flag = 1;
/* Невозможно выполнить эту операцию, хотя 1 входит в константы перечисления; */
flag = eqv; // Верное присвоение
```

4.4 Объявления и инициализация переменных

В программах на языке Си отсутствует блок описания переменных, при использовании переменных необходимо помнить одно правило – *описывать и инициализировать переменную необходимо до ее использования в операциях.* Если Вы используете неописанную переменную, то компилятор генерирует сообщение об ошибке, информируя Вас о том, что для используемого идентификатора не определен тип данных. При описании переменной не происходит ее автоматическая инициализация, как это происходит, например, в языке Pascal. Вы должны сами позаботиться о начальном значении используемой переменной. При работе с динамическими переменными (указателями) необходимо помнить, что выделение памяти, как и инициализация, не происходит автоматически. Ответственность за выделение и освобождение памяти полностью ложится на программиста. Далее приведены примеры инициализации разных типов переменных:

```
int x; // объявление переменной
int k=0;// объявление и инициализация переменной
float z*; // объявление переменной-указателя
```

```
    char m = 'c'; //объявление и инициализация переменной
    int *y = (int*)calloc(sizeof(int)*15); /* объявление указателя и выделение памяти */
    x = 13;// инициализация переменной
    z  =(float*)calloc(sizeof(float)*10);/*  выделение  памяти  под указатель */
    int i; // описание переменной i
    for( i=0;i<5;i++)
    y[i]=i+1; // инициализация значений массива
    for(i=0;i<10;i++)
      z[i] = i/(i+1.); // инициализация значений массива
```

Контрольные вопросы по главе 4

1. Перечислите типы языка Си, относящиеся к простым типам.
2. Перечислите типы языка Си, относящиеся к производным типам.
3. Сколько байт занимает в памяти переменная `z`?
   ```
   struct {
   int x;
   float z;
   char m[10];
   } z;
   ```
4. Какое значение примет переменная `x` после выполнения следующих действий?
   ```
   float x = 12/25;
   ```
5. Какое значение примет переменная `x` после выполнения следующих действий?
   ```
   int x = 25/12;
   ```
6. Сколько памяти занимает в памяти переменная типа `double`?
7. Как связанны между собой типы `char` и `int` в языке Си?
8. Запишите значение, которое будет храниться по адресу `p` после выполнения следующего фрагмента программы:
   ```
   int *p = (int*)malloc(sizeof(int));
   int x = 5;
   *p = 5*x;
   ```
9. Возможно ли изменить значение адреса переменной, описанной следующим образом?
   ```
   int *x = NULL;
   ```

```
int z = 12;
&z = x;
```

10. Каким образом можно получить адрес переменной?
11. Поясните понятие «разыменование указателя».
12. Каким образом можно получить значение, хранящееся по заданному адресу?
13. Запишите структуру, описывающую следующие данные:
    Идентификационный номер;
    Фамилия;
    Имя, Отчество;
    Номер телефона.
14. Как называется тип данных, который позволяет хранить в одной области памяти разнотипные данные?
15. Какими способами можно описать массив целых чисел?

5 Подготовка и исполнение программы на языке Си

5.1 Этапы подготовки программы к исполнению

Текст программы на языке Си передается *препроцессору*, который выполняет директивы, находящиеся в ее тексте.

На самом деле происходит выполнение условных директив, если они имеются. Все остальные директивы препроцессора определяют действия по размещению в программе текста заголовочных файлов (директива `include`), или макросов (директива `define`). Таким образом, после работы препроцессора получается *полный текст программы*.

Далее программа передается *компилятору*, который выявляет синтаксические ошибки в тексте, если таких ошибок не нашлось, компилятор строит *объектный модуль*.

Следующий этап – работа *компоновщика*, который должен сформировать *исполняемый модуль* из нескольких объектных модулей – объектных модулей программы (если исходный текст состоит из нескольких файлов) и объектных модулей библиотек.

После выполнения описанного процесса программа готова к исполнению.

5.2 Директивы препроцессора

5.2.1 Директива `#include`

Ядро языка Си содержит типы данных, операции и операторы. Все функции, используемые в программе, описаны в отдельных файлах, так называемых библиотеках, описания этих функций (*прототипы функций*) хранятся в заголовочных файлах (файлы с расширением `*.h`). Для того чтобы программа не только вычислила значение, но и вывела результат на экран, необходимо подключить используемые заголовочные файлы. Подключение файла – директива компилятора `#include <имя подключаемого файла>` – состоит во включении текста заголовочного файла в файл с написанной программой. Так как по правилам прототипы функций должны быть описаны до использования функций, то файл с программой на Си должен начинаться с подключения заголовочных файлов. Например, директива

```
#include <conio.h>
```

подключает заголовочный файл с прототипами функций, работающими с вводом-выводом информации на консоль.

При подключении пользовательского заголовочного файла имя файла пишется в двойных кавычках, например следующим образом:

`#include "my_func.h"` – подключение нестандартного заголовочного файла с именем `my_func.h`.

В дальнейшем при описании функции всегда будет указываться заголовочный файл, содержащий ее прототип.

5.2.2 Директива `#include_next`

Директива `#include_next` применяется внутри заголовочного файла, включаемого в другой файл, и вызывает поиск нового файла. Поиск начинается с каталога, который следует за тем, где был найден текущий файл.

5.2.3 Директивы `#define, #undef, #ifdef, #ifndef`

Директива `#define` используется для определения макро. Различают два вида макро – простой и с параметрами.

Синтаксис простого макро:

```
#define идентификатор-макро <последовательность лексем>
```

Каждое появление идентификатора макро в тексте программы, следующее за этим определением, будет замещаться последовательностью лексем.

```
#define STOP exit(0);
#define HELLO "Добрый день"
#define hidden
```

Выше описанные макро работают следующим образом:

- все встреченные в программе идентификаторы `STOP` заменятся на вызов функции `exit(0);`
- `HELLO` – на строку `"Добрый день"`;
- `hidden` – на пробел.

Последний макро `hidden` называется *пустым*.

Препроцессор позволяет отменить ранее описанный макро с помощью директивы `#undef`.

Синтаксис: `#undef` идентификатор-макро.

После выполнения этой директивы макроопределение забывается и считается неопределенным.

Директивы `#ifdef` и `#ifndef` используются для проверки, определен ли идентификатор в данный момент или нет.

Для определения макро с параметрами используется синтаксис:

`#define идентификатор-макро(<список-аргументов>)`

последовательность лексем.

Обратите внимание на то, что нельзя использовать разделители между идентификатором макро и «(».

Необязательный список аргументов – это последовательность идентификаторов, разделенных запятыми. Каждый идентификатор играет роль формального аргумента.

Синтаксис вызова макроопределения с параметрами идентичен вызову функции; и в действительности многие функции стандартной библиотеки Си реализованы как макро. Вызов макроопределения с параметрами приводит к двум замещениям. Во-первых, идентификатор макро и аргументы в скобках замещаются последовательностью лексем. Затем все формальные аргументы, встретившиеся в последовательности лексем, заменяются соответствующими аргументами из списка действительных аргументов. Например:

```
#define Func(x,y) { (x) * (x) + (y) * (y)}
...
int v = 4, y, w = 5;
y = Func(v,w);
```

приводит к следующей замене: `y = (v)*(v) + (w)*(w).`

5.2.4 Директивы условной компиляции.

К директивам условной компиляции относятся директивы `#if`, `#elif`, `#else`, `#endif`.

Директивы условной компиляции работают как обычные условные операторы Си. Синтаксис условных директив выглядит следующим образом:

```
#if условие1
<секция1>
<#elif константное выражение1 новая строка секция2>
…
<#elif константное выражениеn новая строка секцияn>
…
<#else последняя секция>
```

```
#endif
```

Если условие1 истинно, строки текста программы (возможно пустые), представленные секцией1 (это могут быть строки с командами препроцессора или строки кода программы), обрабатываются препроцессором и передаются в компилятор Си. Иначе, если условие1 ложно, секция1 пропускается.

При истинном условии препроцессорной обработки секции1 управление передается на соответствующую директиву `#endif`, которая заканчивает условный эпизод, препроцессор начинает обрабатывать программу далее. В случае ложного условия управление передается на следующую директиву `#elif` (если она есть) и вычисляется константное выражение2 и т. д. Каждая директива `#if` должна заканчиваться `#endif`. Константные выражения, используемые в директивах условной компиляции, должны вычисляться в целое константное значение.

5.2.5 Управляющая директива `#line`

Директива `#line` используется для передачи номеров строк в программу для перекрестных ссылок и выдачи ошибок.

Синтаксис:

```
#line целая константа <"имя файла">
```

Следующая строка программы получена из строки номер «целая константа» файла «имя файла».

5.2.6 Директива `#error`

Синтаксис:

```
#error сообщение
```

Выдается сообщение

```
Error: имя файла с текстом программы строка# : error directive: сообщение
```

Эта директива обычно используется с условными директивами, для того чтобы компилятор вывел сообщение об ошибке и остановил компиляцию. Например, возможно с помощью этой директивы осуществить проверку данных на корректность. Например, макроопределение `VAL` в дальнейшем тексте программы может использоваться в качестве делителя, тогда избежать ошибки деления на 0 поможет следующий фрагмент:

```
#if (VAL ==0)
#error VAL не должно равняться 0!
#endif
```

5.2.7 Директива #pragma

Директива #pragma применяется для директив, зависимых от реализации:

```
#pragma имя директивы
```

С помощью #pragma Си может определить директивы, которые не конфликтуют с другими компиляторами, поддерживающими #pragma. Если версия компилятора не распознает имя директивы, директива #pragma игнорируется без выдачи ошибок или предупреждений.

В компиляторе GCC прагма-директивы определяются как два слова. Первое из них «GCC», второе – имя особой прагмы.

#pragma GCC dependency – проверяет отметку времени (timestamp) текущего файла и сравнивает его с отметкой времени другого файла. Если другой файл оказывается более новым, выдается предупреждение.

#pragma GCC poison – так называемая испорченная прагма. Может применяться для того, чтобы при любом использовании указанного имени выдавалось предупреждение. Может использоваться, например, для того, чтобы гарантировать, что определенная функция не будет вызываться.

#pragma GCC system_header. Код, который Вы укажете после директивы, будет считаться кодом системного заголовочного файла.

Помимо директивы #pragma описан оператор _Pragma. Оператор используется для генерации сообщений прагмы внутри макросов. Обычную директиву #pragma внутри макроопределений использовать нельзя.

5.3 Ввод-вывод информации

5.3.1 Функция printf

Для вывода информации на экран Си предоставляет множество возможностей. Есть функции, выводящие на экран только строки, только целые или вещественные числа. Функция printf может использоваться для вывода на экран информации любого типа. Прототип функции описан в заголовочном файле stdio.h – функции стандартного ввода-вывода. Описание функции:

```
printf(Управляющая строка, <аргумент1, аргумент2, …>);
```

Управляющая строка записывается в двойных кавычках и содержит информацию двух типов:

- печатаемые символы (константная строка);

- идентификаторы данных (спецификаторы формата).

Функция принимает список аргументов и применяет к каждому спецификатор формата. Количество спецификаторов формата и аргументов должно быть одинаковым.

Основные спецификаторы формата:

- `%d` – целое десятичное число;
- `%c` – один символ;
- `%s` – строка символов;
- `%e` – экспоненциальная запись числа с плавающей точкой;
- `%f` – десятичная запись числа с плавающей точкой одинарной точности;
- `%u` – десятичное число без знака;
- `%o` – целое восьмеричное число без знака;
- `%x` – целое шестнадцатеричное число без знака.
- `%lf` – десятичная запись числа с плавающей точкой двойной точности.

Помимо этого в спецификаторах используются модификаторы, форматирующие выводимую информацию. Рассмотрим применение модификаторов на спецификаторе `%f`. Аналогично форматируется информация других типов.

Модификатор состоит из двух чисел, разделенных точкой, и может иметь лидирующий знак «-». Записывается модификатор после знака «%», обозначающего начало спецификатора. В общем виде модификатор выглядит следующим образом: `%m.n` спецификатор. Первое число `m` задает ширину поля вывода для всего значения. Второе число `n` используется для форматируемого вывода чисел с плавающей точкой и задает количество дробной части числа, выводимых на экран. Отсутствие знака «-» говорит о том, что вывод будет отформатирован по правой границе поля вывода, присутствие – форматирование по левой границе поля вывода.

При записи спецификатора в следующем виде – `%10.4f` – все выводимое вещественное число запишется в поле из десяти символов. Дробная часть числа будет состоять из 4 знаков.

Например:

```
…
int old = 23;
float key = 15.164;
```

```
char String[15] = "Простая программа";
printf("/%10d/",old); // выведется /    23/
printf("/%-10d/",old); // выведется /23    /
printf("/%10.1f/",key); // выведется /   15.2/
printf("/%-10.4f/",key); //выведется /15.1640 /
printf("/%5.5s/",string); // выведется /Прост/
printf("/%-30s/",string); // выведется /Простая программа/
…
```

Для перевода вывода на другую строку используется специальный символ «\n».

```
…
#define PI 3.141
printf("Пример использования функции printf:\n число PI = %8.2f", PI);
…
```

На экране:

```
Пример использования функции printf:
число PI = 3.14
```

При наличии в строке вывода специальных символов, используемых для форматирования (например, %, \ и тому подобное), эти символы дублируются. Первый символ интерпретируется как специальный, а второй такой символ говорит о том, что это часть выводимой информации.

5.3.2 Функция scanf

Для ввода информации Си предлагает наиболее общую функцию (функцию работающую с разнотипными данными) scanf (заголовочный файл stdio.h).

Описание функции:

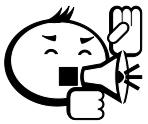
```
scanf(спецификатор формата, указатель на переменную);
```

В функции используются те же спецификаторы формата, что и в функции printf.

Обратите внимание. Имя массива является указателем, поэтому при вводе строк перед именем строки не пишется знак &. При вводе строки с помощью функции scanf строка вводится до первого встреченного пробела. Вся остальная часть строки обрезается.

Например:

```
…
char name[20];
scanf("%s",name); // ввод строкового массива.
int n;
scanf("%d",&n); // ввод целочисленной переменной n.
scanf("%c",&name[3]); //ввод четвертого символа массива name.
…
```



Будьте внимательны, при некорректном вводе (введены данные, не совпадающие с указанным спецификатором) не возникает ошибка выполнения. Но при этом дальнейшая работа программы непредсказуема.

Такие ошибки можно отследить, проанализировав результат работы функции `scanf`. При успешном вводе результат работы функции – количество введенных верно полей. Проверку ошибок ввода можно выполнить, например, следующим образом:

```
…
int y, n;
printf("Введите значение переменной n: ");
y = scanf("%d",&n);
if (y!=1) {printf("Введены неверные данные…\n ");
system("pause");
return 0;
}
else { …}
```

В рассмотренном примере количество вводимых полей – 1 (вводится одна переменная `n`). Следовательно, если переменная `y` не принимает значение единицы – произошла ошибка ввода. На экран выводятся соответствующие сообщения. Программа ожидает нажатия любой клавиши (`system("pause")`) и заканчивает работу (`return 0`). В случае успешной работы выполняется часть программы, описанная в блоке `else`.

При одном вызове функции `scanf()` возможно ввести значения более одной переменной. В этом случае спецификаторы формата пишутся один за другим, без пробелов. Каждому спецификатору должен соответствовать свой адрес переменной. При этом `scanf` при успешном вводе возвращает количество успешно считанных полей. Например:

```
…
```

```
float x,y,z;
printf("Введите значения переменных x,y и z: ");
int m = scanf("%f%f%f",&x,&y,&z);
…
```

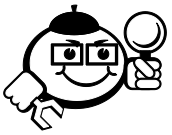
После успешного выполнения программы значение переменной `m` примет значение 3.

5.4 Простая программа на языке Си

Для того чтобы написать первую программу на Си, сформулируем несколько основных требований:

- так как ядро Си состоит только из основных типов и операторов и даже ввод и вывод организованы в виде функций, необходимо подключить библиотеки с описаниями прототипов функций. Для этого используется директива `#include`. Интегрированная среда *DEV-C++* сама включит необходимые для первой программы заголовочные файлы в шаблон программы;
- любая программа на языке Си начинается с выполнения функции `main`;
- любую переменную необходимо описать перед использованием;
- комментарии можно определять двумя способами: знаки `//` показывают, что вся последующая строка до конца – комментарий. Другой способ выделения комментариев – `/*` в начале комментария и `*/` в конце комментария.

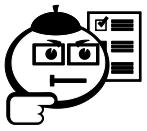
Выполняя все вышеописанные требования, напишем программу, которая запрашивает имя, фамилию и данные для вычисления по формуле.



Пример 5.1

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 system("chcp 1251"); // изменение кодовой страницы консоли
 char surname[30]; // описание строки символов
 char name[20]; // описание строки символов
 // печать строки
 printf("Введите, пожалуйста, свою фамилию: ");
 scanf("%s",surname); // ввод строки с клавиатуры
```

```
 printf("Введите, пожалуйста, Ваше имя: "); //печать строки
 scanf("%s", name); // ввод строки с клавиатуры
 printf("Введите произвольное целое число->");//печать строки
 int x,y; // описание целочисленных переменных
 scanf("%d",&x);// ввод целой переменной
 // печать строки
 printf("Введите второе произвольное целое число->");
 scanf("%d",&y); // ввод целой переменной
 printf("Здравствуйте, %s %s \n",name,surname);
 printf("Сумма двух чисел %d + %d = %d\n",x,y,x+y);
 printf("Разность двух чисел %d-%d = %d\n",x,y,x-y);
 system("pause");// ожидание нажатия клавиши
return(0); // окончание работы функции main
}
```



Попробуйте набрать текст этой программы, скомпилировать и отправить на выполнение.

Контрольные вопросы по главе 5

1. Перечислите основные этапы подготовки программы на языке Си к выполнению.
2. Опишите работу директивы `#include`.
3. Опишите макроопределение `ERROR` – литеральная строка «Ошибка выполнения».
4. Опишите макроопределение `ON` – константа, равная 1, `OFF` – константа, равная 0.
5. Опишите макроопределение с параметрами `GIPO`, которое вычисляет гипотенузу по двум заданным катетам.
6. Напишите программу, которая запрашивает с клавиатуры два вещественных числа и выводит на экран сумму их кубов.
7. Напишите программу, которая запрашивает с клавиатуры два целых числа и выводит на экран их произведение и частное.
8. Напишите программу, которая запрашивает с клавиатуры три символа и выводит их на экран в обратном порядке.

9. Какая функция в языке Си считается основной функцией ввода информации с клавиатуры?
10. Какая функция в языке Си считается основной функцией вывода информации на экран?
11. Какого типа переменную можно ввести функцией `scanf`, записанной следующим образом: `scanf(“%f”,&x);`
12. Что будет выведено на экран при выполнении следующего фрагмента программы:
```
int x = 12;
float y = 25.5;
printf ("\n *** %8.3f ***\n", y/x);
```
13. Что будет выведено на экран при выполнении следующего фрагмента программы:
```
int m = 3, k = 5;
printf ("\n *** \n *** [ %d ] *** \n***\n",k/m);
```
14. Найдите ошибки в следующем фрагменте программы:
```
int x;
scanf("%d",x);
```
15. Найдите ошибки в следующем фрагменте программы:
```
float m;;
scanf(“%d”,&m);
```
16. Найдите ошибки в следующем фрагменте программы:
```
char words[15];
scanf("%s",&words);
```
17. Найдите ошибки в следующем фрагменте программы:
```
int x,y;
scanf("%d%d",&x);
```
18. Найдите ошибки в следующем фрагменте программы:
```
float x = 12.1;
float y = 13.12;
float m = x+y;
printf("Пример вычисления суммы: %f");
```
19. Найдите ошибки в следующем фрагменте программы:
```
float x = 12.1;
float y = 13.12;
float m = x+y;
printf("Пример вычисления суммы: %f");
```
20. Найдите ошибки в следующем фрагменте программы:
```
float x = 12.1;
```

```
float y = 13.12;
printf("Значение x = %f \n Значение y = %f ", &x,&y);
```

6 Конструкции структурного программирования в Си

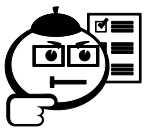
6.1 Следование

Программа, написанная на языке Си, выполняется последовательно, то есть команды программы (операторы, операции и функции) выполняются в порядке их вызова в программе. Поэтому любая простая программа может служить примером последовательного алгоритма. В качестве примера запрограммируем вычисление по заданной математической формуле.

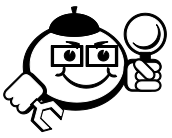
Си поддерживает множество математических функций, прототипы которых описаны в заголовочном файле `math.h`. Познакомимся с некоторыми из них.

- `abs(int x)` возвращает модуль целого числа `x`.
- `acos(double x)` возвращает арккосинус числа `x` в радианах.
- `asin(double x)` возвращает арксинус числа `x` в радианах.
- `atan(double x)` возвращает арктангенс числа `x` в радианах.
- `atof(char *s, double x)` преобразует строку `s` в вещественное число `x`.
- `cos(double x)` возвращает косинус числа `x` (`x` задано в радианах).
- `ceil(double x)` округляет число `x` в большую сторону.
- `exp(double x)` возвращает экспоненту числа `x`.
- `fabs(double x)` возвращает модуль вещественного числа `x`.
- `sin(double x)` возвращает синус числа `x` (`x` задано в радианах).
- `sqrt(double x)` возвращает квадрат числа `x`.
- `tan(double x)` возвращает тангенс числа `x` (`x` задано в радианах).
- `floor(double x)` округляет число `x` в меньшую сторону.
- `fmod(double x, double y)` возвращает остаток от деления числа `x` на число `y`.
- `hipot(double x, double y)` возвращает квадрат суммы числа `x` и числа `y`.
- `log(double x)` возвращает натуральный логарифм числа `x`.
- `log10(double x)` возвращает десятичный логарифм числа `x`.

- `modf(double x,double& y)` возвращает дробную часть числа `x`, по адресу `y` записывается целая часть исходного числа `x`.
- `pow(double x, double y)` возвращает `x` в степени `y`.



Самостоятельно, используя систему помощи интегрированной среды, ознакомьтесь с константами (макроопределениями), описанными в заголовочном файле `math.h`.



Пример 6.1

Написать программу вычисления значения `f` по формуле $f=\frac{x^3+y}{x^2+z-y}+xz-2\sin(y)$. Значения переменных `x,y,z` задавать с клавиатуры.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(int argc, char *argv[])
{
 system("chcp 1251");
 printf("Программа вычисления значения по заданной формуле\n");
 float x,y,z;
 printf("Введите значение x -->");
 scanf("%f",&x);
 printf("Введите значение y -->");
 scanf("%f",&y);
 printf("Введите значение z -->");
 scanf("%f",&z);
 float f = x*x*x + y;
 f = f/(x*x+z-y);
 f +=x*z;
 f -=2*sin(y);
 printf("\n Введены значения  %5.2f %5.2f  %5.2f \n"
" Значение f = %8.3f \n", x,y,z,f);
 system("PAUSE");
 return 0; }
```

6.2 Ветвление

6.2.1 Оператор проверки условия `if <else>`

Для организации ветвления алгоритма в Си используется оператор проверки условия `if (логическое выражение) {действия при истинном значении выражения} else {действия при ложном значении выражения}`. Оператор `else` может отсутствовать, если это обусловлено алгоритмом.

Напомним, что ложью в Си считается нулевое значение, соответственно истиной – любое ненулевое значение. Поэтому выражение `5>3` возвращает ненулевое значение, а выражение `3==0` – нулевое.

Логическое условие может быть сложным, т. е. может состоять из нескольких условий, связанных между собой логическими операциями:

- «И» (конъюнкция), в Си оператор `&&`, все логическое выражение считается истинным только в том случае, если истинны все простые выражения.
- «ИЛИ» (дизъюнкция), в Си оператор `||`, все логическое выражение считается ложным только в том случае, если ложны все простые выражения.

Например, запишем следующее условие «Если переменная `x` меньше переменной `y` и переменная *x* меньше переменной `z`»: `x<y && x<z`. Следующее условие демонстрирует операцию дизъюнкции «Если переменная `m < 10` или переменная `m` равна переменной `x`» : `m<10 || m == x`.

Если в блоках программы, выполняющихся при истинности или ложности условия, необходимо выполнить два или более действий, эти блоки определяются фигурными скобками. Приведем примеры.

- Если `x<y`, то вывести на экран значение суммы `x` и `y`, значение переменной `x` заменить на `10`.

```
if (x<y)
 { printf("Сумма ==> %d",x+y);
  x = 10;
 }
```

- Если `x` не равно `y`, вывести на экран соответствующее сообщение, в противном случае вывести на экран значения переменных, переменную `x` увеличить в два раза.

```
if (x!=y)
```

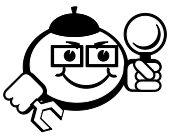
```
 printf("Переменные имеют неравные значения");
else {
     printf("x = %d, y = %d\n",x,y);
     x*=2;
    }
```

Обратите внимание на основные ошибки при работе с оператором `if`:

- вместо операции «==» (равно) используют операцию «=» (присвоить);
- не используются фигурные скобки:

```
if (x<y)
   printf("Сумма ==> %d",x+y);
  x = 10;
else …
```

Дополним пример из предыдущего пункта проверкой корректности ввода данных. При таком условии ошибки выполнения могут быть следующие: вместо числовых данных введены символьные данные, введены такие числовые данные, которые превращают выражение `x*x+z-y` в ноль. В случае таких исходных данных программа должна выдать на экран соответствующее сообщение и закончить свою работу.



Пример 6.2

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(int argc, char *argv[])
{
 system("chcp 1251");
 printf("Программа вычисления по заданной формуле\n");
 float x,y,z;
 printf("Введите значение x -->");
int m = scanf("%f",&x);
if (m !=1) {
               printf("Введены некорректные данные.\n ");
               system("pause");
               return 0; }
 printf("Введите значение y -->");
```

```
 m = scanf("%f",&y);
if (m !=1) {
                printf("Введены некорректные данные.\n ");
                system("pause");
                return 0; }
 printf("Введите значение z -->");
 m = scanf("%f",&z);
     if (m !=1) {
                printf("Введены некорректные данные.\n ");
                system("pause");
                return 0; }
 float f = x*x*x + y;
 float f1 = x*x+z-y;
 if (f1 == 0) { printf("Ошибка - деление на 0!!!\n");
                system("pause");
                return 0; }
 f = f/f1;
 f +=x*z;
 f -=2*sin(y);
 printf("\n Введены значения %5.2f %5.2f  %5.2f \n "
"Значение f = %8.3f \n", x,y,z,f);
system("PAUSE");
return 0; }
```

.....

6.2.2 Множественный выбор

Предположим, необходимо решить следующую задачу: по заданному значению целочисленной переменной `k` выполнить одно из действий:

- если `k = 1` – найти сумму переменных `x` и `y`,
- если `k = 2` – найти произведение `x` и `y`,
- если `k = 3` – найти разность `x` и `y`,
- если `k = 4` – найти разность `y` и `x`.

Если для решения этой задачи использовать оператор `if`, получается следующая структура с несколькими уровнями вложенности:

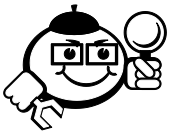
```
if (k==1){…}
else if (k==2){…}
     else if (k == 3) {…}
        else {…}
```

Си предлагает для решения таких задач оператор `switch`.

Синтаксис:

```
switch (выражение)
{
     case значение выражения1: операторы;
     case значение выражения2: операторы;
     …
     default: операторы;
}
```

Выражение в `switch` может принимать целочисленные значения (типы `int`, `long int`, `char` и т. п.). Работа оператора множественного выбора происходит следующим образом – значение выражения сравнивается со значением, указанным в первом блоке `case`. Если значения совпали, выполняются операторы из первого блока `case`. Далее управление без проверки условия передается в последующие блоки `case`. Для того чтобы пропустить их выполнение, в конце каждого блока должен быть выполнен оператор `break`, передающий управление на оператор, следующий за блоком `switch`. В блок `default` управление передается в случае, если не произошло ни одного совпадения выражения, указанного в `switch`, с выражениями, указанными в блоках `case`. Далее приведен текст программы, решающий поставленную задачу.



Пример 6.3

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  system("chcp 1251");
  int x = 2;
  int y = 3;
  int k;
  printf("\n Введите 1, для вычисления суммы x + y \n");
  printf("Введите 2, для вычисления произведения x*y \n");
  printf("Введите 3, для вычисления разности x - y \n");
  printf("Введите 4, для вычисления разности y - x \n");
   scanf("%d",&k);
  switch (k)
  {  case 1:
          printf("Сумма равна %d \n", x + y);break;
      case 2:
          printf("Произведение равно %d \n", x*y); break;
```

```
            case 3:
                printf("Разность равна %d \n",x - y); break;
            case 4:
                printf("Разность равна %d \n",y - x); break;
            default:
       printf("Введено число, не принадлежащее интервалу [1;4]!\n");
        }
     system("PAUSE");
     return 0;  }
. . . . . .
```

6.3 Циклы

6.3.1 Цикл с фиксированным числом операций `for`

Цикл – это конструкция структурного программирования, повторяющая определенные действия (итерации) несколько раз. При заданном количестве итераций в Си используется конструкция `for`. Синтаксис:

```
for(секция инициализации значения; секция проверки условия;
секция коррекции)
```

Значение, инициализируемое в первой секции, называется счетчиком цикла. Повторяемые действия называются телом цикла. Если тело цикла состоит из двух и более действий, оно заключается в фигурные скобки.

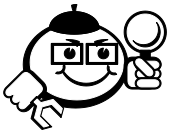
Секция инициализации выполняется один раз, поэтому может содержать описание переменной. На каждом шаге цикла значение счетчика подставляется в условное выражение второй секции, если выражение истинно, то управление передается в тело цикла, в противном случае управление передается за цикл. После каждого выполнения тела цикла выполняется операция из секции коррекции.

Например, цикл

```
int i;
for(i=0;i<3;i++)
 printf("%d \n",i);
```

будет работать следующим образом – счетчик `i` примет значение `0`. Условное выражение второй секции при таком значении счетчика истинно, на экран выведется значение переменной `i`, равное `0`. Управление передается в секцию коррекции и переменная `i` увеличивается на единицу. При этом условие все еще истинно, на экран выводится значение `1`. В ходе следующей итерации переменная-счетчик принимает значение `2`. Условное выражение остается истин-

ным. На экран выводится значение счетчика. После этой итерации переменная i становится равной 3. Условие при таком значении становится ложным, цикл заканчивает свою работу, управление передается следующей части программы.



Пример 6.4

Запишем программу, вычисляющую факториал заданного числа n. Для программирования факториала используем алгоритм произведения. Изначально факториал инициализируется значением 1. Затем организуется цикл, который умножает текущее значение факториала на счетчик цикла (счетчик изменяется как 2,3,4,… n), полученное значение становится текущим значением факториала.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 system("chcp 1251");
int i,n;
printf("Введите целое число n: ");
scanf("%d",&n);
int factorial = 1;
for(i=2;i<=n;i++)
   factorial*=i;
printf("Значение факториала числа %d равно %d \n", n,
factorial);
system("PAUSE");
 return 0;
}
```

Возможности цикла for:

- Уменьшение счетчика – `for(i=10; i>=0; i--)`.
- Изменение шага – `for(i=1, i<=10, i+=4)` – шаг 4.
  `for(i=1, i<=10, i*=2)` на каждой итерации значение i увеличивается в 2 раза.
- Использование не только целочисленных переменных в качестве счетчика – `for(x=0; x<10; x+=0.5)`.
  `for(c = 'A'; c <= 'Я'; c++)`.
- Возможность записывать несколько действий в одной секции –

`for (i=1,j=1;i<10,j<10;i++,j+=4)` – перечисление ведется через запятую.

- Возможность опускать любое из выражений заголовка – например, `for(;;)` – бесконечный цикл, т. к. пустое условие всегда считается истинным.

6.3.2 Циклы `while` и `do while`

Существует множество задач, при выполнении которых циклические действия необходимо проводить до тех пор, пока истинно какое-либо условие. Для реализации таких алгоритмов можно использовать циклы по условию `while` и `do while`.

Синтаксис:

```
while (условное выражение)
{
     тело цикла
}
```

Тело цикла `while` выполняется до тех пор, пока истинно условное выражение. Этот цикл называется циклом с предусловием. Цикл `while` может ни разу не выполниться (т. е. на входе в цикл условие ложно).

Например:

```
int d=134;
int i=1;
while(d>1)
 { d=d/10;
 i++;
}
printf("Количество цифр числа =%d", i);
```

Переменная-счетчик `i` изначально инициализируется значением `1` (число состоит хотя бы из одной цифры).

Пока значение исследуемого числа `d` больше единицы, число `d` делится на `10` (по правилам преобразования типов выполняется целочисленное деление) и значение переменной-счетчика увеличивается на единицу.

Синтаксис:

```
 do {
     тело цикла
} while (условное выражение)
```

Тело цикла `do while` выполняется, пока условие истинно. Этот цикл называют циклом с постусловием. Такой цикл выполнится хотя бы один раз.

Например:

```
int i=0;
do{
i++;
while(i<5);
```

Пока значение переменной `i` меньше пяти, значение `i` увеличивается на единицу. Описанный цикл выполнится 5 раз, и переменная `i` будет равна `5`.

Основные логические ошибки при использовании циклов.

- После заголовка цикла ставится точка с запятой. Такой цикл считается компилятором пустым, т. е. у него нет тела. Например:

```
for(int i=0; i<10; i++);
{ n+=10;
 y-=15;
}
```

При такой записи увеличение переменной `n` и уменьшение переменной `y` происходит за циклом, ровно один раз.

- Условие цикла заведомо ложно. Такой цикл никогда не выполнится.
- Условие цикла никогда не станет ложным. Такой цикл будет бесконечным. Как правило, эта ошибка возникает, если Вы не предусмотрели в теле цикла изменение переменной, от которой зависит условие цикла.

6.3.3 Операторы безусловной передачи управления `continue` и `break`

Оператор `break` досрочно завершает выполнение цикла. Управление передается оператору, следующему за циклом.

```
int n=15;
for(int i=0; i<n; i++)
{
     int z=rand()%200;
     if (z>100) break;
}
```

Цикл должен выполняться 15 раз. В переменную `z` записывается случайное значение в интервале от 0 до 199 (функция `rand()`, `stdlib.h`). Если получено случайное значение, большее 100, цикл заканчивает свою работу.

Оператор `continue` пропускает все последующие операторы тела цикла и передает управление в начало цикла.

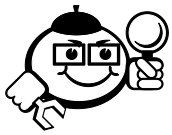
```
int f = 1;
do
{
     int z=rand()%100;
     if (z>30) continue;
     if (z<10) f = 0;
     printf("%d",z);
} while(f);
```

Описанный выше цикл работает следующим образом – цикл будет работать, пока переменная `f` равна единице. Если полученное случайное значение будет больше 30, то вторая проверка условия и функция печати полученного значения будут пропущены. Если полученное значение будет меньше 10, то переменной `f` будет присвоено значение 0. Значение выведется на экран и цикл закончит свою работу.

В итоге на экран будут выводиться числа не больше тридцати и как только будет получено число, меньшее десяти, цикл закончит свою работу.

6.4 Примеры использования операторов цикла

6.4.1 Вычисление суммы бесконечного ряда



Пример 6.5

Найти сумму бесконечного ряда $\sum_{n=1}^{\infty} \frac{(-1)^n}{(n+2)!}$ с заданной точностью.

Решение.

Определим, что значит найти сумму с заданной точностью. По виду общего элемента ряда $\frac{1}{(n+2)!}$ (обозначим q_n) видно, что элементы ряда убывают при увеличении n. Таким образом, начиная с какого-то n, элементы ряда будут меньше заданной точности. А это, в свою очередь, говорит о том, что начиная с

этого элемента приращение суммы никогда не станет больше заданной точности.

Таким образом, найти сумму бесконечного ряда с заданной точностью $0<\varepsilon<1$ – значит просуммировать все значения, $q_1, q_2, ...q_n$, пока не найдется такое n, при котором q_n станет меньше заданного ε. Для того чтобы составленный алгоритм был наиболее эффективным, принято выражать q_{i+1} член ряда через q_i. Для вывода итерационной формулы разделим q_i на q_{i+1}:

$$\frac{1}{(i+2)!}:\frac{1}{(i+3)!}=\frac{(i+3)!}{(i+2)!}=i+3.$$

q_i больше q_{i+1} в $i+3$ раз.

При программировании $(-1)^n$ воспользуемся свойством сложения: если элемент ряда положительный, то прибавим его к общей сумме, иначе отнимем его от общей суммы.

Запишем алгоритм вычисления суммы бесконечного ряда на псевдокоде:

```
алг Вычисление суммы бесконечного ряда нач
n:=1
q:= 1/(n+2)!
ввод ε=0.001
S=0
пока (q>=ε)
если n-четное то S:=S+q иначе S:=S-q
q:= q/(n+3);
n:=n+1;
кц
рез n-1, S.
кон
```

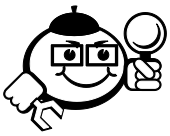
Составим по описанному алгоритму программу:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 system("chcp 1251");
 double eps,q,q1,S=0;
 int n = 1;
 printf("Введите значение точности: ");
 scanf("%lf",&eps);
```

```
      q=1/6.; // Вычисление значения первого элемента ряда
     // Обратите внимание на операцию деления двух констант
      while(q>=eps) {
             if(n%2==0) S+=q; else S-=q;
             q=q/(n+3);
             n++;
             }
      printf("Количество итераций %d\nЗначение суммы %8.3lf\n",
n-1,S);
      system("PAUSE");
      return 0;
     }
```

.....

6.4.2 Вычисления по итерационной формуле



Пример 6.6

Вычислить $z = \sqrt{x}$ по итерационной формуле – $z_{n+1} = \frac{z_n}{2} + \frac{x}{2 \cdot z_n}$.

Вычисления проводятся пока $|z_{n+1} - z_n| > eps$. При этом начальное значение $z_0 = 1.25$. Значение точности вычислений и переменной *x* задавать с клавиатуры. Сравнить результаты, полученные при вычислении по формуле и при использовании стандартной функции `sqrt()`.

```
Решение.
Опишем алгоритм, решающий поставленную задачу.
алг Итерации нач
z := 1.25
ввод eps
d := eps+1 // любое значение, большее заданной точности
ввод x
пока (d >= eps)
   z1:= z/2 + x/(2×z)  /* Вычислить текущее значение переменной, ис-
пользуя предыдущее значение*/
    d := |z - z1|;    /* Вычислить модуль разности между текущим
и предыдущим значением */
        z := z1;  /* Заменить предыдущее значение вычисленным те-
кущим значением */
кц
```

```
рез z.
кон
```

Напишем программу по описанному алгоритму.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 system("chcp 1251");
 double z = 1.25,eps, x,z1;
 printf("Введите значение точности: ");
 scanf("%lf", &eps);
 double d = eps+1;
 printf ("Введите значение x: ");
 scanf("%lf", &x);
 while(d>=eps){
        z1= z/2+x/(2*z);
        d = fabs(z-z1);
        z = z1;
        }
 printf("Квадратный корень из числа %8.3lf равен
%9.4lf\n",x,z);
 printf("С использованием функции sqrt - %9.4lf\n",sqrt(x));
 system("PAUSE");
 return 0; }
```

.....

6.4.3 Программирование численных методов

Рассмотрим еще один вид итерационного процесса – реализацию численного метода. Существует множество методов, позволяющих находить корни нелинейных уравнений, используя приближенные вычисления. Одним из них является метод секущих. Метод решает уравнения вида $f(x)= 0$. Если уравнение имеет несколько корней, необходимо несколько раз использовать метод, задавая различные начальные приближения.

Заменив производную $f'(x_n)$ разностью последовательных значений функции, отнесенной к разности значений аргумента

$$\Delta(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}},$$

получим следующую итерационную формулу:

$$x_{n+1} = x_n - \frac{f(x_n)}{\Delta(x_n)}.$$

Расчет по итерационной формуле заканчивается, когда два последовательных приближения x_n, x_{n+1} станут достаточно близкими $\left(\left|x_n - x_{n+1}\right| < \varepsilon\right)$ или когда $\left|f(x_n)\right| < \varepsilon$. Графическая иллюстрация метода представлена на рисунке 6.1.

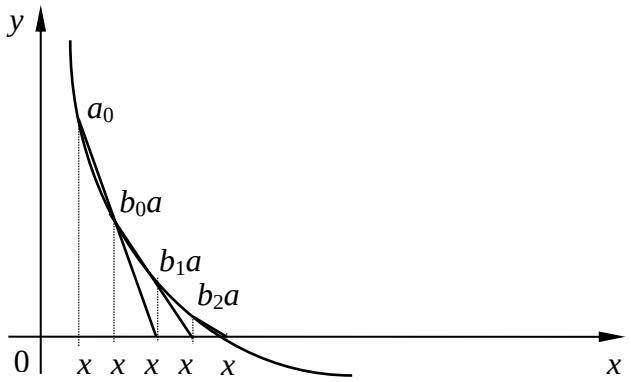
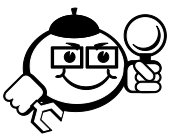


Рис. 6.1 – Графическая иллюстрация метода секущих

Алгоритм метода секущих выглядит следующим образом:

алг Метод секущих нач
ввод x_0, x_1, eps;
пока $|x_0 - x_1| >$ eps
$\text{delta} := \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}$;
$x_2 := x_1 - \dfrac{f(x_1)}{\text{delta}}$
$x_0 := x_1$
$x_1 := x_2$
кц
рез x_1



Пример 6.7

Приведенная ниже программа решает уравнение $\sin x - x + 0{,}15 = 0$, начальное приближение $x_0 = 0{,}5$. Программа выводит на экран найденный корень и выполняет проверку решения, подставляя полученный результат в заданную функцию.

```
#include <stdio.h>
#include <stdlib.h>
#define eps 0.0001 // макро, задающее точность вычислений
// макро, определяющее значение функции при заданном x
#define func(x) (sin(x)-x+0.15)
// макро, определяющее вычисление производной при заданном x
#define delta(x0,x1) (func(x1)-func(x0))/(x1-x0)
int main(int argc, char *argv[])
{
 system ("chcp 1251");
 double x0 =0.5, x1 = 0.52,x2,k;
 double z = func(x1);
 while( fabs(x0-x1)>eps)
  {
      k = delta(x0,x1);
      x2 = x1 - z/k;
      x0 = x1;
      x1 = x2;
      z = func(x1);
  }
 printf("Корень уравнения --> %6.4f\n",x1);
 printf("Проверка решения func(%6.4f) = %6.4f\n", x1,
func(x1));
system("PAUSE");
return 0; }
```

Контрольные вопросы по главе 6

1. Опишите работу оператора `if {…} else {…}` в языке Си.
2. Найдите ошибки в следующем фрагменте программы:

```
int f = 0, k = 5, m = 12;
if f==5
{k++; m++; }
```

3. Найдите ошибки в следующем фрагменте программы:

```
int x = 12, y=5, z = 4;
if (x>y)
y++;
z--;
else y--;
```

4. Что будет выведено на экран при выполнении следующего фрагмента программы?

```
float m = 12.1, n=3.4, f = 1.2;
float x = m;
if (n<x) x = n;
if(x >f) x = f;
printf("/n %3.4f /n",x);
```

5. Что будет выведено на экран при выполнении следующего фрагмента программы?

```
float p = 11.2;
int z = 12, k = 10;
if (p == z){
k+=2; p+=4; }
else{
if (z>k)
{
k =z; p*=2;}
else{
k = p; z = k;}
}
printf(" p = %f, z = %d, k = %d \n",p,z,k);
```

6. Опишите работу оператора `for` в языке Си.
7. Возможно ли записать цикл `for` следующим образом:

```
int i;
for( i =0, j = 0;i<12;i++,j--)?
```

8. Какие три секции выделяются в цикле `for` и для чего они предназначены?
9. Что будет выведено на экран при выполнении следующего фрагмента программы?

```
int k = 12;
int x = 1;
for(i=0;i<k;i++)
{
  x+=i;
  printf("%d \n",i);
}
```

10. Что будет выведено на экран при выполнении следующего фрагмента программы?

```
int n = 4;
int z = 5;
```

```
for(j=0;j<n;j++)
{
  z*=z;
  printf("%d \n",z);
}
```

11. Опишите работу цикла `while`.
12. Запишите пример цикла `while`, который будет выполняться бесконечно.
13. Запишите пример цикла `while`, который ни разу не выполнится.
14. Какой оператор цикла в Си обязательно выполнится хотя бы один раз?
15. Какой оператор цикла в Си называют циклом с постусловием?

7 Функции

7.1 Синтаксис

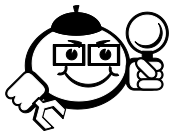
Важным принципом структурного программирования является принцип модульности. В модульной программе отдельные части, предназначенные для решения частных задач, организованы в функции. Вы уже использовали стандартные функции вывода на экран, чтения с клавиатуры и т. д. При такой организации один и тот же фрагмент программы можно использовать несколько раз, не повторяя его текст. Еще одним преимуществом модульного программирования является легкость отладки, чтения и тестирования программы. Приведем пример задачи, которая требует использования модульного подхода к программированию: сформировать три целочисленных массива, элементами которых являются случайные числа – `A[5]`, `B[10]`, `C[25]`. Для каждого массива найти сумму минимального и максимального элементов. Для решения этой задачи можно написать четыре функции – функцию создания массива, функцию печати массива, функции поиска минимального и максимального элементов. В основной функции `main` эти четыре функции будут использоваться для каждого из заданных массивов.

Синтаксически любая функция языка Си описывается следующим образом:

```
< тип возвращаемого результата> имя функции (<список формальных аргументов>)
{
      тело функции
}
```

7.2 Объявление и вызов функций

В языке Си объявить функцию можно несколькими способами. Первый способ – написать функцию до функции `main()`.



Пример 7.1

Напишем функцию поиска минимального числа из трех заданных целых чисел.

```
#include <stdio.h>
```

```
#include <stdlib.h>
// Объявление функции
int min(int a1, int a2, int a3)
/*  Тип возвращаемого результата – целый, формальные аргументы – три целых числа a1,a2,a3.*/
 {
  int m = a1;
  if (m>a2) m = a2;
  if (m>a3) m = a3;
  return m; /* возвращение результата с помощью оператора return */
 }
int main(int argc, char *argv[])
{
 system("chcp 1251");
 int x,y,z;
 printf("Введите значение x - ");
 scanf("%d",&x);
 printf("Введите значение y - ");
 scanf("%d",&y);
 printf("Введите значение z - ");
 scanf("%d",&z);
 int k = min(x,y,z); /* вызов функции с фактическим аргументами x,y,z. Переменной k присваивается возвращаемое значение */
 printf("\n Минимальное значение - %d \n",k);
system("pause");
return 0;
}
```

Итак, функцию можно описать до основной функции `main()`, передать результат в `main()` помогает оператор `return <возвращаемое значение>`. Оператор `return` кроме этого передает управление в вызывающую функцию. Таким образом, оператор может служить и для окончания работы функции. При вызове функции формальные аргументы заменяются фактическими аргументами.

Механизм работы модульной программы может быть описан следующим образом:

- при компиляции программы, имеющей пользовательские функции, для каждой функции создается отдельный исполняемый код;

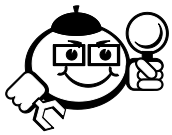
- при вызове функции выполнение программы прерывается, все данные программы сохраняются в стеке, начинает выполняться код тела функции, при этом происходит замена формальных аргументов на фактические, с которыми была вызвана функция;
- при достижении оператора `return` или закрывающей фигурной скобки функции управление передается в точку вызова вызывающей функции, при этом из стека возвращаются все сохраненные данные.

Сама функция может быть написана и после тела вызывающей функции или даже в другом файле, но в этом случае необходимо использовать прототипы функции. Прототипом функции называется указание типа возвращаемого результата, имени функции и списка типов формальных аргументов. Например, для функции `min` прототип будет выглядеть следующим образом:

```
int min(int,int,int);
```

При этом сама функция может располагаться как после функции `main()`, так и в другом файле. Использование прототипов связано с принципом обязательного первоначального описания объектов, используемых в программе, то есть, как и любую переменную, функцию необходимо описать перед ее использованием.

Использование прототипов позволяет объединять функции в библиотеки. Для каждой библиотеки может быть написан заголовочный файл с расширением `h`, в котором будут перечислены прототипы всех функций библиотеки. Сами функции при этом могут храниться в отдельном файле с расширением `c (cpp)`. Заголовочный файл подключается к файлу, содержащему вызывающую функцию с помощью директивы `include`, выполняется многофайловая компиляция программы (создание проекта).



Пример 7.2

Следующий пример описывает небольшую библиотеку простейших арифметических функций.

Файл `ariphm.h`

```
/* функция нахождения наибольшего общего делителя для двух
целых чисел */
int NOD(int,int);
float min(float,float);/* функция нахождения минимального из
двух вещественных чисел */
```

```
float max(float,float);  /* функция нахождения максимального
из двух вещественных чисел */
```

Файл ariphm.cpp

```
// поиск минимального числа
// аргументы функции - два вещественных числа x и y
// функция возвращает результат вещественного типа.
float min (float x, float y)
{
 if (x>y) return y;
          else return x; }
// поиск максимального числа
// аргументы функции - два вещественных числа x и y
// функция возвращает результат вещественного типа.
float max (float x, float y)
{
 if (x>y) return x;
          else return y; }
// поиск наибольшего общего делителя
// аргументы функции - два целых числа x и y
// функция возвращает результат целого типа.
int NOD(int x, int y)
{
 int z = max(x,y);
 int l = min(x,y);
 int k = l;
 while(z%l!=0)
 {
  k = z%l;
  z = max(k,y);
  l = min(k,y);
 }
 return k; }
```

Файл Demo_A.cpp – демонстрация вызова ранее описанных функций.

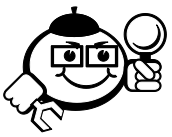
```
#include "ariphm.h" /* подключение собственного заголовочного
файла. */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
system("chcp 1251");
 printf("Введите два числа: ");
 int m,n;
```

```
      scanf("%d",&m);
      scanf("%d",&n);
     /* Вызов функции NOD. Формальные аргументы x,y заменены фактическими m и n. */
      printf("Наибольший общий делитель: %d \n",NOD(n,m));
     /* Вызов функции min. Формальные аргументы x,y заменены фактическими m и n. Результат работы функции явно преобразован к типу int. */
      printf("Минимальное число: %d \n",(int)min(n,m));
     /* Вызов функции min. Формальные аргументы x,y заменены фактическими m и n. Результат работы функции явно преобразован к типу int. */
      printf("Максимальное число: %d \n",(int)max(n,m));
      system("pause");
     return 0;}
```

Для того чтобы данный пример выполнился, необходимо выполнить многофайловую компиляцию. В проект включаются файлы `Demo_A.cpp`, `ariphm.h`, `ariphm.cpp`. Все файлы должны находиться в одной папке.

7.3 Локальные переменные

Переменные, описанные в теле функции, называются локальными переменными. Такие переменные видимы только внутри функции и создаются только при вызове функции. При достижении конца функции эти переменные уничтожаются, память, выделенная под хранение таких переменных, освобождается.



Пример 7.3

```
#include <stdio.h>
#include <stdlib.h>
int x = 1;
void func1 (){
 int m = 12;
 printf("\n x = %d \n", x);
 printf("\n m = %d \n", m);}
```

```
int main(int argc, char *argv[])
{
 system("chcp 1251");
 int m = 2;
 printf("\n x = %d \n", x);
 printf("\n m = %d \n", m);
 func1();
 printf("\n m = %d \n", m);
system("pause");
return 0; }
```

Переменная x по отношению к переменным m, объявленным в функциях main() и func(), называется глобальной переменной. Зона ее видимости – весь файл, содержащий программу. Время существования – время работы функции main().

Переменная m, описанная в функции func(), видна только внутри операторных скобок, ограничивающих тело функции.

Результат работы программы будет следующим:

```
x = 1  // Выводится значение глобальной переменной x.
m = 2 /* Выводится значение переменной m, описанной в функции main(). */
```

При вызове функции func() значение переменной m = 2 сохраняется в стеке. Описывается новая переменная m и инициализируется значением 12. На экран выводится значение переменной x, которая остается доступной и в функции.

```
x = 1 // Выводится значение m.
m = 12 /* При завершении работы функции func() переменная m перестает существовать. Из стека возвращается значение переменной m, описанной в функции main() */.
m = 2.
```

Попробуйте изменить значение переменной x в функции func() и попытайтесь объяснить полученный результат.

7.4 Выход из функций

Выйти из функции (вернуться в вызывающую функцию, закончить выполнение функции) можно несколькими способами.

При достижении закрывающей операторной скобки:

```
void PRINT(int x)
{
     printf("Значение %d ->",x);
}
```

Функция `PRINT(int x)` закончит свое выполнение по достижении закрывающей скобки.

При достижении оператора `return`:

```
float func(float x)
{
     x=x*180/3.14;
     return sin(x);
}
```

Функция `func(float x)` заканчивает свою работу при выполнении оператора `return` и передает в вызывающую функцию значение синуса заданного значения `x`.

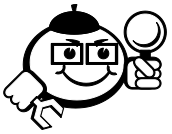
При вызове функции `exit()`:

```
void my_function(int x, int y)
{
if (y==0){
                printf("деление на 0");
                system("pause");exit(0);
           }
float d = ((float)x)/y;
printf("x/y = %f");
}
```

Функция закончит работу, если параметр `y` равен 0.

7.5 Передача параметров по ссылке

При решении некоторых задач требуется в качестве параметров возвращать в вызывающую функцию значения нескольких переменных. В этом случае рекомендуется описывать такие переменные в вызывающей функции, в вызываемую функцию передавать их по ссылке (т. е. передавать функции не их значения, а адреса памяти, в которой они хранятся).



Пример 7.4

Рассмотрим механизм передачи параметров по ссылке на примере создания функции, обменивающей значения двух заданных переменных.

```
#include <stdio.h>
```

```
#include <stdlib.h>
//параметры функции – адреса переменных n и m.
void change(int *n, int* m)
{
 int buf = *n;
 *n = *m;
 *m = buf;
}
/* после окончания работы функции изменения, выполненные по адресам, переданным в качестве параметров функции, сохраняются и в вызывающей функции */
int main(int argc, char *argv[])
{
 system("chcp 1251");
int x,y;
printf("Введите целое x: ");
scanf("%d",&x);
printf("Введите целое y: ");
scanf("%d",&y);
change(&x,&y); // вызов функции с адресами переменных x и y
printf(" x = %d\n y = %d\n",x,y);
system("pause");
return 0; }
```

. . . . . .

7.6 Рекурсивные функции

Рекурсивными называются функции, вызывающие сами себя. Запишем с помощью рекурсивной функции получение *n*-го числа Фибоначчи.

Числа *Fn*, образующие последовательность 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ... называются *числами Фибоначчи*, а сама последовательность – последовательностью Фибоначчи.

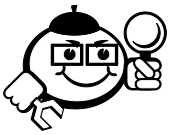
Суть последовательности Фибоначчи в том, что начиная с 1,1 следующее число получается сложением двух предыдущих.

$$x_i = x_{i-1} + x_{i-2},\ i = \overline{2,\infty},\ x_0 = 1,\ x_1 = 1.$$

Параметром функции будет значение *n* – номер искомого числа Фибоначчи.

Написание рекурсивной функции, как правило, начинается с формулирования условия выхода из функции. Если правило выхода не предусмотрено, функция будет бесконечной.

По определению последовательности, функция должна возвращать 1, если *n* равно 0 или *n* равно 1. Во всех остальных случаях функция должна возвращать сумму результатов при *n*–2 и *n*–1.



Пример 7.5

```
#include <stdio.h>
#include <stdlib.h>
int func(int n)
{
  if (n==0) return 1;
  if (n==1) return 1;
  return (func(n-1)+func(n-2));
 }
int main(int argc, char *argv[])
{
 system("chcp 1251");
int n;
printf("Введите целое n: ");
scanf("%d",&n);
int x = func(n);
printf(" x [%d] = %d\n",n,x);
system("pause");
return 0;}
```

Последовательность рекурсивных вызовов функции при `n=5` представлена на рисунке 7.1.

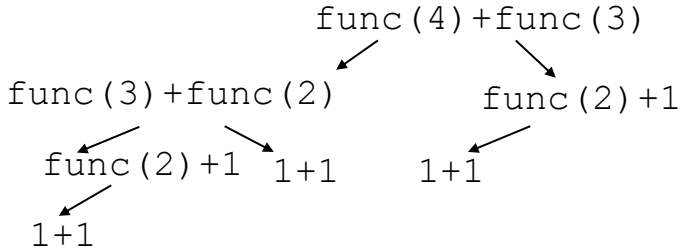


Рис. 7.1 – Дерево рекурсивных вызовов

Обратите внимание: любой алгоритм, записанный с помощью рекурсивной функции, всегда можно реализовать, не используя рекурсию.

Применение рекурсии всегда ограничивается размером программного стека – чем больше глубина рекурсии, тем больше требуется памяти для хранения предшествующих вызовов функции. Однако рекурсивная запись алгоритма (если она возможна) всегда короче, чем нерекурсивная.

Контрольные вопросы по главе 7

1. Опишите механизм вызова функций.
2. Результат какого типа возвращает функция, описанная следующим образом: `char My_F(int x, char m);`?
3. Перечислите параметры функции, описанной следующим образом:
   `int EQV(float x, int y, char c);`
4. Опишите действие оператора `return`.
5. В каком случае оператор `return` можно не использовать в функции?
6. Каким образом можно вернуть управление в вызывающую функцию?
7. Перечислите переменные, локальные для функции `min`:
```
int min(int *x, int n){
int min1=x[0];
for(int i=1;i<n;i++)
  if (x[i]<min1)min1 = x[i];
return min1;
}
```
8. Поясните понятие «прототип функции».
9. Напишите функцию, вычисляющую индекс минимального элемента заданного массива.
10. Напишите рекурсивную версию функции вычисления факториала от заданного числа `n`.
11. Функция `func1` записана следующим образом:
```
void func1(int z){
for(int i=0;i<z+1;i++){
prinf("%d \n",i*z);
}
```
    Что будет выведено на экран при вызове `func1(5);`?
12. Напишите функцию, принимающую значение 0 или 1, если точка с заданными координатами *x* и *y* принадлежит или не принадлежит окружности с заданным центром и радиусом. Параметры функции:

- координаты точки;
- координаты центра окружности;
- радиус.

13. Каким образом можно передать в вызывающую функцию несколько значений?
14. Запишите функцию, выводящую на экран последовательность *n* целых чисел $x_i = n + i - z$, $i = \overline{0;n}$. Параметры функции *n, z.*
15. Реализуйте, используя функции, метод секущих.

8 Массивы

8.1 Одномерные массивы

8.1.1 Инициализация массива

Основные моменты инициализации массивов упоминались в гл. 4, при описании типов данных. При решении задач очень часто размер массива является переменной величиной. В этом случае используются динамические массивы. Инициализация таких массивов выполняется в два этапа. Первый этап – выделение памяти, второй этап – инициализация значений элементов массива. Элементы массива могут вводиться с клавиатуры:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 system("chcp 1251");
// Описание указателя на целое число
 int *x;
 int n,i;
 printf("Введите размерность массива: ");
 scanf("%d",&n);
// выделение памяти
 x = (int*)malloc(sizeof(int)*n);
 for(i=0;i<n;i++){
  printf("x[%d] -> ",i);
// чтение элементов с клавиатуры
  scanf("%d",&x[i]);
 }
// очистка экрана
 system("cls");
// печать элементов массива
 for(i=0;i<n;i++) printf("%5d",x[i]);
 // освобождение памяти
 free(x);
 printf("\n");
 system("PAUSE");
 return 0;}
```

Элементы массива могут вычисляться по формуле:

```
#include <stdio.h>
```

```
#include <stdlib.h>
int main(int argc, char *argv[])
{
 system("chcp 1251");
// Описание указателя на целое число
 int *x;
 int n,i;
 printf("Введите размерность массива: ");
 scanf("%d",&n);
// выделение памяти
 x = (int*)malloc(sizeof(int)*n);
for(i=0;i<n;i++){
// вычисление значения элемента массива xi=i+2i
     x[i] = i+2*i;
// печать элементов массива
     printf("%d ",x[i]);
}
 free(x);
 printf("\n");
 system("PAUSE");
 return 0;
}
```

Элементы массива можно задавать случайным образом. Для этого можно использовать функцию `rand()`. Функция возвращает равномерно распределенную целую случайную величину в интервале от 0 до `MAXINT`. Прототип функции находится в заголовочном файле `stdlib.h`. Следующий фрагмент задает случайным образом элементы целочисленного массива `x`, значения элементов массива лежат в интервале [20,50].

```
…
int i;
for( i=0;i<n;i++)
  {
      x[i] = rand()%31+20;
      printf("%d ",x[i]);
 }…
```

С помощью функции `rand` можно получать и отрицательные значения:

```
…
for(int i=0;i<n;i++){
     x[i] = rand()%31-rand()%31;
     printf("%d ",x[i]);
}…
```

А также вещественные значения:

```
float *x;
…
x = (float*)malloc(sizeof(float)*n);
for(int i=0;i<n;i++)
{
     x[i] = rand()%101/(rand()%31+1.);
     printf("%7.2f ",x[i]);
}
…
```

Обратите внимание на знаменатель дроби. К полученному случайному числу прибавляется `1.`, для того чтобы знаменатель не обратился в `0` (на ноль делить нельзя). Единица должна быть вещественной, тогда знаменатель по правилам преобразования типов считается вещественной величиной и при делении целого числа на вещественное число результат также считается вещественным.

Попробуйте изменить вещественную константу `1.` на целую константу `1` и объяснить полученный результат.

8.1.2 Поиск значений в массиве

8.1.2.1 Поиск по заданному значению

Очень большое количество задач программирования сводится к поиску заданного элемента в массиве. Результатом такой работы будет индекс найденного элемента или информация о неуспехе поиска.

Напишем функцию, которая возвращает индекс первого встреченного элемента с заданным значением `k`.

Определим тип возвращаемого значения – индекс (номер) элемента является целым числом, поэтому функция будет возвращать тип `int`.

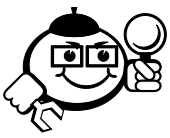
Определим параметры функции. При определении параметров функции необходимо исходить из того, что функция, по принципам структурного программирования, должна быть универсальной. То есть алгоритм, реализованный в функции, не должен зависеть от входных данных. Для этой задачи входными данными будут:

- массив (одна функция может применяться для разных массивов);

- размерность массива (массивы могут содержать разное количество элементов);
- искомое значение (одна функция сможет найти элементы с разными значениями).

Тогда заголовок нашей функции будет выглядеть следующим образом: `int Search(int*x,int k,int n)` – функция `Search` будет искать в целочисленном массиве `x` размерности `n` элемент с заданным значением `k`. Для поиска элементов в массиве с вещественными значениями функция будет выглядеть по другому: `int Search(float*x, float k, int n)`.

Обратите внимание, каким образом массив передается параметром функции.



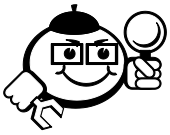
Пример 8.1

```
#include <stdio.h>
#include <stdlib.h>
int Search(int *x, int k, int n){
for(int i=0;i<n;i++)
{
/* цикл закончит свою работу после того, как будет найдено нужное значение */
 if (x[i] == k) return i;
}
// или будут просмотрены все элементы массива
return -1;
}
int main(int argc, char *argv[])
{
// функция рандомизации ядра датчика случайных чисел
 srand();
 int *x;
 int n,i;
 printf("Введите размер массива: ");
 scanf("%d",&n);
 x = (int*)malloc(sizeof(int)*n);
 for(i=0;i<n;i++){
  x[i] = rand()%300;
  printf("%d ",x[i]);
```

```
}
int y;
printf("\nВведите значение для поиска: ");
scanf("%d",&y);
int ind = Search(x,y,n);
if (ind == -1)
printf("В массиве нет элемента со значением %d\n",y);
else
printf("Элемент %d с индексом %d\n", y,ind);
free( x);
system("PAUSE");
return 0; }
```

Подумайте, каким образом организовать поиск последнего встреченного элемента с заданным значением.

Аналогичным образом можно решить задачу поиска второго, третьего и т. д. элемента с заданным значением.



Пример 8.2

Решим следующую задачу: *в массиве вещественных элементов найти сумму элементов, расположенных между первым и вторым нулевым элементами. Нахождение индексов элементов и суммы организовать в виде функций.*

Начнем решение задачи с определения функций. По условию задачи необходимо найти два индекса, но функции языка Си не могут возвращать более одного значения. Поэтому воспользуемся механизмом передачи параметров по ссылке. В общем случае задача может не иметь решения (в массиве может не быть нулевых элементов, или в массиве может быть ровно один нулевой элемент). Пусть функция возвращает значение 0, если в массиве нет нулевых элементов –1, если в массиве один нулевой элемент, и значение 1, если решение существует. Параметрами функции будут массив, его размерность, значение элемента для поиска и ссылки на целочисленные переменные, в которых после работы функции будут храниться значения индексов первого и второго найденных элементов. Заголовок функции в таком случае будет выглядеть следующим образом:

```
int search12(float *x, int n, float k, int* i1, int* i2)
```

Организуем поиск первого элемента с заданным значением с помощью цикла `while`:

```
int i=0;
/* пока не просмотрены все элементы массива и текущее значение не равно k, переходить к следующему элементу */
while(i<n && x[i]!=k)
i++;
// записать текущее значение i по адресу i1.
*i1=i;
```

Если в массиве нет нулевых элементов, то функция уже может закончить свою работу:

```
/* Если цикл закончил свою работу из-за невыполнения первого условия, то в массиве нет нулевых элементов. */
if (i==n) return 0;
```

Поиск индекса второго элемента можно начать с индекса `*i1+1`:

```
i=*i1+1;
/* пока не просмотрены все оставшиеся элементы массива и текущее значение не равно k, переходить к следующему элементу */
while(i<n && x[i]!=k)
i++;
// записать текущее значение i по адресу i2.
*i2=i;
/* если в массиве один нулевой элемент, то закончить выполнение функции */
if (i==n) return -1;
return 1; // решение найдено
```

Функция поиска суммы между двумя заданными элементами должна возвращать полученную сумму. Зависит такая функция от массива, его размерности и значений индексов элементов, между которыми находится сумма. Заголовок такой функции может выглядеть так: `float sum(float *x, int n, int i1, int i2)`.

```
float sum(float *x, int n, int i1, int i2){
if (i1>=i2|| i1<0||i2>n-1)
{ printf ("Некорректные данные");
     return 999;
}
float S = 0;
for(int i =i1+1; i<i2-1;i++)
     S+=x[i];
return S;}
```

Полное решение задачи выглядит следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int search12(float *x,int n,float k,int* i1,int* i2)
{ int i=0;
while(i<n && x[i]!=k)
i++;
*i1=i;
if (i==n) return 0;
i=*i1+1;
while(i<n && x[i]!=k)
i++;
*i2=i;
if (i==n) return -1;
return 1; }
float sum(float *x, int n, int i1, int i2)
{ if (i1>=i2|| i1<0||i2>n-1)
{ printf ("Некорректные данные");
          return 999;
}
float S = 0;
for(int i =i1+1; i<i2-1;i++)
S+=x[i];
return S;}
int main(int argc, char *argv[])
{
 srand();
 float *y;
 int n;
 printf("Введите размер массива: ");
 scanf("%d",&n);
// выделение памяти под вещественный массив
 y = (float*)malloc(sizeof(float)*n);
// формирование элементов массива случайным образом
 for(int i=0;i<n;i++){
         y[i] = rand()%50/(rand()%50+1.);
         printf("%6.3f ",y[i]);
 }
 int index1 = 0, index2 = 0;
// Поиск нулевых элементов
 int rezult = search12(y,n,0,&index1,&index2);
 float Summ;
// анализ выполнения поиска
 switch (rezult)
 { case 0: printf("\nВ массиве нет нулевых элементов.");
break;
```

```
        case -1: printf("\n В массиве один нулевой элемент.");
    break;
        case 1: Summ = sum(y,n,index1,index2);
        printf("\n Сумма элементов между %d-м и %d-м равна
     %8.3f",index1,index2,Summ);
             break;
     }
     free(y);
     system("PAUSE");
     return 0;}
```

.....

8.1.2.2 Поиск экстремальных элементов массива

Очень часто в решении задач требуется найти максимальное или минимальное значение в массиве. Эти алгоритмы подробно описаны в пп. 1.4.2.

Напишем функцию поиска максимального элемента в вещественном массиве, результат работы функции – значение максимального элемента.

```
float max(float *x,int n){
// примем за максимальное значение нулевой элемент массива
float max = x[0];
for(int i=1;i<n;i++)
{
/* если текущий элемент больше, чем максимальный, то изменить
значение максимального элемента на значение текущего элемента */
      if(x[i]>max) max = x[i];
}
 return max;
}
```

Вызов функции `max()` для массива `y` из предыдущего примера выглядит следующим образом:

```
float maxEl = max(y,n);
```

Попробуйте самостоятельно написать функцию поиска индекса максимального элемента.

8.1.3 Сортировка массивов

8.1.3.1 Сортировка обменом

Решение некоторых задач требует располагать элементы массива в упорядоченном виде. Процесс расположения элементов в определенном порядке называется сортировкой массива. Одним из простых методов сортировки на

месте (при работе не требуется дополнительный объем памяти) является сортировка обменом или «пузырьковая» сортировка.

Суть алгоритма состоит в следующем: сравниваются пары рядом стоящих элементов массива, если первый элемент пары меньше второго, то элементы меняются местами. После первого просмотра массива самый большой элемент встает на свое место, а маленькие по значению элементы на один шаг продвигаются к началу массива. Отсюда метод и получил свое название: «легкие» элементы плавно «всплывают» к началу массива. «Тяжелые» элементы быстро «тонут», встают в конец массива.

После того как все пары элементов массива просмотрены, массив еще не будет отсортирован, поэтому необходимо просмотреть пары элементов еще раз, и тогда еще один самый большой элемент встанет на свое место.

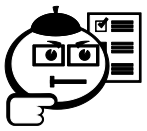
Если массив состоит из n элементов, то пар в таком массиве ровно n−1. На каждом шаге алгоритма самый большой элемент массива становится на свое место. Поэтому количество просматриваемых пар уменьшается с каждым шагом на единицу.

Таким образом, в алгоритме явно просматриваются два вложенных цикла. Внутренний цикл отвечает за просмотр пары элементов, количество шагов этого цикла зависит от внешнего цикла. Чем больше шагов выполнил внешний цикл, тем меньше пар просматривает внутренний цикл. Так как при выполнении одного шага внешнего цикла самый большой элемент становится на место, то количество шагов цикла равно количеству элементов массива −1. Однако массив может быть уже отсортированным до окончания работы внешнего цикла. Можно досрочно остановить выполнение цикла, если на каком-то i-м шаге во внутреннем цикле не произошло ни одного обмена.

Запишем алгоритм обменной сортировки на псевдокоде.

```
алг Сортировка обменом нач
     ввод n, X[n].
     цикл для i от 0 до n-1
          flag := 0; // Обменов нет
       цикл для j от 0 до n-i-1
          если X[j]>X[j+1] то
                    обмен X[j]↔X[j+1]
                    flag := 1; // Обмен есть
       кц
// Если обменов не было
       если flag==0 то
```

```
        Закончить выполнение цикла по i.
    кц
рез X.
кон
```



Запишите программу, реализующую сортировку произвольного массива методом обмена.

8.1.3.2 Сортировка выбором

Сортировка выбором использует другой принцип упорядочивания элементов. На начальном шаге алгоритма выбирается минимальный элемент и ставится на место нулевого элемента. На последующих, i-х шагах выбирается минимальный элемент среди элементов от i-го до (n−1)-го.

Запишем алгоритм сортировки выбором:

```
алг Сортировка выбором нач
ввод n, X[n]
цикл для i от 0 до n-1
// Считать минимальным элемент, расположенный на позиции k.
          k:=i
          цикл для j от i+1 до n
/* Если элемент на позиции j меньше минимального, то изменить позицию */
      Если X[j]<X[k] то k:=j
          кц
/* Если минимальный элемент не стоит на позиции i, то выполнить обмен */
      если k ≠i то Обмен X[i]↔X[k]
кц
рез X
кон
```

8.1.3.3 Сортировка вставками

Сортировка вставками упорядочивает массив, выполняя следующие действия: на начальном шаге алгоритма считаем, что последовательность из одного, первого элемента – упорядоченная последовательность. Найдем место в этой последовательности для второго элемента. После этого упорядочены уже первые два элемента. Далее в текущей упорядоченной последовательности ищутся места для третьего, четвертого и т. д. элементов.

Алгоритм сортировки вставками можно записать следующим образом:

```
алг Сортировка вставками нач
ввод n, X[n]
цикл для i от 1 до n
//Сохранить значение вставляемого элемента во временной переменной
   buf := X[i];
// начать просмотр элементов от j до 0
   j := i;
     // пока текущий элемент меньше предыдущего
   пока buf<X[j-1] И j>0
// заменить текущий элемент на предыдущий
          X[j] = X[j-1]
          j = j-1;
       кц
       X[j] := buf;
кц
рез X
кон
```

8.2 Многомерные массивы

8.2.1 Инициализация матриц

Рассмотрим механизм инициализации динамической двумерной матрицы. Для работы с целочисленной матрицей в программе необходимо описать указатель на указатель:

```
int **x;
```

После этого необходимо выделить память под массив указателей на строки матрицы:

```
x=(int**)malloc(sizeof(int*)*n); //массив из указателей.
```

Каждый из n указателей должен указывать на массив из m элементов

```
for( i=0;i<n;i++)
x[i] = (int)malloc(sizeof(int)*m); // массив из m элементов.
```

У матрицы, заданной таким образом, n строк и m столбцов.

Перед окончанием программы, работающей с динамической матрицей, необходимо освободить используемую память. Освобождение памяти проходит так же в два этапа, но в обратном порядке:

```
for(i=n-1;i>=0;i--)
  free(x[i]);
free(x);
```

Для перебора элементов матрицы используются не один, а два цикла:

```
for(i=0;i<n;i++)
```

```
for(j=0;j<m;j++)
     Обращение к элементу x[i][j]
```

Первый индекс элемента определяет номер строки, в которой он расположен, а второй – номер столбца.

Циклы, организованные выше, просматривают матрицу по строкам: элементы нулевой строки, элементы первой строки и т. д.

А такие циклы –

```
for(i=0;i<m;i++)
     for(j=0;j<n;j++)
          Обращение к элементу x[j][i]
```

просматривают элементы матрицы по столбцам, начиная с нулевого.

На рисунке 8.1 изображено примерное расположение матрицы в памяти.

Рис. 8.1 – Расположение элементов матрицы в памяти

Задать элементы матрицы можно такими же способами, как и элементы массива: ввести с клавиатуры, задать случайным образом, рассчитать по формуле.

Ввод элементов матрицы с клавиатуры:

```
…
float **A;
int n,m;
printf ("Введите количество строк матрицы: ");
scanf("%d",&n);
printf ("Введите количество столбцов матрицы: ");
scanf("%d",&m);
A = (int**)malloc(sizeof(float*)*n);
for(int i=0;i<n;i++)
 A[i] = (int)malloc(sizeof(float)*m);
for(i=0;i<n;i++)
```

```
 for( j=0;j<m;j++)
  { printf("A[%d][%d]= ",i,j);
   scanf("%f",&(A[i][j]));
   }
…
```

Задание элементов матрицы случайным образом (`n` – количество строк матрицы, `m` – количество столбцов):

```
…
float **x;
int n,m;
…
x = (int**)malloc(sizeof(float*)*n);
for(int i=0;i<n;i++)
 x[i] =( int*)malloc(sizeof(float)*n);
for(i=0;i<n;i++)
 for( j=0;j<m;j++)
  x[i][j] = rand()%200/(rand()%100+1.);
…
```

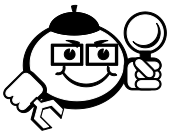
8.2.2 Печать матриц

Для вывода элементов матрицы на экран также используются два цикла. Организуем эти два цикла следующим образом: внешний цикл перебирает строки, внутренний – столбцы, и как только на экран выведены все элементы одной строки, вывод следующей нужно организовать с новой строки экрана:

```
// печать элементов вещественной матрицы x
for(i=0;i<n;i++)
 {
 for( j=0;j<m;j++)
  printf("%8.3f ",x[i][j]);
// переход на новую строку экрана
 printf("\n");
 }
```

Обратите внимание на использование формата в функции `scanf()`.

8.2.3 Примеры решений задач с использованием матриц

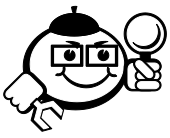


Пример 8.3

В вещественной матрице, заполненной случайными числами, найти суммы элементов строк. Количество строк и столбцов матрицы задавать с клавиатуры. Решение оформить с использованием функций.

```
#include <stdio.h>
#include <stdlib.h>
/* Функция создания вещественной матрицы по заданной размерности. Функция возвращает указатель на матрицу. */
// Параметры матрицы – количество строк и столбцов.
float ** Create(int n, int m){
int i,j;
srand();
float** x = (float**)malloc(sizeof(float*)*n);
for( i=0;i<n;i++)
 x[i] = (float)malloc(sizeof(float)*m);
for(i=0;i<n;i++)
 for(j=0;j<m;j++)
  x[i][j] = rand()%200/(rand()%100+1.)-rand()%50;
return x;
}
// Функция печати матрицы x, с n строками и m столбцами.
void Print(float **x,int n, int m){
int i,j;
for( i=0;i<n;i++)
 {
 for( j=0;j<m;j++)
  printf("%8.3f ",x[i][j]);
 printf("\n");
 }
}
// Функция, вычисляющая сумму строки с номером k в матрице x,
// с n строками m столбцами.
float SummK(float **x,int n, int m, int k){
float S = 0;
/* Проверка корректности введенных данных, функция заканчивает работу, если номер строки k больше n или является отрицательным числом. */
if (k>=n||k<0) {
```

```
                printf("Неверные данные");
                return 999;
                }
     // суммирование элементов строки с номером k.
     for (int i=0;i<m;i++)
      S+=x[k][i];
     return S;
     }
     //Функция, освобождающая память, выделенную под матрицу.
     /*Параметры функции – указатель на матрицу и количество
строк n. */
     void Delete(float ** x,int n)
     {
      for(int i=n-1;i>=0;i--)
       free(x[i]);
      free( x);
     }
     int main(int argc, char *argv[])
     {
     float **M; // описание матрицы
     int n,m,i,j;
     srand();
     printf ("Введите количество строк матрицы: ");
     scanf("%d",&n);
     printf ("Введите количество столбцов матрицы: ");
     scanf("%d",&m);
     // Создание матрицы.
     M = Create(n,m);
     // Печать матрицы.
     Print(M,n,m);
     // Вычисление суммы каждой из n строк.
     for(i=0;i<n; i++)
      {
       float Summ = SummK(M,n,m,i);
       printf("Сумма элементов в строке %d равна %8.3f \n",
i,Summ);
      }
     // Освобождение памяти.
      Delete(M,n);
      system("PAUSE");
      return 0; }
.....................................................................
```



Пример 8.4

В целочисленной квадратной матрице размерности n найти минимальный элемент, лежащий выше главной диагонали (главная диагональ исключается). Вывести на экран индексы минимального элемента. Если таких элементов несколько, то вывести индексы всех элементов. Размерность матрицы вводить с клавиатуры. Матрицу заполнять случайным образом.

Начнем решение с выделения указанной области матрицы. Главная диагональ матрицы обладает одним свойством, которое поможет нам решить поставленную задачу: элементы главной диагонали имеют одинаковые индексы (x[i][i] – элемент главной диагонали, расположенный в строке i). Поиск минимального элемента необходимо осуществлять не во всей матрице. Следовательно, необходимо изменить параметры циклов, в которых будет осуществляться поиск минимального элемента. Рассмотрим рисунок 8.2, на котором представлена квадратная матрица и выделена указанная область.

x[0][0]	**x[0][1]**	…	**x[0][i]**	…	**x[0][n-1]**
x[1][0]	x[1][1]	…	**x[1][i]**	…	**x[1][n-1]**
…	…	…	…	…	…
x[i][0]	x[i][1]	…	x[i][i]	…	**x[i][n-1]**
…	…	…	…	…	…
x[n-1][0]	x[n-1][1]	…	x[n-1][i]	…	x[n-1][n-1]

Рис. 8.2 – Квадратная матрица n×n

Для решения задачи необходимо просмотреть n−1 строк, тогда внешний цикл будет выглядеть следующим образом:

```
for(i=0;i<n-1;i++).
```

В каждой строке с номером i нужно рассмотреть элементы начиная с i+1-го и до n−1. Тогда внутренний цикл будет записан так:

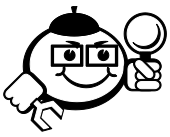
```
for(j=i+1;j<n;j++)
```

Решение задачи может выглядеть следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
// Создание квадратной матрицы размерности n.
int ** Create(int n){
int i,j;
srand();
int** x = (int**)malloc(sizeof(int*)*n);
```

```
for(i=0;i<n;i++)
 x[i] = new int[n];
for(i=0;i<n;i++)
 for( j=0;j<n;j++)
// Элементы матрицы задаются случайным образом.
  x[i][j] = rand()%10;
// Функция возвращает указатель на матрицу.
return x;
}
// Функция печати элементов матрицы.
void Print(int **x,int n){
int i,j;
for( i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
  printf("%6d ",x[i][j]);
 printf("\n");
 }
}
// Функция поиска минимального
//элемента в целочисленной матрице x размерности n.
int min(int** x, int n){
int i,j;
int minimum = x[0][1];
for( i=0;i<n-1;i++)
 for( j=i+1;j<n;j++)
  if (x[i][j]<minimum) minimum = x[i][j];
// Функция возвращает найденное минимальное значение.
return minimum;
}
// Функция, освобождающая память.
void Delete(int ** x,int n)
{int i;
 for( i=n-1;i>=0;i--)
  free(x[i]);
 free( x);
}
int main(int argc, char *argv[])
{
int **I;
int i,j;
int n,m;
srand();
```

```
printf ("Введите размерность матрицы: ");
scanf("%d",&n);
// Создание матрицы I размерности n.
I = Create(n);
// Печать матрицы.
Print(I,n);
// Нахождение минимального элемента.
m = min(I,n);
// Поиск позиций найденного минимального элемента.
printf("Минимальные элементы находятся на позициях: ");
for( i=0;i<n-1; i++)
 for( j=i+1;j<n;j++)
  if(m==I[i][j])
       printf("[%d][%d] \n", i,j);
// Освобождение памяти.
 Delete(I,n);
 system("PAUSE");
 return 0;
}
```



Пример 8.5

В целочисленной матрице размерности `n×m` упорядочить столбцы матрицы по элементам первой строки. Размерность матрицы задавать с клавиатуры. Элементы матрицы определить случайным образом. Решение оформить в виде функций.

Для упорядочивания столбцов будем использовать обменную сортировку. Сравниваться будут рядом стоящие элементы первой строки `x[0][i]` и `x[0][i+1]`. Меняться местами при этом будут не только эти элементы, а полностью столбцы с номерами `i` и `i+1`. Очевидно, что для обмена столбцов необходимо организовать цикл.

Выполним сортировку столбцов в виде функции. Параметрами функции будут указатель на матрицу и размерность матрицы. Сама функция будет возвращать указатель на измененную матрицу.

Для создания, удаления и печати матрицы воспользуемся функциями из предыдущего примера.

```
#include <stdio.h>
#include <stdlib.h>
int ** Create(int n,int m){
```

```
srand();
int i,j;
int** x = (int**)malloc(sizeof(int)*n);
for(i=0;i<n;i++)
 x[i] = (int*)malloc(sizeof(int)*m);
for(i=0;i<n;i++)
 for(j=0;j<m;j++)
  x[i][j] = rand()%10;
return x;
}
void Print(int **x,int n,int m){
int i,j;
for( i=0;i<n;i++)
 {
 for( j=0;j<m;j++)
  printf("%6d ",x[i][j]);
 printf("\n");
 } }
// Функция сортировки столбцов матрицы.
int ** Sort(int **x, int n, int m){
int flag;
int i,j,k;
for( i=0;i<m-1;i++)
{
// Обменов не было.
 flag = 0;
 for ( j=0;j<m-1-i;j++)
 {
// Сравнение двух рядом стоящих элементов первой строки.
// Если первый элемент пары больше второго
  if (x[0][j]>x[0][j+1])
  {
// зафиксируем обмен.
   flag = 1;
// выполним обмен столбцов.
   for (int k=0;k<n;k++)
    {
     int temp = x[k][j];
     x[k][j] = x[k][j+1];
     x[k][j+1] = temp;
    }
  }
 }
```

```
    // Если обменов не было, то закончить сортировку.
      if (!flag) break;
    }
    return x;
    }
    void Delete(int ** x,int n)
    {
    int i;
     for( i=n-1;i>=0;i--)
      free( x[i]);
     free(x);
    }
    int main(int argc, char *argv[])
    {
    int **I;
    int n,m;
    system("chcp 1251");
    printf ("Введите количество строк матрицы: ");
    scanf("%d",&n);
    printf ("Введите количество столбцов матрицы: ");
    scanf("%d",&m);
    I = Create(n,m);
    printf("Исходная матрица: \n ");
    Print(I,n,m);
    printf("Преобразованная матрица: \n ");
    // Вызов функции сортировки.
    I = Sort(I,n,m);
    Print(I,n,m);
    Delete(I,n);
    system("PAUSE");
    return 0;
    }
```

8.3 Строки

8.3.1 Инициализация строк

Язык Си имеет очень мощный механизм для работы со строками. Специального типа для представления строковых данных в Си нет, строка – это массив символов. Для описания строк можно использовать массив символов с постоянной длиной:

```
char str[25]; // строка из 24 символов.
```

Или массив символов с динамическим выделением памяти:

```
// строка из 24 символов
char *str = (char*)malloc(sizeof(char)*25);
```

Задать строку данных можно различными способами:

- задать с помощью строковой константы: `char str[25] = "Пример строки";`
- прочитать с клавиатуры с помощью функции `scanf()`: `scanf("%s",str);`. Обратите внимание, строка символов – массив, а любой массив в Си – это адрес, поэтому перед переменной `str` не ставится знак `&;`, функция `scanf()` читает строку до первого встреченного пробела; остальная часть строки обрезается;
- прочитать с клавиатуры с помощью функции `gets()` – `str = gets(str);`. В случае успешного чтения функция возвращает прочитанную строку, в противном случае – `null;`. Функция может читать произвольную строку, однако надо помнить, что с клавиатуры можно ввести не более 160 символов;
- задать строку посимвольно – `str[i] = 'A';`.

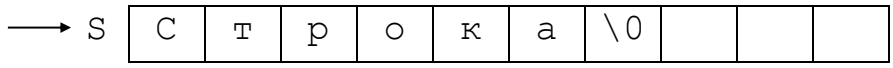
8.3.2 Представление строки в памяти компьютера

Строка Си может состоять из произвольного количества символов. Окончанием любой строки считается так называемый символ «нуль-терминатор» – «`/0`».

Пусть в программе выполнены следующие действия:

```
char* S = (char*)malloc(sizeof(char)*10);
strcpy(S, "Строка");
```

Представление такой строки в памяти выглядит следующим образом:



После «`\0`» в памяти могут быть расположены любые символы, но каждое обращение к строке подобно просмотру памяти, начиная с адреса `S`, пока не будет встречен «`\0`». Символ «`\0`» считается при выделении памяти под строку. Поэтому строка `S` может содержать только 9 символов. Если выполнена попытка записать в выделенную память более чем 9 символов, работа программы может стать непредсказуемой. Такие ошибки работы со строками наиболее часты.

Стандартные функции считывания строки автоматически добавляют символ окончания строки. Если же Вы формируете строку посимвольно, не забудьте записать в последнюю позицию строки символ «\0».

При работе со строками всегда нужно помнить, что имя строки – указатель, поэтому присваивание типа:

```
char *my = (char*)malloc(sizeof(char)*15);
char* z = (char*)malloc(sizeof(char)*15);
strcpy(my, "Hello");
strcpy(z, "world");
my = z;
```

приведет к тому, что и адрес `z`, и адрес `my` будут указывать на строку `"world"`. Если Вы хотите, чтобы в памяти появилась две копии строки `"world"`, необходимо выполнить копирование строки с помощью функции `strcpy()`.

В Си нельзя сравнивать и складывать строки с помощью стандартных арифметических операций. Для этих действий также используются специальные функции.

8.3.3 Стандартные функции для работы со строками

Прототипы ниже описанных функций находятся в заголовочном файле `string.h`.

`char *strcat(char *dest, const char *src);` – присоединяет копию строки `src` к концу строки `dest`. Длина полученной строки равна `strlen(dest) + strlen(src)`. Возвращает указатель на объединенную строку.

`char *strchr(const char *s, int c);`– просматривает строку `s` в прямом направлении в поисках заданного символа `c`. Ищет *первое* вхождение символа `c` в строку `s`. Возвращает указатель на первое вхождение символа `c` в строку `s`, если `c` не входит в строку `s`, функция возвращает `null`.

`int strcmp(const char *s1, const char *s2);` – производит беззнаковое сравнение символов строк `s1` и `s2`, начиная с первого символа каждой строки и продолжая делать то же с последующими, пока не встретятся два соответствующих не совпадающих символа или не будет достигнут конец данных строк. Возвращает следующие значения: отрицательное число, если `s1`

меньше s2; 0, если s1 совпадает с s2; положительное число, если s1 больше s2.

int strcmpi(const char *s1, const char *s2); – выполняет беззнаковое сравнение s1 и s2. Возвращает следующие значения: отрицательное число, если s1 меньше s2; 0, если s1 совпадает с s2; положительное число, если s1 больше s2.

char *strcpy(char *dest, const char *src);– копирует строку src в строку dest. Копирование завершается после достижения нуль-терминатора. Возвращает dest.

size_t strcspn(const char *s1, const char *s2); – возвращает длину начального сегмента строки s1, который полностью состоит из символов, не встречающихся в s2.

size_t strlen(const char *s);– вычисляет длину строки s. Возвращает количество символов в строке s, не считая нуль-терминатора.

char *strlwr(char *s); – преобразует прописные буквы (от A до Z) в строчные (от a до z). Никакие другие символы не изменяются. Возвращает указатель на строку s.

char *strncat(char *dest, const char *src, size_t maxlen); – копирует не более maxlen символов из src в конец dest и добавляет нуль-терминатор. Максимальная длина полученной строки равна strlen(dest)+maxlen. Возвращает указатель на dest.

int strncmp(const char *s1, const char *s2, size_t maxlen);– выполняет беззнаковое сравнение, проверяя не более maxlen символов. Она начинает сравнение с первого символа каждой строки и продолжает то же с последующими, пока не встретятся два соответствующих не совпадающих символа или не будет проверено maxlen символов. Возвращает следующие значения: отрицательное число, если s1 меньше s2; 0, если s1 совпадает с s2; положительное число, если s1 больше s2.

char *strncpy(char *dest, const char *src, size_t maxlen); – копирует не более maxlen символов из src в dest, усекая dest или дополняя нуль-терминаторами. Строка dest может не заканчиваться нуль-терминатором, если длина src больше или равна maxlen. Возвращает указатель на dest.

`char *strnset(char *s, int ch, size_t n);`– помещает символ `ch` в первые `n` байтов строки `s`. Если `n > strlen(s)`, вместо `n` берется значение `strlen(s)`. Функция завершается, если `n` символов уже заполнены или обнаружен нуль-терминатор. Возвращает `s`.

`char *strpbrk(const char *s1, const char *s2);` – просматривает строку `s1`, пока не встретит вхождение любого символа из `s2`. Возвращает указатель на первое вхождение любого из символов, содержащихся в `s2`. Если ни один из символов `s2` не содержится в `s1`, функция возвращает `null`.

`char *strrchr(const char *s, int c);` – в поисках указанного символа `strrchr` просматривает строку в обратном направлении. Эта функция ищет последнее вхождение символа `c` в строку `s`. Возвращает указатель на последнее вхождение символа `c`. Если `c` не содержится в `s`, возвращается `null`.

`char *strrev(char *s);` – изменяет порядок следования символов в строке на обратный, за исключением нуль-терминатора. Возвращает указатель на реверсированную строку.

`char *strset(char *s, int ch);`– заполняет всю строку `s` символом `ch`. Она завершает работу при достижении нуль-терминатора. Возвращает указатель на строку `s`.

`size_t strspn(const char *s1, const char *s2);` – `strspn` ищет начальную подстроку строки `s1`, целиком состоящую из символов, содержащихся в строке `s2`. Возвращает длину начальной подстроки `s1`, состоящей только из символов, содержащихся в строке `s2`.

`char *strstr(const char *s1, const char *s2);` – просматривает строку `s1` до первого обнаружения вхождения подстроки `s2`. Возвращает указатель на первый символ первого вхождения `s2` в `s1`. Если подстроки `s2` в `s1` не содержится, `strstr` возвращает `null`.

`double strtod(const char *s, char **endptr);` – преобразует символьную строку `s` в значение типа `double`. `s` есть последовательность символов, которая может быть интерпретирована как значение типа `double`; она должна иметь следующий обобщенный формат:

`[ws] [sn] [ddd] [.] [ddd] [fmt [sn] ddd]`

`[ws]` – необязательные пробелы;

`[sn]` – необязательный знак (+ или –);

[ddd] – необязательные цифры;

[fmt] – необязательный символ e или E;

[.] – необязательная десятичная точка.

Функция прекращает работу при обнаружении первого символа, который не может быть интерпретирован как соответствующая часть значения типа double. Возвращает значение типа double, полученное из строки s.

char *strtok(char *s1, const char *s2); – предполагает, что строка s1 состоит из последовательности из нуля или более лексических единиц, разделенных одним или несколькими символами из строки разделителей s2. Первое обращение к strtok возвращает указатель на первый символ первой лексической единицы в s1 и записывает нуль-терминатор в s1 непосредственно за этой лексической единицей. Последующие обращения с первым аргументом, установленным в нуль, будут продолжать просматривать строку s1, пока не исчерпаются все содержащиеся в ней лексические единицы. Возвращает указатель на лексическую единицу, обнаруженную в s1. Нулевой указатель возвращается, если таковых больше нет.

long strtol(const char *s, char **endptr, int radix); – преобразует символьную строку s в значение типа long. Строка s есть последовательность символов, которая может быть интерпретирована как значение типа long; она должна иметь следующий обобщенный формат:

[ws] [sn] [0] [x] [ddd]

[ws] – необязательные пробелы;

[sn] – необязательный знак (+ или –);

[0] – необязательный ноль (0);

[x] – необязательный символ x или X;

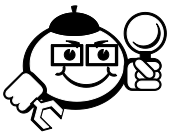
[ddd] – необязательные цифры.

Возвращает значение преобразованной строки или 0 в случае ошибки.

unsigned long strtoul(const char *s, char **endptr, int radix); – работает так же, как и strtol, за исключением того, что строка s преобразуется в значение типа unsigned long, в то время как strtol преобразует в значение типа long. Возвращает преобразованное значение типа unsigned long или 0 в случае ошибки.

`char *strupr(char *s);` – преобразует строчные буквы (a–z), содержащиеся в строке `s`, в прописные (A–Z). Никакие другие символы не изменяются. Возвращает указатель на строку `s`.

8.3.4 Примеры решений задач со строками

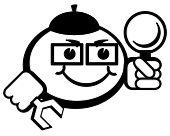


Пример 8.6

Рассмотрим пример, формирующий строку `words` по следующему правилу: дана произвольная строка символов `dest`, в строку `words` записать все первые символы слов строки. Словом считается последовательность символов, ограниченная пробелами или знаками препинания и не имеющая пробелов внутри себя.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
// Строка для ввода с клавиатуры
 char dest[200];
 printf("Вводите строку символов: ");
// Ввод строки
 gets(dest);
// Используем для работы вспомогательную строку buf.
// Выделение памяти под вспомогательную строку
 char *buf = (char*)malloc(sizeof(char*)*([strlen(dest)+1));
// Копируем строку dest в строку buf
 strcpy(buf,dest);
 char *words;
// Подсчет количества слов
 int k = 0;
// Выделить первое слово
 char *temp = strtok(buf," ,;.!?-");
// Выделение следующих слов
 while (temp!=NULL)
 {
 temp = strtok(NULL," ,;.!?-");
 k++;
 }
// Если строка содержит хотя бы одно слово
 if (k){
```

```
     /* выделить память под строку, которая будет хранить началь-
ные символы слов */
      words = (char*)malloc(sizeof(char)*(k+1));
      k = 0;
     // провести выделение слов вновь
      temp = strtok(dest," ,;.!?-");
     /* сохраняя первый символ каждого выделенного слова в строке
words. */
      words[k++] = temp[0];
      while (temp!=NULL)
      {
      temp = strtok(NULL," ,;.!?-");
      if (temp)
      words[k++] = temp[0];
      }
     /* последним символом полученной строки записать нуль-
терминатор */
      words[k] = '\0';
     // вывести строку на экран
      puts(words);
      }
     system("PAUSE");
     return 0;
     }
```

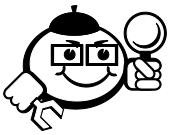


Пример 8.7

Дана произвольная строка символов, найти слово с максимальной длиной и вывести его на экран.

```
     #include <stdio.h>
     #include <stdlib.h>
     int main(int argc, char *argv[]){
      char dest[200];
     printf("Вводите строку символов: ");
      gets(dest);
      char *buf = (char*)malloc(sizeof(char)*(strlen(dest)+1));
      strcpy(buf,dest);
     // Выделить первое слово строки
      char *temp = strtok(buf," ,;.!?-");
     // Принять это слово за слово с максимальной длиной
      int max = strlen(temp);
```

```
 char *strmax = ( char*)malloc(sizeof(char)*(max+1));
 strcpy(strmax,temp);
// Выделять последующие слова и сравнивать их со
// словом с максимальной длиной
 while (temp!=NULL)
 {
 temp = strtok(NULL," ,;.!?-");
 if (!temp)break;
 if (max<strlen(temp))
  {
     free( strmax);
     max = strlen(temp);
     strmax = (char*)malloc(sizeof(char)*(max+1));
     strcpy(strmax,temp);
  } }
 printf("В строке %s\n слово с максимальной длиной %s", dest,
strmax);
 printf("\n Его длина - %d",max);
 free( strmax);
 free( buf);
system("PAUSE");
return 0;
 }
```



Пример 8.8

В произвольной строке символов найти количество символов, не являющихся буквами.

Для решения этой задачи воспользуемся функцией `isalpha()`, которая передает ненулевое значение, если проверяемый символ является буквой и нуль в противном случае. Прототип функции описан в заголовочном файле `ctype.h`.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
 char dest[200];
 printf("Вводите строку символов: ");
 gets(dest);
 int k = 0,i;
 for(i=0;i<strlen(dest);i++)
```

```
      { if(!isalpha(dest[i])) k++; }
      printf("Количество символов строки, не являющихся буквами -
%d\n",k);
     system("PAUSE");
     return 0;
     }
```

Контрольные вопросы по главе 8

1. Перечислите способы инициализации одномерных массивов.
2. Запишите фрагмент программы, выделяющий память под массив из `m` целочисленных элементов.
3. Что будет выведено на экран при выполнении следующего фрагмента программы? Массив `X` задан и его элементы перечислены ниже:

   ```
   X = {0,3,2,1,5,4}
   int n = 6;
   for(int i=0;i<n; i++)
   if (x[i]%2==0) printf("%d \n",x[i]);
   ```

4. Запишите фрагмент программы, который ищет минимальный элемент среди положительных элементов массива `X`. Размерность массива равна `n`.
5. Напишите фрагмент программы, который считает количество положительных элементов в матрице `X[n×m]`; элементы матрицы целые числа.
6. Запишите фрагмент программы, выделяющий память под вещественную матрицу размерности `n×m`.
7. Напишите функцию, которая заменяет все положительные элементы в матрице `X[n×m]` на номер строки, в которой расположен этот элемент; элементы матрицы целые числа.
8. Напишите фрагмент программы, который заменяет элементы побочной диагонали в матрице `X[n×n]` на 1; элементы матрицы целые числа.
9. Сортировка обменом выполнила два шага алгоритма (за один шаг один элемент встает на свое место) на предложенном массиве. Сколько сравнений произойдет на третьем шаге? Массив: 9 1 3 2 7 5 9 9.

10. Сортировка выбором выполнила два шага алгоритма (за один шаг один элемент встает на свое место) на предложенном массиве. Сколько сравнений произойдет на третьем шаге? Массив: 1 3 4 4 5 4 7 3.
11. Сортировка вставками выполнила три шага алгоритма (за один шаг один элемент встает на свое место) на предложенном массиве. Сколько сравнений произойдет на третьем шаге? Массив: 1 3 4 1 5 4 7 1.
12. Напишите функцию, подсчитывающую количество вхождений заданного символа в строку. Параметры функции – символ для поиска, строка символов.
13. Напишите функцию, подсчитывающую количество слов строки, начинающихся с заданного символа.
14. Напишите функцию, определяющую, встречается ли в строке заданный символ.
15. Напишите функцию, определяющую, встречаются ли в строке повторяющиеся символы.

9 Файлы в Си

9.1 Типы файлов в Си

Библиотечные функции для работы с файлами можно разделить на две группы – ***префиксные*** и ***потоковые***. Для каждой открытой программы операционная система формирует уникальную таблицу открытых файлов, каждый файл имеет свой собственный номер – префикс. Первые 4 префикса зарезервированы под устройства стандартного ввода-вывода – `stdin` – клавиатура, `stdout` – экран, `stderr` – экран, `stdaux` – порт *COM1*, `stdprn` – принтер. Если в программе открывается файл, то операционная система автоматически присваивает ему первый свободный номер, этот номер будет префиксом файла.

Для каждой из групп файлов установлены два режима доступа к файлу – двоичный и текстовый. При текстовом режиме доступа символы `0DH 0AH` (перевод каретки) преобразуются в один символ «`\n`», соответственно при записи символ перевода каретки преобразуется в пару символов. При считывании из файла в текстовом режиме чтение символа `1AH` заканчивает считывание информации из файла (символ конца файла).

При двоичном доступе к файлу каждый символ считывается отдельно, как не имеющий никакого смысла. Режим доступа к файлу задается непосредственно при использовании библиотечной функции открытия или через переменную `_fmode (stdio.h)`, которая по умолчанию установлена в `O_TEXT`, для двоичного доступа – `O_BINARY`.

Функции поточного ввода-вывода называют стандартными функциями ввода-вывода. Си создает внутреннюю структурную переменную по шаблону `FILE` (описание находится в `stdio.h`).

9.2 Механизм чтения-записи

Файл *– это поименованная область на жестком диске, заполненная какой-либо информацией.*

В конце каждого файла записывается символ окончания файла, который помогает операционной системе корректно работать с файлами.

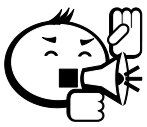
Различают понятия «открыть файл для записи» и «открыть файл для чтения». При первом способе открытия файла вся имеющаяся в нем информация стирается, и Вы можете записать в него новую информацию. При открытии файла создается переменная, которая содержит в себе адрес памяти, где записан просматриваемый файл. При записи этот указатель меняет свое положение – передвигается по файлу. Как только произошла запись, указатель содержит адрес, по которому, возможно, произойдет следующая запись.

Аналогичен и механизм чтения информации из файла. В этом случае указатель переходит к следующей позиции после считывания информации. Таким образом, если произошло чтение символа, то указатель передвинется в файле на 1 байт, если Вы считываете строку – на размер строки и т. д. В двоичном файле передвижение проходит на размер считываемого элемента. Например, при считывании целочисленных данных указатель чтения-записи передвигается на 4 байта, при считывании данных типа `float` – на 4 байта.

9.3 Функции для поточного доступа к файлам

`FILE* fopen (const char *filename, const char* mode);` – возвращает указатель на переменную типа `FILE`, `mode` – заданный режим открытия файла. При неуспешной работе функция возвращает `NULL`. Во избежание ошибок при открытии файла необходимо проверять результат выполнения функции, например, как в следующей программе:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 FILE *f;
 char name[] = "prim.txt";
if ((f = fopen(name,"rb"))==NULL) {
  printf( "Ошибка открытия файла: ");
system("pause");
  }
  else…
…
}
```



Обратите внимание: очень распространенная ошибка – при указании полного имени файла символы «\» строке имени файла должны удваиваться (например: `n:\\c++\\file.txt`).

Если Вы не указываете полное имя, то файл ищется (создается) в папке проекта.

Режимы доступа к файлу:

`r` – открыть для чтения.

`w` – создать для записи. Если файл с таким именем уже существует, он будет перезаписан.

`a` – открыть файл для обновления, открывает файл для записи в конец файла или создает файл для записи, если файла не существует.

`r+` – открыть существующий файл для корректировки (чтения и записи).

`w+`– создать новый файл для корректировки (чтения и записи). Если файл с таким именем уже существует, он будет перезаписан.

`a+` – открыть для обновления; открывает для корректировки (чтения и записи) в конец файла, или создает, если файла не существует.

`t` – открыть файл в текстовом режиме.

`b` – открыть файл в двоичном режиме.

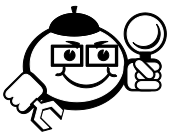
Два эти аргумента указываются вторыми символами в строковой переменной `mode`, например `r+b`.

По умолчанию установлен текстовый режим доступа к файлу.

`int fclose(FILE *fp)` – закрывает файл `fp`, при успешной работе возвращает 0, при неуспешной – `EOF`).

`int closeall(void)` – закрывает все файлы, открытые в программе, при успешной работе возвращает число закрытых потоков, при неуспешной – `EOF`.

`FILE *freopen(const char *filename, const char* mode, FILE *stream)` – закрывает поток `stream`, открывает поток `filename` с новыми правами доступа, установленными в `mode`. Если потоки разные, то происходит переадресация потока `stream` в поток `filename`.



Пример 9.1

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 char name[] = "prim.txt";
 int n;
 FILE *f;
 printf("Введите переменную n: ");
scanf("%d", & n);
freopen(name,"wt",stdout);
// переопределить стандартное устройство вывода – экран
  for (int i=0;i<n;i++)
      printf("%d\n",i);
// печать будет осуществляться в текстовый файл с именем
// prim.txt
printf("Press any button...");
system("PAUSE");
 return 0;
}
```

ch = fgetc(<указатель на файл>) – возвращает символ ch из файла, с которым связан указатель.

ch = getc(<указатель на файл>) – возвращает символ ch из файла, с которым связан указатель.

fputc(ch, <указатель на файл>) – записать символ ch в указанный файл.

putc(ch, <указатель на файл>) – записать символа ch в файл.

fgets(str, n, <указатель на файл>) – прочитать строку str длиной n символов или до первого встреченного \n из файла.

fputs(str, <указатель на файл>) – записать строку str в файл. Символ перевода на другую строку в файл не записывается.

fscanf(<указатель на файл>, управляющая строка, ссылка) – универсальная функция считывания из текстового файла. Например, fscanf(f, "%d",&n) считывает из файла f целое число в переменную n.

`fread(ptr,size,n,<указатель на файл>)` – считывает `n` элементов размером `size` в область памяти, начиная с `ptr`. В случае успеха возвращает количество считанных элементов, в случае неуспеха – `EOF`.

`fwrite(ptr,size,n,<указатель на файл>)` – записывает `n` элементов размером `size` из памяти, начиная с `ptr`, в файл. В случае успеха возвращает количество записанных элементов, в случае неуспеха – `EOF`.

`fprintf(<указатель на файл>, управляющая строка, [список аргументов])` – форматированный вывод в файл.

В Си к любому файлу может быть осуществлен прямой доступ. Для прямого доступа используются следующие функции:

`rewind(<указатель файла>)` – установить указатель файла на начало файла.

`int fseek(<указатель файла>, offset, fromwheare)` – установить указатель чтения-записи файла на позицию `offset`, относительно позиции `fromwheare`. `fromwheare` может принимать значения `SEEK_END` – от конца файла, `SEEK_SET` – от начала файла, `SEEK_CUR` – от текущей позиции.

`long int n = ftell(<указатель на файл>)` – в переменную `n` передать номер текущей позиции в файле.

`int z = fgetpos(<указатель файла>, npos);` – в динамической памяти по адресу `npos` записать номер текущей позиции в файле, в случае успеха функция возвращает `0`; в противном случае – любое ненулевое число.

`int unlink(<имя файла>)` – удаление файла, при успехе функция возвращает `0`, при неуспехе – `–1`.

`int rename(<старое имя>,<новое имя>)` – переименование файла, при успехе функция возвращает 0, при неуспехе – `–1`.

`int feof(<указатель на файл>)` возвращает `0`, если конец файла не достигнут, любое ненулевое число, если достигнут.

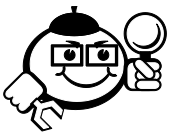
`int ferror(<указатель на файл>)` возвращает ненулевое значение, если при работе с файлом возникла ошибка, `0` – в противном случае.

В Си описана группа функций для управления работы с файлами – проверка на существование файла или папки, поиск файлов и др.

9.4 Примеры работы с текстовыми файлами

9.4.1 Запись данных в текстовый файл

В текстовый файл можно записать данные различных типов. В дальнейшем содержимое такого файла можно просмотреть в любом текстовом редакторе.

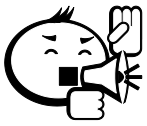


Пример 9.2

Создать вещественный массив случайным образом и сохранить его в текстовом файле.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
 char name[25];
 int n;
 FILE *f;
 int flag = 1,i;
// Создадим цикл, позволяющий корректировать ввод имени файла
 do
 {
 printf( "Введите имя создаваемого файла: ");
 scanf("%s",name);
// Попытка открыть файл для чтения. Если такой файл уже
// существует, то задать вопрос пользователю
 if ((f = fopen(name,"r"))!=NULL)
 {
// Заменить существующий файл?
     printf("Файл уже существует. Заменить? (y/n)");
     char ch = getch();
//Если пользователь нажал кнопку «n», вернуться к началу цикла
     if (ch == 'n') { continue;}
 }
/* В этот блок программы управление попадет, только если
пользователь подтвердил замену или задал имя несуществующего файла.*/
// Создать файл.
     if ((f=fopen(name,"w"))==NULL)
      {
      printf("Ошибка создания файла");
```

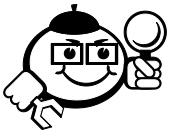
```
        system("pause");
        break;
        }
      printf("\nВведите размерность массива: ");
      scanf("%d",&n);
      for( i=0;i<n;i++ )
       { float y = rand()%100/(rand()%50+1.)-rand()%30;
// На одной строке файла печатать только 10 элементов.
        if (i>=10&&i%10==0) fprintf(f,"\n");
        fprintf(f,"%8.3f",y);
        }
// Закрыть файл.
      fclose(f);
// Закончить цикл
      flag = 0;
 } while (flag);
 printf("Файл создан");
 system("PAUSE");
 return 0;
 }
```



Обратите внимание: в программе не использовался массив. Так как данные сохраняются в файле, для решения задачи достаточно одной целочисленной переменной.

9.4.2 Чтение данных из текстового файла

Предположим, в текущем каталоге существует файл `data.txt`.



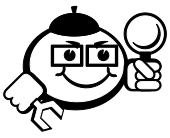
Пример 9.3

На первой строке в файле записана размерность целочисленной матрицы. Далее – сама матрица. Считать матрицу в память и вывести ее на экран. Данные записаны в файле `my.txt`. Создать такой файл можно, например, в блокноте или любом другом текстовом редакторе. Если не указан полный путь к файлу, то файл должен находиться в папке проекта.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
FILE *f;
f = fopen("my.txt", "r");
// Проверка ошибки открытия файла
if (f==NULL) {
printf("Файл не найден... /n ");
system(“pause”);
return 0;
}
int n,m,i,j;
// Чтение размерности матрицы
fscanf(f,"%d",&n);
fscanf(f,"%d",&m);
int **a;
// Выделение памяти под матрицу
a = (int**)malloc(sizeof(int*)*n);
for( i=0;i<n;i++)
 a[i] = (int*)malloc(sizeof(int)*m);
// Чтение матрицы
for(i=0;i<n;i++)
 for( j=0;j<m;j++)
  fscanf(f,"%d",&a[i][j]);
printf("Прочитана матрица: \n");
// Печать элементов матрицы
for(i=0;i<n;i++)
{
 for( j=0;j<m;j++)
   printf("%5d",a[i][j]);
 printf("\n");
}
system("PAUSE");
 return 0;
}
```

В предыдущем примере размерность матрицы считывалась из файла. Но можно организовать считывание элементов и без заданной размерности. В этом случае файл сканируется по условию, пока не найден конец файла, одновременно ведется подсчет считанных элементов.



Пример 9.4

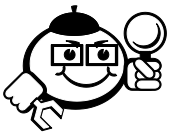
В текстовом файле записано произвольное количество чисел. Считать данные из файла в массив и вывести на экран. Файл `my.txt` находится в папке проекта.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
FILE *f;
f = fopen("my.txt", "r");
// Проверка ошибки открытия файла
if (f==NULL) {
printf("Файл не найден....");
system("PAUSE");
 return 0;
}
int n=0,y;
int *a;
// пока не конец файла f
while (!feof(f))
{
// читать элемент и
int z = fscanf(f,"%d",&y);
// если произошла ошибка при считывании, то закончить
// выполнение цикла
if(z!=1) break;
// увеличивать счетчик.
 n++;
}
// После окончания цикла в переменной n хранится
// количество целых чисел, записанных в файле.
// Выделить память под массив.
a = ( int*)malloc(sizeof(int) *n);
// Указатель чтения-записи файла передвинуть в начало.
fseek(f,0,SEEK_SET);
// Читать n целых чисел из файла в массив.
for( i=0;i<n;i++)
  fscanf(f,"%d",&a[i]);
printf("Прочитан массив: \n");
for(i=0;i<n;i++)
```

```
      printf("%5d",a[i]);
   system("PAUSE");
    return 0;}
```

9.4.3 Изменение текстового файла

При решении некоторых задач не требуется считывать все данные из файла в оперативную память. Си позволяет выполнять изменения непосредственно в файле, используя механизм прямого доступа.



Пример 9.5

В текстовом файле расположен произвольный текст. Не считывая весь текст в память, изменить все первые буквы слов на прописные.

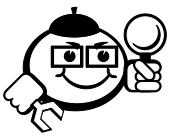
Для решения этой задачи будем использовать свойство функции `fscanf` – функция читает строки до первого встреченного пробела. Поэтому если организовать цикл по условию, пока не найден конец файла, и в теле цикла использовать функцию `scanf` для чтения строковых данных, то на каждом шаге цикла будет считываться ровно одно слово.

В прочитанном слове изменим первый символ на прописной, используя функцию `toupper(char ch)`, функция преобразует символ `ch` в прописной, если это возможно. Результат работы функции – измененный символ.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
/* Откроем файл для чтения с дополнением. Символ окончания
файла в этом случае автоматически удаляется. */
FILE *f = fopen("text.txt","r+");
if (f==NULL) {
          printf("Файл не найден. \n");
          system("PAUSE");
           return 0;
          }
char word[100];
long pos1;
// Организуем бесконечный цикл для чтения файла.
while (1){
```

```
     /* Если прошло неуспешно, значит достигнут конец файла, в
этом случае нужно закончить выполнение цикла */
      if ( fscanf(f,"%s",word)!=1) break;
      /* В переменную pos1 получить текущую позицию указателя чте-
ния-записи. */
     pos1 = ftell(f);
     /* Установить указатель на позицию, с которой было считано
слово. */
      fseek(f,pos1-strlen(word),SEEK_SET);
      word[0] = toupper(word[0]);
      printf(" %s\n",word);
     // Записать в файл измененное слово.
      fprintf(f,"%s",word);
     // Установить указатель чтения-записи на позицию,
     // находящуюся после измененного слова.
      fseek(f,pos1,SEEK_SET);
     }
     fclose(f);
     return 0;
     }
```

Рассмотрим еще один пример работы с текстовыми файлами.



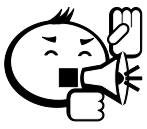
Пример 9.6

В текстовом файле записано произвольное количество целых чисел. Не считывая всю информацию в память, найти максимальное число и дописать в файл строку, содержащую найденное значение и его порядковый номер в файле. Гарантируется, что в файле записано хотя бы одно число.

Решение этой задачи может выглядеть следующим образом. Откроем файл для чтения. Прочитаем первое число из файла и примем его за максимальное число, положив номер максимального числа, равный 1. Так как количество чисел в файле произвольно, организуем цикл чтения, пока не будет достигнут конец файла. В этом же цикле организуем поиск максимального с запоминанием его номера. Откроем в программе еще один поток, указывающий на данный файл, но с атрибутом дополнения «a». Во второй поток выведем информацию о найденном максимальном значении. Реализация решения этого задания представлена ниже.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
system("chcp 1251");
// Поток f открыт для чтения, f1 – для дополнения
FILE *f = fopen("text.txt","r");
FILE *f1 = fopen("text.txt","a");
if (f==NULL) {
          printf("Файл не найден. \n");
          system("PAUSE");
           return 0;
          }
int max, nmax = 1,a,i=1;
// Чтение первого числа
fscanf(f,"%d",&max);
// Чтение всех остальных чисел
while (!feof(f)){
 if ( fscanf(f,"%d",&a)!=1) break;
 i++;
 if (max<a) {max = a; nmax = i;}
}
// Вывод информации во второй поток
fprintf(f1,"\nМаксимальное число - %d[%d]\n",max,nmax);
fclose(f);
fclose(f1);
system("pause");
return 0;
}
```

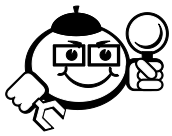


Обратите внимание: если для решения задачи требуется одновременно читать информацию из файла и дополнять ее, можно связывать один файл с двумя потоками, один из которых открыт для чтения, а второй – для дополнения. В этом случае создаются два указателя на файл, один из которых указывает на позицию, с которой будет происходить следующая операция считывания, а второй указывает на позицию, с которой будет происходить запись.

9.5 Двоичные файлы

9.5.1 Запись и чтение информации в двоичный файл

Рассмотрим сохранение и последующее чтение числовой информации в двоичном представлении.



Пример 9.7

Записать в двоичный файл n вещественных чисел, прочитать созданный файл и вывести на экран в виде матрицы с числом столбцов m. Решение оформить в виде функций.

Напишем функцию, создающую двоичный файл. Функция не будет возвращать значений, параметрами функции будут имя создаваемого файла и количество записываемых чисел:

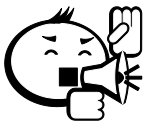
```
void create_file(char * name, int n);
```

Функция чтения также не будет возвращать значений, а параметрами функции будут имя читаемого файла и количество столбцов выводимой информации:

```
void read_file(char* name, int m);

#include <stdio.h>
#include <stdlib.h>
void create_file(char *name, int n)
{
  int i;
 FILE *f = fopen(name,"wb");
 if (f==NULL) {
          printf("Ошибка создания файла. \n");
          system("pause");
          return ;       }
// Запись n чисел в файл
 for( i=0;i<n;i++)
 {
  float z = rand()%200/(rand()%100+1.)-rand()%70;
  fwrite(&z,sizeof(float),1,f);
  }
  fclose(f); }
void read_file(char *name, int m)
{
```

```
 FILE *f = fopen(name,"rb");
 if (f==NULL) {
          printf("Файл не найден. \n");
          system("pause");
          return ;
          }
//Переменная для подсчета количества уже выведенных значений.
 int i = 0;
 float z;
// Пока не конец файла
 while(!feof(f))
 {
// если количество выведенных элементов делится без
// остатка на заданное количество столбцов,
// перейти на следующую строку экрана.
  if (i>=m&&i%m==0)
    printf("\n");
  if(fread(&z,sizeof(float),1,f)!=1) break;
  printf("%8.3f",z);
  i++;
  }
  fclose(f);
}
int main(int argc, char *argv[])
{
char Fname[30];
int n,m;
printf("Введите имя создаваемого файла: ");
scanf("%s",Fname);
printf("Введите количество записываемых чисел: ");
scanf("%d",&n);
printf("Введите количество столбцов: ");
scanf("%d",&m);
create_file(Fname,n);
read_file(Fname,m);
system("pause");
return 0; }
```



Обратите внимание: значение функции `feof` формируется только при попытке чтения из файла, поэтому внутри цикла выполнена проверка значения функции `fread`:

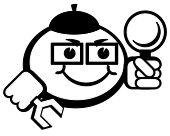
```
if(fread(&z,sizeof(float),1,f)!=1) break;
```

Если чтение на данном шаге проведено неуспешно (найден конец файла), то необходимо закончить работу цикла.

. . . . . .

9.5.2 Реализация прямого доступа в двоичном файле

В некоторых задачах требуется читать только указанную информацию. Механизм работы с файлами в Си позволяет обращаться к элементам, записанным на заданных позициях файла, без чтения предыдущих элементов.

. . . . .  Пример 9.8 . . . . . .

В двоичном файле сохранена следующая информация – размерность `n` квадратной матрицы и сама вещественная матрица. Вывести на экран элементы заданного столбца `k`.

В двоичном файле элементы матрицы сохранены следующим образом:

```
x[0][0],x[0][1],…x[0][n],x[1][0],…,
   x[n-1][n-2],x[n-1][n-1].
```

Очевидно, что позиция `k`-го элемента рассчитывается по формуле:

```
k*sizeof(элемента)+i*sizeof(элемента),
```

где `i` – номер строки.

На рисунке 9.1 представлено расположение в двоичном файле целочисленной матрицы с `n=5`. Нумерация позиций в файле начинается с нуля. Число типа `int` занимает в памяти 4 байта. Поэтому элемент матрицы, находящийся в нулевом столбце и нулевой строке, сохранен в файле с позиции с номером 0, элемент с индексами 0 и 1 – с позиции 4, элемент с индексами `i` и `j` – с позиции `j*4+i*4`.

0	4	8	12	16
20	24	28	32	36
40	44	48	52	56
60	64	68	72	76
80	84	88	92	98

Рис. 9.1 – Расположение элементов в двоичном файле

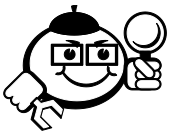
Воспользуемся этой закономерностью для решения задачи:

```
#include <stdio.h>
#include <stdlib.h>
```

```
/* Создание двоичного файла с именем name, содержащем n*n ве-
щественных чисел. */
void create_file(char *name, int n)
{
 FILE *f = fopen(name,"wb");
 if (f==NULL) {
          printf("Ошибка создания файла. \n");
          system("pause");
          return ;
          }
// Запишем в файл переменную n.
 fwrite(&n,sizeof(n),1,f);
int i;
// Запишем в файл элементы квадратной матрицы.
 for( i=0;i<n*n;i++)
 {
  float z = rand()%200/(rand()%100+1.)-rand()%70;
  fwrite(&z,sizeof(float),1,f);
  }
  fclose(f);
}
// Функция чтения файла с именем name.
void read_file(char *name)
{
 int m;
 FILE *f = fopen(name,"rb");
 if (f==NULL) {
          printf("Файл не найден. \n");
          system("pause");
          return ;
          }
 int i = 0;
// Чтение размерности матрицы.
 float z;
 fread(&m,sizeof(int),1,f);
 // Чтение всей матрицы и вывод ее на экран.
 while(!feof(f))
 {
  if (i>=m&&i%m==0)
    printf("\n");
  if(fread(&z,sizeof(float),1,f)!=1) break;
  printf("%8.3f",z);
  i++;
```

```
  }
  fclose(f);
}
// Чтение элементов k-го столбца
void read_k(char *name, int k)
{
 int m;
 FILE *f = fopen(name,"rb");
 if (f==NULL) {
          printf("Файл не найден. \n");
          system("pause");
          return ;
          }
 float z;
int i;
 fread(&m,sizeof(int),1,f);
 for( i=0;i<m;i++)
 {
  int l =i*m*sizeof(float) + k*sizeof(float)+sizeof(int);
  fseek(f, l,SEEK_SET);
  fread(&z,sizeof(z),1,f);
  printf("%8.3f",z);
  }
  fclose(f);
}
int main(int argc, char *argv[])
{
char Fname[30];
int n,m;
printf("Введите имя создаваемого файла: ");
scanf("%s",Fname);
printf("Введите количество строк матрицы: ");
scanf("%d",&n);
printf("Введите номер столбца: ");
scanf("%d",&m);
create_file(Fname,n);
printf("Матрица: \n");
read_file(Fname);
printf("Элементы столбца с номером %d \n",m);
read_k(Fname,m);
system("pause");
return 0; }
```

Аналогичным образом решаются задачи, требующие изменения уже существующих файлов.



Пример 9.9

В двоичном файле записано произвольное количество целых чисел. Не считывая все содержимое файла в память, поменять порядок элементов на обратный – поменять местами первый и последний элементы, второй и предпоследний и т. д.

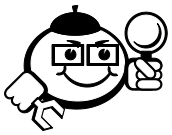
Будем считать, что файл уже создан с помощью функций, подобных функциям из предыдущих примеров.

Напишем функции печати заданного двоичного файла и изменения заданного файла.

```
#include <stdio.h>
#include <stdlib.h>
/* Функция чтения файла, функция возвращает количество считанных данных. */
int RP_f(char *name){
FILE *f = fopen(name,"rb");
int z;
int n = 0;
while(!feof(f))
{
 if(fread(&z,sizeof(z),1,f)!=1) break;
 printf("%4d",z);
 n++;
}
printf("\n");
fclose(f);
return n;
}
/* Функция изменения файла. Параметры функции – имя открываемого файла, количество элементов, записанных в файле. */
void change_f(char *name,int n){
 int i= 0;
 int z,y,k;
 int j = (n-1)*sizeof(int);
// i – номер позиции в начале файла.
// j – номер позиции в конце файла.
```

```
 FILE *f = fopen(name, "r+b");
// Всего необходимо провести n/2 обменов.
 for( k=0;k<n/2;k++)
 {
// Прочитать число с позиции i в переменную z.
  fseek(f,i,SEEK_SET);
  fread(&z,sizeof(z),1,f);
// Прочитать число с позиции j в переменную y.
  fseek(f,j,SEEK_SET);
  fread(&y,sizeof(y),1,f);
// Записать число z на позицию j.
  fseek(f,i,SEEK_SET);
  fwrite(&y,sizeof(y),1,f);
 // Записать число y на позицию i.
 fseek(f,j,SEEK_SET);
  fwrite(&z,sizeof(z),1,f);
//Увеличить номер позиции i, уменьшить номер позиции j.
  i+=sizeof(int);
  j-=sizeof(int);
 }
fclose(f);
}
int main(int argc, char *argv[])
{
// Прочитать файл My_int.fil, количество записей в файле
// сохранить в переменную k.
int k = RP_f("My_int.fil");
// «Перевернуть» файл
 change_f("My_int.fil",k);
// Напечатать измененный файл
 k = RP_f("My_int.fil");
 system("pause");
return 0; }
```

Рассмотрим еще один пример работы с двоичными файлами.



Пример 9.10

В двоичном файле хранятся размерность матрицы (два целых числа) и сама целочисленная матрица. Гарантируется, что количество столбцов матрицы

больше количества строк. Удалить столбцы матрицы, начиная с заданного, сделав ее квадратной.

Язык Си не предоставляет возможностей по удалению информации из файла, поэтому можно предложить следующий алгоритм.

Просматривая матрицу по строкам, будем переписывать в новый файл только те элементы столбцов, которые не подлежат удалению. Например, если в файле записана матрица из 5 строк и 9 столбцов. Пусть необходимо удалить столбцы, начиная со второго. Для того чтобы матрица стала квадратной, необходимо удалить 4 столбца. В новый файл перепишем столбцы с номерами 0, 1, 6, 7, 8. Реализация решения приведена ниже.

```
#include <stdio.h>
#include <stdlib.h>
//функция создания файла с именем name
// n – количество строк матрицы m – количество столбцов
// В созданном файле сначала записана размерность файла,
// затем сама матрица
void create_file(char *name, int n,int m)
{
 FILE *f = fopen(name,"wb");
 if (f==NULL) {
          printf("Файл не создан. \n");
          system("pause");
          return ;
          }
 fwrite(&n,sizeof(n),1,f);
 fwrite(&m,sizeof(m),1,f);
int i;
  for( i=0;i<n*m;i++)
 {
  int z = rand()%200-rand()%70;
  fwrite(&z,sizeof(int),1,f);
  }
  fclose(f);
}
// Функция чтения двоичного файла с именем name
void read_file(char *name)
{
 int m,n;
 FILE *f = fopen(name,"rb");
 if (f==NULL) {
          printf("Файл не найден. \n");
```

```
          system("pause");
          return ;
          }
 int i,j,z;
  fread(&n,sizeof(int),1,f);
  fread(&m,sizeof(int),1,f);
  printf("\n%d %d\n",n,m);
 for(i=0;i<n;i++){
  for(j=0;j<m;j++){
     if(fread(&z,sizeof(int),1,f)!=1) break;
     printf("%5d",z);}
  printf("\n");
  }
  fclose(f);
  printf("\n");
}
// Функция удаления столбцов
void delete_st(char *name,int k)
{
   int m,n;
// Открываются два файла, f – для чтения f1 – для записи
 FILE *f = fopen(name,"rb");
 FILE *f1 = fopen("temp","wb");
 if (f==NULL) {
          printf("Файл не найден. \n");
          system("pause");
          return ;
          }
 int i,j,z,l;
// Чтение размерности файла
  fread(&n,sizeof(int),1,f);
  fread(&m,sizeof(int),1,f);
// Проверка корректности заданного номера столбца
  if (k>n) {printf("Введен неверный номер столбца");
         system("pause");
         return;}
// Запись в новый файл новой размерности матрицы
  fwrite(&n,sizeof(int),1,f1);
  fwrite(&n,sizeof(int),1,f1);
// Количество удаляемых столбцов
  int d = m-n;
// Просматриваем все строки
  for(i=0;i<n;i++)
```

```
    // Просматриваем столбцы от 0 до k
      { for(j=0;j<k;j++)
    // Считываем и перезаписываем в новый файл элементы матрицы
        {fread(&z,sizeof(int),1,f);
        fwrite(&z,sizeof(int),1,f1);}
    // Получаем текущую позицию в файле
       l = ftell(f);
    // Пропускаем удаляемые столбцы
       l+=d*sizeof(int);
       fseek(f,l,SEEK_SET);
    /* И вновь считываем и перезаписываем элементы матрицы в но-
вый файл */
        for(j=k+d;j<m;j++){
          fread(&z,sizeof(int),1,f);
          fwrite(&z,sizeof(int),1,f1);}
        }
      fclose(f); fclose(f1);
    // Удаляем старый файл
      remove(name);
    // Новому файлу даем имя старого файла
      rename("temp",name);
     }
    int main(int argc, char *argv[])
    { system("chcp 1251");
     create_file("My.bin",5,9);
     read_file("My.bin");
     delete_st("My.bin",0);
     read_file("My.bin");
     system("PAUSE");
     return 0;
    }
```

В описанном выше примере создается файл `My.bin`, содержащий элементы матрицы [5×9]. Из матрицы, находящейся в файле, удаляются четыре столбца, начиная с нулевого. Измененный файл выводится на экран.

Контрольные вопросы по главе 9

1. Какие типы файлов существуют в языке Си?
2. Каким образом должен быть описан указатель на файл?

3. Какая функция открывает файл?
4. С помощью каких функций можно закрыть открытые в программе файлы?
5. Каким образом указывается тип доступа к файлу (чтение, запись, изменение и т. д.)?
6. Перечислите различия между двоичными и текстовыми файлами.
7. Запишите фрагмент программы, создающий текстовый файл, содержащий `n` вещественных чисел.
8. Рассмотрим фрагмент программы:

```
FILE *f;
f = fopen("a1.dat", "r+");
while (!feof(f))
 {
  fscanf(f, "%d",&k);
  printf ("%d",k);
 }
```

Укажите:

а) имя открываемого в фрагменте файла;

б) с каким атрибутом доступа открыт файл;

с) какого типа данные считываются из файла.

9. Возможно ли организовать прямой доступ к файлу в языке Си?
10. Опишите работу функции `fread`.
11. Опишите работу функции `fwrite`.
12. В двоичном файле записано `5*8` вещественных чисел. С какой позиции расположен элемент, находящийся в строке с номером `3` и столбце с номером `2`?
13. На диске хранится двоичный файл, содержащий матрицу целых чисел размерности `n`. Напишите фрагмент программы, который изменяет элементы главной диагонали матрицы на единицы.
14. Запишите фрагмент программы, создающий двоичный файл, содержащий `n` вещественных чисел.

10 Управление выводом в консоль

10.1 Win32 API

Win32 API (WinAPI) – это набор функций (API – *Application Programming Interface*), работающих под управлением операционной системы Windows. Они содержатся в библиотеке windows.h. С помощью WinAPI можно *создавать* различные оконные процедуры, диалоговые окна, программы и даже игры. Библиотека WinAPI является базовой в освоении программирования Windows Forms, MFC, т. к. эти интерфейсы являются надстройками этой библиотеки.

В этой главе учебного пособия рассмотрим функции WinAPI, с помощью которых можно управлять выводом в консольное окно – изменять положение курсора и цвет выводимых символов.

10.2 Типы данных *WINDOWS*

Все типы данных, описанные ниже, определены в заголовочном файле windows.h. Вы можете самостоятельно посмотреть их описание. Имена типов создавались в так называемой венгерской нотации. Один из первых разработчиков Windows венгр Чарльз Симонаи для именования переменных использовал способ, который в дальнейшем получил название венгерской нотации. Идею этой системы можно определить несколькими правилами:

- каждое слово в имени переменной пишется с прописной буквы и слитно с другими словами, например MyMax, YourMax, VariableForSavingIndexMax и т. д.;
- каждый идентификатор начинается с нескольких строчных букв, которые определяют его тип.

Такие приставки называются префиксами, примеры префиксов приведены в таблице 10.1.

Таблица 10.1 – Венгерская нотация

Префикс	Тип данных
b	BYTE (unsigned char)
cx, cy	short (используются как ширина и длина объектов типа RECT и окон)
dw	DWORD (unsigned long)

Префикс	Тип данных
fn	function
H	HANDLE
i	int
l	LONG (long)
n	int или short
s	string
sz	string terminated by zero
w	WORD (unsigned int)
x, y	short (используются как координаты)
c	char

Опишем типы, которые будут использоваться в функциях управления выводом в консольное окно.

Тип COORD определяет координаты символьной матрицы в консоли. Имеет два целочисленных поля x и y.

Тип CONSOLE_SCREEN_BUFFER_INFO – структура для хранения информации о текущих характеристиках консоли. Структура определена следующим образом:

```
typedef struct _CONSOLE_SCREEN_BUFFER_INFO {
COORD dwSize; // размер окна консоли
COORD dwCursorPosition; // текущие координаты курсора
WORD wAttributes; // атрибуты цвета
SMALL_RECT srWindow; // местоположение консоли
COORD dwMaximumWindowSize; // максимальный размер окна
} CONSOLE_SCREEN_BUFFER_INFO ;
```

10.3 Функции *WinAPI*

В этом параграфе опишем функции, которые будем использовать для управления выводом в консоли.

Для выполнения анализа нажатой клавиши используются две функции, описанные ниже.

Функция GetAsyncKeyState() определяет состояние виртуальной клавиши.

Описание: int GetAsyncKeyState(int Key);

Параметры:

Key – код виртуальной клавиши.

Возвращаемое значение:

Если установлен старший байт, клавиша Key находится в нажатом положении, если младший, то клавиша Key была нажата после предыдущего вызова функции.

Функция keybd_event() синтезирует нажатие клавиши. Вызывает функцию keybd_event программа обработки прерываний драйвера клавиатуры.

Описание:

```
void keybd_event( BYTE bVk, BYTE bScan, DWORD dwFlags,
                  PTR dwExtraInfo);
```

Параметры:

bVk определяет код виртуальной клавиши. Код должен быть значением в диапазоне от 1 до 254.

bScan не используется.

dwFlags определяет различные виды операций функции. Этот параметр может состоять из одного или нескольких следующих значений:

KEYEVENTF_EXTENDEDKEY – если параметр установлен, коду клавиши предшествует префиксный байт, имеющий значение 0xE0 (224).

KEYEVENTF_KEYUP – если параметр установлен, клавиша была отпущена. Если параметр не установлен, клавиша была нажата.

dwExtraInfo определяет дополнительное значение, связанное с нажатием клавиши.

Возвращаемых значений функция не имеет.

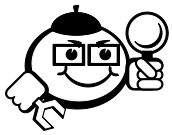
В таблице 10.2 описаны коды виртуальных клавиш.

Таблица 10.2 – Коды виртуальных клавиш

Константа	Код	Клавиша	Константа	Код	Клавиша
VK_LBUTTON	01	Левая кнопка мыши	VK_RBUTTON	02	Правая кнопка мыши
VK_CANCEL	03	Ctrl–Break	VK_MBUTTON	04	Средняя кнопка мыши
VK_BACK	08	BACKSPACE	VK_TAB	09	TAB
VK_CLEAR	0c	CLEAR	VK_RETURN	0d	ENTER
VK_SHIFT	10	SHIFT	VK_CONTROL	11	CTRL
VK_MENU	12	ALT	VK_PAUSE	13	PAUSE
VK_CAPITAL	14	CAPS LOCK	VK_ESCAPE	1b	ESC
VK_SPACE	20	SPACEBAR	VK_PRIOR	21	PAGE UP
VK_NEXT	22	PAGE DOWN	VK_END	23	END
VK_HOME	24	HOME	VK_LEFT	25	LEFT ARROW
VK_UP	26	UP ARROW	VK_RIGHT	27	RIGHT ARROW

Константа	Код	Клавиша	Константа	Код	Клавиша
VK_DOWN	28	DOWN ARROW	VK_SELECT	29	SELECT
VK_PRINT	2a	PRINT	VK_EXECUTE	2b	EXECUTE
VK_SNAPSHOT	2c	PRINT SCREEN	VK_INSERT	2d	INS
VK_DELETE	2e	DEL	VK_HELP	2f	HELP
VK_A – VK_Z	от 41 до 5a	'a' – 'z'	VK_0 – VK_9	от 30 до 39	'0' – '9'
VK_NUMPAD0 – VK_NUMPAD9	от 60 до 69	'0'–'9'	VK_MULTIPLY	6a	Умножение
VK_ADD	6b	Сложение	VK_SEPARATOR	6c	Разделение
VK_SUBTRACT	6d	Вычитание	VK_DECIMAL	6e	Десятичная точка
VK_DIVIDE	6f	Деление	VK_F1 – VK_F24	от 70 до 87	
VK_NUMLOCK	90	NUMLOCK	VK_SCROLL	91	SCROLL

Рассмотрим следующий пример:



Пример 10.1

Приведенная ниже программа выводит сообщение о нажатой пользователем клавише. Анализируются только клавиши управления курсором. При нажатии клавиши *ESC* программа заканчивает работу. При нажатии на любую другую клавишу программа ничего не выполняет

```
#include <stdio.h>
#include <stdlib.h>
// Подключим заголовочный файл с описаниями API функций
#include <windows.h>

int main(int argc, char *argv[])
{  system("chcp 1251");
  printf("Нажмите клавишу управления курсором: ");
 // организуем бесконечный цикл
  while(1){
// Если нажата клавиша «Вверх»
if (GetAsyncKeyState(VK_UP))
{keybd_event(VK_UP,0,KEYEVENTF_KEYUP,0);
 printf("\nНажата клавиша Вверх\n");
 system("pause");
 system("cls"); // очистить экран
printf("Нажмите клавишу управления курсором: ");
```

```
 }
// Если нажата клавиша «Вниз»
if (GetAsyncKeyState(VK_DOWN))
{keybd_event(VK_DOWN,0,KEYEVENTF_KEYUP,0);
 printf("\nНажата клавиша Вниз\n");
 system("pause");
system("cls");
printf("Нажмите клавишу управления курсором: ");}
// Если нажата клавиша «Вправо»
 if (GetAsyncKeyState(VK_RIGHT))
{keybd_event(VK_RIGHT,0,KEYEVENTF_KEYUP,0);
printf("\nНажата клавиша Вправо\n");
system("pause");
system("cls");
printf("Нажмите клавишу управления курсором: "); }
// Если нажата клавиша «Влево»
if(GetAsyncKeyState(VK_LEFT))
{keybd_event(VK_LEFT,0,KEYEVENTF_KEYUP,0);
printf("\nНажата клавиша Влево\n");
system("pause");
system("cls");
printf("Нажмите клавишу управления курсором: "); }
// Если нажата клавиша ESC
if(GetAsyncKeyState(VK_ESCAPE))
{keybd_event(VK_ESCAPE,0,KEYEVENTF_KEYUP,0);
 printf("\nКонец работы\n");
      system("pause");
     system("cls");
     break; // закончить выполнение цикла
     }   }
  return 0;
}
```

Для управления курсором `WinAPI` предлагает описанные далее функции.

Функция `GetStdHandle` – получение дескриптора консоли.

Описание: `HANDLE GetStdHandle(HANDLE hCon);`

Параметры – дескриптор `hCon`.

Возвращаемое значение – если функция завершается успешно, возвращаемое значение – дескриптор определяемого устройства. Если функция завершается с ошибкой, возвращаемое значение – флаг `INVALID_HANDLE_VALUE`.

Кратко определим, что такое дескриптор окна. Это некоторый идентификатор, по которому операционная система будет отличать его от остальных окон. Через дескриптор можно обращаться к окну в процессе работы в других функциях, например для того, чтобы изменить параметры окна. Все окна, создаваемые операционной системой, имеют свой уникальный дескриптор. Консоль – это такое же окно Windows, поэтому консольное окно, как и все окна, имеет свой собственный деструктор.

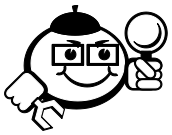
Следующая функция – SetConsoleCursorPosition, используется для установки курсора в заданную позицию экранного буфера консоли.

Описание: BOOL SetConsoleCursorPosition(HANDLE hCon, COORD cPos);

Параметры – дескриптор экранного буфера консоли hCon, структура cPos, которая устанавливает новую позицию курсора. Координаты должны иметь значения в пределах границ экранного буфера консоли.

Возвращаемые значения – если функция завершается успешно, величина возвращаемого значения – любое ненулевое значение. Если функция завершается с ошибкой, величина возвращаемого значения – ноль.

Рассмотрим пример использования описанных функций.



Пример 10.2

Программа, приведенная ниже, ожидает ввода произвольного символа. Затем при нажатии клавиш «A» (влево), «D» (вправо), «S» (вниз), «W» (вверх) «передвигает» символ по экрану. Если происходит выход за границы экрана, символ «появляется» с противоположной стороны.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>
#define WIDTH 79 // ширина окна
#define HEIGHT 22 // высота окна

int main(int argc, char *argv[])
{
  system("chcp 1251");
     // определение типа перечисления, элементы которого равны
     // кодам символов A, S, D, W
  enum number {right=100,left=97,up=119,down=115};
```

```
  HANDLE hCon;
// получить дескриптор консоли
  hCon = GetStdHandle(STD_OUTPUT_HANDLE);
  int z;
  char c;
  puts("Введите символ: ");
// ввод символа без подтверждения (без ENTER)
  c = getch();
  system("cls");
  COORD cPos;
// задать начальные координаты
  cPos.Y = 0;
  cPos.X = 0;
// установить курсор в заданную позицию
  SetConsoleCursorPosition(hCon, cPos);
// вывести символ
  putc(c,stdout);
// пока не нажата клавиша ESC
  while ((z = getch())!=27)
    switch(z){
// если нажата клавиша «Вправо» и символ не находится в
// последней позиции строки консоли
          case right: if (cPos.X<WIDTH) {
                 // установить курсор в старую позицию
                // и вывести символ ' ' (стереть символ)
SetConsoleCursorPosition(hCon, cPos);
                 putc(' ',stdout);
                // увеличить координату X
                 (cPos.X)++;
                // вывести символ на новой позиции
                 SetConsoleCursorPosition(hCon, cPos);
                 putc(c,stdout);}
           // если символ стоит в последней позиции строки,
           // то вывести его на первой позиции этой же строки
                 else {SetConsoleCursorPosition(hCon, cPos);
                 putc(' ',stdout);cPos.X = 0;} break;
          case left: if (cPos.X>0) {SetConsoleCursorPosi-
tion(hCon, cPos);
                 putc(' ',stdout);
                 (cPos.X)--;
                 SetConsoleCursorPosition(hCon, cPos);
                 putc(c,stdout);}
                 else {SetConsoleCursorPosition(hCon, cPos);
```

```
                    putc(' ',stdout);cPos.X = WIDTH-2;} break;
            case up: if (cPos.Y>0) {SetConsoleCursorPosi-
tion(hCon, cPos);
                    putc(' ',stdout);
                    (cPos.Y)--;
                    SetConsoleCursorPosition(hCon, cPos);
                    putc(c,stdout);}
                    else {SetConsoleCursorPosition(hCon, cPos);
                    putc(' ',stdout);cPos.Y = HEIGHT-2;} break;
            case down: if (cPos.Y<HEIGHT)
                   {SetConsoleCursorPosition(hCon, cPos);
                    putc(' ',stdout);
                    (cPos.Y)++;
                    SetConsoleCursorPosition(hCon, cPos);
                    putc(c,stdout);}
                    else {SetConsoleCursorPosition(hCon, cPos);
                    putc(' ',stdout);cPos.Y = 0;} break;
            }
      system("PAUSE");
      return EXIT_SUCCESS;
    }
.....................................................................
```

Выполнение действий при нажатии клавиш A, S, W аналогично, меняются лишь координата, которую необходимо изменять, и шаг изменения.

Следующие функции предназначены для изменения цвета фона и текста.

В файле `windows.h` определены три константы, которые определяют коды основных цветов:

`FOREGROUND_RED FOREGROUND_GREEN FOREGROUND_BLUE`

и константа, изменяющая интенсивность выводимых символов –

`FOREGROUND_INTENSITY`.

Все остальные цвета формируются комбинациями из четырех описанных значений. Например, ярко-желтый цвет определяет комбинация

`FOREGROUND_GREEN| FOREGROUND_BLUE |FOREGROUND_INTENSITY`

В таблице 10.3 описаны комбинации констант для получения основных цветов.

Принцип определения цвета фона аналогичен принципу определения цвета текста, для комбинаций используются константы:

`BACKGROUND_RED BACKGROUND_GREEN`

`BACKGROUND_BLUE BACKGROUND_INTENSITY`.

По умолчанию для консольного окна определен цвет фона – черный, цвет символов – белый.

Таблица 10.3 – Основные цвета

Комбинация констант	**Цвет**
0	Черный
FOREGROUND_RED	Красный
FOREGROUND_RED | FOREGROUND_INTENSITY;	Ярко-красный
FOREGROUND_GREEN;	Зеленый
FOREGROUND_GREEN | FOREGROUND_INTENSITY;	Ярко-зеленый
FOREGROUND_GREEN | FOREGROUND_RED;	Желтый
FOREGROUND_GREEN | FOREGROUND_RED | FOREGROUND_INTENSITY;	Ярко-желтый
FOREGROUND_BLUE;	Синий
FOREGROUND_BLUE | FOREGROUND_INTENSITY;	Ярко-синий
FOREGROUND_RED | FOREGROUND_BLUE;	Розовый
FOREGROUND_RED | FOREGROUND_BLUE | FOREGROUND_INTENSITY;	Ярко-розовый
FOREGROUND_BLUE | FOREGROUND_GREEN;	Голубой
FOREGROUND_BLUE | FOREGROUND_GREEN;	Светло-голубой
FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_RED;	Серый
FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_RED | FOREGROUND_INTENSITY;	Белый

Функция `SetConsoleTextAttribute` устанавливает заданные значения цвета и фона.

Описание:

```
BOOL SetConsoleTextAttribute(HANDLE hCont, WORD wAttr);
```

Параметры:

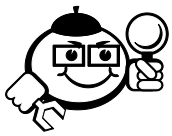
`hCon` – дескриптор экранного буфера;

`wAttr` – цвет текста и фона.

Возвращаемое значение: если функция завершается успешно, величина возвращаемого значения – любое ненулевое значение. Если функция завершается с ошибкой, величина возвращаемого значения – ноль.

Например, вызов функции `SetConsoleTextAttribute(hCon, FOREGROUND_RED|FOREGROUND_INTENSITY)` установит следующие текстовые атрибуты: цвет символов – ярко-красный, цвет фона – черный (по умолчанию).

Рассмотрим пример использования функций.



Пример 10.3

Напишем программу, которая реализует простейшее меню. Пользователю предлагается с помощью клавиш управления курсором выбрать цвет символов и цвет фона, после этого программа выводит системное сообщение, используя выбранные цвета.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
// тип-перечисление для определения цвета символа
enum TextColor {
BLACK = 0,
RED = FOREGROUND_RED,
LIGHT_RED = FOREGROUND_RED | FOREGROUND_INTENSITY,
GREEN = FOREGROUND_GREEN,
LIGHT_GREEN  = FOREGROUND_GREEN | FOREGROUND_INTENSITY,
YELLOW = FOREGROUND_GREEN | FOREGROUND_RED,
LIGHT_YELLOW   = FOREGROUND_GREEN | FOREGROUND_RED | FORE-
GROUND_INTENSITY,
BLUE = FOREGROUND_BLUE,
LIGHT_BLUE = FOREGROUND_BLUE | FOREGROUND_INTENSITY,
MAGNETTA = FOREGROUND_RED | FOREGROUND_BLUE,
LIGHT_MAGNETTA  =  FOREGROUND_RED  |  FOREGROUND_BLUE  |  FORE-
GROUND_INTENSITY,
CYAN = FOREGROUND_BLUE | FOREGROUND_GREEN,
LIGHT_CYAN  = FOREGROUND_BLUE | FOREGROUND_GREEN,
GRAY = FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_RED,
WHITE = FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_RED |
FOREGROUND_INTENSITY,
};
// тип-перечисление для определения цвета фона
enum BackColor {
BBLACK = 0,
BRED = BACKGROUND_RED,
BLIGHT_RED = BACKGROUND_RED | BACKGROUND_INTENSITY,
BGREEN = BACKGROUND_GREEN,
BLIGHT_GREEN  = BACKGROUND_GREEN | BACKGROUND_INTENSITY,
BYELLOW = BACKGROUND_GREEN | BACKGROUND_RED,
BLIGHT_YELLOW   = BACKGROUND_GREEN | BACKGROUND_RED | BACK-
GROUND_INTENSITY,
BBLUE = BACKGROUND_BLUE,
```

```
    BLIGHT_BLUE = BACKGROUND_BLUE | BACKGROUND_INTENSITY,
    BMAGNETTA = BACKGROUND_RED | BACKGROUND_BLUE,
    BLIGHT_MAGNETTA = BACKGROUND_RED | BACKGROUND_BLUE | BACK-
GROUND_INTENSITY,
    BCYAN = BACKGROUND_BLUE | BACKGROUND_GREEN,
    BLIGHT_CYAN  = BACKGROUND_BLUE | BACKGROUND_GREEN,
    BGRAY = BACKGROUND_BLUE | BACKGROUND_GREEN | BACKGROUND_RED,
    BWHITE = BACKGROUND_BLUE | BACKGROUND_GREEN | BACKGROUND_RED
| BACKGROUND_INTENSITY
    };
    // Функция вывода пунктов меню на экран
    /* Параметры функции – номер активного пункта меню Item, за-
головок меню Title, массив строк-пунктов меню str, количество
пунктов меню n. */
    void ShowMenu(int iItem, char * Title, char **str, int n)
    {  int i;
       system("cls");
       puts(Title);
    // Если номер выводимого пункта совпадает с номером активного
    // пункта, то вывести знак >
       for(i=0;i<n;i++)
         printf("%s %s\n",iItem == i ? ">" : " ",str[i]);   }

    int main(int argc, char *argv[])
    {
      char* str[] =
    {"BLACK","RED","LIGHT_RED","GREEN","LIGHT_GREEN","YELLOW",
"LIGHT_YELLOW","BLUE","LIGHT_BLUE","MAGNETTA","LIGHT_MAG-
NETTA","CYAN",  "LIGHT_CYAN","GRAY","WHITE"};
       system("chcp 1251");
       int iItem = 0;// Номер активного пункт меню
       int nLast = 14;// номер последнего пункта меню
       enum TextColor TC;
       enum BackColor BC;
       char T[] = "Выберите цвет символов: ";
       ShowMenu(iItem,T,str,15);
       HANDLE hCon;
       CONSOLE_SCREEN_BUFFER_INFO SCRN_INFO;
       hCon = GetStdHandle(STD_OUTPUT_HANDLE);
       GetConsoleScreenBufferInfo(hCon, &SCRN_INFO);
       while(1)
       {
         if(GetAsyncKeyState(VK_UP))
```

```
  {
    keybd_event(VK_UP, 0, KEYEVENTF_KEYUP, 0);
    if(0 <= iItem - 1)
         iItem = iItem - 1;
    else
         iItem = nLast;
    ShowMenu(iItem,T,str,15);
  }
  if(GetAsyncKeyState(VK_DOWN))
  {
    keybd_event(VK_DOWN, 0, KEYEVENTF_KEYUP, 0);
    if(iItem < nLast)
      iItem = iItem + 1;
    else
      iItem = 0;
    ShowMenu(iItem,T,str,15);
  }
  if(GetAsyncKeyState(VK_RETURN))
  {
    keybd_event(VK_RETURN, 0, KEYEVENTF_KEYUP, 0);
    switch (iItem){
        case 0: TC = BLACK;break;
        case 1: TC = RED;break;
        case 2: TC = LIGHT_RED;break;
        case 3: TC = GREEN;break;
        case 4: TC = LIGHT_GREEN;break;
        case 5: TC = YELLOW;break;
        case 6: TC = LIGHT_YELLOW;break;
        case 7: TC = BLUE;break;
        case 8: TC = LIGHT_BLUE;break;
        case 9: TC = MAGNETTA;break;
        case 10: TC = LIGHT_MAGNETTA;break;
        case 11: TC = CYAN;break;
        case 12: TC = LIGHT_CYAN;break;
        case 13: TC = GRAY;break;
        case 14: TC = WHITE;break;
        }
    break;
  }
}
strcpy(T,"Выберите цвет фона: ");
iItem = 0;
ShowMenu(iItem,T,str,15);
```

```
while(1)
{
  if(GetAsyncKeyState(VK_UP))
  {
    keybd_event(VK_UP, 0, KEYEVENTF_KEYUP, 0);
    if(0 <= iItem - 1)
      iItem = iItem - 1;
    else
      iItem = nLast;
    ShowMenu(iItem,T,str,15);
  }
  if(GetAsyncKeyState(VK_DOWN))
  {
    keybd_event(VK_DOWN, 0, KEYEVENTF_KEYUP, 0);
    if(iItem < nLast)
      iItem = iItem + 1;
    else
      iItem = 0;
    ShowMenu(iItem,T,str,15);
  }
  if(GetAsyncKeyState(VK_RETURN))
  {
    keybd_event(VK_RETURN, 0, KEYEVENTF_KEYUP, 0);
    switch (iItem){
        case 0: BC = BBLACK;break;
        case 1: BC = BRED;break;
        case 2: BC = BLIGHT_RED;break;
        case 3: BC = BGREEN;break;
        case 4: BC = BLIGHT_GREEN;break;
        case 5: BC = BYELLOW;break;
        case 6: BC = BLIGHT_YELLOW;break;
        case 7: BC = BBLUE;break;
        case 8: BC = BLIGHT_BLUE;break;
        case 9: BC = BMAGNETTA;break;
        case 10: BC = BLIGHT_MAGNETTA;break;
        case 11: BC = BCYAN;break;
        case 12: BC = BLIGHT_CYAN;break;
        case 13: BC = BGRAY;break;
        case 14: BC = BWHITE;break;
        }
    break;
  }
}
```

```
    SetConsoleTextAttribute(hCon,TC|BC);
    system("pause");
    return 0;
}
```

Контрольные вопросы по главе 10

1. Как называется библиотека функций, предоставляющая свои ресурсы для управления выводом в консоли?
2. Следуя венгерской нотации, какого типа должна быть переменная `nLastElement`?
3. Каким типом определяются переменные, которые содержат координаты курсора?
4. Дайте определение дескриптора окна.
5. В каком заголовочном файле определены константы, задающие коды основных цветов?
6. Запишите комбинацию основных цветов для получения серого цвета символов.
7. Запишите комбинацию основных цветов для получения красного цвета фона.
8. Какими будут цвета символов и фона консоли, если функция `SetTextAttributes` определяет их следующим образом:
   `BACKGROUND_BLUE | BACKGROUND_GREEN | BACKGROUND_RED`.

Заключение

В заключение хочется дать несколько рекомендаций читателям данного учебного пособия. Основные читатели учебника – студенты, решившие получить образование, используя дистанционную форму обучения. Такой способ получения знаний обусловливает выполнение студентом огромного объема самостоятельной работы.

Студентов, обучающихся дистанционно, можно разделить на две группы – студенты, получающие не первое высшее образование, и студенты, получающие первое высшее образование. Студенты первой группы уже имеют за спиной определенный багаж знаний и учебное пособие по дисциплине «Информатика и программирование» поможет систематизировать имеющиеся данные и применить их на практике.

Если Вы уже умеете программировать на каком-то языке программирования, учебное пособие может быть использовано в качестве справочного пособия по языку программирования Си. Рассмотренные примеры помогут Вам самостоятельно освоить синтаксис языка и подготовиться к контрольным и лабораторным работам и итоговому экзамену.

Если Вы только начинаете программировать, начните изучение с первой главы. Уделите особое внимание самому процессу алгоритмизации. Практика показывает, что умение правильно построить алгоритм поставленной задачи – это 80% успеха. В первой главе изложены основные типы алгоритмов, комбинируя которые можно решить большинство заданий и примеров, представленных в учебнике. За рамками учебного пособия остались эффективные алгоритмы сортировки, алгоритмы поиска подстроки в строке, алгоритмы работы с динамическими структурами. Если Вас заинтересовала алгоритмизация и Вы хотите узнать описание классических алгоритмов, не приведенных в этом учебном пособии, обратитесь к дополнительной литературе [7, 8].

И для начинающих программировать студентов, и для уже имеющих опыт самостоятельного программирования будет полезна информация, содержащаяся во второй главе учебного пособия. Основная часть этой главы посвящена основам программирования в интегрированной среде *DEV-C++*. Обратите внимание на процесс отладки программ – при самостоятельном изучении любого языка программирования очень важно умение находить ошибки. А совре-

менные средства отладки программ позволяют сделать процесс нахождения ошибок быстрым и эффективным.

Для закрепления практических навыков программирования Вам будет предложено выполнить ряд лабораторных работ. Перед их выполнением, оцените самостоятельно свои навыки программирования. Попробуйте написать программы, реализующие задания, предложенные для выполнения. Если выполнение этих заданий будет успешным, Вы можете быть уверены в хороших результатах контрольных работ и итогового экзамена по курсу.

Надеюсь, что изучение языка программирования Си станет начальной ступенькой для изучения других языков программирования и курсов, тесно связанных с программированием и информатикой. Знания и умения, полученные при изучении этой дисциплины, будут базовыми знаниями при изучении таких дисциплин, как «Объектно-ориентированный анализ и программирование», «Компьютерная графика», «Базы данных» и других.

Литература

1. Петров Ю. П. История и философия науки. Математика, вычислительная техника, информатика / Ю. П. Петров. – СПб. : БХВ-Петербург, 2010. – 444 с.
2. Bohm C. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules / C. Bohm, G. Jacopini // Communications of the ACM 9. – 1996. – № 5. – P. 366–371.
3. Mills Harlan D. Mathematical Foundations for Structured Programming [Electronic resource] / Harlan D. Mills // The Harlan D. Mills Collection. – URL: http://trace.tennessee.edu/utk_harlan/56 (дата обращения: 08.04.2016).
4. Дейкстра Э. Заметки по структурному программированию // Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М. : Мир, 1975. – С. 7–97.
5. Кушниренко А. Г. Основы информатики и вычислительной техники / А. Г. Кушниренко, Г. В. Лебедев, Р. А. Сворень. – 2-е изд. – М. : Просвещение, 1991. – 224 с.
6. Gough B. An Introduction to GCC for the GNU Compilers gcc and g++ [Electronic resource] / Brian Gough, Richard M. Stallman. – 2014. – URL : http://www.network-theory.co.uk/docs/gccintro/index.html (дата обращения: 08.04.2016).
7. Кнут Д. Э. Искусство программирования / Д. Э. Кнут. – 3-е изд. – М. : Вильямс, 2005. – Т. 1. Основные алгоритмы. – 712 с.
8. Седжвик Р. Фундаментальные алгоритмы на C++ / Р. Седжвик. – 3-е изд. – М. : DiaSoft, 2001. – 687 с.

Глоссарий

`char` – тип данных языка Си для представления символьных данных.

`double` – тип данных языка Си для представления вещественных чисел в диапазоне ±1.7E±308.

`float` – тип данных языка Си для представления вещественных чисел в диапазоне ±3.4E±38.

GNU CC (GNU Compiler Collection) – набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU. Является свободным программным обеспечением, распространяется фондом свободного программного обеспечения (FSF). Используется как стандартный компилятор для свободных UNIX-подобных операционных систем. Изначально разрабатывался для компиляции программ на Си. GCC расширен для компиляции исходных кодов на таких языках программирования, как C++, Objective-C, Java, Фортран и Ada.

`int` – тип данных языка Си для представления целых чисел в диапазоне от –2 147 483 648 до 2 147 483 647.

`long` – приставка типа данных, используемая с типами `double` и `int`. При этом размер типа `double` увеличивается до 12 байт.

`short` – приставка типа данных, используемая с типом `int`, уменьшает размер типа вдвое.

`signed` – приставка типа данных, говорящая о том, что значения данного типа знаковые. Используется по умолчанию.

`unsigned` – приставка типа данных, говорящая о том, что значения данного типа беззнаковые, т. е. все биты, отведенные для хранения числа, используются для представления числа.

`void` – тип данных языка Си. В качестве имени типа значения, возвращаемого функцией, указывает на то, что функция не возвращает значения, а вызов такой функции является void-выражением. В составе декларатора функции указывает на то, что функция имеет прототип и не имеет параметров. В качестве имени целевого типа операции приведения означает отказ от значения приводимого выражения. В составе имени типа void-указателя представляет значения любых указателей на объектные и неполные типы.

WinAPI (Windows Application Programming Interface) – набор функций, работающих под управлением операционной системы Windows.

Алгоритм – это описание данных и действий, производимых над ними для получения нужного результата.

Алгоритм Евклида – алгоритм нахождения наибольшего общего делителя (НОД).

Алфавит – множество неделимых символов, использующихся для создания программ на языке программирования.

Аргумент – переменная, передаваемая функции.

Бём Коррадо (Corrado Böhm) – итальянский математик, совместно с Д. Якопини сформулировавший и доказавший в 1965 г. теорему о структурном программировании.

Бинарный оператор – оператор, принимающий два операнда (например, оператор сложения +).

Блок-диаграмма (блок-схема) – распространенный тип схем (графических моделей), описывающих алгоритмы или процессы.

Блок-схема (блок-диаграмма) – распространенный тип схем (графических моделей), описывающих алгоритмы или процессы.

Венгерская нотация – правила организации идентификаторов, используемых в программе, предложенная Чарльзом Симонаи, венгром по происхождению.

Внутренние данные (локальные) – временные данные, используемые в алгоритме, например для сохранения промежуточных результатов вычислений.

Входные данные – данные, значения которых вводятся в алгоритм извне.

Выходные данные – данные, значения которых формируются в результате работы алгоритма.

Данные – это символьная или числовая информация, которая обрабатывается или используется в процессе выполнения алгоритма.

Дейкстра Эдсгер Вибе – нидерландский учёный, труды которого оказали влияние на развитие информатики и информационных технологий; один из разработчиков концепции структурного программирования.

Дескриптор окна – уникальный идентификатор, который присваивает окну операционная система.

Диаграмма Насси – Шнейдермана (Nassi – Shneiderman diagram) – графический способ представления структурированных алгоритмов и программ, раз-

работанный в 1972 г. американскими аспирантами Б. Шнейдерманом и А. Насси.

Заголовочный файл – файл, содержащий определения типов данных, структуры, прототипы функций, перечисления, макросы препроцессора.

Идентификатор – уникальное лексическое слово, которое обозначает сущность (переменную, объект, функцию и т. д.).

Индекс элемента – целое число, определяющее положение элемента в массиве.

Интегрированная среда разработки – система программных средств, используемая программистами для разработки программного обеспечения. Среда разработки включает: текстовый редактор, компилятор и/или интерпретатор, средства автоматизации сборки, отладчик.

Итерация – повторение определенной последовательности данных.

Ключевое слово – слово, зарезервированное для специального предназначения. Например, ключевое слово `if` – часть условной конструкции.

Консоль (консольное окно) – текстовое окно, предназначенное для ввода-вывода информации.

Константа – данное, не изменяющее свое значение на протяжении работы всего алгоритма.

Компиляция программы – трансляция программы, составленной на исходном языке программирования высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду.

Локальные данные (внутренние) – временные данные, используемые в алгоритме, например для сохранения промежуточных результатов вычислений.

Лексема – группа символов входной последовательности, идентифицируемая на выходе процесса как слово или разделитель языка.

Массив – это вектор однотипных данных, который имеет конечное число элементов.

Матрица – это таблица однотипных данных, которая имеет конечное количество строк и столбцов.

Оператор – лексема, которая переключает некоторые вычисления, когда применяется к переменной или к другому объекту в выражении.

Переменная – данное, значение которого может изменяться в процессе работы алгоритма.

Препроцессор – это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы, например такой, как компилятор.

Псевдокод – язык описания алгоритмов, использующий ключевые слова языков программирования, но опускающий подробности и специфический синтаксис.

Развилка – конструкция структурного программирования, осуществляющая выбор дальнейших действий программы при выполнении или невыполнении заданного условия.

Рекурсия – способ организации функции (подпрограммы), при котором эта подпрограмма (функция) в ходе выполнения вызывает сама себя.

Синтаксис языка программирования – набор правил, описывающий комбинации символов алфавита, считающиеся правильно структурированной программой или её фрагментом.

Следование – линейная последовательность действий программы.

Сортировка – упорядочивание набора однотипных данных по возрастанию или убыванию.

Ссылка – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается.

Стек – структура данных, представляющая собой список элементов, организованных по принципу LIFO (англ. *last in – first out*, «последним пришёл – первым вышел»).

Структурное программирование – методология разработки программного обеспечения, основанная на представлении программы в виде иерархической схемы блоков.

Теорема о структурном программировании – теорема, доказывающая, что любой алгоритм может быть написан с использованием только трех основных конструкций (следование, развилка, цикл).

Тернарный оператор – оператор, принимающий три операнда; в языке программирования Си существует только один тернарный оператор.

Тип данных – множество значений и операций на этих значениях.

Точка останова – это преднамеренное прерывание выполнения программы, при котором выполняется вызов отладчика. После перехода к отладчику программист может исследовать состояние программы, с тем чтобы определить, правильно ли ведёт себя программа. После остановки в отладчике про-

грамма может быть завершена либо продолжена с того же места, где произошёл останов.

Указатель (англ. *pointer*) – переменная, диапазон значений которой состоит из адресов ячеек памяти или специального значения `null` (нулевого адреса).

Унарный оператор – оператор, принимающий один аргумент. Например, оператор инфиксного увеличения ++.

Файл – область на носителе, какого-либо накопителя, содержащая логически объединённую информацию и названная конкретным именем.

Фактический аргумент – значения, которыми заменяются формальные аргументы при вызове функции.

Формальный аргумент – аргумент, указываемый при объявлении или определении функции.

Функция – поименованный фрагмент программного кода, к которому можно обратиться из другого места программы.

Цикл – конструкция структурного программирования, повторяющая определенные действия (итерации) несколько раз.

Якопини Джузеппе (Giuseppe Jacopini) – итальянский математик, совместно с К. Бёмом сформулировавший и доказавший в 1965 г. теорему о структурном программировании.