5. Указатели и массивы

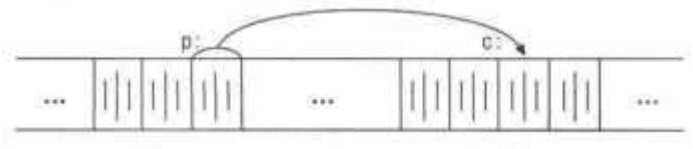
Указатель — это переменная, содержащая адрес переменной. Указатели широко применяются в Си — отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и массивы тесно связаны друг с другом; в данной главе мы рассмотрим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

5.1. Указатели и адреса

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 `c` имеет тип `char`, а `p` — указатель на `c`, то ситуация выглядит следующим образом:



Унарный оператор `&` выдает адрес объекта, так что инструкция

```
p = &c;
```

присваивает переменной `p` адрес ячейки `c` (говорят, что `p` указывает на `c`). 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 мнемоничным — 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. Мы вернемся к этому в параграфе 5. 11.)

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.

5.2. 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 `swap`, меняющая местами два неупорядоченных элемента. Однако недостаточно написать

```
swap(a, b);
```

где функция `swap` определена следующим образом:

```
void swap(int x, int y) /* НЕВЕРНО */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Поскольку `swap` получает лишь копии переменных `a` и `b`, она не может повлиять на переменные `a` и `b` той программы, которая к ней обратилась.

Чтобы получить желаемый эффект, вызывающей программе надо передать указатели на те значения, которые должны быть изменены:

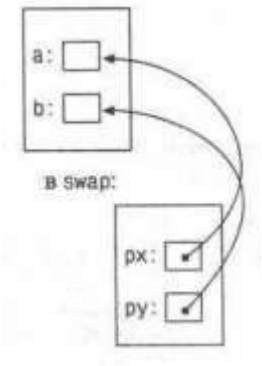
```
swap(&a, &b);
```

Так как оператор `&` получает адрес переменной, `&a` есть указатель на `a`. В самой же функции `swap` параметры должны быть объявлены как указатели, при этом доступ к значениям параметров будет осуществляться косвенно.

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

Графически это выглядит следующим образом:

в вызывающей программе:



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. Похожая схема действует и в программе `scanf`, которую мы рассмотрим в параграфе 7.4.

Показанный ниже цикл заполняет некоторый массив целыми числами, полученными с помощью `getint`.

```
int n, array[SIZE], getint (int *);
for (n = 0; n < SIZE && getint (&array[n]) != EOF; n++)
    ;
```

Результат каждого очередного обращения к `getint` посылается в `array[n]`, и `n` увеличивается на единицу. Заметим, и это существенно, что функции `getint` передается адрес элемента `array[n]` . Если этого не сделать, у `getint` не будет способа вернуть в вызывающую программу переведенное целое число.

В предлагаемом нами варианте функция `getint` возвращает `EOF` по концу файла; нуль, если следующие вводимые символы не представляют собою числа; и положительное значение, если введенные символы представляют собой число.

```
#include <ctype.h>

int getch (void);

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

Везде в `getint` под `*pn` подразумевается обычная переменная типа `int`. Функция `ungetch` вместе с `getch` (параграф 4.3) включена в программу, чтобы обеспечить возможность отослать назад лишний прочитанный символ.

Упражнение 5.1. Функция `getint` написана так, что знаки `-` или `+`, за которыми не следует цифра, она понимает как "правильное" представление нуля. Скорректируйте программу таким образом, чтобы в подобных случаях она возвращала прочитанный знак назад во ввод.

Упражнение 5.2. Напишите функцию `getfloat` — аналог `getint` для чисел с плавающей точкой. Какой тип будет иметь результирующее значение, выдаваемое функцией `getfloat`?

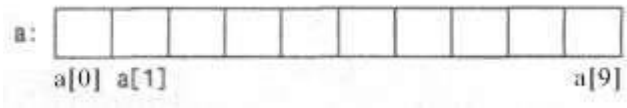
5.3. Указатели и массивы

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]`.



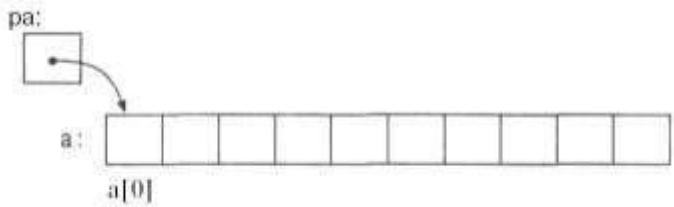
Запись `a[i]` отсылает нас к `i`-му элементу массива. Если ра есть указатель на `int`, т. е. объявлен как

```
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

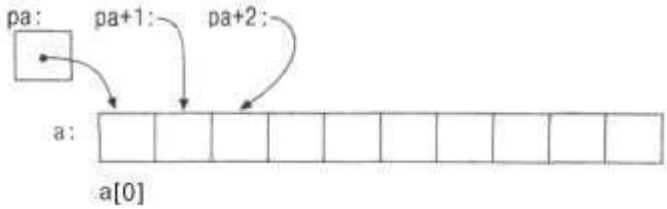
```
x = *pa;
```

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 pa=a или pa++. Но имя массива не является переменной, и записи вроде a=pa 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:

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

правомерны.

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 p[-1], p[-2] и т. д. не противоречат синтаксису языка и обращаются к элементам, стоящим непосредственно перед p[0]. Разумеется, нельзя "выходить" за границы массива и тем самым обращаться к несуществующим объектам.

5.4. Адресная арифметика

Если p есть указатель на некоторый элемент массива, то p++ увеличивает p так, чтобы он указывал на следующий элемент, а p += 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.

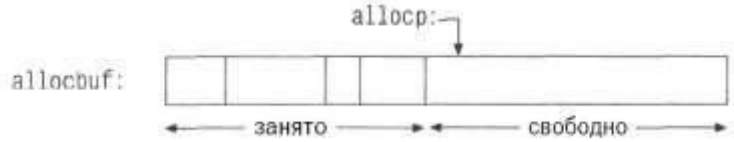
Си последователен и единообразен в своем подходе к адресной арифметике. Это соединение в одном языке указателей, массивов и адресной арифметики — одна из сильных его сторон. Проиллюстрируем сказанное построением простого распределителя памяти, состоящего из двух программ. Первая, alloc(n), возвращает 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; в параграфе 8.7 мы покажем, как они выглядят.

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.

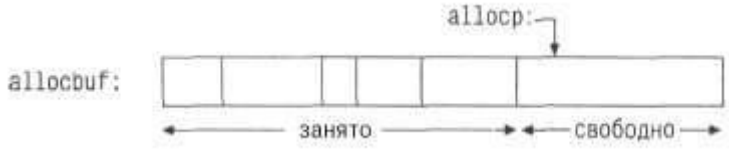
Естественно, нам нужно знать, сколько элементов массива allocbuf уже занято. Мы введем указатель allocp, который будет указывать на первый свободный элемент. Если запрашивается память для n

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 п 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*`).