

Введение в язык программирования С.

Лекция 5

Память.

Сергей Леонидович Бабичев

Память. Массивы.

Массив

- *Массив* — фундаментальная строительная единица в большинстве языков программирования.
- Массив — нечто, что позволяет хранить в себе много пронумерованных элементов одного и того же типа, причём время доступа к любому из элементов одинаково.
- *Простой массив* — набор однотипных элементов, количество которых известно во время компиляции.

```
int ar[100];  
double z[32];
```

- У каждого простого массива есть свойства:
 - ▶ имя (ar или z);
 - ▶ тип его элементов (элементы ar имеют тип int, элементы z имеют тип double);
 - ▶ и количество элементов (100 элементов в массиве ar, 32 элемента в массиве z).

Массив

- Количество элементов простого массива должно быть известно в момент компиляции программы то есть размер простого массива должен быть *константным выражением*
- После создания массива к его элементам можно обращаться по *индексу*.

```
ar[10] = 15;
```

```
z[0] = 3.1415926;
```

Массив, объявленный `int ar[100]`, содержит ровно 100 элементов, которые нумеруются от 0 до 99. Обращение по индексам вне этих границ приводит к непредсказуемым результатам (*UB, Undefined Behavior*).

Операции над массивами

- Выражение вида `ar[i]` является l-value, оно может использоваться точно в тех же местах программы, где могла бы использоваться обычная переменная.

```
int ar[100], i;
...
i = 0;
if (ar[i] < ar[i+1]) {
    int t = ar[i];
    ar[i] = ar[i+1];
    ar[i+1] = ar[i];
}
```

Обменяли местами элементы `a[i]` и `a[i+1]`, если они расположены в порядке возрастания.

В Си нет встроенных операций над массивами, как над целыми объектами. Для работы с массивами нужно явным образом пройти по всем его элементам.

Обработка массивов

- Чаще всего используют цикл for

```
for (int i = 0; i < 100; i++) {  
    ar[i] = 0;  
}
```

- В математической нотации, множество индексов есть $[0..100)$ или множество закрытое слева и открытое справа.

Если есть два одинаковых по размерам массива из однотипных элементов.

```
int ar1[100], ar2[100];  
...
```

то скопировать один массив в другой как `ar2 = ar1`; не получится.

Надо так:

```
for (int i = 0; i < 100; i++) {  
    ar2[i] = ar1[i];  
}
```

Автоматическая память

- Переменные, объявленные где-то внутри функции называются *автоматическими* и место для них выделяется в *стеке*.
- Переменные, объявленные в любом *блоке* {...} автоматически исчезают,

Объявление в блоке переменной с именем, которое уже существует в *охватывающем* блоке, разрешается и такое объявление *скрывает* внешнее имя. Иногда это полезно, но будьте с этим осторожны.

```
int f(int n) {  
    double x = 123;  
    if (n > 0) {  
        int x = 5; // We hide double x  
        // int x has own memory  
    }  
}
```

Глобальная и статическая память

- Под одну и ту же автоматическую переменную могут выделяться разные места в стеке в разное время.
- Если функция вызывает сама себя, *рекурсивная*, то в одно время может существовать несколько копий переменной.
- Существуют переменные, которые сохраняют своё место всё время исполнения программы.
- Это — *глобальные переменные* и *статические переменные*

Глобальная и статическая память

```
int gv, c[100000]; // Глобальные переменная и массив
int func() {
    gv = 0;
    for (int i = 0; i < 100000; i++)
        gv += c[i];
}
int bar() {
    static int sv; // Статическая переменная
    if (sv == 0) {
        printf("Вызвали функцию bar\n");
        sv = 1;
    }
}
int main() {
    func();
    printf("gv=%d\n", gv);
    for (int i = 0; i < 100; i++)
        bar();
}
```

Глобальная и статическая память

Статические и глобальные переменные не меняют своего адреса во время исполнения программы и инициализируются нулями (если они не инициализированы иначе).

- Глобальный массив с [100000] инициализирован нулями, поэтому сумма его элементов тоже нулевая.
- Сообщение о том, что вызвана функция `bar` было выведено на экран только один раз, так как значение переменной `sv` внутри функции `bar` сохраняется между вызовами функции и, однажды став единицей, больше не изменится.
- Переменные `gv` и `c` *видны* во всех местах программы после их объявления, а переменную `sv` видно только внутри функции `bar`.
- Имя `sv` как статической переменной функции `bar` вне функции использовать нельзя, мы *не видим* эту переменную, несмотря на то, что она существует всё время исполнения программы.

- Истинно глобальные переменные можно использовать и в других *единицах компиляции*.

- Для этого достаточно *объявить* их, как *внешние*, добавив ключевое слово `extern`:

```
// Файл main.c
int gv, c[100000];
...
// Файл user.c
extern int gv, c[100000];
```

- Без ключевого слова `extern`, компиляция (точнее, сборка) завершится неудачей: каждое определение глобальной переменной добавляет имя в *глобальное пространство имён*.
- В это же глобальное пространство имён добавляются и имена функций (если их не объявить `static`).

Постарайтесь ограничить применение глобальных переменных. Не загрязняйте пространство имён.

Указатели

- Указатели — самое интересное и самое необычное в Си.
- Указатели существовали в нескольких языках до Си, но только в Си они получили чрезвычайно большую мощность.
- В 1970-х годах, практически одновременно появились два небольших языка программирования, претендующих на универсальность, Pascal (на 3 года раньше) и Си.
- Первый Си был гораздо несовершеннее нынешнего, но именно на Си стали разрабатывать и системные программы, и расчётные программы, и прикладные программы.
- Pascal пережил всплеск, связанный с почившей ныне фирмой Borland, но во все времена количество *промышленного* кода, написанного на Си во много раз превосходило таковое на любом из диалектов Pascal.
- Если бы при проектировании языка Си не изобрели бы указатели с их арифметикой, то Си не выжил бы в конкурентной борьбе с другими языками.

Указатель — переменная, содержащая в себе адрес какой-то области памяти.

Указатели

- Унарная операция `&` — операция *взятия адреса*.

```
01 int v = 5;
```

```
02 int *pv;
```

```
03 pv = &v;
```

```
04 *pv = 7;
```

- Знак звёздочки `*` перед переменной при её объявлении показывает нам, что переменная будет указателем на тот тип, что находится слева от звёздочки.
- Вторую строчку следует читать так: объявляем переменную `pv`, которая является указателем (`*`) на целое (`int`).
- Третью — так: присвоить переменной `pv` адрес переменной `v`;
- Четвёртую: по адресу, который находится в переменной `pv` положить число 7.

Работа с указателями

Что было после первых двух строк:

v 7

pv ??? \longrightarrow ? *pv = крах

Что произошло в третьей строке:

v 5

pv &v *pv = 5

После четвёртой строки:

v 7

pv &v *pv = 7

Использование указателей

- Операция взятия адреса & к тому, что может иметь адрес, например, /-значениям: например, к переменным, элементам массивов и структур.
- Операция не применима к литералам (кроме строк), и выражениям типа `10 * 20`.
- Указателям можно (и нужно) присваивать какие-то адреса, причём адреса не первые попавшиеся, а адреса кусков в памяти, содержащих значения нужного типа.

```
int v = 5;
double *pv;
pv = &v; // This is bad practice!
```

- Хороший стиль программирования: любое объявление переменной стоит совмещать с инициализацией. Особенно это относится к указателям.

При объявлении указателей всегда инициализируйте их либо корректным адресом, либо специальным значением `NULL`, гарантирующим, что по этому адресу ничего нет.

Указатели и функции

- Пока указатели похожи на игрушку.
- Напишем функцию `swap`, которая меняет местами значения двух переменных.

```
#include <stdio.h>
void swap(int x, int y) {
    x ^= y; y ^= x; x ^= y;
    printf("swap: x=%d y=%d\n", x, y);
}
int main() {
    int x = 5, y = 7;
    swap(x, y);
    printf("x=%d y=%d\n", x, y);
}
```

Что выведет эта функция?

Указатели и функции

- Пока указатели похожи на игрушку.
- Напишем функцию `swap`, которая меняет местами значения двух переменных.

```
#include <stdio.h>
void swap(int x, int y) {
    x ^= y; y ^= x; x ^= y;
    printf("swap: x=%d y=%d\n", x, y);
}
int main() {
    int x = 5, y = 7;
    swap(x, y);
    printf("x=%d y=%d\n", x, y);
}
```

Что выведет эта функция?

```
swap: x = 7 y = 5
x = 5 y = 7
```

Указатели и функции: распечатаем значения указателей

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

```
void swap(int x, int y) {  
    x ^= y; y ^= x; x ^= y;  
    printf("swap: &x=%p x=%d &y=%p y=%d\n", &x, x, &y, y);  
}
```

```
int main() {  
    int x = 5, y = 7;  
    swap(x, y);  
    printf("main: &x=%p x=%d &y=%p y=%d\n", &x, x, &y, y);  
    return 0;  
}
```

Запуск программы:

```
swap: &x=0x7ffeef406afc x=7 &y=0x7ffeef406af8 y=5
```

```
main: &x=0x7ffeef406b18 x=5 &y=0x7ffeef406b14 y=7
```

Использование указателей

Чтобы разрешить изменять функцией `swap` `x` и `y`, мы должны это явным образом указать: передать в `swap` не сами переменные, а их адреса. Только тогда функция `swap` будет что-то в состоянии сделать с переменными, находящимися в `main`.

```
#include <stdio.h>

void swap(int *x, int *y) {
    *x ^= *y; *y ^= *x; *x ^= *y;
    printf("swap: &x=%p x=%d &y=%p y=%d\n", x, *x, y, *y);
}

int main() {
    int x = 5, y = 7;
    swap(&x, &y);
    printf("main: &x=%p x=%d &y=%p y=%d\n", &x, x, &y, y);
    return 0;
}
```

Запуск программы:

```
swap: &x=0x7ffeec169b18 x=7 &y=0x7ffeec169b14 y=5
main: &x=0x7ffeec169b18 x=7 &y=0x7ffeec169b14 y=5
```

Указатели и функции

- Всё заработало.
- Функция `swap` стала менее красивой, нам приходится каждый раз использовать операцию *взятие значения по адресу*, которая обозначается `*`. Зато мы решили поставленную задачу.

Единственным способом изменить значение локальной переменной при передаче её в функцию — передать её адрес и принять его как указатель в вызванной функции.

- Си — один из немногих языков, который позволяет при вызове функции легко определить, может ли изменить функция свои аргументы.
- По вызову функции `swap` в `main` мы видим знаки `&` перед переменными и понимаем, что значения этих переменных после вызова функции *могут* поменяться.

Структуры

Структуры

- Массивы — полезно, но все элементы должны иметь один и тот же тип.
- Если мы хотим создать модель автомобиля, то описать его одним, пусть даже большим массивом, содержащим элементы одного типа, не удастся.

Структура — тип данных, позволяющий сгруппировать объекты, возможно, разных типов и работать с ними как с единым целым. Сама структура становится *объектом*, а её составные части становятся *подобъектами* или *полями*.

Структуры: анатомия

- Каждое поле имеет *тип* и *имя*.
- Для модели автомобиля структура, например, может состоять из полей `num_of_wheels` — количестве колёс целого типа, `velocity` — текущей скорости автомобиля вещественного типа и `reg_number` типа *массив СИМВОЛОВ*.
- Такую структуру можно было бы описать так:

```
struct car_s {  
    int num_of_wheels;  
    double velocity;  
    char reg_number[12];  
};
```

У нас появился новый тип данных — `struct car_s`. Теперь мы можем создавать переменные, которые имеют соответствующий тип:

```
struct car_s a,b,c;
```


Формат описания структуры

- За ключевым словом `struct` следует идентификатор `car_s`, который называется *тегом* или *ярлыком* структуры.
- Если не хочется писать `struct car_s` каждый раз, можно использовать ключевое слово `typedef`.
- Появился ещё один тип данных — `car`, который можно использовать везде, где допустимо имя типа.

```
typedef struct car_s car;
```

```
car a, road[1000], *pa = &a;
```

Мы объявили одиночный автомобиль `a`, массив из 1000 автомобилей `road` и указатель на автомобиль `pa`, который инициализировали адресом автомобиля `a`.

Операции над структурами

- Структура — полноценный тип данных и над ней можно производить универсальные для всех типов данных операции: копировать, передавать в функции, определять адрес, возвращать значение данного типа из функции.
- Вполне корректна запись:

```
car some_function() {  
    car ret;  
    ...  
    return ret;  
}
```

```
...  
car t = some_function();  
cat copy_t = t;
```

Операции над структурами

- Для обращения к полю структуры применяется соответствующая операция *точка*.

```
car ret;  
ret.number_of_wheels = 4;  
ret.velocity = 10.7;  
strcpy(ret.reg_number, "A789AB150");
```

- После применения этой операции (она ещё носит названия *квалификация поля*) поле структуры становится неотличимым от обычной переменной.
- Оно может быть и l-значением, и r-значением.
- Если оно имеет элементарный тип, то разрешены операции типа += или ++.
- С целой структурой такие операции уже не существуют.
- Мы не можем написать новый тип, похожий на встроенные.
- Это ограничение, которое в Си никак не обходится. Если нужно, чтобы новый тип был «как родной» в языке, придётся использовать C++.

Указатели и структуры

- Структуры — тоже объекты языка, имеют адреса, к ним тоже применимо взятие адреса и указатели на структуры.
`car a, *ra = &a;`
- Объявив одиночный автомобиль `car a`, мы выделили память под хранение всех полей структуры и эта память обязана располагаться рядом и в заданной описанием последовательности. Указатель на автомобиль `ra` мы инициализировали адресом автомобиля `a`.
- Будет ли правильным, если мы напишем `*ra.velocity = 123.4;`?
- Нет. Приоритет операции квалификации поля `.` выше, чем операции взятия значения по адресу `*`, поэтому порядок выполнения операций будет следующим: `*(ra.velocity) = 123.4;`
- Можно писать так: `(*ra).velocity = 123.4;`, это верно, но громоздко?

Для обращения к элементу структуры по указателю на структуру применяется операция `->`, например `ra->velocity = 123.4;`

Операция sizeof

- Для любого элемента данных и любого типа данных имеется операция sizeof в двух вариантах.
- Во-первых, операндом sizeof может быть любое выражение.
- Во-вторых, операндом может быть любое имя типа, заключённое в круглые скобки.
- sizeof возвращает количество минимально адресуемых единиц памяти, *байтов*, требуемых для размещения в памяти операнда или объектов такого типа.
- Известно, что sizeof(char)=1. Остальное зависит от архитектуры компьютера.

```
int sizeint = sizeof(int); // Скорее всего 4
double d = 1.0;
int sized = sizeof d; // Скорее всего 8
int arr[100];
int sizearr = sizeof arr; // Скорее всего 400
```

Выравнивание элементов структур

- Сколько байт будет выделено для хранения следующей структуры?

```
struct some_1 {  
    double x;  
    int y;  
    short z;  
    char c;  
};
```

- А для этой:

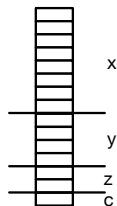
```
struct some_2 {  
    char c;  
    double x;  
    short z;  
    int y;  
};
```

Выравнивание элементов структур

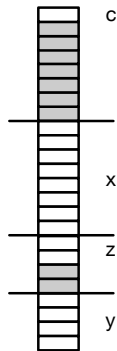
- Эти структуры содержат одни и те же элементы, поэтому они потребуют одинаковое количество памяти?
- Нет.
- Скорее всего, первая структура займёт 16 байт, а вторая — 24 байта.
- Проблема в том, что архитектура современных компьютеров такова, что для быстрого доступа к элементам данных требуется *выравнивание* этих элементов на определённые адреса.
- Если поместить объект типа `double` по адресу, не кратному 8, то все операции с ним потребуют больше *процессорных тактов*.
- Часто *невыворненные* данные вообще запрещены.
- 32-битные объекты нужно размещать по адресам, кратным четырём.
- 64-битные объекты нужно размещать по адресам, кратным восьми.

Выравнивание элементов структур

```
struct some_1 {  
    double x;  
    int y;  
    short z;  
    char c;  
};
```



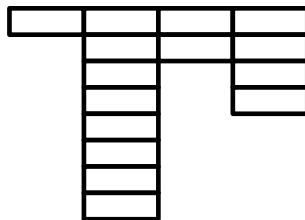
```
struct some_2 {  
    char c;  
    double x;  
    short z;  
    int y;  
};
```



Объединения

- Иногда нам нужно очень жёстко экономить память.
- Для этого мы можем использовать другую модификацию структур — *объединение*, `union`.

```
union some_3 {  
    char c;    // 0  
    double x; // 0  
    short z;   // 0  
    int y;     // 0  
} us3;
```



c x z y

- Присваивание любому элементу объединения меняет все остальные!

```
us3.c = 5;
```

Чему станет равно x? Чему-то непредсказуемому.

Указатели и массивы

- Настало время подробнее понять, что происходит, когда мы пишем, например, `int a[100];`.
- Мы называли такие массивы *простыми*.
- Под него выделяется память на момент его создания.
- Оказывается, имя массива (в данном случае `a`) не что иное, как указатель на выделенную системой память.
- Так `a` — массив или указатель?
- Если `a` — указатель, то как с ним работать?
- Что у нас имеется кроме операции `*` для обращения к той памяти, на которую указатель показывает?
- В обращении к элементам массива нам помогает *адресная арифметика*.

Адресная арифметика

- Пусть начальным адресом нашего массива a будет 10000.
- Тогда обращение $*a$ разрешено и это будет нулевым элементом массива.
- К нулевому элементу массива можно обратиться и как $a[0]$.
- Давайте запишем первое выражение немного по-другому: $*(a+0)$. Тогда $a[0]$ есть синоним к $*(a+0)$.
- Память под массивы гарантированно выделяется *непрерывным куском*, поэтому элемент под номером 1, $a[1]$ будет находиться по адресу 10004, если размер одного элемента типа `int` равен четырём.
- $a[1]$ есть синоним к $*(a+1)$, $a[i]$ — для $*(a+i)$.
- Арифметика становится не совсем обычной: к указателю, равному 10000 прибавили единицу и он стал равен 10004.
- Для всех ли типов данных подобное верно?
- Для всех с одной поправкой: степень увеличения адреса зависит от размера элемента массива.

Указатели допускают следующие операции:

- сложение указателя с целым $ptr + i$. Результат показывает на элемент того же типа, отстоящий на i элементов дальше ptr ;
- вычитание из указателя целого $ptr - i$. Результат показывает на элемент того же типа, находящийся за i элементов перед ptr ;
- вычитание указателей одного типа $ptr2 - ptr1$. Результат — количество элементов этого типа между адресами.

Массивы и функции

- Можно ли передавать массивы в функции?
- Да, конечно!
- Для совсем наивных можно описывать аргументы функций как массивы:

```
int sum_elems(int array[100]) {  
    int i, sum = 0;  
    for (i = 0; i < 100; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

Массивы и функции

- Почему «для наивных»?
- Потому, что в Си массив как целое передать в функцию невозможно.
- Имя массива — просто указатель на нечто, который нельзя изменить.
- Пока мы в области определения простого массива, видимости точки его создания, компилятор знает его размер.
- После того, как массив, точнее, указатель на него, переданы в функцию, никто внутри функции не способен определить его размер.

```
#include <stdio.h>

void bar(int b[100]) {
    int sizebar = sizeof b / sizeof b[0];
    printf("sizebar = %d\n", sizebar);
}

void func() {
    int arr[100];
    int sizearr = sizeof arr / sizeof arr[0];
    printf("sizearr = %d\n", sizearr);
    bar(arr);
}

int main() {
    func();
}

sizearr = 100
sizebar = 2
```

Массивы и функции

- В функцию передан указатель на массив, размер указателя здесь 8 байт, размер типа `int` здесь четырем байтам, отсюда `sizeof = 2`.
- Обман? Зачем всё это?
- Чтобы не отпугнуть тех, кто хочет мигрировать с FORTRAN на Си.
- Функция не может отличить указатель на отдельную переменную от указателя на кусок памяти (массив).
- Число в квадратных скобках, которое, вроде бы, должно показать нам количество элементов массива, для одномерных массивов игнорируется.
- Следующие три объявления функции эквивалентны:

```
int add_elems(int arr[10000]);  
int add_elems(int arr[]);  
int add_elems(int *arr);
```


Массивы и функции

- Теперь у нас есть универсальная функция, которая готова вычислить сумму элементов любого целочисленного массива любой длины.
- Нужно только, чтобы она знала количество элементов.
- В классическом Pascal такую функцию написать было невозможно.
- Математикам такие функции позарез необходимы, для линейной алгебры, для векторов и матриц.

Массивы и функции

- Наша функция `add_elems` пока не знает количество элементов массива для суммирования.
- Нужно добавить в неё ещё один аргумент.
- Обычно в Си его добавляют после имени массива.
- А какой у этого аргумента тип?
- Это тот тип, который может содержать любые размеры для массивов.
- Это не `int`, появилось бы ограничение на размеры массивов.
- В современных компьютерах памяти много и массив может содержать триллионы элементов.
- Для размеров массивов и индексов лучше применять тип `size_t`.
- Он описан в файле `<stddef.h>`, или `<stdlib.h>` или `<stdio.h>`
- Рядом имеется тип `ptrdiff_t`, который может иметь знак, в отличие от `size_t`.

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

void add_elems(int *b, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum += b[i];
    }
    return sum;
}

int main() {
    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6};
    int sum_arr = add_elems(arr, sizeof arr / sizeof arr[0]);
}
```

Инициализация простых массивов

Простой массив можно объявить с его инициализацией. Его размер будет известен при компиляции по количеству инициализаторов.

```
int x[] = {1,2,3,4,5,6,7,8,9,10}; // OK
int y[10] = {1,2,3,4,5,6,7,8,9,10}; // OK
int z[20] = {1}; // OK, z[1..19] = 0
int t[5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Error.
```