Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Ю. А. Луцик, А. М. Ковальчук, Е. А. Сасин

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ: ЯЗЫК СИ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальностей I ступени высшего образования, закрепленных за УМО

Минск БГУИР 2015

УДК [004.421+004.438](076)
ББК 32.973.26-018.2я73+32.973.26-018.1я73
Л86

Р е ц е н з е н т ы:
кафедра дискретной математики и алгоритмики
Белорусского государственного университета
(протокол №10 от 21.02.2014);

доцент кафедры систем автоматизированного проектирования
Белорусского национального технического университета,
кандидат технических наук, доцент И. Л. Ковалева

Луцик, Ю. А.

Л86 Основы алгоритмизации и программирования: язык Си : – учеб.-метод. пособие / Ю. А. Луцик, А. М. Ковальчук, Е. А. Сасин. – Минск : БГУИР, 2015. – 170 с. : ил.

ISBN 978-985-543-093-4.

В учебно-методическое пособие включены вопросы, связанные с разработкой алгоритмов решения задач. Рассмотрены основные конструкции языка Си.

Будет полезно студентам всех специальностей, магистрантам и аспирантам.

УДК [004.421+004.438](076)
ББК 32.973.26-018.2я73+32.973.26-018.1я73

ISBN 978-985-543-093-4

Введение

Язык С (С++) часто называют языком среднего уровня. Это означает, что С (С++) объединяет элементы языков высокого уровня с функциональностью Ассемблера.

Языки высокого уровня поддерживают концепцию типов данных. Тип данных определяет набор значений, которые переменная может хранить, и набор операций, которые могут выполняться над переменными. Наряду с тем, что в языке С (С++) представлены все основные типы данных, он не так жестко типизирован, как языки Паскаль или Ада. Язык С (С++) позволяет осуществлять большинство преобразований типов. Контроль за выполнением этих преобразований, а также проверка некоторых ошибок (например, выход за границы массива) возлагается на программиста.

Реализованная в С (С++) возможность напрямую манипулировать битами, байтами, словами и указателями необходима для программирования на системном уровне.

Язык С (С++) считается структурированным языком. Отличительной чертой структурированного языка является разделение кода и данных. Одним из способов решения этой проблемы является использование подпрограмм (функций), широко использующих локальные переменные. Необходимо отметить, что излишнее использование глобальных переменных может приводить к фатальным ошибкам.

Как и ряд других структурированных языков, С (С++) поддерживает ряд операторов цикла, условных операторов и операторов ветвления. Наряду с этим нежелательно использование оператора goto.

Язык С (С++) содержит стандартные библиотеки, предоставляющие функции, выполняющие наиболее типичные задачи. Эти библиотеки легко могут быть подключены, а также дополнены.

Язык С (С++) позволяет разбивать программу на части и выполнять их раздельную компиляцию. Откомпилированные таким образом файлы объединяются для создания полного объектного кода. Преимущество раздельной компиляции в том, что при изменении одного файла не требуется перекомпиляции всей программы.

В учебно-методическом пособии использованы материалы литературных источников [1−10].

1. Ведение в язык С (С++)

1.1. Основные понятия языка С (С++)

Программа на языке С (С++) состоит из последовательности *инструкций* (*операторов*), описывающих действия, выполняемые программой. Операторы используют *ключевые слова*, значение которых зафиксировано и не может использоваться для других целей (например, в качестве имени переменных). В С (С++) практически все операторы оканчиваются точкой с запятой «;». Фигурные скобки {} в С (С++) применяются для обрамления логически завершенных и обособленных от остальной программы участков кода.

Алфавит языка – это набор символов и цифр, который используется при составлении инструкций (операторов). Данный набор включает латинские прописные и строчные буквы, арабские цифры и специальные символы.

Идентификатор – это имя, которым обозначается некоторый объект (чаще всего данные) в программе. Данные в оперативной памяти размещаются по некоторым адресам, которые программисту заранее неизвестны. Для того чтобы иметь возможность обращаться к этим данным и обрабатывать их, программист этим данным даёт условные имена, которые компилятор в программе заменит адресами данных в оперативной памяти.

На идентификатор – имя переменной или функции – накладывается ряд ограничений. Он должен состоять только из букв латинского алфавита (прописных и строчный), арабских цифр и символа «_» (который считается буквой). При этом начинаться идентификатор должен с буквы.

Так как строчные и прописные буквы различаются, то идентификаторы BETA, beta, Beta будут различными. При выборе идентификатора необходимо учитывать следующее:

− идентификатор не должен совпадать с ключевыми словами языка и именами функций и констант из библиотек языка С (С++);

− не рекомендуется начинать идентификатор со знака подчеркивания, т. к. этот символ используется в именах некоторых библиотечных функций и констант.

Ограничений на количество символов в идентификаторе нет, однако воспринимаются и анализируются только первые 31.

Примеры правильных идентификаторов: sum, i, c10, Beta, beta, _func.

Ошибочные идентификаторы: a+b, -omega, 9c, if, int, &b, %f.

Ключевые слова – это зарезервированные имена, используемые в языке С (С++) с заранее определённым однозначным смыслом. Все зарезервированные слова пишутся строчными и только строчными буквами. В противном случае эти имена будут интерпретированы как имена переменных. Ключевые слова нельзя использовать в качестве идентификаторов объектов (имен переменных или функций) пользователя. Ключевые слова сообщают компилятору о типе данных, способе их организации, о последовательности и способе выполнения операторов и т. д. (например: break, case, char, const, continue, do, else, float, for, if, int, long, return, short, signed, struct, union, void и др.).

Переменные. Программы в процессе своего функционирования оперируют различными данными. В большинстве случаев эти данные могут изменять свои значения по ходу выполнения программы – это и есть *переменные*. Однако некоторые данные в процессе выполнения программы не могут изменить свое первоначальное значение. Такие переменные называются *константами*.

Спецификатор типа – одно или несколько ключевых слов, определяющих тип переменной. Например: int, char, short, unsigned и др.

Необходимо помнить, что в С (С++) имя переменной (идентификатор) никогда не определяет ее тип.

1.2. Первая программа

Традиционно первой программой, которую предлагается написать при изучении языка, является программа, выводящая на экран строку Hello, world.

```
int main()
{
   printf("Hello, world");
   return 0;
}
```

Функция main() является точкой входа в программу, и она должна присутствовать в любой программе на языке С (С++). Поэтому написание всех программ будет начинаться с создания этой функции. В фигурных скобках будут заключены операторы языка, необходимые для выполнения некоторых действий над данными.

Оператор printf("Hello, world") непосредственно выводит фразу "Hello, world" на экран консоли, при этом символы двойных ковычек не выводятся.

Оператор return 0 завершает выполнение функции main() и является точкой выхода из программы. Нуль после ключевого слова return сообщает операционной системе код завершения программы.

1.3. Типы данных и комментарии

Данные в С (С++) могут быть простыми и структурированными. К данным простого типа относятся целые и вещественные числа, текстовая информация, указатели (адреса). В С (С++) имеются следующие пять базовых *типов данных*:

1) **char** – используется для обозначения данных символьного типа;
2) **int** – используется для обозначения данных целого типа;
3) **float** – используется для обозначения данных вещественного типа одинарной точности;
4) **double** – используется для обозначения данных вещественного типа двойной точности;
5) **void** – пустой тип.

```
int i;          // i – переменная целого типа
char c;         // c – переменная символьного типа
float f;        // f – переменная вещественного типа
double d;   // d – переменная вещественного типа двойной точности
```

За исключением типа **void** перечисленные типы данных могут иметь *модификаторы*, уточняющие характеристики типа. К модификаторам относятся: **unsigned** (беззнаковый), **signed** (знаковый), **short** (короткий), **long** (длинный).

Тип данных и модификатор типа определяют:

− *формат* хранения данных в оперативной памяти (внутреннее представление данных);

− *диапазон* значений, в пределах которого может изменяться переменная;

− *операции,* которые могут выполняться над данными соответствующего типа.

Все типы данных можно разделить на две категории: скалярные и составные (рис. 1).

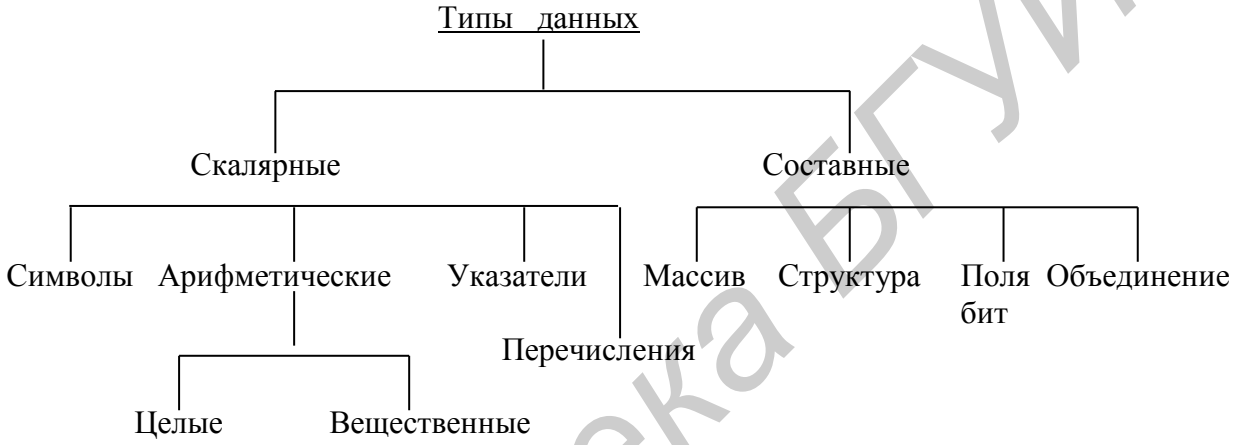


Рис. 1. Типы данных

Для рассмотрения различия указанных типов данных (с точки зрения их размеров) надо познакомиться с терминами «биты», «байты», «слова».

Наименьшая единица памяти называется бит. Она может принимать одно из двух значений: 0 или 1. Наименьшей адресуемой единицей памяти ПЭВМ является байт. Байт представляет собой комбинацию из восьми бит. Для удобства обмена информацией байты могут быть объединены (информационно) в слова (16-, 32-, 64-разрядные).

Целые числа и числа с плавающей запятой в ЭВМ отличаются по размеру и способу хранения.

Все переменные, массивы и другие структуры данных, которые используются в программе, необходимо декларировать. Возможны две формы декларации: *объявление*, не приводящее к выделению памяти, и *определение*, при использовании которого в зависимости от типа данных будет выделен соответствующий типу объём оперативной памяти. Выделенному участку памяти присваивается имя, которое в дальнейшем используется в программе.

1.4. Данные целого типа

В табл. 1 приведены характеристики данных целого типа.

Таблица 1

Название типа	Байт	Диапазон значений
char	1	–128...127
signed char	1	–128...127
unsigned char	1	0...255
int (16 разрядов) (32 разряда)	2 4	–32 768 ... 32 767 –2 147 483 648 … 2 147 483 647
signed int (16 разрядов) (32 разряда)	2 4	–32 768 ... 32 767 –2 147 483 648 … 2 147 483 647
unsigned int (16 разрядов) (32 разряда)	2 4	0 … 65 535 0 … 4 294 967 295
short int	2	–32 768 ... 32 767
signed short int	2	–32 768 ... 32 767
unsigned short int	2	0…65 535
long int	4	–2 147 483 648 … 2 147 483 647
signed long int	4	–2 147 483 648 … 2 147 483 647
unsigned long int	4	0 ... 4 294 967 295
long long int	8	–9 223 372 036 854 775 808 … +9 223 372 036 854 775 807
signed long long int	8	–9 223 372 036 854 775 808 … +9 223 372 036 854 775 807
unsigned long long int	8	0 …18 446 744 073 709 551 615

Как видно из приведённой таблицы, размер памяти для типов **int** и **unsigned int** не фиксирован в языке С (С++). По умолчанию размер **int** соответствует размеру, принятому на архитектурном уровне. Например, на 16-разрядной машине тип **int** всегда равен 16 битам или 2 байтам (слово). На 32-разрядной машине тип **int** всегда равен 32 битам или 4 байтам (двойное слово). Таким образом, тип **int** эквивалентен типам **short int** или **long int** в зависимости от реализации. Аналогично тип **unsigned int** эквивалентен типам **unsigned short int** или **unsigned long int**. Поэтому программы, зависящие от спецификации размера **int** и **unsigned int**, могут быть непереносимы.

В процессе объявления переменные могут быть инициализированы, т. е. им присваиваются начальные значения, которые в ходе выполнения программы могут изменяться. При этом компилятор, как правило, не выполняет проверку на соответствие присваиваемого значения и типа переменной (например int i=999999), а выдаёт лишь предупреждение.

Если в процессе работы потребуется информация о размере объекта (или его типа), то может быть использована операция **sizeof**(имя_объекта), возвращающая число байт объекта. Например:

```
i=sizeof(int);          //  i равно числу байт, отводимых для объектов типа int
j=sizeof(long int);  //  j аналогично, но для данных типа long int
k=sizeof(i);          //  k равно числу байт, отводимых под переменную i
```

В заключение отметим, что от места расположения объявлений данных в процедуре они различаются на **глобальные** и **локальные**. Первые объявляются вне любой из функций и доступны для многих из них. В отличие от них локальные переменные являются внутренними по отношению к функциям. Они начинают существовать при входе в функцию и уничтожаются при выходе из нее.

1.5. Данные вещественного типа

Числа с плавающей точкой представляются в соответствии с IEEE стандартом в виде двух частей – мантиссы M и порядка P числа в двоичной системе счисления [11]:

$A=M*2^P$.

Величины типа **float** занимают 4 байта. Из них 1 бит отводится для знака, 8 бит для избыточной экспоненты и 23 бита для мантиссы. Мантисса – это число между 1.0 и 2.0. Так как старший бит мантиссы всегда равен единице, он не запоминается в памяти. Такое представление даёт диапазон значений приблизительно от 1.17E–38 до 3.37E+38. Величины типа **double** занимают 8 байт памяти. Их формат аналогичен формату **float**. Биты памяти распределяются следующим образом: 1 бит для знака, 11 бит для экспоненты и 52 бита для мантиссы. С учётом опущенного старшего бита мантиссы диапазон значений получается приблизительно от 2.23E–308 до 1.67E+308 (табл. 2).

Таблица 2

Название типа	Байт	Диапазон значений	Точность
float	4	1.17E–38…3.37E+38	7
double	8	2.23E–308…1.67E+308	15
long double	10	3.37E–4932…1.2E+4932	19

Признаком константы с плавающей точкой является наличие в её записи точки, символа «E» или «e».

1.6. Константы

В отличие от переменных константы не изменяют своего значения в процессе выполнения всей программы. Аналогично переменным константы могут быть следующих основных типов:

- целые;
- вещественные;
- символьные;
- константное выражение, состоящее из констант, объединённых знаками операций.

Константа целого типа. Примером константы целого типа является число 241. Если требуется ввести константу типа **long**, то для этого надо в конце числа

указать признак L или l, например 143L. Признак L гарантирует, что для константы 143 в памяти будет отведено соответствующее число (4) байт. Это может быть важным для достижения совместимости при использовании константы с другими переменными и константами типа **long.** Если необходимо указать, что константа будет беззнаковой, то дополнительно используется признак U или u.

Кроме десятичной формы представления, константы целого типа могут быть записаны в виде:

− восьмеричного числа (если запись константы начинается с цифры 0), например 016, что соответствует десятичному числу 14;

− шестнадцатеричного числа (если число начинается с символов 0x или 0X), например 0x16, что соответствует десятичному числу 22.

Ниже приведены примеры целочисленных констант:

4356; – десятичная константа,

431L; – десятичная константа типа **long**,

0427; – восьмеричная константа,

0x136; – шестнадцатеричная константа,

0xFUL; – шестнадцатеричная беззнаковая константа типа **long**, равная 15.

Символьная константа. Символьные константы представляют собой одиночные символы, заключенные в апострофы, например:

```
simv='S';
```

Если в апострофы заключено более одного символа, то компилятор трактует это как ошибку:

```
simv='SS';
```

При описании символьной константы вместо символа может быть использован его ASCII-код, например:

```
simv='\123';    // 123 – ASCII-код символа S
```

В качестве символьных переменных могут использоваться управляющие символы (табл. 3).

Таблица 3

Управляющий знак	Наименование	Код
\n	Переход на новую строку	\x0A
\t	Горизонтальная табуляция	\x09
\v	Вертикальная табуляция	\x0B
\b	Возврат на одну позицию	\x08
\r	Перевод курсора в начало строки	\x0C
\f	Новая страница	\x0D
\a	Звонок (сигнал)	\x07
\'	Одиночная кавычка	\x27
\"	Двойная кавычка	\x22
\\	Наклонная черта влево(обратный слэш)	\x5C
\ddd	ASCII-символ в восьмеричном представлении	\ddd
\xdd	ASCII-символ в шестнадцатеричном представлении	\xdd

При присваивании символьной переменной эти символы должны быть заключены также в апострофы:

```
simv='\n';   simb='\f';
```

Вещественные константы. В языке С (С++) допустимо несколько способов описания вещественных чисел. Наиболее общим является способ, при котором последовательность цифр включает в себя десятичную точку и символ e(E): 3.142e–2, –1.732E+4. Знак «+» необязателен в записи числа. Можно пропускать либо десятичную точку, либо экспоненциальную часть, но не обе одновременно: 2.5348, .34e–2, 34e–2, .34, 34. Использование пробелов в записи константы недопустимо.

В процессе работы константы с плавающей точкой представляются в формате **double**, т. е. им отводится по 8 байт памяти.

```
int main()
{  float sm;
   sm=3.45*3.5;
   . . .
}
```

Результат операции умножения **3.45*3.5** имеет двойную точность. При выполнении присваивания происходит усечение результата до размера **float.** Это позволит достичь максимальной точности.

Использование препроцессора (директивы препроцессора **#define**, будет рассмотрено далее) является еще одним механизмом, позволяющим существенно упростить процедуру определения и изменения значения констант.

1.7. Модификатор const

В языке С (С++) имеется модификатор **const**, используемый для контроля за способом доступа или модификации переменных. Переменные, используемые с модификатором **const**, не могут изменять своего значения в процессе работы программы, они называются константами или константными переменными. То есть объект (переменная) с таким модификатором будет доступен только для чтения.

Число байт для хранения переменной с модификатором аналогично числу байт без него. Например:

```
const char s[]="БГУИР";
const float ff=23.527;
const i=341;
```

Применение модификатора без указания типа константы подразумевает по умолчанию тип **int.**

Обычно модификатор **const** используют при передаче адресов параметров структурного типа в функции для запрещения изменения значений, находящихся по этим адресам.

1.8. Переменные перечисляемого типа

Ключевое слово **enum** в С (С++) используется для объявления еще одного типа данных – перечисляемого. Он предназначен для описания целых констант (типа int) из некоторого заданного множества, например:

```
enum number {zero, one, two};
```

enum months {jan=1, feb, mar, apr, may, jun};

Здесь определены два типа **number** и **months**. Для определения переменной типа months запишем

enum months ms;

Переменная ms может принимать любое значение из списка констант, перечисленных в фигурных скобках. Каждому значению из списка соответствует целое десятичное число, начиная с нуля. Каждая следующая переменная имеет значение на единицу больше, чем предыдущая. Так как в типе months первая константа равна 1, то все последующие будут отсчитываться от неё (а не от нуля) с шагом, равным 1: jan=1, feb=2, mar=3 и т. д.

enum number i1,i2;

Каждая из переменных i1 и i2 может принимать одно из трёх значений: zero, one, two. Определение переменных можно выполнить и при объявлении типа zero=0, one=1, two=2, например:

enum number {zero=0, one, two} i1=two, i2=one;

Перечисление может быть описано и без задания имени типа. Имена в различных перечислениях должны отличаться друг от друга. Значения констант внутри одного перечисления могут совпадать. Например:

enum {one, two=one, four=4, six=4, nine} i1=one, i2=two;

В этом случае переменные i1 и i2 будут равны нулю и ассоциироваться с константой one. Константы four и six будут равны четырём, а константа nine – пяти.

В перечислении константам можно задавать значения не по порядку, при этом если не все значения констант явно специфицированы, то они продолжают прогрессию, начиная от последнего специфицированного значения:

enum number{one=2, two, four=two+one-1, six=two+3} i1=two, i2=four;

В этом случае значения именованных констант будут следующими:

one=2, two=3, four=4, six=6.

Переменные типа enum могут использоваться в индексных выражениях, как операнды в арифметических выражениях и в операциях отношения. Имя константы из списка перечисления эквивалентно её числовому значению. Именованным константам можно устанавливать как положительные, так и отрицательные значения.

1.9. Комментарии

Комментарий – это сообщение, поясняющее, что выполняется в данном месте программы. Комментарии позволяют повысить читабельность программы. При компиляции исходного текста компилятор игнорирует все комментарии.

Комментарий может быть одно- и многострочным. Если комментарий занимает только одну строку, то он начинаться символами «//», за которыми следует текст комментария. Если комментарий занимает более одной строки, то либо каждую его строку нужно начинать с «//», либо выделить все строки комментария символами «/*» (начало) и «*/» (конец). Весь текст между этими символами будет игнорироваться компилятором.

Несмотря на то что текст комментария может быть любым, принято, чтобы текст комментария был лаконичный, логически завершённый и объяснял, что происходит в том блоке кода, к которому он относится.

2. Операции и выражения

В программах широко используются выражения, состоящие из одного или более операндов, объединённых знаками операций. Операции – это специальные комбинации символов, обозначающие действия по преобразованию различных величин. Компилятор интерпретирует каждую из этих комбинаций как самостоятельную единицу, называемую лексемой. В качестве операндов могут быть использованы переменные, константы или другие выражения. Имеются три основных класса операций: арифметические, отношения (логические), побитовые. Различают операции унарные (с одним операндом), бинарные (с двумя) и операции с тремя и более операндами.

Операции, используемые в С (С++), должны использоваться точно так, как они представлены в табл. 4: без пробельных символов между символами в тех операциях, которые представлены несколькими символами.

Таблица 4

Знак операции	Наименование операции	Пример
1	2	3
Арифметические операции		
+	Сложение	a=b+c; если b=6, c=4, то a=10
–	Вычитание	a=b–c; если b=3, c=8, то a= –5
–	Арифметическое отрицание	a=–b; если b=132, то a= –132
*	Умножение	a=b*c; если b=2, c=4, то a=8
/	Деление	a=b/c; если b=7.0, c=2.0, то a=3.5
%	Остаток от деления	a=b%c; если b=10, c=3, то a=1
Логические операции		
&&	Логическое «И» (конъюнкция)	a=b && c; a=0, если b и (или) c равны нулю, в противном случае a=1
||	Логическое «ИЛИ» (дизъюнкция)	a=b || c; a=0, если b и c равны нулю, в противном случае a=1
!	Логическое отрицание (инверсия)	a= !b; a=1, если b равно нулю, иначе a=0
Побитовые операции		
&	Поразрядная конъюнкция	a=b & c; если оба сравниваемых бита единицы, то бит результата равен 1, в противном случае – 0. Например: b=1010, c=0110, тогда a=0010
|	Поразрядная дизъюнкция	a=b | c; если любой (или оба) из сравниваемых бит равен 1, то бит результата устанавливается в 1, в противном случае – в нуль. Например: b=1010, c=0110, тогда a=1110

Продолжение табл. 4

1	2	3
^	Поразрядное «исключающее ИЛИ»	a=b^c; если один из сравниваемых бит равен 0, а второй бит равен 1, то бит результата равен 1, в противном случае (оба бита равны 1 или 0) – 0. Например: b=1010, c=0110, тогда a=1100
~	Поразрядная инверсия	a=~b; если b=1010, то a=0101
<<	Сдвиг влево	a=b<<c; осуществляется сдвиг значения b влево на c разрядов; в освободившеся разряды заносятся нули. Например: b=1011, c=2, тогда a=1100
>>	Сдвиг вправо	a=b>>c; осуществляется сдвиг значения b вправо на c разрядов; в освободившееся разряды заносятся нули, если b имеет тип unsigned и копии знакового бита в противном случае. Например: b=1011, c=2, тогда a=0010
Операции отношения		
==	Сравнение на равенство	a==b; формируется 1, если a равно b, и 0 – в противном случае
>	Больше	a>b; формируется 1, если a больше b, и 0 – в противном случае
>=	Больше или равно	a>=b; формируется 1, если a больше или равно b, и 0 – в противном случае
<	Меньше	a<b; формируется 1, если a меньше b, и 0 – в противном случае
<=	Меньше или равно	a<=b; формируется 1, если a меньше или равно b, и 0 – в противном случае
!=	Не равно	a!=b; формируется 1, если a не равно b, и 0 – в противном случае
Операции присваивания		
=	Простое присваивание	a=b; a присваивается значение b
++	Унарный инкремент	a++; (++a;) значение a увеличивается на единицу
--	Унарный декремент	a--; (--a;) значение a уменьшается на единицу
znak=	Составная операция присваивания; znak∈{*, /, %, +, –, <<, >>, &, |, ^}	a znak=b; понимается как a=a znak b; Например: a+=b, если a=3, b=4, то a=7

Окончание табл. 4

1	2	3
Операции адресации и разадресации		
&	Адресация	ptr=&b; ptr присваивается адрес b
*	Разадресация	a=*****ptr; a присваивается значение по адресу ptr
Операция последовательного вычисления		
,	Запятая	a=(c--**,**++b); операция «,» вычисляет два своих операнда слева направо. Результат операции имеет значение и тип второго операнда. Если c=2, b=3, то a=4, c=1, b=4
Операция условного выражения		
?:	Условная операция	a=(b<0)?-b**:**b; переменной a присваивается абсолютное значение b
Size-операция		
sizeof	Определение размера в байтах	a=sizeof(int); определяет размер памяти, который соответствует идентификатору или типу. Переменной a присваивается размер памяти, занимаемый элементом типа int

2.1. Операция присваивания

Операция присваивания имеет знак «=». В результате выполнения этой операции переменная, стоящая слева от знака «=», принимает значение выражения, расположенного справа. Отличительной чертой операции присваивания в языке C (C++) является то, что она может быть использована в одном выражении более одного раза. Например:

```
int main()
{
   int i,j,k;
   i=j=k=23;
   return 0;
}
```

В этом примере присваивания выполняются справа налево: сначала переменная k принимает значение 23, затем j присваивается значение переменной k и, наконец, i принимает значение переменной j. То есть все переменные (i, j и k) будут равны 23.

2.2. Арифметические операции

Операция сложения имеет знак «+». Выполнение этой операции приводит к сложению двух величин, стоящих справа и слева от этого знака. Операнды, используемые в операции, могут быть как переменными, так и константами.

```
i=j+k;
```

Операция вычитания имеет знак «–». В результате выполнения этой операции переменная получает значение разности чисел, стоящих соответственно слева и справа от знака «–»:

```
i=j–k;
```

Операция изменения знака «–» может использоваться для задания или изменения алгебраического знака некоторой величины (выражения):

```
i=–10;
i=– (j–k);
n=–j;
```

Операция умножения имеет знак «*». Операция используется для вычисления произведения двух операндов соответственно слева и справа от операции умножения:

```
i=j*k;
```

Операция деления имеет знак «/». В языке С (С++) операция деления a/b выполняется следующим образом: операнды делятся, и получаемое частное имеет тип double. Далее полученное частное приводится к максимальному типу одного из операндов. Следовательно, результатом деления целых чисел будет число целое. При этом дробная часть у результата отбрасывается, т. е. выполняется «усечение». Результат деления целых чисел округляется не до ближайшего целого, а всегда до меньшего целого. Например:

```
int main()
{
   int i;          // переменная целого типа
   float k;        // переменная вещественного типа
   i=6/4;          //  результат i=1
   k=7./4.;        //  результат k=1.75
   k=6/4.;         //  результат k=1.25
   k=6/4;          //  результат k=1
   return 0;
}
```

Операция деления по модулю имеет знак «%». Операция используется в целочисленной арифметике. В результате получается остаток от деления. Например, в результате 14%4 получаем 2.

Операндами операции «%» должны быть целые числа. Остальные операции выполняются над целыми и плавающими операндами. Типы первого и второго операндов могут отличаться. Арифметические операции выполняют обычные арифметические преобразования операндов. Типом результата является тип операндов после преобразования.

```
#include <stdio.h>
int main()
{
   int i, j, k, l, m, n;
   float f,f1;
   char c,c1;
   i=12;
   k=32767; // максимальное значение для int 32767
   k=k+1;     // при увеличении значения переменной k происходит переполнение
```

```
    c='б';      // переменной с присваивается значение ASCII-кода символа б
    c=c+1;    // переход к следующей букве (в)
    printf(" i=%d,  j = %d    k=%d   c=%c   c=%d  i-5=%d   i % 3%d",
           i,j=i+1,k,c,c,i−5,i%3);
    return 0;
}
```

2.3. Операции поразрядной арифметики

Поразрядные операции используются для преобразования информации, содержащейся в памяти ПЭВМ, путем изменения некоторых отдельных либо всех разрядов (бит) памяти. К числу поразрядных операций относятся сдвиговые операции (вправо «>>» и влево «<<») и операции конъюнкции «&», дизъюнкции «|», исключающее ИЛИ «^» и поразрядная инверсия «~».

Операции сдвига «>>» и «<<» сдвигают свой левый операнд влево или вправо соответственно на количество бит, указанных в правом операторе:

```
i=j>>k;
i=j<<k;
```

Операции «&» (поразрядное И), «|» (поразрядное ИЛИ) и «^» (исключающее ИЛИ) применяются для своих операндов поразрядно (побитно):

```
i=j&k;
i=j^k;
i=j|k;
```

Операция «~» приводит к тому, что все разряды (биты), соответствующие переменной k, изменяют свои значения на противоположные:

```
j=~k;
```

Операнды побитовых операций должны быть целого типа. Результат операции сдвига не определен, если второй операнд отрицательный.

Необходимо отметить, что кроме перечисленных выше операций в языке C (C++) используется более короткая запись некоторых стандартных операций:

```
 переменная#=значение;
```

Эта запись аналогична записи вида переменная=переменная#значение. Здесь символ «#» соответствует одной из операций (+, −, *, /, %, <<, >>, &, |, ^).

2.4. Логические операции

Логические операции выполняют логическое отрицание «!», логическое И «&&» и логическое ИЛИ «||». Операнды логических операций могут быть целого, вещественного или адресного типа. Типы первого и второго операндов могут быть различными. Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции (значение первого операнда равно нулю для логического И и единице для логического ИЛИ), то второй операнд не вычисляется.

Логические операции не выполняют стандартные арифметические преобразования. Вместо этого они вычисляют каждый операнд с точки зрения его эквивалентности нулю. Результатом логической операции является 0 – ложь или 1 – истина. Тип результата есть int.

```
int x=1, y=0, z;
z=(x && y);     // в переменную z запишется 0
z=(x || y);     // в переменную z запишется 1
```

2.5. Операции отношения

Бинарные операции отношений используются для вычисления соотношения между операндами. Операция формирует значение 1 (истина) или 0 (ложь). Типом результата является int. Операнды могут быть целого, плавающего или адресного типа. Типы первого и второго операндов могут различаться.

Операции отношения: «==» – сравнение на равенство, «!=» – сравнение на неравенство, «>» и «<» – больше и меньше, «>=» и «<=» – больше или равно и меньше или равно соответственно.

```
int x=10, y=15, z;
z=(x == y);     // в переменную z запишется 0
z=(x <= y);     // в переменную z запишется 1
```

2.6. Инкрементные и декрементные операции

Операция «++» – инкрементирование (увеличения), а «--» – декрементирование (уменьшения) значения операнда на 1. Операнд должен быть целого, плавающего или адресного типа. Тип результата соответствует типу операнда.

Существуют две возможности использования операции: когда знак «++» («--») находится слева от переменной – «префиксная» форма, и справа – «постфиксная» форма. Если операция является префиксом своего операнда, то операнд инкрементируется или декрементируется перед его использованием в выражении, а если постфиксом – после его использования.

```
int n, m=1;
n=++m +2;   // вначале m увеличивается, затем используется в выражении
n=m++ +2;   // вначале m используется, затем увеличивает свое значение
```

2.7. Операция sizeof

Операция **sizeof** – унарная операция, возвращающая размер памяти в байтах, занимаемый переменной или типом, помещённых в скобках. Выражение **sizeof** имеет форму

```
sizeof(name);
```

Здесь name – или идентификатор, или имя типа. Имя типа не может быть **void**. Значением выражения **sizeof** является размер памяти в байтах, соответствующий идентификатору или типу.

2.8. Порядок выполнения и приоритет операций

Изменение порядка выполнения операций в выражении может привести к различным результатам, следовательно, требуется ввести набор непротиворечивых правил, определяющих порядок действий. В языке С (С++) это осуществляется путем задания приоритета той или иной операции. Приоритет – это уровень старшинства (первоочередности выполнения) операции. Если несколько операций имеют один и тот же уровень приоритета, то они выполняются в том

порядке, в котором записаны в операторе (выражении). Операции деления и умножения имеют более высокий приоритет по сравнению со сложением и вычитанием. Это означает, что в операторе i=35*j–k вначале выполняется 35*j, затем из полученного значения вычитается k. Если требуется, чтобы сложение или (и) вычитание в операторе выполнялось ранее деления и умножения, необходимо использовать механизм скобок. Так, в операторе i=35*(j+k) вначале выполняется (j+k), затем умножение.

В приведенной табл. 5 показан приоритет (порядок вычисления) операций языка С (С++). Операции упорядочены по приоритету сверху вниз.

Таблица 5

Приоритет	Операция	Описание операции	Порядок выполнения
1	:: -> . [] () ()	разрешение контекста, извлечение; индексирование массива; вызов функции; преобразование типа, выражение	слева направо
2	++ -- ~ ! – + & * sizeof	инкремент, декремент, инверсия; унарный –, унарный +; получение адреса; разадресация указателя; определение размера	справа налево
3	* / %	умножение, деление, остаток	справа налево
4	+ –	бинарное сложение и вычитание	слева направо
5	<< >>	сдвиг	слева направо
6	< <= > >=	сравнение	слева направо
7	== !=	отношения (равно, не равно)	слева направо
8	&	побитовое И	слева направо
9	^	XOR (исключающее ИЛИ)	слева направо
10	|	побитовое ИЛИ	слева направо
11	&&	логическое И	слева направо
12	||	логическое ИЛИ	слева направо
13	?:	тернарная операция	справа налево
14	= *= /= %= += и т. д.	операция присвоения	справа налево
15	,	следование	слева направо

Необходимо отметить, что если в вычисляемом выражении (или его некоторой части) скобки отсутствуют, то порядок вычисления составляющих его операндов в С (С++) (как и в ряде других языков) не фиксирован. Это означает, что если в выражении x=операнд1+операнд2 значение операнда1 влияет на величину операнда2, то необязательно значение операнда1 будет вычислено ранее операнда2.

Порядок вычисления аргументов функции также не определен и на разных компиляторах может давать несовпадающие результаты.

```
printf("%d %d",n++,n+i);
```

Во избежание этого до входа в функцию или до вычисления сложного выражения все операнды, влияющие друг на друга, должны быть вычислены.

```
n++;
printf("%d %d",n,n+i);
```

2.9. Преобразование типов

Для получения верного результата выражения желательно, чтобы операнды, входящие в выражение, и результирующая переменная были одного типа. В отличие от некоторых компиляторов с других языков это не приводит к фатальной ошибке на этапе компиляции. Вместо этого компилятор использует правила автоматического преобразования типов к некоторому общему типу:

1) **char** и **short** преобразуется в **int**, **float** – в **double** (все операции с вещественными числами производятся с двойной точностью). Имеют место следующие преобразования:

- **char** в **int** (со знаком);
- **unsigned char** в **int** (старший байт всегда нулевой);
- **signed char** в **int** (в знаковый разряд int передается знак из char);
- **short** в **int** (знаковый или беззнаковый);
- **float** в **double**;

2) **enum** преобразуется в **int**;

3) если один из операндов имеет тип **double** (**long**, **unsigned**), то другие преобразуются к **double** (**long**, **unsigned**) соответственно, результат будет иметь тип **double** (**long**, **unsigned**);

4) последовательность имен типов, упорядоченных от «высшего» типа к «низшему», имеет следующий вид: **double**, **float**, **long**, **int**, **short** и **char**. Применение модификатора **unsigned** повышает ранг соответствующего типа данных.

«Повышение» типа обычно заканчивается успешно, в то время как «понижение» может закончиться не всегда успешно. Это связано с тем, что всё число может не поместиться в элементе данных низшего типа.

```
#include<stdio.h>
int main()
{
   char c;          // объявление символьной переменной
   int i;           // объявление переменной целого типа
   float f;         // объявление переменной вещественного типа
   f=i=c='C';
   printf("c= %c, i= %d, f= %2.3f \n",c,i,f);
   c=c+1;           // преобразования выполнятся успешно, т. к. размерность
   i=f+c;           // памяти для переменных c, i, f  достаточна для
   f=i/c;           // размещения в них значений вычисленных выражений
   printf("c= %c, i= %d, f= %f \n",c,i,f);
   c=3.67e17;    // для числа 3.67e17 требуется > 1 байта (ошибка)
   printf("c= %c \n",c);
   return 0:
}
```

В языке С (С++) при вычислении все величины типа **float** преобразуются в тип **double**, что уменьшает вероятность погрешности при округлении. Конечный результат преобразуется к типу **float** (если это требуется).

2.10. Операция приведения

Рассмотренные в подразд. 2.9 преобразования типов выполняются автоматически. Существует возможность точно указать тип данных, к которому необходимо привести (преобразовать) некоторое выражение (переменную). Это заключается в том, что перед приводимым к требуемому типу выражением ставится тип в круглых скобках: (тип) выражение. Например:

```
int i:
i=(int)3.4+(int)'A';
```

При разработке программ желательно не смешивать типы, во многих языках это запрещено. Однако в некоторых случаях этого избежать нельзя или это упрощает программу. В языке С (С++) вся ответственность за подобные преобразования ложится на программиста.

2.11. Операция запятая

Операция «,» обеспечивает преобразование своих операндов слева направо. Так, например, в следующем выражении

```
k=(i=j+2,++j+4);
```

вычисления выполняются слева направо, и всё выражение примет значение последней вычисленной формулы. Например, если j=1, то переменная i примет значение 3, переменная j – значение 2, а значение всего выражения будет равно 6 (переменная k получит значение 6).

3. Ввод и вывод информации

3.1. Форматированный ввод/вывод

Ввод/вывод (в/в) информации является одной из частей языка С (С++), которая встречается практически во всех программах. Качество организации в/в во многом определяет программный интерфейс. Рассмотрим основные функции в/в: **printf()**, **scanf()**, **putchar()**, **getchar()**, **puts()** и **gets()**. Первые две предназначены для организации форматированного в/в данных. Вначале рассмотрим работу функции вывода данных.

Общий формат записи функции printf имеет следующий вид:

int printf("управляющая строка",аргумент1,аргумент2,...);

Управляющая строка представляет собой символьную строку, заключенную в кавычки «" "» и определяющую порядок и формат печати выводимой информации. Управляющая строка может содержать информацию двух видов:

− обычная текстовая информация;

− спецификации формата вывода, согласно которому осуществляется вывод одного из аргументов, содержащихся в списке аргументов.

Также возможен случай, когда в управляющей строке содержится и текстовая информация и спецификации форматов вывода.

Каждая спецификация начинается с символа «%» и заканчивается символом преобразования типа выводимого значения. Ниже приведен формат спецификации:

% [знак] [размер поля вывода][.точность] символ_преобразования.

Знак, расположенный после «%», имеет следующий смысл (табл. 6).

Таблица 6

Знак	Действие
−	Выравнивание влево выводимого числа в пределах выделенного поля. Правая сторона выделенного поля дополняется пробелами. Если этот знак не указан, то по умолчанию производится выравнивание вправо, т. е. пробелы ставятся перед числом
+	Выводится знак числа. Знак «–» при отрицательных значениях выводится всегда и не зависит от наличия данного флага
Пробел	Выводится знак пробела перед положительным числом
0	Заполняет поле нулями
#	Действие зависит от установленного для аргумента типа формата. Для целых чисел выводится идентификатор системы счисления: 0 – перед восьмеричным числом; 0x или 0X – перед шестнадцатеричным. При указании типа формата e, E или f происходит вывод десятичной точки. Действие данного символа при использовании формата g и G идентично действию при e и E

Размер поля вывода задает количество позиций для вывода символов. Если число подлежащих выводу символов меньше, чем указано в этом поле, то слева или справа добавляются пробелы для достижения указанного значения.

Точность задаёт число подлежащих выводу десятичных знаков и должно начинаться точкой. Действие поля зависит от типа данных (табл. 7).

Таблица 7

Символ	Действие
d, i, u, o, x, X	Указывает минимальное число выводимых цифр
e, E, f	Указывает число цифр, которые выводятся после десятичной точки. Последняя цифра округляется
g, G	Выводится указанное число значащих цифр
C	Не действует. Выводится соответствующий символ
S	Указывает максимальное число выводимых символов

Далее следует один из символов преобразования (тип формата) (табл. 8).

Таблица 8

Тип формата	Представление данных при выводе
c	Отдельный символ
s	Символьная строка
d, i	Целое десятичное число
u	Целое беззнаковое десятичное число
o	Целое беззнаковое восьмеричное число
x	Целое беззнаковое шестнадцатеричное число (для вывода используются символы 0…f)
X	Целое беззнаковое шестнадцатеричное число (для вывода используются символы 0…F)
f	Число с плавающей запятой в записи с фиксированной десятичной точкой
e	Значение со знаком в форме [–]d.dddde[+|–]ddd
E	Значение со знаком в форме [–]d.ddddE[+|–]ddd
g	Значение со знаком в формате 'e' или 'f', в зависимости от значения и указанной точности
G	Значение со знаком в формате 'E' или 'f' в зависимости от значения и указанной точности
p	Указатель
N	Число записанных на данный момент символов
[...]	Соответствует самой длинной строке, которая состоит из перечисленных в скобках символов
[^...]	Соответствует самой длинной строке, которая не содержит перечисленных в скобках символов
%	Сам знак «%», преобразование не производится

Перед типом формата могут стоять модификаторы типа (табл. 9).

Таблица 9

Модификтор	Значение
h	Если тип формата d, i, o, x или X, то тип параметра – short int. При типе формата u тип параметра – unsigned short int
l	При типе формата d, i, o, x или X тип параметра – long int, при u – unsigned long. При типе формата e, E, f, g или G тип параметра – double вместо float
l	При типе формата e, E, f, g или G тип параметра – long double

Аргументами функции **printf** могут быть переменные, константы, выражения, вызовы функции. Основным требованием правильной работы функции является соответствие выводимой информации и спецификаций, описывающих эту информацию. В противном случае результат будет неверным и компилятор выдаст сообщение об ошибке.

Рассмотрим пример использования функции **printf.**

```
int i = 32;              // переменная целого типа
float f = 132.45;        // переменная вещественного типа
double d = −132e3;  // переменная удвоенной точности
int t = 0xF;             // объявление целого типа
printf("\n i= %d f= %f d= %lf",i,f,d);
printf("\n i= %d f= %1.3f d= %.4e",i,f,d);
printf("\n t= %#x",t);
```

Далее рассмотрим описание функции ввода данных scanf(), имеющей следующий формат записи:

int scanf("управляющая строка",аргумент1,аргумент2,...);

По виду функции **printf** и **scanf** отличаются только названием. Обе имеют управляющую строку и список аргументов. Основное отличие этих функций состоит в особенностях этого списка. Функция **printf** использует имена переменных, константы и выражения, в то время как функция **scanf** только указатели на переменные. При этом надо помнить следующее:

− если требуется ввести некоторое значение и присвоить его переменной одного из рассмотренных выше типов, то перед именем переменной требуется поставить символ & (адрес);

− если для ввода используется указатель (например, на массив или строку), то перед ним ставить символ & не нужно.

Управляющая строка содержит спецификацию преобразования, в записи которой могут быть использованы также пробелы, символы табуляции и символ перехода на новую строку. Формат спецификации преобразования аналогичен функции **printf**. В спецификации преобразования допустимы следующие символы:

1) d или i – ожидается ввод десятичного числа;
2) o – ожидается ввод восьмеричного целого числа;
3) x – ожидается ввод шестнадцатеричного целого числа;
4) u – ожидается ввод беззнакового числа;

5) c – ожидается ввод одиночного символа;
6) s – ожидается ввод символьной строки;
7) e, f или g – ожидается ввод вещественного числа;
8) p – ожидается ввод указателя (адреса) в виде шестнадцатеричного числа;
9) n – получается значение, равное числу прочитанных символов из входного потока на момент обнаружения %n;
10) [] – сканируется множество символов.

Перед символами d, o, x, f может стоять буква l. В первых трёх случаях переменные, которым присваивается вводимое значение, должны иметь тип **long**, в последнем – тип **double**. В отличие от функции printf спецификация преобразования функции **scanf** имеет следующие особенности:

- отсутствует спецификация %g;
- спецификации %f и %e эквивалентны;
- для ввода целых чисел типа **short** применяется спецификация %p.

Ниже приведен пример использования функции **scanf**().

```
long l;
float f;
unsigned int k;
char st[10], ss;
scanf("%ld %f %u %c\n%s",&l,&f,&k,&ss,st); // ввод 4 переменных и строки
```

3.2. Посимвольный ввод/вывод

Для ввода/вывода символов могут быть использованы также функции **getchar**() и **putchar**(). За одно обращение к функции **getchar**() (**putchar**()) осуществляется ввод (вывод) одного символа из (в) текстового потока. Текстовый поток – это последовательность символов, разбитая на строки. Пример записи функции **getchar** (**putchar**) имеет следующий вид.

```
char c;
. . .
c=getchar();
putchar(c);
```

Наряду с функцией **getchar** имеется функция **getch**. От **getchar** она отличаются тем, что вводит символ без отображения его на экране.

Рассмотрим простейший пример программы, осуществляющей копирование по одному символу с входного потока в выходной.

```
#include <stdio.h>
int main()                  // пример ввода/вывода символа
{
   int c;
   c=getchar();            // ввод символа
   while(c!=EOF)           // пока не нажата комбинация Ctrl+z
   {
      putchar(c);          // вывод символа
      c=getchar();         // ввод очередного символа
   }
   return 0;
}
```

В рассматриваемом примере переменная для вводимого символа имеет тип **int**, хотя для символа достаточно переменной типа **char**. Это делается для того, чтобы иметь возможность окончить ввод символов при прочтении из входного потока символа **EOF** (end of file). Для прочтения символа **EOF** недостаточно переменной типа char (один байт). Символ **EOF** выбран таким образом, чтобы он был отличен от любых других символов.

4. Директивы препроцессора

4.1. Директива препроцессора #include

Директива препроцессора #**include** вставляет текст внешнего файла в текст программы, содержащей директиву. Директива имеет один из следующих форматов:

#include<имя файла>

#include "имя файла"

Файл, указываемый в директиве #**include**, должен быть в формате ASCII. Угловые скобки (<имя файла>) указывают препроцессору искать включаемый файл в каталогах, определенных по умолчанию в IDE. Двойные кавычки ("имя файла") указывают вначале производить поиск в текущем каталоге, а затем, если файл не найден, – в тех же каталогах, что и с угловыми скобками. Директива #**include** чаще всего используются для *включения заголовочных файлов*. Заголовочные файлы представляют собой библиотечные функции и имеют расширение «.h».

4.2. Директива #define

Директива предназначена для выполнения макроописаний и реализует функцию «найти и заменить». Директива имеет следующий формат:

#define argument1 argument 2

Здесь argument1 – идентификатор, не содержащий пробелов, argument2 – может содержать любые символы. Директива заменяет все вхождения argument1 в тексте программы на argument2.

Директива препроцессора позволяет определять и изменять константы. Заменяемые в ней аргументы называют *предопределенными константами*. Определить можно константу любого типа, включая строкового.

5. Операторы языка С (С++)

Все операторы языка С (С++) могут быть условно разделены на следующие категории:

− условные операторы, к которым относятся оператор условия **if** и оператор выбора **switch**;

− операторы цикла (**for**, **while**, **do while**);

− операторы перехода (**break**, **continue**, **return**, **goto**);

− другие операторы (оператор «выражение», пустой оператор).

Операторы в программе могут объединяться в составные операторы с помощью фигурных скобок. Любой оператор в программе может быть помечен меткой, состоящей из имени и следующего за ним двоеточия.

Все операторы языка С (С++), кроме составных операторов, заканчиваются точкой с запятой «;».

5.1. Понятие пустого и составного операторов

Составной оператор – это несколько операторов, объединённых в блок с помощью фигурных скобок {} или разделённых запятой. Такой блок можно рассматривать как один оператор, например:

```
#include<stdio.h>
int main()
{
   {     //   первый вариант примера составного оператора
      int i=1, j=2, k;
      k=i+j;
      printf("\n  k=i+j =  %d", k);
      k*=2;
      printf("\n  k=  %d",  k);
   }
   {    //   второй вариант примера составного оператора
      int i=0, j, k;
      k=i+=j, j=4;
   }
   return 0;
}
```

Пустой оператор состоит только из знака «;», например:

```
#include<stdio.h>
int main()
{
   int i= 2;
   i*=3; ;          //   обычный и пустой операторы
   ;                //   пустой оператор
   ; ;              //   двойной пустой оператор
   return 0;
}                   //   конец составного оператора
```

5.2. Условные операторы

В языке С (С++) существует группа операторов, позволяющая организовать ветвление в программе. В С (С++) имеются три оператора, которые могут нарушить простой линейный характер выполнения программы. К ним относятся **if** … **else** (*оператор ветвления*), **switch** (*переключатель*) и **goto** (*безусловный переход*).

Оператор ветвления предназначен для выбора в программе из нескольких возможных вариантов единственного варианта продолжения вычислительного процесса. Выбор выполняется исходя из результатов анализа значения некоторого выражения.

Оператор **if** имеет следующую общую форму записи:

```
if (проверка условия)
{группа операторов 1 }
else
{группа операторов 2 }
```

При выполнении оператора **if** сначала выполняется проверка условия. Если результат – истина (любое отличное от нуля значение), то выполняется группа операторов 1.

Если результат анализа условия – ложь (равен нулю), то выполняется группа операторов 2. Если слово **else** отсутствует, то управление передается на первый оператор, следующий после группы операторов 1.

В качестве условия может использоваться арифметическое, логическое выражение, выражение сравнения, целое число, переменная целого типа, вызов функции с соответствующим типом возвращаемого значения.

Ниже приведен пример функции, использующей оператор **if ... else**:

```
#include <stdio.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian" );
   short i;
   printf("введите число i \n i= ");
   scanf("%d",&i);
   printf("введённое число ");
   if (i>10) printf("больше 10 ");
   else printf("меньше 10 ");
   return 0;
}
```

Если необходимо осуществить выбор из более чем двух условий, то можно конструкцию **if ... else** расширить конструкцией **else if**. Это позволяет ввести в анализ дополнительное условие, что приведет к увеличению ветвления в программе.

Одним из условий правильности работы программы, использующей конструкции вида **if ...; else ...; if ...; else** ...; является соблюдение следующих правил:

− каждый оператор **else** соответствует ближайшему оператору **if**, если

только ближайший **if** не входит в группу операторов, выполняющихся при условии верхнего (предыдущего **if**);

− условия проверяются в том порядке, в котором они перечислены в программе;

− если одно из условий истинно, то выполняется оператор, соответствующий этому условию, а проверка оставшихся условий не производится;

− если ни одно из проверенных условий не дало истинного результата, то выполняются операторы, относящиеся к последнему **else**;

− последний **else** является необязательным, следовательно, он и относящийся к нему оператор могут отсутствовать.

Если анализируется более одного логического условия, то они заключаются в круглые скобки () и разделяются логическими операциями || (ИЛИ) и && (И).

Понятия «истина» и «ложь». В языке С (С++) каждое выражение, в том числе и условное, имеет значение. При этом каждое условное значение принимает либо истинное, либо ложное значение.

```
#include <stdio.h>
int  main()
{
   int tru, fals;
   tru=(4>2);          // истина
   fals=(3>5);         // ложь
   printf(”true=%d  false=%d”,tru,fals);
   return 0;
}
```

Результат работы программы:

```
true = 1   false = 0
```

Следовательно, значение «истина» – это 1, а «ложь» – 0. Более того, как следует из приводимого ниже примера, только значение 0 соответствует ложному высказыванию, при остальных значениях выражение принимает истинное значение.

```
#include <stdio.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian" );
   if (1) printf("истина1");
   else printf("ложь1");
   if (0) printf("истина 0");
   else printf("ложь 0");
   if (−2) printf("истина 2");
   else printf("ложь 2");
   return 0;
}
```

Это свойство может быть использовано для сокращения логических условий. Например, выражения if(k!=0) и if(k) эквивалентны, т. к. оба принимают значение «ложь» только в случае, если k равно нулю.

Необходимо различать следующие высказывания:

if(k=3) и if(k==3).

Первое из них имеет всегда истинное значение, т. к. в результате его выполнения переменная k принимает значение 3, и, следовательно, выражение в скобках не равно 0 (истинно всегда). Во втором случае выражение истинно только в случае, когда k примет значение 3.

Переключатель switch. Использование оператора **if** позволяет выбрать один из двух путей. В отличие от него оператор **switch** используется для выбора одного из многих путей. Общая структура оператора **switch** имеет следующий вид:

```
switch (выражение)
{
   case константа 1:  вариант 1;  break;
                .  .  .  .  .
   case  константа n:  вариант n;  break;
   [default: вариант n+1; break;]
}
```

В операторе **switch** производится проверка на совпадение вычисленного значения выражения с одним из значений, входящих во множество констант. Константа представляет собой целочисленную константу или константное выражение (символьные константы автоматически преобразуются в целочисленные). В операторе **switch** не должно быть одинаковых констант. В процессе выполнения оператора **switch** выражение сравнивается с каждой из констант. При совпадении выполняется соответствующий константе вариант (один или более операторов), иначе – вариант, помеченный ключевым словом **default**, а при его отсутствии выполняются операторы, следующие за «}». Оператор **break** приводит к немедленному выходу из **switch** (или далее из ниже рассматриваемых операторов цикла). Кроме **break** для выхода из **switch** может быть использован также **return.**

Ниже приведены несколько примеров программ, в которых используется переключатель **switch.**

```
#include <stdio.h>
int main()
{
   int i;
   char c;
   while((i=getchar())!=EOF)    // контроль на нажатие Ctrl + z – выход
   {
      switch (i)
      {
         case 97: c='a';  // из-за отсутствия оператора break выполняется
                          // следующая ветвь и с принимает значение 'b'
         case 98: c='b'; break;
         case 99: c='c'; break;
         default: c=NULL; return;   // выход из функции
      }
      printf("c= %c\n",c);
   }
   return 0;
}
```

Пример простейшего калькулятора:

```
#include <stdio.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian" );
   float a, b, res;   int  prizn= 1;
   char opr;
   do
   {
      puts("\nВведите выражение  (число_1  знак  число_2 ):");
      scanf("%f%c%f", &a, &opr, &b);
   } while( ( b == 0 ) && (( opr ==’/’) || (opr == ‘:’));
   switch (opr)
   {
      case '+':  res=a+b; break;
      case '–':  res=a–b;  break;
      case '*':   res=a*b;  break;
      case ':': case '/': res=a/b;  break;
      default: printf("\операция не определена"); pr= 0;
   }
   if(pr)  printf("\n Результат: %f",   res);
   return 0;
}
```

Операция условия «?:». В языке С (С++) имеется короткий способ записи оператора **if ... else**. Он называется «условным выражением» и записывается с помощью операции «?:». Эта операция состоит из двух частей и содержит три операнда. Общий вид условного выражения следующий:

переменная=операнд1?операнд2:операнд3;

Операнд1 вычисляется с точки зрения его эквивалентности нулю. Он может быть целого, плавающего или адресного типа. Если операнд1 имеет ненулевое значение, то вычисляется операнд2 и результатом условной операции является значение выражения операнда2. Если операнд1 равен нулю, то вычисляется операнд3 и переменная принимает вычисленное значение. Вычисляется только один из операндов. Например:

```
n=(k>3)?n–k:n+5;
```

То есть если k>3, то n=n–k, иначе n=n+5.

В качестве операндов 2 и 3 могут быть использованы более сложные выражения, например:

```
i=j>1?(j=4*n,k+=j):(n--,++m*=2);
```

5.3. Операторы организации цикла

Цикл предназначен для организации повторения выполнения некоторой инструкции (группы инструкций) некоторое число раз, например суммы чисел некоторого массива. Таким образом, под *циклом* понимается оператор или группа операторов, повторяющихся некоторое количество раз. Каждый проход по телу цикла называется *итерацией*.

Цикл называется простым, если в его теле не содержится других циклов, иначе цикл называют сложным. В любом циклическом процессе в ходе вычислений необходимо решать вопрос: требуется ли выполнять очередную итерацию. Ответ на этот вопрос получают из анализа некоторого условия. Таким образом, циклический процесс является разветвляющимся процессом с двумя ветвями, из которых одна возвращается на предыдущие блоки, т. е. реализует цикл.

Если в цикле выполняется группа инструкций, то они должны быть выделены в блок (заключены в фигурные скобки). В языке С (С++) существуют три оператора организации циклов: **for** (для), **while** (пока) и **do...while** (делать пока).

5.3.1. Оператор цикла for

Оператор **for** формально записывается в следующем виде:

for(выражение1;выражение2;выражение3)
{тело цикла;}

Тело цикла составляют одна либо некоторое подмножество инструкций, заключённых в фигурные скобки. В выражениях 1, 2, 3 фигурирует переменная, называемая управляющей. В операторе **for** устанавливается нижняя и верхняя граница изменения переменной цикла и величина (шаг) её изменения. Каждое из выражений в круглых скобках разделены точкой с запятой. Первое выражение служит для инициализации управляющей переменной. Оно выполняется только один раз в начале выполнения цикла. Выражение 2 устанавливает условие, при котором цикл **for** повторяет свое выполнение. Проверка условия осуществляется перед каждым выполнением цикла. Выражение 3 задаёт приращение (уменьшение) управляющей переменной. Следует также отметить, что тело цикла может не содержать ни одной инструкции. В этом случае цикл может использоваться, например, для реализации временной задержки:

```
for (k=1; k<500; k++);
```

В отличие от аналогичных операторов цикла других языков в С (С++) оператор **for** имеет некоторые особенности:

− в качестве управляющей переменной может использоваться не только число, но и символьная переменная, например

```
for (char c='A'; c<'H'; c++)  { . . . };
```

− цикл убывающий имеет тот же вид, что и возрастающий, но вместо c++ записывается c--, например

```
for (char c='H'; c>'A'; c--)  { . . . };
```

− выражение 2 может иметь произвольный вид:

```
for (k=1; k+j<=56; k*=2) { . . . };
```

− выражение 3 может быть любым правильно составленным выражением:

```
for (k=1; k>45; k=(j*4)+++3) { . . . };
```

− выражение 1 может осуществлять инициализацию более чем одной переменной, а в выражениях 2 и 3 могут анализироваться и (или) изменяться более одной переменной:

```
for (i=1, j=2; i>10 && j>=i; i++, j=i+2)  {. . .};
```

− выражение 1 может не выполнять функцию инициализации управляю-

щей переменной. Вместо этого там может стоять оператор специального типа (например printf)

```
for (printf("\nвведите число : "); k!=10;)
scanf("%d",&k);
```

− любое из трёх выражений цикла **for** может отсутствовать, однако «;» должна оставаться. Если нет выражений 1 или 3, то управляющая переменная не используется. Если отсутствует выражение 2, то считается, что оно истинно, и цикл не оканчивается. Таким образом,

```
for (;;) {. . .}
```

есть бесконечный цикл, выход из которого осуществляется принудительно.

Рассмотрим примеры программ, в которых используется цикл **for**.

Пример вычисления факториала числа (n!).

```
#include<stdio.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian" );
   unsigned int a, n;
   unsigned long  f=1;
   printf("Введите n: ");
   scanf("%d",&n);                    // ввод числа для нахождения факториала
   if(n)                              // проверка на то, что n!=0
      for( a=2; a<=n; f*=a,  a++ ) ;  // цикл вычисления факториала
   printf("\n%u! = %lu", n,f);
   return 0;
}
```

Пример программы вычисления суммы чётных чисел и количества нечётных чисел, вводимых с клавиатуры. Ввод чисел окончить при получении суммы 30 или вводе заданного количества чисел.

```
#include <stdio.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian" );
   int i, j, k, n, sm;
   n=0;
   printf(" введите количество чисел =");
   scanf("%d",&k);
   for(i=0, sm=0; i<j && sm<30; i++)  // в выражении 2 используется
                                      // операция &&, если вместо && поставить ',' то
                                      //  цикл выполняется только при условии     j<3
   {
      scanf("%d",&k);
      if ( !( k%2 ) ) sm+=k;     //  накапливается сумма чётных  чисел
      else n++;                  //  иначе увеличивается число нечётных чисел
   }
   printf("  сумма=%d,  количество=%d \n",sm , n );
   return 0;
}
```

5.3.2. Оператор цикла while

Общий формат записи оператора **while** имеет вид

while (выражение)
{тело цикла}

В качестве выражений наиболее часто используются условные выражения, однако это могут быть выражения произвольного типа. Принцип работы оператора **while** состоит в следующем. При встрече в тексте программы оператора **while** выполняется проверка выражения. Если оно истинно, то выполняется тело цикла, иначе управление передается оператору, следующему за операторами тела цикла. Характерной особенностью цикла **while** по сравнению с циклом **for** является необходимость инициализации счётчика (переменной выражения) до начала действия цикла.

Оба рассмотренных цикла являются циклами с предусловием, т. е. проверка условия осуществляется перед началом каждой итерации. В С (С++) имеется конструкция цикла с постусловием: **do**...**while**.

5.3.3. Оператор цикла do…while

В отличие от цикла **while** в цикле **do…while** условие проверяется после каждой итерации (повтора). В общем виде цикл **do…while** записывается в следующем виде:

do	**do {**
оператор тела цикла;	**группа операторов тела цикла;**
while(выражение);	**} while(выражение);**

Тело цикла **do**...**while** всегда выполняется, по крайней мере один раз. Цикл прекращается, если выражение в скобках принимает ложное значение.

5.3.4. Вложенные циклы

Вложенным называется цикл, находящийся внутри другого цикла. Ограничений на вложенность циклов нет. Ниже приведен пример вложенного цикла.

```
for (i=1; i<5; i++)                    // внешний цикл
for (j=1; j<10; j++) { . . . };        // вложенный цикл
```

В данном фрагменте вложенный цикл (по переменной j) выполняется 50 раз.

5.4. Операторы перехода (break, continue, return, goto)

Оператор **break** обеспечивает прекращение выполнения операторов **switch**, **do**, **for**, **while**. После выполнения оператора **break** управление передаётся первому внешнему оператору.

Оператор **continue** используется только внутри операторов цикла. При его встрече выполнение текущей итерации цикла прерывается и выполняется переход к его очередной итерации. Рассмотрим пример, в котором суммируются только чётные числа, вводимые с клавиатуры.

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

```
int main()
{
   setlocale(LC_ALL,"Russian" );
   int i, k, sm=0;
   for(i=0; i<30; i++)  // вводится 30 чисел с клавиатуры
   {
      scanf("%d",&k);
      if ( !( k%2 ) ) { sm+=k;   continue; } //  накапливается сумма чётных  чисел
      printf("%3d",i);                        // выводится на экран нечётное число
   }
   printf("\nсумма чётных чисел=%d\n",sm);
   return 0;
}
```

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

return [выражение] ;

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие необязательно.

Если в какой-либо функции отсутствует оператор **return**, то передача управления в вызывающую функцию происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то её нужно объявлять с типом **void**.

Использование оператора безусловного перехода **goto** в практике программирования на языке С (С++) настоятельно не рекомендуется, т. к. он затрудняет понимание программ и возможность их модификаций.

Формат этого оператора следующий:

```
goto имя_метки;
. . .
имя_метки:  оператор;
```

Оператор **goto** передаёт управление на оператор, помеченный меткой имя_метки. Помеченный оператор должен находиться в той же функции, что и оператор **goto**, а используемая метка должна быть уникальной, т. е. одно имя_метки не может быть использовано для разных операторов программы. Имя_метки – это идентификатор.

Любой оператор в составном операторе может иметь свою метку. Используя оператор **goto**, можно передавать управление внутрь составного оператора. Но нужно быть осторожным при входе в составной оператор, содержащий объявления переменных с инициализацией, т. к. объявления располагаются перед выполняемыми операторами и значения объявленных переменных при таком переходе будут не определены.

5.5. Примеры программ

Ниже рассматриваются примеры программ, демонстрирующих выполнение операций и основных операторов языка C/C++.

Пример 1. Разработать программу перевода числа из десятичной системы счисления в систему с основанием, меньшим либо равным 33.

```
#include <stdio.h>
#include <iostream>
int main()
{
   int k1 = 10, k2 = 11, t, z ,i, ss;
   char znak = '+'; // для знака результата
   double num;     // число в десятичной системе счисления
   char ms[20];    // массив (обнуляется при описании)
   setlocale(LC_ALL,"Rus");  // подключение русского языка
   printf("\n Введите исходное число (между целой и др. частью - ,) :");
   scanf("%lf",&num);
   num<0 ? znak = '–', num *= –1 : num;  // получение знака числа
   printf("\n Введите основание новой с/с :");
   scanf("%d",&ss);
   printf("\n Введите точность для дробной части числа : ");
   scanf("%d",&t);
   z = num;         // выделение целой части числа
   num –= z;        // выделение дробной части числа
   while(z >= ss) // перевод целой части числа
   {
      i = z % ss;   // получаем остаток от деления на основание
      z /= ss;      // выделяем целую часть от деления
      ms[k1--] = i > 9 ? i – 10 + 'A' : i + '0';  // формирование символа цифры
   }                                              // числа в новой системе счисления
   ms[k1--] = z > 9 ? z – 10 + 'A' : z + '0';
   ms[k1] = znak;
   ms[k2++] = '.';                               // выставление десятичной точки
   while(num != 0 && k2 – 11 <= t)           // перевод дробной части числа
   {
      num *= ss;
      i = num;                                     // в i записывается целая часть от dr
      ms[k2++] = i > 9 ? i – 10 + 'A' : i + '0';   // i заносится в массив
      num –= i;                                    // отбрасываем целую часть
   }
   printf("\n Ваше число : ");
   for (i = k1; i < k2; i++)
      printf("%c",ms[i]);  // вывод числа на экран
   return 0;                  // функция возвращает 0 (завершение программы)
}
```

Пример 2. Разработать программу нахождения простых чисел в заданном интервале.

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

```
    int num1, num2, i, del;
    setlocale(LC_ALL,"Rus");   // подключение русского языка
    do
    {
       printf("Введите нижнюю и верхнюю границы интервала : \n");
       scanf("%d%d",&num1,&num2);
       if (num1 >= num2)
          printf("\n Ошибка при вводе границ! Повторите ввод! \n");
    }while(num1 >= num2);
    printf("\nСписок простых чисел : ");
    for(i = num1; i < num2; i++)   // перебор всех чисел на заданном интервале
    {
       for(del = 2; del < i; del++)  // перебор всех чисел от 2 до i
          if (i % del == 0)             // если i разделится на del без остатка, то
                        break;        // i не простое число
       if (del == i) printf("%d ",i);  // вывод простого числа i, если del=i
    }
    return 0;       // функция возвращает 0 (завершение программы)
}
```

Пример 3. Ввести целые числа m и n. Определить, являются ли они взаимно простыми, т. е. не имеют общих делителей.

```
#include <conio.h>
#include <stdio.h>
#include <iostream>
int main()
{
    int i, m, n, min;
    setlocale(LC_ALL,"Rus"); // подключение русского языка
    printf("Введите m и n: ");
    scanf("%d%d",&m,&n);     // ввод переменных m и n
    if (m < n) min = m;              // поиск меньшего числа
    else min = n;
    for (i = 2; i <= min / 2; i++) // поиск делителя в интервале [2,min/2]
       if (m % i == 0 && n % i == 0) // если оба разделились на i без остатка,
       {                                  // то i – общий делитель
          printf("\n Числа %d и %d не взаимно простые. ",n,m);
          printf("Общий делитель: %d. \n",i);
          return 0;
       }
    if (m % min == 0 && n % min == 0) // проверка, что min – общий делитель.
    {
       printf("\n Числа %d и %d не взаимно простые. ",n,m);
       printf("Общий делитель: %d. \n",i);
    }
    else
       printf("\n Числа %d и %d – взаимно простые.\n",n,m);

    return 0;       // функция возвращает 0 (завершение программы)
}
```

Пример 4. Найти совершенные числа на отрезке [N1;N2]. Совершенное число – число, равное сумме всех своих делителей, включая 1, но не включая самого себя.

```
#include <conio.h>
#include <stdio.h>
#include <iostream>
int main()
{
   int num1, num2, del, i, sm;
   setlocale(LC_ALL,"Rus");
   do
   {
      system("cls");         // очистка консоли
      puts("\nВведите границы интервала для поиска: ");
      if (scanf("%d%d",&num1,&num2) != 2)   // ввод границ интервала
      {
         fflush(stdin);      // очистка буфера ввода/вывода
         continue;
      }
      if (num1 > num2)
        puts("\nОшибка ввода!");
   }while(num1 >= num2);
   printf("\n Совершенные числа : ");
   for (i = num1; i < num2; i++) // поиск совершенных чисел
   {                             // проход по всему интервалу
      sm = 0;                    // "накапливатель" делителей
      for (del = 1; del < i; del++)
         if (i % del == 0)       // если остаток от деления равен 0,
            sm += del;           // то прибавляем del к sm
      if (sm == i && i != 0)     // если sm=i,
         printf(" %d",i);        // то найдено совершенное число i
   }
   return 0;
}
```

6. Массивы и указатели

6.1. Одномерные массивы

Массивы – объекты структурированного типа данных. Массив представляет собой совокупность элементов одного типа данных. Чтобы создать массив, необходимо компилятору сообщить тип элементов, образующих массив, и требуемый класс памяти. Кроме того, для отведения памяти «под» массив должно быть известно количество элементов массива. Определение массива имеет два формата:

[класс памяти] спецификатор_типа идентификатор [константное_выражение];
[класс памяти] спецификатор_типа идентификатор [ ];

Идентификатор – это имя массива. Спецификатор_типа задаёт тип элементов объявляемого массива. Константное_выражение в квадратных скобках задаёт количество элементов массива. Константное_выражение при объявлении массива может быть опущено, если при объявлении массив инициализируется.

В языке С (С++) принято, что имя массива – это адрес памяти, начиная с которого расположен массив, т. е. адрес первого элемента массива.

Рассмотрим пример описания массивов:

```
int mas1[]={1,2,3,4};          //  определение и инициализация массива
short mas2[3];                 //  определение автоматического массив
static float mas3[2]={0,0};    //  инициализация статического массива
```

Доступ к элементу массива осуществляется указанием его номера (индекса) в массиве. Например:

```
int mas[9]={0};   // все элементы массива mas инициализируются нулем
for(i=0; i<9; i++) mas[i]=mas[i]+k;
```

Необходимо помнить следующее: индексация элементов массива начинается с номера 0. Число в квадратных скобках в объявлении массива говорит о количестве элементов в массиве, таким образом, максимальный номер элемента будет на единицу меньше их общего числа.

6.2. Примеры программ

Рассмотрим некоторые примеры программ, демонстрирующие работу с одномерными массивами.

Пример 1. В программе вводятся с клавиатуры десять чисел, а затем выводятся на экран в обратном порядке.

```
#include <stdio.h>   // подключение библиотек;
#include <locale.h>
#include <windows.h>
#define  num 10      // глобальное объявление размера массива;
int main()
{
   int i=0,mas[num];    // объявление массива чисел и переменной счётчика
   setlocale(LC_ALL,"Russian"); // русификация консоли
   printf("Введите %d целых чисел",num);
   for(i=0;i<num;i++)   // цикл для ввода массива чисел
   {
      printf("\nmas[%d]:",i);
      scanf("%d",&mas[i]);
```

```
    }
    printf("\nВ обратном порядке:");
    for(i=num−1;i>=0;i--)   // цикл для вывода массива чисел
    {
       printf("%5d",mas[i]);
    }
    Sleep(1000);            // задержка результата программы на экране
    return 0;
}
```

Пример 2. Ввести массив чисел и вычислить сумму элементов массива.

```
#include <stdio.h>          // подключение библиотек
#include <conio.h>
#include <locale.h>
#include <windows.h>
int main( )
{
    int mas[10],i=0,sum=0;          // объявление массива чисел на 10 элементов;
    setlocale(LC_ALL,"Russian");   // русификация консоли
    printf("Введите 10 целых чисел");
    for(i=0;i<10;i++)    // цикл для ввода и суммирования элементов массива;
    {
       printf("\nmas[%d]:",i);
       scanf("%d",&mas[i]);  // ввод элементов массива
       sum+=mas[i];          // суммирование элементов массива
    }
    printf("\nСумма элементов массива=%d",sum); // вывод суммы на экран;
    Sleep(1000);              // задержка результата программы на экране;
    return 0;
}
```

Пример 3. Ввести массив чисел. За один просмотр массива определить, сколько чисел в какой десяток попадает. Операторы if и switch не использовать. Количество вводимых чисел <=100.

```
#include<stdio.h>    // подключение библиотек;
#include<locale.h>
#include<windows.h>
int main()
{
    int mas[15],num[15],i=0; // массивы: num-порядки,mas-вводимые числа
    setlocale(LC_ALL,"Russian"); // русификация консоли
    for(i=0;i<15;i++)          // цикл для обнуления массива счётчика порядков
    {
       num[i]=0;
    }
    printf("Введите 15 целых чисел");
    for(i=0;i<15;i++) // цикл для ввода массива чисел
    {
       printf("\nmas[%d]:",i);
       scanf("%d",&mas[i]);
    }
    for(i=0;i<15;i++) // цикл для нахождения количества порядков
```

```
    {
        num[mas[i]/10]++; // увеличение счётчика, соответствующего
    }                     // порядку i-го числа массива
    for(i=0;i<15;i++)     // вывод результата
    {
        printf("\nЧисел в %d-м десятке: %d",i+1,num[i]);
    }
    Sleep(1000);          // задержка результата программы на экране
    return 0;
}
```

Пример 4. Ввести массив чисел. Определить k (число k ввести с клавиатуры) максимальных элементов массива и вывести их на экран.

```
#include<stdio.h>     // подключение библиотек
#include<locale.h>
#include<windows.h>
#define n 15          // размерность массива
int main()
{                     // объявление переменных
    int mas[n],max[100],k=0,i=0,j=0,j1=0,check_1=0,check_2=0;
    setlocale(LC_ALL,"Russian"); // русификация;
    printf("Введите k:");
    scanf("%d",&k);
    printf("\nВведите массив");
    for(i=0;i<n;i++)      // цикл для ввода массива
    {
        printf("\nmas[%d]=",i);
        scanf("%d",&mas[i]);
    }
    for(i=0;i<k;i++)      // установка элементов max на −1
    {
        max[i]= −1;
    }
    for(i=0;i<k;i++)      // поиск номеров максимальных элементов
    {
        for(j=0;j<n;j++)  // выбор начального максимального элемента
        {
            for(check_1=0;max[check_1]!=-1 && max[check_1]!=j && check_1<k;
                check_1++);
            if(max[check_1]== −1) // проверка
            {
                max[check_1]=j;      // стартовый индекс поиска
                break;
            }
        }
        for(j1=j;j1<n;j1++)          // поиск номера максимального элемента
        {
            if(mas[j1]>mas[max[check_1]])
            {
                for(check_2=0;max[check_2]!=j1 && check_2<check_1; check_2++);
                if(max[check_2]!=j1)
                {
```

```
                max[check_1]=j1;
            }
        }
    }
}
printf("\nМассив :");
for(i=0;i<n;i++)          // вывод исходного массива
{
   printf("%4d",mas[i]);
}
printf("\nНайдено %d  max чисел : ",k<n?k:n);
for(i=0;i<k&&i<n;i++)   // вывод k максимальных элементов
{
   printf ("%3d",mas[max[i]]);
}
Sleep(1000);              // задержка результата программы на экране
return 0;
}
```

Пример 5. Ввести два массива вещественных чисел. Второй массив упорядочен по возрастанию своих значений. Определить, какие числа первого массива содержатся во втором массиве.

```
#include<stdio.h>      // подключение библиотек
#include<conio.h>
#include<locale.h>
#include<windows.h>
int main()
{
   int i=0,midle=0,start=0,end=0,n1=0,n2=0,fl=0;
   float mas1[100],mas2[100];
   setlocale(LC_ALL,"Russian"); // русификация консоли
   do
   {
      printf("Введите размеры массивов");  // ввод размеров
      printf("\nmas1=");
      scanf("%d",&n1);
      printf("\nmas2=");
      scanf("%d",&n2);
   }while(n1<0 || n2<0 || n1>100 || n2>100);
   printf("\nВведите первый массив: \n");
   for(i=0;i<n1;i++)     // ввод первого массива
   {
      printf("\nmas1[%d]=",i);
      scanf("%f",&mas1[i] );
   }
   printf("\nВведите второй массив, упорядоченный по возрастанию: \n");
   for(i=0;i<n2;i++)     // ввод второго массива
   {
      printf("\nmas2[%d]=",i);
      scanf("%f",&mas2[i] );
   }
```

```
    if(n1==n2 && n1==1) // проверка при 1
    {
       if(mas1[0]==mas2[0])
       {
          printf("\nЧисло %f входит в оба массива.", mas1[0]);
       }
       else
       {
          printf("\nЧисло %f не входит в массив mas2.", mas1[0]);
       }
       Sleep(1000);
       return 0;
    }
    for(i=0;i<n1;i++)  // выбор числа из исходного массива mas1
    {
       start=0;          // границы интервала поиска
       end=n2;
       while(start<end−1)
       {
          fl=0;
          midle=(start+end)/2;     // середина интервала
          if(mas1[i]==mas2[midle])
          {
             printf("\nЧисло %f входит в оба массива.", mas1[i]);
             break;
          }
          if(mas1[i]<mas2[midle]) // исходный элемент находится
          {
             end=midle;     // слева; сдвиг конца интервала влево
          }
          else
          {
             start=midle;    // справа; сдвиг начала интервала вправо
          }
          if(start==end−1) // проверка последнего элемента
          {
             if(mas1[i]==mas2[start])
             {
                printf("\nЧисло %f входит в оба массива.", mas1[i]);
                fl=1;
             }
             break;
          }
       }
       if(mas1[i]!=mas2[midle] && fl==0)
       {
          printf("\nЧисло %f не входит в массив mas2.",mas1[i]);
       }
    }
    return 0;
}
```

Пример 6. Ввести первый массив, расположив его элементы по убыванию, второй – по возрастанию. Не используя сортировок, сформировать третий массив по возрастанию. Все массивы целочисленные.

```
#include <stdio.h>      // подключение библиотек
#include <locale.h>
#include <windows.h>
#define n 3
#define m 5
int main()
{
   int mas1[n],mas2[m],mas3[n+m],i1,i2,i3; // объявление переменных
   setlocale(LC_ALL,"Russian"); // русификация консоли
   printf("Введите массив, упорядоченный по возрастанию");
   for(i1=0;i1<n;i1++)              // ввод первого массива
   {
      printf("\nmas1[%d]=",i1);
      scanf("%d",&mas1[i1]);
   }
   printf("Введите массив, упорядоченный по убыванию");
   for(i2=0;i2<m;i2++)              // ввод второго массива
   {
      printf("\nmas1[%d]=",i2);
      scanf("%d",&mas2[i2]);
   }
   i1=0;                        // счетчик 1-го массива
   i2=m−1;                      // счетчик 2-го массива (с конечного элемента)
   i3=0;                        // счетчик 3-го массива;
   while(i1<n && i2>=0)    // пока в двух массивах есть элементы
   {
      while(mas1[i1]<=mas2[i2] && i1<n) // запись первого массива, пока его
      {                                 // элемент наименьший
         mas3[i3++]=mas1[i1++];
      }
      while(mas2[i2]<mas1[i1] && i2>=0) // запись второго массива, пока его
      {                                 //элемент наименьший
         mas3[i3++]=mas2[i2--];
      }
   }
   while(i1<n) // запись остатков массивов (если остались элементы);
   {
      mas3[i3++]=mas1[i1++]; // первого массива
   }
   while(i2>=0)
   {
      mas3[i3++]=mas2[i2--];  // второго массива
   }
   printf("\nРезультирующий массив:");
   for(i3=0;i3<(n+m);i3++)      // вывод результата;
   {
      printf("%3d",mas3[i3]);
```

```
    }
    Sleep(1000);        // задержка результата программы на экране
    return 0;
}
```

6.3. Многомерные массивы (матрицы)

В языке С (С++) определены только одномерные массивы, но поскольку элементом массива может быть массив, можно определить и многомерные массивы. Многомерный массив – это массив, элементами которого являются массивы. Размерность массива соответствует количеству индексов, используемых для доступа к элементу массива. Двухмерные массивы часто называют матрицами. Пример объявления двухмерного массива:

```
int mas[5][15];
```

Константное выражение, определяющее одну из размерностей массива, не может принимать нулевое значение:

```
int  mas[0][7]; // ошибка
```

Можно инициализировать и многомерные массивы. Инициализация выполняется построчно, т. е. в порядке возрастания самого правого индекса. Именно в таком порядке элементы многомерных массивов располагаются в памяти компьютера, например:

```
int mas[2][3]={2,14,36,23,1,9};
```

Проинициализированный таким образом массив будет выглядеть так:

```
 2   14   36
23    1    9
```

Многомерные массивы могут инициализироваться и без указания одной (самой левой) из размерностей массива. В этом случае количество элементов компилятор определяет по количеству членов в списке инициализации. Например, для массива mas будет получен тот же, что и в предыдущем примере результат:

```
int mas[][3]={2,14,36,23,1,9};
```

Если необходимо проинициализировать не все элементы строки, а только несколько первых элементов, то в списке инициализации можно использовать фигурные скобки, охватывающие значения для этой строки, например:

```
int mas[][3]={{ 0 },
              { 10, 11},
              { 21, 21, 22}};
```

Если в какой-то группе {…} отсутствует значение, то соответствующему элементу присваивается 0. Предыдущий оператор будет эквивалентен следующему определению:

```
int mas[][3]={{ 0,   0,   0 },
              { 10, 11, 0 },
              { 21, 21, 22}};
```

Отличие в работе с многомерными массивами от работы с одномерными состоит в том, что для доступа к элементу многомерного массива необходимо указать все его индексы:

```
n = mas[i][j]; // работа с элементом, расположенным в i-й
```

```
mas[i][j] = n; // строке и в j-м столбце матрицы mas
mas[i,j] = n;  // неправильный вариант
n = mas[i];    // неправильный вариант
```

В языке С (С++) можно использовать сечения массива, однако на использование сечений накладывается ряд ограничений. Сечения формируются вследствие опускания одной или нескольких пар квадратных скобок. Пары квадратных скобок можно отбрасывать только справа налево и строго последовательно. Сечения массивов используются при организации вычислительного процесса в функциях. Например:

```
int mt[2][3];
```

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

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

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

6.4. Примеры программ

Рассмотрим некоторые примеры программ с использованием многомерных матриц.

Пример 1. Транспонировать матрицу относительно главной диагонали.

```
#include <stdio.h>
#include <Windows.h>
#include <locale>
#define n 3
int main()
{
   setlocale(LC_ALL,"Russian"); // установка русского языка в консоли
   int mas[n][n], i, j, temp;          // объявление переменных
   printf("Введите матрицу");
   for(i = 0; i < n; i++)       // цикл по строкам матрицы
      for(j = 0; j < n; j++)    // цикл по столбцам матрицы
      {
         printf("\nвведите элемент mas[%d][%d] = ", i, j);
         scanf("%d", &mas[i][j]);
      }
   system("cls");               // очистка консоли
   printf("\nИсходная матрица "); // вывод матрицы на экран
   for(i = 0; i < n; i++)       // цикл по строкам матрицы
   {
      printf("\n");
      for(j=0; j < n; j++)      // цикл по столбцам матрицы
      printf("%5d", mas[i][j]);
   }
            /* Пример транспонирования матрицы, используя оператор for*/
   for(i = 0; i < n; i++)
      for(j  = i + 1; j < n; j++)
```

```
    {
        temp = mas[i][j];     // замена элемента i-й строки j-го столбца
        mas[i][j] = mas[j][i]; // на элемент j-й строки i-го столбца
        mas[j][i] = temp;
    }
        /* Пример транспонирования матрицы, используя оператор do…while*/
i = 0;
do
{
    j = i;
    do
    {
        temp = mas[i][j];     // замена элемента i-й строки j-го столбца
        mas[i][j] = mas[j][i]; // на элемент j-й строки i-го столбца
        mas[j][i] = temp;
        j++;
    }while(j < n);
    i++;
}while(i < n);
      /* Пример транспонирования матрицы, используя оператор while*/
i = 0;
while(i < n)
{
    j = i + 1;
    while(j < n)
    {
        temp = mas[i][j];     // замена элемента i-й строки j-го столбца
        mas[i][j] = mas[j][i]; // на элемент j-й строки i-го столбца
        mas[j][i] = temp;
        j++;
    }
    i++;
}
printf("\nТранспонированная матрица"); // вывод матрицы на экран
for(i = 0; i < n; i++)     // цикл по строкам матрицы
{
   printf("\n");
   for(j = 0; j < n; j++)  // цикл по столбцам матрицы
   printf("%5d", mas[i][j]);
}
return 0;     // функция возвращает 0 (завершение программы)
}
```

Пример 2. Разработать программу выявления седловой точки в матрице размерностью n строк на n столбцов. Седловой точкой в матрице называют элемент, одновременно наибольший в своей строке и наименьший в своем столбце.

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

```
#include <conio.h>
#define n 4
int main()
{
   setlocale(LC_ALL,"Russian");        // установка русского языка в консоли
   int ms[n][n], i, i1, st, j, j1, max, flag; // объявление переменных
   system("cls");                      // очистка консоли
   printf("Введите матрицу\n");
   for(i = 0; i < n; i++)              // цикл по строкам
      for(j = 0; j < n; j++)           // цикл по столбцам
      {
         printf("\nms[%d][%d]= ", i, j);
         scanf("%d", &ms[i][j]); // считывание числа с клавиатуры
      }
   system("cls");             // очистка консоли
   printf("Введенный массив MS\n");
   for(i = 0; i < n; i++)      // цикл по строкам
   {
      printf("\n");
      for(j = 0; j < n; j++)  // цикл по столбцам
         printf("%3d", ms[i][j]);
   }
   for(i = 0; i < n; i++)      // блок поиска седловой точки
   {
      max = ms[i][0];
      i1 = 0;                  // начальное значение координат в матрице
      j1 = 0;                  // для поиска седловой точки
      for(j = 0; j < n; j++)  // цикл анализа i-й строки
         if (ms[i][j] > max) // найдено новое максимальное значение в строке
         {
            max = ms[i][j]; // запоминаем найденное значение
            i1 = i;              // и его координаты в матрице
            j1 = j;
          }
      flag = 0;
      for(st = 0; st < n; st++) // проверка на минимальность числа в строке
         if(ms[i1][j1] >= ms[st][j1] && i1 != st)
            flag = 1;
      if(flag == 0)                // вывод седловой точки и её координат
      {
         printf("\nСедловая точка = %d", ms[i1][j1] );
         printf("\nкоординаты : MS[%d][%d]", i1, j1);
      }
   }
   return 0;
}
```

Пример 3. Разработать программу умножения двух целочисленных матриц чисел.

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

```
int main()
{
    setlocale(LC_ALL,"Russian"); // установка русского языка в консоли
    int i, j, k;                          // объявление переменных
    int a[3][4], b[4][2], c[3][2]; // объявление матриц
    printf("Введите матрицу a");
    for(i = 0; i < 3; i++)           // ввод матрицы a
    for(j = 0; j < 4; j++)
    {
      printf("\na[%d][%d]=", i, j);
      scanf("%3d", &a[i][j]);     // считывние элемента матрицы a
    }
    printf("\n Введите матрицу b");
    for(i = 0; i < 4; i++)           // ввод матрицы b
    for(j = 0; j < 2; j++)
    {
       printf("\nb[%d][%d]=",i,j);
       scanf("%3d",&b[i][j]);     // считывние элемента матрицы b
    }
       /* Умножение матриц, используя оператор while*/
    i = 0;
    while(i < 3)   // цикл выбора i-й строки матрицы a
    {
       j = 0;
       while(j < 2)  // цикл выбора j-го столбца матрицы b
       {
          c[i][j] = 0;  // обнуление элемента матрицы c для накопления
          k = 0;
          while(k < 4)   // цикл перебора элементов i-й строки матрицы a
          {                  // и j-го столбца матрицы b для их умножения
             c[i][j] += a[i][k] * b[k][j]; // накопление произведения этих элементов
             k++;
          }
          j++;          // переход к новому столбцу матрицы b
       }
       i++;        // переход к новой строке матрицы a
    }
       /* Умножение матриц, используя оператор do … while*/
    i = 0;
    do          // цикл выбора i-й строки матрицы a
    {
       j = 0;
       do       // цикл выбора j-го столбца матрицы b
       {
          c[i][j] = 0;
          k = 0;
          do   // цикл перебора элементов i-й строки матрицы a
          {      // и j-го столбца матрицы b для их умножения
             c[i][j] += a[i][k] * b[k][j];    // накопление произведения
             k++;
          }while(k < 3);
```

```
            j++;
        }while(j < 2);
        i++;
    }while(i < 3);
    system("cls"); // очистка консоли
    puts("\nРезультирующая матрица c");
    for(i = 0; i < 3; i++)          // вывод матрицы c
    {
        printf("\n");
        for(j = 0; j < 2; j++)
            printf("%3d", c[i][j]); // вывод каждого элемента матрицы c
    }
    return 0;
}
```

Пример 4. Построить спираль Улама в квадратной матрице размерностью n строк на n столбцов. В центральную клетку матрицы ставится 1, затем справа от неё ставится 2, ниже – 3 и т. д. По ходу часовой стрелки ставятся последовательные числа натурального ряда. Результирующая матрица выводится на экран, при выводе все простые числа заменяются звездочкой «*».

```
#include <stdio.h>
#include <conio.h>
#include <locale>
#include <Windows.h>
#define n 5
int main()
{
    setlocale(LC_ALL,"Russian"); // установка русского языка в консоли
    int i,i1,i2,j,j1,j2,                    // объявление переменных
        k = 0,                              // направление движения
        s = 2;                              // значение, заносимое в матрицу
        /* Объявление и зануление матрицы*/
    int mt[n][n]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    i1 = i2 = j1 = j2 = n / 2; // координаты середины матрицы
    mt[i1][j1] = 1;            // 1 ставится в центр матрицы
    while(s < 25)              // заполнение матрицы
    {
       switch(k)
       {
          case 0:      // движение вправо
             if(j2 < (n – 1) )
                j2++; // изменение правой границы
             for(j = j1 + 1; j <= j2; j++)
                if(mt[i1][j] == 0)
                {
                   mt[i1][j] = s;
                   s++;
                }
             k = 1;    // изменение направления движения вниз
             break;
          case 1:      // движение вниз
```

```
            if(i2 < (n – 1) )
               i2++;    // изменение нижней границы
            for(i = i1 + 1; i <= i2; i++)
               if(mt[i][j2] == 0)
               {
                  mt[i][j2] = s;
                  s++;
               }
            k = 2;       // изменение направления движения влево
            break;
         case 2:         // движение влево
            if(j1 > 0)
               j1--;     // изменение левой границы
            for(j = j2 – 1; j >= j1; j--)
               if(mt[i2][j] == 0)
               {
                  mt[i2][j] = s;
                  s++;
               }
            k = 3;      // изменение направления движения вверх
            break;
         case 3:         // движение вверх
            if(i1 > 0)
               i1--;     // изменение верхней границы
            for(i = i2 – 1; i >= i1; i--)
               if(mt[i][j1] == 0)
               {
                  mt[i][j1] = s;
                  s++;
               }
            k = 0;      // изменение направления движения вправо
            break;
      }
   }
   system("cls"); // очистка консоли
   puts("Спираль Улама\n");
   for(i = 0; i < n; i++) // вывод матрицы (спирали)
   {
      printf("\n");
      for(j = 0; j < n; j++)
      {
         k = 0;
         for(s = 2; s < mt[i][j] – 1; s++) // цикл проверки, является ли mt[i][j]
            if(mt[i][j] % s == 0)             // простым числом
            {
               k = 1;
               break;
            }
         if(k == 0 && mt[i][j] != 1) // вывод *, если число оказалось простым
            printf("%s", " * ");
         else
```

```
            printf("%3d", mt[i][j]);
        }
    }
    return 0;
}
```

6.5. Понятие указателя

В языках высокого уровня обычно нет необходимости знать адреса памяти, где расположены данные и оперирующие ими переменные. В языке С (С++) предусмотрен удобный механизм получения адресов данных. И соответственно, если имеется адрес, то несложно получить (или разместить) значение по этому адресу.

Указатель – это переменная, содержащая адрес некоторой переменной или группы переменных (массива). Если переменная объявлена как указатель, то она содержит адрес памяти, по которому может находиться скалярная величина любого типа. При объявлении переменной типа «указатель» необходимо определить тип объекта данных, адрес которых будет содержать переменная, и имя указателя с предшествующей звездочкой (или группой звездочек). Формат объявления указателя:

[модификатор] спецификатор_типа*идентификатор;

Память ЭВМ можно представить в виде массива последовательно пронумерованных и проадрессованных ячеек, содержимым которых можно оперировать по отдельности или связными группами. Унарная операция & выдаёт адрес объекта, таким образом, инструкция p=&k присваивает адрес первого байта переменной k в переменную p. В этом случае говорят, что p указывает на k. Различие между ними состоит в том, что p – это переменная, а &k – константа. Операция & применяется только к объектам, непосредственно расположенным в памяти (переменные, элементы массива, структуры). Её операндом не может быть ни выражение, ни константа, ни регистровая переменная.

Если мы имеем переменную p, являющуюся ссылкой (адресом), то для доступа к значению по этому адресу можно воспользоваться операцией *косвенной адресации* «*****» (разадресация). Выражение kk=*p позволяет переменной kk передать значение, на которое указывает p (расположенное по адресу p).

Как уже отмечалось, для того чтобы иметь возможность оперировать с некоторым объектом, он должен быть описан как объект определённого типа. Аналогично и с указателями на эти объекты.

6.6. Описание указателей

Из рассмотренного ранее материала следует, как описать переменную любого из известных типов. Для описания переменной типа «указатель» необходимо указать, на переменную какого типа возможна ссылка через объявляемый указатель. Это связано с тем, что для выполнения некоторых адресных операций необходимо знать размеры памяти, отводимой под объекты, на которые указывает указатель. Ниже приводятся примеры описания указателей:

```
int *p;              // p – указатель на элементы типа int
float *k,*k1;        // k и k1 – два указателя на элементы типа float
char *s;             // s – указатель на элементы символьного типа
```

Адрес – это целое число без знака, поэтому при выводе на печать значения указателя (p, k, k1, s) будем использовать формат %u. Исключением является бестиповый указатель, т. е. указатель вида

```
void *pp;
```

Указатель типа **void*** содержит адрес, указывающий на место в оперативной памяти, но *не содержит* информации о *типе* объекта, расположенном по этому адресу, и, следовательно, может быть использован для адресации данных любого типа. Однако для выполнения арифметических и логических операций над этими указателями или объектами, на которые они указывают, необходимо явно определить их тип. Это может быть выполнено с помощью операции приведения типов. Иначе компилятору неизвестно, какой длины поле памяти должно использоваться в операции.

Для правильной работы указатель должен ссылаться на объекты только соответствующего ему типа. При этом компилятором не контролируется нарушение этого требования. Рассмотрим эту ситуацию на примере перезаписи значения переменной a в переменную b.

```
#include <stdio.h>
int main()
{
   int *p;
   float a=10.25,b;
    p=&a;      // ошибка: cannot convert from 'float *' to 'int *'
   b=*p;       // в b записывается значение, содержащееся по адресу p
   printf("a= %f     b= %f",a,b);
   return 0;
}
```

Значение переменной a не будет перенесено в переменную b, т. к. p является целочисленным указателем (т. е. имеет тип int *) и адрес переменной a (типа float *) не может быть приведен по умолчанию к типу указателя p.

6.7. Операции с указателями

Указатели могут встречаться в выражениях. То есть если p – указатель на объект некоторого типа, то он может использоваться в выражении наряду с другими указателями, а также переменными и константами, не являющимися указателями, например:

```
*p=–2;
*p/=i–1;
(*p)--;
```

В первом выражении значение «–2» заносится в ячейку памяти по адресу p, во втором выражении это значение уменьшается в i–1 раз, в третьем – уменьшается на единицу. В третьем выражении использованы скобки, т. к. операции с одинаковым приоритетом выполняются справа налево, таким образом, *p-- уменьшит адрес, содержащийся в указателе p.

Указатели могут использоваться в качестве операндов в арифметических выражениях. Так, например, если p – указатель, то p++ является адресом следующего элемента. Следовательно, конструкция p+n (p – указатель, n – целое число) задаёт адрес n-го элемента, на который указывает указатель p+n.

Между адресами могут быть выполнены операции сравнения. Также любой адрес может быть проверен на равенство (==) или неравенство (!=) со специальным значением **NULL. NULL** определяет указатель, который ничего не адресует.

К указателю типа **void*** применяются следующие операции:

= , == , != , > , < , <= , >=

6.8. Связь между указателями и массивами

В языке С (С++) между индексированием и адресацией существует тесная взаимосвязь. Доступ к элементам массива с помощью индексирования может быть заменен доступом к ним с использованием адресов. При этом второй вид доступа выполняется быстрее. Например:

```
#include <stdio.h>
int main()
{
   int *p,i,ms[]={1,2,3,4,5};
   p= ms;     //  можно это заменить на  p=&ms[0]
   scanf("%d",&i);
   printf("ms[%d]= %d\n",i,ms[i]);    // эта и следующая инструкции
   printf("p+i= %d\n",*(p+i));        // работают одинаково
   return 0;
}
```

6.9. Массивы указателей

Кроме обычных массивов, в языке С (С++) используются массивы указателей. Они представляют собой массивы, элементами которых являются указатели (например, на элементы другого массива). Описание массива указателей имеет вид

```
float *mas[6];
```

Это соответствует массиву из шести элементов, являющихся адресами объектов типа float.

В массиве указателей последовательность элементов задаётся последовательностью размещения указателей в массиве. Тогда изменение порядка следования (включение, исключение, упорядочение, перестановка), которое в обычном массиве заключается в перемещении самих элементов, в массиве указателей должно сопровождаться соответствующими операциями над указателями. Очевидные преимущества возникают, когда сами указуемые элементы являются достаточно большими (массивы, строки, структуры и т. д.).

6.10. Многоуровневые указатели

Рассмотрим определение переменной

```
double **p;
```

В соответствии принципом контекстного определения pp нужно интерпретировать как переменную, при косвенном обращении к которой (разодреса-

ции) получается указатель на переменную типа **double**, т. е. как указатель на указатель или адрес указателя. Но поскольку любой указатель в С (С++) может ссылаться как на отдельную переменную, так и на область памяти (массив), то в применении к двойному указателю получаются четыре варианта структур данных, а именно:

− указатель на одиночный указатель на переменную типа **double**;

− указатель на одиночный указатель на массив переменных типа **double**;

− указатель на массив, содержащий указатели на одиночные переменные типа **double**;

− указатель на массив, содержащий указатели на массивы переменных типа **double**.

В четвертом случае структура выделенной памяти для хранения данных имеет вид, приведенный на рис. 2.

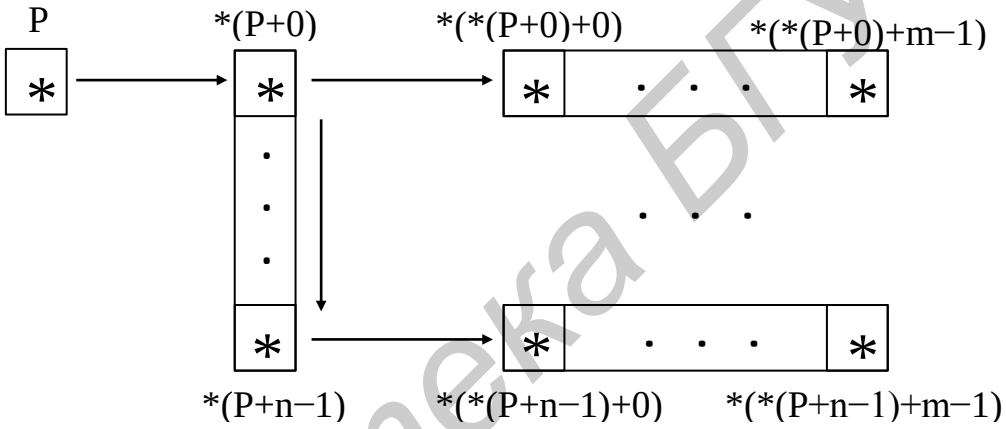


Рис. 2. Структура выделения памяти для объявления int **P

6.11. Примеры программ

Пример 1. Разработать программу вывода элементов массива в упорядоченном виде (без изменения их местоположения в нём). Для этого использовать массив указателей на элементы числового массива. В процессе упорядочивания элементов их адреса заносятся в массив указателей. После сортировки в массиве указателей будут содержаться упорядоченные адреса элементов числового массива.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian"); // установка русского языка в консоли
   int i, j, k;
   int array1[5], *array2[5], n;
   int flags[5] = {0, 0, 0, 0, 0};
```

```
    for(i = 0; i < 5; i++) // ввод элементов массива чисел
    {
       printf("\nВведите %d-й элемент массива: ", i+1);
       scanf("%d", &array1[i]);
    }
    for(i = 0; i < 5; i++) // блок сортировки элементов
    {
       n = i;            // выбор исходного элемента для анализа
       k = 32767;     // максимальное число для анализа
       for(j = 0; j < 5; j++)
       {
          if ((array1[j] < k) && (flags[j] == 0)) // найден номер меньшего
          {                                       // элемента и его ещё не выбирали
             k = array1[j];                        // запоминаем значение и номер
             n = j;
          }
       }
       array2[i] = &array1[n];   // записываем адрес
       flags[n] = 1;                // запрет выбора к анализу
    }
    system("cls");
    printf("Исходный массив:\n");
    for(i = 0; i < 5; i++) printf("%d ", array1[i]);   // вывод исходного массива
    printf("\n\nУпорядоченный массив:\n");
    for(i = 0; i < 5; i++) printf("%d ", *array2[i]);  // вывод упорядоченного массива
    printf("\n\n");
    system("pause");
    return 0;
}
```

Пример 2. Используя указатели, выделить память и ввести два массива целых чисел.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian");          // установка русского языка в консоли
   int *m1, *m2, n1, n2, i;
   printf("Введите размер первого массива: ");
   scanf("%d", &n1);
   printf("Введите размер второго массива: ");
   scanf("%d", &n2);
   if(!(m1 = (int *) calloc(n1, sizeof(int))))    // выделение памяти под массив и
   {                                    // одновременная проверка, выделилась ли она
      return 0;
   }
   if(!(m2 = (int *) calloc(n2, sizeof(int)))) // выделение памяти под массив и
   {                                   // одновременная проверка, выделилась ли она
      free(m1);      // если память не выделилась, то очистить первый массив
      return 0;
```

```
    }
    printf("\nВведите первый массив: "); // ввод первого массива
    for(i = 0; i < n1; i++)
    {
       scanf("%d", m1+i);
    }
    printf("Введите второй массив: ");     // ввод второго массива
    for(i = 0; i < n2; i++)
    {
       scanf("%d", m2+i);
    }
    system("cls");
    printf("Первый массив: ");
    for(i = 0; i < n1; i++)
    {
       printf("%d ", *(m1+i));                    // вывод первого массива
    }
    printf("\nВторой массив: ");
    for(i = 0; i < n2; i++)
    {
       printf("%d ", *(m2+i));                    // вывод второго массива
    }
    free(m1);                                    // освобождение памяти
    free(m2);
    printf("\n\n");
    system("pause");
    return 0;
}
```

Пример 3. Используя указатели, выделить память и ввести два упорядоченных по возрастанию массива целых чисел. Сформировать третий массив, также упорядоченный по возрастанию.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian"); // установка русского языка в консоли
   int *m1, *m2, *m3, n1, n2, i1, i2, i3;
   printf("Введите размер первого массива: ");
   scanf("%d", &n1);
   printf("Введите размер второго массива: ");
   scanf("%d", &n2);
   if(!(m1 = (int *) calloc(n1, sizeof(int)))) // выделение памяти под массив и
   {                                  // одновременная проверка, выделилась ли она
      return 0;
   }
   if(!(m2 = (int *) calloc(n2, sizeof(int)))) // выделение памяти под массив и
   {                                  // одновременная проверка, выделилась ли она
      free(m1);    // если память не выделилась, то очистить первый массив
      return 0;
```

```
}
if(!(m3 = (int *) calloc(n1+n2 ,sizeof(int))))
{
   free(m1);    // если память не выделилась, то очистить первый массив
   free(m2);    // если память не выделилась, то очистить второй массив
   return 0;
}
printf("\nВведите первый массив: ");
for(i1 = 0; i1 < n1; i1++)
{
   scanf("%d", m1+i1);    // ввод первого массива
}
printf("Введите второй массив: ");
for(i2 = 0; i2 < n2; i2++)
{
   scanf("%d", m2+i2);    // ввод второго массива
}
system("cls");
printf("Первый массив: ");
for(i1 = 0; i1 < n1; i1++)
{
   printf("%d ", *(m1+i1)); // вывод первого массива
}
printf("\nВторой массив: ");
for(i2 = 0; i2 < n2; i2++)
{
   printf("%d ", *(m2+i2)); // вывод второго массива
}
i1 = i2 = i3 = 0;
do
{
   /* Пока элементы первого массивы меньше либо равны элементам
      второго массива и не конец первого массива */
   while((*(m1+i1) <= *(m2+i2)) && (i1 < n1))
   {
      *(m3+i3++) = *(m1+i1++); // заносим из первого массива в третий
   }
    /* Пока элементы второго массивы меньше элементов первого
      массива и не конец второго массива */
   while((*(m1+i1) > *(m2+i2)) && (i2 < n2))
   {
      *(m3+i3++) = *(m2+i2++); // заносим из второго массива в третий
   }
} while((i1 < n1) && (i2 < n2));   // пока не закончился один из массивов
while(i1 < n1)
{
   *(m3+i3++) = *(m1+i1++);     // записываем оставшиеся элементы первого
}                                // массива
while(i2 < n2)
{
   *(m3+i3++) = *(m2+i2++);     // записываем оставшиеся элементы второго
```

```
    }                                   // массива
    printf("\nТретий массив: ");
    for(i3 = 0; i3 < n1+n2; i3++)
    {
       printf("%d ", *(m3+i3));         // вывод третьего массива
    }
    printf("\n\n");
    system("pause");
    return 0;
}
```

Пример 4. Используя указатели, выполнить сортировку элементов квадратной матрицы для чисел типа **float**, расположенных ниже главной диагонали.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
int main()
{
    setlocale(LC_ALL,"Russian");   // установка русского языка в консоли
    float *matrix, min;
    int i, i1, j, j1, n;
    printf("Введите размерность матрицы: ");
    scanf("%d", &n);
    if(!(matrix = (float *) malloc(n*n*sizeof(float)))) // выделение памяти с проверкой
    {
       puts("Нет свободной памяти!");
       return 0;
    }
    printf("Введите элементы в матрицу\n");      // ввод элементов в матрицу
    for(i = 0; i < n; i++)
    {
       for(j = 0; j < n; j++)
       {
          printf("Матрица[%d][%d]: ", i+1, j+1);
          scanf("%f", matrix+i*n+j);
       }
    }
    system("CLS");
    printf("\nВведенная матрица: ");  // вывод элементов матрицы
    for(i = 0; i < n; i++)
    {
       printf("\n");
       for(j = 0; j < n; j++) printf("%5.1f", *(matrix+i*n+j));
    }
    for(i = 1; i < n; i++)
    {
       for(j = 0; j < i; j++)                  // выбор элемента для записи в него
       {
          min = *(matrix+i*n+j);            // записываем минимальное значение
          for(i1 = n−1; i1 >= i; i1--)       // просмотр массива с конца
          {
             for(j1 = i1−1; j1 >= 0; j1--) // с последнего элемента строки
```

```
                {
                    /* Если выбранный адрес больше исходного адреса и новое
                    значение меньше исходного*/
                   if(matrix+i*n+j < matrix+i1*n+j1 &&  min > *(matrix+i1*n+j1))
                   {
                      min = *(matrix+i1*n+j1);     // замена значений
                      *(matrix+i1*n+j1) = *(matrix+i*n+j);
                      *(matrix+i*n+j) = min;
                   }
                }
             }
          }
       }
       printf("\n\nОтсортированная матрица: "); // вывод отсортированной матрицы
       for(i = 0; i < n ; i++)
       {
          printf("\n");
          for(j = 0; j < n; j++)
          {
             printf("%5.1f", *(matrix+i*n+j));
          }
       }
       free(matrix); // очистка памяти
       printf("\n\n");
       system("pause");
       return 0;
    }
```

Пример 5. Используя указатели, выполнить сортировку элементов квадратной матрицы, расположенных выше побочной диагонали.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
int main()
{
   setlocale(LC_ALL,"Russian"); // установка русского языка в консоли
   float *matrix;
   int i, i1, j, j1, n;
   float min;
   printf("Введите размерность массива: ");
   scanf("%d", &n);
   if((matrix = (float *) malloc(n*n*sizeof(float))) == NULL)
   { // выделение памяти с проверкой
      puts("Нет свободной памяти!");
      return 0;
   }
   printf("Введите элементы матрицы: \n"); // ввод матрицы
   for(i = 0; i < n; i++)
   {
      for(j = 0; j <n ; j++)
      {
         printf("Матрица[%d][%d]: ", i+1, j+1);
```

```
            scanf("%f", matrix+i*n+j);
        }
    }
    system("CLS");                          // очистка экрана
    printf("\nВведённая матрица: ");  // вывод введённой матрицы
    for(i = 0; i < n; i++)
    {
        printf("\n");
        for(j = 0; j < n; j++)
        {
            printf("%5.1f", *(matrix+i*n+j));
        }
    }
    for(i = 0; i < n−2; i++)
    {
        for(j = 0; j < n−i−1; j++)       // выбор элемента для записи в него
        {
            min = *(matrix+i*n+j);      // запись минимального значения
            for(i1 = n−2; i1 >= i; i1--) // просмотр массива с конца
            {
                for(j1 = n−i1−2; j1 >= 0; j1--)  // с последнего элемента строки
                {
                    /* Если выбранный адрес большее исходного адреса и
                       новое значение меньше исходного */
                    if((matrix+i*n+j < matrix+i1*n+j1) && (min > *(matrix+i1*n+j1)))
                    {
                        min = *(matrix+i1*n+j1);  // замена значений
                        *(matrix+i1*n+j1) = *(matrix+i*n+j);
                        *(matrix+i*n+j) = min;
                    }
                }
            }
        }
    }
    printf("\n\nОтсортированная матрица: "); // вывод отсортированной матрицы
    for(i = 0; i < n; i++)
    {
        printf("\n");
        for(j = 0; j < n; j++)
        {
            printf("%5.1f", *(matrix+i*n+j));
        }
    }
    free(matrix); // очистка памяти
    printf("\n\n");
    system("pause");
    return 0;
}
```

7. Символьные строки

7.1. Декларирование символьных строк

Под строкой будем понимать совокупность непрерывно расположенных в памяти символов (ASCII-кодов). В языке С (С++) отсутствует тип данных «символьная строка». Механизм работы со строками обычно реализуется через указатели типа **char *.**

Это указатель на область памяти, каждый элемент которой имеет размер один байт. Как и обычный указатель, указатель на тип **char** может быть инициализирован при его описании. Для этого используется строковая константа, при этом адрес её первого символа будет присвоен указателю, например:

```
char *str="строка";
```

При этом выделяется память (семь байт) для строки и указатель инициализируется адресом первого символа. Признаком окончания символьной строки является нуль-символ «'\0'». При его отсутствии информация в массиве будет являться набором символов, но не символьной строкой. В выражениях, где применяется указатель, компилятор подставляет адрес строковой константы. Строковая константа – это любое выражение, заключённое в двойные кавычки. При встрече строковой константы её символы и символ «'\0'» записываются в последовательные ячейки памяти. Строковые константы размещаются в статической памяти.

Также можно создать массив указателей типа **char** (массив строк):

```
char *str[5];   // массив из 5-ти указателей
```

Инициализацию массива строк (массива указателей **char***) можно выполнить следующим образом:

```
char str1[][4]={"строка1","строка2"};
char *str2[2];
str2[0]="строка1";
str2[1]="строка2";
char *str3[2]={ "строка1","строка2"};
```

Строковая константа, как и любая другая константа в С (С++), может быть также определена с помощью директивы #**define**, например:

```
#define st1 "Минск"
#define st2 "каф. ЭВМ"
```

В чём состоит отличие объявления символьной строки через указатель на тип **char** и с использованием массива типа **char**. Эти два подхода имеют общие стороны: имя массива является адресом его начала, указатель также содержит адрес начала некоторой области памяти. В то же время имя массива (например mas) является константным значением. То есть мы не можем изменить значение mas, т. к. это по существу означало бы изменение адреса массива в памяти. Следовательно, для доступа к очередному элементу массива можно использовать выражение вида mas+1, но не ++mas.

При использовании указателей происходит выделение в динамической памяти области для размещения в ней строки, а переменная str, являющаяся указателем на эту область (строку), размещается в статической памяти. При

этом значение этой переменной может изменяться, таким образом, ++str будет указывать на следующий символ строки.

7.2. Ввод/вывод строк

Ранее были подробно рассмотрены функции **scanf**() и **printf**(). Они могут быть использованы и для организации ввода/вывода символьных строк. Наряду с ними в С (С++) имеется еще пара функций **gets**() и **puts**(), предназначенных непосредственно для ввода/вывода символьных строк.

Общий формат записи функции **gets()** имеет вид

char *gets(char *buffer);

Функция **gets**() считывает символьную строку из стандартного входного потока и помещает её по адресу, заданному указателем buffer. Ввод строки заканчивается при встрече символа «'\n'» (нажатии клавиши Enter). Этот символ заменяется символом «'\0'» и помещается в конец введённой строки. Функция **gets()** возвращает указатель на считанную строку.

Общий формат записи функции **puts()** имеет вид

int puts(const char *string);

Функция **puts**() записывает символьную строку в стандартный поток данных. Нуль-символ «'\0'» конца строки заменяется символом перехода к новой строке «'\n'». Функция возвращает неотрицательное значение, если оканчивается успешно, иначе возвращает «EOF».

Ниже рассмотрен пример с несколькими вариантами использования функции **gets**().

```
#include<stdio.h>
int main()
{
   char mas[5];  // символьный массив
   char *str;       // указатель на символьную строку
        // вариант 1 ввода строки в массив mas
   gets(mas);     // ввод строки в mas
   str=mas;       // str указывает на строку в массиве mas
   printf("\n mas=%s *str=%s",mas,str);
        // вариант 2 ввода строки str
                  // вначале необходимо выделить память для ввода строки
   str=(char*)malloc(80);  // или str=(char*)malloc(80*sizeof(char));
   gets(str);       // ввод строки непосредственно в выделенную память
   puts(str);       // вывод на экран введённой строки
   free (str);       // освобождение памяти занимаемой строкой
   return 0;
}
```

В первом варианте строка вводилась в массив mas, и выполнялось присвоение str=mas, сообщая указателю, где находится введённая строка. Во втором варианте ввода сроки **gets**() необходимо вначале выделить память для ввода в неё символьной строки. В функцию **gets**() передаётся указатель на выде-

ленную для строки память. При этом указатель str уже будет указывать на зарезервированный под строку участок памяти.

Кроме того, функция **gets**() включает разряд проверки ошибки. То есть если произошла ошибка ввода или **gets**() встретила символ **EOF**, она возвращает нулевой адрес (**NULL**):

```
char str[10];
while(gets(str)!=NULL) puts(str); // ввод и вывод пока не введены Ctrl+z
```

Сравненим с тем, как это выполняется при использовании функции **getchar**():

```
char c;
while((c=getchar())!=EOF);
```

При работе с функцией **gets**() необходимо помнить, что она не выполняет контроль на переполнение символьной строки. В некоторых случаях это может привести к ошибке (порча памяти). Наряду с этим рекомендуется перед использованием функции **gets**() выполнить очистку буфера клавиатуры, т. к. некоторые функции ввода считывают данные из буфера до символа «'\13'», оставляя его в буфере, а функция **gets**(), считывая его, прекращает ввод информации, при этом строка может оказаться пустой:

```
fflush(stdin);
```

Инициализацию строки можно выполнить и с помощью функции **scanf**(), используя указатель:

```
char *st ;
st=(char *) malloc(10);
scanf("%s",st);
```

Ввод символьной строки завершается при встрече либо символа «' '» (пробел), либо символа «'\n'».

Для правильной работы программы следует соблюдать следующие требования:

− для переноса содержимого строки с одного места памяти в другое необходимо предварительно выделить новое место в памяти, затем неиспользуемый участок памяти требуется освободить;

− копирование строк с использованием непроинициализированного указателя является грубой ошибкой программирования;

− при работе со строками требуется обращать внимание на предупреждение типа «Подозрительное преобразование указателя» (Suspicious pointer conversion) или «Использование указателя до инициализации» (Possible use of...before deffinition);

− при выделении места для строки-назначения следует предусмотреть место для нуль-символа (признака конца строки).

При написании программ обычно необходимо производить разные манипуляции со строками: перемещать их с одного места памяти в другое, выполнять поиск в строке некоторой последовательности символов, объединять строки, вставлять одну строку в другую, удалять из строки часть информации и т. д. В языке С (С++) имеется набор функций, обеспечивающих эти и другие действия.

7.3. Функции работы со строками

Рассмотрим объявления некоторых наиболее часто встречающихся функций работы со строками (требуется отметить, что прототипы нижеприведённых функций располагаются в заголовочном файле <**string.h**>):

1) **size_t strlen(const char *s)** – возвращает длину в байтах строки s, не включая «'\0'»;

2) **char *strcat(char *dest, const char *src)** – присоединяет строку src в конец строки dest;

3) **char *strcpy(char *dest, const char *src)** – копирует строку src в место памяти, на котоое указывает dest;

4) **char * strstr(char *s1, const char *s2)** – отыскивает первое вхождение строки s2 в строку s1;

5) **int strcmp(const char *s1, const char *s2)** – сравнивает две строки в лексикографическом порядке с учётом различия прописных и строчных букв. Возвращает значение меньше нуля, если s1 располагается в упорядоченном по алфавиту порядке раньше, чем s2, и больше, если наоборот. Функция возвращает нуль, если строки идентичны.

7.4. Примеры программ

Пример 1. Удалить из строки символов подстроку, заключенную в [ ].

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<iostream>
int main()
{
   char *str;
   int i, j, k, n;
   setlocale(LC_ALL, "Russian");
   str = (char *)malloc(100); // выделение памяти под символьную строку
   if( ! str)          // если память не выделена, то завершение программы
   {
      return 0;
   }
   printf("введите строку символов  ");
   fflush(stdin);   // чистка входного потока
   gets(str);       // ввод строки
   printf("Исходная строка символов :  %s\n",str); // вывод исходной строки
   j = – 1;                      // позиция открывающей скобки  '[' в строке
   for(i = 0;*(str + i);i ++ )   // цикл прохода по строке до её конца
   {
      if (*(str + i) == '[')      // скобка найдена
      {
         j = i + 1;              // i – позиция символа,стоящего после скобки
         while(*(str + j) != ']' && *(str + j) != '[' && *(str + j) != '\0') // поиск скобок
         {
            j ++;
         }
```

```
            if (*(str + j) == '[') // позиция вложенной скобки [
            {
               i = j – 1;          // вложенная скобка становится основной
            }
            else
            {
               if(*(str + j++) == ']')           // позиция закрывающей скобки
               {
                  n=i;
                  while((*(str+n)=*(str+j))) // удаление найденной подстроки
                  {
                     n ++;
                     j ++;
                  }
               }
            }
         }
      }
      printf("преобразованная строка: %s",str);
      getch(); // ожидание нажатия клавиши
      return 0;
   }
```

Пример 2. Ввести символьную строку и выполнить в ней удаление предпоследнего слова. Слова в строке разделены одним пробелом.

```
   #include<stdio.h>
   #include<stdlib.h>
   #include<conio.h>
   #include<iostream>
   int main()
   {
      int buf, i;
      char *str;
      str = (char*)malloc(100); // выделение памяти под символьную строку
      setlocale(LC_ALL, "Russian");
      if( ! str)          // проверка на выделение памяти
      {
         return 0;
      }
      fflush(stdin);     // чистка входного потока
      printf("введите строку: ");
      gets(str);          // ввод строки
      printf("исходная строка: %s\n", str); // вывод исходной строки
      i=0;
      while(*(str + i) != '\0')         // переход в конец строки, i – позиция ‘\0’
      {
         i++;
      }
      // поиск пробела, i – позиция пробела, стоящего перед последним словом
      while(*(str + i) != ' ')
      {
         i--;
```

```
            if (i == 0)             // если в строке лишь 1 слово
            {
                printf("преобразованная строка: %s",str);
                return 0;
            }
        }
        buf = i;     // buf – позиция пробела
        i -- ;       // i – позиция последнего символа предпоследнего слова
        while(*(str + i)!= ' ') // поиск позиции пробела перед предпоследним словом
        {
            i--;
            if(i == 0) // если в строке лишь 2 слова
            {
                break;
            }
        }
        if(i != 0)
        {           // сдвиг на первый символ предпоследнего
            i++;    // слова, если слов больше чем 2
        }
        buf ++;     // первый символ последнего слова
        while((*(str + i) = *(str + buf)) != 0)    // удаление предпоследнего слова
        {
            i++;
            buf++;
        }
        printf("преобразованная строка: %s",str); // вывод преобразованной строки
        getch();
        return 0;
    }
```

Пример 3. Ввести символьную строку и выполнить в ней замену местами последнего и предпоследнего слов.

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <conio.h>
    #include <iostream>
    int main()
    {
        int j, i, n;
        char *str1, *str2;
        setlocale(LC_ALL, "Russian");
        str1 = (char*)malloc(100); // выделение памяти под символьную строку 1
        str2 = (char*)malloc(100); // выделение памяти под символьную строку 2
        if( ! str1 || ! str2)             // проверка на выделение памяти
        {
            return 0;
        }
        printf("введите строку:  ");
        fflush(stdin); // чистка входного потока
        gets(str1);    // ввод строки 1
        printf("исходная строка: %s\n", str1); // вывод исходной строки
```

```
i = 0;
while(*(str1 + i) != '\0')  // переход в конец строки, i – позиция ‘\0’
{
   i++;
}
while(*(str1+i)!=' ')        // поиск пробела, i – позиция пробела
{
   i--;
   if (i == 0)    // если в строке одно слово, то перестановка бессмысленна
   {
      printf("преобразованная строка: %s", str1);
      getch();
      return 0;
   }
}
i--;                          // i – позиция последней буквы предпоследнего слова
while(*(str1 + i) != ' ')   // поиск пробела, i – позиция
{                             // пробела перед предпоследним словом
   i--;
   if (i == 0)    // в строке два слова
   {
      break;
   }
}
if(i != 0)
{
   i++;    // i – позиция первой буквы предпоследнего слова,
}          // если слов больше, чем два
j = i;       // запоминаем позицию первого символа этого слова
n = 0;
while(*(str1+i) != ' ') // в строку str2 записываем предпоследнее слово
{
   *(str2 + n) = *(str1 + i);
   n++;
   i++;
}
*(str2 + n) = '\0';  // конец строки
i++;                 // i – позиция первой буквы последнего слова
while(*(str1 + i))    // запись последнего слова на предпоследнее
{
   *(str1 + j)=*(str1 + i);
   i++;
   j++;
}
*(str1 + j) = ' ';
j++;        // вставка пробела после слова
n = 0;      // дозапись слова из строки str2
while(*(str1 + j) = *(str2 + n))
{
   j++;
   n++;
```

```
    }
    printf("преобразованная строка: %s", str1); // вывод преобразованной строки
    getch();
    free(str1);    // освобождение памяти строки 1
    free(str2);    // освобождение памяти строки 2
    return 0;
}
```

Пример 4. Рассмотрим ещё один пример программы замены двух соседних слов в строке без использования дополнительных строк. Пусть, например, необходимо выполнить замену местами первого и второго слов.

```
#include<stdlib.h>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
#include<iostream>
int main()
{
    char c, *str;
    int n_sl1, k_sl1, n_sl2, k_sl2, n, i, j;
    setlocale(LC_ALL,"Rus");
    printf("Введите длину строки символов : ");
    scanf("%d",&n);
    printf("\n Введите строку : ");
    if ((str = (char *)malloc(n)) == NULL)
    {
        puts("нет свободной памяти");
        return 0;
    }
    fflush(stdin);        // чистка входного буффера
    gets(str);
    printf("\n Исходная строка : %s",str);
    i = 0;
    while(*(str + i) == ' ') i++;  // пропуск пробелов в начале строки
    n_sl1 = i;                     // начало первого слова
    while(*(str + i) != ' ' && *(str + i) != '\0') i++;
    k_sl1 = i – 1;                  // конец первого слова
    while(*(str + i) == ' ') i++;  // пропуск пробелов перед вторым словом
    n_sl2 = i;                     // начало второго слова
    while(*(str + i) != ' ' && *(str + i) != '\0' ) i++;
    k_sl2 = i – 1;                 // конец второго слова
    if(k_sl1 – n_sl1 >= k_sl2 – n_sl2) // если первое слово длиннее второго
    {
        for(; k_sl2 – n_sl2 >= 0;)  // замена всех символов меньшего и части
        {                           // большего слова местами
            c = *(str + n_sl1);
            *(str + n_sl1) = *(str + n_sl2);
            *(str + n_sl2) = c;
            n_sl1++; n_sl2++;
        }
        if(k_sl1 != n_sl1)              // если слова были неодинаковой длины
```

```
            for(; k_sl1 – n_sl1>=0; ) // сдвиг остатка большего слова к
            {                         // переписанной выше первой части
               j = k_sl1--;
               c = *(str + j);
               while(j <= k_sl2) *(str + j++) = *(str + j + 1);
               *(str + k_sl2--) = c;
            }
         }
         else       // если второе слово длиннее первого
         {
            for(; k_sl1 – n_sl1 >= 0; )   // замена всех символов меньшего и части
            {                             // большего слова местами
               c = *(str + n_sl1);
               *(str + n_sl1) = *(str + n_sl2);
               *(str + n_sl2) = c;
               n_sl1++; n_sl2++;
            }
            for(; k_sl2 – n_sl2 >= 0;)    // сдвиг остатка большего слова к
            {                             // переписанной выше первой части
               j = n_sl2++;
               c = *(str + j);
               while(j >= n_sl1) *(str + j--) = *(str + j – 1);
               *(str + n_sl1++) = c;
            }
         }
         printf("\n\n Преобразованная строка : %s \n",str);
         free(str);
         return 0;
      }
```

Пример 5. Ввести символьную строку и найти в ней слово максимальной длины.

```
      #include <stdio.h>
      #include <stdlib.h>
      #include <conio.h>
      #include <iostream>
      int main()
      {
         char *str;
         int i, dl, maxdl, n, nmax;
         setlocale(LC_ALL, "Russian");
         str = (char *)malloc(100); // выделение памяти под символьную строку
         if( ! str) // проверка на выделение памяти
         {
            puts("ошибка выделения памяти");
            return 0;
         }
         fflush(stdin); // чистка входного потока
         printf("введите строку: ");
         gets(str);      // ввод строки
         i = 0;
         dl = 0;         // длина слова
```

```
    maxdl = 0;                  // максимальная длина
    while(*(str + i) != '\0')   // цикл, идущий до конца строки
    {
       while(*(str + i) == ' ') // пропуск пробелов
       {
          i++;
       }
       n = i;             // n – позиция начала слова
       while(*(str + i) != ' ' && *(str + i) != '\0')
       {
          i++;
          dl++;           // подсчёт длины слова
       }
       if(maxdl < dl)
       {
          maxdl = dl; // запоминание максимальной длины
          nmax = n;  // запоминание координаты начала у самого длинного слова
       }
       dl = 0;           // обнуление значения длины
    }
    puts("слово max длины: ");
    while(*(str + nmax) != ' ' && *(str + nmax) != '\0')
    {
       printf("%c",*(str + nmax)); // вывод самого длинного слова
       nmax++;
    }
    getch();
    return 0;
}
```

8. Функции

8.1. Общие сведения о функциях

Принцип программирования на языке С (С++), как и в ряде других языков программирования, основан на понятии *функции.* В рассмотренных ранее программах мы уже пользовались функциями. К их числу относятся функции ввода/вывода, работы с символьными строками и ряд других. Эти функции являются системными, в то же время мы разработали несколько функций, использующих вышеназванные и озаглавленных одним общим именем **main**(). Несколько подробнее остановимся на вопросе, как создавать свои собственные функции и делать их доступными для функции **main**(), а также друг для друга.

Что такое функция? Функция − самостоятельная единица программы, разработанная для выполнения некоторого функционально законченного действия. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные. Функции разрешается располагать в любом порядке, исходную программу можно подразделять на произвольное число файлов. Следует различать понятия «*прототип функции*» и «*определение функции*».

8.2. Прототип функции

Прототип функции – это её имя с указанием типа возвращаемого результата и перечислением в круглых скобках (через запятую) типов передаваемых в неё параметров. Прототип функции завершается точкой с запятой. Например:

```
double step(float x, int n);
char * func(char *, int);
```

Прототип является объявлением функции. Прототип необходим компилятору для того, чтобы проконтролировать корректность вызова функции и в необходимых случаях выполнить преобразование аргументов к типу принимаемых параметров, а также сгенерировать корректный возвращаемый результат.

Имена параметров в прототипе можно опустить, однако рекомендуется их указывать. Компилятор будет использовать имена параметров при выдаче диагностических сообщений о несоответствии типов аргументов и параметров. В том случае, когда в функцию не передаются аргументы, в прототипе в круглых скобках вместо списка параметров записывается слово **void** либо ничего не указывается.

8.3. Определение функции

Определение (или описание) функции – это её полная реализация. Определение функции может располагаться в любом месте программы, но определение одной функции не может вмещать в себя определение другой функции, но может содержать её прототип, если она вызывает эту функцию.

Если определение некоторой функции располагается ниже точки её вызова или в другом файле, то для такой функции выше точки её вызова обязательно должен быть помещен прототип. В соответствии с синтаксисом языка С (С++) определение функции имеет следующую форму:

[спецификатор_класса_памяти] [спецификатор_типа] имя_функции ([список_формальных_параметров])
{тело_функции}

Необязательный спецификатор_класса_памяти задает класс памяти функции, который может быть **static** или **extern**. Спецификатор_типа функции задает тип возвращаемого результата. Результат может быть любого стандартного типа, а также типа, определённого пользователем (структура, объединение, класс и др.). Функция возвращает явно только одно значение указанных типов. Допускается реализация функций, не имеющих параметров и/или не возвращающих никаких значений.

Передача (возврат) значения из вызванной функции в вызывающую осуществляется посредством использования оператора возврата **return**, имеющего следующий вид:

return выражение; или **return (выражение);**

Выражение может являться как некоторой константой, так и вычисляемым выражением:

return −12; или **return (−12);**
return i+5/j; или **return (i+5/j);**

Оператор **return** в теле функции может встречаться более одного раза. При его встрече происходит прекращение выполнения функции и осуществляется передача управления очередной инструкции (команде) в вызывающей функции. Если в операторе **return** отсутствует выражение, а вызываемая функция должна вернуть некоторое значение, то компилятор генерирует ошибку. При отсутствии оператора **return** функция будет выполнена до конца. В этом случае функция также явно не возвращает результат, перед именем такой функции в её прототипе и в её определении должно быть записано слово **void.**

Если тип выражения в операторе **return** не соответствует типу значения, возвращаемому функцией, то будут выполнены автоматически соответствующие преобразования типа выражения к типу возвращаемого функцией значения. В случае невозможности таких преобразований выводится сообщение об ошибке.

Функция *не может* в качестве результата возвращать *массив* любого типа, но может передать *указатель* на такой массив. Это связано с тем, что в С (С++) нет операции присваивания массивов.

Ниже приводится пример программы, в которой из функции **main**() производится вызов поочередно двух функций (fun1, fun2):

```
#include <stdio.h>
int fun1(int , int );       //  прототип функции fun1
int fun2(int *, int *);     //  прототип функции fun2
int main()
{
   int a,b,c;
   scanf("%d%d",&a,&b);
   c=fun1(a,b);         //  вызов функции fun1
```

```
    printf("\n 1: результат %d – %d = %d",a,b,c);
    c=fun2(&a,&b);    //  вызов функции fun2
    printf("\n 2: результат %d – %d = %d",a,b,c);
    return 0;
}
int fun1(int a,int b)
{
    a--;                    // изменённое локальное значение переменной a
    return a–b;          // в main передаётся значение разности a–b
}
int fun2(int *ptr_a,int *ptr_b)
{
    (*ptr_a)++;               // изменяется значение по адресу  a
    return *ptr_a–*ptr_b;
}
```

Функция fun1 принимает в качестве аргументов значения переменных a и b, копируя их в локальные (свои) переменные a и b. Изменение переменной a в функции не приводит к изменению значения переменной a, описанной в **main**().

Функция fun2 принимает адреса переменных a и b, копируя их в указатели. В этом случае изменение значения по адресу ptr_a приводит к изменению значения и переменной a в функции **main**().

Любая функция может завершаться при вызове системной функции **abort**() или **exit**(**n**), где n – целое число, являющееся кодом завершения. Код завершения обычно используется программой, которая породила текущий процесс. В этом случае прекращается выполнение всей программы, автоматически закрываются все открытые файлы и освобождаются все области динамической памяти.

8.4. Параметры функции

Параметры в списке аргументов подразделяются на формальные и фактические. Фактические параметры – это параметры, формируемые в вызывающей функции и передаваемые в списке при вызове функции. Формальные – это параметры, описываемые в списке параметров вызываемой функции. Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех её объявлениях. Типы фактических параметров при вызове функции должны быть совместимы с типами соответствующих формальных параметров.

```
void fun(int,char,double);      // прототип функции
int k;                          // глобальная переменная
int main()
{
    int n=1;
    char c='a';
    fun(n,c,1.5);                //  n, c, 1.5 – фактические параметры
    return 0;
}
void fun(int nn, char cc, double dd) // nn, cc, dd – формальные параметры
{
 . . .
}
```

В функцию параметры могут передаваться следующими способами:

− используя глобальные переменные, что соответствует неявной передаче параметров;

− используя в качестве формального параметра ссылки;

− по значению, посредством копирования некоторого значения фактического параметра в формальный (локальная копия);

− используя в качестве параметров (фактического и формального) адреса.

В третьем случае изменение формального параметра не влияет на значение фактического, в остальных изменение формального параметра приводит к изменению фактического.

Область действия внешней переменной или функции распространяется от места её декларирования до конца файла, в котором они расположены. Если же на внешнюю переменную требуется сослаться до того, как она определена, или если она определена в другом файле, то при её декларировании используется слово **extern**. Например, если декларации вида

```
int i;
float mas[5];
```

расположены вне всех функций, то они определяют внешние переменные i и mas, т. е. декларирование и выделение памяти под эти объекты выполнено для остальной части файла. В то же время инструкции вида

```
extern int i;
extern float mas[];
```

декларируют, что переменная i и массив mas, имеющие тип **int** и **float**, не создаются, а память им выделяется при их объявлении в другой функции.

Для всего множества функций, образующих программу, должно быть не более одного определения каждой из внешних переменных. В объявлении массивов необходимо указывать их размерность, однако при декларировании массива с **extern** это не требуется.

8.5. Параметры по умолчанию

Иногда в объявлении функции описывается больше параметров, чем требуется в некотором частном случае при её вызове. Например, если мы хотим в некоторой функции выделить память для динамического размещения в ней числового массива, то наряду с размерностью выделяемой памяти (первый параметр) в функцию можно передать и некоторый инициализатор (второй, необязательный параметр). Однако если мы его не хотим явно передавать, то ему может быть присвоено некоторое значение по умолчанию, например:

```
void* memory(long, int =0);
```

Инициализатор второго параметра является параметром по умолчанию. То есть, если в вызове дан только один параметр, в качестве второго используется параметр по умолчанию. Задавать параметр по умолчанию возможно только для последних параметров, поэтому

```
int f(int, int =0, char* =0);     // верно
int g(int =0, int =0, char*);     // ошибка
int h(int =0, int, char* =0);     // ошибка
```

Отметим, что пробел между «*» и «=» является существенным («*=» является операцией присваивания):

```
int f(char*=0);  // синтаксическая ошибка
```

8.6. Передача массива в функцию

Если в качестве передаваемого аргумента используется массив, то в функцию передаётся *указатель на массив*.

Можно использовать *три варианта описания массива* в качестве формального параметра функции.

1. Параметр в функции может быть объявлен как массив соответствующего типа с указанием его размера.

```
#include <stdio.h>
#include <locale.h>
int max(int ms[4])      // адрес ms[0] совпадает с адресом mas[0]
{
   int k=ms[0];              //  в функции выполняется поиск
   for(int i=1; i<4; i++)    //  максимального элемента в переданном
   if(k<ms[i]) k=ms[i];      //  в неё массиве
   return k;                 //  возврат найденного значения
}
int main()
{
   setlocale(LC_ALL,"Russian" );
   int mas[4], i;
   for(i=0; i<4; i++) scanf("%d",&mas[i]);
   printf("\nмаксимальное значение = %d",max(mas));
   return 0;
}
```

В этом случае выполняется преобразование mas[i] к указателю на целый тип, и в функцию передается адрес mas[0].

2. Массив в качестве параметра в функции может быть объявлен без указания его размера. Так как сам массив в системный стек не копируется, то размер массива в общем случае компилятору не требуется.

```
int max(int ms[])     //  описание функции max
{    // тело функции аналогично описанному выше
}
```

3. Наиболее распространенный способ – объявление параметра-массива указателем на соответствующий тип данных. Функцию можно описать следующим образом:

```
int max(int *ms)
{    // тело функции аналогично описанному выше
}
```

Во всех случаях в функцию передается адрес первого элемента массива, т. е. указатель соответствующего типа.

8.7. Класс памяти

В языке С (С++) каждая переменная и функция принадлежит некоторому

классу памяти. Существуют следующие классы памяти переменных: автоматический (**auto**), статический (**static**), регистровый (**register**) и внешний (**extern**). Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти **auto**, т. е. они являются локальными.

Автоматические переменные. Переменные этого класса являются локальными, т. е. они доступны (существуют) в пределах блока, в котором они объявлены.

Память для неё автоматически выделяется каждый раз в начале блока и освобождается в конце. Доступ к такой переменной возможен только в блоке, в котором она определена.

Статические переменные. Если некоторая локальная переменная в функции декларирована со словом **static**, то в отличие от автоматических они не возникают только на период выполнения функции, а существуют в памяти постоянно. Память под **static**-переменную выделяется один раз при первом обращении к функции, в которой она декларирована.

```
void fun()
{   static int i;      // статическая переменная
 .   .   .             //   тело функции
}
```

Регистровые переменные. Спецификация **register** при объявлении переменных сообщает компилятору, что декларируемая переменная будет использоваться интенсивно. Такую переменную желательно размещать на одном из регистров машины. Это приведёт к ускорению работы программы. Пример декларации

```
register int i,j;
register char k;
```

При этом компилятор имеет право проигнорировать указание разместить переменные в регистре машины. Спецификация **register** может применяться только к арифметическим переменным и к формальным параметрам функции, например:

```
fun(register unsigned int n, register char c)
{ register int k;
  . . .
}
```

Ограниченное число регистров накладывает в свою очередь ограничение на число **register**-переменных.

8.8. Примеры программ

Пример 1. Разработать функцию, выполняющую анализ: является ли одна строка подстрокой для второй строки.

```
#include <stdio.h>
#include <conio.h>
#include <locale.h>
   /* Функция поиска получает указатели на анализируемые строки */
int search(char str1[],char str2[])
{
```

```
    int i, j;
    for ( i=0; *(str1+i) ;)
    {
       for ( j=0; *(str1+i ) && *(str2+j); ) // посимвольное сравнение строк str1 и str2
       {
          if (*(str1+i++) != *(str2+j++))
          {
             break;    // выход из цикла если символы в строках не совпали
          }
          if (!*(str2+j))
          {
             return 1; // достигнут конец строки str2
          }
       }
    }
    return 0;            // возврат признака, что строка str2 не найдена в str1
}
int main()
{
    setlocale(LC_ALL,"RUS");
    char *str="abcdefg    12345"; // объявление строки str
    printf("Строка %s ","123");
    if(!search(str,"123"))
    {
       printf(" не ");     // строка "123" не найдена в строке str
    }
    printf("найдена в строке %s\n",str);
    getch();
    return 0;
}
```

Пример 2. Разработать функцию, выполняющую сортировку слов в строке в порядке убывания (выбором).

```
#include<stdio.h>
#include<conio.h>
   /* Функция возвращает указатель начала слова или NULL, если нет слов */
char *find(char *str)
{
   int len, max_len; char *max_word;
   for (len=0, max_len=0, max_word=NULL; *str!='\0'; str++)
   {
      if ( *str!=' ')    // символ, адресуемый через str, не равен
      {
         len++;       // символ слова увеличит счётчик длины слова
      }
      else
      {       // фиксация максимального значения перед сбросом счётчика
         if (len>max_len)
         {
            max_len=len;
            max_word=str−len;
         }
```

```
            len=0;
        }
    }
    if (len > max_len)   // то же самое для последнего слова
    {
        max_word=str−len;
    }
    return max_word;   // возвращается указатель на максимальное слово
}
    /* Сортировка слов в строке в порядке убывания (выбором) */
void sort(char *in, char *out)
{
    char *max_word;
    while((max_word=find(in))!= NULL)   // получить индекс очередного слова
    {
        for (; *max_word!=' ' && *max_word!='\0'; )
        {
            *(out++) = *max_word;
            *(max_word++) =' ';   // переписать с затиранием
        }
        *(out++) =' ';        // после слова добавить пробел
    }
    *(out--)='\0';            // пробел в конце заменяется на символ конца строки
}
int main()
{
    char str1[]="this is unsorted words", str2[80];   // объявление строк
    sort(str1,str2);    // сортировка str1 и запись результата в str2
    puts(str2);
    getch();
    return 0;
}
```

Пример 3. Ввести массив строк. Для каждой строки во всех последующих строках найти слова, совпадающие со словом из выбранной строки и удалить их (в исходной строке слова не удалять).

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<locale.h>
int str_cmp(char *,char *);
void word_del(char *);
int main()
{
    setlocale(LC_ALL,"RUS");
    char **str;
        // i_seek и j_seek − индексы поиска в последующих строках
    int i,i_seek,j,j_seek,n,m;
    system("CLS");
    printf("\nВведите количество строк и их длину:  ");
    scanf("%d%d",&n,&m);
```

```
    str=(char**)calloc(n,sizeof(char*));
    for(i=0;i<n;i++)   // ввод строк
    {
       *(str+i)=(char*)malloc(m*sizeof(char));
       fflush(stdin);  // очистка входного потока
       printf("\nВводите %d-ю символьную строку :",i+1);
       gets(*(str+i));
    }
    printf("\nСтроки до преобразования:");
    for (i=0;i<n;i++)
    {
       printf("\n    %s",str[i]);
    }
    for(i=0;i<n-1;i++)              // перемещение между строками
    {
       for(j=0;*(*(str+i)+j);j++)  // перемещение внутри строк
       {
          if (j==0 || *(*(str+i)+j-1)==' ')  // выбрано исходное слово
          {       // поиск исходного слова в других строках
             for(i_seek=i+1;i_seek<n;i_seek++)
             {
                for(j_seek=0;*(*(str+i_seek)+j_seek);j_seek++)
                {      // выбрано слово для сравнения
                   if (j_seek==0 || *(*(str+i_seek)+j_seek-1)==' ')
                   {
                      if (!str_cmp(*(str+i)+j,*(str+i_seek)+j_seek)) // слова совпали
                      {      // удаление исходного слова из новой строки
                         word_del(*(str+i_seek)+j_seek);
                         j_seek--; // возврат к началу слова
                      }
                   }
                }
             }
          }
       }
    }
    printf("\nСтроки после преобразования");
    for (i=0;i<n;i++)
    {
       printf("\n    %s",str[i]);
    }
    getch();
    return 0;
}
/* Функция сравнения двух слов */
int str_cmp(char *str1,char *str2)
{    // продвижение по словам
    while(*str1==*str2 && *str1 && *str2 && *str1!=' ' && *str2!=' ')
    {
       str1++;
       str2++;
```

```
    }
     /* Отличие символа конца строки от пробела не должно повлиять
        на результат сравнения */
    if(*str1=='\0' && *str2==' ' || *str1==' ' && *str2=='\0')
    {
       return 0;
    }
    else
    {
       return *str1−*str2;   // возвращается разность кодов
    }
}
/* Функция удаления из строки слова */
void word_del(char *word)
{
    int i=0,j=0;
    while(*(word+i) && *(word+i)!=' ')
    {
       i++;    // подсчёт длины слова
    }
    if(*(word+i))
    {
       i++;    // выход на первую букву следующего слова
    }
    while(*(word+i))
    {
       *(word+j++)=*(word+i++);   // сдвиг строки влево на размер слова
    }
    *(word+j)='\0';   // усечение строки
}
```

8.9. Указатели на функции

В языке С (С++) функция не может быть значением переменной, в то же время аналогично переменным функция физически расположена в памяти и соответственно имеет адрес.

Указатель на функцию – это такой тип переменной, которой можно присваивать *адрес точки входа* в функцию, т. е. адрес первой исполняемой команды. Эта переменная в дальнейшем может использоваться для вызова функции вместо её имени, а также позволяет передавать функцию как обычный параметр в другую функцию. Следовательно, с указателем на функцию можно обращаться, как с переменной (передавать его другим функциям, помещать в массивы и т. д.). *Определение* указателя на функцию имеет следующий общий вид:

тип_результата (*имя_указателя)(список_типов_параметров);

В объявлении

```
int (*fun)(int, int *);
```

переменная fun является указателем на функцию с двумя параметрами: типа **int** и указателем на **int**. Сама функция должна возвращать значение типа **int**. Круг-

лые скобки, содержащие имя указателя fun и признак указателя «*», обязательны, иначе запись

```
int *fun (int, int *);
```

будет интерпретироваться как объявление функции fun, возвращающей указатель на **int**, т. к. приоритет префиксного оператора «*» ниже, чем приоритет «()».

Вызов функции возможен только после инициализации значения указателя fun и имеет вид

```
(*fun)(i,&j);
```

В этом выражении для получения адреса функции, на которую ссылается указатель fun, используется операция разадресации «*». Вызов функции через указатель возможен также обычным способом:

```
fun(i,&j);
```

Ниже приведён пример использования указателя на функцию, инициализированного адресом системной функции strcmp.

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
# define n 80
void check(char *, char *, int ( cmp)(const char *,const char *));
int main()
{
   setlocale(LC_ALL,"Russian" );
   char s1[n],s2[n];
   int (*p)(const char *,const char *);
   p=strcmp;              // инициализация указателя p адресом функции strcmp()
   gets(s1);  gets(s2); // ввод двух строк
   check(s1,s2,p);       // вызов функции check
   return 0;
}
void check(char *a, char *b, int ( cmp)(const char *,const char *))
{
   printf("\n строки ");
   if (!(*cmp)(a,b)) printf("равны");
   else printf("не равны");
}
```

При вызове функции check ей передается два указателя на символьные строки и один на функцию. Аналогично можно обращаться к check, используя и указатели на другие функции, если возвращаемый тип и передаваемые параметры совпадают с рассмотренными выше. Выражение if(!(*cmp)(s1,s2)) можно заменить на if(!cmp(s1,s2)). Можно вызвать функцию check также следующим образом: check(s1,s2,strcmp). При этом не требуется использовать дополнительно указатель p.

8.10. Примеры программ

Пример 1. Использование указателя на функцию в качестве параметра функции, вычисляющей производную от функции cos(x).

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

```
#include <Windows.h>
#include <locale.h>
#include <math.h>
double DerivativeOfTheFunction(double x, double dx,
                                double(*functionPointer)(double x));
double Function(double x);
int main()
{
   double x;       //  точка, в которой вычисляется производная функции
   double dx;      //  приращение функции
   double value; //  значение производной функции в точке x
      /* Указатель на функцию, возвращающую значение типа double
         и принимающую переменную типа double */
   double (*functionPointer)(double x);
   setlocale(LC_ALL, "Russian");
   printf("Введите x = ");
   scanf("%lf", &x);   // lf – модификатор типа double
   printf("Введите dx = ");
   scanf("%lf", &dx);
    /* Указателю на функцию присвоили адрес вызова функции Function */
   functionPointer = Function;
   value = DerivativeOfTheFunction(x, dx, functionPointer);
   printf("Производная функции в точке %lf = %lf\n", x, value);
   system("pause");   // программа просит нажать любую клавишу
   return 0;
}
  /* Функция, вычисляющая производную в точке x */
double DerivativeOfTheFunction(double x, double dx, double (*f)(double x))
{
   return (f(x + dx) – f(x)) / dx;
}
  /* Функция, от которой вычисляется производная */
double Function(double x)
{
   return (cos(x));
}
```

Пример 2. Использование указателя на функцию на примере вычисления корня нелинейного уравнения методом «половинного деления».

```
#include <stdio.h>
#include <locale.h>
#include <math.h>
double Function(double);
double BisectionMethod(double, double, double, double(*)(double));
     //Bisection – (англ.) половинное деление
int main()
{
   double a, b, accurasy;  // accurasy – (англ.) точность
   double (*fP)(double x); // указатель на функцию Function
   setlocale(LC_ALL,"Russian");
   printf("Введите левую границу a = ");
   scanf("%lf", &a);
```

```
    printf("\nВведите правую границу b = ");
    scanf("%lf", &b);
    printf("\nВведите точность вычисления корня accurasy = ");
    scanf("%lf", &accurasy);
    fP = Function;
    printf("a = %5.2lf  b = %5.2lf  корень = %6.4lf\n", a, b, BisectionMethod(a, b,
          accurasy, fP));
    system("pause");
    return 0;
}
double BisectionMethod(double a, double b, double accurasy, double(*fP)(double))
{
    double fa, fb, fc, c;
    do
    {
        fa = (*fP)(a);    // нахождение значений функции на концах
        fb = (*fP)(b);    // выбранного интервала   [a,b]
        c = (a + b) / 2; // деление отрезка [a,b] пополам
        fc = (*fP)(c);    // значение функции в середине отрезка
        if (fa * fc > 0)
            a = c;          // левой границей интервала становится c, т. е. [c,b]
        else
            if (fb * fc > 0)
                b = c;      // правой границей интервала становится c, т. е. [a,c]
            else
            {
                puts("Найден точный корень");
                return c;
            }
    }
    while (fabs(a − b) > accurasy); // цикл − пока не достигнута точность  accurasy
        return c;
}
double Function(double x)  // вычисление значения функции f(x)=x^2−3 в точке x
{
    return (x * x − 3);
}
```

Пример 3. Использование указателя на функцию на примере вычисления интеграла методом «прямоугольников».

```
#include <stdio.h>
#include <locale.h>
#include <math.h>
double Functon(double);
double RectanglesMethod(double(*)(double), double, double, double, int);
// Rectangle − (англ.) прямоугольник
int main()
{
    double a, b, accurasy;
    double (*fP)(double);
    int rectanglesCount;
    setlocale(LC_ALL, "Russian");
```

```
    printf("Введите границу a = ");    // a – левая граница интегрирования
    scanf("%lf", &a);
    printf("\nВведите границу b = "); // b – правая граница интегрирования
    scanf("%lf", &b);
    printf("\nВведите точность вычисления корня accurasy = ");
    scanf("%lf", &accurasy);
    printf("\nВведите количество прямоугольников на интервале [a,b] = ");
    scanf("%d", & rectanglesCount);
    fP = Functon;
    printf("  Integral = %6.4lf\n",
              RectanglesMethod(fP, a, b, accurasy,rectanglesCount));
    system("pause");
    return 0;
}
double RectanglesMethod(double(*fP)(double), double a, double b,
                                double accurasy, int rectanglesCount)
{
     // высота прямоугольника на оси X, где b и a – границы интегрирования
    double currentSum = 0.0, previouSum, x, height = (b – a) / rectanglesCount;
   do
    {
       previouSum = currentSum;
       currentSum = 0.0;
         /* Находим сумму всех прямоугольников на интервале [a,b],
            где (*fP)(x) * height – площадь одного прямоугольника*/
       for (x = a; x < b; x += height)
          currentSum += (*fP)(x)* height;
         // уменьшаем высоту, чтобы с большей точностью находить площади
       height /= 2;
    }
    while (fabs(previouSum - currentSum) > accurasy);
     /* Сравниваем текущую сумму площадей с суммой на предыдущем
        шаге интегрирования */
    return currentSum;
}
double Functon(double x)
{
    return (x*x – 4);    // вычисление значения функции f(x)=x^2–4 в точке x
}
```

8.11. Рекурсия

Функция является рекурсивной, если некоторая инструкция этой функции содержит вызов самой этой функции. Компилятор допускает любое число рекурсивных вызовов. Рекурсивный вызов не создает новую копию функции. При каждом вызове для формальных параметров и переменных с классом памяти **auto** и **register** выделяется новая область памяти, так что их значения из предыдущих вызовов не теряются, но в каждый момент времени доступны только значения текущего вызова.

Переменные, объявленные с классом памяти **static**, не требуют выделения новой области памяти при каждом рекурсивном вызове функции и их значения доступны в течение всего времени выполнения программы.

Хотя компилятор языка С (С++) не ограничивает число рекурсивных вызовов функций, это число ограничивается ресурсом памяти компьютера. Большое число рекурсивных вызовов функции может привести к переполнению стека, что в свою очередь приведёт к ошибочному окончанию работы программы. При разработке рекурсивной функции следует, используя операторы **if** и **return**, предусмотреть возможность завершения её работы. В противном случае возможно «зацикливание» программы.

8.12. Примеры программ

Пример 1. Программа нахождения наибольшего общего делителя двух чисел.

```
#include <stdio.h>
#include <conio.h>
#include <locale.h>
/* Рекурсивная функция поиска НОД двух чисел, GCD = НОД */
int findGCD (int firstNumber, int secondNumber);
int main()
{
   int firstNumber, secondNumber;
   setlocale(LC_ALL, "Russian");       // устанавливаем русский язык
   printf("Введите два числа, НОД которых вы хотите найти: ");
   scanf("%d%d", &firstNumber, &secondNumber);   // принимаем два числа
   printf("НОД чисел %d и %d  =  %d", firstNumber, secondNumber,
            findGCD(firstNumber, secondNumber));
   getch();
   return 0;
}
int findGCD(int firstNumber, int secondNumber)
{
   if (secondNumber == 0)   // если второе число равно 0,
      return firstNumber;      // то возвращаем первое число
   else                        // иначе вызываем функцию с новыми параметрами
      return findGCD(secondNumber, firstNumber % secondNumber);
}
```

Пример 2. Вычисление факториала числа n (n!).

```
#include <stdio.h>
#include <conio.h>
#include <Windows.h>
#include <locale.h>
int Factorial1(int number); // первый вариант функции поиска факториала
int Factorial2(int number); // второй вариант функции поиска факториала
int Factorial3(int number); // третий вариант функции поиска факториала
int main()
{
   int number, temp;
```

```
    setlocale(LC_ALL, "Russian"); // устанавливаем русский язык
    do // запускаем проверку на ввод
    {
       system("cls"); // очищаем экран
       puts("\nВведите число >= 0 для вычисления его факториала :");
       temp = scanf("%d", &number);    // получаем число
    } while (temp < 1 || number < 0);     // пока ничего не введено или число < 0
    printf("\n%d! = %d   вариант 1", number, Factorial1(number));
    printf("\n%d! = %d   вариант 2", number, Factorial2(number));
    printf("\n%d! = %d   вариант 3", number, Factorial3(number));
    getch();
    return 0;
}
int Factorial1(int number)
{
    if (number <= 1)  // если число = 0 или 1
       return 1;         // факториал равен 1
    else           // иначе заново вызываем функцию с уменьшенным на 1 числом
       return Factorial1(number – 1) * number;       // и умножаем на текущее число
}
int Factorial2(int number)
{
    static long i;  // i – статическая переменная, сохраняющая своё последнее
                    // значение при каждом новом вызове функции
    int factorial;
    if(number <= 1)     // если число = 0 или 1
       return 1;          // факториал равен 1
    if (++i < number)   // используем прекремент, т. е. предварительное
    {                         // увеличение i на единицу
       factorial = i*Factorial2(number); // заново вызываем функцию факториала
       i--;
       return factorial;
    }
    i--;
    return number;
}
int Factorial3(int number)
{ /* Аналогичен первому варианту, но используется тернарный оператор (?:) */
    return (number>1) ? number*Factorial3(number – 1) : 1;
}
```

Пример 3. Программа вывода последовательности Фибоначчи, состоящей из 15 членов.

```
#include <stdio.h>
#include <conio.h>
#include <locale.h>
int fibonacci(int number);   // функция вывода последовательности Фибоначчи
int main()
{
    int i;
    setlocale(LC_ALL, "Russian");   // устанавливаем русский язык
    printf("\nПервые 15 членов последовательности Фибоначчи:\n");
```

```
    for (i = 0; i < 15; i++)   // цикл для вывода 15 членов последовательности
        printf("%4d", fibonacci(i));
    getch();
    return 0;
}
int fibonacci(int n)
{
    return ((n == 0 || n == 1)) ? 1 : fibonacci(n – 1) + fibonacci(n – 2);
}
```

Пример 4. Программа перевода чисел из десятичной системы счисления в систему счисления с основанием до 32.

```
#include <stdio.h>
#include <conio.h>
#include <locale.h>
void transferIntegerPart(int);                  // функция перевода целой части числа
void transferFractionalPart(double, int); // функция перевода дробной части числа
int newRadix; // основание новой с/с
char sign = '+'; // по умолчанию знак = '+'
int main()
{
    int precision;                          // точность перевода
    double number;                          // исходное число
    setlocale(LC_ALL, "Russian");
    printf("Введите исходное число: ");
    scanf("%lf", &number);                  // lf – модификатор типа double
    number < 0 ? sign = '–', number *= –1 : number; // выделяем знак числа
    printf("\nВведите основание новой с/с:\n");
    scanf("%d", &newRadix);
    printf("\nВведите точность для дробной части числа с/с:\n");
    scanf("%d", &precision);
        // переводим целую часть
    transferIntegerPart((int)number);   // в скобках приведение типа double к int,
    putchar('.');                           // вставляем точку после целой части
        // переводим дробную часть
    transferFractionalPart(number – (int)number, precision);
    getch();
    return 0;
}
void transferIntegerPart(int number)
{
    int modulo;                             // остаток от деления
    if (number >= newRadix)
    {  // получаем остаток от деления на основание
        modulo = number % newRadix;
        number /= newRadix;                 // получаем целую часть от деления
        transferIntegerPart(number);  // рекурсивный вызов функции
            /* Используется посимвольный вывод +'A' для вывода букв
               алфавита в с/с больших 9; + '0' для вывода цифр */
        printf("%c", modulo > 9 ? modulo – 10 + 'A' : modulo + '0');
    }
    else
```

```
        {
            printf("%c %c", sign, number > 9 ? number – 10 + 'A' : number + '0');
        }
    }
    void transferFractionalPart(double number, int precision)
    {
        int count;              // счётчик для достижения заданной точности
        int integerPart;        // целая часть
        if (number != 0 && (count++)<precision)
        {     // получаем произведение и выделяем целую часть
            integerPart = number *= newRadix;
            printf("%c", integerPart>9 ? integerPart – 10 + 'A' : integerPart + '0');
            transferFractionalPart(number – (int)number, precision); // рекурсивный вы-
зов
        }
        else
        {
            printf("%c", number > 9 ? (int)number – 10 + 'A' : (int)number + '0');
        }
    }
```

8.13. Аргументы в командной строке

В операционной среде имеется возможность передавать аргументы запускаемой программе посредством командной строки. *Аргументы командной строки* – это информация, следующая за именем программы в командной строке операционной системы. Функция main() может получать три аргумента. Первые два из них обычно обозначаются **argc** и **argv**. В параметре **argc** содержится количество передаваемых в командной строке аргументов. Значение argc не может быть меньше единицы, т. к. имя программы рассматривается как первый аргумент. Второй параметр – **argv**[] является указателем на массив символьных строк. Каждый элемент данного массива указывает на аргумент командной строки. Аргументы командной строки должны отделяться пробелами или табуляцией. Запятые, точки и прочие символы не рассматриваются как разделители.

```
C:\>fun.exe 3 aaa bbb ccc  –  argc=4
C:\>fun.exe 3 aaa,bbb,ccc  –  argc=2
```

Если необходимо в качестве аргумента передать строку, содержащую пробелы и табуляции, то её следует заключить в двойные кавычки. Например:

```
C:\>fun.exe 3 aaa "bbb ccc" ddd "eee fff"  –  argc=5
```

Объявление **char *argv[]** указывает на то, что массив **argv** не имеет фиксированной длины. В **main**() с массивом **argv** можно работать как с обычным массивом символьных строк. Например:

```
# include<stdio.h>
int main(int argc, char *argv[])
{
    int i,j;
    for(i=1;i<argc;i++) // рассматриваются только переданные аргументы
    {
```

```
        j=0;                // вывод посимвольно каждой из переданных строк
        while(argv[i][j]) printf("%c",argv[i][j++]);
        printf("\n");
    }
    return 0;
}
```

Помимо **argc** и **argv**, имеется третий аргумент – **env**. Этот параметр позволяет программе получить доступ к информации о среде операционной системы и следует за **argc** и **argv**. Как и **argv**, **env** является указателем на массив строк, где каждая строка – это строка среды, определённая операционной системой. Параметр **env** не имеет аналога параметра **argc**, сообщающего о количестве строк среды. Для этого используется нулевая последняя строка среды.

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

Вместо традиционных имен аргументов **argc**, **argv** и **env** можно использовать любые другие имена.

8.14. Функции с переменным числом параметров

В языке С (С++) наряду с использованием функций с фиксированным числом параметров можно использовать и функции с переменным числом параметров, т. е. функции, в которые можно передавать данные, не описывая их в прототипе и заголовке функции. Описания этих данных заменяются *тремя точками*. В таких функциях должен находиться также один или более постоянный параметр. Если в функции имеется несколько постоянных параметров, то сначала перечисляются эти параметры, а затем ставятся три точки.

В случае использования функции с переменным числом аргументов программист сам контролирует реальное количество аргументов, находящихся в стеке, и отвечает за выбор из стека дополнительных аргументов. В стек информация заносится словами. Целочисленные данные преобразуются к типу **int**, а данные с плавающей точкой – к типу **double**.

Возможность передачи переменного списка параметров определяется способом (порядком) записи аргументов функции в стек программы и способом считывания параметров из стека.

В С (С++) аргументы, заданные в списке фактических параметров при вызове функции, размещаются в стеке с *конца к началу списка* параметров. При этом в вершине стека размещается первый аргумент, например, при вызове **sum(a,b,c)** аргументы **a**, **b** и **c** в стеке размещаются следующим образом (рис. 3):

Рис. 3. Структура стековой памяти

Возможны *два способа задания длины* переменного списка параметров:

− указание числа аргументов, передаваемых в функцию;

− задание признака конца списка аргументов.

Реализовать функции с переменным числом параметров можно *тремя способами*:

− используя указатель без типа, например **void** *;

− используя указатель, соответствующий типу переменных списка параметров, например **int** *, **double** *;

− используя указатель, определённый самой системой программирования (с помощью стандартных макросов **va_list**, **va_start**, **va_arg**, **va_end**).

Существует несколько способов описания данных, передаваемых в функцию. Рассмотрим их на примерах.

```
#include<stdio.h>
#include <locale.h>
int sum(int,…);
int main()
{
   int a,b,c;
   setlocale(LC_ALL,"Russian" );
   scanf("%d%d%d",&a,&b,&c);
   printf("сумма 2 чисел равна %d",sum(3,a,b,c));
   printf("сумма 3 чисел равна %d",sum(2,a,b));
   return 0;
}
int sum(int k,…)          // в функции вычисляется сумма произвольного
{
   int s=0,*p;            // числа параметров, где k – их число
   p=&k;                  // указатель на количество аргументов
   p++;                   // указатель на первый суммируемый аргумент
   for(i=0;i<k;i++)
   s+=*p++;               // подсчёт суммы параметров
}
```

В примере первый аргумент k является количеством чисел, которые будут считаны из стека.

Приводимый ниже пример иллюстрирует применение бестипового указателя для обработки списка переменной длины, в котором в качестве параметров передаются данные различных типов: **int**, **float**, **char**. При вызове функции задаётся длина списка аргументов и тип данных.

```
#include<stdio.h>
#include<conio.h>
enum Type{Char,Int,Float};  // объявление перечисляемого типа Type
void fun(enum Type,...);       // прототип функции с переменным числом
int main()                          // параметров
{
   fun(Char,'a','b','c','d','e','\0');         // вызов функции с параметрами типа  char
   fun(Int,3,4,7,9,0);                          // вызов функции с параметрами типа  int
   fun(Float,1.2,(float)5,5.,(float)0);   // вызов ф-ции с параметрами типа  float
   return  0;
}
void fun(enum Type t,...)
{
   void *p=&t;
   p=((int *)p)+1;    // устанавливаем указатель на первый нефиксированный
   puts("\n");          // параметр, приводя указатель void* к типу параметра,
   switch(t)            // следующего за фиксированным
   {
     case Char :              // блок считывания параметров типа char
            while(*(int *)p)
            {  printf("%c",*((int *)p));  // вывод считанной буквы
               p=(int *)p+1;                // переход к следующему параметру
            }  break;
     case Int  :               // блок считывания параметров типа int
           while(*(int *)p)
           {  printf("%d ",*((int *)p));   // вывод считанного int-числа
              p=((int *)p)+1;                // переход к следующему параметру
           }  break;
     case Float:              // блок считывания параметров типа double
           while(*(double *)p)
           {  printf("%f ",*((double *)p)); // вывод считанного double-числа
              p=((double *)p)+1;             // переход к следующему параметру
           }
   }
}
```

В языке C (C++) имеются четыре стандартные функции (макрокоманды) для работы со списком переменной длинны, описанные в библиотеке **stdarg.h**: **va_list**, **va_start**, **va_arg**, **va_end**. Изменим рассмотренную выше функцию fun, чтобы в ней были использованы эти функции:

```
void out(int k,enum Type t,...)
{
  va_list pr;
   va_start(pr,t);     // указатель pr на начало списка параметров
```

```
    printf("\n");
    while(k--)
    {
      switch(t)        //  считывание параметров из стека
      {  case Char: printf("%c",va_arg(pr,int)); break;          // параметры типа char
         case Int: printf(" %d ",va_arg(pr,int)); break;         // параметры типа int
         case Float: printf("  %f  ",va_arg(pr,double)); break; // параметры типа double
         default: puts("ошибка");
       }
     }
     va_end(pr);
}
```

Макрокоманда **va_list** позволяет определить бестиповой указатель pr, далее используемый для работы со списком переменной длины. Указатель используется макрокомандами **va_start**, **va_arg** и **va_end**.

Макрокоманда **va_start**(pr) устанавливает переменную pr (имеющую тип **va_list**) в качестве указателя на первый необязательный аргумент в списке параметров, передаваемых функции. Макрокоманда **va_start** должна вызываться перед первым вызовом макрокоманд **va_arg** или **va_end**.

Макрокоманда **va_arg**(pr,t) при каждом очередном вызове позволяет извлечь из стека очередной элемент списка переменных параметров и присвоить переменной pr адрес очередного элемента из стека. Второй аргумент t функции **va_start** является именем типа данного, считываемого из стека. В макрокоманде **va_arg** можно использовать типы данных, определенные языком C (C++).

Макрокоманда **va_end**(pr) устанавливает значение указателя pr равным **NULL** и должна быть вызвана только после считывания последнего аргумента. В противном случае её вызов может повлечь непредсказуемое поведение программы. После выполнения **va_end** нарушается связь указателя с элементами стека, но информация в стеке сохраняется до окончания работы этой функции.

8.15. Примеры программ

Пример 1. Реализовать функцию с произвольным числом параметров различного типа. Доступ к данным (параметрам) реализовать с помощью бестиповых указателей (**void** *pointer).

```
void **function(int, ...);
int main()
{
   double d1, d2, d3, d4;
   int i1, i2, i3, i4;
   void **arrayOfSums; // для возврата из function адресов полученных сумм
   setlocale(LC_ALL, "Russian");
   puts("Введите целые числа:");
   scanf("%d%d%d%d", &i1, &i2, &i3, &i4);
   puts("Введите вещественные числа:");
   scanf("%lf%lf%lf%lf", &d1, &d2, &d3, &d4);
   arrayOfSums = function(3, 'd', 3, d1, d2, d3, 'i', 4, i1, i2, i3, i4, 'd', 2, d3, d4);
```

```
    printf("\n Сумма целых чисел = %d\n Сумма дробных чисел = %5.2lf",
            *((int*)*arrayOfSums), *((double*)*(arrayOfSums + 1)));
    _getch();
    return 0;
}
/* Функция суммирования чисел типа int и double, результаты
   возвращаются в main */
void ** function(int n, ...)  // n – число групп
{
    int i, j;
    void **array; // массив бестиповых указателей
    int *intSum;  // накопитель сумм целых чисел
    double *doubleSum; // накопитель сумм вещественных чисел
    char c;
    array = (void **)calloc(2, sizeof(void *));
    intSum = (int *)calloc(1, sizeof(int));
    doubleSum = (double *)calloc(1, sizeof(double));
    void *pointer = &n;              // pointer устанавливается на начало списка
    pointer = ((int *)pointer) + 1; // параметров переменной длинны

    while (n--)
    {
        c = *(char *)pointer;             // считывание символа
        pointer = ((int *)pointer) + 1; // указатель на число переменных в группе
        switch (c)
        {
        case 'd':
            i = *(int *)pointer;                // считывание числа переменных в группе
            pointer = ((int *)pointer) + 1; // сдвиг указателя на первую переменную
                                                // в группе
            for (j = 0; j < i; j++)
            {
                *doubleSum += *((double *)pointer);  // накопление суммы double чисел
                pointer = ((double *)pointer) + 1;       // сдвиг указателя
            }
            break;
        case 'i':
            i = *((int *)pointer);               // считывание числа переменных в группе
            pointer = ((int *)pointer) + 1;  // сдвиг указателя на первую переменную
            for (j = 0; j < i; j++)
            {
                *intSum += *((int *)pointer);  // накопление суммы int чисел
                pointer = ((int *)pointer) + 1; // сдвиг указателя
            }
        }
    }
    array[0] = (void *)intSum;        // запись адресов полученных
    array[1] = (void *)doubleSum; // сумм в массив
    return array;
}
```

Пример 2. Реализовать функцию с переменным числом параметров, принимающую произвольное число символьных строк и выводящих их на экран. Рассмотреть различные способы реализации этой функции (используя указатель **char***, **void*** и макросы **va_list**, **va_start**, **va_arg**, **va_end**).

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <locale.h>
#include <stdarg.h>

void firstFunction(char *, ...);        // прототипы функций с одним
void secondFunction(char *, ...);   // постоянным параметром
void thirdFunction(char *, ...);
int main()
{
   char *firstString = "aaa aaaa aaaa",
        *secondString = "bb bbb b",
        *thirdString = "cccc cccccc  ccccc",
        *fourthString = NULL;
   setlocale(LC_ALL, "Russian");
   firstFunction(firstString, secondString, thirdString, fourthString);
   secondFunction(firstString, secondString, thirdString, NULL);
   thirdFunction(firstString, secondString, thirdString, fourthString);
   return 0;
}
  /* Передача в функцию строк, используя параметры переменной длины
     считывание указателей на передаваемые в функцию строки осуществляется с использованием указателя на указатель (char **pointer) */
void firstFunction(char *string, ...)
{
   char **pointer;
   pointer = &string;
   puts("\n Первая функция");
   while (*pointer)    // цикл пока в стеке не NULL
   {
      printf("\n%s", *pointer);    // вывод строки на экран
      pointer++;       // переход к новому элементу (адресу строки) в стеке
   }       // pointer увеличивается на величину, равную размеру указателя
   _getch();
}
 /* Передача в функцию строк, используя параметры переменной длины.
   Считывание указателей на передаваемые в функцию строки осуществляется с использованием бестипового указателя на указатель (void **pointer) */
void secondFunction(char *string, ...)
{
   void **pointer;
   pointer = (void **)&string;
   puts("\n Вторая функция");
```

```
    while (*(char **)pointer)      // цикл пока в стеке не NULL
    {
       printf("\n%s", *(char **)pointer);  // вывод строки на экран
       pointer++;   // переход к новому элементу (адресу строки) в стеке
    }       // **pointer можно не приводить к типу (char **)
          // т. к. приращение pointer равно величине указателя
    _getch();
}
  /* Передача в функцию строк, используя параметры переменной длины
     считывание указателей на передаваемые в функцию строки осуществля-
     ется с использованием макросов (va_list, va_start, va_arg, va_end) */
void thirdFunction(char *string, ...)
{
    va_list pointer; // объявление бестипового указателя
  // pointer устанавливается на первый переменный параметр ("ааа аааа аааа")
    va_start(pointer, string);
    puts("\n Третья функция");
    while (string != NULL) // цикл пока в стеке не NULL
    {
       printf("\n%s", string);    // вывод строки на экран
          // переход к новому элементу (адресу строки) в стеке
       string = va_arg(pointer, char *);
    }
    _getch();
}
```

Пример 3. Используя функцию с переменным числом параметров, получающую список массивов целых чисел, вычислить и возвратить в **main**() максимальную сумму элементов массива.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <locale.h>
int function(int, int*, int, ...);
int main()
{
    int firstArray[] = { 1, 2, 3 },
        secondArray[] = { 4, 5, 6, 7 },
        thirdArray[] = { 8, 9 },
        firstArraySize,
        secondArraySize,
        thirdArraySize;
    setlocale(LC_ALL, "Russian");
    firstArraySize = sizeof(firstArray) / sizeof(int);
    secondArraySize = sizeof(secondArray) / sizeof(int);
    thirdArraySize = sizeof(thirdArray) / sizeof(int);
    puts("Результат первого вызова функции:");
    printf("%d", function(3, firstArray, firstArraySize, secondArray,
           secondArraySize, thirdArray, thirdArraySize));
    puts("\n\nРезультат второго вызова функции:");
```

```
    printf("%d", function(2, firstArray, firstArraySize, thirdArray,
                                thirdArraySize));
    _getch();
    return 0;
}
    /* Первый параметр, передаваемый в функцию – число массивов,
       второй – указатель на массив, третий – размерность массива */
int function(int arraysNumber, int *array, int arraySize, ...)
{
    int *pointer1 = &arraySize; // указатель на размерность массива
    int **pointer2 = &array; // указатель на указатель в стеке на массив
    int maxSum = 0, sum, i;
    while (arraysNumber--) // k == 0, если все массивы обработаны
    {
        sum = 0; // сумма для накопления по очередному массиву
        for (i = 0; i < *pointer1; i++)
            sum += *(*pointer2 + i);
        if (maxSum < sum) // сравнение с контрольной суммой
            maxSum = sum;
        pointer2 += 2; // pointer2 передвигается на очередной массив (указатель)
        pointer1 += 2; // pointer1 передвигается на его размерность
    }
    return maxSum; // возврат найденной суммы
}
```

9. Сортировка

Сортировка применяется во многих программах, оперирующих информацией. Цель сортировки – упорядочить информацию и облегчить поиск требуемых данных. Существует много методов сортировки данных. Причём для разных типов данных иногда целесообразно применять разные методы сортировки.

Практически каждый алгоритм сортировки можно разбить на три части:

− сравнение двух элементов, определяющее упорядоченность этой пары;

− перестановку, меняющую местами неупорядоченную пару элементов;

− сортирующий алгоритм, определяющий выбор элементов для сравнения и отслеживающий общую упорядоченность массива данных.

Сортировка данных в оперативной памяти называется внутренней, а сортировка данных в файлах называется внешней. Ниже будут рассматриваться методы сортировки данных в оперативной памяти.

9.1. Пузырьковая сортировка

Название метода отражает его суть. «Легкие» (например, меньшие по значению) элементы массива «всплывают» вверх (в начало), а «тяжелые» опускаются вниз (в конец). Пузырьковая сортировка проходит по массиву снизу вверх. Каждый элемент массива сравнивается с элементом, который находится непосредственно над ним. И если при их сравнении оказывается, что они удовлетворяют условию их перестановки, то она выполняется. Процесс сравнений и перестановок продолжается до тех пор, пока «легкий» элемент не «всплывет» вверх. В конце каждого прохода по массиву верхняя граница сдвигается на один элемент вниз (вправо). Сортировка продолжается, анализируя все меньшие массивы. В примере выполняется сортировка элементов массива размерностью k.

```
void puzirek(int *ms, int k)
{
   int i,j,m;
   for(i=0;i<k−1;++i)          // выбор верхней границы массива
   for(j=k−1;j>i;--j)          // просмотр массива «снизу»  «вверх»
   {
      if(ms[j−1]>ms[j])        // условие замены выполнено
      {
         m=ms[j−1];            // замена j−1  и j  элементов, используя
         ms[j−1]=ms[j];        // дополнительный элемент m
         ms[j]=m;
      }
   }
}
```

Внутренний цикл for выполняет проход по подмассиву в направлении, обратном внешнему циклу. Если изменить направление этих циклов, то не наименьший будет «всплывать», а наибольший «тонуть».

9.2. Шейкер-сортировка

В этом методе учитывается тот факт, что от последней перестановки до конца массива будут находиться уже упорядоченные данные. Тогда просмотр

имеет смысл делать не до конца массива, а до последней перестановки на предыдущем просмотре. Если же просмотр делать попеременно в двух направлениях и фиксировать нижнюю и верхнюю границы неупорядоченной части, то получим шейкер-сортировку. Ниже приведён пример функции, использующей шейкер-сортировку элементов массива размерностью k.

```
void shaker(int *ms, int k)
{
   int i,a,b,c,d;
   c=1;        // номер стартового элемента для сортировки слева направо
   b=k−1;     // номер элемента, на котором произошла остановка
   d=k−1;     // номер стартового элемента для сортировки справа налево
   do
   {
      for(i=d;i>=c;--i) // проход по массиву справа налево
      {
         if (ms[i−1]>ms[i])
         {
            a=ms[i−1];
            ms[i−1]=ms[i];
            ms[i]=a;
            b=i;      // крайний слева упорядоченный элемент
         }
      }
      c=b+1;     // изменение левой границы массива для сортировки
      for(i=c;i<=d;++i)  // проход по массиву слева направо
      {
         if (ms[i−1]>ms[i])
         {
            a=ms[i−1];
            ms[i−1]=ms[i];
            ms[i]=a;
            b=i; // крайний справа упорядоченный элемент
         }
      }
      d=b−1;    // изменение правой границы массива для сортировки
   } while(c<=d);
}
```

9.3. Сортировка вставкой

В этом методе просмотр элементов начинается со второго элемента массива. При этом просмотр производится от текущей позиции к началу массива. Выбранный к упорядочиванию элемент массива сравнивается с элементами, лежащими левее его (эта часть массива уже упорядочена). Определяется номер первого элемента в отсортированной части массива, который больше (например, при сортировке по возрастанию) выбранного. Все элементы с найденной позиции и до позиции выбранного элемента смещаются на один элемент вправо. В освободившуюся позицию (найденного элемента) переписывается выбранный к упорядочиванию элемент массива. Переходим к следующему элементу и так до тех

пор, пока не будут просмотрены все элементы массива. Если для некоторого элемента не находится в упорядоченной части слева элемента для перестановки, то просто переходим к анализу очередного элемента правее его. Пример функции, использующей сортировку вставкой элементов массива размерностью k:

```
void vstavka(int *ms, int k)
{
   int i,j,n;
   for(i=1;i<k;++i)              // индекс элемента для упорядочивания
   {
         j=i−1;                  // индекс предыдущего элемента
         n=ms[i];                // значение предыдущего элемента
         while (j>=0 && n<ms[j])
            ms[j--+1]=ms[j];   // сдвиг всех элементов направо
         ms[j+1]=n;              // запись в освободившийся или в тот же элемент
   }
}
```

9.4. Сортировка выбором

Выбирается очередной исходный элемент массива. Последовательно проходя весь массив, сравниваем этот элемент со всеми, находящимися после него, и, при нахождении элемента, удовлетворяющего условию замены, запоминаем его индекс. После просмотра всего массива переставляем выбранный элемент с исходным. После перебора всех элементов получим отсортированный массив. Пример функции, реализующей сортировку выбором, приведен ниже.

```
void sort(int *ms, int k)
{
   int i,i1,j,m;
   for(i=0;  i<k−1; i++)       // выбор исходного элемента к сравнению
   {
      i1=i;                    // запоминаем позицию выбранного элемента
      for(j=i+1; j>i; j--)     // просмотр массива «снизу»  «вверх»
      if(ms[k]>ms[j]) i1=j;   // фиксируем координату элемента в массиве
      if(i!=i1)                // проверка, найден ли «лучший» элемент
      {
         m=ms[i];              // замена i1  и  i  элементов
         ms[i]=ms[i1];
         ms[i1]=m;
      }
   }
}
```

Приведённые выше методы сортировки не являются эффективными и могут быть использованы лишь на малых объёмах данных. В программах, где требуется высокая производительность работы сортировок, применяются методы Шелла и Хора.

9.5. Метод Шелла

Метод был предложен автором (Donald Lewis Shell) в 1959 г. Основная идея алгоритма состоит в том, что на начальном этапе реализуется сравнивание

и, если требуется, перемещение далеко отстоящих друг от друга элементов. Интервал между сравниваемыми элементами (gap) постепенно уменьшается до единицы, что приводит к перестановке соседних элементов на последних стадиях сортировки (если это необходимо).

Реализуется метод Шелла следующим образом. Начальный шаг сортировки примем равным n/2, т. е. 1/2 от общей длины массива, и после каждого прохода будем уменьшать его в два раза. Каждый этап сортировки включает в себя проход всего массива и сравнение отстоящих на gap элементов. Проход с тем же шагом повторяется, если элементы переставлялись. Заметим, что после каждого этапа отстоящие на gap элементы отсортированы.

Рассмотрим пример. Рассортировать массив чисел: 41, 53, 11, 37, 79, 19, 7, 61.

В строке после массива в круглых скобках указаны индексы сравниваемых элементов и указан номер внешнего цикла.

41 53 11 37 79 19 7 61 – исходный массив;
41 19 11 37 79 53 7 61 – (0,4), (1,5) – 1-й цикл;
41 19 7 37 79 53 11 61 – (2,6), (3,7) – 1-й цикл;
7 19 41 37 11 53 79 61 – (0,2), (1,3), (2,4), (3,5), (4,6), (5,7) – 2-й цикл;
7 19 11 37 41 53 79 61 – (0,2), (1,3), (2,4), (3,5), (4,6), (5,7) – 2-й цикл;
7 11 19 37 41 53 61 79 – сравнивались соседние – 3-й цикл.

Пример функции, реализующей рассмотренный метод сортировки, приведен ниже.

```
void sort(int *ms, int k)
{
   int  i, j, n;
   int  gap;   // шаг сортировки
   intflg;       // флаг окончания этапа сортировки
   for(gap = k/2; gap > 0; gap /= 2)
   do
   {
      flg = 0;
      for(i = 0, j = gap; j < k; i++, j++)
      if(*(ms+i) > *(ms+j))  // сравниваем отстоящие на gap элементы
      {
         n = *(ms+j);
         *(ms+j) = *(ms+i);
         *(ms+i) = n;
         flg = 1;   // есть ещё нерассортированные данные
      }
   } while (flg);    // окончание этапа сортировки
}
```

9.6. Метод Хоара

Метод Хоара (Charles Antony Richard Hoare) быстрой сортировки (Quicksort), предложенный в 1960 г., построен на основе идеи разбиения множества на подмножества. Общая процедура заключается в выборе такого элемента, который разбивает множество на два подмножества так, что в одном (слева от делящего) все элементы

меньшие, а в другом (справа от делящего) большие делящего. Этот процесс повторяется для каждого из полученных подмножеств, и так до тех пор, пока массив не будет отсортирован. Данный процесс по своей природе является рекурсивным. В качестве делящего элемента выбирается, например, срединный элемент множества.

Например, необходимо рассортировать массив: 13, 3, 23, 19, 7, 53, 29, 17. Переставляемые элементы будем подчёркивать, а срединный элемент выделим жирным шрифтом. Индекс i будет задавать номер элемента слева от срединного, а j − справа.

13 3 23 **19** 7 53 29 17
13 3 17 **19** 7 53 29 23
13 3 17 7 **19** 53 29 23
Делим на два подмножества
13 **3** 17 7 и 19 **53** 29 23
3 13 17 7 и 19 23 29 53
Правая часть отсортирована; берём подмножество из левой части
13 **17** 7
13 **7** **17**
Выделяем подмножество
13 7
7 **13**
В результате будет получено
3 7 13 17 19 23 29 53
Текст функции, реализующей метод быстрой сортировки, приведён ниже.

```
//  l – левая, r – правая границы сортируемого массива
void hoar(int  *ms, int l, int r)   // адрес массива, левая и правая границы
{                                   // упорядочиваемой его части
   int i, j, k;
   int sr = ms[(l+r)/2];       // срединный элемент
   i = l;    j = r;                 // начальные значения границ массива
   do
   {
      while(ms[i] <sr) i++;  // ищем слева элемент больше среднего
      while(ms[j] >sr)  j--;   // ищем справа элемент меньше среднего
      if(i<= j)          // если левая граница не прошла за правую,
      {
         k = ms[i];  // перестановка элементов
         ms[i] = ms[j];
         ms[j] = k;
         i++;  j--;     // переходим к следующим элементам
      }
   } while(i<= j);      // пока границы не совпали,
                        //  массив разбит на два подмножества
   if(i<r)              // если есть что-нибудь справа,
     hoar(ms,i,r);   // сортируем  правый подмассив
   if(j>l)              // если есть что-нибудь слева,
     hoar(ms,l,j);   // сортируем  левый подмассив
 }
```

10. Структуры

Как отмечалось ранее, массив предназначен для работы с данными, имеющими один тип. Рассматриваемые ниже объекты используются для снятия этого ограничения.

Структуры – это составной объект, в который входят элементы любых типов, за исключением функций. В отличие от массива, который является однородным объектом, структура может быть неоднородной. Для удобства работы с этими данными они сгруппированы под одним именем. При разработке программ структуры помогают в организации хранения и обработке сложных данных, не разобщая их по различным объектам, а группируя в одном. Кроме того, объекты, входящие в структуру, сами могут быть структурой, т. е. структуры могут быть вложенными. Структуры позволяют группу связанных между собой переменных использовать как множество отдельных элементов, а также как единое целое. Как и массив, структура представляет собой совокупность данных, но отличается от него тем, что к её элементам (компонентам) необходимо обращаться по имени и её элементы могут быть различного типа. Структуры целесообразно использовать там, где необходимо объединить разнообразные данные, относящиеся к одному объекту.

Примером использования структуры может служить, например, строка платёжной ведомости. В ней должна содержаться информация о служащем: ФИО, адрес, табельный номер, зарплата и т. д.

Как и другие объекты в С (С++) (переменные, массивы и т. д.), структуры должны быть определены. Для этого создаётся (объявляется) некоторый тип, являющийся структурой, а затем по мере необходимости определяются переменные этого типа (тип структура).

Объявление структуры осуществляется с помощью ключевого слова **struct**, за которым следует имя, называемое **тегом структуры**, и список элементов (полей структуры), заключённых в фигурные скобки. Тег даёт название структуре данного вида и в дальнейшем служит кратким обозначением той части описания структуры, которая заключена в фигурные скобки, т. е. является спецификатором. В общем виде определение структуры может быть представлено так:

```
struct [имя типа структуры]
{ тип_элемента_1 имя_элемента_1;
            . . . . . .
  тип_элемента_n имя элемента_n;
} [имя_структурной_переменной];
```

Имена структур должны быть уникальными в пределах их области видимости (действия), для того чтобы компилятор мог различать эти структуры.

Именем элемента структуры может быть любой идентификатор. Как и ранее, при описании нескольких идентификаторов одного типа в структуре они могут быть описаны через запятую. Имена элементов структуры (полей) могут совпадать с именами обычных переменных (не являющихся элементами струк-

туры), т. к. они всегда различимы по контексту. Более того, одни и те же имена элементов могут встречаться в объявлениях различных структур.

Задание типа структуры не влечёт выделения «под него» памяти компилятором. Объявление типа структуры предоставляет компилятору необходимую информацию об элементах структурной переменной для резервирования места в оперативной памяти и организации доступа к ней при определении структурной переменной и использовании отдельных элементов структурной переменной.

Определение переменной, имеющей тип структуры, аналогично определению переменных, рассмотренных ранее типов, например:

```
struct inform        // объявление типа структуры
{
   char fam[30];   // фамилия
   int god;          // год рождения
   float stag;       // стаж работы
};
inform rab;        // определение переменной rab типа struct inform
```

С синтаксической точки зрения эта запись аналогична записи вида

```
char fam[30];     // фамилия
int god;          // год рождения
float stag;       // стаж работы
```

В обоих случаях компилятор выделит под каждую переменную количество байт памяти, согласно определению этой переменной (элемента структуры).

Объявление структуры и определение структурной переменной могут быть совмещены в одной записи, например:

```
struct book          // объявление типа структуры «книга»
{
   char  title [20];    // заголовок книги
   char autor [30];   // фамилия автора
   int   page;          // число страниц
}  bk1, bk2, *ptr_bk=&bk1;   // две структурные переменные и указатель
```

В этом случае, если имя структуры **book** более нигде не используется, оно может быть опущено.

```
struct               // объявление безымянной структуры
{
   char  title [20];
   char autor [30];
   int   page;
}  bk1, bk2, *ptr_bk=&bk1;
```

10.1. Доступ к элементам структуры

Доступ к полям структуры осуществляется с помощью операции «.» (точка) при непосредственной работе со структурой и «->» (стрелка) – при использовании указателей на структуру. Возможны три вида спецификации доступа к элементам структуры:

имя_переменной_структуры . имя_поля;
имя_указателя_на_структуру -> имя_поля;
(*имя_указателя_на_структуру) . имя_поля;

Ниже приведён пример, демонстрирующий несколько способов доступа к полям структуры.

```
struct str
{
   int i;
   float j;
}  st1, st2[3],*st3,*st4[5];
```

Прямой доступ к элементу:

```
st1.i=12;       //  используется операция  «.»
st2[1].j= .52;
```

Доступ по *указателю*:

```
st3->i=12;     // используется операция  «->»
(st4+1)->j=1.23;
(*st3).j=23;   // используется операция  «*»
*(st4[2]).i=123;
*st3.j=23;      //  ошибка, операция «.» имеет больший приоритет, чем «*»
```

Можно определить указатель на элементы, имеющие тип структуры, и проинициализировать его. Например:

```
st3=&st1;
st3->i=5;      // аналогично  (*st3).i=5
```

Объявление структуры в качестве одного из полей не может содержать переменные своего типа, т. е. неверным будет следующее объявление шаблона:

```
struct  str
{
   char pole_1;
   double pole_2;
   str pole_3;                 // ошибка
};
```

Однако в объявлении структуры могут *включаться* элементы, являющиеся *указателями* на эту же структуру, например:

```
struct  str
{
   char pole_1;
   double pole_2;
   str *pole_3;                // верно
};
```

10.2. Инициализация структур

Ранее мы уже рассматривали процедуру инициализации переменных и массивов. Как и обычные переменные, элементы, входящие в состав структуры (определяющие структурную переменную), могут быть *инициализированы*. При этом можно инициализировать как все элементы, определяющие структурную переменную при её декларировании,

```
struct st
{
   char c;
   int i1,i2;
};
struct st a={'a',105,25};
```

так и некоторые из них в процессе работы с ними:

```
a.c='a';
a.i2=25;
```

При выполнении инициализации структурных переменных необходимо следовать некоторым правилам:

− присваиваемые значения должны совпадать по типу с соответствующими полями структуры;

− можно объявлять меньшее количество присваиваемых значений, чем количество полей. В этом случае остальным полям структуры будут присвоены нулевые значения;

− список инициализации последовательно присваивает значения полям структуры вложенных структур и массивов.

Например:

```
struct comp
{
   char nazv[20];     // название
   float speed;       // быстродействие
   int cena;          // цена
} cp[3]={  {"Celeron",0.6,525},
{"Duron",1.0,547},
{"Atlon",1.4,610}};
```

Так же как и массив, структура может быть введена поэлементно. Например, для описанной выше структуры **st:**

```
gets(a.c);
scanf("% d % f", &a.i1, &a.i2);
```

Как отмечалось выше, структуры могут быть вложены друг в друга:

```
struct FIO      // объявление типа структуры FIO
{
   char F[15], I[10], O[12];  // фамилия, имя, отчество
} person;
struct rabotnic      // объявление типа структуры rabotnic
{
   struct FIO fio;     // структурная переменная типа FIO
   int tab;            // табельный номер
   float zarpl;        // размер заработной платы
} rb;
```

Структура **rabotnic** содержит структуру **FIO**. Доступ к элементам структуры **FIO** и **rabotnic** осуществляется следующим образом:

```
person. F="Петров";
person.I =" ";
rb.fio.O="Иванович";
```

Единственно возможные операции над структурами – это их копирование, присваивание, взятие адреса с помощью & и осуществление доступа к её элементам (полям). Следует отметить, что оператор присваивания выполняет то, что называется *поверхностной* копией в применении к структурам-переменным. Поверхностная копия представляет собой копирование бит за битом значений полей переменной-источника в соответствующие поля переменной-приёмника. При этом может возникнуть проблема с такими полями, как *указатели*, поэтому использовать поверхностное копирование структур надо осторожно.

Копирование всех полей одной структуры в другую может быть выполнено как поэлементно, так и целиком, как это показано ниже:

```
#include <stdio.h>
struct book
{
   char avt[10];
   int izd;
} st1,st2;              // декларирование структурных переменных st1 и st2
int main()
{
   gets(st1.avt);              // ввод поля avt структуры st1
   scanf("%d",&st1.izd);  // ввод поля izd структуры st1
   st2=st1;                    // копия всех полей из st1 в st2
   return 0;
}
```

Необходимо отметить, что копирование структур (**st2=st1**) допустимо, только если **st1** и **st2** являются структурными переменными, соответствующими одному структурному типу.

Структуры нельзя сравнивать, т. е. выражение **if(st1==st2)** неверно. Сравнение структур требуется выполнять поэлементно.

Если данные, объединённые в структуры, должны содержать информацию не об одном объекте, а о некотором их множестве (например, каталог библиотеки, телефонный справочник и т. д.), то удобно использовать массивы структур.

10.3. Указатели на структуры

Как и для других структурированных типов данных, указатели могут быть использованы и для структур. Это имеет следующие положительные черты:

– указателями на структуры легче пользоваться, чем самими структурами (например в задаче сортировки);

– ряд данных удобно представлять в виде структур, полями которых являются указатели на другие структуры.

Рассмотрим пример, показывающий, как определять указатель на структуру и использовать его для работы с элементами структуры.

```
#include<stdio.h>
#include <locale.h>
#define DL 30 // max число символов в строке char
struct name {char fio[DL];}; // ФИО объекта
```

```
struct inform
{
   name nm;          // поле «ссылка на структуру name»
   char prof[DL];    // поле «профессия»
   int god;          // поле «год рождения»
   float okl;        // поле «оклад»
};
int main()
{
   setlocale(LC_ALL, "Russian");
   static inform mas[2]= { "Иванов","конструктор",1965,15000.50,
                           "Петров","оператор",1967,12350.50};
   inform *him;                    // указатель на структуру
   printf("адрес mas[0]: %u mas[1]: %u \n",&mas[0],&mas[1]);
   him=mas;                        // him содержит адрес mas[0]
   printf("адрес mas[0]: %u mas[1]: %u \n",him,him+1);
   printf("him->okl = %.2f (*him).okl = %.2f \n",him->okl, (*him).okl);
   him++;                          // him содержит адрес следующей структуры
   printf("him->okl = %.2f (*him).okl = %.2f \n",him->okl, (*him).okl);
   ++him->okl;                     // увеличение значения поля okl на 1
   printf("him->okl = %.2f (*him).okl = %.2f \n",him->okl, (*him).okl);
   return 0;
}
```

Синтаксис, используемый при описании указателя на структуру, такой же, как и для других структурированных данных:

struct имя_структуры * имя_указателя;

Таким образом, раз him – указатель на структуру inform, то *him есть сама структура, а (*him).okl – одно из полей структуры.

В выражении (*him).okl используются скобки, т. к. приоритет операции «.» выше чем «*». Для упрощения доступа к элементам структуры и избегания ошибок, связанных с учётом приоритета, была введена операция «->» (операция косвенного получения элемента). Эта операция имеет следующую форму записи:

имя_указателя -> имя_поля_структуры;

Этот оператор состоит из двух знаков: «-» и «>». Операция «->» имеет наивысший приоритет наряду с операциями «()» и «[]». Таким образом, чтобы перейти к полю okl следующей структуры, необходимо записать

(++him)->okl; или (him++)->okl;

Рассмотрим ещё один пример объявления структуры:

```
struct
{
   char * str;
   float f;
} *pr;
```

Как и выше, *pr->str есть содержимое объекта, на который ссылается str; *pr->str++ передвинет указатель pr->str после считывания объекта, на который он указывал; (*pr->str)++ увеличит значение объекта (символ), на который ссы-

лается str; *pr++->str передвинет указатель pr после получения значения, на которое указывает str.

10.4. Структуры и функции

В функцию можно наряду со стандартными типами данных передавать и структуры.

```
struct men              // объявление структуры men
{
   char *String;
   int Number;
   int Marks[10];
};
men sum(men);          //  первый прототип функции sum
void sum (men*);       //  второй прототип функции sum
```

Выше приведены два прототипа функции sum. В первом описывается функция, в которой создаётся локальная структурная переменная, в которую и производится копирование значения переменной типа структуры men. Все изменения, выполняемые над полями этой локальной структурной переменной, локальны в функции и, следовательно, чтобы были видны за пределами функции sum, должны быть возвращены.

Второй прототип описывает функцию, в которую передаётся указатель на некоторую структурную переменную. В функции преобразования выполняются над полями структуры, расположенной в памяти, по указателю, который в неё передан. Поэтому функция может ничего не возвращать. Этот вариант функции лучше, нежели первый, он будет работать быстрее и требует меньшего количества памяти.

При передаче структуры в функцию по значению она может быть передана либо вся целиком, либо по полям (компонентам).

Передача функции полей (компонентов) структуры. В общем случае элемент структуры является переменной (массивом) некоторого типа и может быть передан как аргумент функции. Рассмотрим следующую несложную программу, выполняющую суммирование окладов нескольких профессий, и нахождение max из них.

```
#include <stdio.h>
float summa(float,float *);
struct str
{
   char *prof[5];          // профессия
   float okl[5],sm;        // оклад и сумма
} ok = {"проф1","проф2","3","4","", 1.5,2.06,31.4,3.0,0,0};
int main()
{
   float max;
   int i;
   for(i=0;i<5;i++) max=summa(ok.okl[i],&ok.sm);
   printf("max значение =%5.2f\n сумма = %5.2f",max,ok.sm);
   return 0;
```

```
}
float summa(float i,float *j)
{
   static float max=0;
   *j+=i;                    // вычисление суммы
   max=(max>i)?max:i;   // поиск max значения
   return max;
}
```

Рассматриваемая программа состоит из двух функций: main() и summa(). Функции summa() в качестве параметров передаются один из суммируемых окладов (ok.okl[i]) и адрес элемента структуры, куда помещается сумма (&ok.sm). Функция summa() также возвращает вычисленное максимальное значение оклада.

Передача всей структуры в функцию. В этом случае выполняется передача данных всей структуры в функцию со всеми полями, которые в неё входят (массивы, другие структуры и т. д.). При этом никакие изменения структуры внутри функции не влияют на структуру, используемую в качестве аргумента. При использовании структуры в качестве параметра тип аргумента должен соответствовать типу параметров. Это можно сделать, определив структуру глобально, а затем использовать её для объявления необходимых структурных переменных и параметров.

```
#include <stdio.h>
float sr_ball(struct str);       // прототип функции, принимающей структуру
struct str                       // объявление структуры
{
   char prof[15];
   int oc[4];
};
int main()
{
   float bl;
   struct str  s= {"Иванов",5,4,3,5};  // инициализация структурной переменной s
   bl=sr_ball(s);          // передача структурной переменной s в функцию sr_ball
   printf("среднийбалл =%5.2f",bl);
   return 0;
}
float sr_ball(struct str st)              // описание функции sr_ball, принимающей
{                                         // структурную переменную
   return (st.oc[1]+st.oc[2]+st.oc[3]+st.oc[4])/4;
}
```

10.5. Примеры программ

Пример 1. Среди абитуриентов, сдавших вступительные экзамены в институт, определить количество абитуриентов, проживающих в городе Минске и сдавших экзамены со средним баллом не ниже восьми, распечатать их фамилии в алфавитном порядке.

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

```
#include <locale.h>
#include<windows.h>
#define NUMBER_OF_STUDENTS 3
#define MAX_SIZE 256
#define NUMBER_OF_EXAMS 3
struct student                    // объявление структуры
{
   char *town;                          // город
   char surname[MAX_SIZE];              // фамилия
   int mark[NUMBER_OF_EXAMS];   // массив оценок за экзамены
};
int main()
{
   int i, j, finalNumberOfStudents = 0;
   char c;
     /* Объявление переменных для преобразования русских символов в
        строке в другую кодировку для корректных операций со строками */
   char strTown[1024];
   char strSurname[1024];
   struct student *person;              // объявление указателя на структуру
   person = (struct student*)malloc(NUMBER_OF_STUDENTS*
              sizeof(struct student)); // выделение памяти под массив структур
   setlocale(LC_ALL, "Russian");
   printf("Введите информацию об абитуриентах");
       // цикл, проходящий по каждому элементу массива структур
   for (i = 0; i<NUMBER_OF_STUDENTS; i++)
   {
      printf("\nВведите фамилию %d-го абитуриента :", i+1);
      fflush(stdin);
      gets(person[i].surname);  // ввод фамилии
      if (!*person[i].surname)     // если фамилия не введена,
         break;                    // выход из цикла
      printf("\nВведите город абитуриента  :");
        // выделение памяти под строку
      person[i].town = (char *)malloc(MAX_SIZE * sizeof(char));
      gets(person[i].town);         // ввод города
      if (!*person[i].town)            // если город не введен,
         break;                        // выход из цикла
      printf("\nВведите оценки (физика, математика, русский язык) :");
        // ввод оценок за экзамены
      scanf("%d%d%d", person[i].mark, person[i].mark + 1, person[i].mark + 2);
   }
      /* Цикл, проходящий по каждой букве алфавита, для упорядочивания
         фамилий в алфавитном порядке */
   for (c = 'А'; c <= 'Я'; c++)
      for (j = 0; j < NUMBER_OF_STUDENTS; j++)
      {
            /* Преобразование строк с русскими символами в другую кодировку
               для удобства операций со строками */
         OemToAnsi(person[j].town, strTown);
         OemToAnsi(person[j].surname, strSurname);
```

```
            /* Проверка соответствия города и совпадения первой буквы
               фамилии с текущей буквой алфавита */
            if ((!strcmp(strTown, "Минск")) && (strSurname[0] == c))
               // подсчёт среднего балла за экзамены и его проверка (выше ли он 8)
            if ((person[j].mark[0] + person[j].mark[1] + person[j].mark[2]) / 3 >= 8)
            {
                printf("\n%s", strSurname);
                  // подсчёт числа абитуриентов, удовлетворяющих условию задачи
                finalNumberOfStudents++;
            }
          }
    printf("\nИтого абитуриентов = %d\n", finalNumberOfStudents);
    return 0;
}
```

Пример 2. Кадровые данные сотрудников (фамилия и инициалы, год рождения, должность и оклад) ведутся с помощью структур, причём фамилия и инициалы находятся во вложенной структуре. Требуется:

− вывести адреса записей структур двумя способами, учитывая, что переменная типа «структура» определена как указатель;

− перейти ко второй записи структуры, увеличить значение зарплаты на единицу и вывести двумя способами;

− вернуться к первой записи структуры, вывести фамилию двумя способами.

```
#include<stdio.h>
#include<locale.h>
#include<stdlib.h>
#define MAX_SIZE 30 // max число символов в строке char
struct Worker              // объявление структуры; содержит ФИО сотрудника
{
    char surname[MAX_SIZE]; // фамилия
    char name;                // имя
    char patronymic;          // отчество
};
struct inform // объявление структуры; содержит информацию о сотруднике
{
    Worker person;                  // структурная переменная типа Worker
    char profession[MAX_SIZE];  // профессия
    int yearOfBirth;                // год рождения
    float salary;                   // оклад
};
int main()
{
    setlocale(LC_ALL, "Russian");
      // объявление и инициализация статического массива структур
    static inform mas[2] = {"Иванов", 'И', 'И', "конструктор", 1989, 15000.50,
                            "Петров", 'П', 'П', "оператор", 1987, 12350.50};
    inform *him;     // объявление указателя на структуру  inform
    printf("адрес mas[0]: %u   mas[1]: %u \n",&mas[0],&mas[1]);
    him=mas;         // him присваивается адрес mas[0]
```

```
    printf("адрес mas[0]: %u   mas[1]: %u \n",him,him+1);
    printf("him->salary = %.2f   (*him).salary = %.2f \n", him->salary, (*him).salary);
    him++;              // him присваивается адрес следующего элемента структуры
    him->salary++; // значение переменной salary увеличивается на единицу
    printf("him->salary = %.2f   (*him).salary = %.2f \n", him->salary, (*him).salary);
    him--;              // в этом случае выполняется переход к mas[0]
    printf("him->FIO.surname = %.20s   (*him).FIO.surname = %.20s \n",
           him->person.surname, (*him).person.surname);  // затем вывод
    return 0;
}
```

10.6. Поля бит

Существует несколько разновидностей структур. Одна из них – это поля бит. Поле представляет собой последовательность соседних двоичных разрядов (бит) внутри одного целого значения. Каждое поле может иметь тип **unsigned int** или **signed int** и размещается в машинном слове целиком, а вся группа полей может выходить за пределы машинного слова. Поле бит – особый тип структуры, определяющий длину каждого её элемента. Общий вид объявления полей бит имеет вид

```
struct имя стуктуры
{ тип имя_1: длина;
     .  .  .
  тип имя_n: длина;
};
```

Наиболее распространенный подход к использованию полей можно показать на примере объявления следующей структуры:

```
struct pole
{
   int p1:1;
   unsigned  p2:2;
   int:6;
   int p3:4;
} pl;
```

Объявление такого вида обеспечивает размещение стуктуры (полей бит) в памяти следующим образом. В полях типа **signed** крайний левый бит является знаковым. Таким образом, поле **signed int** p:1 может быть использовано для хранения значений «-1» и «0», т. к. любое ненулевое значение поля p будет интерпретироваться как «-1». Поле **signed int** p:2 в отличие от **int** p:1 может принимать три значения: «-1», «0», «1». В объявлении структуры имеется ещё два поля (**int**:6; **int** p3:4). Первое указывает, что структура содержит 6 неиспользуемых бит, второе предназначено для чисел в диапазоне от минус 7 до плюс 7.

Все особенности, связанные с использованием полей, (например возможность поля перейти границу слова), зависят от аппаратной реализации.

При определенном удобстве работа с полями может повлечь некоторые трудности. Они связаны, например, с тем, что на одних машинах поля разме-

щаются слева направо, на других – справа налево. Это в свою очередь влечёт некоторые трудности при перенесении программ. Поля не могут быть массивами и не имеют адресов и, следовательно, операция & к ним не применима.

10.7. Объединения

Объединение представляет собой структурированную переменную с несколько иным способом размещения элементов в памяти. Главной их особенностью является то, что для каждого из объявленных элементов выделяется одна и та же область памяти, т. е. они перекрываются. Размерность её определяется максимальной размерностью элемента объединения. Таким образом, при использовании объединений появляется возможность работы с данными различных типов и размеров в одной и той же области памяти. Хотя доступ к этой области памяти возможен с использованием любого из элементов объединения, но элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

Формат объявления объединения имеет следующий вид:

```
union имя_объединения
{ тип имя_элемента_1;
    .  .  .
  тип имя_ элемента_n;
};
```

Спецификация доступа к элементам объединения, как и у полей бит, полностью соответствует тому, как это осуществляется у обычных структур:

имя_объединения . имя_поля_объединения;

или при работе с указателями:

имя_указателя _на_объединение -> имя_поля_объединения;

Рассмотрим пример объявления и некоторых вариантов использования объединения.

```
union un_type   // пример объявления объединения
{
   int i;
   double d;
   char c;
};
un_type un;             // переменная типа объединения
un_type un_mas[6];  // массив из шести элементов
un_type *pt;            // указатель на переменную типа un_type
```

Первое описание приводит к выделению памяти под одну переменную un. При этом компилятор выделяет достаточно памяти для размещения максимальной из переменных, описанных в объявлении un_type, т. е. 8 байт (тип **double**). Для массива un_mas[6] выделяется 48 байт памяти (6 элементов по 8 байт). Механизм использования объединения может быть рассмотрен на следующем примере.

```
un.i=14;    // 14 записывается в переменную un (используется 2 байта)
un.d=3.4;  // 14 стирается, 3.4 записывается (используется 8 байт)
```

```
un.c='k';    // 3,4 стирается, записывается символ k (используется 1 байт)
```

В каждый момент времени запоминается только одно значение; нельзя записать char и int одновременно в одно объединение, даже если для этого достаточно памяти. Приведённый ниже фрагмент демонстрирует ошибочное использование полей объединения:

```
un.d=2.4;
un.i=14;
f=5.2*un.d;    // ошибка (будет получено неверное значение)
```

Ошибка заключается в том, что в момент использования un.d его значение 2.4 «затёрто» оператором un.i=14;

Как отмечалось выше, с объединениями можно использовать операцию «->» аналогично, как это делалось со структурами.

```
pt=&un;
f=pt->d;          // аналогично f=un.d
```

Объединение может быть использовано не только для экономии памяти. Если записать в один элемент объединения некоторое значение, то через другой элемент это же значение можно прочитать уже в другой форме представления. Таким образом, форму представления данных в памяти можно менять совершенно свободно.

10.8. Переменные с изменяемой структурой

Очень часто некоторые объекты программы относятся к одному и тому же классу, отличаясь лишь некоторыми деталями. Рассмотрим, например, представление геометрических фигур. Общая информация о фигурах может включать такие элементы, как площадь, периметр. Однако соответствующая информация о геометрических размерах может оказаться различной в зависимости от их формы.

Рассмотрим пример, в котором информация о геометрических фигурах представляется на основе комбинированного использования структуры и объединения.

```
struct figure
{
   double area,perimetr;    // общие компоненты
   int type;                // признак компонента
   union                    // перечисление компонентов
   {
     double radius;         // окружность – радиус
     double a[2];           // прямоугольник – 2 строны
     double b[3];           // треугольник – 3 стороны
   } geom_fig;
}  fig1, fig2;
```

В общем случае каждый объект типа figure будет состоять из трёх компонентов: area, perimetr, type. Компонент type называется меткой активного компонента, т. к. он используется для указания компонента объединения geom_fig, который является активным в данный момент. Такая структура называется пе-

ременной структурой, потому что её компоненты меняются в зависимости от значения метки активного компонента (значение type).

Отметим, что вместо компонента type типа int, целесообразно было бы использовать перечисляемый тип. Например:

```
enum figure_chess {  CIRCLE = 0,    BOX,    TRIANGLE };
```

Константы CIRCLE, BOX, TRIANGLE получат значения соответственно равные 0, 1, 2. Переменная type может быть объявлена как имеющая перечислимый тип:

```
enum figure_chess type;
```

В этом случае компилятор C (C++) предупредит программиста о потенциально ошибочных присвоениях, таких, например, как

```
figure.type = 40;
```

В общем случае переменная структуры будет состоять из трёх частей: набора общих компонентов, метки активного компонента и части с меняющимися компонентами. Общая форма переменной структуры имеет следующий вид:

```
struct имя_структуры
{  общие компоненты;
   метка активного компонента;
   union имя_объединения
   { описание компонента_1;
                .    .    .
     описание компонента_n;
   } идентификатор_объединения;
} идентификатор_структуры;
```

10.9. Примеры программ

Пример 1. Используя поля бит, написать программу, вычисляющую остаток от деления целого числа на два и на четыре.

```
#include<stdio.h>
#include<locale.h>
#include <Windows.h>
struct Field       // объявление структуры – битовое поле
{
   unsigned int lastBit:1;      // младший (последний) бит числа
   unsigned int prelastBit:1; // следующий (предпоследний) бит числа
};
int main()
{
   struct Field *pointer; // объявление указателя на структуру Field
   int number, residue;
   setlocale(LC_ALL, "russian");
   printf("Введите число:\n");
   scanf("%d", &number);
   system("cls");
   pointer=(struct Field*)&number; // указателю pointer явно присваивается адрес
                                   // переменной number
```

```
    if(pointer->lastBit)  // если младший бит равен 1,
       residue=1;         // остаток будет равен 1
    else                  // если младший бит не равен 1,
       residue=0;         // остаток будет равен 0
    printf("\nОстаток от деления числа %d на 2: %d\n", number, residue);
    if(pointer->prelastBit && pointer->lastBit) // если последние два бита
                                                // соответственно равны 11,
      residue=3;                                // остаток будет равен трём
    else if(pointer->prelastBit&& !pointer->lastBit) // если последние два бита
                                                     // соответственно равны 10,
         residue=2;                                  //остаток будет равен двум
    else if(!pointer->prelastBit && pointer->lastBit) // если последние два бита
                                                      // соответственно равны 01,
         residue=1;                                   // остаток будет равен единице
    else              // если последние два бита соответственно равны 00,
       residue=0;  // остаток будет равен нулю
    printf("\nОстаток от деления числа %d на 4: %d\n", number, residue);
    return 0;
}
```

Пример 2. Используя поля бит и объединения, можно изменить рассмотренную выше программу, вычисляющую остаток от деления целого числа на два и четыре.

```
#include<stdio.h>
#include<locale.h>
#include<Windows.h>
  // структура - битовое поле для вычисления остатка от деления на 2
struct BitField1
{
   unsigned int lastBit: 1; // младший (последний) бит числа
};
  // структура - битовое поле для вычисления остатка от деления на 4
struct BitField2
{
   unsigned int bitCount: 2; // два младших (последних) бита числа
};
  // объединение, содержащее переменные структур, описанных выше
union Fields
{
   struct BitField1 field1;
   struct BitField2 field2;
};
int main()
{
   union Fields *pointer;
   int number, residue2, residue4;
   setlocale(LC_ALL,"russian");
   printf("Введите число:\n");
   scanf("%d",&number);
     // указателю pointer явно присваивается адрес переменной number
   pointer=(union Fields *)&number;
```

```
    system("cls");
      // присваивается значение lastBit (младшего бита числа)
    residue2=pointer->field1.lastBit;
      // присваивается значение bitCount (двух младших бит числа)
    residue4=pointer->field2.bitCount;
    printf("\nОстаток от деления на 2: %d\n",residue2);
    printf("\nОстаток от деления на 4: %d\n",residue4);
    return 0;
}
```

11. Организация списков и их обработка

Динамические переменные чаще всего реализуются как связанные структуры, списки.

Список – это набор динамических элементов (чаще всего структурных переменных), связанных между собой каким-либо способом.

Списки бывают линейными и кольцевыми, односвязными и двусвязными.

Динамические переменные, как правило, имеют тип «структура», т. к. должны содержать помимо значения (целого, вещественного и т. д.) ссылку на другую динамическую переменную. Чтобы связать динамические переменные в цепочку, надо в каждом компоненте иметь ссылку на предыдущий компонент, т. к. это не массив и различные компоненты могут быть размещены в произвольных областях памяти.

Описание структуры и указателя в этом случае может иметь вид

```
typedef struct element //  структура элемента хранения
{
   float val;      //  элемент списка
   element *n;  //  указатель на элемент хранения
} SPIS;
SPIS *p;    //  указатель текущего элемента
SPIS *sp;  //  указатель на начало списка
```

Для выделения памяти под элементы хранения необходимо пользоваться функцией **malloc(sizeof(element))** или **calloc(1,sizeof(element))**. Формирование списка осуществляется операторами:

```
sp=NULL;
while(1)
{
   puts("продолжить ?");
   if(getchar=='n') break   // ввод символа n – окончание формирования списка
   p=( SPIS*)malloc(sizeof(SPIS));  // создание нового элемента списка
   scanf("%f",&p->val);              // ввод значения в созданный элемент
   p->n=sp;                          // связывание его с предыдущим
   sp=p;                             // модификация указателя на начало списка
}
```

В последнем элементе хранения (конец списка) указатель на соседний элемент имеет значение **NULL**. Получаемый список изображён на рис. 4.

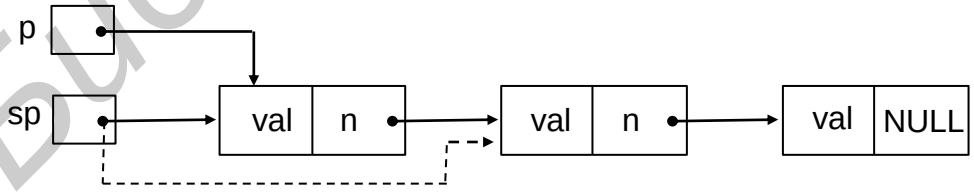


Рис. 4. Связное хранение линейного списка

При работе со списками на практике чаще всего приходится выполнять следующие операции:

- поиск элемента с заданным свойством;
- определение первого элемента в линейном списке;

− добавление нового элемента до или после указанного узла;
− исключение определённого элемента из списка;
− упорядочивание узлов линейного списка.

В подразд. 11.1 приведены фрагменты программ и схемы, демонстрирующие некоторые из операций над списками.

11.1. Операции со списками при связном хранении

Удаление элемента, следующего за узлом, на который указывает p. Исключение элемента из цепочки данных сводится к изменению одной единственной ссылки и не требует перемещения остальных элементов, как это имело бы место, например, для массивов (рис. 5).

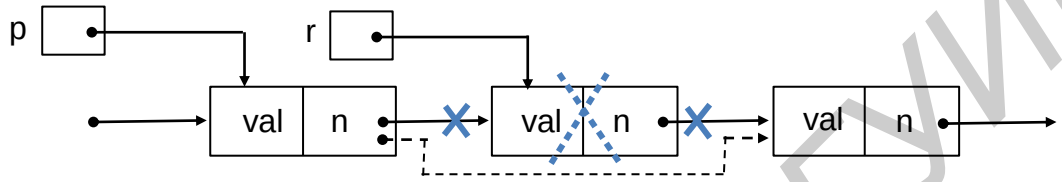


Рис. 5. Схема удаления элемента из списка

```
r=p->n;              // указатель на удаляемый элемент списка
p->n=r->n;           // исключение из списка удаляемого элемента
free(r);             // удаление элемента
```

Добавление нового элемента в цепочку данных. Для этого достаточно изменить одну ссылку. Ниже показана вставка нового узла со значением **new** за элементом, определённым указателем **p** (рис. 6):

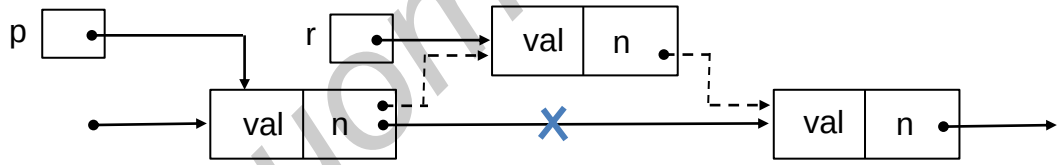


Рис. 6. Схема добавления элемента в список

```
r=( SPIS*)malloc(1,sizeof(SPIS));  // создание нового элемента
r->val=new_val;                    // инициализация нового элемента
r->n=p->n;                         // ссылка нового элемента на последующий
p->n=r;                            // ссылка предыдущего на новый элемент
```

Печать значений элементов списка. Рассмотрим печать значения элемента списка, определяемого (адресуемого) указателем **r**:

```
int i=1;          // номер выводимого элемента списка
r=sp;             // рабочий указатель на начало списка
while(r)          // пока указатель не равен NULL
{
   printf("\n элемент %d равен %f ",i++,r->val);
   r=r->n;        // переход к новому элементу списка
}
```

Частичное упорядочение списка предполагает изменение порядка следования двух элементов списка. Это решается путем изменения указателя на следующий элемент. В результате этого изменяется порядок следования их в списке. При этом упорядочиваемые элементы могут быть как соседние в списке, так и нет. Изменение указателей для первого варианта рассмотрено на рис. 7.

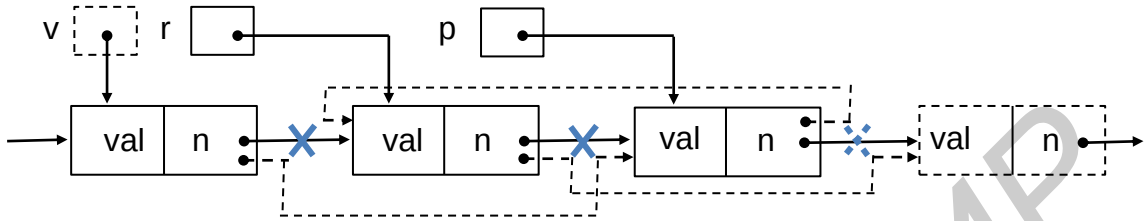


Рис. 7. Схема частичного упорядочения списка

```
SPIS *r,*p,*v;          // sp – указатель на начало списка
v=sp;                   // элемент, предшествующий анализируемым (r и p)
r=v->n;                 // предыдущий элемент списка
while( r->n!=NULL )
{
   p=r->n;              // следующий элемент списка
   if (p->val<r->val)   // значение следующего меньше, чем у предыдущего
   {
      r->n=p->n;        // замена указателей
      p->n=v->n;
      v->n=p;
   }
   r=r->n;              // переход к следующему элементу списка для сравнения
}
```

Если этот фрагмент доработать и выполнить несколько раз (как, например, в методе пузырька), то список будет упорядочен.

11.2. Стек

Стек – это конечная последовательность некоторых однотипных элементов – скалярных переменных, массивов, структур или объединений. Размещение новых элементов в стек и удаление существующих из стека производится только с одного его конца, называемого вершиной стека.

Стек предполагает добавление и удаление из него элементов. Следовательно, стек является динамической, постоянно меняющейся структурой. Согласно определению, в стеке имеется только одно место для размещения новых элементов – его вершина. Последний размещаемый элемент находится в вершине стека. Стек иногда называется списком с организацией **LIFO** – «последний размещенный извлекается первым». Если необходимо поддерживать информацию о промежуточных состояниях стека, то она должна размещаться вне стека. Внутри самого стека эта информация не поддерживается.

Допустимыми операциями над стеком являются:

− проверка стека на пустоту;
− добавление нового элемента на вершину стека;
− удаление элемента с вершины стека.

Стек можно представить как стопку книг на столе, где добавление или взятие новой книги возможно только сверху.

Ниже приводится текст программы, иллюстрирующий работу со стеком.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#define SIZE 50
struct linkedList
{
    char data[SIZE];
    struct linkedList *next;
};
void push(struct linkedList **);
void pop(struct linkedList **);
void see(struct linkedList *);
void sortAcrossContent(struct linkedList *);
void sortAcrossPointers(struct linkedList **);
   // массив указателей на функции push, pop и sortAcrossPointers
void (*f[])(struct linkedList **) = { push, pop, sortAcrossPointers };
   // массив указателей на функции see и sortAcrossContent
void (*ff[])(struct linkedList *) = { see, sortAcrossContent };
int main()
{
    struct linkedList *stack = NULL;
    char choice;    // (англ.) пункт меню
    setlocale(LC_ALL, "Rus4ian");
    while (1)
    {
        puts("Вид операции:");
        puts("  1 – создать/добавить");
        puts("  2 – удалить");
        puts("  3 – просмотреть");
        puts("  4 – сортировка 1");
        puts("  5 – сортировка 2");
        puts("  0 – окончить");
        fflush(stdin);
        choice = getchar();
        switch (choice)
        {
            case '1': (*f[0])(&stack);                 break;
            case '2': if (stack)  (*f[1])(&stack);  break;
            case '3': (*ff[0])(stack);                 break;
            case '4': if (stack)  (*ff[1])(stack);    break;
            case '5': if (stack)  (*f[2])(&stack);   break;
            case '0': return 0;
            default: printf("Ошибка, повторите ввод!\n");
        }
```

```
    }
}
// Функция создания/добавления в стек
void push(struct linkedList **stack)
{
    struct linkedList *stackElement = *stack;
    do
    {   // выделяем память под один элемент стэка
        if (!(*stack = (struct linkedList *)calloc(1, sizeof(struct linkedList))))
        {
            puts("Нет свободной памяти!");
            return;
        }
        puts("Введите данные:");
        fflush(stdin);
        gets((*stack)->data);      // ввод строки в поле data
           // новый элемент стека указывает на предыдущий
        (*stack)->next = stackElement;
        stackElement = *stack;    // элементу присваиваем новый элемент стека
        puts("Продолжить (y/n)?");
        fflush(stdin);
    } while (getchar() == 'y');
}
   // Функция просмотра элементов стека
void see(struct linkedList *stack)
{
    struct linkedList *stackElement;
    stackElement = stack;
    if (!stack)
    {
        puts("Стек пуст");
        return;
    }
    puts("Элементы стека:");
    do
    {
        printf("%s\n", stackElement->data);
        stackElement = stackElement->next;
    } while (stackElement);
    puts("Вывод стека закончен\n");
}
   // Функция удаления последнего элемента стека
void pop(struct linkedList **stack)
{
    struct linkedList *stackElement;
    stackElement = *stack;
    *stack = (*stack)->next;
    free(stackElement);
    puts("Последний элемент стека удален\n");
}
   // Функция сортировки элементов стека
```

```
   // перестановкой содержимого элементов стека
void sortAcrossContent(struct linkedList *stack)
{
    struct linkedList *min, *s2;
    for (; stack->next; stack = stack->next) // цикл выполняется, пока  != NULL
    {
        min = stack;        // элемент стека для упорядочивания
        // перебор последующих элементов справа от i-го
        for (s2 = stack->next; s2; s2 = s2->next)
            if (strcmp(min->data, s2->data) > 0) // найдено новое МИНИМАЛЬНОЕ
                min = s2;                        // значение по новому адресу
        if (min != stack)    // как обычный swap()
        {
            strcpy(s2->data, min->data);
            strcpy(min->data, stack->data);
            strcpy(stack->data, s2->data);
        }
    }
    puts("Сортировка стека выполнена\n");
}
   // Функция сортировки элементов стека
   // перестановкой адресов элементов стека
void sortAcrossPointers(struct linkedList **stack)
{
    struct linkedList *sortedStack = NULL;
    struct linkedList *min, *s3; // переменные для перемещения указателей
    struct linkedList *s2, *s4;   // пременные для передвижения по стеку
    s4 = (struct linkedList *)calloc(1, sizeof(struct linkedList));
    s4->next = *stack;              // ссылка на вершину стека
    for (; s4->next->next;)
    {
        min = s4->next;        // элемент стека для упорядочивания
           // перебор последующих элементов стека
        for (s2 = s4->next; s2->next; s2 = s2->next)
            if (strcmp(min->data, s2->next->data) > 0) // найдено новое
            {                                          // МИНИМАЛЬНОЕ значение
                min = s2->next; // адрес текущего элемента
                s3 = s2;            // адрес элемента перед минимальным
            }
        if (min != s4->next)     // если новый минимальный элемент не следующий,
        {
            s3->next = min->next; // следующим за s3 элементом становится
                                  // следующий за min (выбили из стэка)
            min->next = s4->next; // следующим за min элементом становится
                                  //следующий за s4
            s4->next = min; // минимальный элемент становится следующим
        }                         // (вернули в стэк)
        if (!sortedStack)  // это условие выполнится только в первый раз
            s2 = s4;
        s4 = s4->next; // указатель s4 переместился на следующий элемент стека
        if (!sortedStack)   // это условие выполнится только в первый раз
```

```
            {
                free(s2);
                sortedStack = min; // сохраняем указатель на ПЕРВЫЙ
            }                      // МИНИМАЛЬНЫЙ элемент в стеке
        }
        *stack = sortedStack;
        puts("Сортировка стека выполнена\n");
    }
```

11.3. Построение обратной польской записи

Одной из главных причин, лежащих в основе появления языков программирования высокого уровня, явились вычислительные задачи, требующие больших объёмов рутинных вычислений. Поэтому к языкам программирования предъявлялись требования максимального приближения формы записи вычислений к естественному языку математики. В этой связи одной из первых областей системного программирования сформировалось исследование способов трансляции выражений. Здесь получены многочисленные результаты, однако наибольшее распространение получил метод трансляции с помощью обратной польской записи, которую предложил польский математик Я. Лукашевич.

Арифметическое выражение, записанное в скобочной форме,

(A+B)*(C+D)−E,

в форме обратной польской записи будет иметь вид

AB+CD+*E−.

Характерные особенности этого выражения состоят в следовании символов операций за символами операндов и в отсутствии скобок.

Обратная польская запись обладает рядом замечательных свойств, которые превращают её в идеальный промежуточный язык при трансляции. Во-первых, вычисление выражения, записанного в обратной польской записи, может проводиться путём однократного просмотра, что является весьма удобным при генерации объектного кода программ. Во-вторых, получение обратной польской записи из исходного выражения может осуществляться весьма просто на основе простого алгоритма, предложенного Э. В. Дейкстрой. Для этого вводится понятие стекового приоритета операций (табл. 10).

Таблица 10

Операция	Приоритет
)	0
(	1
+ −	2
* /	3

Просматривается исходная строка символов слева направо, операнды переписываются в выходную строку, а знаки операций заносятся в стек на основе следующих соображений:

а) если стек пуст, то операция из исходной строки переписывается в стек;

б) операция выталкивает из стека все операции с большим или равным приоритетом в выходную строку;

в) если очередной символ из исходной строки есть открывающая скобка, то он помещается на вершину стека;

г) закрывающая круглая скобка выталкивает из стека все операции до ближайшей открывающей скобки, сами скобки в выходную строку не переписываются, а уничтожают друг друга.

Рассмотрим текст программы, выполняющей рассмотренные выше действия.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <locale.h>
// Описание элемента стека
struct stack
{
   char symbol;
   struct stack *next;
};
struct stack *push(struct stack *, char); // занесение в стек
char pop(struct stack **);                // чтение из стека
int priority(char);                       // возвращает приоритет операции
int main()
{
   struct stack *head = NULL; // стек операций пуст
   char in[80], out[80];          // входная и выходная строки
   int k, m;                  // переменные, необходимые для перемещения по
                              // строкам in и out соответственно
   setlocale(LC_ALL, "Russian");
   do
   {
      puts("Введите выражение(в конце '=') : ");
      fflush(stdin);
      gets(in);               // ввод арифметического выражения
      k = m = 0;
      while (in[k] != '\0' && in[k] != '=') // пока не дойдем до «=»
      {
         if (in[k] == ')')    // если очередной символ – «)»,
         {
            while ((head->symbol) != '(') // то удаляем из стека в
               out[m++] = pop(&head);   // выходную строку все знаки операций
                                        // до открывающей скобки
            pop(&head); // удаляем из стека открывающую скобку
         }
         if (in[k] >= 'a' &&in[k] <= 'z') // если символ – буква, то
            out[m++] = in[k];             // заносим её в выходную строку
         if (in[k] == '(')                // если очередной символ – «(», то
            head = push(head, '(');    // заносим её в стек
         if (in[k] == '+' || in[k] == '– ' || in[k] == '/' || in[k] == '*')
```

```
            { /* если следующий символ – знак операции, то все на-
                 ходящиеся в стеке операции с большим или равным
                 приоритетом переписываются в выходную строку */
              while ((head != NULL) && (priority(head->symbol) >= priority(in[k])))
                 out[m++] = pop(&head);
              head = push(head, in[k]); // запись в стек новой операции
            }
            k++;          // переход к следующему символу входной строки
         }
         while (head) // после рассмотрения всего выражения
            out[m++] = pop(&head); // перезапись операций
         out[m] = '\0';              // из стека в выходную строку
         printf("\n%s\n", out);  // и её вывод на экран
         fflush(stdin);
         puts("\nПовторить(y/n)?");
      } while (_getch() != 'n');
}
   /* Функция push записывает в стек символ a, возвращает
      указатель на новую вершину стека */
struct stack *push(struct stack *head, char a)
{
    struct stack *pointer;
    if (!(pointer = (struct stack *)malloc(sizeof(struct stack))))
    {
        puts("Нет памяти.");
        exit(-1);
    }
    pointer->symbol = a;
    pointer->next = head; // pointer – новая вершина стека
    return pointer;            // возвращаем указатель на новую вершину стека
}
 /* Функция pop удаляет символ с вершины стека. Возвращает
    удаляемый символ. Изменяет указатель на вершину стека */
char pop(struct stack **head)
{
    struct stack *pointer;
    char a;
    if (!(*head))        // если стек пуст, возвращаем «'\0'»
        return '\0';
    pointer = *head; // pointer указывает на вершину стека
    a = pointer->symbol;
    *head = pointer->next; // изменяем адрес вершины стека
    free(pointer);      // освобождаем память
    return a;           // возвращаем символ с вершины стека
}
    // Функция priority возвращает приоритет арифметической операции.
int priority(char a)
{
    switch (a)
    {
        case '*':case '/': return 3;
```

```
        case '–':case '+': return 2;
        case '(': return 1;
    }
    return 0;
}
```

11.4. Односвязный линейный список, очередь

Односвязный список (очередь) – такая упорядоченная и конечная последовательность некоторых однотипных элементов, которые могут удаляться с одного её конца (**начало** очереди), а добавляются в другой её конец (**конец** очереди).

Очередь организована по принципу **FIFO**: «первый вошёл – первый вышел». Для очереди определены следующие операции:

- проверка очереди на пустоту;
- добавление нового элемента в конец (хвост) очереди;
- удаление элемента из начала (головы) очереди;
- поиск и модификация элемента в очереди;
- упорядочивание элементов очереди.

Ниже приводится текст программы, иллюстрирующий работу с однонаправленной очередью.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <locale.h>
struct Queue
{
    char inf[50];           // информация элемента очереди
    struct Queue *next; // указатель на следующий элемент
};
void AddElementToQueue(struct Queue**, struct Queue**);
void DeleteFirstElement(struct Queue**, struct Queue**);
void DeleteAnyElement(struct Queue**, struct Queue**, char*);
void ShowQueue(struct Queue*);
void SortQueue(struct Queue*);
int main()
{
    setlocale(LC_ALL, "Russian");
    struct Queue *head, *tail; // указатели на голову и хвост очереди
    char *str;
    str = (char *)malloc(10);
    head = tail = NULL;         // устанавливаем указатели на голову и хвост на 0
    while (1)                   // бесконечный цикл
    {
        puts("\nвид операции: 1 – создать очередь");
        puts("              2 – вывод содержимого очереди");
        puts("              3 – удаление элемента с головы очереди");
        puts("              4 – удаление любого элемента из очереди");
        puts("              5 – сортировка очереди");
        puts("              0 – окончить");
```

```
        fflush(stdin);
        switch (_getch())
        {
           case '1': AddElementToQueue(&tail, &head); // добавление в хвост
                     system("CLS");
                     break;
           case '2': ShowQueue(tail);        // вывод очереди на экран
                     system("CLS");
                     break;
           case '3':
              if (head)     // если очередь уже создана
                 DeleteFirstElement(&tail, &head); // удаление с головы очереди
              system("CLS");
              break;
           case '4':
              if (head)     // если очередь уже создана
              {
                 fflush(stdin);
                 puts("Введите информацию, которую нужно удалить ");
                 gets(str);
                 DeleteAnyElement(&tail, &head, str); // удаление с любого
              }                                       // места в очереди
              system("CLS");
              break;
           case '5':
              if (head)        // если очередь уже создана
                   SortQueue(tail); // сортировка очереди
              system("CLS");
              break;
           case '0': return; // завершение работы программы
           default:
              printf("Ошибка, повторите \n");
              system("PAUSE");
              system("CLS");
        }
     }
     return 0;
}
     // Функция создания очереди
void AddElementToQueue(struct Queue **tail, struct Queue **head)
{   // временная переменная для добавления нового элемента в очередь
     struct Queue *temp;
     puts("Создание очереди \n");
     do
     {   // выделение памяти под temp; если память не выделилась,
        if (!(temp = (struct Queue *)calloc(1, sizeof(struct Queue))))
        {
           puts("Ошибка выделения памяти");
           system("PAUSE");
           exit(0);   // то завершение работы программы
        }
```

```
        puts("Введите информацию ");
        scanf("%s", temp->inf);
        if (!*tail)    // если очередь ещё не создана,
           *tail = *head = temp; // то устанавливаем оба указателя (голова и хвост)
                                 // на единственный элемент очереди
        Else        // иначе, если очередь уже существует,
        {
           temp->next = *tail; // добавляем элемент в конец очереди
           *tail = temp;          // устанавливаем указатель на конец очереди
        }                         // на последний элемент
        puts("Продолжить (y/n): ");
        fflush(stdin);
    } while (_getch() == 'y');
    puts("");
    system("PAUSE");
}
    // Функция вывода содержимого очереди, начиная с хвоста
void ShowQueue(struct Queue *tail)
{
    puts("Содержимое очереди:\n");
    if (!tail)       // если очередь не создана
    {
        puts("Очередь пуста");
        system("PAUSE");
        return;    // выход их функции
    }
    do
    {
        printf("\n %s", tail->inf); // выводим на экран текущий элемент очереди
        tail = tail->next; // передвигаем указатель temp на следующий элемент
    } while (tail); // пока указатель tail не дошёл до головы очереди
    puts("");
    system("PAUSE");
}
    // Функция удаления элемента с головы очереди
void DeleteFirstElement(struct Queue **tail, struct Queue **head)
{
    struct Queue *tempTail = *tail; // временная переменная
    if (*tail == *head) // если в очереди только один элемент
    {
        free(*head);     // освобождаем выделенную под head память
        *tail = *head = NULL; // «зануляем» указатели
        return;            // выходим из функции
    }
    while (tempTail->next != *head) // перемещаем указатель tempTail
        tempTail = tempTail->next;    // на второй элемент очереди
    free(*head);         // удаляем первый элемент очереди
    *head = tempTail; // присваиваем указателю на голову адрес второго элемента
    (*head)->next = NULL;   // «зануляем» указатель на следующий элемент
    puts("Элемент удалён.\n");
    system("PAUSE");
```

```
}
   // Функция удаления любого элемента очереди
void DeleteAnyElement(struct Queue **tail, struct Queue **head, char *str)
{
     // указатели на текущий анализируемый элемент и на предыдущий
   struct Queue *current, *previous;
     // если в очереди только один элемент и он совпадает с элементом,
     // выбранным для удаления
   if (*tail == *head && (!strcmp((*head)->inf, str) || *str == '\0'))
   {
      free(*tail);  // освобождаем память под единственный элемент очереди
      *tail = *head = NULL;     // «зануляем» указатели на хвост и голову
      puts("Элемент удален.\n");
      system("PAUSE");
      return;       // выход их функции
   }
      // удаление всех элементов str, стоящих подряд в хвосте очереди
   while (*tail && !strcmp((*tail)->inf, str))
   {
      current = *tail;        // сохраняем адрес удаляемого элемента
      *tail = (*tail)->next; // перемещение указателя на голову очереди
                             // на следующий элемент
      free(current);         // удаляем элемент
   }
   current = (*tail)->next; // следующий за элементом хвоста очереди
   previous = *tail;          // элемент перед ним
   while (current)            // проходим по очереди с хвоста до головы
   {
      if (!strcmp(current->inf, str)) // удаление всех элементов str с головы очереди
      {   // исключение из очереди удаляемого элемента
         previous->next = current->next;
         free(current);       // удаление из очереди
         current = previous->next; // переход к анализу следующего элемента
      }
      else
      {
         current = current->next;   // переход к анализу следующего элемента
         previous = previous->next; // элемент перед текущим
      }
   }
   current = *tail;
   while (current->next)          // проход по очереди
      current = current->next; // установка указателя на головной элемент
   *head = current;               // коррекция адреса головы очереди
   puts("Элемент(ы) удален(ы).\n");
   system("PAUSE");
}
   // Функция сортировки содержимого очереди пузырьком
void SortQueue(struct Queue *tail)
{
   system("cls");
```

```
    char swapStr[30];                // переменная для сортировки двух строк
    struct Queue *temp = NULL; // указатель для прохода по очереди
    // проходим по очереди с хвоста до головы
    for (; tail; tail = tail->next)
    {
       // проходим по очереди от указателя tail до головы
       for (temp = tail->next; temp; temp = temp->next)
       {
          // если выполняется условие, то меняем строки местами
          if (strcmp(tail->inf, tail->next->inf) > 0)
          {
             strcpy(swapStr, tail->inf);
             strcpy(tail->inf, tail->next->inf);
             strcpy(tail->next->inf, swapStr);
          }
       }
    }
    puts("Очередь отсортирована.\n");
    system("PAUSE");
}
```

11.5. Двусвязный линейный список

Связанное хранение линейного списка называется списком с двумя связями или двусвязным списком, если каждый элемент хранения имеет два указателя (на предыдущий и последующий элементы линейного списка).

В программе двусвязный список можно реализовать с помощью описаний:

```
typedef struct ndd
{
   float val;   //  значение элемента
   ndd * n;    //  указатель на следующий элемент
   ndd * m;   //  указатель на предыдующий элемент
} NDD;
NDD  *dl, *p, *r;
```

Графическая интерпретация списка с двумя связями приведена на рис. 8.

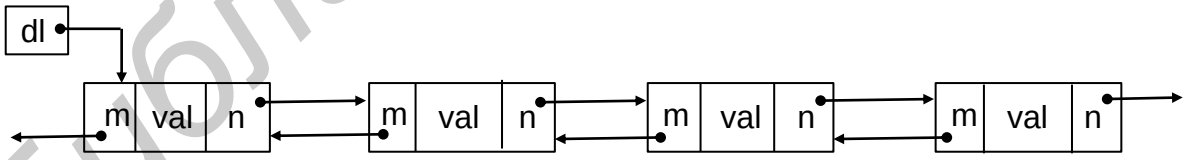


Рис. 8. Схема хранения двусвязного списка

Вставка нового узла со значением new за элементом, определяемым указателем p, осуществляется при помощи операторов:

```
r=(NDD*)malloc(sizeof(NDD));
r->val=new;
r->n=p->n;
(p->n)->m=r;
p->=r;
```

Удаление элемента, следующего за узлом, на который указывает p:

```
p->n=r;
p->n=(p->n)->n;
((p->n)->n)->m=p;
free(r);
```

Ниже приводится текст программы, иллюстрирующий работу с двунаправленным списком.

Пример. Написать программу, содержащую функции работы с двусвязным списком: создание двусвязного списка, вставка элементов, удаление элементов списка, вывод списка, сортировка списка.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <locale.h>
struct note       // структура – элемент двусвязного списка
{
   char information[50]; // объявление строковой переменной
   note *previous;       // объявление указателей типа note
   note *next;           // для двусторонней связи
};
void add(note **, note **); // прототипы функций: создания списка
void ins(note **, char *);    // вставки элемента в список
void del_any(note **, note **, char *); // удаления элемента
void see(note *, int);        // просмотра списка
void sort(note **, note **); // сортировки списка
int main()
{
   note *tale, *head; // объявление указателей на хвост и голову списка
   char l, *string;
   string = (char *)malloc(10); // выделение памяти на строку
   tale = head = NULL; // зануление указателей на хвост и голову списка
   system("CLS");
   setlocale(LC_ALL, "Russian");
   while (1)
   {
      puts("вид операции:     1 – создать очередь");
      puts("                  2 – вывод содержимого очереди");
      puts("                  3 – вставка нового элемента в очередь");
      puts("                  4 – удаление любого элемента из очереди");
      puts("                  5 – сортировка очереди");
      puts("                  0 – окончить");
      fflush(stdin);
      switch (getch()) // выбор операции в соответствии с введенной цифрой
      {
            case '1': add(&tale, &head); break; // вызов функции создания списка
            case '2': system("CLS");
               puts("0-просмотр с хвоста\n1– просмотр с головы");
               l = getch();
               if (l == '0')
```

```
                see(tale, 0);   // вызов функции просмотра списка (с хвоста)
              else
                see(head, 1); // вызов функции просмотра списка (с головы)
            break;
            case '3': if (tale)
            {
              puts("\nВведите вставляемый элемент");
              gets(string);
              ins(&tale, string); // вызов фунцкции вставки элемента в список
            } break;
            case '4': if (tale)
            {
              puts("\nВведите элемент, который вы хотите удалить");
              gets(string);
              del_any(&tale, &head, string);
            } break;
            case '5': if (tale)
                        sort(&tale, &head); // вызов функции сортировки списка
                      break;
            case '0': return 0; // выход из программы
            default: printf("Ошибка, повторите \n");
        }
    }
    return 0;
}
   /* Функция создания очереди  и добавления (только) в хвост списка*/
void add(note **pointerTail, note **pointerHead)
{
    note *n;
    puts("Создание очереди \n");
    do
    {
        if (!(n = (note *)calloc(1, sizeof(note))))
        { // выход при неудачном выделении памяти
            puts("Нет свободной памяти");
            return;
        }
        puts("Введите элемент очереди");
        scanf("%s", n->information);
        if (!*pointerTail || !*pointerHead) // очередь ещё не создана
        {
            *pointerTail = n;    // указатель на хвост списка
            *pointerHead = n; // указатель на голову списка
        }
        else
        {
            n->next = *pointerTail;          // указатель на элемент (хвост) списка
            (*pointerTail)->previous = n; // хвост указывает на новый элемент
            *pointerTail = n; // передвигаем указатель хвоста на новый элемент
        }
        puts("Продолжить (y/n): ");
```

```
        fflush(stdin);
    } while (getch() == 'y');
}
   /* Функция вывода содержимого списка; вывод начинать с хвоста (i==0) или
      головы (i==1)*/
void see(note *p, int i)
{
    puts("Вывод содержимого очереди \n");
    if (!p)
        {
            puts("Очередь пуста");
            return;
        }
    do
    {
        printf("%s\n", p->information);
        if (!i)
            p = p->next;        // если вывод с головы списка
        else
            p = p->previous; // если вывод с хвоста списка
    } while (p);
    return;
}
   /* Функция добавления элемента в очередь  (перед хвостом очереди)
      добавление работает правильно, только если очередь упорядочена
      с хвоста по возрастанию  [ (хвост)A C D E(голова)  вставить B ] */
void ins(note **pointerTail1, char *string1)
{
    note *p = *pointerTail1, *n;
     // если существует p и среди списка не существует элемента string1
    while (p && strcmp(p->information, string1)<0)
    p = p->next;    // переход к следующему элементу
    if (!(n = (note *)calloc(1, sizeof(note))))
    {
        puts("Нет свободной памяти");
        return;
    }
    strcpy(n->information, string1);    // копирует s n->information
    /* Теперь поле next предыдущего элемента списка указывает на
       вставляемый элемент n*/
    p->previous->next = n;
    // Поле next элемента n указывает на следующий  элемент списка p
    n->next = p;
     /* Поле previous вставляемого элемента списка n указывает на
       предыдущий элемент списка p */
    n->previous = p->previous;
     /* Поле previous последующего элемента списка p указывает на
       вставляемый элемент n */
    p->previous = n;
}
    /* Функция удаления любого одного элемента очереди
```

```
pTail – указатель на хвост  pHead – на голову очереди */
void del_any(note **pTail, note **pHead, char *string)
{
   note *pointerTail = *pTail;
   if (!*pTail || !pHead)
   {
      puts("Очередь пуста");
      return;
   }
   /* В очереди только один элемент и он не равен введённой строке или
      строки не существует */
if (pointerTail->next == NULL &&
   (!strcmp(pointerTail->information, string) || *string == '\0'))
   {
      free(pointerTail);
      *pTail = *pHead = NULL;    // опустошаем список
      return;
   }
   while (pointerTail && strcmp(pointerTail->information, string))
      pointerTail = pointerTail->next;
    // найден элемент, совпадающий с введённой строкой string
   if (!strcmp(pointerTail->information, string))
   {
      if (pointerTail == *pTail)    // удаляемая вершина – хвост очереди
      {
         *pTail = (*pTail)->next; // новый указатель переходит на хвост очереди
         (*pTail)->previous = NULL;
      }
      else if (pointerTail == *pHead)     // удаляемая вершина – голова очереди
      {
         *pHead = (*pHead)->previous; // новый указатель на голову очереди
         (*pHead)->next = NULL;
      }
      else
      {   // обходим элемент  previous
         pointerTail->previous->next = pointerTail->next;
         // обходим элемент next
         pointerTail->next->previous = pointerTail->previous;
      }
      free(pointerTail); // удаляем элемент  previous  очереди
   }
}
   // Функция сортировки содержимого очереди
void sort(note **pointerTail, note **pointerHead)
{
   note *s1, *s2, *s3;
       // цикл, просматривающий все элементы очереди, начиная с первого
   for (s2 = *pointerHead; s2; s2 = s2->previous)
           // цикл, просматривающий все элементы очереди, кроме первого
      for (s1 = *pointerTail; s1->next; s1 = s1->next)
      {
```

```
            // если следующий элемент меньше данного (s1)
            if (strcmp(s1->next->information, s1->information)>0)
            {
               s3 = s1->next;  // указатель s3 устанавливается на больший элемент,
               if (!s1->previous) // если s1 последний элемент очереди
                  *pointerTail = s1->next;
               // s1-next указывает через вершину (через элемент)
               s1->next = s1->next->next;
               if (s1->next)                    // если это не первый элемент очереди,
                  s1->next->previous = s1; // создаём двустороннюю связь элементов
               else // если первый
                  s2 = *pointerHead = s1;  // указатель на голову устанавливаем на s1
               s3->previous = s1->previous; // вставка элемента s1
               s3->next = s1;
               if (s3->previous)     // если не последний элемент очереди
                  s3->previous->next = s3;
               s1->previous = s3;
               s1 = s1->previous; // откат для s1=s1->next в цикле for
            }
         }
      puts("Сортировка выполнена");
}
```

11.6. Одно- и двусвязный циклический список, кольцо

Связанное хранение линейного списка называется циклическим списком, или кольцом, если его последний элемент указывает на первый элемент списка, а указатель на список – на последний элемент списка (рис. 9).

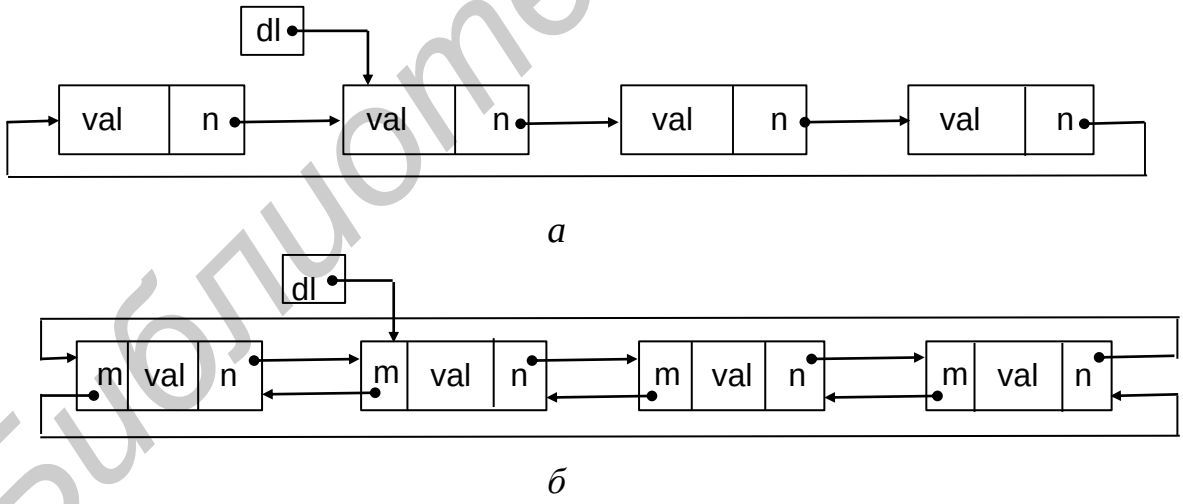


Рис. 9. Схема циклического хранения списка:
а – односвязного; *б* – двусвязного

Как и в случае линейного списка, элементы циклического списка могут иметь несколько ссылок. Текст программы, иллюстрирующий работу с двунаправленным циклическим списком, приведен ниже.

Пример. Написать программу, содержащую функции работы с двусвязным кольцом: создания/добавления элементов в кольцо, удаление элементов кольца, вывод кольца, сортировка кольца (двумя способами: через указатели и через информацию).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <windows.h>
#include <locale.h>
struct Ring          // структура – узел двусвязного кольца
{
   char inf[50];     // объявление информационной строковой переменной
   struct Ring *nextLeft, *nextRight;   // указатели на предыдущий и последующий
                                        // элементы циклического списка
};
struct Ring* AddNode(struct Ring *head);
void DisplayRing(struct Ring *head);
struct Ring* DeleleNode(struct Ring *head);
void SortInformation(struct Ring *head);
struct Ring *SortPointers(struct Ring *head);
struct Ring * direction(char c,struct Ring *node);
int main()
{    // объявление указателя на структуру Ring и его зануление
   struct Ring *head=NULL;
   setlocale(LC_ALL,"russian");
   while(1)
   {
      puts("Выберите операцию:");
      puts("1 – Создать кольцо");
      puts("2 – Вывод содержимого кольца");
      puts("3 – Удаление элемента из кольца");
      puts("4 – Сортировка (изменяя указатели на элементы)");
      puts("5 – Сортировка (изменяя содержимое элементов)");
      puts("0 – Выход");
      fflush(stdin);
      switch(getch()) // выбор операции в соответствии с введённой цифрой
      {
         case '1':
            head=AddNode(head);
            break;
        case '2':
            if(head)
               DisplayRing(head);
            else
               puts("Кольцо пустое");
            getch();
            break;
         case '3':
            if(head)
```

```
                head=DeleleNode(head);
             break;
          case '4':
             if(head)
                head=SortPointers(head);
             break;
          case '5':
              if(head)
                 SortInformation(head);
              break;
          case '0':
             return;
       }
       system("CLS");
    }
   return 0;
}
    // Функция создания/добавления в кольцо
struct Ring* AddNode(struct Ring *head)
{  // добавление выполняется вправо от элемента входа в кольцо
    struct Ring *tail,*temp; // объявление указателей на структуру Ring
    if (!head) // кольца ещё не существует (указатель на кольцо равен NULL)
    {
       head=(struct Ring *) malloc(sizeof(struct Ring));
       if(!head) // если память не выделена
       {
          puts("Нет свободной памяти");
          return NULL;
       }
       puts("Введите информацию в inf");
       scanf("%s",head->inf);   // ввод информации в первый элемент кольца
       head->nextLeft=head;    // указатели nextLeft и nextRingt
       head->nextRight=head;  // указывают на первый элемент
       tail=head; // указателю tail присваивается указатель на первый элемент
    }
    else // кольцо уже существует
       tail=head->nextRight;
       /* указателю tail присваивается последний элемент кольца
          (head->nextRight указывает на последний элемент) */
    do
    {
       temp=(struct Ring *) calloc(1,sizeof(struct Ring));
       if(temp==NULL) // если память не выделена
       {
          puts("Нет свободной памяти");
          return NULL;
       }
       puts("Введите информацию в inf");
       scanf("%s",temp->inf); // ввод информации в новый элемент кольца
       tail->nextLeft=temp;     // указателю nextLeft последнего элемента tail
                                // присваивается указатель на новый элемент temp
```

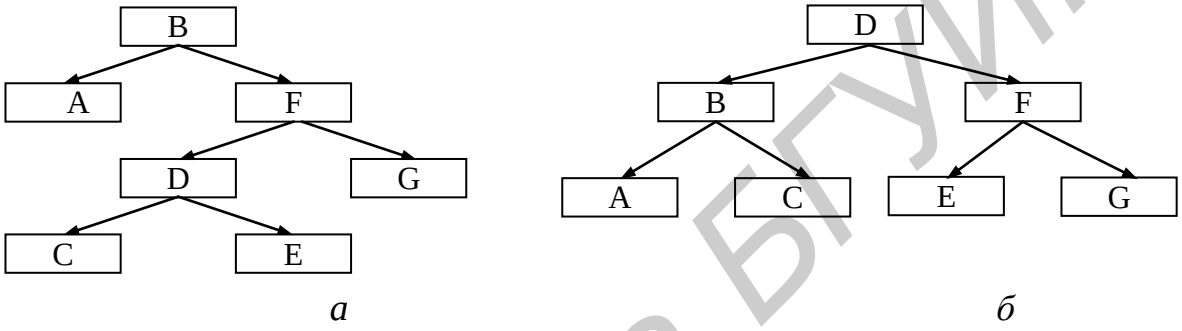
```
        temp->nextRight =tail;  // указателю nextRingt нового элемента присваива-
                                // ется указатель на последний элемент tail
         tail=temp;             // новый элемент становится последним
        puts("\nЧтобы продолжить, введите 'y':\n ");
        fflush(stdin);
     } while(getch()=='y'); // цикл продолжается, пока вводится 'y'
    tail->nextLeft=head;  // указателю nextLeft последнего элемента присваива-
                          // ется указатель на первый элемент (список замыкается)
    head->nextRight=tail;  // указателю nextRight первого элемента
                          // присваивается указатель на последний элемент (список
                           // замыкается в обратную сторону)
    return head;   // возвращается указатель на первый элемент кольца
}
    // Функция вывода содержимого кольца
void DisplayRing(struct Ring *head)
{
    struct Ring *temp; // объявление указателя на структуру Ring
    char sym;
    temp=head; // указателю temp присваивается указатель на голову кольца
    puts("r – по часовой, l – против часовой\n");
    fflush(stdin);
    sym=getch(); // ввод направления вывода
    if(sym=='l' || sym=='L' || sym=='r' || sym=='R')
    do
    {
         printf("%s\n",temp->inf); // вывод информации данного элемента
         temp=direction(sym,temp); /* указателю temp, в соответствии с
                                      выбранным направлением,  присваивается
                                      указатель на следующий элемент кольца */
     } while(temp!=head);         // пока не будут просмотрены все элементы
    puts("Вывод кольца закончен");
    return;
}
    // Функция удаления элемента кольца
struct Ring* DeleleNode(struct Ring *head)
{
    struct Ring *temp; // объявление указателя на структуру Ring
    char str[50];
    temp=head; // указателю temp присваивается указатель на голову кольца
    puts("Введите информацию в inf");
    scanf("%s", &str);
    do
    {
        if(strcmp(temp->inf,str)) // если строки не равны,
            temp=temp->nextLeft; // переход к следующему элементу кольца
        else // если равны,
        {
            if (temp->nextRight==temp) // если удаляемый элемент в кольце один,
            {
                free(temp);   // то данный элемент удаляется
                return NULL;
```

```
            }
            if(head==temp) // если удаляемый элемент – первый элемент кольца,
               head=head->nextLeft; // то указатель на первый (удаляемый) элемент
                                    // передвигается на второй (второй элемент
                                    // становится началом кольца)
            temp->nextLeft->nextRight=temp->nextRight;
            temp->nextRight->nextLeft=temp->nextLeft
            free(temp);   // удаление данного элемента
            return head; // возвращается указатель на первый элемент кольца,
         }
      } while(temp!=head); // пока не будут просмотрены все элементы
      printf("Записи с информацией  '%s'  в кольце нет \n", str);
      getch();
      return head; // возвращается указатель на первый элемент кольца
}
      // Функция сортировки методом отбора (изменяя содержимое элементов)
void SortInformation(struct Ring *head)
{
      struct Ring *unsortedElement,*temp;
      char str[50], sym;
      puts("направление r – по часовой, l – против часовой\n");
      fflush(stdin);
      sym=getch(); // ввод направления сортировки
      unsortedElement=head;
      do
      {   // в str записывается строковая информация для замены
         strcpy(str,unsortedElement->inf);
         temp=unsortedElement; // temp указывает на элемент для сортировки
         do
         {
            temp=direction(sym,temp);
            if(strcmp(str,temp->inf)>0) // если строка для замены больше строки
            {                           // следующего элемента кольца,
               strcpy(unsortedElement->inf,temp->inf); // то меняем местами строки
               strcpy(temp->inf,str); // в  элементах
               strcpy(str,unsortedElement->inf);
            }
         } while(direction(sym,temp)!=head); // пока не будут просмотрены все
                                             // элементы списка
         unsortedElement=direction(sym,unsortedElement);
      } while(direction(sym,unsortedElement)!=head);
        // пока не будут просмотрены все элементы для сортировки
}
      // Функция сортировки методом отбора (изменяя указатели на элементы)
struct Ring *SortPointers(struct Ring *head)
{   // рассмотрен случай сортировки вправо (по возрастанию)
      struct Ring *temp,*unsortedElement,*minElement;
      int i;
      unsortedElement = head;
      do
      {     // temp указывает на второй элемент кольца
```

```
        temp = unsortedElement->nextRight;
        minElement = unsortedElement;
        do      // цикл отбора элемента с наименьшим значением
        {
           if(strcmp(minElement->inf,temp->inf)>0)
              minElement = temp;
           temp = temp->nextRight; // переход к следующему неотсортированному
                                   // элементу списка
        } while(temp != head);         // пока не будут просмотрены все элементы
        if(minElement != unsortedElement)   // если минимальный элемент не
        {                                   // является первым неотсортированным
              if(head == unsortedElement) // если первый элемент кольца является
                                          // неотсортированным,
                 head = minElement;          // то минимальный элемент – новая точка
                                          // входа в кольцо
              /* исключение указателей nextRight и nextLetf минимального
                 элемента (т. е. исключение всего элемента из кольца) */
           minElement->nextLeft->nextRight = minElement->nextRight;
           minElement->nextRight->nextLeft = minElement->nextLeft;
              /* перемещение указателя первого неотсортированного элемента
                 на минимальный */
           unsortedElement->nextLeft->nextRight = minElement;
           minElement->nextRight = unsortedElement; // вставка минимального эле-
                                 // мента перед первым неотсортированным элементом
           minElement->nextLeft = unsortedElement->nextLeft;
           unsortedElement->nextLeft = minElement;
        }
        else   // если минимальный элемент является первым неотсортированным
           unsortedElement = unsortedElement->nextRight;
     } while(unsortedElement->nextRight != head);   // пока не будут просмотрены
                                                    // все неотсортированные элементы
     return head;    // возвращается указатель на первый элемент кольца
}
     // Функция выбора направления по введённому символу
struct Ring * direction(char c,struct Ring *node)
{
     switch(c)
     {
        case 'r':  // если символ 'r' или 'R'
        case 'R': // возвращает указатель на следующий элемент
          return node->nextRight; // для прохождения кольца по часовой стрелке
        case 'l':  // если символ 'l' или 'L'
        case 'L': // возвращается указатель на следующий элемент
          return node->nextLeft; // для прохождения кольца против часовой стрелки
     }
}
```

11.7. Деревья

Дерево − непустое конечное множество элементов, один из которых называется **корнем**, а остальные делятся на несколько непересекающихся подмножеств, каждое из которых также является деревом. Одними из разновидностей деревьев являются **бинарные деревья**. Бинарное дерево имеет один корень и два непересекающихся подмножества, каждое из которых само является бинарным деревом. Эти подмножества называются **левым** и **правым поддеревьями** исходного дерева. Ниже (рис. 10) приведены графические изображения бинарных деревьев. Если один или более узлов дерева имеют более двух ссылок, то такое дерево не является бинарным.



а *б*

Рис. 10. Структура бинарного дерева

На рис. 10, *а* изображено дерево, у которого B – корень дерева, A (F) – корень левого (правого) поддерева. При этом B – родительский узел, A (F) – левый (правый) дочерний узел. Узел, не имеющий дочерних узлов A, C, E, G – **лист дерева**. Если каждый узел бинарного дерева, не являющийся листом, имеет непустые правые и левые поддеревья, то дерево называется **строго бинарным деревом** (рис. 10, *б*). Строго бинарное дерево с n листами всегда содержит 2n-1 узлов.

Уровень узла в бинарном дереве может быть определён следующим образом. Корень дерева имеет **уровень** 0, уровень любого другого узла дерева на 1 больше уровня его родительского узла. **Глубина** бинарного дерева − максимальный уровень листа дерева (длина максимального пути от корня к листу). **Полное бинарное дерево уровня n** – дерево, в котором каждый узел уровня n является листом, и каждый узел уровня меньше n имеет непустые левое и правое поддеревья. **Почти полное бинарное дерево** − бинарное дерево, для которого существует неотрицательное целое k такое, что выполняются следующие условия:

− каждый лист в дереве имеет уровень k или k+1;

− если узел дерева имеет правого потомка уровня k+1, то все его левые потомки, также листья, имеют уровень k+1.

На любом уровне k бинарное дерево может содержать от 1 до 2k узлов. Число узлов, приходящееся на уровень, является показателем плотности дерева. На рис. 10, *а* дерево содержит 7 узлов при глубине 4, в то время как дерево,

изображенное на рис. 10, *б*, при том же количестве узлов, имеет глубину 3. Дерево может быть также вырожденным, если у него есть единственный лист и каждый нелистовой узел имеет только одного сына. Вырожденное дерево эквивалентно связанному списку.

Деревья с большой плотностью очень важны в качестве структур данных, т. к. они содержат пропорционально больше элементов вблизи корня, т. е. с более короткими путями от корня. Плотное дерево позволяет хранить большее количество данных и осуществлять эффективный доступ к его элементам. Быстрый поиск – главное, что обусловливает использование деревьев для хранения данных.

Бинарные деревья с успехом могут быть использованы, например, при сравнении и организации хранения очередной порции входной информации с информацией, введённой ранее, и при этом в каждой точке сравнения может быть принято одно из двух возможных решений. Так как информация вводится в произвольном порядке, то нет возможности предварительно упорядочить её и применить бинарный поиск. При использовании линейного поиска время пропорционально квадрату количества анализируемых слов. Каким же образом, не затрачивая большое количество времени, организовать эффективное хранение и обработку входной информации? Один из способов постоянно поддерживать упорядоченность имеющейся информации − это перестановка её при каждом новом вводе информации, что требует существенных временных затрат. Построение бинарного дерева осуществляется на основе *лексикографического упорядочивания* входной информации. Различным элементам входной информации ставятся в соответствие различные узлы бинарного дерева, каждый из которых является структурой, содержащей:

− информационную часть;
− указатель на левый сыновний узел;
− указатель на правый сыновний узел.

Лексикографическое упорядочивание информации в бинарном дереве заключается в следующем. Считывается первая информация и помещается в узел, который становится корнем бинарного дерева с пустыми левым и правым поддеревьями. Затем каждая вводимая порция информации сравнивается с информацией, содержащейся в корне. Если значения совпадают, то, например, наращиваем число повторов и переходим к новому вводу информации. Если же введённая информация меньше значения в корне, то процесс повторяется для левого поддерева, иначе для правого. Так продолжается до тех пор, пока не встретится дубликат или пока не будет достигнуто пустое поддерево. В этом случае число помещается в новый узел данного места дерева.

Процесс поочередного доступа к каждой вершине дерева называется *обходом (вершин) дерева*. Существует три способа обхода дерева:

− обход симметричным способом (симметричный обход);
− обход в прямом порядке (прямой обход, упорядоченный обход, обход сверху или обход в ширину);
− обход в обратном порядке (обход в глубину, обратный обход, обход снизу).

При симметричном обходе обрабатывается сначала левое поддерево, затем корень, после чего правое поддерево. При прямом обходе обрабатывается сначала корень, затем левое поддерево, а потом правое. При обходе снизу сначала обрабатывается левое поддерево, затем правое и наконец корень. Последовательность доступа при каждом методе обхода показана ниже на примере дерева, изображенного на рис 10, *б*.

Симметричный обход **A B C D E F G**
Прямой обход **D B A C F E G**
Обход снизу **A C B E G F D**

Ниже приведён текст программы, иллюстрирующий работу с бинарным деревом.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <Windows.h>
#define MAX_LEN 20
typedef struct tree
{
   char *info; // информационное поле
   int repeats; // число встреч информационного поля в бинарном дереве
   struct tree *left; // указатель на левое поддерево
   struct tree *right; // указатель на правое поддерево
} TREE;
void showTreeRecursive(TREE *);   // функция рекурсивного вывода дерева
void showTreeUsingStack(TREE *); // функция вывода дерева с помощью стека
void showTree(TREE *, int level); // функция вывода дерева в структурном виде
TREE *createTree(TREE *);         // функция создания дерева
TREE *createTreeRecursive(TREE *, char *); // рекурсивная функция создания
                                           // дерева
TREE *deleteNode(TREE *);       // функция удаления узла дерева
int main()
{
   TREE *root = NULL; // адрес корня бинарного дерева
   char * info;
   int level = 0;            // первоначальный уровень дерева
   SetConsoleCP(1251); // устанавливаем русский язык
   SetConsoleOutputCP(1251);
   system("CLS");
   while (1)
   {
      puts("          вид операции:");
      puts("1 – создать дерево / добавить элементы нерекурсивно");
      puts("2 – создать дерево / добавить элементы рекурсивно");
      puts("3 – рекурсивный вывод содержимого дерева");
      puts("4 – нерекурсивный вывод содержимого дерева");
      puts("5 – вывод содержимого дерева");
      puts("6 – удаление любого элемента из дерева");
      puts("7 – выход");
```

```
        fflush(stdin);
        switch (getch())
        {
                case '1':
                    root = createTree(root);
                    break;
                case '2':
                    info = (char*)malloc(MAX_LEN); // выделяем память под строку
                    while (1)
                    {
                        puts("Введите информацию в узел дерева:\n");
                        gets(info);
                        if (*info) // если информация введена
                            root = createTreeRecursive(root, info);
                        else
                            break;
                    }
                    free(info); // освобождаем память, выделенную под строку
                    break;
                case '3':
                    showTreeRecursive(root);
                    getch();
                    break;
                case '4':
                    showTreeUsingStack(root);
                    getch();
                    break;
                case '5':
                    showTree(root, level);
                    getch();
                    break;
                case '6':
                    deleteNode(root);
                    break;
                case '7':
                    return 0;
        }
        system("CLS"); // очищаем экран
    }
    return 0;
}
    //  Функция создания дерева (добавления элементов)
TREE* createTree(TREE *root)
{
    TREE *currentTree, *subtree; // создаём два указателя на дерево
    char *info;
    info = (char *)malloc(MAX_LEN); //выделяем память под
    while (1)
    {
        puts("Введите информацию в узел дерева:\n");
        gets(info);
```

```
if (!*info) // если ничего не введено
{
   free(info); // освобождаем выделенную память
   if (!root) // если дерево не создано
      return NULL; // возвращаемся в main с кодом 0
   else
      return root; // возвращаем указатель на корень дерева
}
currentTree = (TREE *)malloc(sizeof(TREE)); // создаём новый узел дерева
  //выделяем память под информацонное поле
currentTree->info = (char *)malloc(strlen(info) + 1);
strcpy(currentTree->info, info); // заносим в него введённую информацию
currentTree->repeats = 1; // увеличиваем счётчик встреч
currentTree->left = currentTree->right = 0;
if (!root)
{
   root = currentTree;
}
else
{
   subtree = root; // создаём поддерево
   while (subtree) // пока не дошли до конца поддерева
   {  // если такая информация уже содержится в дереве
      if (!strcmp(currentTree->info, subtree->info))
      {
         subtree->repeats++; // увеличиваем счётчик встреч
         break;
      }
      else
      {  // если новый элемент < значения в узле
         if (strcmp(currentTree->info, subtree->info) < 0)
         {
            if (!subtree->left) // если не создан левый узел поддерева
            {
               subtree->left = currentTree; // заполняем его новым элементом
               subtree = currentTree->left;
            }
            else
            {  // переходим на левое поддерево следующего уровня
               subtree = subtree->left;
            }
         }
         else
         {
            if (!subtree->right) // если не создан правый узел поддерева
            {
              subtree->right = currentTree; // заполняем его новым элементом
              subtree = currentTree->right;
            }
            else
            {   // переходим на правое поддерево следующего уровня
```

```
                    subtree = subtree->right;
                }
            }
        }
    }
    free(info);    // освобождаем память, выделенную под строку
    return root;  // возвращаем указатель на корень дерева
}
    // Функция рекурсивного создания дерева / добавления элементов в дерево
TREE* createTreeRecursive(TREE *root, char* info)
{
    if (!root)       // если корень дерева ещё не создан
    {
        root = (TREE *)malloc(sizeof(TREE)); // создаём корень дерева
        // выделяем память под информационное поле дерева
        root->info = (char *)malloc(strlen(info) + 1);
        strcpy(root->info, info); // копируем информацию в поле дерева
        root->repeats = 1; // увеличиваем счётчик встреч
        root->left = 0;
        root->right = 0;
    }
    else
    {
        if (!strcmp(info, root->info)) // если такой узел уже есть в дереве
        {
            root->repeats++; // увеличиваем счётчик встреч
        }
        else
        {   // если новый элемент меньше значения в узле
            if (strcmp(info, root->info) < 0)
            {   // переходим на узел слева
                root->left = createTreeRecursive(root->left, info);
            }
            else
            {   // переходим на узел справа
                root->right = createTreeRecursive(root->right, info);
            }
        }
    }
    return root; // возвращаем указатель на корень дерева
}
    // Рекурсивный вывод содержимого бинарного дерева
void showTreeRecursive(TREE *root)
{
    if (root)
    {
        printf("узел содержит :   %s , число встреч %d\n", root->info, root->repeats);
        if (root->left)
            // рекурсивный вызов функции для левого узла дерева
```

```
            showTreeRecursive(root->left);
        if (root->right)
            // рекурсивный вызов функции для правого узла дерева
            showTreeRecursive(root->right);
    }
}
    /* Нерекурсивный вывод содержимого бинарного дерева, используя стек
       для занесения адресов узлов дерева */
void showTreeUsingStack(TREE *root)
{
    struct stack // создаём стек
    {
        TREE *tree; // указатель на дерево
        struct stack *next; // указатель на следующий элемент стека
    } *stackElement, *stackPointer = NULL;
    int flag = 1;
    if (!root)
    {
        printf("Дерево пусто!\n");
        return;
    }
    stackElement = (struct stack *)calloc(1, sizeof(struct stack));

    stackElement->tree = root;  // в стек заносится элемент, содержащий указа-
                                // тель на корень дерева для прохода по левому и
                                // правому поддеревьям
    stackElement->next = stackPointer; // указатель на следующий элемент стека
    printf("узел содержит :   %s , число встреч %d\n", root->info, root->repeats);
    while (stackElement || root->right)
    {
        do
        {
            if (flag && root->left)
                root = root->left;            // переход на узел слева
            else
            {
                if (root->right)
                    root = root->right;       // переход на узел справа
            }
            flag = 1;                         // сброс принудительного движения вправо
            if (root->left && root->right) // узел с 2 связями вниз
            {
                stackPointer = stackElement;
                stackElement = (struct stack *)calloc(1, sizeof(struct stack));
                stackElement->tree = root;            // указатель на найденный узел
                stackElement->next = stackPointer; // на следующий элемент стека
            }
            printf("узел содержит :   %s , число встреч %d\n", root->info,
                    root->repeats);
        } while (root->left || root->right);
        if (stackElement)
```

```
        {
            root = stackElement->tree;              // возврат на узел ветвления
            stackPointer = stackElement->next; // адрес узла выше удаляемого
            free(stackElement); // удаление из стека указателя на узел
        }                               // после прохода через него налево
        stackElement = stackPointer;
        if (root->right)
            flag = 0;  // признак принудительного перехода на узел, расположенный
                       // справа от root, т. к. root->info уже выведен при проходе слева
    }
}
    // Рекурсивный вывод дерева по уровням
void showTree(TREE* root, int level)
{
    int i;
    if (root)
    // рекурсивный вызов функции для вывода правого узла следующего уровня
        showTree(root->right, level + 1);
    for (i = 0; i < level; ++i)
        printf("   ");
    if (root)
        printf("%s (%d)\n", root->info, root->repeats);
    else
        printf("@\n");
    if (root)
      // рекурсивный вызов функции для вывода левого узла следующего уровня
        showTree(root->left, level + 1);
}
    // Функция удаления узла дерева
TREE* deleteNode(TREE *root)
{
    TREE *parentNode, *currentNode, *tempNode;
    char *info;       // строка для удаления
    int comparing; // результат сравнения двух строк
    int flag;         // индикатор для выхода из цикла
    if (!root)
    {
        puts("Дерево не создано \n");
        return NULL;
    }
    puts("Введите информацию в для поиска удаляемого узла: ");
    info = (char *)malloc(sizeof(char)*MAX_LEN); // выделяем память под строку
    fflush(stdin);
    gets(info); // получаем узел для удаления
    if (!*info)    // если ничего не введено
        return NULL;
    currentNode = parentNode = root;
    flag = 0; // 1 – признак выхода из цикла поиска
    do          // блок поиска удаляемого из дерева узла
    {
        if (!(comparing = strcmp(info, parentNode->info))) // есть такой узел
```

```
            flag = 1;           // выходим из цикла do ... while
        if (comparing < 0) // введённая строка < строки в анализируемом узле
        {
            if (parentNode->left) // если у анализируемого узла есть левое поддерево
            {
                currentNode = parentNode;        // запоминаем текущий узел
                parentNode = parentNode->left; // переходим на левое поддерево
            }
            else
                flag = 1;        // выходим из цикла do ... while
        }
        if (comparing > 0) // введённая строка > строки в анализируемом узле
        {
            if (parentNode->right) // если у узла есть правое поддерево
            {
                currentNode = parentNode;          // запоминаем текущий узел
                parentNode = parentNode->right; // переходим на правое поддерево
            }
            else
                flag = 1;        // выход из цикла do ... while
        }
    } while (!flag);
    free(info);        // освобождаем память, выделенную под строку
    if (comparing) // если не найдено совпадений
    {
        puts("Требуемый узел не найден \n");
        getch();
        return root; // возвращаем указатель на корень дерева
    }
    else               // если совпадения найдены
    {   // сравниваем вершины поддеревьев
        comparing = strcmp(parentNode->info, currentNode->info);
        tempNode = parentNode;
        if (comparing < 0) // удаляемая вершина < предыдущей
        {
            tempNode = parentNode->right; // переходим в правое поддерево,
            if (!tempNode)                          // если оно пусто
                currentNode->left = NULL; // удаляем необходимый узел,
            else                                    // если правое поддерево есть
            {
                while (tempNode->left)  // пока есть левое поддерево,
                    // переставляем его элементы на один уровень вверх
                  tempNode = tempNode->left;
                  // сдвиг ветви, начинающейся с адреса parentNode->r влево
                currentNode->left = parentNode->right;
                tempNode->left = parentNode->left;
            }
        }
        else if( comparing > 0) // удаляемая вершина > предыдущей
        {
            tempNode = parentNode->left; // переходим в левое поддерево
```

```
            if (!tempNode)                    // если оно пусто
               currentNode->right = NULL; // удаляем необходимый узел
            else
            {
               while (tempNode->right)  // пока есть правое поддерево, переставляем
                                        // его элементы на один уровень вверх
                  tempNode = tempNode->right;
                  // сдвиг ветви, начинающейся с адреса parentNode->l вправо
               currentNode->right = parentNode->left;
               tempNode->right = parentNode->right;
            }
         }
         else     // попытка удалить корень дерева
         {
            printf("Данная функция не предусматривает удаление корня дерева!\n");
            getch();
         }
      }
   }
```

12. Файлы

12.1. Основные сведения о файловой системе

Файловая система ввода/вывода в языке С (С++) реализуется с помощью взаимосвязанных библиотечных функций, а не ключевых слов. Для их работы требуется заголовок **#include <stdio.h>**. Это позволяет подключить файл **stdio.h**, содержащий прототипы функций ввода/вывода, и определяет следующие три типа: **size_t**, **fpos_t** и **FILE**. Типы **size_t** и **fpos_t** представляют собой определённые разновидности целого беззнакового типа. А третий тип, **FILE**, рассмотрим ниже. Кроме этого, определяются несколько констант (точнее макроопределений): **NULL**, **EOF**, **FOPEN_MAX**, **SEEK_SET**, **SEEK_CUR** и **SEEK_END**. Макроопределение NULL определяет пустой (null) указатель. Макроопределение EOF, часто определяемое как «−1», является значением, возвращаемым тогда, когда функция ввода пытается выполнить чтение после конца файла. **FOPEN_MAX** определяет целое значение, равное максимальному числу одновременно открытых файлов. Другие макроопределения используются вместе с функцией fseek(), выполняющей операции прямого доступа к файлу.

Файл – это именованный объект, хранящий данные любого типа на каком-либо носителе. Файлы используются для размещения данных, предназначенных для длительного хранения. Каждому файлу присваивается уникальное имя, используемое далее для обращения к нему. Файл, как и массив, – это совокупность данных и в этом их сходство, но также существуют некоторые различия:

− файлы в отличие от массивов располагаются не в оперативной памяти, а на некотором носителе информации (жёсткий диск, flash-карты);

− файл не имеет фиксированной длины, т. е. может увеличиваться и уменьшаться;

− перед работой с файлом его необходимо открыть, а после работы – закрыть.

В языке С (С++) имеются два способа работы с файлами: через указатель на тип **FILE** и используя дескриптор файла. Рассмотрим вначале организацию работы с файлами, основанную на использовании указателя. Указатель файла определяет конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода. Для выполнения в файлах операций чтения и записи программы должны использовать указатели соответствующих файлов. Указатель на файл – это переменная, идентифицирующая конкретный дисковый файл (его адрес), используется для организации ввода/вывода. Он указывает на структуру **FILE**, содержащую различные сведения о файле, такие, как имя файла, статус и указатель текущей позиции в начало файла и др. Определение указателя на файл имеет вид

```
FILE *f;
```

Прежде чем производить действия с файлом (чтение, запись), файл должен быть открыт. Функция **fopen()** открывает для использования поток, связывает файл с данным потоком и возвращает указатель на открытый файл. Функция **fopen()** имеет следующий прототип:

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

Здесь **mode** указывает на строку, содержащую режим открытия файла. Возможны следующие режимы (табл. 11).

Таблица 11

Режим	Действие
r	Файл открывается только для чтения. Если нужного файла на диске нет, то возникает ошибка
w	Файл открывается только для записи. Если файла с заданным именем нет, то он будет создан, если же такой файл существует, то перед открытием прежняя информация уничтожается
a	Файл открывается для дозаписи новой информации в конец файла
r+	Файл открывается для редактирования его данных. Возможны и запись, и чтение информации
w+	То же, что и для r+
a+	То же, что и для a, только запись можно выполнять в любое место файла. Доступно и чтение файла
t	Файл открывается в текстовом режиме. Используется вместе с режимами r, w, a, r+, w+, a+
b	Файл открывается в двоичном режиме. Используется вместе с режимами r, w, a, r+, w+, a+

Значение **b** вызывает открытие файла в двоичном режиме. По умолчанию все файлы открываются в текстовом режиме. В текстовом режиме имеет место преобразование некоторых символов, например, последовательность символов «возврат каретки/перевод строки» превращается в символ новой строки. Если же файл открывается в двоичном режиме, такого преобразования не выполняется. Любой файл независимо от того, что в нём содержится – отформатированный текст или необработанные данные – может быть открыт как в текстовом, так и в двоичном режиме. Отличие между ними заключается только в отсутствии или наличии упомянутого символьного преобразования.

Функция **fopen()** инициализирует указатель адресом открываемого для работы файла. В случае если спецификация файла задана неверно, то **fopen()** возвращает указатель **NULL**. Ниже приводится пример использования функции **fopen()** для открытия файла по имени «test».

```
FILE *f;
f = fopen("test", "r+");
```

Более правильно данный код написать немного иначе:

```
FILE *f;
if ((f = fopen("test","r+"))==NULL)
{
   printf("Ошибка при открытии файла.\n");
   exit(1);
}
```

Этот метод помогает при открытии файла обнаружить ошибку (например,

отсутствие файла на носителе) до попытки из этого файла что-либо прочитать. Всегда нужно вначале получить подтверждение, что функция **fopen()** выполнилась успешно, и лишь затем выполнять с файлом операции ввода и вывода.

Функция **fclose()** разрывает связь указателя с файлом и закрывает его к дальнейшему использованию до тех пор, пока он вновь не будет открыт. При этом записываются в файл все данные, которые ещё оставались в дисковом буфере, и выполняет закрытие файла на уровне операционной системы. Завершение работы с файлом без закрытия потока может привести к потере данных, порче файла и другим ошибкам. Прототип функции **fclose()** имеет вид

```
int fclose(FILE *stream);
```

Здесь stream – указатель файла. Возвращение нуля означает успешную операцию закрытия. В случае же ошибки возвращается **EOF**. Чтобы точно узнать причину ошибки, можно использовать функцию **ferror()**.

Если требуется изменить режим доступа к файлу, то для этого сначала необходимо закрыть данный файл, а затем вновь его открыть, но с другими правами доступа. Для этого используют стандартную функцию **freopen**, описанную в **stdio.h** как

```
FILE* freopen (const char* file_name, const char* mode, FILE  *stream)
```

Эта функция сначала закрывает файл, связанный с потоком stream (как это делает функция **fclose**), а затем открывает файл с именем filename и правами доступа mode, записывая информацию об этом файле в переменную stream.

Далее кратко остановимся на некоторых, наиболее важных функциях языка С, обеспечивающих работу с файлами по организации ввода и вывода информации. В табл. 12 привёден список наиболее часто используемых из них.

Таблица 12

Наименование	Назначение функции
fopen()	Открывает файл
fclose()	Закрывает файл
putc(), fputc()	Записывает символ в файл
getc(), fgetc()	Читает символ из файла
fgets()	Читает строку из файла
fputs()	Записывает строку в файл
fseek()	Устанавливает указатель текущей позиции на определённый байт файла
ftell()	Возвращает текущее значение указателя текущей позиции в файле
fprintf()	Для файла то же, что printf() для консоли
fscanf()	Для файла то же, что scanf() для консоли
feof()	Возвращает значение true, если достигнут конец файла
ferror()	Возвращает значение true, если произошла ошибка
rewind()	Устанавливает указатель текущей позиции в начало файла и сбрасывает состояние ошибки в потоке
remove()	Удаляет файл
fflush()	Запись потока в файл

12.2. Организация посимвольного ввода и вывода

В языке C (C++) определены две эквивалентные функции, предназначенные для вывода (записи) символов в файл: **putc()** и **fputc()**. Функция **putc()** записывает символы в открытый ранее файл для записи. Прототип **putc()** имеет вид

```
int putc(int ch, FILE* stream);
```

Здесь stream – это указатель на файл, а ch – выводимый в него символ. В случае если запись выполнилась успешно, то возвращается записанный символ, иначе возвращается **EOF.**

Ввод (чтение) символа из файла, открытого для чтения, обеспечивается парой функций: **getc()** и **fgetc()**. Прототип функции **getc()** имеет вид

```
int getc(FILE *stream);
```

Функция **getc()** возвращает целое значение, младший байт которого содержит ASCII-код символа. Старший байт при этом обнулён. При достижении конца файла функция **getc()** возвращает **EOF**. Пример фрагмента программы чтения символов до конца текстового файла:

```
do
{  ch = getc(f);
}  while(ch!=EOF);
```

Однако **getc()** возвращает **EOF** и в случае ошибки.

12.3. Определение конца файла feof()

Как отмечалось ранее, при прочтении конца файла функция **getc()** возвращает **EOF**. Однако **EOF** может быть возвращён и в случаях, когда конец файла еще не достигнут. Во-первых, когда файл открывается для двоичного ввода, то прочитанное значение может совпадать с **EOF**. При этом программа зафиксирует достижение конца файла, чего на самом деле может и не быть. Во-вторых, функция **getc()** возвращает **EOF** и при ошибке ввода, когда не достигнут конец файла. Для решения этой проблемы в языке C (C++) имеется функция **feof()**, которая определяет, достигнут ли конец файла. Прототип функции **feof()** имеет вид

```
int feof(FILE *stream);
```

Если конец файла достигнут, то **feof()** возвращает 1, иначе − 0. Поэтому следующий код будет читать двоичный файл до тех пор, пока не будет достигнут конец файла:

```
while(!feof(fp))
{
  ch = getc(fp);
  if(!feof(in)) printf("%c",ch);
}
```

12.4. Организация ввода и вывода строк

Функции **fgets()** и **fputs()** обеспечивают построчный ввод и вывод информации. Первая читает строки символов из файла, а вторая записывает строки в файл. Прототипы этих функций следующие:

```
int fputs(const char *str, FILE *stream);
char *fgets(char *str, int len, FILE *stream);
```

Функция **fputs()** записывает в поток строку, на которую указывает **str**. В случае ошибки эта функция возвращает **EOF**.

Функция **fgets()** читает из потока строку до тех пор, пока не будет прочитан символ новой строки «'\n'» или количество прочитанных символов не станет равным len-1. Если был прочитан разделитель строк «'\n'», он преобразуется в «'\0'» и записывается в строку. При успешном завершении работы функция возвращает str, а в случае ошибки – пустой указатель (**NULL**).

В следующем фрагменте программы используется функция fputs() для записи в файл строк, считываемых с клавиатуры. Ввод в файл завершается вводом пустой строки.

```
char str[60];
FILE *f;
if((fp = fopen("test", "w"))==NULL)
{
   printf("Ошибка при открытии файла.\n");
   exit(1);
}
do
{
   printf("Введите строку (ввод пустой строки – выход из программы):\n");
   gets(str);                  // ввод строки с клавиатуры
   if(*str='\0') break;        // введена пустая строка
   strcat(str, "\n");          // добавление разделителя строк
   fputs(str, fp);             // запись введённой строки в файл
} while(*str!='\n');
```

12.5. Удаление файлов

Функция **remove()** удаляет указанный файл. Прототип функции имеет вид

```
int remove(const char *file_name);
```

В случае успешного выполнения эта функция возвращает нуль, иначе – ненулевое значение.

Во фрагменте программы демонстрируется удаление файла, имя которого вводится с клавиатуры.

```
char str[80];
printf("\n Введите имя файла для удаления ");
gets(str);
if(*str)
   if(remove(str))
   {
      printf("\nНельзя удалить файл.\n");
      exit(1);
   }
```

12.6. Дозапись потока

Для дозаписи содержимого выводного потока в файл применяется функция **fflush()**:

```
int  fflush(FILE *stream).
```

Все данные, находящиеся в буфере, записываются в файл, который указан

с помощью stream. При вызове функции **fflush()** с пустым (**NULL**) указателем файла stream будет выполнена дозапись во все файлы, открытые для вывода.

При успешном выполнения **fflush()** возвращает нуль, иначе – **EOF**.

12.7. Позиционирование в файле

Каждый открытый файл имеет так называемый указатель на текущую позицию в файле. Все операции над файлами (чтение и запись) выполняются с данными с этой позиции. При каждом выполнении функции чтения или записи указатель смещается на количество прочитанных или записанных байт, т. е. устанавливается сразу за прочитанным или записанным блоком данных в файле. Это так называемый последовательный доступ к данным. Последовательный доступ удобен, когда необходимо последовательно работать с данными в файле. Но иногда необходимо читать или писать данные в произвольном порядке. Это достигается путём установки указателя на некоторую заданную позицию в файле функцией **fseek()**.

```
int fseek(FILE *stream, long offset, int whence);
```

Параметр offset задаёт количество байт, на которое необходимо сместить указатель в направлении, указанном whence. Приведём значения, которые может принимать параметр whence:

SEEK_SET 0 − смещение выполняется от начала файла;
SEEK_CUR 1 − смещение выполняется от текущей позиции указателя;
SEEK_END 2 − смещение выполняется от конца файла.

Величина смещения может быть как положительной, так и отрицательной, но нельзя сместиться за пределы начала файла.

Наряду с функцией **fseek()** для перемещения указателя текущей позиции в файле на начало может быть использована функция **rewind()**. Кроме того, эта функция выполняет сброс состояния ошибки в потоке, если она возникла при работе с файлом. Прототип функции имеет вид

```
void rewind(FILE *f);
```

12.8. Текстовые и двоичные файлы

Различают два вида файлов: *текстовые* и *бинарные*.

Текстовый файл – это последовательность ASCII-кодов символов. В стандарте Си считается, что текстовый файл организован в виде строк. В текстовом файле могут выполняться определённые преобразования символов. Например, символ новой строки может быть заменён парой символов – возврата каретки и перевода строки 13, 10 (0xD, 0xA). Поэтому может и не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся в файле. Текстовые файлы могут быть просмотрены и отредактированы текстовым редактором.

Двоичный файл – это последовательность байт, причём в двоичном файле никакого преобразования символов не происходит. Бинарные файлы – файлы, не имеющие структуры текстовых файлов. Каждая программа для своих бинарных фалов определяет собственную структуру. Кроме того, количество тех байт, которые пишутся (читаются), и тех, которые хранятся на внешнем устройстве, одинаково.

Библиотечный файл **stdio.h** содержит функции для работы как с текстовыми, так и с бинарными фалами. Примеры их использования рассмотрены в подразд. 12.10.

12.9. Функции fread() и fwrite()

Для чтения и записи блоками данных любого типа в файловой системе языка С (С++) имеется две функции: **fread()** и **fwrite()**. Прототипы этих функций имеют соответственно следующий вид:

```
size_t fread(void *buf, size_t size, size_t count, FILE *stream);
size_t fwrite(const void *buf, size_t size, size_t count, FILE *stream);
```

Здесь buf – это указатель на область памяти, в которую будут прочитаны данные из файла (**fread()**), или указатель на данные, которые будут записаны в файл (**fwrite()**); stream – это указатель на файл. Значение count определяет, сколько считывается или записывается элементов данных, причём длина каждого элемента в байтах равна size.

Функция **fread()** возвращает количество прочитанных элементов. Если достигнут конец файла или произошла ошибка, то возвращаемое значение может быть меньше, чем count. А функция **fwrite()** возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению count.

Наряду с доступом к информации в файле, используя указатель типа **FILE***, может быть использован подход, основанный на использовании *дескриптора* (номера) файла. Дескриптор файла имеет тип int. Всякий раз, когда осуществляется ввод/вывод, идентификация файла осуществляется по его номеру.

При работе с файлом может возникнуть необходимость выделить для него большее дисковое пространство или, наоборот, уменьшить его размер. Для этого используется функция chsize. Эта функция требует подключения файла io.h. Прототип функции chsize имеет вид

```
int chsize (int handle, long size);
```

Здесь handle – идентификатор файла, который можно получить, используя функцию **fileno**. Параметр size задаёт требуемый размер файла. При успешном выполнении функция **chsize** возвращает «0». В случае ошибки **chsize** возвращает «−1» и присваивает глобальной переменной **errno** одно из следующих значений (табл. 13).

Таблица 13

Константа	Значение
EACCES	Ошибка доступа к файлу
EBADF	Неверный идентификатор файла
ENOSPC	Недостаточно места (UNIX)

При расширении файла функция **chsize** заполняет вновь выделенное место двоичными нулями.

Прототип функции **fileno**, возвращающей дескриптор (номер) файла, имеет следующий вид:

```
int  fileno(FILE *stream);
```

Здесь stream – указатель на открытый файл.

12.10. Примеры программ

Пример 1. Создать два файла: file1 – с чётными, file2 с нечётными числами, используя функцию, объединить оба файла в один jointFile.

```
#include <stdio.h>
#include <windows.h>
#define  MAX_NUMBER 10
void fcopyVersion1(FILE *sourceFile, FILE *jointFile);
void fcopyVersion2(FILE *sourceFile, FILE *jointFile);
void PrintNumberInFile(FILE *file1, FILE*file2);
int main()
{
   FILE *file1, *file2, *jointFile;
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
        // открытие файлов для записи и чтения
   if (!(file1=fopen("file1.txt","w+")) || !(file2=fopen("file2.txt","w+")))
   {
      puts("Файл не может быть создан");
      return;
   }
   PrintNumberInFile(file1,file2);
   if (!(jointFile=fopen("jointFile.txt","w"))) // открытие файла для записи
   {
      puts("Файл не может быть создан");
      return;
   }
   fcopyVersion1(file1,jointFile)  ; // вариант 1 копирование  file1 в jointFile
   fcopyVersion1(file2,jointFile);   // добавление file2 в jointFile
   /*fcopyVersion2(file1,jointFile); // вариант 2
   fcopyVersion2(file2,jointFile);*/
   fclose(file1); // закрытие файла
   fclose(file2);
   fclose(jointFile);
   return 0;
}
void PrintNumberInFile(FILE *file1, FILE*file2) // запись цифр в файлы
{
   int i = 0;
   for (i = 1; i < MAX_NUMBER; i++)
   {
      if (!(i % 2))
      {
         fprintf(file1, "%d", i); // запись чётных чисел в file1
      }
      else
      {
         fprintf(file2, "%d", i); // запись нечётных чисел в file2
```

```
        }
    }
    rewind(file1); // возврат в начало файла, сброс признака всех ошибок
    rewind(file2);
}
    /*Ниже приводятся 2 варианта копирования информации.
      Вариант 1 – используется цикл while для считывания из файла sourceFile.
      считывание выполняется по одному байту. Вариант 2 – используется цикл
      do{}while для считывания из файла sourceFile*/
    // Копирование файла вариант 1
void fcopyVersion1(FILE *sourceFile,FILE *jointFile)
{
    int symbol = 0;
    while ((symbol = getc(sourceFile)) != EOF) // цикл работает пока не EOF
    {
        putc(symbol, jointFile);
    }
}
    // Копирование файла вариант 2
void fcopyVersion2(FILE *sourceFile,FILE *jointFile)
{
    int symbol = 0;
    do
    {
        fread(&symbol,1,1,sourceFile); // считывание одного символа
        if(feof(sourceFile))       // выход, если следующий символ EOF
            break;
        fwrite(&symbol,1,1,jointFile);    // запись символа в файл
    } while(!feof(sourceFile)); // цикл работает, пока не EOF
}
```

Пример 2. Написать программу добавления в текстовый, упорядоченный по возрастанию файл чисел, вводимых с клавиатуры, не нарушая его упорядоченности.

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#define MAX_ELEMENTS_AMOUNT 5
int ScanNumber(FILE *file);
int PrintInfInFileAndTakeAmount(FILE *file, int originalElements[]);
void PrintNewElements(FILE *file, int originalElements[]);
int PrintNotMaxElement(FILE *file, int elementsAmount, int inputElement,
int shiftAmount, fpos_t position1, fpos_t position2);
int main()
{
    FILE *file;
    int originalElements[]={5,7,10,11,14};
    int elementsAmount = 0;
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    file = fopen("file.txt","w+"); // открытие файла для записи и чтения
```

```
    PrintNewElements(file ,originalElements);
    fclose(file); // закрытие файла
    return 0;
}
    // Ввод полученных данных
void PrintNewElements(FILE *file, int originalElements[])
{
    fpos_t startPosition,currentPosition;
    int inputElement, fileElement, elementsAmount = 0;
    int shiftAmount = 0;
      // ввод данных, получение количества элементов
    elementsAmount = PrintInfInFileAndTakeAmount(file, originalElements);
    while(1)
    {
        // считывание элементов, вводимых с клавиатуры
        inputElement = ScanNumber(file);
        if (inputElement == 999)
            break;
        rewind(file);    // позиционирование на начало файла
        do
        {
            fgetpos(file,&startPosition); // УТПФ на начало поля считываемого числа
            fscanf(file,"%d",&fileElement);
              // достигнут EOF или считано число больше введённого
            if (feof(file) || fileElement>inputElement)
                break;
        } while(1);
          // EOF и в файле нет числа больше, чем введённое
        if (feof(file) && fileElement < inputElement)
        {
            rewind(file); // установка УТПФ в начало файла
            fseek(file, 0, 2); // выход на конец файла
            fprintf(file, "%3d", inputElement); // дозапись в конец файла
            elementsAmount++;
            continue;
        }
        rewind(file); // установка УТПФ в начало файла
        fseek(file, –3, 2); // УТПФ на последний элемент файла
        shiftAmount = 1;
        do   // сдвиг всех чисел в массиве до currentPosition
        {
            // currentPosition указывает на последний элемент
            fgetpos(file, &currentPosition);
            fscanf(file, "%d", &fileElement);  // считывание последнего элемента
            rewind(file);
            currentPosition += 3;
            fsetpos(file, &currentPosition);   // установка УТПФ на конец файла
            fprintf(file, "%3d", fileElement);   // запись элемента в конец файла
            shiftAmount++;
              // установка УТПФ на максимальный исходный элемент
            fseek(file, –3 * shiftAmount – 3, 2);
```

```
            // выход, если вводимый элемент немаксимальный
            if (elementsAmount + 1 == shiftAmount)
                break;
        } while(startPosition != currentPosition);
          // запись немаксимальных элементов
        elementsAmount = PrintNotMaxElement(file, elementsAmount, inputElement,
            shiftAmount, startPosition, currentPosition);
        puts("Элемент добавлен в файл");
    }
    return;
}
    // Считывание числа с клавиатуры
int ScanNumber(FILE *file)
{
    int inputElement;
    system("cls");
    puts("Введите элемент для записи или 999 для выхода");
    scanf("%d", &inputElement);   // считывание элемента с клавиатуры
    return inputElement;
}
    // Ввод исходных данных
int PrintInfInFileAndTakeAmount(FILE *file, int originalElements[])
{
    int temp = 0, elementsAmount = 0;
    for (temp = 0; temp < MAX_ELEMENTS_AMOUNT; temp++)
    {
        fprintf(file, "%3d", originalElements[temp]); // запись исходных данных
        elementsAmount++;   // счётчик количества элементов
    }
    return elementsAmount;
}
    // Ввод максимального элемента
int PrintNotMaxElement(FILE *file, int elementsAmount, int inputElement,
    int shiftAmount, fpos_t startPosition, fpos_t currentPosition)
{
    if (elementsAmount + 2 < shiftAmount)
        {
            fsetpos(file, &currentPosition);    // запись inputElement на место числа,
            fprintf(file, "%3d", inputElement); // с которого произведён сдвиг вниз
            elementsAmount++;
        }
        else
        {
              // запись элемента, который меньше всех данных
            fsetpos(file, &startPosition);
            fprintf(file, "%3d", inputElement);
            elementsAmount++;
        }
    return elementsAmount;
}
```

Пример 3. Пусть имеются два файла (file1.txt и file2.txt), оба упорядочены по возрастанию. Необходимо переписать их в результирующий file3.txt без нарушения упорядоченности. Все файлы текстовые.

```
#include<stdio.h>
#include<limits.h>
#include<conio.h>
voidPrintEndElements(FILE *file1, FILE *file2, FILE *fileResult, int isScanfElement1,
                     int elementOfFile1, int elementOfFile2);
voidPrintElementsInNewFile(FILE *file1, FILE *file2, FILE *fileResult);
int main()
{
   FILE *file1, *file2, *fileResult;
   int elementOfFile1, elementOfFile2, isScanfElement1, isScanfElement2;
   if (!(file1 = fopen("file1.txt", "r")) ||        // открытие file1 для чтения
      !(file2 = fopen("file2.txt", "r")) ||        // открытие file2 для чтения
      !(fileResult = fopen("file3.txt", "w"))) // открытие file3 для записи
   {
      puts("error");
      getch();
      return;
   }
   PrintElementsInNewFile(file1, file2, fileResult);
   fclose(file1); // закрытие файла
   fclose(file2);
   fclose(fileResult);
   return 0;
}
// Запись элементов в новый файл
void PrintElementsInNewFile(FILE *file1, FILE *file2, FILE *fileResult)
{
   int elementOfFile1 = 0, elementOfFile2 = 0;
   int isScanfElement1 = 0, isScanfElement2 = 0;
   // предварительное считывание начальных значений из каждого файла
   fscanf(file1, "%d", &elementOfFile1);
   fscanf(file2, "%d", &elementOfFile2);
   do
   {
      // выбранный элемент 1 файла меньше элемента 2 файла
      if (elementOfFile1 < elementOfFile2)
      {
         // пока не EOF file1 и элемент 1 файла меньше элемента 2 файла
         while (!feof(file1) && elementOfFile1 < elementOfFile2)
         {
            // запись элемента 1 файла в результирующий файл
            fprintf(fileResult, "%d ", elementOfFile1);
            // считывание следующего элемента из файла 1
            isScanfElement1 = fscanf(file1, "%d", &elementOfFile1);
         }
          // дозапись последнего элемента из файла 1, если после последнего
          // элемента EOF
```

```
            if (feof(file1) && elementOfFile1 < elementOfFile2 && isScanfElement1 > 0)
            {
                fprintf(fileResult, "%d ", elementOfFile1);
                elementOfFile1 = INT_MAX; // присваивание максимального значения
            }
        }
        else
        {
            // пока не EOF file2 и элемент 1 файла больше элемента 2 файла
            while (!feof(file2) && elementOfFile1 >= elementOfFile2)
            {
                // запись элемента 2 файла в результирующий файл
                fprintf(fileResult, "%d ", elementOfFile2);
                // считывание следующего элемента из файла 2
                isScanfElement2 = fscanf(file2, "%d", &elementOfFile2);
            }
            // дозапись последнего элемента из файла 2, если после последнего
            // элементата EOF
            if (feof(file2) && elementOfFile1 >= elementOfFile2 && isScanfElement2 > 0)
            {
                fprintf(fileResult, "%d ", elementOfFile2);
                elementOfFile2 = INT_MAX; // присваивание максимального значения
            }
        }
    } while(!feof(file1) || !feof(file2));        // пока не достигнут EOF файлов
    PrintEndElements(file1, file2, fileResult, isScanfElement1,
        elementOfFile1, elementOfFile2); // запись последних элементов из файлов
    return;
}
// Запись последних элементов из файлов
void PrintEndElements(FILE *file1, FILE *file2, FILE *fileResult, int isScanfElement1,
                      int elementOfFile1, int elementOfFile2)
{
    // запись последнего элемента из file1, если он меньше
    if (elementOfFile1 < elementOfFile2 && isScanfElement1 > 0)
    {
        fprintf(fileResult, "%d ", elementOfFile1);
        if (elementOfFile2 != INT_MAX)   // если элемент file2 немаксимальный int
            // запись последнего элемента из file2
            fprintf(fileResult, "%d", elementOfFile2);
    }
    if (elementOfFile1 >= elementOfFile2&&isScanfElement1 > 0 &&
        elementOfFile2 != INT_MAX)
    {
        fprintf(fileResult, "%d ", elementOfFile2);
        if (elementOfFile1 != INT_MAX)    // если элемент file1 немаксимальный int
            // запись последнего элемента из file2
            fprintf(fileResult, "%d", elementOfFile1);
    }
    return;
}
```

Литература

1. Керниган, Б. Язык программирования С / Б. Керниган, Д. Ритчи. – СПб. : Невский диалект, 2001. – 352 с.

2. Керниган, Б. Язык программирования С / Б. Керниган, Д. Ритчи. – 2-е изд. – М. : Издательский дом «Вильямс», 2009. – 304 с.

3. Керниган, Б. Язык программирования С / Б. Керниган, Д. Ритчи. – 3-е изд. – М. : Издательский дом «Вильямс», 2013. – 304 с.

4. Романовская, Л. М. Программирование в среде С (С++) для ПЭВМ / Л. М. Романовская, Т. В. Русс, С. Г. Свитковский. – М. : Финансы и статистика, 1992. – 252 с.

5. Шилд, Г. Программирование на Borland C++ / Г. Шилд. – Минск : ООО «Попурри» 1998. – 800 с.

6. Подбельский, В. В. Программирование на языке Си / В. В. Подбельский, С. С. Фомин. – М. : Финансы и статистика, 2004. – 600 с.

7. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – СПб. : Невский Диалект, 2001. – 352 с.

8. Демидович, Е. М. Основы алгоритмизации и программирования. Язык Си / Е. М. Демидович. – СПб. : БХВ-Петербург, 2008. – 440 с.

9. Хусаинов, Б. С. Структуры и алгоритмы обработки данных. Примеры на языке Си / Б. С. Хусаинов – М. : Финансы и статистика, 2004. – 464 с.

10. Кочан, С. Программирование на языке С / С. Кочан. – М. : Издательский дом «Вильямс», 2007. – 496 с.

11. Луцик, Ю. А. Арифметические и логические основы вычислительной техники : учеб. пособие / Ю. А. Луцик, И. В. Лукьянова. – Минск : БГУИР, 2014. – 165 с.

Содержание

Св. план 2014, поз. 39

Учебное издание

Луцик Юрий Александрович
Ковальчук Анна Михайловна
Сасин Евгений Александрович

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ: ЯЗЫК СИ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *Е. С. Чайковская*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *А. А. Лущикова*

Подписано в печать 15.01.2015. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 10,11. Уч.-изд. л. 10,0. Тираж 250 экз. Заказ 258.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6