Указатели

А. Г. Фенстер, `fenster@fenster.name`

20 февраля 2010 г.

На этой лекции мы разберём указатели — понятие, которое должно быть хорошо знакомо каждому, кто писал на C хотя бы несколько программ. К сожалению, практика показывает, что до самого конца первого курса (да и дальше) остаются студенты, которые не понимают эту действительно несложную тему. Надеюсь, что этот конспект уменьшит количество таких студентов.

Дополнительное чтение: статья (перевод; автор — Joel Spolsky).

1 Понятие указателя

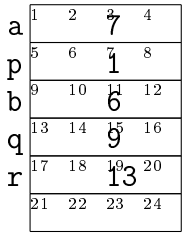
Начнём с самых простых, базовых вещей. Любой переменной соответствует место в памяти, в которой она размещается (практически любой — некоторые могут быть размещены на регистре процессора). Определить, где размещена переменная, можно при помощи *оператора взятия адреса* `&`, который может быть применён к переменной.

Полученный адрес не обязан совпадать (и не совпадает) с физическим адресом физической ячейки памяти — современные операционные системы поддерживают виртуальную память, про устройство которой вам расскажут в курсе операционных систем.

Любое значение в C имеет некоторый тип (`int`, `float`, ...); должны иметь тип и адреса ячеек памяти. Хотя физически они представляют собой целые числа, в языке выделены специальные типы для хранения адресов: адрес переменной типа `T` — значение типа `T *`. Итак, можно завести переменную типа, к примеру, `int *` и присвоить ей адрес некоторого значения типа `int` в памяти — т. е. адрес места в памяти, где это значение размещено. Такая переменная называется *указателем* на `int`.

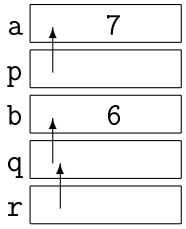
Если в некоторой вымышленной операционной системе байты доступной программе памяти нумеруются, начиная с единицы, то при выполнении следующего кода возможна ситуация, показанная на рисунке. Одна клетка соответствует четырём байтам — обычному размеру типа `int` и размеру указателя на машинах архитектуры x86.

```
int a = 7;    /* переменная a размещена в ячейке 1, содержит число 7 */
int *p = &a;  /* переменная p размещена в ячейке 5, содержит адрес переменной a (1)*/
int b = 6;    /* переменная b размещена в ячейке 9, содержит число 6  */
int *q = &b;  /* переменная q размещена в ячейке 13, содержит адрес переменной b (9) */
int **r = &q; /* переменная r размещена в ячейке 17, содержит адрес переменной q (13) */
```



Под адресом переменной мы понимаем адрес первой ячейки, которую занимает переменная: в примере выше `a` занимает четыре байта (номера 1, 2, 3, 4), но адресом её считается 1.

Это был первый и последний раз, когда мы придумывали «номера ячеек памяти», в будущем подобные картинки мы будем рисовать, используя стрелки. Сравните нарисованное выше и ниже:



Операция разыменования — получение значения ячейки памяти по её адресу, на рисунке разыменованию соответствует один проход по стрелке. Операция разыменования обозначается символом `*` слева от разыменуемого адреса. В примере выше выражение `*p` имеет значение 7, выражение `*q` — 6, а выражение `*r` равно адресу переменной `b` (таким образом, `**r` имеет значение 6).

Мысленно сдвигая «звёздочки» в определении переменных `p`, `q`, `r` влево или вправо, легко можно понять, какой тип имеет каждое из перечисленных выражений. Например, все перечисленные строки эквивалентны и определяют одну и ту же переменную `r`:

```
int **r; /*    **r имеет тип int      */
int* *r; /*    *r имеет тип int *     */
int** r; /*    r имеет тип int **     */
```

Мне больше нравится писать «звёздочки» ближе к имени переменной, потому что при определении двух и более переменных `int *x, y, *z;` «звёздочки» всегда относятся к именам переменных, а не к типу (таким образом, `y` будет иметь тип `int`, а `z` — тип `int *`.

Результат операции разыменования является *левосторонним значением* (lvalue), т. е. может стоять в левой части оператора присваивания. В примере на предыдущем рисунке присваивания

```
b = 8;
*q = 8;
**r = 8;
```

выполняют одно и то же действие: изменяют значение переменной `b`.

Тот факт, что указатели являются типизированными, важен для операции разыменования: её результатом является значение именно того типа, на который указывает данная переменная-указатель. Здесь возможны интересные эффекты:

```
int a = 11 * 256 * 256 * 256 + 22 * 256 * 256 + 33 * 256 + 44;
        /* 0x11223344 */
int *p = &a;
char *q = (char *)p; /* явное приведение типа */
printf("%d\n", *q);
```

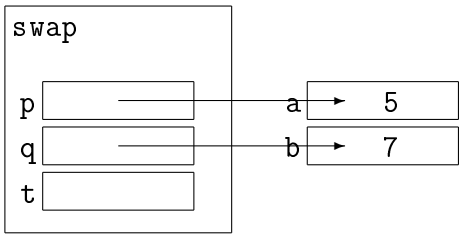
В этом примере `q` имеет то же значение, что и `p`, но результатом операции разыменования станет значение первого байта из (обычно) четырёх, формирующих значение типа `int`. В зависимости от используемой архитектуры (little-endian или big-endian) будет напечатано либо 44, либо 11.

2 Передача адресов значений в функцию

Первое (и, возможно, наиболее частое) применение указателей в C — создание функций, которые могут изменять значения переданных им фактически параметров. Смысл трюка заключается в том, что функции при вызове передаётся **адрес** интересующего значения, который сам по себе не изменяется (вспоминаем конец предыдущего занятия), но при этом функция может изменить находящееся по этому адресу значение. Классический пример — функция `swap`:

```
void swap(int *p, int *q)
{
    int t = *p;
    *p = *q;
    *q = c;
}

...
int a = 5, b = 7;
swap(&a, &b); /* передаём адреса интересующих переменных */
```



Другой классический пример — функция `scanf`, которая не смогла бы сохранить прочитанное значение в переменной, если бы мы не передавали адрес переменной для сохранения:

```
int a;
scanf("%d", &a); /* передали адрес переменной a */
```

3 Указатели и массивы

Оказывается, имя любого массива — это по сути неизменяемый указатель на начало (нулевой элемент) этого массива. Более того, здесь впервые пригождается знание того, адрес значения какого типа хранится в переменной-указателе: оказывается, добавление числа к указателю даёт новый указатель, сдвинутый на такое же количество элементов этого

типа вправо. Так как массивы всегда занимают в памяти непрерывный блок ячеек, получаем, что для массива `int A[5]` адрес i-го элемента `&A[i]` совпадает с `A + i`. Вообще говоря, выполняется тождество

```
A[i]    ==    *(A + i)
```

Более того, благодаря коммутативности сложения допускается (но не приветствуется) даже запись `i[A]` (и даже `0[A]`, `1[A]` и так далее).

4 Передача массивов в функцию

При передаче массивов в функцию два следующих описания полностью эквивалентны:

```
void f(int A[5])
{
    ...
}
```

```
void f(int *A)
{
    ...
}
```

Более того, число 5, указанное в первом случае, никакого значения не имеет и функции не доступно. Проще говоря, при передаче в функцию массива функция получает лишь адрес его начала, а определить количество элементов в нём не может никак — это забота программиста (передавать длину отдельно либо вставлять информацию о длине прямо в массив, как это сделано со строками и символом с кодом 0 — про это подробнее мы поговорим позже).

Если же функции передаётся двумерный массив, то необходимо указать **второй** его размер — без него функция не сможет определить, как вычислять положение элемента в массиве (нарисуйте двумерный массив в виде матрицы, чтобы понять, зачем нужно знать второй размер). Первый размер можно опустить. Опять же, следующие примеры эквивалентны:

```
void f(int A[3][5])
{
    ...
}
```

```
void f(int A[][5])
{
    ...
}
```

```
void f(int (*A)[5])
{
    ...
}
```

Странная запись в третьем примере — *указатель на массив* из пяти элементов. Идентификатор `A` будет указывать на начальный (нулевой) блок из пяти элементов, `A + 1` — на следующий (то есть на следующую строку массива), и так далее.

В общем случае, при передаче N-мерного массива нужно указать все размеры, кроме самого левого.

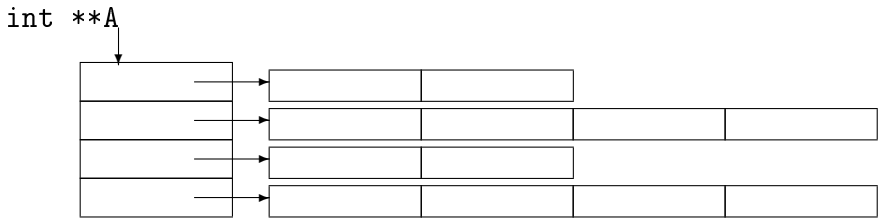
5 Выделение и освобождение памяти

Зачастую нам необходимо сделать так, чтобы некоторые данные в памяти находились после выхода из блока, в котором мы определили массив с этими данными. Если статическое размещение нас по каким-либо причинам не устраивает, приходится брать управление выделением и освобождением памяти в свои руки и использовать функции `malloc`, `realloc`, `calloc`, `free`.

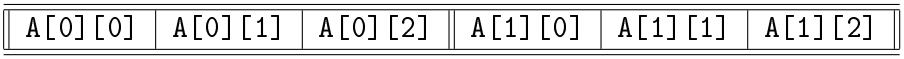
Работа с динамической памятью в C описана в отдельном файле.

6 Массивы указателей

Нельзя путать двумерные массивы и массивы указателей. Массив указателей может выглядеть, например, так:



Двумерный массив может выглядеть так:




```
int A[2][3];   /* двумерный массив -- память выделена автоматически */

int *A[3];     /* массив указателей на int */
int **A;       /* указатель на указатель на int */
```

Несмотря на то, что обращение к элементам в обоих случаях выглядит одинаково (`A[i][j]`), нужно понимать, что в первом случае элементы размещены в памяти сплошным массивом (строка за строкой), тогда как во втором случае `A` — массив указателей на `int`, каждый из которых может указывать (а может и не указывать) на целое число или массив из целых чисел, возможно, различных размеров.

Получить корректно инициализированный массив указателей можно, например, при помощи следующего кода:

```
int **A = calloc(N, sizeof(int *));   /* или malloc(N * sizeof(int *)) */
if (!A)  /* проверяем, выделена ли память */
{
    perror("calloc");
    /* действия при ошибке */
}
for (i = 0; i < N; i++)
{
    A[i] = calloc(M, sizeof(int));    /* или malloc(M * sizeof(int))   */
    if (!A[i])
    {
        perror("calloc");
        /* действия при ошибке */
    }
}
```

Освобождается эта память в обратном порядке:

```
for (i = 0; i < N; i++)
{
    free(A[i]);
}
free(A);
```

Второй параметр функции `main` представляет собой как раз массив указателей, поэтому два заголовка функции `main` по сути эквивалентны:

```
int main(int argc, char *argv[])
{
    ...
}
```

```
int main(int argc, char **argv)
{
    ...
}
```