

Основы алгоритмизации и программирования

Лабораторная работа №5.

Цель работы - изучить указатели в языке программирования Си.

Содержание отчёта:

1. Титульный лист
2. Формулировка цели и задач работы
3. Описание результатов выполнения
4. Выводы, согласованные с целью

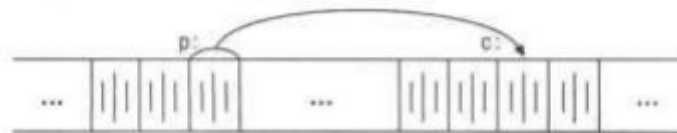
Указатели

Указатель — это переменная, содержащая адрес переменной. Указатели широко применяются в Си — отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и массивы тесно связаны друг с другом; в данной главе мы рассмотрим эту зависимость и покажем, как ею пользоваться.

Наряду с goto указатели когда-то были объявлены лучшим средством для написания малопонятных программ. Так оно и есть, если ими пользоваться бездумно. Ведь очень легко получить указатель, указывающий на что-нибудь совсем нежелательное. При соблюдении же определенной дисциплины с помощью указателей можно достичь ясности и простоты. Мы попытаемся убедить вас в этом.

Изменения, внесенные стандартом ANSI, связаны в основном с формулированием точных правил, как работать с указателями. Стандарт узаконил накопленный положительный опыт программистов и удачные нововведения разработчиков компиляторов. Кроме того, взамен `char *` в качестве типа обобщенного указателя предлагается тип `void *` (указатель на `void`).

Начнем с того, что рассмотрим упрощенную схему организации памяти. Память типичной машины представляет собой массив последовательно пронумерованных или проадресованных ячеек, с которыми можно работать по отдельности или связными кусками. Применительно к любой машине верны следующие утверждения: один байт может хранить значение типа `char`, двухбайтовые ячейки могут рассматриваться как целое типа `short`, а четырехбайтовые — как целые типа `long`. Указатель — это группа ячеек (как правило, две или четыре), в которых может храниться адрес. Так, если `s` имеет тип `char`, а `p` — указатель на `s`, то ситуация выглядит следующим образом:



Унарный оператор `&` выдает адрес объекта, так что инструкция `p = &s;` присваивает переменной `p` адрес ячейки `s` (говорят, что `p` указывает на `s`). Оператор `&` применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Его операндом не может быть ни выражение, ни константа, ни регистровая переменная. Унарный оператор `*` есть оператор косвенного доступа. Примененный к указателю, он выдает объект, на который данный указатель указывает. Предположим, что `x` и `y` имеют тип `int`, а `ip` — указатель на `int`.

Следующие несколько строк придуманы специально для того, чтобы показать, каким образом объявляются указатели и как используются операторы & и *.

```
int x = 1, y = 2, z[10];
int *ip; /* ip - указатель на int */
ip = &x; /* теперь ip указывает на x */
y = *ip; /* y теперь равен 1 */
*ip = 0; /* x теперь равен 0 */
ip = &z[0]; /* ip теперь указывает на z[0] */
```

Объявления `x`, `y` и `z` нам уже знакомы. Объявление указателя `ip` `int *ip;` мы стремились сделать mnemonic — оно гласит: "выражение `*ip` имеет тип `int`". Синтаксис объявления переменной "подстраивается" под синтаксис выражений, в которых эта переменная может встретиться. Указанный принцип применим и в объявлениях функций. Например, запись `double *dp, atof (char *)`; означает, что выражения `*dp` и `atof(s)` имеют тип `double`, а аргумент функции `atof` есть указатель на `char`.

Вы, наверное, заметили, что указателю разрешено указывать только на объекты определенного типа. (Существует одно исключение: "указатель на `void`" может указывать на объекты любого типа, но к такому указателю нельзя применять оператор косвенного доступа).

Если `ip` указывает на `x` целочисленного типа, то `*ip` можно использовать в любом месте, где допустимо применение `x`; например, `*ip = *ip + 10`; увеличивает `*ip` на 10. Унарные операторы `*` и `&` имеют более высокий приоритет, чем арифметические операторы, так что присваивание `y = *ip + 1` берет то, на что указывает `ip`, и добавляет к нему 1, а результат присваивает переменной `y`. Аналогично `*ip += 1` увеличивает на единицу то, на что указывает `ip`; те же действия выполняют `++*ip` и `(*ip)++`.

В последней записи скобки необходимы, поскольку если их не будет, увеличится значение самого указателя, а не то, на что он указывает. Это обусловлено тем, что унарные операторы `*` и `++` имеют одинаковый приоритет и порядок выполнения — справа налево.

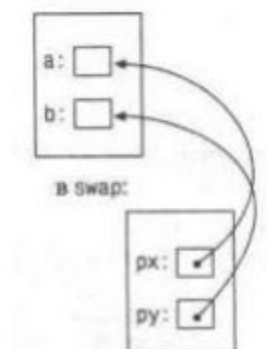
И наконец, так как указатели сами являются переменными, в тексте они могут встречаться и без оператора косвенного доступа. Например, если `iq` есть другой указатель на `int`, то `iq = ip` копирует содержимое `ip` в `iq`, чтобы `ip` и `iq` указывали на один и тот же объект.

Указатели и аргументы функций

Поскольку в Си функции в качестве своих аргументов получают значения параметров, нет прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции. В программе сортировки нам понадобилась функция `swar`, меняющая местами два неупорядоченных элемента. Однако недостаточно написать `swar(a, b)`; где функция `swar` определена следующим образом:

```
void swar(int *px, int *py) /* перестановка *px и *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Графически это выглядит следующим образом:



Аргументы-указатели позволяют функции осуществлять доступ к объектам вызвавшей ее программы и дают возможность изменить эти объекты. Рассмотрим, например, функцию `getint`, которая осуществляет ввод в свободном формате одного целого числа и его перевод из текстового представления в значение типа `int`. Функция `getint` должна возвращать значение полученного числа или сигнализировать значением `EOF` о конце файла, если входной поток исчерпан. Эти значения должны возвращаться по разным каналам, так как нельзя рассчитывать на то, что полученное в результате перевода число никогда не совпадет с `EOF`.

Одно из решений состоит в том, чтобы `getint` выдавала характеристику состояния файла (исчерпан или не исчерпан) в качестве результата, а значение самого числа помещала согласно указателю, переданному ей в виде аргумента.

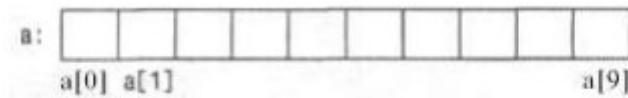
```
/* getint: читает следующее целое из ввода в *pn */
int getint(int *pn) {
    int c, sign; while (isspace(c = getc())); /* пропуск символов-разделителей */
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetc(c); /* не число */
        return 0;
    }
    sign = (c == '-' ) ? -1 : 1;
    if (c == '+' || c == '-')
        c = getc();
    for (*pn = 0; isdigit(c); c = getc())
        *pn = 10 * *pn + (c - '0' );
    *pn *= sign;
    if (c != EOF)
        ungetc(c);
    return c;
}
```

Указатели и массивы

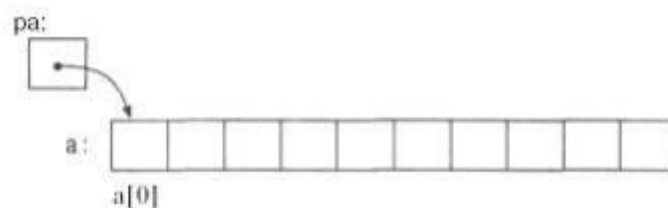
В Си существует связь между указателями и массивами, и связь эта настолько тесная, что эти средства лучше рассматривать вместе. Любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен с помощью указателя.

Вариант с указателями в общем случае работает быстрее, но разобраться в нем, особенно непосвященному, довольно трудно.

Объявление `int a[10]`; определяет массив `a` размера 10, т. е. блок из 10 последовательных объектов с именами `a[0]`, `a[1]`, ..., `a[9]`.

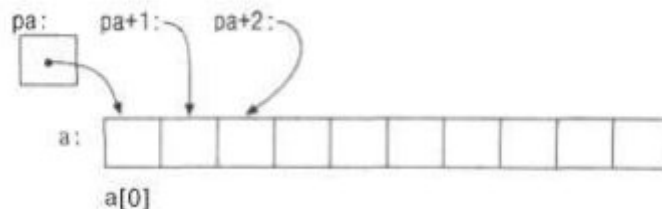


Запись `a[i]` отсылает нас к `i`-му элементу массива. Если `pa` есть указатель на `int`, т. е. объявлен как `int *pa`; то в результате присваивания `pa = &a[0]`; `pa` будет указывать на нулевой элемент `a`, иначе говоря, `pa` будет содержать адрес элемента `a[0]`.



Теперь присваивание `x = *pa`; будет копировать содержимое `a[0]` в `x`.

Если `pa` указывает на некоторый элемент массива, то `pa+1` по определению указывает на следующий элемент, `pa+i` — на `i`-й элемент после `pa`, а `pa-i` — на `i`-й элемент перед `pa`. Таким образом, если `pa` указывает на `a[0]`, то `*(pa+1)` есть содержимое `a[1]`, `a+i` -адрес `a[i]`, а `*(pa+i)` — содержимое `a[i]`.



Сделанные замечания верны безотносительно к типу и размеру элементов массива `a`. Смысл слов "добавить 1 к указателю", как и смысл любой арифметики с указателями, состоит в том, чтобы `pa+1` указывал на следующий объект, а `pa+1` — на 1-й после `pa`.

Между индексированием и арифметикой с указателями существует очень тесная связь. По определению значение переменной или выражения типа массив есть адрес нулевого элемента массива. После присваивания `pa = &a[0]`; `pa` и `a` имеют одно и то же значение. Поскольку имя массива является синонимом расположения его начального элемента, присваивание `pa=&a[0]` можно также записать в следующем виде: `pa = a`;

Еще более удивительно (по крайней мере на первый взгляд) то, что `a[i]` можно записать как `*(a+i)`. Вычисляя `a[i]`, Си сразу преобразует его в `*(a+i)`; указанные две формы записи эквивалентны. Из этого следует, что полученные в результате применения оператора `&` записи `&a[i]` и `a+i` также будут эквивалентными, т. е. и в том и в другом случае это адрес 1-го элемента после `a`. С другой стороны, если `pa` — указатель, то его можно использовать с индексом, т. е. запись `pa[i]` эквивалентна записи `*(pa+i)`. Короче говоря, элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. Указатель — это переменная, поэтому можно написать `ra=a` или `ra++`. Но имя массива не является переменной, и записи вроде `a=ra` или `a++` не допускаются.

Если имя массива передается функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот аргумент является локальной переменной, содержащей адрес. Мы можем воспользоваться отмеченным фактом и написать еще одну версию функции `strlen`, вычисляющей длину строки.

```
/* strlen: возвращает длину строки */
int strlen(char *s) {
    int n;
    for (n = 0; *s != '\0' ; s++)
        n++;
    return n;
}
```

Так как переменная `s` — указатель, к ней применима операция `++`; `s++` не оказывает никакого влияния на строку символов функции, которая обратилась к `strlen`. Просто увеличивается на 1 некоторая копия указателя, находящаяся в личном пользовании функции `strlen`. Это значит, что все вызовы, такие как: (правомерны).

```
strlen("Здравствуй, мир"); /* строковая константа */
strlen(array); /* char array[100]; */
strlen(ptr); /* char *ptr; */
```

Формальные параметры `char s[]`; и `char *s`; в определении функции эквивалентны. Мы отдаем предпочтение последнему варианту, поскольку он более явно сообщает, что `s` есть указатель. Если функции в качестве аргумента передается имя массива, то она может рассматривать его так, как ей удобно — либо, как имя массива, либо как указатель, и поступать с ним соответственно. Она может даже использовать оба вида записи, если это покажется уместным и понятным.

Функции можно передать часть массива, для этого аргумент должен указывать на начало подмассива. Например, если `a` — массив, то в записях `f(&a[2])` или `f(a+2)`.

Функции `f` передается адрес подмассива, начинающегося с элемента `a[2]`. Внутри функции `f` описание параметров может выглядеть как `f(int arr[]) {...}` или `f(int *arr) {...}` Следовательно, для `f` тот факт, что параметр указывает на часть массива, а не на весь массив, не имеет значения.

Если есть уверенность, что элементы массива существуют, то возможно индексирование и в "обратную" сторону по отношению к нулевому элементу; выражения `r[-1]`, `r[-2]` и т. д. не противоречат синтаксису языка и обращаются к элементам, стоящим непосредственно перед `r[0]`. Разумеется, нельзя "выходить" за границы массива и тем самым обращаться к несуществующим объектам.

Адресная арифметика

Если `r` есть указатель на некоторый элемент массива, то `r++` увеличивает `r` так, чтобы он указывал на следующий элемент, а `r += i` увеличивает его, чтобы он указывал на `i`-й элемент после того, на который указывал ранее. Эти и подобные конструкции — самые простые примеры арифметики над указателями, называемой также адресной арифметикой.

Си последователен и единообразен в своем подходе к адресной арифметике. Это соединение в одном языке указателей, массивов и адресной арифметики — одна из сильных его сторон. Проиллюстрируем сказанное построением простого распределителя памяти, состоящего из двух программ. Первая, `alloc(n)`, возвращает указатель `p` на `n` последовательно расположенных ячеек типа `char`; программой, обращающейся к `alloc`, эти ячейки могут быть использованы для запоминания символов. Вторая, `afree(p)`, освобождает память для, возможно, повторной ее утилизации. Простота алгоритма обусловлена предположением, что обращения к `afree` делаются в обратном порядке по отношению к соответствующим обращениям к `alloc`. Таким образом, память, с которой работают `alloc` и `afree`, является стеком (списком, в основе которого лежит принцип "последним вошел, первым ушел"). В стандартной библиотеке имеются функции `malloc` и `free`, которые делают то же самое, только без упомянутых ограничений.

Функцию `alloc` легче всего реализовать, если условиться, что она будет выдавать куски некоторого большого массива типа `char`, который мы назовем `allocbuf`. Этот массив отдадим в личное пользование функциям `alloc` и `afree`. Так как они имеют дело с указателями, а не с индексами массива, то другим программам знать его имя не нужно. Кроме того, этот массив можно определить в том же исходном файле, что и `alloc` и `afree`, объявив его `static`, благодаря чему он станет невидимым вне этого файла. На практике такой массив может и вовсе не иметь имени, поскольку его можно запросить с помощью `malloc` у операционной системы и получить указатель на некоторый безымянный блок памяти.

Естественно, нам нужно знать, сколько элементов массива `allocbuf` уже занято. Мы введем указатель `allocp`, который будет указывать на первый свободный элемент. Если запрашивается память для `n` символов, то `alloc` возвращает текущее значение `allocp` (т. е. адрес начала свободного блока) и затем увеличивает его на `n`, чтобы указатель `allocp` указывал на следующую свободную область. Если же пространства нет, то `alloc` выдает нуль. Функция `afree[p]` просто устанавливает `allocp` в значение `p`, если оно не выходит за пределы массива `allocbuf`.

Перед вызовом `alloc`:



После вызова `alloc`:



```
#define ALLOCSIZE 10000 /* размер доступного пространства */
```

```
static char allocbuf[ALLOCSIZE]; /* память для alloc */
static char *allocp = allocbuf; /* указатель на своб. место */
```

```
char *alloc(int n) /* возвращает указатель на n символов */
{
```

```

    if (allocbuf + ALLOCSIZE - allocp >= n) {
        allocp += n; /* пространство есть */
        return allocp - n; /* старое p */
    } else /* пространства нет */
        return 0;
}

void afree(char *p) /* освобождает память, на которую указывает p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}

```

В общем случае указатель, как и любую другую переменную, можно инициализировать, но только такими осмысленными для него значениями, как нуль или выражение, приводящее к адресу ранее определенных данных соответствующего типа. Объявление `static char *allocp = allocbuf;` определяет `allocp` как указатель на `char` и инициализирует его адресом массива `allocbuf`, поскольку перед началом работы программы массив `allocbuf` пуст. Указанное объявление могло бы иметь и такой вид: `static char *allocp = &allocbuf[0];` поскольку имя массива и есть адрес его нулевого элемента.

Проверка `if (allocbuf + ALLOCSIZE - allocp >= n) { /* годится */` контролирует, достаточно ли пространства, чтобы удовлетворить запрос на `n` символов. Если памяти достаточно, то новое значение для `allocp` должно указывать не далее чем на следующую позицию за последним элементом `allocbuf`. При выполнении этого требования `alloc` выдает указатель на начало выделенного блока символов (обратите внимание на объявление типа самой функции). Если требование не выполняется, функция `alloc` должна выдать какой-то сигнал о том, что памяти не хватает. Си гарантирует, что нуль никогда не будет правильным адресом для данных, поэтому мы будем использовать его в качестве признака аварийного события, в нашем случае нехватки памяти.

Указатели и целые не являются взаимозаменяемыми объектами. Константа нуль — единственное исключение из этого правила: ее можно присвоить указателю, и указатель можно сравнить с нулевой константой. Чтобы показать, что нуль — это специальное значение для указателя, вместо цифры нуль, как правило, записывают `NULL` — константу, определенную в файле `<stdio.h>`. С этого момента и мы будем ею пользоваться.

Проверки `if (allocbuf + ALLOCSIZE - allocp >= n) { /* годится */` и `if (p >= allocbuf && p < allocbuf + ALLOCSIZE)` демонстрируют несколько важных свойств арифметики с указателями. Во-первых, при соблюдении некоторых правил указатели можно сравнивать. Если `p` и `q` указывают на элементы одного массива, то к ним можно применять операторы отношения `==`, `!=`, `<`, `>=` и т. д. Например, отношение вида `p < q` истинно, если `p` указывает на более ранний элемент массива, чем `q`. Любой указатель всегда можно сравнить на равенство и неравенство с нулем. А вот для указателей, не указывающих на элементы одного массива, результат арифметических операций или сравнений не определен. (Существует одно исключение: в арифметике с указателями можно использовать адрес несуществующего "следующего за массивом" элемента, т. е. адрес того "элемента", который станет последним, если в массив добавить еще один элемент.) Во-вторых, как вы уже, наверное, заметили, указатели и целые можно складывать и вычитать. Конструкция `p + n` означает адрес объекта, занимающего `n`-е место после объекта, на который указывает `p`. Это справедливо безотносительно к типу объекта, на который указывает `p`; `n` автоматически домножается на коэффициент, соответствующий размеру объекта. Информация о размере неявно присутствует в объявлении `p`. Если, к примеру, `int` занимает четыре байта, то коэффициент умножения будет равен четырем.

Допускается также вычитание указателей. Например, если p и q указывают на элементы одного массива и $p < q$, то $q - p + 1$ есть число элементов от p до q включительно. Этим фактом можно воспользоваться при написании еще одной версии `strlen`:

```
/* strlen: возвращает длину строки s */
int strlen(char *s) {
    char *p = s;
    while (*p != '\0')
        p++;
    return p - s;
}
```

В своем объявлении p инициализируется значением s , т. е. вначале p указывает на первый символ строки. На каждом шаге цикла `while` проверяется очередной символ; цикл продолжается до тех пор, пока не встретится `'\0'`. Каждое продвижение указателя p на следующий символ выполняется инструкцией `p++`, и разность $p - s$ дает число пройденных символов, т. е. длину строки. (Число символов в строке может быть слишком большим, чтобы хранить его в переменной типа `int`. Тип `ptrdiff_t`, достаточный для хранения разности (со знаком) двух указателей, определен в заголовочном файле `<stddef.h>`. Однако, если быть очень осторожными, нам следовало бы для возвращаемого результата использовать тип `size_t`, в этом случае наша программа соответствовала бы стандартной библиотечной версии. Тип `size_t` есть тип беззнакового целого, возвращаемого оператором `sizeof`.)

Арифметика с указателями учитывает тип: если она имеет дело со значениями `float`, занимающими больше памяти, чем `char`, и p — указатель на `float`, то `p++` продвинет p на следующее значение `float`. Это значит, что другую версию `alloc`, которая имеет дело с элементами типа `float`, а не `char`, можно получить простой заменой в `alloc` и `afree` всех `char` на `float`. Все операции с указателями будут автоматически откорректированы в соответствии с размером объектов, на которые указывают указатели.

Можно производить следующие операции с указателями: присваивание значения указателя другому указателю того же типа, сложение и вычитание указателя и целого, вычитание и сравнение двух указателей, указывающих на элементы одного и того же массива, а также присваивание указателю нуля и сравнение указателя с нулем. Других операций с указателями производить не допускается. Нельзя складывать два указателя, перемножать их, делить, сдвигать, выделять разряды; указатель нельзя складывать со значением типа `float` или `double`; указателю одного типа нельзя даже присвоить указатель другого типа, не выполнив предварительно операции приведения (исключение составляют лишь указатели типа `void*`).

Символьные указатели

Строковая константа, написанная в виде "Я строка" есть массив символов. Во внутреннем представлении этот массив заканчивается нулевым символом `'\0'`, по которому программа может найти конец строки. Число занятых ячеек памяти на одну больше, чем количество символов, помещенных между двойными кавычками. Чаще всего строковые константы используются в качестве аргументов функций, как, например, в `printf("здравствуй, мир\n");`

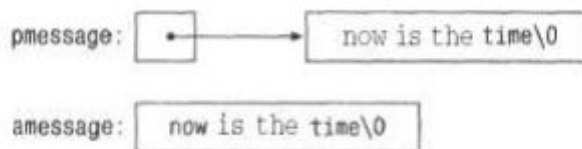
Когда такая символьная строка появляется в программе, доступ к ней осуществляется через символьный указатель; `printf` получает указатель на начало массива символов. Точнее, доступ к строковой константе осуществляется через указатель на ее первый элемент.

Строковые константы нужны не только в качестве аргументов функций. Если, например, переменную `pmessage` объявить как `char *pmessage` то присваивание `pmessage = "now is the time";` поместит в нее указатель на символьный массив, при этом сама строка не копируется, копируется лишь указатель на нее. Операции для работы со строкой как с единым целым в Си не предусмотрены.

Существует важное различие между следующими определениями:

```
char amessage[] = "now is the time"; /* массив */
char *pmessage = "now is the time"; /* указатель */
```

`amessage` — это массив, имеющий такой объем, что в нем как раз помещается указанная последовательность символов и `'\0'`. Отдельные символы внутри массива могут изменяться, но `amessage` всегда указывает на одно и то же место памяти. В противоположность ему `pmessage` есть указатель, инициализированный так, чтобы указывать на строковую константу. А значение указателя можно изменить, и тогда последний будет указывать на что-либо другое. Кроме того, результат будет неопределен, если вы попытаетесь изменить содержимое константы.



Дополнительные моменты, связанные с указателями и массивами, проиллюстрируем на несколько видоизмененных вариантах двух полезных программ, взятых нами из стандартной библиотеки. Первая из них, функция `strcpy(s, t)`, копирует строку `t` в строку `s`. Хотелось бы написать прямо `s=t`, но такой оператор копирует указатель, а не символы. Чтобы копировать символы, нам нужно организовать цикл. Первый вариант `strcpy`, с использованием массива, имеет следующий вид:

```
/* strcpy: копирует t в s; вариант с индексируемым массивом */
void strcpy(char *s, char *t) {
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Для сравнения приведем версию `strcpy` с указателями:

```
/* strcpy: копирует t в s: версия 1 (с указателями) */
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Поскольку передаются лишь копии значений аргументов, strcpy может свободно пользоваться параметрами s и t как своими локальными переменными. Они должным образом инициализированы указателями, которые продвигаются каждый раз на следующий символ в каждом из массивов до тех пор, пока в копируемой строке t не встретится '\0'.

На практике strcpy так не пишут. Опытный программист предпочтет более короткую запись:

```
/* strcpy: копирует t в s; версия 2 (с указателями) */
void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0')
        ;
}
```

Приращение s и t здесь осуществляется в управляющей части цикла. Значением *t++ является символ, на который указывает переменная t перед тем, как ее значение будет увеличено; постфиксный оператор ++ не изменяет указатель t, пока не будет взят символ, на который он указывает. То же в отношении s: сначала символ запомнится в позиции, на которую указывает старое значение s, и лишь после этого значение переменной s увеличится. Пересылаемый символ является одновременно и значением, которое сравнивается с '\0'. В итоге копируются все символы, включая и заключительный символ '\0'.

Заметив, что сравнение с '\0' здесь лишнее (поскольку в Си ненулевое значение выражения в условии трактуется и как его истинность), мы можем сделать еще одно и последнее сокращение текста программы:

```
/* strcpy: копирует t в s; версия 3 (с указателями) */
void strcpy(char *s, char *t) {
    while (*s++ = *t++)
        ;
}
```

Хотя на первый взгляд то, что мы получили, выглядит загадочно, все же такая запись значительно удобнее, и следует освоить ее, поскольку в Си-программах вы будете с ней часто встречаться.

Что касается функции strcpy из стандартной библиотеки <string.h>, то она возвращает в качестве своего результата еще и указатель на новую копию строки.

Вторая программа, которую мы здесь рассмотрим, это strcmp(s, t). Она сравнивает символы строк s и t и возвращает отрицательное, нулевое или положительное значение, если строка s соответственно лексикографически меньше, равна или больше, чем строка t. Результат получается вычитанием первых несовпадающих символов из s и t.

```
/* strcmp: выдает < 0 при s < t, 0 при s == t, > 0 при s > t */
int strcmp(char *s, char *t) {
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

Та же программа с использованием указателей выглядит так:

```
/* strcmp: выдает < 0 при s < t, 0 при s == t, > 0 при s > t */
int strcmp(char *s, char *t) {
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Поскольку операторы ++ и -- могут быть или префиксными, или постфиксными, встречаются (хотя и не так часто) другие их сочетания с оператором *. Например: *--p уменьшит p прежде, чем по этому указателю будет получен символ. Например, следующие два выражения: (являются стандартными для послыки в стек и взятия из стека).

```
*p++ = val; /* поместить val в стек */
val = *--p; /* взять из стека значение и поместить в val */
```

Используемые источники:

1. Язык программирования Си. Авторы: Б. Керниган, Д. Ритчи.
2. Онлайн компилятор языка Си: https://www.onlinegdb.com/online_c_compiler
3. Онлайн блок-схемы: <https://app.diagrams.net>

Задание:

При решении каждой задачи, необходимо создать блок-схемы. Решением работы должен служить файл-отчёт, в котором будут располагаться скриншоты кода, выполнения программ, а также сами блок-схемы.

1.
 - 1.1. Написать функцию для нахождения количества чётных чисел в массиве.
 - 1.2. Написать функцию для инвертирования массива.
 - 1.3. Написать функцию, которая вернёт массив всех чисел, встречающихся в строке.
2.
 - 2.1. Написать функцию для нахождения суммы всех чётных чисел в массиве.
 - 2.2. Написать функцию для удаления элемента из массива по индексу.
 - 2.3. Написать функцию для удаления всех повторяющихся символов в строке.
3.
 - 3.1. Написать функцию для нахождения среднего арифметического в массиве.
 - 3.2. Написать функцию для замены элемента в массиве по индексу.
 - 3.3. Написать функцию, которая вернёт позицию самого длинного слова в строке.
4.
 - 4.1. Написать функцию для нахождения максимального числа в массиве.
 - 4.2. Написать функцию для внесения нового элемента в массив по индексу.
 - 4.3. Написать функцию, которая вернёт количество слов в строке.
- 5.

5.1. Написать функцию для нахождения суммы всех чисел до первого отрицательного в массиве.

5.2. Написать функцию сдвига массива вправо на N-шагов.

5.3. Написать функцию, которая будет определять, является ли строка палиндромом.

6.

6.1. Написать функцию для нахождения количества всех отрицательных чисел в массиве.

6.2. Написать функцию, которая будет менять местами максимальный и минимальный элемент в массиве.

6.3. Написать функцию для удаления подстроки в строке.

7.

7.1. Написать функцию для нахождения произведения всех нечётных чисел в массиве.

7.2. Написать функцию, которая будет удалять все отрицательные числа в массиве.

7.3. Написать функцию для внесения подстроки в строку по указанной позиции.

8.

8.1. Написать функцию для нахождения всех чисел больших 15 и меньших 30 в массиве.

8.2. Написать функцию, которая будет сортировать массив методом выборки.

8.3. Написать функцию, которая определит количество строчных и прописных букв в строке.