Указатели

Как мы уже знаем, память состоит из ячеек размером 1 байт, каждая из которых хранит 8-битовое число в двоичном представлении (от 0 до 255 включительно). Каждый байт имеет адрес, а адрес это число. Напомним, что данные типа `int` занимают 4 байта[1].

Пока что мы оперировали только с данными, которые были "просто числами". Но мы можем делать с адресами почти всё то же самое, например, создавать переменные, которые будут хранить адреса других переменных.

Создадим переменную `px`, которая хранит адрес переменной `x`:

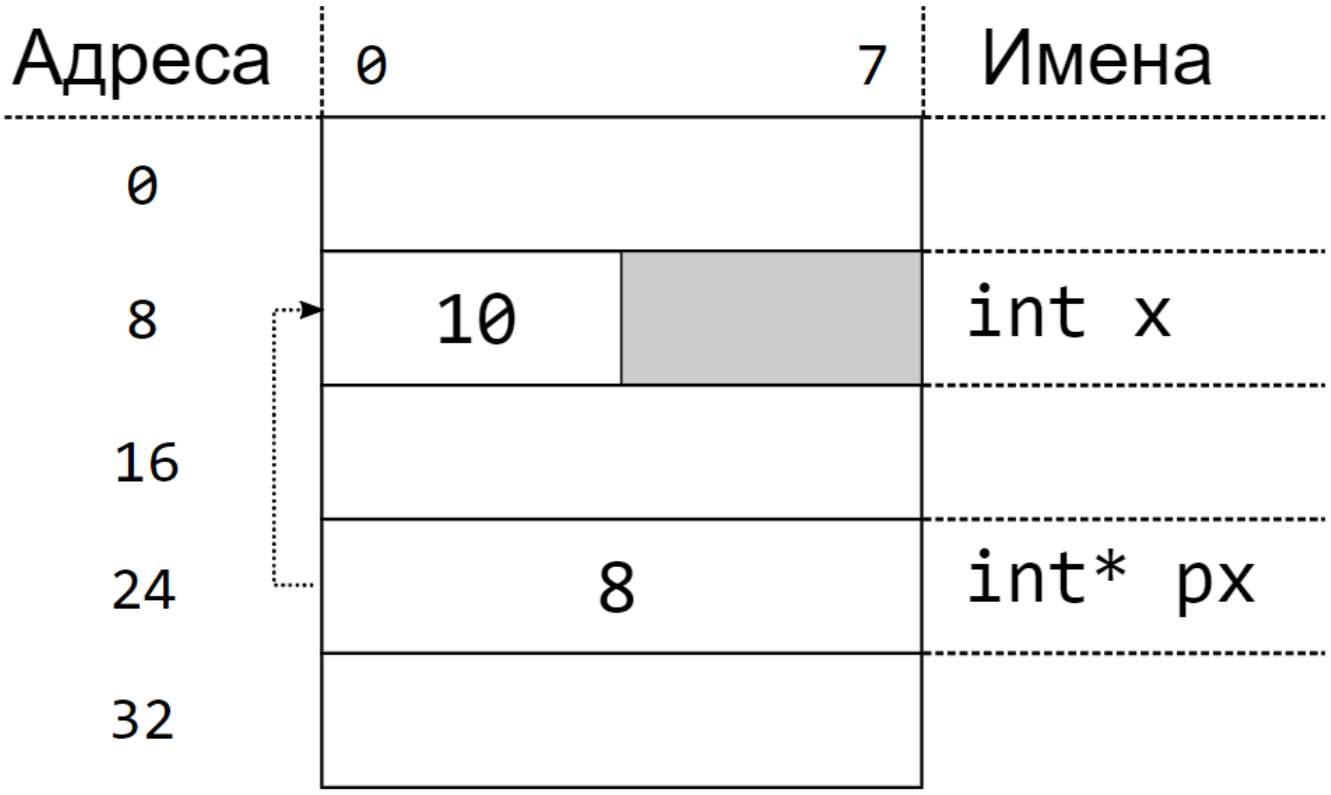
```
int   x = 10; // переменная типа int

int* px = &x; // переменная типа int*.

              // С помощью оператора & мы берём адрес переменной x.
```

Если тип выражения обозначить `T`, то тип указателя на данные такого типа записывается `T*`. Можно добавлять сколько угодно *уровней косвенности*: указатель на `int` это `int*`, а указатель на указатель на `int` это `int**`, и так далее. Скоро узнаем, зачем это нужно.

Чтобы легко понять суть указателей, достаточно представить память компьютера. Для примера предположим, что:

- Программа выполняется на 64-разрядной архитектуре, где каждый адрес занимает 8 байт;
- Переменная `x` оказалась в памяти по адресу 8 и она занимает 4 байта;
- Переменная `px` оказалась в памяти по адресу 24 и она занимает 8 байт (любой указатель на такой архитектуре занимает 8 байт).

Тогда память можно представить такой картинкой; пунктирная стрелка показывает, на какой адрес ссылается восьмибайтовое число 8 по адресу 24. По адресу 8 начинается число 10, которое занимает адреса 8, 9, 10 и 11.



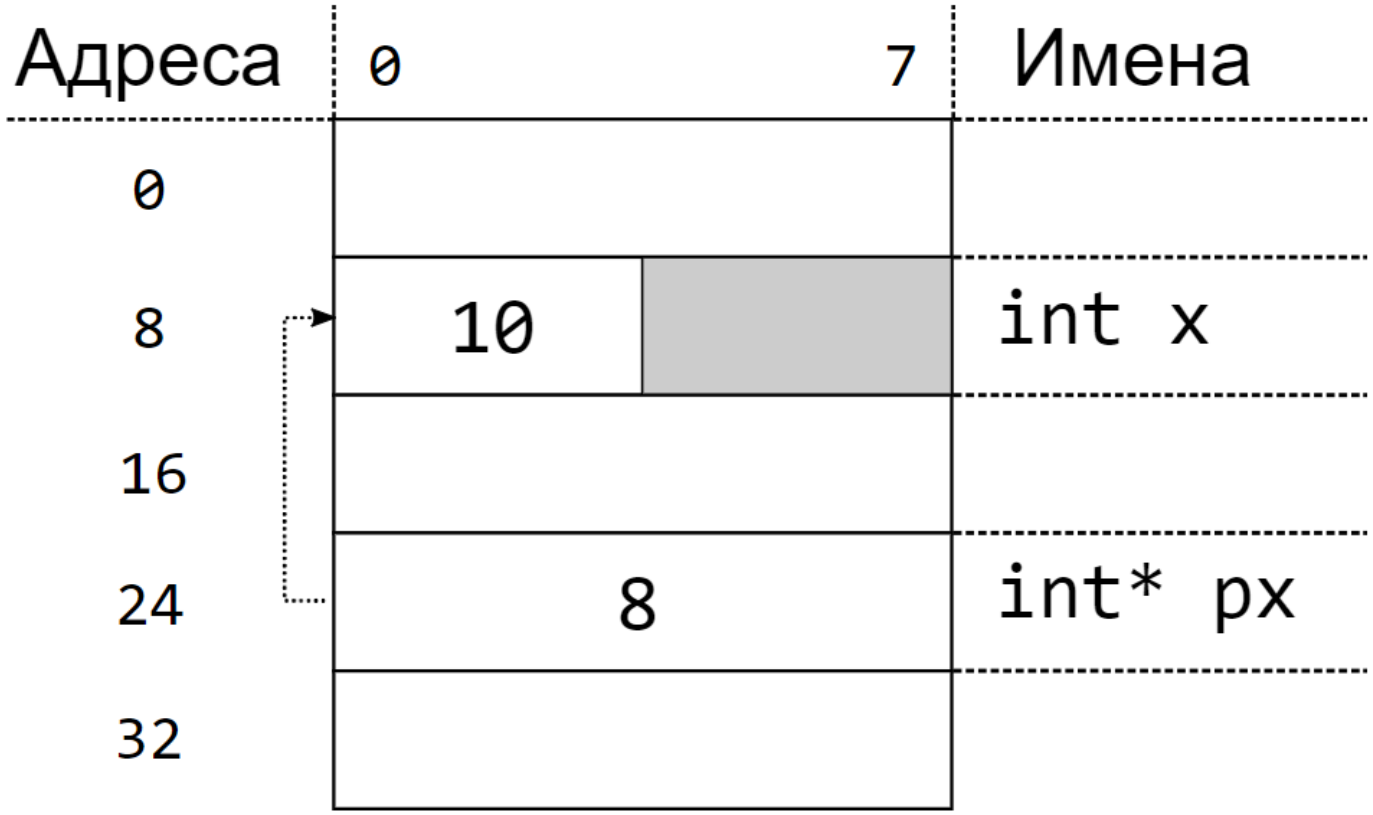
Гуляем по памяти

Имея адрес мы можем *пройти (dereference, разыменовывание) по этому адресу* с помощью оператора `*`. Это значит, что вместо адреса данных мы будем оперировать с самими данными. Например:

```
int   x = 10;

int* px = &x;




print_int( *px ); // выведет 10

                  // px хранит адрес x, поэтому *px соответствует x


*px = 42;         // px хранит адрес x, поэтому *px соответствует x; в x мы
запишем 42
```

```
print_int( *px ); // выведет 42

print_int( x );   // выведет 42
```

На схемах памяти разыменовывание означает проход по стрелке:



Тип указателя

Введём второй тип численных данных `char`. Несмотря на запутывающее название, воспринимать этот тип следует не как "букву" или "символ", а как число размером 1 байт. Мы пока знаем только типы `int` и `char`.

Данные разных типов занимают разное количество байт в памяти, но все указатели занимают одинаковое количество байт: скажем, адрес `char` не отличается по размеру от адреса `int`. Более того, во время выполнения программы, и те и другие будут просто числами, и нигде в памяти не будет храниться информация, позволяющая отличить один указатель от другого (и указатели от данных других типов).

Однако в исходном коде программы мы различаем типы указателей: `char*` это **не то же самое**, что и `int*`. Это необходимо чтобы читать данные из памяти по указателям и записывать их. Например, если мы хотим "прочитать данные по указателю на `int`", потребуется прочитать 4 байта (а не один, два или 16) начиная с адреса, который хранится в указателе. Если тип указателя известен, то и размер данных, на которые он указывает, тоже известен, поэтому ясно, сколько байт в памяти затронет чтение или запись по нему.

Применения указателей

У указателей масса применений, например:

- Изменение переменной, которую мы создали вне функции (локальной переменной вызывающей функции)
- Создание сложных структур данных с внутренними связями, навигация по ним (см. один из следующих уроков).
- Вызов функции по её адресу; в зависимости от того, куда направляет нас указатель, будет вызвана та или иная функция (см. один из следующих уроков).

Мы будем постепенно знакомить вас с применениями указателей.

Связь с вызывающей функцией

Мы уже умеем передавать числа в функции. При передаче численного значения оно копируется, и функция никак не может изменить значение в том месте, откуда оно пришло:

```
void f( int x ) {

   x = x + 1;          // мы изменяем x, который ведёт себя как локальная
переменная для f

   printf("%d", x);

}



void main() {

   int a = 10;

   f(a);             // напечатает 11, значение переменной `a` осталось 10

   f(a);             // напечатает 11, значение переменной `a` осталось 10

   f(a);             // напечатает 11, значение переменной `a` осталось 10



   return 0;

}
```

Указатели дают нам возможность передать адрес каких-то ячеек памяти и изменить их внутри самой функции, через адреса. Разумеется, эти изменения будут видны во всей программе.

```
void inc( int* px ) {
```

```
    *px = *px + 1;        // *px позволяет обратиться к данным, на которые
указывает px

}


void main() {


    int a = 10;

    inc( &a );           // a = 11

    inc( &a );           // a = 12

    inc( &a );           // a = 13


    printf("%d", a); // выведет 13


    return 0;

}
```

Когда мы передаём значение в функцию как обычно, напрямую и без указателя, говорят, что переменная передаётся **по значению.** Когда же мы передаём указатель на переменную, мы говорим, что переменная передаётся **по указателю,** или **по ссылке**.

При передаче по значению функция работает с копией переменной и не может изменить саму переменную. При передаче переменной по указателю мы можем пройти по нему и изменить саму переменную.

Напишите функцию `swap`, которая принимает два указателя на переменные типа `int` и меняет значения этих переменных местами.

Sample Input:

```
1 2
```

Sample Output:

```
2 1
```

1task – program

Напишите функцию `normalize`, которая принимает указатель на число и делит это число на 2 пока оно чётное и положительное.

Например, число 100 станет 25, а число 5 останется собой.

Гарантируется, что эта функция не будет запускаться на отрицательных числах.

Sample Input:

```
100
```

Sample Output:

```
25
```

2task – program

Применение указателей для "возврата" нескольких значений в вызывающую функцию

Чтобы прочитать число с клавиатуры в C используется функция `scanf`. Она, как и `printf`, принимает сначала строку символов, где указаны спецификаторы того, что нужно прочитать; затем она принимает **адреса** тех переменных, которые она должна заполнить.
Возвращает `scanf` количество прочитанных элементов.

Например, такой вызов `scanf` прочитает два числа, разделённые пробелом[1], и запишет эти числа в переменные `x` и `y`.

```
int x;

int y;

int count = scanf("%d %d", &x, &y);


// если мы ввели "hello", то count == 0

// если мы ввели "42 hello", то count == 1, x = 42, y может быть любым!

// если мы ввели "42 99", то count == 2, x = 42, y == 99
```

Функция `scanf` это одна из таких функций, которым нужно вернуть несколько кусочков данных, а не один. Но вернуть из функции мы можем только одно значение! Многие программисты достигают нужного эффекта с помощью указателей:

- `scanf` всегда возвращает число -- количество успешно прочитанных элементов. Если ничего не получилось прочитать, вернётся ноль.

- В аргументах `scanf` принимает указатели на то, куда записать считанные данные. Они как бы тоже "возвращаются" из функции.

Теперь мы понимаем, как сделать функцию `read_int`, читающую число с клавиатуры и "красиво" возвращающее его. К сожалению, она забывает результат работы `scanf`, и если прочитать ничего не получилось, эта функция просто вернёт 0, как если бы на вход подали число 0. В реальных приложениях мы всегда хотим проверять результаты взаимодействия с пользователем: получилось или нет?

```
int read_int() {
    int res = 0;
    scanf( "%d", &res );
    return res;
}
```

[1] Более того, если в форматной строке для `scanf` встречается один пробел, `scanf` пропустит любое количество последовательно идущих пробелов.

Задача факторизации (разложения числа на множители) имеет важное применение в криптографии. Сейчас не существует эффективного алгоритма, который бы разложил большое число порядка 10^{2000} на множители за разумное время. В криптосистеме RSA чтобы взломать шифр злоумышленнику необходимо разложить большое число на два множителя. При этом известно, что данное число является произведением двух простых чисел.

Числа, которые умещаются в диапазон представления типа `int`, гораздо меньше, почти всегда меньше 10^{55}, поэтому мы можем и располагая достаточно небольшими вычислительными ресурсами наивно разложить число на произведение двух других чисел.

Заполните тело функции, которая принимает число n и с помощью указателей заполняет два числа a и b так, что:

- Если $n=1$ или n простое, то положим $a = 1$, $b = n$.
- Если n составное, то:

$\begin{cases} ab=n\\ 1 < a \le b \end{cases}$ $ab=n 1<a\le b$ и a наименьший делитель n, больший единицы.

- Гарантируется, что $n>0$.

Если ваше решение делает слишком много лишних действий, то оно не пройдёт проверку по времени в седьмом тесте. Но это не алгоритмическая задача, никаких особых оптимизаций тут делать не нужно. Для решения достаточно проверить делители в цикле от 2 до $\frac{n}{2}$ $2n$.

Sample Input:

```
20
```

Sample Output:

```
2 10
```

3task – program

Неинициализированные указатели

Ячейки памяти всегда хранят какие-то значения, поэтому если создать переменную-указатель и ничего в неё не записать, она всё равно будет уже равняться какому-то числу (адресу). При этом неизвестно, указывает ли такой произвольный адрес вроде `10231234` хоть на какие-то определённые данные, или это "мусорное" значение. Чтобы отличать некорректные указатели мы используем специальное значение "неинициализированного" указателя `NULL`.

```
int* x = NULL; // мы точно знаем, что этот указатель никуда не ведёт
```

```
int* x = 0;     // альтернативная форма записи
```

Использование числа 0 в выражении, от которого ожидается, что оно имеет тип указателя, эквивалентно использованию `NULL`.

Часто `NULL` используется при начальной инициализации указателей; если какая-то функция возвращает указатель на данные, она может вернуть `NULL` чтобы сигнализировать об ошибке:

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

```
...
```

```
int* a = f();
if (a == NULL) {
   /* при выполнении f возникла какая-то ошибка */
}
```

Так всё-таки, что означает указатель на 0? *Есть два контекста, в котором вы можете использовать число 0 в языке C:*

1. *Просто число 0, как в выражениях x * 0 или 0 + 4*
2. *Сравнение указателя с 0 или присваивание 0 в переменную-указатель.*

Во втором контексте 0 не обязательно значит "целое число в котором все биты равны нулю", но всегда значит "специальное значение указателя, которое никуда не ведёт". На некоторых архитектурах это специальное значение может быть, например, -1 (в этом двоичном числе все биты, наоборот, равны 1).

На всех архитектурах код ниже будет работать:

```
int* px = ... ;



// Эквивалентные способы проверить, что указатель инициализирован.

if ( px )

if ( px != 0)

if ( px != NULL)




// Эквивалентные способы проверить, что указатель НЕ инициализирован.

if (!px )

if ( px == 0 )

if ( px == NULL )
```

Оператор sizeof

Чтобы узнать, какой размер занимает тот или иной тип или данные используется оператор `sizeof`. Синтаксически он выглядит как вызов функции, но на самом деле никакой код он не выполняет. Вместо выражения `sizeof(<expr or type>)` компилятор подставляет число.

Скажем, `sizeof(char)` будет заменён на 1 ещё до выполнения программы. Также `sizeof` можно применять к любым выражениям; эти выражения не будут выполняться, компилятор просто выведет тип выражения и подставит его размер. Например, в этом примере функция `printf` не будет вызвана:

```
char print() {

  printf("Hello!");

  return 0;

}
```

```
...

int x = sizeof( print() );  // x = 1, нет вывода на экран.
```

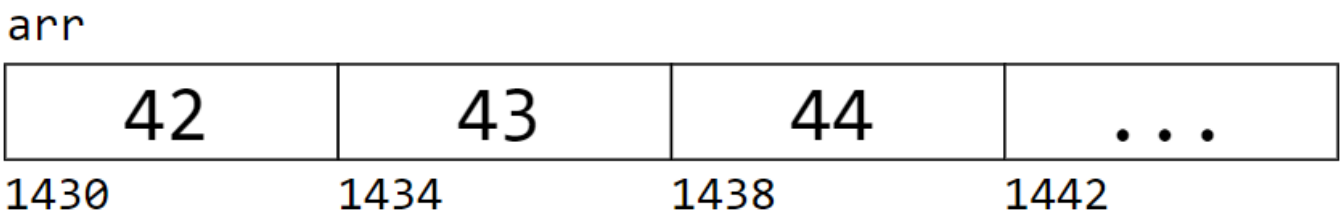
Массивы

Массив это структура данных, в которой хранится фиксированное число однотипных элементов. Например, в массиве могут находиться 10 чисел типа `int`.

В памяти массивы всегда занимают непрерывный участок. Это означает, что данные уложены последовательно, элемент за элементом. Возьмём для примера массив из трёх элементов 42, 43 и 44, который объявляется так:

```
int arr[] = {42, 43, 44};
```

Предположим, что в памяти он начинается с адреса 1430 (а размер `int` равен 4 байта). Тогда элементы располагаются в памяти вот так:



В массиве размера nn каждому элементу соответствует порядковый номер от 0 до n-1$n-1$; этот номер используется чтобы обращаться к элементам массива:

```
printf("%d",   arr[2] );  // выведет 44
```

Начальный элемент массива расположен со смещением 0 относительно адреса начала массива. Следующий элемент расположен со смещением 4 (это размер одного элемента), затем 8, и так далее. Поэтому и индексы в массиве начинаются с нуля: фактически, элемент с индексом ii хранится со смещением (i \times \texttt{размер элемента})(i×размер элемента).

Имя массива можно использовать как адрес его первого элемента:

```
*arr = 999 ; // эквивалентно arr[0] = 999
```

После последнего элемента массива в памяти тоже *что-то лежит*. Если обратиться к десятому элементу массива `arr`, никакой ошибки может не произойти: просто мы считаем из памяти какие-то мусорные значения. В момент выполнения программы про массив не известны ни тип значений в нём, ни его длина (если только мы специально не сохраним где-то информацию о них). Поэтому программист сам следит, чтобы не выйти за границы массива; обычно это означает, что вместе с массивом в функции нужно передавать его длину отдельным аргументом.

Можно также определить массив, указав его размер. В зависимости от того, указаны ли инициализаторы в фигурных скобках, есть варианты:

```
// массив не инициализирован, значения могут быть мусорными

int arr[5];



// оставшиеся члены инициализированы нулями: {1, 2, 0, 0, 0}

int arr[5] = {1, 2};



// частный случай предыдущего: массив заполнен нулями

int arr[5] = {0};



// все нули кроме специально обозначенных элементов: { 0 0 29 0 15 }

int arr[5] = { [2] = 29, [4] = 15 };
```

Мы не можем во время выполнения программы решить, насколько большой массив нам нужен – например, прочитать с клавиатуры число и создать массив такого размера. Для этого используются аллокаторы памяти – мы поймём, как пользоваться стандартным аллокатором, и напишем свой позднее в этом курсе.

Однако мы можем создавать массивы такого размера, который можно посчитать во время компиляции. Например:

```
char array[1024 * 1024];



// помните, sizeof всегда считается во время компиляции

char array[1024 * sizeof( int ) ];
```

Указатели и массивы

С помощью указателей и арифметики можно обращаться к разным частям массивов. Для примера, возьмём указатель `p` типа `int*`. Мы можем прибавить к нему любое целое число n, при этом язык позаботится о том, чтобы фактическое значение указателя изменилось на $n \times \texttt{sizeof(int)}$.

```
int* p = 10;

p = p + 3;     // 10 + 3 * sizeof(int) = 22

p = p - 2      // 22 - 2 * sizeof(int) = 14
```

Результат сложения с числом является корректным указателем пока мы не выходим за пределы массива справа или слева.

Теперь создадим два указателя одного типа `p` и `q`, где-то в середине массива:

```
int array[100];



int* p = array + 40; // адрес 40-го элемента

int* q = &array[60]; // адрес 60-го элемента (альтернативная форма)
```

С двумя указателями мы можем делать следующие операции:

```
if (p <= q) {          // проверить, что p указывает на более левый элемент

   int s = q-p;        // посчитать количество элементов между p и q

   printf("%d\n", s);

}
```

Вычитать можно только меньший указатель из большего, или из равного ему. При этом `p` и `q` обязаны *указывать внутрь одного массива*, иначе результаты не будут иметь смысла.

```
int a;

int b;



int* p = &a;

int* q = &b;
```

```
int s = p-q; // не имеет смысла, т.к.
             // a и b не находятся в одном массиве
             // между ними могут быть ещё какие-то данные
```

Невозможно сложить два указателя, перемножить или поделить их; унарный минус также не имеет смысла.

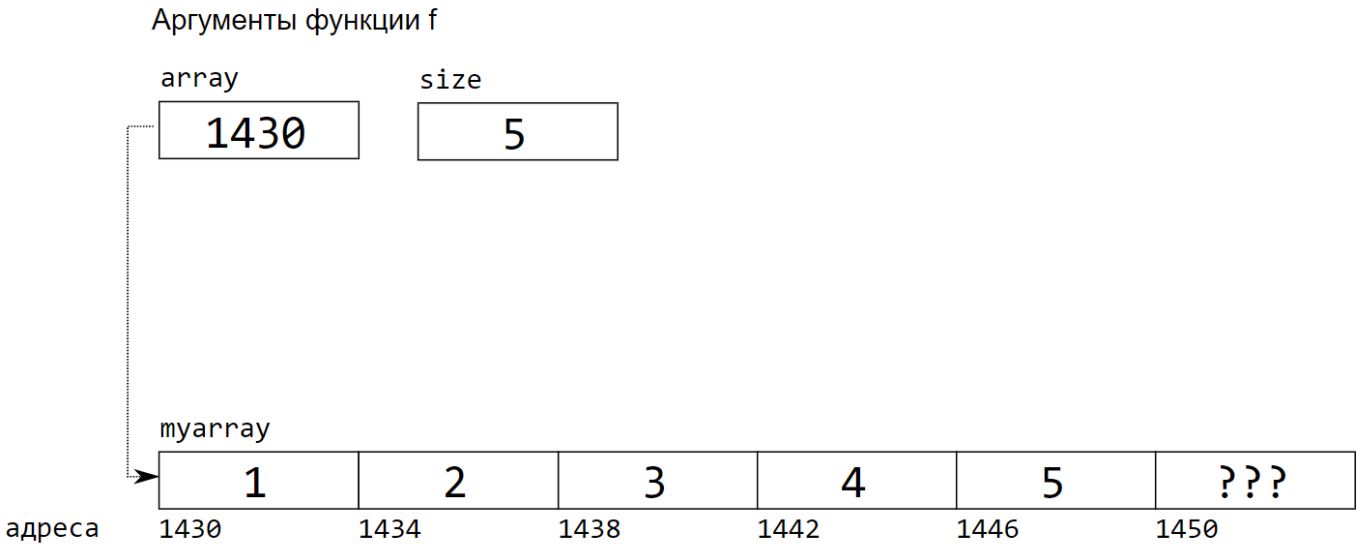
Мы видим, что массивы и указатели очень похожи. Для любого указателя или массива `p`:

- `p[i]` означает то же самое, что `*(p + i)`
- `&p[i]` означает то же самое, что `p + i`

Традиционно, есть два способа передачи массивов в функции:

1. Передать указатель на первый элемент и размер массива

```
2. void print_int(int arg) { printf("%d\n", arg); }
3.
4. // array -- указатель на первый элемент, size -- количество элементов
5. int f(int* array, int size) {
6.    for (int i = 0; i < size; i = i + 1) {
7.     print_int(array[i]);
8.    }
9. }
10.
11. ...
12.
13. int myarray[] = { 1, 2, 3, 4, 5 };
14. int myarray_size = 5;
    f( myarray, myarray_size );
```



2. Передать указатель на начало массива и на элемент *сразу после* последнего.
В этом случае можно пройтись по массиву увеличивая указатель так, что он сначала указывает на первый элемент, затем на второй и так далее. Когда мы дошли до указателя за последним элементом, можно остановиться.
Почему передается не указатель на последний элемент, а сразу за ним? Это позволяет передавать массивы нулевой длины: для них указатель на начало массива будет равен указателю и элемент "сразу за последним".

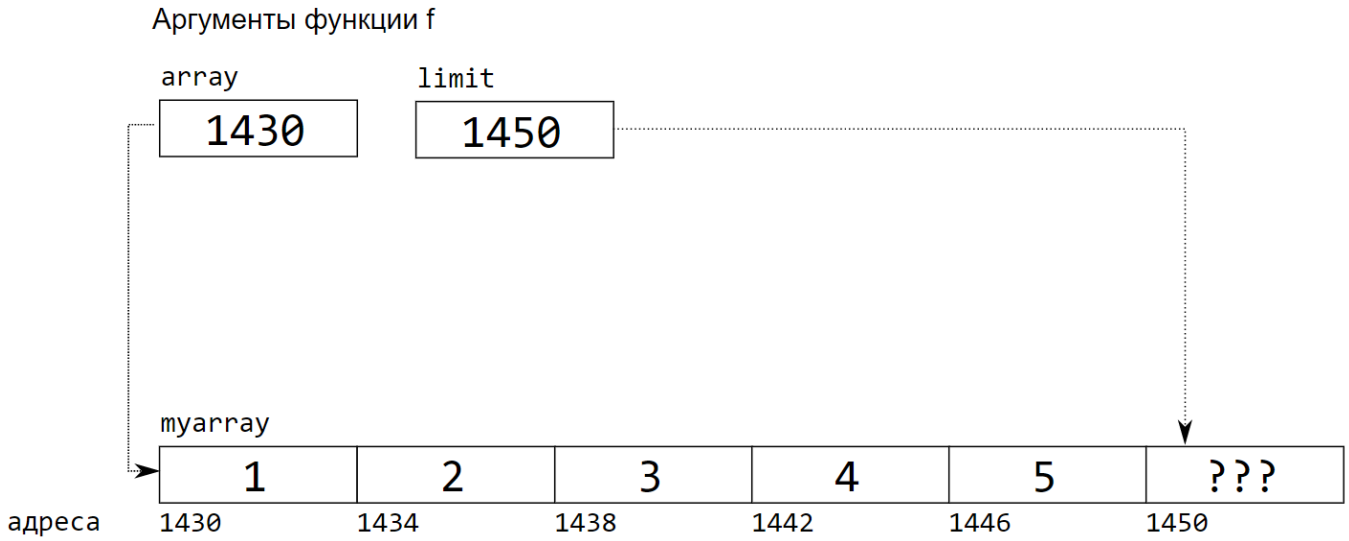
```
void print_int(int arg) { printf("%d\n", arg); }
```

```
// array -- указатель на первый элемент, limit -- указатель на элемент "сразу за последним"

int f(int* array, int* limit) {

   for (int* current = array; current < limit; current = current + 1) {

      print_int( *current );

   }

}
```

```
...
```

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

```
f( myarray, myarray + 5);
```



Напишите две функции, которые переворачивают массив, принимая его первым и вторым способом. В перевернутом массиве элементы идут в обратном порядке, например, массив `10 24 3 4 5` станет `5 4 3 24 10`. Изменить нужно тот массив, который функция получает в качестве аргумента; его не нужно выводить.

4task – program

Напишите функцию `array_fib`, которая заполнит массив числами Фибоначчи по порядку. Первые два числа это 1 и 1; каждое следующее является суммой двух предыдущих:

1 1 2 3 5 8 13 ...

Помните, что в массиве может быть любое количество чисел.

5task – program

Наращиваем уровни косвенности

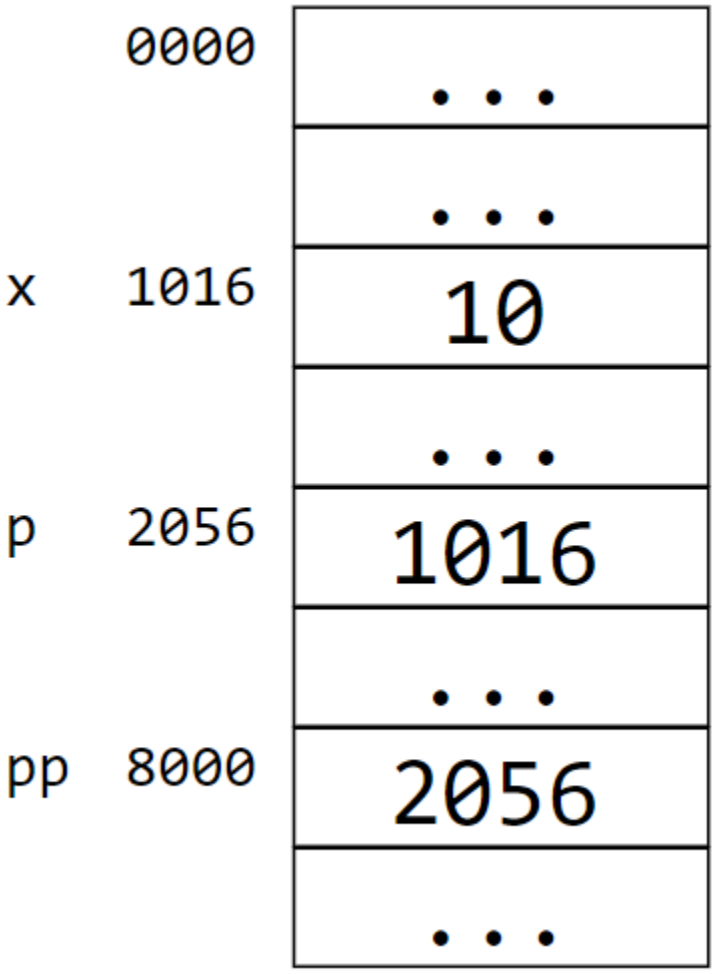
Указатели можно создавать не только на переменные численных типов (`int`, `char`) но и вообще на любые данные, у которых есть адрес. В частности, на другие указатели. Про то, зачем это бывает нужно – на следующем слайде, а пока небольшая тренировка.

Рассмотрим состояние памяти после выделения следующих переменных:

```
int    x  = 10;
```

```
int*  p  = &x;

int** pp = &p;
```

Допустим, что `x` находится по адресу 1016, `p` по адресу 2056, `pp` по адресу 8000. Можно изобразить это следующим образом:



Теперь мы увидим типичную ситуацию, при которой функция принимает указатель на указатель. Зачем тут два уровня косвенности?

Иногда из функции нужно вернуть указатель, но возвращать его через `return` неудобно.Рассмотрим функцию `array_min`, которая находит адрес минимального числа в массиве.

```
// min -- адрес указателя на минимальный элемент в массиве

// функция возвращает 0 если массив пустой

int array_min(int* array, int* limit, int** min) {

    if (array >= limit) return 0;
```

```
    *min = array;
    for( int* cur = array + 1;
                cur < limit;
                cur = cur + 1 )
    {
        if ( *cur < **min ) {
            *min = cur;
        }
    }
    return 1;
}



int main() {
    int array[] = {4,29,42,2,3};
    int* lmin = NULL;
    // только в том месте, где объявлен массив, мы можем вычислить его длину
    // sizeof(array) вернет длину в байтах
    // sizeof(array[0]) или sizeof(int) -- размер одного элемента


    if ( array_min( array,
                    array + sizeof(array)/sizeof(array[0]),
                    &lmin ) )
    {
        printf("Min is: %d\n", *lmin );
    }
```

```
    else
    {
        printf("Array is empty\n");
    }
    return 0;
}
```

Минимальный элемент есть только в непустых массивах, поэтому нам надо предусмотреть две ситуации:

- массив пустой, тогда минимума не существует.
- массив непустой, тогда вернём минимальный элемент.

Если мы просто будем возвращать минимальный элемент из функции, мы не сможем сигнализировать о том, что массив был пустым. Чтобы передать в вызывающую функцию больше информации, мы будем возвращать только число типа `int`:

- ноль (ложь), если минимума не существует
- один (истину), если минимум существует. В том случае нам также необходимо передать в вызывающую функцию указатель на минимальный элемент массива. Тип этого указателя `int*`. Как это сделать, если возвращаемое значение уже занято?

В примере функция `array_min` вызывается из функции main. Создадим в ней переменную `int* lmin`, в которую должен попасть указатель на минимальный элемент массива. Чтобы перезаписать `lmin` через указатель нужно передать адрес lmin в функцию array_min; адрес lmin имеет тип int**. Отсюда и аргумент `array_min`: `int** min`.

Иными словами, `int** min` это адрес, по которому нужно записать указатель на минимальный элемент массива.

Упражнение. Вам дана функция `predicate`, чью реализацию вы не знаете; вы можете только её вызывать. Вы знаете, что она принимает число и возвращает 0 или 1. Напишите функцию `array_contains`, которая найдёт первый элемент в массиве, который удовлетворяет условию `predicate`. Как и функция array_min из примера, функция array_contains возвращает 1 если элемент найден, 0 если не найден. Кроме того, через указатель `int** position` она должна вернуть адрес найденного элемента.

6task – program

Указатели на локальные переменные

Удобно передавать локальные переменные функции в другие функции по указателям:

```
void g( int* a ) {
   *a = 42;
}

void f() {
   int x = 0;

   g( &x );        // :)

}
```

Это работает потому что время жизни локальных переменных `f` больше, нежели переменных любой функции, которую `f` вызывает. Те функции отработают и вернутся в `f`, а затем уже произойдёт возврат из `f`.

Правило: *можно передавать локальные переменные функции* `f` *по указателям в те функции, которые* `f` *вызывает, но не "вверх" в функции, вызывающие* `f`*.*

Но нельзя передавать из функции указатель на её локальную переменную, например, так:

```
int* g() {
   int a = 42;
   return &a;    // !!!
}

void f() {
   int x = 0;

   x = *g();
```

```
}
```

Когда функция `g` завершит свою работу, её переменные "уничтожатся" вместе со всем стековым кадром функции `g`.

Указатели на уже несуществующие данные называются *висячими* (dangling pointers).

Висячие указатели

При чтении по висячему указателю может произойти:

- ничего;
- прочитается мусорное значение;
- программа аварийно завершится.

При записи по висячему указателю может произойти:

- ничего
- программа аварийно завершится сразу
- внутренний код программы, скрытый от програмиста, будет повреждён. Запись по висячему указателю нарушит внутренние структуры програмы в памяти, нарушив её логику работы. Программа продолжит работу, но в неожиданный момент аварийно завершится, выполняя, казалось бы, корректное действие вроде вызова функции или выделения массива.

Это трудноуловимые ошибки, будьте внимательны к ним.

Строки

В C строчки являются массивами из чисел типа `char`. Тип строки, обычно, `char*`, для строк нет специального типа `string`. Каждый байт в строке хранит код символа по таблице ASCII, задающей соответствие между символами и их кодами. В таблице ниже для удобства представлены коды в десятичной (Dec), шестнадцатиричной (Hex) и восьмеричной (Oct) системах счисления.

В начале таблицы располагаются символы, которые не имеют печатного представления (например, 10ый символ используется для перевода строки). Цифры занимают диапазон 48-57, прописные буквы 65-90, строчные – 97-122.

Не запоминайте коды символов! В исходном коде всегда можно написать символ в одинарных кавычках: такое выражение равно его ASCII-коду, например `'J'` равно `74`. Так гораздо легче читать исходный код.

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

За последним символом строки всегда идёт байт равный нулю, который называется *нуль-терминатор.* Таким образом, строка из *n* символов занимает *n+1* байт. Например, строчка `Hi` занимает три байта: `72`, `105`, `0`. Мы можем вручную сформировать её:

```
char str[] = {72, 105, 0};

printf( str );  // выведет "Hi"

// помните, что имя массива это адрес его начала
```

или, что лучше, мы можем использовать коды символов:

```
char str[] = {'H', 'i', 0};

printf( str );  // выведет "Hi"
```

Написав текст в двойных кавычках мы создадим нуль-терминированную строку в неизменяемой области памяти; значением выражения `"строка"` будет адрес первого символа этой строки, а её типом будет `char*`, указатель на однобайтовое число (код символа). Мы можем использовать эту строку для инициализации массива char'ов: тогда создастся массив, инициализированный символами этой строки и его можно будет менять.

```
char str[] = "Hi";
```

```
printf( str );  // выведет "Hi"

str[0] = 'X';   // так можно



char* cstr = "Hi"; // это указатель на неизменяемую строку

printf( cstr );    // выведет "Hi"

cstr[1] = 'X';     // попытка её изменения приведёт к ошибке
```

Одного байта хватит на 256 различных символов, чего очень мало: даже кириллица и китайские иероглифы туда не влезут. Для того, чтобы кодировать много символов, придумали следующую уловку: диапазон делится на две части, и если значение байта меньше 128, то он кодирует символ из таблицы ASCII выше; иначе же байт является началом кода символа, занимающего несколько байт. Так работает кодировка символов UTF-8, одна из самых распространённых кодировок стандарта Unicode.

У использования UTF-8 есть два любопытных следствия:

- Так как символы занимают от 1 до 4 байт, подсчитать длину строчки в байтах (или понять, сколько символов в строке) нетривиально.
- Чтобы получить, скажем, 6-ой символ в строке, придётся пройтись по её байтам с самого начала, т.к. шестой символ не всегда соответствует шестому байту.

Чтобы сейчас не учиться работать с UTF-8, мы используем в строках только латиницу, что делает их "простыми" -- достаточно использовать ASCII таблицу, и все символы занимают 1 байт.

Напишите функции:

- `string_count`: считает длину строки в байтах (не включая нуль-терминатор)
- `string_words`: считает количество слов в строке.

Между словами стоит произвольное количество *пробельных символов*:

- пробела `' '`
- табуляции `'\t'`
- перевода строки `'\n'`

Пробельные символы также могут стоять в начале или конце строки; **не гарантируется**, что в строке будет хотя бы один символ или хотя бы одно слово.

Sample Input:

```
  hello  world
```

Sample Output:

```
length: 12 words: 2
```

7task – program