МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Н. В. Груздева, Ю. С. Корухова

Семинары по курсу «Алгоритмы и алгоритмические языки, язык Си»

(учебное пособие)

Москва
2026

Введение

В учебном пособии представлены материалы основной части семинарских занятий в поддержку лекционного курса «Алгоритмы и алгоритмические языки» для студентов 1 курса факультета ВМК МГУ. Курс состоит из трех разделов:

– элементы общей теории алгоритмов;

– язык программирования Си [1, 2], как пример алгоритмического языка;

– структуры данных и классические алгоритмы.

В дополнении к учебным пособиям [1, 2], рассматриваются основные темы семинарских занятий, связанные со второй и третьей частями курса. При рассмотрении тем кратко формулируются основные понятия и расставляются акценты. Приводятся решения типовых задач, которые авторы рекомендуют рассмотреть на семинарских занятиях, а некоторые из них могут быть предложены студентам в качестве домашних заданий.

Часть занятий проводится на компьютерах. Номера задач, выполнение которых рекомендуется на компьютере, выделены в списке задач подчеркнутым шрифтом.

В рамках курса предусмотрено проведение нескольких контрольных работ и проверка домашних заданий. Выполнение практических и домашних заданий предполагает составление алгоритма, его запись на языке программирования, приобретение навыков тестирования и отладки программ. В рамках семинаров рассматривается работа в системах семейства UNIX. Курс не ориентирован на изучение одной конкретной среды программирования: работа с программой, ее компиляция, запуск, отладка предполагаются непосредственно из командной строки.

Тема 1.

Первая программа на языке С.

Константы, переменные, типы данных, выражения (начало).

Первая программа

Код программы (содержимое файла `first.c`):

```
1  #include <stdio.h>
2  int main()
3  { int x = 10, y = 3;
4    int t;
5    t = x / y;
6    printf("Result = %d и %lf, t = %d\n", 12/5, 12.0/5, t);
7    return 0;
8  }
```

Прокомментируем, что же тут написано.

В первой строке директива подключения библиотеки, содержащей функции ввода и вывода. Со второй строки начинается описание функции с именем `main` – с этой функции начинается выполнение программы. `int` – тип результата функции, `main` - имя функции, у неё не задаются аргументы – поэтому после имени пустые скобки. Тело функции – содержимое фигурных скобок (строки 3-8).

Строка 3 – описание двух переменных целого типа (`int`) с инициализацией начальными значениями – переменная с именем «`x`» и начальным значением `10`, переменная с именем «`y`» и начальным значением `3`. Одиночный знак равенства – операция присваивания.

Строка 4 – описание целочисленной переменной `t` без инициализации. Под эту переменную отводится память, но начального значения в этот участок памяти не записывается, там какой-то «мусор» – то, что осталось от прежних вычислений, в которых участвовал этот участок памяти.

Строка 5 – переменной `t` присваивается результат целочисленного деления x на y. Почему деление целочисленное? Потому, что его операнды – переменные `x` и `y` – имеют целочисленный тип. Если хотя бы один из операндов вещественный, то и деление тогда будет делением вещественных чисел.

Строка 6 – вывод на экран строки, содержащей три значения. Первый аргумент `printf` называется форматной строкой. Это записанная в двойных кавычках строка, внутри которой встречаются `%d` и `%lf`. В выводимой строке `%d` заменяется на целое число, `%lf` заменяется на вещественное число. Самый левый `%d` заменится на значение выражения, следующего за форматной строкой (второй аргумент, результат целочисленного деления `12/5`). `%lf` заменится на результат вещественного деления (третий аргумент, результат вещественного деления `12.0/5`). Правый `%d` заменится на значение целочисленной переменной `t`. Символ `'\n'` означают переход на новую строку.

Строка 7 – возврат в операционную систему значения ноль после завершения работы программы. Возвращаемый ноль означает, что вычисления прошли нормально, ошибок не возникло. В случае возникновения ошибок в процессе вычисления программы рекомендуется значение, отличное от нуля.

Исходный файл с программой на языке С должен иметь расширение «`.c`». Назовём файл с нашей первой программой `first.c` и сохраним в текущей директории. Чтобы программа могла исполниться, нужно её откомпилировать. Для этого в командной строке нужно набрать:

```
gcc first.c –ansi –o first
```

Обращение к компилятору языка С `gcc` для компиляции файла `first.c` согласно стандарту ANSI (флаг `–ansi`), имя выходного исполняемого файла `first` (флаг `–o` с именем `first` после него). Если опустить `–o first`, то имя выходного исполняемого файла будет `a.out`. После успешной компиляции программы можно запустить исполняемый файл. В Unix-системах для этого в командной строке нужно набрать имя исполняемого файла, возможно, с указанием пути к нему.

```
./first
```

Здесь `.` обозначает текущий каталог, в нем будет взят файл `first` (без расширения).

Типы данных

При описании программ необходимо уделить внимание двум основным аспектам – тому, как описываются действия (операторы), и тому, как представляются объекты (данные), с которыми они работают. Мы начнем с изучения способов представления данных языка Си.

В программировании под типом данных понимается множество объектов, в чем-то близких друг к другу (например, чисел или символов), плюс набор операций, который можно к ним применять.

Типы используются при описании объектов языка. При описании переменных эта информация (тип переменной) нужна компилятору для того, чтобы отвести необходимый для хранения значения этой переменной участок памяти (для каждого типа известно, сколько байт памяти нужно отвести для хранения значения этого типа). В описании функции описываются типы её аргументов и тип возвращаемого объекта. В базовый набор типов языка Си включены следующие простые типы:

`char` (символы), `int` (целые), `float` (вещественные), `double` (вещественные двойной точности).

К названиям некоторых типов могут быть добавлены квалификаторы - слова `short` или `long` (они влияют на размер типа – количество отводимых для хранения значений байтов: `short` – короткие – меньше байтов, или `long` – длинные – больше байтов). Пара других слов-квалификаторов – это слова `signed` и `unsigned`. Они не влияют на размер типа, но влияют на то, как хранящиеся в памяти последовательности 0 и 1, кодирующих значение этого типа, будут восприниматься компьютером – как числа со знаком (`signed`) или числа без знака (`unsigned`).

Все возможные сочетания в названии типов и комментарии к ним приведены ниже.

`char`	символы, под хранение значений отводится 1 байт
`signed char`	знаковые байты, диапазон значений от -128 до 127
`unsigned char`	беззнаковые байты, диапазон значений от 0 до 255
`signed` **`short`** `int`	короткие знаковые целые
`unsigned short` `int`	короткие беззнаковые целые
`signed` **`int`**	знаковые целые
`unsigned` `int`	беззнаковые целые
`signed` **`long`** `int`	длинные знаковые целые
`unsigned long` `int`	длинные беззнаковые целые
`signed` **`long long`** `int`	расширенные длинные знаковые целые (C99)
`unsigned long long` `int`	расширенные беззнаковые целые (C99)
`float`	вещественные

`double` вещественные двойной точности

`long double` вещественные расширенной точности (ещё больше памяти для представления числа)

В названиях типов, состоящих из нескольких слов, часть слов названия может быть опущена. В названиях типов подчёркнутым выделена та часть, которая должна быть обязательно указана для типа, остальные слова названия типа могут быть опущены. Например, можно заметить, что по умолчанию все целые считаются знаковыми (signed может быть опущено в названии целочисленного типа).

Можно узнать размер типа с помощью функции `sizeof`(тип). Ее результатом является количество байт, отводимых под хранение значения указанного в аргументе типа. Размер типа зависит от реализации, но для любой реализации выполняются все следующие неравенства:

sizeof(char) < sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
sizeof(char) = 1
sizeof (short) < sizeof(long)
sizeof(int) < sizeof(long long)

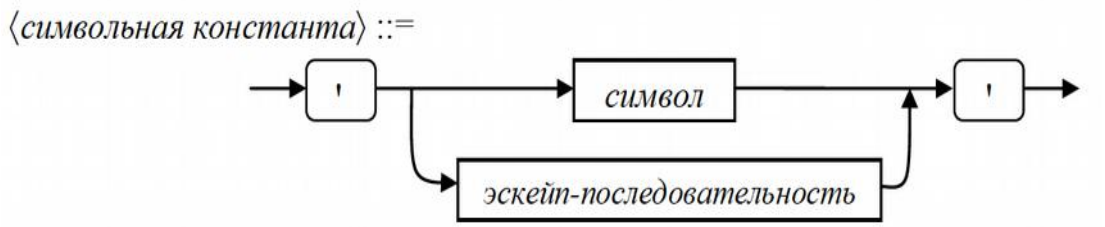
Рассмотрим объекты основных типов данных.

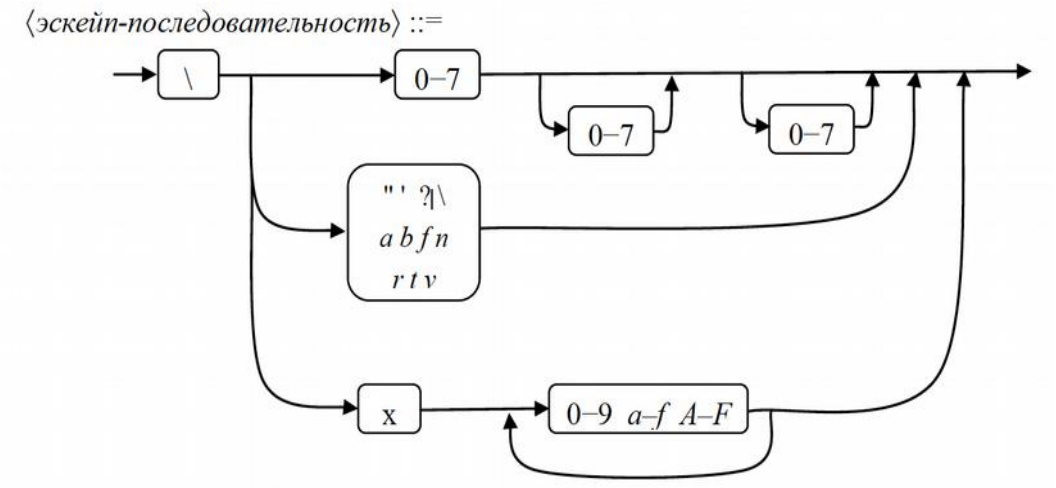
К типу **`char`** относятся

– символы, заключенные в одинарные кавычки (например, `'a'`, `'4'`, `'*'`),

– специальные символы, такие как переход на следующую строку `'\n'`, табуляция `'\t'`.

Введем определение символьной константы с помощью синтаксических диаграмм.

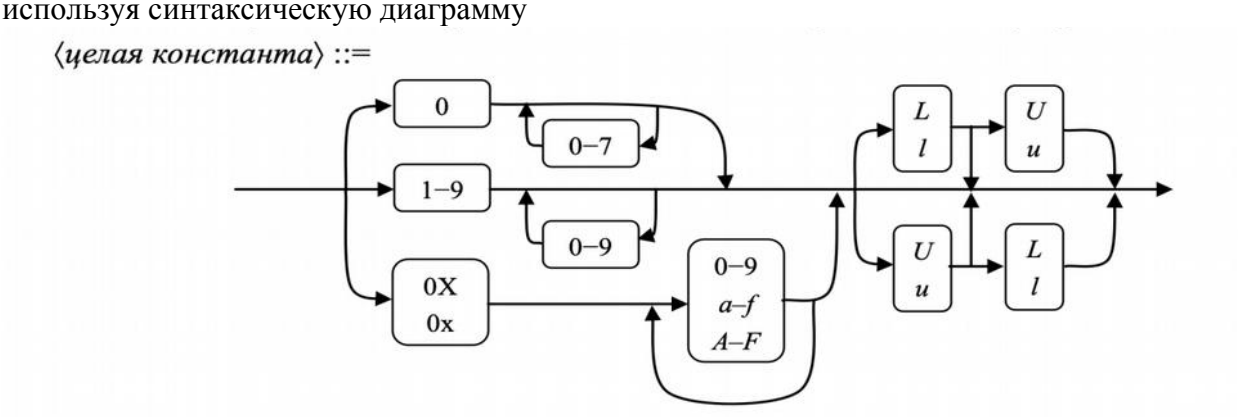




Каждому символу сопоставлен код – целое число, соответствие между символами и кодами зафиксировано в так называемой таблице кодировки, и именно этот код хранится вместо символа в памяти машины Он преобразуется в графическое представление символа при выполнении операций вывода. Кроме задания самого символа в кавычках

можно явно задать его код в десятичной, восьмеричной или шестнадцатеричной системе счисления: например, символ `'*'` может быть представлен как `42`, либо `'\52'`, либо `'\x2A'`. Для некоторых объектов типа char введены специальные обозначения:
`'\n'` – новая строка, `'\a'` – сигнал-звонок, `'\r'` – возврат каретки,
`'\b'` – возврат на позицию назад, `'\t'` – табуляция, `'\f'` – перевод страницы
`'\v'` – вертикальная табуляция.

Объектами **целого типа** являются целые числа. Введем определение целого числа, используя синтаксическую диаграмму

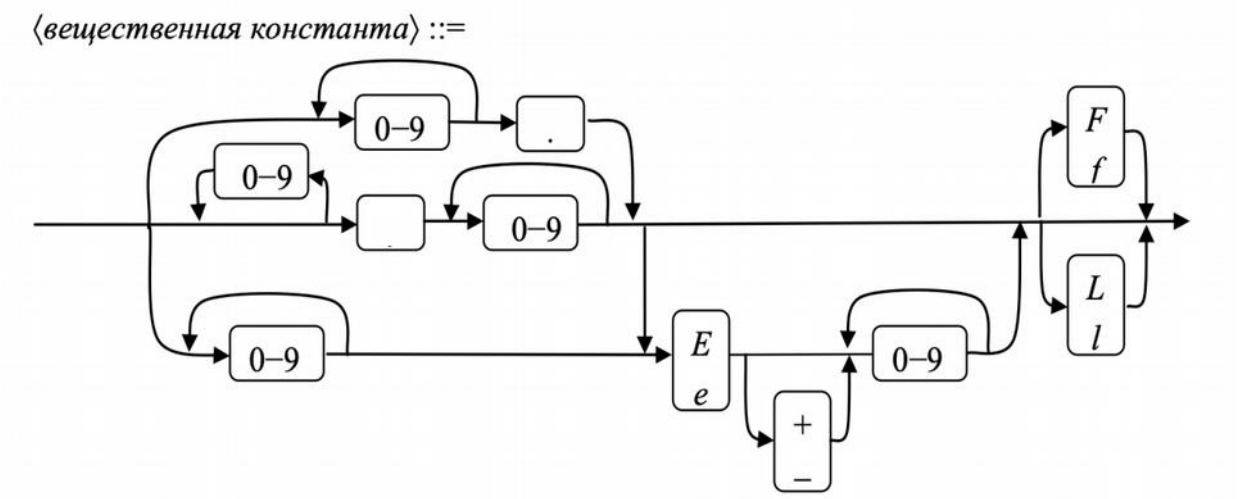


Целочисленные константы могут быть записаны в восьмеричной системе счисления, тогда их запись начинается с `0`, или в шестнадцатеричной, начинающейся с `0x`. Так константы `31`, `037` и `0x1F` представляют собой запись одного и того же числа. Число `0` в языке С является восьмеричным.

Объектами **вещественного типа** являются вещественные (действительные числа). В общем случае в записи вещественного числа указываются три вещи – целая и дробная части числа (мантисса), а также порядок[1]. Например, в записи вещественного числа `314.159E-2` присутствуют `314` – целая часть, `159` – дробная часть, `E-2` –порядок.
В записи вещественного числа необходима либо точка, либо символ `E` (или `e`). Опишем допустимые формы задания вещественных чисел в виде синтаксической диаграммы.

1

 О представлении вещественных чисел в памяти компьютера можно подробнее посмотреть в описании стандарта IEEE 754



В конце записи вещественного числа может быть указан суффикс `F` (либо `f`), указывающий тип константы - `float`, либо суффикс `L` (либо `l`), для указания типа `long double`. Если суффикс не указан, константа представляет собой число типа `double`.

Арифметические и логические операции

Обозначение арифметических операций такое же, как в большинстве языков:

Унарные плюс и минус + и −

Бинарные плюс, минус и умножение + − *

Деление обозначается как /. Если оба операнда целые, то это операция целочисленного деления (деления нацело). Если хотя бы один из операндов – вещественное число, то это операция вещественного деления. Типы операндов можно определить по типу операнда-переменной или по записи операнда-числа. Чтобы сказать, что число является вещественным, можно в его записи использовать точку. Например, `5` – это целое число, а `5.0` – это вещественное число, поэтому

`11 / 5` будет равно `2` (частное для целочисленного деления)

`11.0 / 5` будет равно `2.2` (частное для вещественного деления)

Операция нахождения остатка от целочисленного деления обозначается % (11 % 5 будет равно 1 – остаток целочисленного деления 11 на 5).

Значение выражения 1/2 + 1/3 + 1/4 + … + 1/1024 будет равно нулю, поскольку в этом выражении используется операция целочисленного деления (результат деления единицы на большее число равен нулю для целочисленного деления, сумма нулей – нуль).

Для логических операций используются следующие обозначения:

Сравнения:	> < >= <=
Сравнение на равенство	==
Сравнение на неравенство	!=
Логическое отрицание	!
Конъюнкция	&&
Дизъюнкция	||

В качестве лжи в языке С выступает целочисленное значение `0`. Любое отличное от него целочисленное значение играет роль истины, но если истина получилась в результате вычисления, то она представлена именно единицей.

Задача.

Выполняется ли в Си закон двойного отрицания: !!x == x?

Ответ:

Нет. Например, двойное отрицание значения 3 не равно этому значению

(отрицание 3 – это 0, отрицание нуля – это 1, поэтому `!!3 ≠ 3`)

Операции конъюнкции и дизъюнкции вычисляются слева направо, причем реализована так называемая короткая логика (ленивые вычисления): если по значению первого (левого) операнда уже можно вычислить значение операции, то второй и последующие операнды не вычисляется вовсе. Поэтому логический закон «`x && y` тождественно равно `y && x`» в языке Си не выполняется.

Рассмотрим пример:

```
x != 0 && y == b / x
```

Если при вычислении этого выражения значение переменной `x` не равно нулю, то будет вычисляться второй операнд, в котором есть деление на `x`. Если же значение переменной `x` равно нулю, то второй операнд вычисляться не будет (значение первого операнда – ложь, поэтому значение конъюнкции будет `0` независимо от значения второго операнда, второй операнд не нужно вычислять, результат всей операции конъюнкции уже определён), поэтому ошибки деления на ноль не будет.

Операция присваивания

В языке С присваивание является операцией, в отличие от ряда языков программирования (Паскаль, Фортран и другие), в которые включен оператор присваивания. Синтаксис операции присваивания: `<переменная> = <выражение>`

Сначала выполняется вычисление правого операнда (выражения). В качестве левого операнда рассмотрим имя переменной. В нее будет записан результат вычисления выражения. Присвоенное значение и объявляется результатом всего выражения и может быть использовано в дальнейших вычислениях. Чтобы выражение, содержащее операции присваивания, сделать оператором, нужно после него поставить точку с запятой[2].

```
x = 3;
```

Переменной `x` присваивается значение `3`.

```
y = x = 1;
```

С учетом правой ассоциативности операции присваивания можно поставить опущенные скобки так: «`y = (x = 1);`». Сначала вычисляется правое присваивание – переменной `x` присваивается единица, результат этой операции – тоже `1`, она же присваивается переменной `y`.

```
y = (x = 85 – 1)/2;
```

Вычисление этого выражения происходит следующим образом: сначала нужно вычислить выражение в скобках. Операция присваивания имеет более низкий приоритет, чем операция минус, поэтому сначала вычисляется `85 – 1`, получается результат `84`. Этот результат операции минус присваивается переменной `x`, результат операции присваивания – тоже `84`, он же является результатом вычисления всего подвыражения в скобках. Этот результат используется в операции деления (у неё приоритет тоже больше, чем у присваивания), `84` делится на `2` и получается `42`. Значение `42` – правый операнд внешней операции присваивания, результатом её выполнения будет присваивание переменной `y` значения `42`. В итоге после вычисления этого выражения переменная `x` получит значение `84`, а переменная `y` – значение `42`. Значение `42` будет и результатом

---

2 Это частный случай оператора-выражения, подробнее об операторе-выражении – в материале темы 2

вычисления всего выражения, но оно далее не используется.

Введенные группы операций рассматривались в порядке убывания их приоритетов. Полная информация о приоритетах операций языка Си приведена в таблице 1.

Таблица 1. Приоритет операций и способ их вычисления

операции (в порядке убывания приоритета)	способ вычисления
() [] → . постфиксные **++** и **--**	слева направо (→)
! ~ префиксные **++** и **--** унарные **+ - * & (**тип**) sizeof**	справа налево (←)
*** / %**	слева направо (→)
бинарные **+** −	слева направо (→)
<< >>	слева направо (→)
< <= > >=	слева направо (→)
== !=	слева направо (→)
&	слева направо (→)
^	слева направо (→)
|	слева направо (→)
&&	слева направо (→)
||	слева направо (→)
? :	справа налево (←)
= + = −= *= /= %= &= ^= |= <<= >>=	справа налево (←)
,	слева направо (→)

Задачи

1.1 Верно ли записаны константы, представляющие целочисленные значения? Для верно записанных констант определить их значение, тип.

```
123         1E6             123456789LU     -5              0XFUL
'0'         058             '\x7'           0X-1AD          '\122'
00123       0xffffffL       01A             -'x'            "x"
'a'U        0731UL          '\n'            +0xaf           0X0
```

1.2 Верно ли записаны константы с плавающей точкой? Для верно записанных констант определить их значение, тип.

```
1.71        1E-6        0.314159E1F     .005            0051E-04
5.E+2       0e0         0x1             A1.5            05.5 0
0X1E6       0F          1234.56789L     1.0E-10D        3.1415U
1e-2f       -12.3E-6    +10e6           123456L         E-6
```

1.3 Есть ли разница (с точки зрения языка Си) между числами

a) `100` и `100.0`,

b) `20` и `2E1`

1.4 Указать неправильные записи идентификаторов:

а) `task1.3` б) `test1_1` в) `prog1v1` г) `4you` д) x_1 е) `LXIV`

1.5 Убрать из выражения лишние скобки, т.е. такие, удаление которых не изменит порядок выполнения операций в выражении:

а) `((a*b) / c) % (a+b);`

б) `(-n) % m;`

в) `( z > 0 ) && ( z < 50 );`

г) `(x!=y) || ((a/x)/(b/y) >0)`

1.6 Проверить на компьютере

а) как работает операция `%` с отрицательными операндами
б) сколько байт отводится под типы `char`, `short`, `int`, `long`

1.7 Используя только операции сложения, вычитания и присваивания, поменять местами значения двух целочисленных переменных x и y. Начальными значениями x и y являются положительные целые числа из отрезка [0,60].

1.8 Используя операцию присваивания и необходимые арифметические операции, записать в переменную `t`

а) среднее арифметическое целых чисел x, y и z;

б) сумму квадратов вещественных x и y;

в) квадрат расстояния между точками с координатами (x1, y1) и (x2, y2);

г) объем цилиндра радиуса r высоты h.

1.9. Присвоить целой переменной `h` третью от конца цифру в записи положительного целого числа `k` (например, если `k==130985`, то `h==9`).

1.10. Целой переменной `s` присвоить сумму цифр трехзначного целого числа `k`.

1.11. Идет k-я секунда суток. Определить, сколько полных часов (h) и полных минут (m) прошло к этому моменту (например, h=3 и m=40, если k=13257= 3*3600+40*60+57).

1.12 Записать на языке Си выражение, истинное при выполнении указанного условия и ложное (равное `0`) в противном случае:

а) целое k делится на 7;

б) уравнение $ax^2+bx+c=0$ ($a\neq0$) не имеет вещественных корней;

в) точка (x, y) лежит вне круга радиуса r с центром в точке (1, 0);

г) точка x принадлежит отрезку [–1, 1];

д) $0 < x < 1$;

е) x является наибольшим из x, y, z;

ж) $x \neq \max(x, y, z)$ (операцию ! не использовать);

з) целое `y` делится на 4, но не делится на 100;

и) целое n делится на 7, а целое k делится на 13;

к) целое n делится на 2 или 5.

1.13 Написать эквивалентное выражение, не содержащее операции **!**

```
! ( a>b )            ! ( 2*a == b+4 )          ! ( a<b && c<d )
! ( a<2 || a>5)      ! ( a<1 || b<2 && c<3 )
```

1.14 Используя логические операции и операции сравнения, записать выражение, истинное (равное `1`) при выполнении указанного условия и ложное (равное `0`) в противном случае:

а) x принадлежит отрезку [0, 1];

б) x лежит вне отрезка [0, 1];

в) x принадлежит отрезку [2, 5] или [–1, 1] (операцию ! не использовать);

г) x лежит вне отрезков [2, 5] и [–1, 1];

д) каждое из чисел x, y, z положительно;

е) хотя бы одно из чисел x, y и z положительно;

ж) ни одно из чисел x, y и z не является положительным;

з) только одно из чисел x, y и z положительно;

и) год с порядковым номером y является високосным (год високосный, если его номер кратен 4, однако из кратных 100 високосными являются лишь кратные 400; например, 1700, 1800, 1900 и 2001 – невисокосные годы, а 1600 и 2000 – високосные).

1.15 Нарисовать на плоскости (x, y) область, в которой и только в которой истинно указанное логическое выражение:

а) `y>=x && y+x>=0 && y<=1;`

б) `x * x + y * y < 1 || y > 0 && fabs(x) <=1;`

fabs – функция из библиотеки math, возвращающая модуль своего аргумента. Для подключения этой библиотеки необходимо в начале программы указать директиву компилятору

```
#include <math.h>
```

При компиляции такой программы в операционной системе Unix необходимо указать флаг *–lm*, например

```
gcc f.c –ansi –o f -lm
```

в) `(fabs(x)<=1) > (fabs(y) >= 1);`

г) `(x * x + y * y <=4) == (y <= x);`

д) `(fabs(x) + fabs(y)<=1) == (x * x + y * y > 4);`

1.16 Записать на Си выражение, зависящее от x и y, которое принимает значение true, только если точка с координатами (x,y) принадлежит заштрихованной области (см. рис. 1). (Пунктирная граница означает, что она не входит в область.)

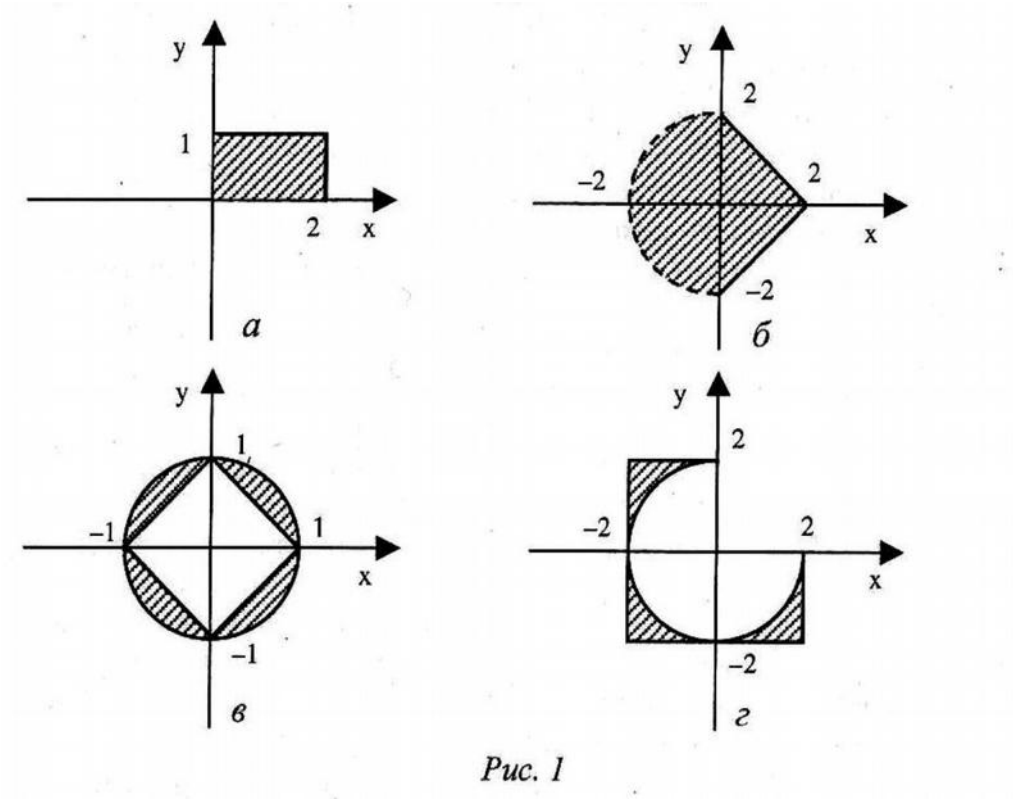


Рис. 1

1.17 Выписать оператор присваивания, в результате выполнения которого переменная `t` получает значение 1 (истина), если выполняется указанное условие, и значение 0 (ложь) иначе:

а) число x положительно;

б) целое число k нечетно;

в) p делится нацело на q (p и q – натуральные числа);

г) числа x, y, z равны между собой;

д) из чисел x, y, z только два равны между собой;

е) уравнение $ax^2+bx+c=0$, где a, b и c могут равняться 0, имеет ровно один корень;

ж) цифра 5 входит в десятичную запись трехзначного целого числа k

Тема 2

Оператор-выражение, операции с побочным эффектом, точки последовательных вычислений. Составной оператор, условный оператор.

Одним из наиболее часто используемых операторов в программах на Си является **оператор-выражение**. Оператор-выражение получается из выражения, после которого поставлена точка с запятой.

Несмотря на то, что `;` может превратить в оператор любое выражение (в том числе `1+2; 44;`) наибольший интерес представляют **выражения с побочным эффектом**, то есть такие, при вычислении которых есть не только результат вычислений, но и меняется состояние памяти, либо происходит запись информации в поток вывода, либо считывание данных. Ранее был рассмотрен пример операции с побочным эффектом – присваивание. Так у выражения `x=0` нам важен не столько его результат, сколько запись `0` в переменную `x`. В дополнение к операции присваивания = язык Си позволяет записывать целую группу операций присваивания вида `op=`, где `op` – арифметическая операция: `+`, `-`, `*`, `/`, `%`.

Операции группы присваивания `X op= E` сначала вычисляют выражение `E`, затем выполняют операцию `op` над операндами – значением переменной `X` и результатом вычисления выражения `E`, после чего присваивают полученный результат в `X`. Результатом такой операции считается присвоенное значение.

Приведем пример использования операций присваивания. Пусть в переменной `x` находится значение `8`. После выполнения `x += 2` значением переменной `x` станет `10`. Если в одном выражении встречаются несколько операций группы присваивания, они выполняются справа налево: при вычислении выражения `a += b += c = 2` сначала в переменную `c` будет записано значение `2`, затем исходное значение переменной `b` увеличится на `2`, затем к исходному значению переменной `a` будет добавлено (`b+2`).

Рассмотрим и другие операции с побочным эффектом.

Операции `++` и `--`.

Префиксная операция `++` имеет один операнд-переменную (ячейку памяти), указывается перед ней, например, `++x` – увеличивает значение переменной `x` на `1`, результат её вычисления – новое увеличенное значение переменной. Если переменная `x` имела значение `5`, то результат вычисления `++x` будет `6`, побочный эффект – значение переменной x тоже станет 6. Аналогичным образом вычисляется префиксная операция `--`, только при её вычислении происходит уменьшение значения переменной на `1`. Например, если значение переменной `x` было `5`, то значением `--x` будет `4`, побочный эффект – значение переменной `x` станет `4`.

Постфиксные `++` и `--` имеют те же самые побочные эффекты, но результат их вычисления другой, чем у аналогичных префиксных операций. Например, если переменная `x` имела значение `5`, то результатом вычисления `x++` будет `5` ("старое" значение `x`), а побочный эффект – значение переменной `x` станет `6`. Если переменная `x` имела значение `5`, то результат выражения `x--` равен значению `5`, а побочный эффект вычисления этой операции – у переменной `x` значение станет равным `4`.

Побочный эффект – изменение значения переменной – может быть произведён не сразу, а отложен на некоторое время. При вычислении выражений есть так называемые точки последовательных вычислений. В такой точке гарантируется, что все присваивания и другие побочные эффекты, которые встретились в выражении до нее, будут выполнены.

Точек последовательных вычислений (sequence points) всего пять:

1. **Конец выражения**.
   При завершении вычисления выражения все побочные эффекты, накопившиеся к соответствующему моменту, будут гарантированно произведены.

2. **Логические операции `&&` и `||`**.
   После вычисления левого операнда и до вычисления правого будут произведены все побочные эффекты, которые возникли при вычислении левого операнда.
3. **Точка в тернарной условной операции `?:`**.
   Тернарная операция `?:` имеет следующий формат: `E1 ? E2 : E3`.
   Сначала вычисляется условие – выражение `E1`. Если его значение отлично от нуля, то значением всего выражения будет значение выражения `E2` (`E3` при этом не вычисляется), в противном случае – значение выражения `E3` (`E2` в этом случае не вычисляется). Вопрос `?` является точкой последовательных вычислений – после вычисления `E1` (и до вычисления `E2` или `E3`) гарантированно будут произведены все побочные эффекты, которые появились при вычислении выражения `E1`.
4. **Точка последовательных вычислений в операции `,`** (операция запятая)
   Операция имеет следующий синтаксис `E1,E2` (где `E1` и `E2` - выражения) . Она позволяет из нескольких выражений сделать одно. Вычисление сводится к последовательному вычислению сначала E1, затем E2. Результат E1 никуда не записывается, а результатом всего выражения объявляется результат E2 (правого операнда). После вычисления левого операнда и перед вычислением правого гарантируется, что все присваивания в левом операнде будут произведены.
   Приведем пример: в выражении `(x=2, y = x*10)` сначала будет вычислено `x=2`. При переходе через операцию `,` мы можем быть уверены, что изменение ячейки памяти `x` точно произошло, и в подвыражении `y = x*10` уже используется значение x=2. Результатом выражения будет `20`.
5. **Точка входа в функцию**. К моменту входа в функцию гарантируется, что все побочные эффекты, содержащиеся в параметрах, будут выполнены.

Если между двумя точками последовательных вычислений одна и та же переменная меняется дважды (либо меняется и используется), то результат вычисления такого выражения не определён. Такие выражения будем считать ошибочными.

Рассмотрим еще несколько примеров выражений.

```
(a = 2) + a
```

Это «плохое» выражение – переменная `a` меняется операцией присваивания и используется между точками последовательных вычислений. Поскольку порядок вычисления аргументов сложения не фиксирован, точек последовательных вычислений внутри этого выражения нет, то совершенно не понятно, какое значение переменной `a` будет у правого аргумента сложения: старое, которое было до присваивания 2, или уже новое, равное числу 2. Поэтому результат вычисления такого выражения не определён.

```
(a = 1) + (b = 85)
```

«Хорошее» выражение – каждая переменная используется только один раз, хоть и в операциях с побочным эффектом. Результат вычисления всего выражения – 86, побочные эффекты после вычисления этого выражения – переменная `a` получит значение 1, переменная `b` получит значение 85.

```
(a = 2) && (b = a + 4);
```

Выражение «хорошее». Сначала вычисляется левый операнд конъюнкции, результат вычисления левого операнда – значение `2`, оно же будет записано в переменную `a`. До вычисления правого операнда побочный эффект присваивания точно уже будет выполнен, так как мы проходим через операцию `&&`, а в ней есть точка последовательных вычислений. Поэтому переменной `b` присвоится значение `6`. Результат вычисления всего

выражения – `2 && 6` – будет равен `1` (два отличных от нуля значения – оба трактуются как истина, результат конъюнкции – истина, в С логическая операция в этом случае возвращает `1`).

```
x = 4; y = 7;
(z = x > y)? z += x-- : z -= ++y;
```

Значение последнего выражения зависит только от значений переменных `x` и `y` – несмотря на то, что переменная `z` входит в выражение три раза, причём является операндом операций с побочным эффектом. Выражение содержит точку последовательных вычислений – `?`. Вычисление этого выражения происходит следующим образом: сначала вычисляется условие (то, что находится перед символом `?`). Значение подвыражения `x > y` ложно при заданных выше значениях переменных (x=4, y=7), поэтому операция `>` вернёт значение `0`, оно будет записано в переменную `z` и будет результатом этой операции присваивания. Поскольку условие ложно, далее будет вычисляться третья часть выражения, записанная после двоеточия. Но к моменту вычисления этого подвыражения побочный эффект операции присваивания переменной `z` значения `0` уже гарантированно будет выполнен, т.е. переменная `z` уже будет иметь значение `0`. Вычисление выражения

```
z -= ++y
```

начнётся с вычисления операции ++, имеющей более высокий приоритет, чем операции присваивания. `++y` вернёт `8`, побочный эффект – переменная `y` получит значение `8` (старое значение увеличится на единицу), далее операция `-=` уменьшит значение переменной `z` на `8`, т.е. значение `z` станет `-8`. Второе подвыражение тернарной операции `z += x--` при заданных значениях переменных `x` и `y` не будет вычисляться, поэтому содержащийся в нём побочный эффект уменьшения значения переменной `x` не будет выполнен.

```
printf("%d %d %d", ++a, ++a, ++a);
```

Это «плохое» выражение, поскольку порядок вычисления аргументов функции не фиксирован, точек последовательных вычислений в этом выражении нет, какое значение переменной `a` будет при вычислении каждого из аргументов – не определено. Если переписать его следующим образом:

```
x = ++a; y = ++a; z = ++a;
printf("%d %d %d", x, y, z);
```

то проблема будет решена – до вычисления функции `printf` все побочные эффекты – изменения значения переменной `a` и присваивания переменным `x`, `y` и `z` этих значений – будут выполнены.

Пустой, составной и условный операторы

Пустой оператор – это одиночная точка с запятой `;` Такой оператор ничего не делает, но может быть использован там, где по синтаксису языка требуется указать какой-либо оператор, а делать ничего не нужно.

Если согласно синтаксису языка нужно написать один оператор, а необходимо выполнить последовательность действий, то несколько операторов можно объединить в один, используя составной оператор. **Составной оператор** – это несколько операторов, заключенных в фигурные скобки `{ }`.

Условный оператор описывается следующим образом:

```
if (условие) Op1 else Op2
```

После `if` записывается условие – условное выражение, которое обязательно должно быть заключено в скобки. Если условие выполняется (значение выражения

отлично от нуля – то есть истинно с точки зрения языка С), то выполняется оператор `Op1`. Если значение условного выражения равно нулю (ложь с точки зрения языка С), то выполняется `Op2`. Часть условного оператора `else Op2` может отсутствовать – в этом случае при ложном условии сразу делается переход на следующее действие в программе.

Оператор `Op1` или `Op2` может быть составным, если нужно сделать несколько действий.

Пример программы с условным и составным операторами:

```
#include <stdio.h>
int main()
{  int x = 1, y;
   unsigned a;
   x = a = 5;
   if (x == 1)
   {
       x = 0;
       y = 1;
    }
   else
      printf("No");
   return 0;
}
```

Задачи

2.1 Напишите программу, решающую следующую задачу. В переменные x, y, z присвоены некоторые значения. Переупорядочить эти значения так, чтобы в x находилось наибольшее значение, в z — наименьшее из трех, а промежуточное — в y (x>y>z). Вывести в стандартный вывод новые значения x, y, z.

2.2 Написать программу для решения следующей задачи. В переменные a, b, c присвоить три числа. Найти корни квадратного уравнения с заданными коэффициентами a, b и c. Использовать функцию вычисления квадратного корня – `sqrt(аргумент)` из библиотеки `math`. Печать вещественного числа x – `printf("%lf\n", x)`

2.3. Верно ли решена задача:
«значение целочисленной переменной `c` увеличить на 1; целочисленной переменной `a` присвоить значение, равное удвоенному значению переменной `c` »?

```
int a, c; c = 5;
а) c ++ ;          б) a = 2 * c++ ;  в) c += 1;
   a = 2 * c;                            a = c + c;

г) a = c++ + c;          д)++c;            е) a = ++ c + c;
                           a = c + c;

ж)a = c += 1 + c; з)a = (c+=1)+c
```

2.4 Верно ли решена задача:
«значение целочисленной переменной `c` уменьшить на 1; целочисленной переменной `a` присвоить значение, равное частному от деления переменной `c` на 2»?

```
int a, c; c = 5;

а) -- c ;               б) a = -- c / 2;     в) c -= 1;
   a = c / 2;                                      a = c % 2;

г) a = c -- / 2;              д) a = c -= 1/2;

е)  a = (c = c - 1)/2;        ж) a = (c -= 1)/2;
```

2.5 Верно ли записаны выражения? Если да, то вычислить их и написать, какими станут значения

переменных a, b, c . Если выражение записано неверно, объяснить, в чём ошибка.

а)
```
int a, b, c;
a = c = 1;
a = a++;
b = (c || 2) % (c && 2);
```

б)
```
int a, b, c;
a = b = 7;
c = a++ <= --b || ++b >= 1 && 3.7;
```

в)
```
int a, b, c;
a = b = c = 3;
a += b += c;
```

г)
```
int a, b, c;
a = b = 1;
c = ++a || ++b &&++a;
a = 2.81 && 1 / 8;
```

2.6 Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" – объясните почему.

а)
```
int x;
x = 5;
++ x =10;
printf ("%d\n", x);
```

б)
```
int x, y;
x = 5;
y = x && ++ x;
printf("%d %d\n", x, y);
```

в)
```
int x = 2, y = 1, z = 0;
y = x && y || z;
x = x || !y && z;
z = x / ++x;
printf(" %d %d %d\n", x, y, z);
```

г)
```
int x, y, z; x = y = z = 1;
x += y += z;
printf("%d\n", x < y ? y++ : x++);
printf("%d\n", z+=x<y ? ++x : y--);
printf("%d %d %d\n", x, y, z);
printf("%d\n", z>=y && y>=x);
```

д)
```
int x, y, z, i; x = y = z = 1;
i = ++x || ++y && ++z;
printf("%d%d%d%d\n", x,y,z,i);
i = x++ <= --y || ++z >= i;
printf("%d%d%d%d\n", x,y,z,i);
```

е)
```
int i, x, y;
x = 5; y = 10; i = 15;
x = ( y = 0, i = 1);
printf("%d %d %d\n", i, x ,y);
(x = y == 0) , i=1;
printf("%d %d %d\n", i, x, y);
```

ж)
```
int x = 2, y, z;
x *= 3+2; x *= y = z = 4;
printf ("%d %d %d\n", x, y, z);
x = y == z;
```

```
x == ( y = z );
printf ("%d %d %d\n", x, y, z);
```

2.7 Для каких значений a и b результаты выражений будут различны? Указать хотя бы одну пару таких a и b, и вычислить выражения при этих значениях a и b. Какие значения будут в a и b после вычисления каждого из выражений?

Выражение 1 : `a ++ && (b = a)` Выражение 2 : `++a && (b = a)`

2.8 Определить, какое значение будет иметь переменная x после выполнения операторов:

а) `if (x>0) x = 1; else if (y>0) x = 2; else x = 3;`
б) `if (x>0) x = 1; if (y>0) x = 2; else x = 3;`
в) `if (x>0) x = 1; else if (y>0) x = 2;`
г) `if (x>0) x = 1; if (y>0) x = 2;`
д) `if (x>0) if (y>0) x = 1; else x = 2; else x = 3;`
е) `if (x>0) if (y>0) x = 1; else x = 2;`
ж) `if (x>0){if (y>0) x = 1;} else x = 2;`

если вначале переменным x и y присвоены следующие значения:

1) `x=4, y=4;` 2) `x=-4, y=4;` 3) `x=4, y=-4;` 4) `x=-4, y=-4`

2.9 Известно, что из четырех чисел a_1, a_2, a_3 и a_4 одно отлично от трех других, равных между собой; присвоить номер этого числа переменной *n*. Записать указанное действие в виде одного условного оператора.

2.10 Написать программу для решения указанной задачи.

В переменные a, b, c присвоены три вещественных числа. Если нельзя построить треугольник с такими длинами сторон, то вывести 0, иначе вывести числа 3, 2 или 1 в зависимости от того, равносторонний это треугольник, равнобедренный или какой-либо иной.

2.11 Пусть в три вещественные переменные r1 , x1 , y1 присвоены радиус и координаты центра первой окружности, а в переменные r2, x2, y2 – второй окружности. Определить

a) являются ли эти окружности концентрическими,

b) является ли первая окружность вложенной во вторую.

2.12 Написать **фрагмент программы** «умного ошейника» кота. Ошейник замеряет три показателя: уровень голода (hungry – целое число от 0 до 100), уровень сонливости (sleepy — целое число от 0 до 100), наличие человека рядом (human — переменная типа int: 1 — человек дома, 0 — нет). Определить и вывести в стандартный вывод текущее настроение кота по заданным значениям трех переменных согласно правилам:

1. «Кот в ярости»: Если уровень голода строго больше 80, кот никого не слушает и требует еду. Это самое приоритетное условие.

2. «Кот спит»: Если голод в норме (80 и меньше), но уровень сонливости больше 70, кот уходит спать.

3. «Кот готов бегать»: Если кот не голоден и не хочет спать, но человека нет дома (0), ему скучно, и он начинает бегать по квартире.

4. «Кот мурчит»: Если кот не голоден, не хочет спать, и человек есть дома (1), кот приходит мурчать.

2.13 Написать **фрагмент программы** контроллера для «умного аквариума». В переменные temp, filter и light присвоены значения с трех датчиков: температуры (в °C), загрязнения фильтра (% загрязнения от 0 до 100) и освещения (1 — свет включен, 0 — выключен). Требуется проанализировать эти данные и вывести в стандартный вывод отчет согласно правилам:

1. «Критическая опасность!»: Если температура воды упала ниже 18°C или поднялась выше 30°C, рыбки могут погибнуть. Это самое важное условие.
2. «Нужна чистка!»: Если температура в норме, но фильтр загрязнен более чем на 80%, вода становится мутной.
3. «Рыбки спят»: Если температура и фильтр в норме (<=80), свет в аквариуме выключен.
4. «Все отлично»: Если температура в норме, фильтр чистый, а свет включен, рыбки весело плавают.

Тема 3.

Операторы цикла, операторы break и continue .

Ввод и вывод символов.

Операторы цикла

В языке С существует три вида операторов цикла. Рассмотрим синтаксис и семантику выполнения каждого из них.

Цикл while выглядит следующим образом:

```
while (E)
   S
```

После `while` в скобках записывается выражение `E`, которое определяет, будет ли выполняться тело цикла – оператор `S`. Если в цикле нужно выполнить несколько операторов, то необходимо из них сделать один составной оператор с помощью `{}`. Сначала вычисляется выражение `E`. Если вычисление этого выражения вернуло значение, отличное от нуля (`0` – это ложь в языке С, всё остальное – истина), то вычисляется тело цикла – оператор `S`. После этого снова вычисляется выражение `E`, и если оно истинно, то вычисляется тело цикла `S`, и т.д. Если же вычисление выражения `E` дало `0`, то условие цикла стало ложно, в этом случае переходим на следующий после цикла оператор.

Цикл do-while записывается так:

```
do
   S
while (E);
```

Цикл `do-while` вычисляется почти так же, как и цикл `while`. Только в самом начале вычисляется тело цикла `S`, а затем проверяется условие продолжения цикла – выражение `E`. Если значение этого выражения отлично от нуля, то опять выполняется тело цикла `S`, а затем проверяется условие продолжения цикла `E`. Если же вычисление выражения `E` дало ноль, то происходит переход на следующий после цикла оператор.

Цикл for имеет следующий синтаксис:

```
for(E1; E2; E3)
   S
```

Назовём `E1` выражением инициализации. Оно вычисляется ровно один раз в самом начале, в нём часто переменным, используемым в цикле, присваиваются начальные значения. Затем вычисляется выражение `E2` – условие выполнения тела цикла `S`. Если выражение `E2` дало значение, отличное от нуля (истину), то выполняется тело цикла `S`. После его выполнения вычисляется выражение `E3`, в котором часто содержатся изменения переменных цикла – те вычисления, которые необходимо выполнить до перехода к следующей итерации цикла. Дальше вычисляется выражение `E2`, и если оно истинно, то опять вычисляется тело цикла `S` и т.д. Если же вычисление выражения `E2` дало значение `0`, то происходит переход на следующий за циклом оператор. Любое из трёх выражений `E1`, `E2` и `E3` может отсутствовать, но разделяющие их точки с запятой не могут быть опущены при записи цикла `for`. В случае отсутствия `E1` не происходит инициализация переменных, но тогда инициализация переменных начальными значениями может быть сделана, например, перед циклом. Если отсутствует выражение `E2`, то условие

выполнения цикла считается истинным всегда. Если отсутствует выражение E3, то после выполнения тела цикла ничего не делается – нет изменений значений переменных. Но переменные могут изменяться в самом теле цикла, в языке С их менять не запрещено.

Цикл for без начальной инициализации переменных записывается следующим образом:

```
for(; E2; E3)  S
```

Цикл без начальной инициализации переменных и без изменения переменных:

```
for(; E2; ) S
```

Такой цикл эквивалентен циклу while:

```
while (E2) S
```

Цикл, в котором отсутствуют все три выражения:

```
for(;;) S
```

по сути является бесконечным циклом, эквивалентным такому циклу:

```
while (1) S
```

Условие выполнения таких циклов всегда истинно, поэтому закончиться они могут только тогда, когда в их теле выполнится оператор break; он осуществляет выход из ближайшего объемлющего цикла.

Примеры с циклом for:

```
N = 10;
for(i = 0; i<N; ++i)
   {
      printf("*");
      N--;
   }
```

Сколько звёздочек напечатается в результате выполнения этого фрагмента кода? Считаем, что переменные i и N описаны раньше в программе. Для определения количества * нужно отследить изменения переменных i и N после печати каждой звёздочки, т.е. после каждого выполнения тела цикла. До начала выполнения цикла i=0 (инициализация переменной i в выражении E1 до первой точки с запятой), N=10 – инициализирована перед циклом.

Условие цикла i<N выполняется (0<10), печатается первая звёздочка. После её печати выполняется вычисление выражения N-- в теле цикла и выражения ++i (в описании синтаксиса цикла это выражение называлось E3 – после второй точки с запятой). Значения переменных становятся соответственно i=1 и N=9.

Сравнение 1<9 дает истину, поэтому тело цикла и затем вычисление выражения ++i выполнятся ещё один раз, напечатается вторая звёздочка, значения переменных станут i=2 и N=8.

В условии цикла 2<8 – истина, поэтому действия будут выполнены ещё раз – печать третьей звёздочки, значения переменных станут i=3 и N=7.

3<7, поэтому ещё раз выполняем все действия – печать четвёртой звёздочки, значения переменных станут i=4 и N=6.

4<6, ещё одно выполнение цикла – печать пятой звёздочки, значения переменных станут i=5 и N=5.

Условие i<N при таких значениях переменных уже не выполняется, поэтому цикл больше не станет выполняться. Таким образом, в результате работы этого цикла for напечатается всего пять звёздочек.

Инициализацию переменной N можно было сделать прямо в выражении E1 цикла

`for` – чтобы инициализировать несколько переменных, нужно сделать из нескольких операций присваивания одно выражение с помощью операции запятая. Тогда цикл будет выглядеть так:

```
for(N = 10, i = 0; i<N; ++i)
   {
      printf("*");
      N--;
   }
```

В общем случае может быть несколько переменных цикла, и они могут изменяться в выражении `E3` произвольным образом. Например, если нам нужен цикл со счётчиком i и шагом 2, то соответствующие выражения в скобках после for будут выглядеть так:

```
for( i = 0; i<N; i += 2)
```

В теле цикла может также встретиться оператор `continue;` — переход на следующую итерацию цикла. В циклах `while` и `do-while` он передает управление на вычисление условия цикла, в цикле `for(E1;E2;E3)...` выполняет переход на вычисление `E3`, далее – на проверку условия продолжения цикла (`E2`). В случае выполнения оператора `continue;` часть операторов в теле цикла может быть пропущена. Например:

```
for(N = 10, i = 0; i < N; ++i)
   {  printf("Hello \n");
      continue;
      i++;
   }
```

Цикл напечатает 10 строк: счётчик `i` увеличивается только в заголовке цикла `for` (при вычислении `++i)`, а не в теле. Причина в том, что `continue` досрочно завершает текущую итерацию и передаёт управление на вычисление выражения `E3` (в данном цикле `++i`), а затем на проверку условия (выражение `E2`). Обычно `break` и `continue` используются в условном операторе, который входит в тело цикла: если условие истинно, происходит либо выход из цикла (`break`), либо переход к новой итерации (`continue`).

Символьный ввод и вывод

Для считывания очередного символа используют функцию `getchar()` – она возвращает очередной символ из входного потока. Под переменную символьного типа отводится 1 байт памяти, и его достаточно для хранения кодов всех символов. Однако при вводе может возникнуть ситуация «конец файла» – `EOF`, когда входной поток программы закрывается, и дальнейший ввод из него будет невозможен. В этом случае функция чтения символа должна вернуть значение, отличное от кодов всех известных символов. Поэтому для результата функции нужна ячейка памяти больше, чем char, — результат необходимо записывать в `int`. Значение, возвращаемое функцией `getchar()`, необходимо присвоить переменной типа `int`.

```
int c;
c = getchar();
```

Константа `EOF` будет результатом функции чтения символа со стандартного ввода, если на клавиатуре была нажата комбинация клавиш Ctrl + D (в ОС Unix) или Ctrl + Z (в ОС Windows). Указанную комбинацию клавиш нужно нажать дважды, если мы находимся не в первой позиции вводимой строки. Значение константы `EOF` равно `-1`, но

рекомендуется в программе использовать её имя, а не непосредственно указывать `-1`. Код любого успешно прочитанного символа будет целым неотрицательным числом из диапазона байта (от 0 до 255).

Для вывода символа в выходной поток используют функцию `putchar(аргумент)`, где аргумент – это явно заданный в одинарных кавычках символ (например, `'a'` или `'\n'`) или имя переменной, значением которой является выводимый символ, либо целое число, задающее код символа.

Пример. Прочитать со стандартного входного потока последовательность символов (до возникновения `EOF`), посчитать и вывести количество прочитанных цифр.

```
#include <stdio.h>
int main()
{  int c, count = 0;
    /* c – для ввода символа, count – для подсчета цифр */
   while ((c = getchar()) != EOF)
      if (c >= '0' && c <= '9')   count += 1;
   printf("%d\n", count);
   return 0;
}
```

Если сигналом окончания считывания символов могут быть несколько констант (например, если считывание до конца файла, то есть до получения `EOF`, или до конца строки – символа `'\n'`), то условие цикла `while` будет более сложным:

```
while ((c = getchar()) != EOF && c != '\n')
```

Важно помнить, что обращение к функции `getchar` в условии должно быть только одно, чтобы считывался и проверялся только один символ. Поскольку логическая операция `&&` является точкой последовательных вычислений, то во втором операнде конъюнкции `c!='\n'` в переменной `c` уже будет находиться прочитанный символ.

Задачи

3.1 Верно ли решена задача: «найти сумму первых 100 натуральных чисел»?

```
а) i = 1; sum = 0;
    for ( ; i < 100; i++) sum += i;
б) sum = 1;
    for ( i = 1; i <100;) sum += ++ i;
в) for ( i = 1, sum = 0; i++ <= 100; sum += i++);
г) for ( i = 1, sum = 0; i <= 100; sum += i++);
д) for ( i = 0, sum = 0; i++, i <= 100; sum += i);
```

3.2 Выразить семантику цикла `for(E1;E2;E3)S` с помощью цикла `while`. При любых ли `E1,E2,E3,S` полученные фрагменты программ эквивалентны?

3.3 Улучшить стиль (структуру) фрагмента программы на Си

а) Записать без оператора `continue`:

```
while ( E1 ) {
    for (E2; E3; E4) {
      S1
      continue;
      S2
    }
}
```

б) Записать без оператора `break`:

```
while ( E1 ) {
    for (E2; E3; E4) {
      S1
      break;
      S2
    }
}
```

Замечание: здесь E_1,E_2,E_3,E_4 — выражения допустимого в этом случае типа;

S_1,S_2 — операторы, не являющиеся операторами перехода и не содержащие внутри себя операторов перехода.

3.4 Подсчитать количество натуральных чисел `n (111 <= n <= 999)`, в записи которых есть две одинаковые цифры.

3.5 Подсчитать количество натуральных чисел `n (102 <= n <= 987)`, в которых все три цифры различны.

3.6 Подсчитать количество натуральных чисел `n (11 <= n <= 999)`, являющихся палиндромами, и распечатать их.

3.7 Написать фрагмент программы вычисления `f = 10!` с помощью каждого из трех операторов цикла.

3.8 Можно ли написать фрагмент программы на Си, эквивалентный данному, используя один оператор цикла `for` с пустым оператором в качестве тела цикла?

```
i = 0; c = getchar();
while (c != ' ' && c != '\n' && c !=' \t' && c != EOF){
  i++;
  c = getchar();
}
```

3.9 Дан фрагмент программы на языке С:

```
int i,  N  = 20;
for (i = 0; i < N; i--)
  printf("*");
```

Изменив ровно один символ (можно вставить в любую позицию в тексте один символ, либо стереть один символ, либо один из символов текста программы заменить другим), сделать фрагмент таким, чтобы при запуске программы символ * напечатался с помощью цикла ровно 20 раз.

3.10 Для данного вещественного числа x и натурального n вычислить:

а) $\sin x + \sin^2 x + \ldots + \sin^n x$

б) $\sin x + \sin(x^2) + \ldots + \sin(x^n)$

в) $\sin x + \sin(\sin x) + \ldots + \underbrace{\sin(\ldots \sin}_{n \text{ синусов}}(\sin x) \ldots)$

3.11 Вычислить 1 - 1/2 + 1/3 - 1/4 + ...+1/9999 - 1/10000 следующими способами:

а) последовательно слева направо;

б) последовательно справа налево;

в) последовательно слева направо вычисляются 1 +1/3 + 1/5 + ... + 1/9999 и 1/2 + 1/4 + ... + 1/10000, затем второе значение вычитается из первого;

г) последовательно справа налево вычисляются 1 +1/3 + 1/5 + ... + 1/9999 и 1/2 + 1/4 + ... + 1/10000, затем второе значение вычитается из первого.

Для хранения результата использовать переменную типа `float`.

Сравнить и объяснить полученные результаты.

В следующих задачах 3 главы текстом будем считать последовательность символов, оканчивающуюся переводом строки (`'\n'`) либо концом файла, если в условии задачи не сказано иное.

3.12 Вывести строку `true`, если в заданном тексте буква `'a'` встречается чаще, чем буква `'b'`, и строку `false` иначе. Использовать для решения задачи

а) цикл `while` б) цикл `for`

3.13 Написать программу, проверяющую, правильно ли в заданном тексте расставлены круглые скобки. Считать, что скобки расставлены правильно, если справа от каждой открывающей скобки `'('` находится соответствующая ей закрывающая скобка `')'`, а слева от каждой закрывающей – соответствующая ей открывающая. Ответ – строка `"yes"` или `"no"`.

3.14 Написать программу, которая посимвольно считывает десятичные цифры со стандартного ввода (прекращая чтение, если встретился символ отличный от цифры) и формирует из них целое число. Накапливать число по мере чтения символов, используя схему Горнера.

3.15 Написать программу, которая посимвольно считывает шестнадцатеричные цифры '0'...'9','A'...'F' со стандартного ввода (прекращая чтение, если встретился символ отличный от шестнадцатеричной цифры) и формирует из них целое число в переменной `Num` типа `int`.

3.16 Написать программу, которая выведет на экран таблицу следующего вида. Для вывода использовать только функцию putchar

а)	б)	в)
100...00	999...99	0123456789
020...00	088...88	1234567890
…	…	…
000...09	000...01	9012345678

3.17 Написать программу, которая подсчитывает, сколько раз пара символов `th` входит в заданный текст.

3.18 Написать программу, которая посимвольно вводит со стандартного ввода текст и выводит его в стандартный вывод, выполнив заданное преобразование.

Замечание. *Поскольку ввод символов с входного потока происходит с участием буфера, содержимое которого выталкивается в программу после нажатия Ctrl + D или Enter, то вывод результирующей строки (её печать) произойдёт только после окончания ввода исходной. Символы вводимой и выводимой строки чередоваться на экране не будут.*

Варианты преобразований текста:

а) заменить все символы `'?'` на `'!'`;

б) удалить из текста все цифры и удвоить знаки `'+'` и `'-'`;

в) удалить все символы, не являющиеся строчными латинскими буквами;

г) удалить все знаки `'+'`, непосредственно за которыми идет цифра;

д) удалить все буквы `'b'`, непосредственно перед которыми находится буква `'c'`;

е) заменить все пары `'p''h'` на букву `'f'`;

ж) заменить все прописные латинские буквы строчными (другие символы копировать в выходной поток без изменения);

з) заменить все строчные латинские буквы прописными (другие символы копировать в выходной поток без изменения);

и) каждую группу рядом стоящих символов `'+'` заменить одним таким символом;

к) каждую группу из n рядом стоящих символов `'_'` заменить группой из n/2 рядом стоящих символов `'+'` (n >= 2); одиночные `'_'` копировать в выходной поток без изменения;

л) удалить из каждой группы подряд идущих цифр все начальные незначащие нули (если группа состоит только из нулей, то заменить эту группу одним нулём);

м) удалить все комбинации символов `the`;

н) оставить только те группы цифр, которые составлены из подряд идущих цифр с возрастающими значениями; все остальные цифры и группы цифр удалить (другие символы копировать в выходной поток без изменения);

о) заменить все комбинации символов `child` комбинациями символов `children`;

п) удалить группы символов, расположенные между фигурными скобками `{` и `}`. Скобки тоже должны быть удалены. Предполагается, что скобки сбалансированы, и внутри каждой пары скобок других фигурных скобок нет.

3.19 Написать программу, решающую следующую задачу. В непустой текст входят только цифры и большие латинские буквы. Определить, удовлетворяет ли он следующему свойству:

а) текст является десятичной записью числа, кратного 9;

б) текст является записью чётного числа в семеричной системе;

в) текст является десятичной записью числа, кратного 6;

г) текст является десятичной записью числа, кратного 4;

д) текст является шестнадцатеричной записью числа, кратного 5;

е) текст начинается с некоторой ненулевой цифры, за которой следуют только буквы, и их количество равно числовому значению этой цифры;

ж) текст начинается с k букв ($1 \le k \le 9$), за которыми следует только один символ – цифра с числовым значением k;

з) текст совпадает с начальным отрезком ряда 0123456789 (например: 0, 01, 012);

и) текст совпадает с конечным отрезком ряда 0123456789 (например: 9, 89, 789);

к) текст совпадает с каким-то отрезком ряда 0123456789 (например: 2, 678);

л) текст состоит только из цифр, причём их числовые значения образуют арифметическую прогрессию (например: 2468, 741, 3);

м) текст содержит (помимо букв) только одну цифру, причем её числовое значение равно длине текста;

н) сумма числовых значений цифр, входящих в текст, равна длине текста.

В следующих задачах при необходимости ввода целого числа воспользоваться решением задачи 3.14

3.20 Выписать фрагмент программы для решения указанной задачи.

а) Среди степеней числа 10 (1, 10, 100, …) найти *p* – первую (наименьшую), которая больше или равна x ($x>0$).

б) Подсчитать *s* – сумму цифр неотрицательного целого числа *n*.

в) Подсчитать *k* – количество цифр неотрицательного целого числа *n*.

г) Найти *d* – старшую (левую) цифру неотрицательного целого числа *n*.

д) Определить, сколько раз (*k*) цифра 8 встречается среди первых пяти цифр дробной части положительного вещественного числа *x*.

е) Переменной *t* присвоить значение *1*, если натуральное число *k* является степенью 10 (1, 10, 100, …), и значение *0* иначе.

ж) Получить целое число *k* добавлением слева цифры 6 к десятичной записи натурального числа *n* (например, *k*=6107 при *n*=107).

3.21 Определить, верно ли, что куб суммы цифр натурального числа n равен n^2.

3.22 Определить, является ли натуральное число `n` степенью числа 3.

3.23 Для данного вещественного числа `a` среди чисел 1, 1 + (1/2), 1 + (1/2) + (1/3), ... найти первое, большее `a`.

3.24. Для данного вещественного положительного числа `a` найти наименьшее целое положительное n такое, что 1 + 1/2 +1/3+ ... + 1/n > a.

3.25 Дано натуральное число n. Получить все его натуральные делители.

3.26 Дано целое число m > 1. Получить наибольшее целое k, при котором $4^k < m$.

3.27 Дано натуральное число n. Получить наименьшее число вида 2^r, превосходящее n.

3.28 Написать программу, печатающую в стандартный вывод все простые числа из диапазона [2, *n*]. Число P называется простым, если P>=2 и делится только на 1 и на себя.

3.29 Распечатать первые n чисел Фибоначчи: $f_0=1,\ f_1=1,\ f_{k+1}=f_{k-1}+f_k,\ k=1,2,3...$

Тема 4.

Функции

Мы уже встречались с функцией `main` – это главная функция (точка входа), с которой начинается выполнение программы. Печать строки, а также символьный ввод и вывод – тоже функции, которыми мы уже пользовались. Кроме них, в программе могут быть и другие функции. Они нужны, чтобы вынести и оформить некоторый код для многократного применения. Для использования (вызова) таких функций необходимо привести описание того, что они делают.

Общий вид (полного) описания функции следующий:

```
<тип результата> <имя функции>(<список параметров>)
{
   <тело функции – объявления переменных и операторы>
}
```

В списке параметров перед каждым параметром обязательно указывается его тип. Все параметры функций передаются по значению. Приведем пример описания функции, возвращающей сумму кубов своих аргументов:

```
int sum3(int a, int b)
{
   return a*a*a + b*b*b;
}
```

Первая строка в описании функции, содержащая тип возвращаемого значения, имя и список параметров, называется заголовком функции. В языке Си действует принцип: «сначала опиши, потом используй». Иначе говоря, перед функцией, использующей другую функцию, должна быть указана информация об используемой функции. Минимально необходимая информация о функции – это тип результата, имя функции и типы её параметров (имена параметров можно не указывать). Такая информация об используемой функции называется заголовком (прототипом, профилем). Для функции `sum3` заголовок — это ее первая строка, после которой ставим точку с запятой:

```
int sum3(int a, int b);
```

Необходимо, чтобы заголовок появился до первого обращения к функции, тогда как само описание может встретиться в тексте программы и позднее. Отметим также, что указание имен параметров в заголовке не является обязательным:

```
int sum3(int , int );
```

При таком подходе программа будет иметь следующую структуру:

```
#include <stdio.h>
//заголовки используемых функций
int sum3(int, int);
//функция main
int main()
{
   …
   x = sum3(3, 5);
   …
}
//реализация функций
```

```
int sum3(int a, int b)
{
   return a*a*a + b*b*b;
}
```

Чаще всего описание функции `main` находится вверху программы, а описания остальных функций располагаются ниже. В таком случае перед функцией `main` необходимо разместить заголовки остальных функций. Возможным является также вариант, когда в программе описаны все используемые функции, и только после них появляется `main`, однако при появлении значительного количества функций такую программу сложнее читать.

Оператор `return E;` служит для завершения функции. Выражение E, указанное после `return`, вычисляется, и его значение возвращается в точку вызова. Если тип E не совпадает с типом результата функции, указанным в заголовке, значение неявно приводится к этому типу, по тем же правилам, что и при присваивании. Операторы, стоящие после return, не выполняются. Например, если в теле функции есть такая последовательность операторов:

```
return 1;
x = 8;
return x;
```

то два последних оператора никогда не будут выполнены – после `return 1;` сразу происходит возврат в точку вызова функции со значением 1.

В функции может быть несколько операторов `return`, но обычгно их используют внутри условных операторов: если усолвие истинно, возвращают значение, иначе продолжают вычисления.

Если функция ничего не должна возвращать (то есть выполняет только действия), тип ее результата указыывается как `void`. Если тип возвращаемого значения функции не указан, то по умолчанию он будет иметь тип `int`.

Пример функции, которая не возвращает значение – функция, рисующая розового слона. Её параметр – высота слона. Если заданная при вызове высота слона слишком мала, то функция ничего не рисует.

```
void draw_elefant(int h)
{ if (h < 1) return;
  …
}
```

В этой функции после `return` нет выражения, т.к. ничего возвращать не нужно. Вызов этой функции выглядит следующим образом:

```
…
draw_elefant(25);
…
```

Если в теле функции были выполнены все операторы, а оператор `return` так и не встретился, то управление возвращается в точку вызова. Если функция объявлена возвращающей значение, а возврат значения с помощью `return` не произошёл, то в точку вызова функции в качестве результата возвращается «мусор». Попытка его использовать приводит к неопределенному поведению программы.

Напишем реализацию полезной для решения задач функции `readnum`, реализующей ввод целого числа без знака в заданной системе счисления. Параметр функции – основание системы счисления вводимого числа, результат – введённое число. Будем считать, что основание системы счисления не превышает 10, число записано использующимися в этой системе счисления цифрами, за которыми следует отличный от

цифр символ — разделитель (который тоже будет прочитан). Например, при вычислении выражения `x=readnum(3)` и строки `"12 "` во входном потоке в переменную `x` запишется значение `5`. При считывании числа из входного потока будет прочитан и следующий за ним пробельный символ.

```
int readnum(int q)    /* q – основание системы счисления */
{  int c, S = 0;
   while((c = getchar()) >= '0' && c – '0' < q)
      S = S * q + c - '0';    /* считывание числа */
   return S;
}
```

Задачи

4.1 Написать функцию, которая читает из стандартного входного потока последовательность символов (до перевода строки) и подсчитывает количество пробелов в ней. Требуются ли параметры такой функции?

4.2 Написать программу, которая вводит два числа в десятичной системе счисления, используя только что рассмотренную функцию `readnum`, находит их наибольший общий делитель и печатает его.

4.3 Написать функцию с одним параметром — неотрицательным числом, проверяющую, является ли число n совершенным. Совершенными называются числа, равные сумме всех своих делителей. Пример: `6` – совершенное число, так как `6 = 1+2+3`.

4.4 Написать функцию возведения целого числа x в заданную степень n.

4.5 Написать функцию вычисления факториала заданного положительного числа типа unsigned short. Предусмотреть, что при вычислении факториала может быть получен результат, выходящий за пределы unsigned short. Каким нужно сделать тип возвращаемого значения функции?

4.6 Используя только один оператор цикла и не вводя дополнительные функции, написать функцию, которая для заданного вещественного x и целого n вычисляет значение

а) $\sum_{n=0}^{\infty} \frac{x^n}{n!}$ б) $\sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$ в) $\sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$

Тема 5

Массивы

Объявление массива имеет следующий вид:

```
<тип элементов> <имя>[<количество элементов>];
```

Перед именем массива указывается тип его элементов, после имени в квадратных скобках задается количество элементов массива.

Особенности массивов:

– элементы одного типа,

– элементы располагаются в памяти подряд, единым блоком,

– количество элементов задается при объявлении массива и не меняется в процессе работы с ним.

Нумерация элементов массива в языке С начинается с нуля – в начале массива находится элемент с индексом 0. При описании массива можно сразу инициализировать его элементы начальными значениями – значения элементов указываются в фигурных скобках через запятую. Доступ к отдельному элементу массива возможен через индекс элемента – он указывается после имени массива в квадратных скобках. Например, `A[3]` – это элемент массива `A` с индексом `3` (четвертый от начала массива элемент, поскольку номер самого левого элемента – 0). Пример:

```
int A[5] = {0, 1, 2, 88, 100};
```

Это описание массива целых (тип `int`) с именем `A`, состоящим из пяти элементов, с инициализацией элементов начальными значениями. Элемент с индексом ноль `A[0]` получает значение 0, элемент с индексом 1 – `A[1]` – получает значение 1, и т.д.

Можно инициализировать не весь массив, а только несколько начальных элементов массива, задав их значения в фигурных скобках через запятую. Те элементы массива, которым не хватило значений, не будут инициализированы при создании массива, в них будет «мусор». При создании переменной типа массив в памяти резервируется такой блок расположенных последовательно байтов, чтобы в нём разместились все элементы массива. Инициализация начальными значениями происходит только тех элементов массива, для которых указаны начальные значения, у остальных элементов массива – случайный двоичный код, оставшийся от предыдущих использований этого участка памяти («мусор»).

Напишем реализацию нескольких полезных функций для работы с массивами, элементами которых являются целые числа. Поскольку элементы массива часто обрабатываются однотипным образом, полезным для работы с ними является цикл for. Операции над массивами в большинстве случаев сводятся к операциям над их элементами, их необходимо реализовать. Напишем функцию печати элементов массива. Аргументы функции – массив и количество его элементов. Для печати описанного выше массива из пяти элементов вызов функции должен выглядеть следующим образом:

```
printarray(A, 5);
```

Задача этой функции – печать элементов массива, возвращать ей ничего не нужно, поэтому тип результата – `void`. Описание функции

```
void printarray(int A[], int k)
/* k – количество элементов */
{  int i;
   for(i = 0; i < k; i++)
     printf("%d ", A[i]); /*печать элементов через пробел*/
   putchar('\n');   /*перевод строки после вывода элементов*/
}
```

Для реализации функции ввода элементов целочисленного массива воспользуемся описанной нами в предыдущем параграфе функцией `int readnum(int q)`,

позволяющей ввести число в системе счисления, заданной в параметре. Заголовок функции ввода прохож на функцию печати: задаем имя массива и количество элементов, тип результата функции – `void`. Описание функции

```
void readarray(int A[], int k)
{  int i;
   for(i = 0; i < k; i++)
      A[i] = readnum(10);
}
```

Напоминание: если определения функций в программе находятся ниже функции `main`, то перед функцией `main` должны быть записаны объявления функций.

Задачи

5.1 Написать функцию для получения указаного значения по заданному целочисленному массиву из n элементов.
а) вычислить количество отрицательных чисел в массиве;
б) определить индекс первого элемента целочисленного массива из n элементов, значение которого равно заданному числу x. Если такого элемента в массиве нет, то вернуть –1;
в) найти наибольшее число в массиве;
г) посчитать индекс минимального значения в массиве (если такой элемент встречается в массиве несколько раз, выдать индекс его первого вхождения);
д) вычислить среднее арифметическое элементов массива (необходимо учесть, что оно, в общем случае, является вещественным числом);
е) вычислить $x_0 - x_1 + x_2 - \ldots - x_{n-2} + x_{n-1}$;
ж) вычислить $x_0 x_{n-1} + x_1 x_{n-2} + \ldots + x_{n-1} x_0$;
з) вычислить $x_0^2 + x_2^2 + x_4^2 + \ldots + x_{n-1}^2$ (считаем, что n – нечетное);
и) вычислить $x_{n-1} (x_{n-1}+x_{n-2}) (x_{n-1}+x_{n-2}+x_{n-3}) \ldots (x_{n-1}+ \ldots +x_0)$;
к) вычислить сумму элементов массива, предшествующих первому вхождению 0 в массив, и всех элементов, если 0 в массиве нет;
л) вычислить количество элементов, у которых равные "соседи" (предыдущий и следующий элементы)

5.2 Реализовать в виде функций следующие операции над массивами A и B, содержащими одинаковое количество элементов.
а) Скалярное произведение: вычислить $A_0B_0 + A_1B_1 + \ldots + A_{n-1}B_{n-1}$.
б) Сравнение массивов: вернуть 1 если массивы A и B (поэлементно) равны, и значение 0 иначе.
в) Копирование массива: переписать содержимое массива A в массив B.

5.3 Описать функцию, которая возвращает 1, если указанное свойство элементов массива выполнено и 0 в противном случае. Считать, что в массиве не менее трех элементов. Проверяемые свойства:
а) элементы массива образуют возрастающую последовательность;
б) элементы массива образуют арифметическую прогрессию (со знаменателем отличным от 0);
в) элементы массива образуют геометрическую прогрессию;
г) массив симметричен (его равноудаленные от концов элементы равны);
д) все элементы массива являются степенями 2;
е) все элементы с четными индексами являются числами Фибоначчи.

5.4 Написать функцию, которая для целочисленного массива из n элементов выполнит заданное преобразование. Вспомогательные массивы не заводить.

а) поменять местами максимальный и минимальный элементы массива (считать, что в массиве все элементы различны);

б) элементы массива расположить в обратном порядке:

$(x_0, x_1, \ldots, x_{n-2}, x_{n-1}) \rightarrow (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$;

в) переставить местами левую и правую половины массива;

г) элементы массива циклически сдвинуть на одну позицию влево:

$(x_0, x_1, x_2, \ldots, x_{n-1}) \rightarrow (x_1, x_2, \ldots, x_{n-1}, x_0)$;

д) элементы массива циклически сдвинуть на одну позицию вправо;

е) элементы массива циклически сдвинуть на две позиции влево;

ж) ненулевые элементы массива перенести в его начало, а все нулевые – в конец;

з) переставить элементы массива так, чтобы слева от (новой позиции) x_0 оказались меньшие его элементы, а справа – бóльшие или равные.

5.5 Написать функцию сортировки массива по неубыванию (т.е. переставить его элементы так, чтобы для всех k выполнялось $x_k <= x_{k+1}$), и продемонстрировать ее работу на нескольких массивах. Методы сортировки:

а) Сортировка выбором. Найти максимальный элемент и поменять его местами с последним элементом. Затем применить этот же метод ко всем элементам, кроме последнего (он уже находится на своем окончательном месте), и т. д. Когда второй слева элемент окажется на своем месте, массив упорядочен.

б) Пузырьковая сортировка. Последовательно сравнивать пары соседних элементов x_k и x_{k+1} (k=0, 2, 3, ..., n–2) и, если $x_k > x_{k+1}$, переставлять их; тем самым наибольший элемент окажется на своем месте в конце массива. Затем применить этот же метод ко всем элементам, кроме последнего, и т.д.

в) Сортировка вставками. Пусть первые k элементов массива уже упорядочены по неубыванию. Сравнивая (k+1)-й элемент с предыдущими элементами (в обратном направлении), найти его место среди этих элементов и вставить его сюда, предварительно освободив это место сдвигом части предыдущих элементов на одну позицию вправо; тем самым будут упорядочены уже k+1 первых элементов массива. Повторять эти действия для всех k от 0 до n–2.

5.6 а)Написать функцию, которая находит для заданного массива положительных целых чисел мажоритарный элемент – значение, которому равны не менее половины элементов массива. Если такого элемента в массиве нет, выдать в качестве ответа `-1`.

б) Если в массиве точно присутствует мажоритарный элемент, какое минимальное количество операций сравнения требуется выполнить, чтобы гарантированно этот элемент найти?

Тема 6

Указатели. Параметры функций — указатели. Связь указателей и массивов.

Указатели – это переменные, хранящие адреса. Адресом называется номер ячейки памяти компьютера. Указатели позволяют обращаться к объектам в памяти по их адресам. Переменная типа «указатель на некоторый тип» хранит адрес объекта этого типа, в описании перед именем такой переменной ставится символ звёздочка. Например,

```
int *p;
```

является описанием переменной p, тип которой «указатель на целое» (`int *`). Такая переменная предназначена для хранения адреса объекта типа int. Для доступа к самому объекту (целому числу) используется унарная звёздочка – операция разыменования указателя: `*p`. Выражение `*p` обозначает объект типа `int`, расположенный по адресу, который хранится в p. Иначе говоря, значение `p` — это адрес числа, а `*p`— само это число.

Есть специальная константа `NULL` (ее значение равно 0) — она обозначает нулевой указатель, то есть указатель, который ни на что не указывает. `NULL` можно присваивать указателю любого типа, независимо от того, на объект какого типа он должен указывать. Смысл этого значения — «объекта нет». Разыменовывать такой указатель нельзя: попытка получить значение по нулевому указателю приводит к неопределённому поведению — программа может аварийно завершиться или повести себя непредсказуемо.

Обратная к операции разыменования – это операция взятия адреса объекта. Если у нас описана целая переменная `k`

```
int k = 12;
```

то выражение `&k` обозначает адрес переменной `k`, т.е. адрес ячейки памяти, в которой находится значение `12` и которая названа `k`. Этот адрес можно, например, присвоить переменной-указателю `p`:

```
p = &k;
```

С этого момента в p хранится адрес переменной `k`: `p` указывает на переменную `k`, а выражение `*p` и имя `k` обозначают одну и ту же ячейку памяти.



После выполнения оператора-выражения `*p = 123;` значения переменных станут такими:



Операция взятия адреса `&` возвращает адрес указанного после неё объекта памяти. Адрес по сути – это число, номер байта памяти, с которого размещается указанный объект, а числу ничего присвоить нельзя, т.е. выражение вида `&k` в качестве левого операнда операции присваивания недопустимо. Операция `*` дает в качестве результата ячейку памяти, которая может быть указана как слева, так и справа от присваивания.

Операция ++, применённая к указателю, меняет его значение: ++p увеличивает значение p на размер объекта, на который он указывает, так что p начинает указывать на

следующий объект того же типа. Однако в нашем случае p указывает на одиночную переменную k, а что находится в памяти сразу за ней, неизвестно; поэтому применять ++ к такому указателю не имеет смысла. Операции ++ и – – применяются к указателям на элементы массива: они сдвигают указатель на следующий или предыдущий элемент соответственно. В то же время выражение ++(*p) изменяет значение объекта, на который указывает p. В нашем контексте это увеличит на 1 значение переменной k.

В языке C передача параметров функции возможна только по значению. Это означает, что при вызове функции в памяти выделяется место для локальных переменных — параметров функции. Выражения, переданные в качестве аргументов (фактические параметры), вычисляются, и их значения присваиваются этим локальным переменным. Таким образом, функция работает с копиями значений аргументов, а после завершения функции и возврата результата эти локальные переменные уничтожаются.

Пусть стоит задача написать функцию, которая меняет местами значения двух переменных. Приведенная ниже функция swap именно из-за передачи параметров по значению не решает задачу.

```
#include <stdio.h>
void swap(int a, int b)
{
  int c = a;
  a = b; b = c;
}
int main()
{
   int x = 1, y = 2;
   swap(x, y);
   printf("x = %d, y = %d\n", x, y);
   return 0;
}
```

Эта программа напечатает строку "x = 1, y = 2". При вызове swap(x,y) в памяти создаются локальные переменные a, b и c, в локальную переменную a записывается значение выражения x (значение переменной x, т.е. 1), в локальную переменную b записывается значение выражения y (значение переменной y, т.е. 2). Далее функция swap с помощью переменной c меняет местами значения ***локальных*** переменных a и b, что никак не затрагивает переменные x и y из функции main, значения которых требовалось поменять. Когда swap завершает работу, локальные переменные a, b и c уничтожаются. Со значениями переменных x и y ничего не произошло, в них останутся значения, заданные при описании.

Как же исправить функцию swap, чтобы она меняла местами значения переменных, переданных ей в качестве аргументов? Для этого в функцию нужно передавать не сами значения, а адреса переменных x и y.

Правильный вариант этой функции приведён ниже:

```
#include <stdio.h>
void swap(int *a, int *b)
{
   int c = *a;
   *a = *b; *b = c;
}
int main()
{
   int x = 1, y = 2;
   swap(&x, &y);
   printf("x = %d, y = %d\n", x, y);
```

```
    return 0;
}
```

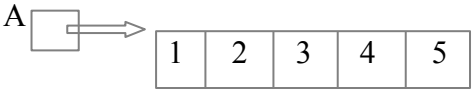
В этой функции также создаются локальные переменные `a` и `b`, но в них записываются не значения переменных `x` и `y`, а адреса этих переменных. При такой передаче параметров на время работы функции `*a` становится синонимом переменной `x` (той же самой ячейкой памяти), а `*b` становится синонимом `y`. С помощью локальной переменной `c` значения, находящиеся по адресам `a` и `b` (т.е. значения переменных `x` и `y`), меняются местами. После окончания работы функции `swap` локальные переменные `a` и `b` уничтожаются, но побочный эффект – изменение значений переменных `x` и `y` – остаётся, поэтому программа распечатает строку "`x = 2, y = 1`".

Указатели и массивы

Рассмотрим, что происходит при описании переменной-массива и при передаче массива в качестве аргумента в функцию. Пусть есть описание переменной-массива A:

```
int A[5]= {1,2,3,4,5};
```

Для такого описания в памяти выделяется блок последовательных байтов, в которых можно разместить пять целых чисел (размер блока = `sizeof(int) * 5`). Кроме этого выделяется неизменяемая память для хранения одного адреса, которая получает имя `A`. В эту память записывается адрес начала выделенного блока, менять это значение (адрес начала массива) запрещено. Тип `A` – указатель на целое, т.к. описан массив целых, а в переменной A находится адрес первого (самого левого, с индексом 0) элемента массива.



Можно завести указатель на целое и присвоить ему адрес начала массива A:

```
int *p = A;
```

Переменную `A` нельзя менять (адрес начала массива – константа, которую запрещено менять), но на переменную `p` это ограничение не распространяется, её менять можно.

В языке С определены следующие операции над указателями:

1. Разыменование: `*p` – это целое число, адрес которого находится в переменной `p`.
2. Взятие адреса: `&p` результатом является адрес ячейки памяти, в которой находится p, то есть адрес адреса. Если p было описано как `int * p;` то тип результата выражения `&p` – это `int **`.
3. Прибавление или вычитание целого числа. При прибавлении к указателю или вычитании из него целого числа значение указателя изменяется на это число, умноженное на размер типа, адресуемого указателем. Например, p + 1 — указатель на следующий элемент того же типа: к адресу, хранящемуся в p, прибавляется не 1, а sizeof(*p) (в нашем случае — размер int). При вычитании целого числа из указателя его значение уменьшается на то же число, умноженное на размер того же типа.
4. Вычислять разность указателей. Разность адресов, значений указателей, делится на размер данных – результатом такой разности будет количество данных, которое поместится между двумя адресами в памяти. Если указатели содержат адреса независимых данных, то такая разность не имеет смысла. Если же оба указателя указывают на элементы одного массива, то их разность – количество элементов массива между ними.
5. Сравнивать адреса. Сравнение адресов на больше или меньше имеет смысл только в том случае, когда это адреса элементов одного массива. К адресам одного массива можно применять все шесть операций сравнения (`<, <=, >, >=, ==, !=`). Все

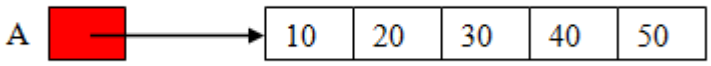
остальные адреса имеет смысл сравнивать только на равенство (==) или неравенство (!=).

6. Присваивание указателей. В языке С запрещено присваивание указателей разных типов – указателю может быть присвоен только указатель такого же типа. Исключение составляет обобщенный указатель `void *`, он может быть операндом присваивания как слева, так и справа. Однако указатель типа `void *` нельзя разыменовывать, предварительно не сделав его приведение к указателю на какой-то тип.
7. Поскольку для указателей определено присваивание и сложение с целым числом, а также вычитание числа, то операции `+=` и `-=` также применимы, если их левый операнд – указатель, а правый – целочисленное выражение. Операции `++` и `--` работают с указателями, могут быть записаны как в префиксной, так и в постфиксной форме.
8. Индексирование. Запись `A[i]` представляет собой более короткую запись выражения `*(A+i)`, которое обозначает элемент массива по адресу `A + i`. Отметим также, что в выражении `*(A+i)` скобки обязательны, так как первым должно выполняться именно сложение указателя `A` и числа `i`. Только после этого происходит разыменование указателя, полученного в результате сложения.

Рассмотрим пример работы с массивом с использованием указателя на него. После каждого оператора будем выписывать, какие изменения происходят в памяти компьютера.

```
int A[5] = {10, 20, 30, 40, 50};
```

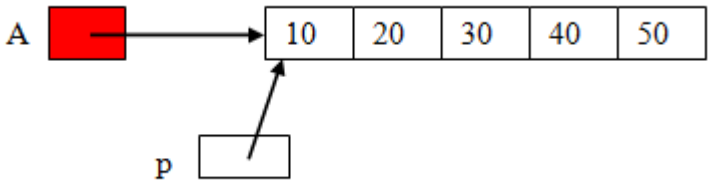
Описание массива целых чисел с начальной инициализацией элементов массива числами 10, 20, 30, 40 и 50 соответственно. В памяти отводится место для хранения пяти чисел и для хранения константной ссылки на начало массива:



Красным цветом обозначим, что этот адрес начала массива менять нельзя (константа).

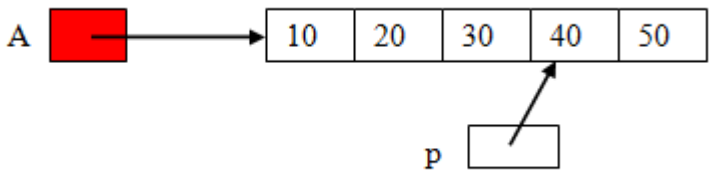
```
int *p = A;
```

Создается переменная p, она указывает на начало массива A, то есть хранит адрес элемента `A[0]`.




```
p = p+3;
```

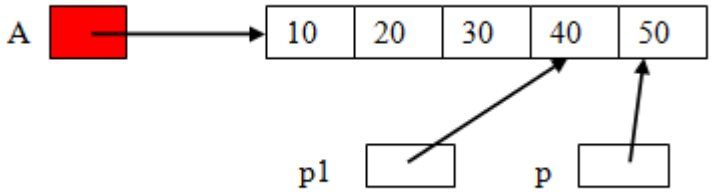
Сдвиг указателя `p` на три числа вправо, `p` теперь указывает на элемент `A[3]`.




```
int *p1 = p++;
```

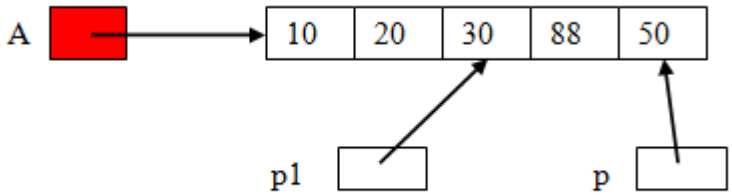
Создается переменная `p1`, она получает текущее значение переменной `p` – адрес `A[3]`

(постфиксная операция ++ возвращает текущее значение переменной), после чего указатель `p` сдвигается на одно число вправо (побочный эффект операции ++), его значение – адрес `A[4]`.



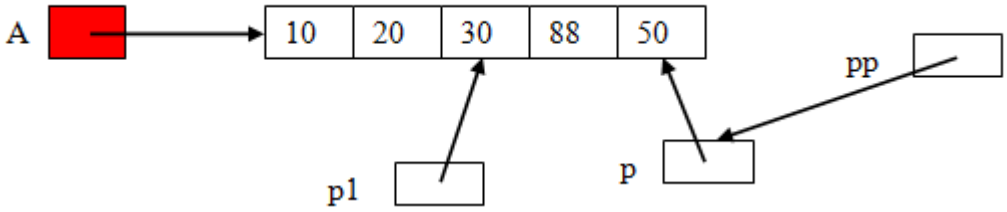

```
*p1-- = 88;
```

Постфиксная операция `--` имеет более высокий приоритет, чем префиксная операция `*`, поэтому выражение `*p1--` вычисляется как `*(p1--)`, т.е. возьмётся «старое» значение указателя `p1`, значение-число `*p1` по этому адресу заменится на число `88`, после чего изменится значение самого указателя `p1` (побочный эффект постфиксной операции `--`) – он сдвинется на предыдущее число (будет указывать на `A[2]`).




```
int **pp;
pp = &p;
```

Создаётся переменная `pp` – указатель на указатель, ему присваивается значение – адрес переменной `p`.



Поскольку имя массива – это константный указатель на начальный элемент `A[0]`, то передача массива в функцию – это передача по значению адреса начала массива. Поэтому при описании аргумента-массива функции не важно, как этот аргумент описать – как `int A[]` или как указатель на значение типа `int *A`. В любом случае функции необходимо также передать информацию о количестве элементов массива в дополнительном параметре. Например, при описании функции, вычисляющей среднее арифметическое элементов массива, её заголовок может быть таким:

```
double average(int A[], int k);
```

или таким:

```
double average(int *A, int k);
```

Эти два заголовка эквивалентны с точки зрения языка С. Возможен также и вариант

заголовка

```
double average(int A[100], int k);
```

Указанное в скобках число не передается в функцию (существенными являются в этой записи именно скобки), заголовок эквивалентен предыдущим.

В функцию, работающую с массивом, при желании можно передать не весь массив, а только его часть, передав указатель на начало участка массива и количество элементов этого участка. Например, для описанного выше массива `A` из пяти элементов возможно вычисление среднего арифметического трёх его начальных или конечных элементов с помощью вызовов `average(A,3)` и `average(A+2,3)` соответственно.

Операция индексирования – это операция разыменования указателя, сдвинутого относительно начала массива на указанное количество элементов. Поэтому следующие записи эквивалентны:

`A[0]` эквивалентно `*A`
`A[2]` эквивалентно `*(A+2)`
`A[k]` эквивалентно `*(A+k)`

Использование указателей для получения результата функции через параметры
Форматный ввод – функция scanf

Указатели используются в библиотечной функции **`scanf`** для ввода различных объектов встроенных типов – целых и вещественных чисел, строк и т.д. Первым её аргументом является форматная строка, в которой указывается, что нужно ввести. Остальные аргументы – адреса, по которым будут сохраняться (записываться в память) введённые значения. Сама функция scanf в качестве результата возвращает количество считанных с входного потока объектов – так можно проверить, все ли объекты, указанные в форматной строке, были считаны. Не сто́ит указывать в форматной строке лишние символы – они будут ожидаться во входном потоке при считывании данных. Приведём примеры ввода значений различных типов с помощью функции scanf:

```
int x;
scanf("%d", &x);  /* считывание целого в переменную x */
double y;
scanf("%lf", &y);  /* считывание вещественного в переменную y */
char c;
scanf("%c", &c);  /* считывание символа в переменную c */
int a, b;
scanf("%d%d", &a, &b);    /* считывание двух целых чисел в
 переменные a и b, если функция вернула 2 – считаны оба числа */
```

В последнем примере между вводимыми числами допустим в качестве разделителя любой пробельный символ (пробел, табуляция или перевод строки). Если в последнем примере `scanf` в качестве результата вернула `1`, то это означает, что было считано только одно число из двух. Если её результат 0 – ни одно из двух чисел не было считано (например, во время чтения произошло нажатие Ctrl+D, файл чтения закрылся). Поэтому вызов функции `scanf` нередко ставят в условие `if`:

```
if (scanf("%d%d", &a, &b) != 2) /* ошибка */
```

Если в буфере ввода содержится последовательность символов `25 45`, то последний вызов `scanf` считает в переменную `a` число `25`, в переменную `b` – число `45`. Если же в буфере ввода `2 5 45`, то в переменную `a` считается число `2`, в переменную `b` – число `5`, а `45` останется в буфере ввода.

Указатели используются и для возврата значений из описанных пользователем функций. Например, если нужно реализовать функцию, которая получает два целых числа и должна вычислить и вернуть три значения – сумму квадратов, разность квадратов и

произведение переданных ей чисел, то одно из значений может быть возвращено как результат функции, а остальные два – через аргументы функции. Аргументы для возвращаемых из функции значений должны быть переданы в функцию как указатели для записи по переданным адресам вычисленных значений. Пример определения функции, вычисляющей три указанных выше значения:

```
int compute(int a, int b, int *sum, int *dif)
{  *sum = a*a + b*b;
   *dif = a*a – b*b;
   return a*b;
}
```

Пример вызова этой функции:

```
int main()
{  int sum, dif, pr;
   pr = compute(4, 5, &sum, &dif);
   …
   return 0;
}
```

Ранее для ввода целых чисел в массив целых использовалась написанная нами функция `readnum`, теперь для ввода числа можно использовать функцию `scanf`:

```
for(i = 0; i < k; ++i) scanf("%d", A+i);
```

Во втором аргументе `scanf` можно было бы написать конструкцию `&A[i]` – адрес i-того элемента массива. Но вспомним, что имя массива – это адрес его начала (адрес элемента с индексом 0), поэтому адрес i-го элемента – это `A+i`, т.е. `&(A[i])` – это `&(*(A+i))`, то есть `A+i`.

Ещё немного о присваивании указателей:

Запрещено присваивание указателей разных типов. Но есть одно исключение – обобщённый указатель. Его описание выглядит следующим образом:

```
void *vp;
```

Ему можно присвоить значение указателя любого типа, например:

```
vp = p;
```

Но прежде чем его разыменовывать, нужно преобразовать его к указателю нужного типа. Например, для доступа к целому по адресу, скопированному из переменной p, `*(int *)vp`.

Например, если у нас есть два указателя – на целое и на вещественное, то напрямую мы не сможем значение одного указателя присвоить другому указателю:

```
int *ip; double *dp;
ip = dp;  /* ошибка, в языке С так нельзя */

void *vp;
vp = dp; ip = vp;  /* можно, но на совести программиста */
```

Задачи

6.1 Допустимы ли в программе на Си следующие последовательности операторов? Если "да" - опишите семантику каждого правильного действия (не принимая во внимание ошибочные); если "нет" — объясните, в чём заключается ошибка

```
а). . .
int i, *p, j, *q;
p = &i; q = &p;
j = *p = 1; q = p-1; *p += 1;
i = *++q + * q -= 1; i = *q ++ + *q;
printf("i=%d, j=%d, *p=%d, *q=%d \n", i, j, *p, *q);

б) . . .
int x = 1, y; char c = 'a';
int *pi, *qi; char *pc;
pi = &x; *pi = 3; y = *pi; *pi = c; qi = pi;
pc = qi; *qi+=1; pi++; *(--pi) = 5; y = *qi+1;
pc = &c; ++*pc; (*pc)++; *pc++; *pc+=1;
x = (int)pi; pi=(int*)pc; pi=(int*)x; x = 1+ *pi; pc=(char*)pi;
c = *pc; pc = &y; x = qi – pi; qi = 0; qi+=pi;
y = &pi; y = (int)&pi; pi = pi +5; *(pi+1)=0; pi=&(x+0);
```

6.2. К любому ли объекту в Си можно применять операцию взятия адреса `&` ?

6.3 Для вещественного массива `a`, содержащего n элементов, описать функцию G, определяющую значения максимального и минимального элементов этого массива. Решить задачу за один просмотр массива.

6.4 Описать функцию, определяющую, чередуются ли положительные и отрицательные элементы в целочисленном массиве `a` из n элементов и вычисляющую целочисленное значение `p`. Если элементы чередуются, то `p` - это сумма положительных элементов, иначе `p` – это произведение отрицательных элементов. С помощью этой функции провести анализ целочисленного массива `x[50]`.

6.5 Описать функцию, определяющую, упорядочены ли строго по возрастанию элементы в целочисленном массиве `a` из n элементов, и вычисляющую целочисленное значение `p`. Если элементы упорядочены, то `p` — это произведение разностей рядом стоящих элементов, иначе `p` — это количество нарушений порядка в массиве `a`. С помощью этой функции провести анализ целочисленного массива `b[60]`.

6.6 Описать функцию, определяющую, какое число чаще других встречается в заданном целочисленном массиве `b` из n элементов, и сколько раз оно в массив входит. Если таких чисел несколько, взять минимальное из найденных.

6.7 Описать функцию, определяющую, какое число реже других (но не ноль раз) встречается в заданном целочисленном массиве b из n элементов и сколько раз оно в массив входит. Если таких чисел несколько, взять максимальное из найденных.

Тема 7.

Рекурсия

Что такое рекурсивная функция? Можно считать, что функция рекурсивна, если в её теле есть вызов самой себя. Более точное определение: функция является рекурсивной, если при выполнении её тела может произойти вызов этой же функции. Если в теле функции есть прямой вызов самой себя, говорят о прямой (явной) рекурсии. Если же прямого вызова нет, но есть вызов некоторой другой функции, при выполнении которой происходит вызов исходной, такую рекурсию называют косвенной. Чаще встречается прямая (явная) рекурсия.

Что нужно, чтобы написать рекурсивную функцию:

1. Подумать и выделить те случаи входных данных, при которых ответ может быть вычислен сразу, без каких-либо рекурсивных обращений. Такие случаи называют базой рекурсии. У любой рекурсивной функции должен быть определён такой базовый случай, чтобы вычисления вообще могли закончиться.
2. Переформулировать действия с входными данными, сформулировать рекурсивный алгоритм. Важно, чтобы в рекурсивный вызов подавались более простые данные, чем были на входе функции – тогда в процессе рекурсивных вызовов они когда-нибудь упростятся настолько, чтобы превратиться в базовый случай, для вычисления которого не требуется никакой рекурсии. Упрощение подаваемых в рекурсию данных – гарантия того, что рекурсивные вычисления рано или поздно закончатся.
3. Использовать факт, что рекурсивный вызов вернёт правильно вычисленный результат для поданных ему более простых данных.
4. Правильно вернуть результат вычисления самой функции, основанный на полученном из рекурсии результате.

Рассмотрим пример рекурсивного определения функции, вычисляющей факториал своего аргумента – целого неотрицательного числа. Факториал числа n – это произведение целых чисел от 1 до n. Но это итеративное определение факториала. Из него можно вывести рекурсивное определение:

```
n! = n * (n-1) * … * 2 * 1 = n * ((n-1) * … * 2 * 1) = n * (n-1)!
```

Т.е. рекурсивное определение факториала от числа n такое: чтобы вычислить факториал числа n, нужно вычислить факториал числа (n-1) и умножить его на n.

Что является базой рекурсии? По определению факториал нуля равен 1. Можно ещё вспомнить, что факториал 1 тоже равен 1. Этих двух случаев достаточно (можно даже обойтись только первым из них – для числа 0, но мы и для 1 не будем делать лишних вычислений). Будем считать базовыми случаями вычисление факториала чисел 0 и 1. Для всех остальных случаев входного параметра для вычислений используем рекурсивную формулу `n! = n * (n-1)!` Получаем следующую рекурсивную реализацию функции вычисления факториала

```
int fact(int q)
{  return (q <= 1)? 1: q * fact(q - 1);   }
```

Таким образом, одним из важных моментов является способность увидеть в итеративном описании каких-то вычислений рекурсивную формулу.

Приведём ещё один пример – вычисление двойного факториала. Формула его вычисления:

$$n!! = \begin{cases} 2 * 4 * 6 * \ldots * n, \text{при чётном } n \\ 1 * 3 * 5 * \ldots * n, \text{при нечётном } n \end{cases}$$

Как от этой формулы перейти к рекурсивной формуле? Простейшими случаями можно считать наименьшее четное число (0) и наименьшее нечётное число (1), в обоих этих

случаях двойной факториал равен 1, никаких вычислений не требуется. Если проанализировать формулу для нечётного n, то по аналогии с факториалом получим рекурсивную формулу вычисления:

```
n!! = 1*3*…*(n-2)*n = (n-2)!! * n
```

Для четных n получается та же самая формула: `n!! = (n-2)!! * n`. Причём в начало произведения четных чисел можно добавить умножение на 1 в виде двойного факториала нуля: `n!! = 0!! *2*4…*(n-2)*n`, результат вычислений будет тот же. В рекурсию будет подаваться число, меньше исходного на 2, следовательно, уменьшая каждый раз аргумент в рекурсивном вызове на 2, когда-нибудь аргумент станет 1 (если изначально число было нечётным) или 0 (если изначально аргумент был чётным числом). В обоих случаях вычисления закончатся, т.к. это два базовых случая вычислений, в обоих случаях результат функции будет равен единице.

```
int f2(int n) { return (n <= 1)? 1: n * f2(n-2);  }
```

Рассмотрим пример рекурсивной функции, которая не возвращает значения (имеет возвращаемый тип `void`). Для написания таких функций необходимо в последовательности действий, которые требуется выполнить, тоже увидеть рекурсию. Принцип упрощения подаваемых в рекурсию данных тут тоже остаётся – данные в рекурсивном вызове должны быть проще, чем были изначально. Так, необходимо продумать случаи базовых вычислений, когда рекурсия не нужна.

Рассмотрим в качестве примера решение следующей задачи: распечатать цифры параметра–числа в обратном порядке. Т.е. нужно сначала распечатать последнюю цифру числа, потом предпоследнюю и т.д., в самом конце – первую цифру. Это итеративное описание действий, которые должны быть выполнены. Как эту задачу можно разбить на более простые подзадачи, чтобы их количество не зависело от количества цифр исходного числа? Это можно сделать следующим образом: первая элементарная подзадача – распечатать последнюю цифру числа, вторая уже более сложная задача – распечатать все остальные цифры числа в обратном порядке. Итого две подзадачи, первая из которых элементарная, а вторая сложная, но по сравнению с исходной проще (нужно распечатать на одну цифру меньше) и очень на неё похожая. Такая формулировка действий уже является рекурсивной. По этой формулировке не представляет сложности написать рекурсивное определение соответствующей функции:

```
void printrev(int N)
{   printf("%d", N % 10); /* печать последней цифры */
    if (N >= 10) printrev(N / 10);
     //* печать остальных цифр в обратном порядке */
}
```

Задачи

7.1 Описать рекурсивную функцию вычисления x^n для вещественного x (x != 0) и целого n:

$$x^n = \begin{cases} 1 \text{ при } n = 0 \\ 1/x^{|n|} \text{ при } n < 0 \\ x * x^{n-1} \text{ при } n > 0 \end{cases}$$

7.2 Описать двумя способами: рекурсивным и итеративным функцию вычисления НОД(n,m) - наибольшего общего делителя неотрицательных целых чисел n и m. Использовать соотношение

НОД(n,m) = НОД(m,r), где r - остаток от деления n на m

Сравнить эффективность рекурсивной и нерекурсивной функций вычисления НОД, подсчитав количество выполненных основных операций.

7.3 Описать рекурсивную функцию вычисления НОД (n_1, n_2, n_3, ..., n_m), воспользовавшись для этого соотношением: НОД (n_1, n_2, n_3, ... , n_k) = НОД (НОД (n_1, n_2, n_3, ... , n_{k-1}), n_k), k = 3, 4, … , m. С ее помощью найти НОД (a_1, a_2, a_3, ..., a_{10}).

7.4 Описать рекурсивно и итеративно функцию вычисления n–ого числа Фибоначчи: $f_1 = 1$; $f_2 = 1$; $f_{j+1} = f_{j-1} + f_j$; j = 2, 3,….С помощью каждой из написанных функций вычислить 40-е число Фибоначчи. Сравнить количество выполненных каждой функцией операций сложения.

7.5 Функция Маккарти определяется следующим образом

```
int maccarthy(int n) {
 return (n > 100) ? n - 10 : maccarthy(maccarthy(n + 11));
}
```

Вычислить *f*(106), *f*(99) и *f*(85). Какие значения принимает эта функция?

7.6 Описать рекурсивную функцию вычисления значения A(n,m) - функции Аккермана для неотрицательных целых чисел n и m:

$$A(n,m) = \begin{cases} m+1 \text{ при } n = 0 \\ A(n-1,1) \text{ при } n! = 0, m = 0 \\ A\big(n-1, A(n,m-1)\big) \text{ при } n > 0, m > 0 \end{cases}$$

С помощью этой функции найти значение A(5,8).

7.7 Пусть *N* – неотрицательное целое число. Рекурсивно описать функцию от одного параметра *N*, которая:

а) подсчитывает количество (значащих) цифр числа *N*;

б) находит старшую (левую) цифру числа *N*;

в) находит наибольшую цифру числа *N*;

г) определяет, входит ли цифра 5 в запись числа *N*;

д) подсчитывает количество вхождений цифры 8 в запись числа *N*;

е) используя только символьный вывод (putchar), выводит на экран число *N*;

ж) используя только символьный вывод (putchar), выводит на экран число *N* в двоичном виде;

з) подсчитывает сумму цифр, стоящих на нечётных местах в записи числа *N*.

7.8 На стандартном вводе задана непустая последовательность положительных вещественных чисел, за которой следует отрицательное число. Описав подходящую рекурсивную функцию, найти сумму этих положительных чисел.

7.9 На стандартном вводе задана непустая последовательность положительных целых чисел, за которой следует 0. Описав подходящую рекурсивную функцию, найти наибольшее из этих чисел.

7.10 На стандартном вводе задан текст —последовательность символов, за которой следует `'\n'`. Описав подходящую рекурсивную функцию, подсчитать количество цифр в данном тексте.

7.11 На стандартном вводе задан текст —последовательность символов, за которой следует `'\n'`.

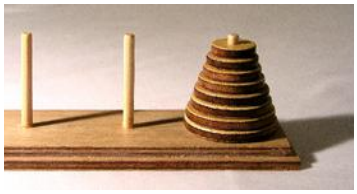
Описав подходящую рекурсивную функцию, вывести символы этого текста в обратном порядке.

7.12 На стандартном вводе задана непустая последовательность ненулевых целых чисел, за которой следует 0. Описав подходящую рекурсивную функцию или процедуру, вывести в стандартный поток вывода сначала все отрицательные числа этой последовательности, а затем – все положительные (в любом порядке).

7.13 «Ханойские башни». Имеются три стержня A, B и C и n дисков разного размера, перенумерованных от 1 до n в порядке возрастания их размеров. Сначала все диски надеты на стержень A в виде пирамиды — самый большой диск находится внизу, самый маленький – наверху.



исходное состояние

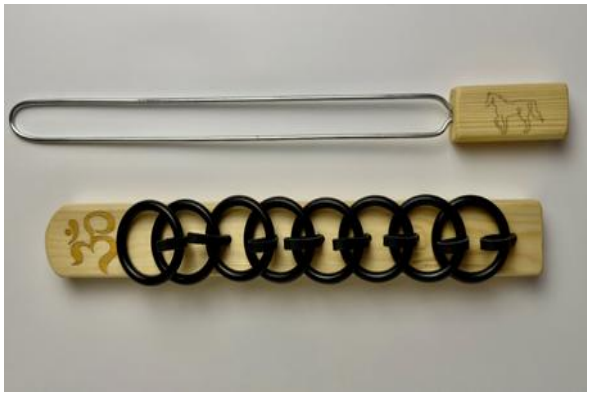


цель

Требуется перенести все диски с A на C, используя стержень B как вспомогательный, соблюдая при этом следующие условия:
1) снимать со стержня можно только верхний диск
2) диски можно переносить только по одному,
3) бóльший диск нельзя класть на меньший.
Описать функцию на языке C, печатающую последовательность действий, которую необходимо выполнить для решения головоломки.

7.14 Калмыцкая головоломка «Нярн Шинж»


Исходное состояние


Разобранная головоломка

Нужно снять 8 колец с планки. Кольца снимаются или надеваются протаскиванием через центральную щель планки. Перенумеруем кольца от левого края к правому, и сформулируем допустимые действия:
– можно снять первое кольцо;
– можно снять вместе первое и второе кольцо;
– можно снять какое-то кольцо, которое является вторым слева среди надетых (кольцо с меньшим на единицу номером надето, а все остальные кольца с меньшими номерами сняты).
Для тех же самых колец можно выполнить действие «надеть»
Написать на языке C программу, которая печатает последовательность действий для снятия 8 колец, указывая номер кольца и действие с этим кольцом. Для решения задачи рекомендуется описать рекурсивную функцию печати действий при снятии n колец и рекурсивную функцию печати действий при надевания n колец.

7.15 Написать рекурсивную функцию, получающую в качестве параметра массив целых чисел и количество его элементов. Функция должна проверить, симметричен ли массив.

7.16 Написать рекурсивную реализацию ***алгоритма быстрой сортировки***. Работа алгоритма состоит из следующих шагов:
1) Выбрать опорный элемент и поделить массив на последовательность элементов не больших опорного и последовательность тех, которые больше него.
2) Поставив опорный элемент на свое место, применить метод сортировки к элементам левее опорного и к элементам, правее опорного – два рекурсивных вызова.

7.17 Даны в порядке возрастания 10 натуральных чисел: a1, a2, …, an (n=10), а также натуральное число S. Определить, можно ли из этих ai выделить такой набор чисел (без повторов), чтобы их сумма равнялась числу S, и, если можно, вывести числа найденного набора.

7.18 Даны натуральные числа m1, m2, …, mn (m1=1, mi<mi+1, n=10) и натуральное число S. Рассматривая mi как достоинства монет и считая, что монет каждого достоинства имеется в неограниченном количестве, определить, каким минимальным числом этих монет можно получить сумму S. Вывести в стандартный вывод монеты такого набора.

7.19 Дано N различных натуральных чисел (N = 5). Вывести на экран все перестановки этих чисел.

7.20 Задача о 8 ферзях: на шахматной доске расставить 8 ферзей так, чтобы они не «били» друг друга. Написать программу, которая находит:
а) одну из таких расстановок;
б) все 92 такие расстановки.

7.21 Программа. Найти расстановку 5 ферзей на шахматной доске, при которой они «бьют» все поля доски.

7.22 На стандартном вводе задана последовательность символов, оканчивающаяся `'\n'`–формула, удовлетворяющая определению:

```
<формула> ::= <цифра> | (<формула> <знак> <формула>)
<знак> ::=  + | - | *
<цифра> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Вычислить и напечатать в стандартный вывод значение формулы.

7.23 На стандартном вводе задана последовательность символов, оканчивающаяся `'\n'`–формула, удовлетворяющая определению:

```
<формула> ::= <цифра> | M(<формула>,<формула>) |
           m(<формула>,<формула>)
<цифра> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

где M означает максимум, а m – минимум. Ввести эту формулу и вычислить ее значение.

Тема 8.

Строки

Строкой называется массив символов, оканчивающйися нулевым байтом – символом с кодом 0, т. е. '\0'. Исходя из такого определения, со строками можно делать все те же операции, которые можно делать с массивами, содержащими элементы другого типа. Однако для строк есть и некоторые дополнительные возможности.

Первая из них касается инициализации строки. Строковую переменную можно инициализировать строкой-константой, представляющей собой последовательность символов в двойных кавычках. Например:

```
char str[100] = "cat";
```

Первые элементы переменной `str` получат значения – коды указанных в строке символов, а после последнего символа строки следующий элемент получит значение ноль. Приведённая выше инициализация эквивалентна следующей (традиционно используемой для других видов массивов):

```
char str[100] = {'c', 'a', 't', 0};
```

Нулевой байт является признаком конца строки, он обязательно должен быть в конце строки – это требование всех библиотечных функций работы со строками. Например, библиотечная функция `printf` распечатывает содержимое строки до нулевого байта:

```
printf("%s\n", str);
```

Строковая переменная может быть аргументом функции `scanf`, апоскольку строковая переменная – это указатель на начало строки, то есть адрес, символ & перед ней не требуется:

```
scanf("%s", str);
```

В переменную-строку `str` считываются и записываются символы входного потока до пробельного символа (пробел, перевод строки, табуляция), после последнего считанного символа записывается нулевой байт, означающий конец строки. Необходимо аккуратно использовать данную функцию, поскольку проверку выхода за границу массива она не делает. Если строка во входном потоке длинная, больше отведённого для строковой переменной размера, то может произойти выход за границы выделенной области памяти, и дальнейшее поведение программы непредсказуемо. Более удобным является следующий вариант чтения строки

```
scanf("%99s", str);
```

при котором будет прочитано не более указанного после % количества литер, но чтения завершится раньше, если будет встречен пробельный символ. Таким образом, если под строку str отведено n байтов памяти, то следует читать в нее не более n-1 символа.

Функции работы со строками содержатся в библиотеке `string.h`. Для вычисления длины строки используют функцию `strlen`. Она возвращает длину своего аргумента-строки. Длиной строк называется количество символов до нулевого байта, не считая его. Например, результат `strlen("cat")` равен `3`.

Самая короткая строка – пустая `""`. Ее длина равна 0, а в памяти по ее адресу сразу находится нулевой байт. Формальный параметр функции `strlen` задан как константа, поскольку для вычисления длины строки не требуется менять строку. Для описания таких аргументов используют слово const. Указав его, мы даем возможность использовать функцию как для константных объектов, так и для изменяемых. Если же слово const опустить, то функция не будет работать с константами, так как не гарантирует, что они останутся неизменными.

Напишем заголовок функции, которая будет работать как библиотечная `strlen`, но добавим к ее названию префикс my, чтобы иметь возможность использовать как нашу функцию, так и библиотечную:

```
int mystrlen(const char *a) или
int mystrlen(char const *a)
```

Два указанных заголовка эквивалентны. А вот следующий

`int mystrlen(char * const a)` – представляет собой другое, не подходящее в нашем случае описание, т.к. в пр таком заголовкефункция обещает не менятьзначение локальной переменной `a` (параметра функции), а нам нужно запретить менять строку, то есть содержимое, находящееся по адресу `a`. Поэтому подойдёт любой из первых двух вариантов записи заголовка, но не третий.

Функция вычисления длины строки может быть реализована следующим образом:

```
int mystrlen(const char *a)
{
   int k = 0;
   while (*a++) k++; /* двигаемся по строке до нулевого байта*/
   return k;
}
```

Особого внимания заслуживает условие цикла `while`. В выражении `*a++` первой должна выполняться операция ++. Поскольку она постфиксная, ее результат — адрес `a` до увеличения. Именно к этому адресу будет применена операция `*`. То есть выражение позволит взять значение по адресу a, произойдет его сравнение с 0, а побочным эффектом выражения будет перестановка указателя на следующий символ строки.

Для копирования содержимого одной строковой переменной в другую строковую переменную нельзя использовать операцию присваивания. Строковая переменная – это указатель на начало строки (как и любой массив), причём это константный указатель, менять который запрещено. Поэтому, если есть две строковых переменных

```
char str1[100] = "cat", str2[100] = "dog";
```

то попытка скопировать содержимое `str2` в переменную `str1` с помощью операции присваивания

```
str1 = str2;
```

приведёт к ошибке, т.к. это попытка константному указателю что-то присвоить. При копировании содержимого одной строки в другую строку используют функцию `strcpy` с двумя аргументами – куда и откуда произвести копирование. В нашем случае нужен вызов функции `strcpy(str1, str2)`. Заметим, что в качестве второго аргумента может быть и строковая константа – явно заданная строка в двойных кавычках, например, `strcpy(str1, "mouse")`, что по сути является записью указанного значения в строковую переменную. Первый аргумент функции должен быть таким, чтобы все символы второй строки и нулевой байт после них поместились в выделенную память. При использовании библиотечной функции следить за этим необходимо тому, кто функцию вызывает.

Напишем реализацию функции, работающей подобно `strcpy`. Первый аргумент-строка подлежит изменению (в неё будет происходить запись нового значения), второй аргумент меняться не должен, поэтому для его описания используем `const`.

```
void mystrcpy(char *s, const char *t)
{  while (*t)  /* пока не конец строки */
      *s++ = *t++;
   *s = 0;
/* после копирования содержимого запись конца строки */
}
```

Функцию можно реализовать короче. используя побочные эффекты операций присваивания и постфиксный ++:

```
void mystrcpy(char *s, const char *t)
{
   while (*s++ = *t++);  /* пока не конец строки копируем и
                            двигаем указатели */
}
```

В этом варианте выход из цикла происходит сразу после копирования нулевого байта – признака конца строки. Иногда значение нулевого байта записывают в виде `'\0'`, чтобы указать его связь с записью строк. Но эта запись так же означает значение ноль.

Операция сравнения `str1 == str2` сравнивает значения указателей, а не содержимого двух строк. Результат этой операции отвечает на вопрос: установлены ли указатели на одну и ту же область памяти? Однако на практике требуется и другое сравнение – сравнение содержимого строк. Для него используется функция `strcmp` с двумя константными аргументами-строками. Значение `strcmp` равно нулю, если строки равны. Сравнение строк – лексикографическое, до первого различающегося символа. Функция возвращает разность кодов несовпадающих символов, т.е. какое-то отрицательное значение, если первая строка лексикографически меньше второй, или положительное значение, если первая строка лексикографически больше второй.

Напишем аналог библиотечной функции `strchr`, которая ищет в строке вхождение заданного символа и возвращает его адрес (если он есть в строке) или `NULL`, если в строке нет такого символа.

```
char *mystrchr(const char *s, char ch)
{  while (*s)
     { if (*s == ch) return s;
       s++;  }
   return NULL;
}
```

Для обработки строки с конца подойдёт следующий алгоритм: установить указатель на конец строки и обрабатывать строку, уменьшая указатель, пока он не совпадет с началом строки.

```
char *p;
p = str + strlen(str) - 1;
do {
…//работа со строкой
} while (p-- != str)
```

Задачи

При решении задачи данного параграфа рекомендуется написать функции ***без использования операции [ ]*** *(индексирования).*

8.1 Описать строку s для хранения не менее 20 символов. Прочитать со стандартного ввода в s строку, и вывести ее в стандартный вывод следующим образом:
а) добавить перед строкой ее длину и символ табуляции, далее вывести строку;
б) вывести строку 10 раз, добавив перед каждой выведенной строкой ее номер и двоеточие:
в) заменив ее первый символ на следующий в типе char;
г) заменив ее последний символ на следующий в типе char;
д) если введена строка "Hello", напечатать после нее в стандартный вывод ", MSU", иначе вывести только исходную строку;
е) заменив все заглавные латинские буквы в строке на соответствующие строчные;
ж) заменив каждую группу подряд идущих одинаковых символов одним таким символом.

8.2 Описать функцию `cmplen(s1,s2)` сравнивающие строки по длине. Для равных строк функция должна выдавать 0 в качестве результата. Если первая строка длиннее второй на k символов, результатом функции выдать `k`, если короче, то `-k`.

8.3 Описать функцию, меняющую строку заданным образом без использования дополнительных массивов:
а) заменить все пробелы в строке на символ '_';

б) если строка имеет четную длину, удалить вторую половину строки, если длина нечетна , оставить строку без изменений;
в) удалить из строки все пробелы;
г) перенести все пробелы в конец строки;
д) заменить все вхождения ing на ed;
е) перевернуть содерджимое строки

8.4 Описать функцию сравнения строк без учета пробелов. Пример :строки "12 3 4" и " 1234 " функция должна считать равными.

8.5 Не используя функции библиотеки string.h, реализовать аналогичные им (добавив в название префикс my). Добавить возможность работы с константными строками там, где сооитветствующие действия возможны, дополнив параметры квалификатором const.
а) `char* mystrrchr(char* s, char c)` – для поиска последнего (самого правого) вхождения заданного символа `c` в строку `s.` Вернуть адрес найденного символа или NULL, если символа `c` в строке `s` не оказалось;
б) `void mystrcat (char* s,char* t)` – для приписывания второй строки-параметра `t` в конец первой строки `s` (считать, что в первой строке выделено достаточно памяти для выполнения копирования)
с) `char* mystrstr (char* s,char* t)` – возвращает указатель на первое вхождение второй строки `t` в первую `s` или NULL, если такового не оказалось.

8.6 Допустимы ли следующие описания в языке С (считать, что параметры функции — строки, оканчивающиеся на `'\0'`)? Если да, описать семантику действий, сравнить работу функций с известными библиотечными из `string.h` При сравнении обратить особое внимание на работу со строками разной длины.
а)
```
int ques ( char *s1, char *s2){
 while (*s1 && *s2 && *s1++ == *s2++ );
 return *--s1 - *--s2;
}
```
б)
```
void ques ( char *s1, char *s2, int n){
  while (*s1 && *s2 && n-- && (*s1 ++ = *s2 ++ ) );
}
```

8.7 Написать рекурсивную функцию, проверяющую, является ли заданная ей строка-палиндромом.

8.8 Написать рекурсивную функцию с одним параметром, переворачивающую содержимое строки, не вводя определения дополнительных функций.

Тема 9.

Схема трансляции программы на С.

Препроцессор

При выполнении команды трансляции программы на С с помощью
`gcc file.c` или `clang file.c`
текст программы проходит несколько этапов преобразований. Конечной целью является создание исполняемого файла — последовательности машинных команд, выполнив которую мы и решим задачу. В UNIX – системах это обычно либо файл без расширения, либо с расширением out, (по умолчанию - a.out) для Windows – систем исполняемый файд традиционно имеет расширение exe. Для получения такого файла последовательно выполняется следующие этапы преобразований исходного текста программы:

1. Работа препроцессора;
2. Трансляция в ассемблер;
3. Построение объектного модуля (ассемблирование);
4. Компоновка.

Можно остановить работу компилятора после выполнения любого шага, указав в команде соответствующий ключ. Используются следующие опции:
`-E` – остановка после работы препроцессора,
`-s` – остановка после трансляции в Ассемблер,
`-c` – остановка после получения объектного модуля.

Чтобы указать, куда поместить результат выполнения очередного этапа нужно указать при трасляции опцию `-o <имя>`, тогда в файл с указанным именем и будет сохранен результат. Например, команда:
`gcc -E file.c -o file.i`
остановит трансляцию после первого этапа обработки — препроцессирования, а ее результат будет в файле `file.i`. Обычно именно расширение `.i` соответствует файлам, обработанным препроцессором языка С. Без указания имени файла – результата, препроцессор просто распечатывает преобразованный текст программы в стандартынй вывод.

Результатом трансляции программы в ассемблер будет также текстовый файл, но на низкоуровневом языке ассемблера: большинство его строк — команды, которые далее напрямую будут переведены в машинные команды. Из последовательности таких машинных команд и состоит результат следующего этапа трансляции – объектный модуль, но он еще не является исполняемым файлом, его необходимо объединить с используемыми библиотечными функциями, с другими объектными модулями — эта работа делается на этапе компоновки. Результатом компоновки является, наконец, исполняемый файл.

Программирование на языке ассемблера, компоновка программы из нескольких файлов будет далее рассмотрена в рамках курса «Архитектура ЭВМ и язык ассемблера» во втором семестре. Сейчас остановимся подробнее на работе препроцессора.

Что происходит на первом этапе во время работы препроцессора? Во-первых, из программы удаляются комментарии: они не участвуют в построении машинных команд и будут исключены из дальнейших этапов трансляции. Во-вторых, работа препроцессора заключается в выполнении текстовых подстановок – замене одних фрагментов текста другими. Результат работы препроцессора – тоже текстовый файл — последовательность символов, которая может быть выведена на экран или просмотрена в текстовом редакторе. Препроцессор обрабатывает специальные строки – директивы, они начинаются с символа #. Мы уже использовали такие директивы для подключения стандартных библиотек, например, библиотеки ввода-вывода `stdio.h`. Рассмотрим подробнее директивы препроцессора.

Первая уже использовавшаяся директива – включение содержимого другого файла в свою программу – директива `include`. Обычно эти директивы пишутся в самом начале программы. Имя библиотечного файла указывается в угловых скобках, имя пользовательского файла – в двойных кавычках. Если позовательский файл находится в том

же каталоге, что и программа, то достаточно указать только имя файла, в противном случае необходимо указывать полное имя файла, включающее путь к нему. Пример (содержимое файла `f.c`):

```
#include <stdio.h>
#include "test.h"
int main()
…
```

В этом примере `test.h` – это созданный нами текстовый файл с каким-либо содержимым. Текст этого файла будет подставлен перед функцией `main` вместо второй строки.

Результат подстановки содержимого указанного в директиве `include` файла можно посмотреть, выполнив указанную выше команду трансляции с опцией `-E`.

Следующая директива препроцессору – это директива `define`. Формат этой директивы следующий:

```
#define <имя>[<параметры>] <текст, пустой или непустой>
```

Директива записывается с начала строки, используется для определения так называемых ***макросов***.

Встретив такую директиву, препроцессор далее по тексту программы, производит текстовую замену указанного имени (возможно, с параметрами) на указанный текст (до перевода строки). Такая замена называется макроподстановкой. Если текст, на который требуется заменять имя, достаточно длинный и занимает несколько строк, то в конце промежуточных строк добавляется символ \.

```
#define TXT This is an example of a very -\
             very - very – very -\
             very long text
```

Отменить текстовую подстановку вместо соответствующего имени можно с помощью директивы `undef`:

```
#undef <имя>
```

Начиная с этой строки макроподстановки для этого имени уже происходить не будут.

Для чего можно использовать такие подстановки? Самый простой вариант – подстановка вместо имени какого-то значения. Например, имя может обозначать размер массива. Тогда при изменении размера потребуется внести изменения только в `#define`:

```
#define N 100
…
int A[N], Number;
…
readarray(A, N);
…
for(i = 0; i < N, ++i) …
…
```

В данном примере имя N играет роль константы. При описании такой константы в `define` слева и справа от неё должны быть разделители (пробельные символы). Указанное в `define` имя `N` заменяется на строку (`100` – это строка, последовательность из трёх символов) везде в тексте программы, за исключением двух случаев:

1. если это имя `N` является частью идентификатора, то замены не будет. Например, в имени `Number` замены N не будет произведено;
2. если это имя `N` содержится внутри строки, например, `"N=%d"`, то тоже замены произведено не будет.

Если где-то ниже встретится директива `undef`

```
#undef N
```

то с этой строки далее по тексту файла, замена N на 100 уже производиться не будет, пока не будет нового `#define N`.

Макрос с параметрами

Директива define используется не только для описания текста констант, как в примере выше, но и для написания макросов с параметрами. Подобно функции, макрос представляет собой механизм повторного использования кода программы. Чтобы не писать несколько раз один и тот же фрагмент, мы задаем его имя и (возможно) параметры. Макрос отличается от функции тем, что представляет собой текстовую замену одних фрагментов текста программы на другие, в то время как вызов функции – это передача управления на другой участок кода, в котором описана функция, с возвратом из него в точку вызова. В случае, когда действие большое и сложное, лучше его оформлять в виде функции. Если действие маленькое и компактное – для него подойдёт макрос.

Параметры макроса записываются после имени макроса в круглых скобках через запятую, в записи имени и параметров макроса не должно быть пробелов, а вот после них – как минимум один пробел перед телом макроса. При обработке макровызова (обращения к макросу в тексте программы) сначала формируется строка подстановки (макрорасширение) – в тело макроса вместо имён параметров подставляются выражения, заданные в качестве аргументов макроса. Сформированная таким образом строка подставляется в текст программы вместо макровызова. Можно считать, что слева и справа от подставленного текста добавятся пробелы – текст не «склеится» с остальными конструкциями программы.

Пример простейшего макроса:

```
#define sum(x,y) x+y
```

При обработке макровызова `sum(20,30)` этот макровызов заменится на выражение `20+30` (текстовая замена на сформированную по заданным параметрам строку подстановки). Но всегда ли такая текстовая замена будет корректна? Если макровызов является подвыражением более сложного выражения, то такая подстановка может оказаться некорректной. Например, выражение `10*sum(20+30)` в результате макорасширения преобразуется в выражение `10*20+30`, которое имеет другой смысл, чем исходное выражение, из-за более высокого приоритета операции умножения. Исправить эту ситуацию можно, заключив выражение – тело макроса – в скобки:

```
#define sum(x,y) (x+y)
```

Но и этот вариант описания тоже может привести к тому, что выражение поменяет смысл, т.к. в качестве аргументов макроса `sum` могут быть заданы выражения, содержащие операции с меньшим приоритетом, чем сложение. Например, выражение `10*sum(a=1,30)` в результате обработки макровызова заменится на макрорасширение `10*(a=1+30)`, которое имеет другой смысл по сравнению с исходным выражением. Для исправления этой ситуации необходимо в теле макроса взять в скобки каждый из аргументов макроса:

```
#define sum(x,y) ((x)+(y))
```

Такой вариант определения макроса `sum` будет правильно обрабатываться в любом контексте вычислений. Поэтому при написании макроса следует придерживаться правила: нужно заключать в скобки каждое вхождение параметров макроса, и все тело макроса, если оно является выражением.

Рассмотрим пример определения макроса `swap`, который меняет местами значения заданных в аргументах переменных заданного в аргументе типа (обе переменных должны быть одного типа, совпадающего с заданным). Для этого будем использовать локальную вспомогательную переменную нужного типа и последовательность операторов выражение. Сразу возникает вопрос, какое имя задать вспомогательной переменной, чтобы оно никогда не совпало с именем одной из переменных-аргументов. В языке С используют договоренность, что локальные имена внутри макросов начинаются с символа подчёркивания _. Это как раз сделано с целью обеспечения уникальности этих имён – для переменных и функций программы, напротив, не рекомендуется использовать такие имена. Макроопределение буцдет выглядеть следующим образом:

```
#define swap(a,b,t) {t _x; _x=a; a=b; b=_x;}
```
Тогда макровызов `swap(x,y,int)` будет заменён на
```
{int _x; _x=x; x=y; y=_x;}
```
Всё вроде бы правильно, но если в программе после макровызова поставят точку с запятой, например:
```
if (t>0) swap(x,y,int); else printf("0\n");
```
то после подстановки возникнет ошибка – точка с запятой будет воспринята как ещё один пустой оператор после составного оператора `{}`, а после условия `if` перед `else` по синтаксису языка должен быть ровно один оператор. Для предотвращения этой ошибки можно использовать такой прием: заключить составной оператор внутрь оператора `do-while` с условием-нулём, чтобы этот оператор цикла выполнился ровно один раз:
```
#define swap(a,b,t) do {t _x; _x=a; a=b; b=_x;} while(0)
```
Если тело макроса слишком длинное, то часть тела можно перенести на следующую строку, указав в конце первой части-строки символ \.

В языке С нет встроенных функций вычисления минимума и максимума, однако их несложно описать с помощью макросов. Например, макрос для вычисления максимума из двух аргументов будет выглядеть следующим образом:
```
#define max(x,y) ((x)>(y)?(x):(y))
```
Для выражения `max(10,30)*10` его макрорасширением будет выражение
```
((10)>(30)?(10):(30))*10
```
Но для выражения `max(++a,++b)` его макрорасширением будет выражение
```
((++a)>(++b)?(++a):(++b))
```
в котором у переменных `a` и `b` будет дважды выполнен побочный эффект. Эта проблема не решается, поэтому не используйте макросы с аргументами, в которых есть побочный эффект – результат вычислений может получиться не тот, ожидался.

Для отмены макроса с текущего места используется та же макрокоманда `undef`:
```
#undef max
```
С этого места макровызовы обрабатываться уже не будут, как и в случае отмены определения константы. Для отмены макроса достаточно указать только имя макроса без параметров.

Некоторые встроенные макросы:

`__LINE__`	номер строки, в которой встретился
`__DATE__`	дата компиляции
`__FILE__`	имя файла, в котором встретился
`__LINUX__`	если работаетв в операционной системе linux , то определен и рпревратится в 1, для других систем имя нем будет определено (для них, как правило, тоже есть соотвектствующие макросы).

Условная компиляция

Условная компиляция – это условные конструкции – директивы компилятору (транслятору), которые обрабатываются на этапе препроцессора. Их общий вид следующий:
```
#if E1
    TEKCT1
#elif E2
    TEKCT2
#else
    TEKCT3
#endif
```

Ветвей `elif` может быть любое количество, ветвь `else` может отсутствовать. В каждой из ветвей могут быть вложенные IF, их обработка всегда корректна, т.к. каждая строка с `if` закрывается своим `endif`.

Условия проверки должны быть такими, чтобы их можно было вычислить на этапе препроцессора, до выполнения программы. Примеры таких проверок: в какой ОС работаем, определено ли выше в тексте какое-то имя и т. д. Часто используются модицикации `if` проверяющие, что заданное имя было ранее объявлено как макрос:

`#ifdef <имя>`

условие считается истинным (выполнено), если указанное имя было ранее объявлено с помощью `#define` и не было «забыто» с помощью `#undef`.

И, наоборот, при анализе условия

`#ifndef <имя>`

проверяется, что имя не было ранее объявлено как макрос.

В качестве примера использования условной компиляции покажем возможность включать или выключать отладочную печать. Можно описать макрос DPR, который при наличии ранее строки

`#define Debug`

будет заменяться на печать своего аргумента, при выключенном режиме отладки (макрос Debug не объявлен)– на пусто:

```
#ifdef Debug
#define DPR(x) printf(x)  /* отладочная печать */
#else
#define DPR(x) /* замена макровызова на пустоту */
#endif
```

Тогда в программе можно использовать макровызовы DPR с отладочной печатью:

```
DPR("Before For");
for(i=…  ) …
…
DPR("Before While");
while(a) …
…
DPR("End");
```

Тогда для включения режима отладки нужно в самом начале, до условной конструкции препроцессора, определить константу `Debug` с помощью директивы `#define`, а для выключения этого режима – убрать или поместить в комментарий директиву определения имени `Debug`.

С помощью директив условной компиляции и встроенных макросов (`__clang__`, `__GNUC__`) мы можем выбрать фрагмент программы, который будет подставлен при использовании конкретного компилятора. Например, следующий фрагмент программы

```
#ifdef __clang__
    printf("Clang compiler\n");
#elif __GNUC__
    printf("GCC compiler\n");
#else
    printf("Undefined compiler\n");
#endif
```

при выполнении напечатает название известного ей компилятора, который используется, причем после работы препроцессора только один из указанных `printf` останется в тексте программы, остальные из текста будут удалены препроцессором наряду с самими строками – директивами.

Задачи

9.1 Написать результат работы препроцессора с файлом, содержащим следующую

программу на языке С.

а)
```
//Программа 1

#define K 10*
#define M 1;
int main (){
int i,s=0;
for (i=M K M) s*=i;
#undef M
{ int M=-1;

  s*=M;

}

#ifndef M

  return 0;

#else

return 1;

#endif

}
```

б)
```
//Программа 2
#define N1 12+
#define N2 10;
int main (){
int i,sum=0;
for (i=N2 N1 N2)
sum+=i;
#undef N2
{ int N2=1;

  sum+=N2;

}

#ifndef N2

return 1;

#else

  return 0;

#endif

}
```

9.2 Напишите макрос kmh_ms `(speed)`, который принимает значение скорости в км/ч и переводит его в метры в секунду.

9.3 Напишите макрос CtoF `(x)`, который переводит температуру x в градусах Цельсия (C) в градусы Фаренгейта (F). Формула перевода:

$$F = C \times \frac{9}{5} + 32$$

9.4 Описать макрос `min3(x,y,z)` для вычисления минимального из трех заданных значений. Чему будут равны результаты вычисления выражений

```
min3(a=0,z=2,a+=z)
min3(1,1,2)>1
```

при выполнении программы, использующей написанный макрос?

9.5 Описать макрос `cube(x)` для вычисления x^3. Чему будут равны результаты вычисления выражений

```
! cube(3)
cube(a=2)
(x=10,cube(x+1))
(x=2, cube(++x))
```

при выполнении программы, использующей написанный макрос?

9.6 Напишите фрагмент программы оплаты покупок интернет-магазина, используя директивы условной компиляции. Если программа работает в тестовом режиме, то должна

значением переменной unsigned long balance должно быть максимально допустимое число (его нужно присвоить в переменную), чтобы тестировать любые покупки. Если программа работает в режиме реального времени, то значение переменной balance объявить равным 0. Режим работы программы определяется наличием макроопределения TST_MODE: если такой макрос определен, то работаем в тестовом режиме, иначе — в обычном.

Литература

1. Л.С. Корухова, М.Р. Шура-Бура "Введение в алгоритмы" – М.: МГУ, 1997.

2. В.Н. Пильщиков, В.Г. Абрамов, А.А. Вылиток, И.В. Горячая "Машина Тьюринга и алгоритмы Маркова. Решение задач" – М.:ф-т ВМК МГУ, МАКС Пресс, 2016

3. Б. Керниган, Д. Ритчи "Язык программирования Си" 2-е изд. — М., С-Пб. : Диалектика (Вильямс), 2020.

4. Руденко Т.В. “Сборник задач и упражнений по языку Си (учебное пособие для студентов II курса)”– М.: Издательский отдел ВМК МГУ, 1999

5. В.Н. Пильщиков "Сборник задачи и упражнений по языку Паскаль" – М.: Научный мир, 2003