Семинар #1: Основы C. Ввод/вывод. Операторы. Циклы. Массивы.

Ввод/вывод

Hello World!

Программа на языке C, которая печатает на экран строку `Hello World!`:

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
}
```

Функции `printf` и `scanf`

- Функция `printf` используется для печати на экран всего что угодно.
- Функция `scanf` используется для считывания значений переменных с экрана.
- Обе эти функции хранятся в библиотеке `stdio.h`. Эту библиотеку нужно подключить с помощью

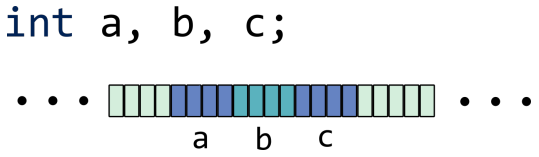
  ```
  #include <stdio.h>
  ```

Целочисленные переменные `int`:

- Переменные типа `int` нужны для хранения целых чисел.

Адрес и размер переменной:

- 1 бит - минимальная единица измерения памяти. В 1 бите может хранится либо `0` либо `1`.
- Вся память делится на ячейки, размером в 8 бит = 1 байт.
- Все эти ячейки занумерованы, номер ячейки называется адресом.
- Все переменные содержатся в памяти. Адрес переменной - это адрес первого байта переменной.
- Чтобы найти адрес переменной, нужно перед ней поставить `&`, например, `&a`
- Функции `scanf` нужно передавать именно адрес переменной, а не само значение переменной.
- Чтобы найти размер переменной в байтах: `sizeof(a)`
- Например, переменная типа `int` имеет размер 4 байта = 32 бита. Значит в ней может хранится максимум 2^{32} значений. То есть переменные типа `int` могут принимать значения от -2^{31} до 2^{31}.



Операторы

Арифметические операторы и операторы присваивания:

```
+          сложение                          =       присвоить левой части правую
-         вычитание                         +=     прибавить к левой части правую
          умножение                         -=      отнять от левой части правую
/   целочисленное деление                   /=     разделить левую часть на правую
%          остаток                          %=   левая часть становится равна остатку
                                            ++              увеличить на 1
                                            --              уменьшить на 1
```

Операторы сравнения и логические операторы:

```
==         равно                            &&    логическое И
!=       не равно                           ||   логическое ИЛИ
 >         больше                            !    логическое НЕ
>=   больше или равно
 <         меньше
<=   меньше или равно
```

Условный оператор:

```
if ( условие1 )
    сделай это
else if ( условие2 )
    сделай это
else
    сделай это
```

Циклы

Цикл `while`:

```
while ( условие )
{
    делай это
}
```

Цикл `for`:

Циклом `for` обычно удобнее пользоваться. Соответствие между циклами `while` и `for`:

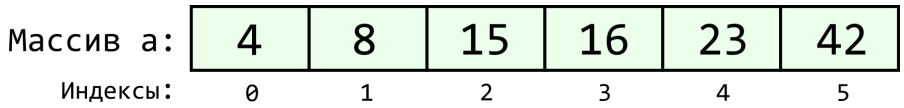
```
инициализация
while ( условие )                    for ( инициализация; условие; обновление )
{                                    {
    делай это              -->           делай это
    обновление                       }
}
```

Массивы

Массивы - это объекты, которые могут хранить внутри себя большое количество других объектов одного типа. Например, мы можем создать массив, который будет хранить 6 чисел типа `int` вот так:

```
int a[6] = {4, 8, 15, 16, 23, 42};
```

После того, как мы создали массив, мы можем получать доступ к каждому элементу массива по номеру. Номер элемента массива также называется его индексом. При этом нумерация в массиве начинается с нуля.



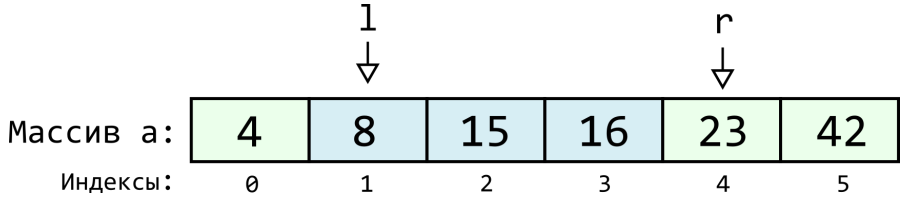
Доступ к элементу по индексу осуществляется через квадратные скобки. Например, если мы хотим поменять в массиве, определённом выше, число 15 на 20 нужно написать:

```
a[2] = 20;
```

Подмассивы

Подмассив - это некоторая последовательная часть массива. В языке C нет никаких специальных средств для работы с подмассивами. Мы будем задавать подмассив в коде как два числа – индексы граничных элементов. Будем обозначать подмассивом `a[l, r]` такую часть массива, элементы которого имеют индекс `i` в диапазоне `l <= i < r`. Обратите внимание, что мы договорились, что элемент `a[r]` не входит в подмассив `a[l, r]`.

Например, в подмассив `[1, 4]` массива `a` входят элементы `8, 15, 16`, а элемент `23` не входит.



Сортировка

Сортировка – это упорядочение элементов по возрастанию, убыванию или по какому-то другому критерию.

Сортировка выбором

Сортировка выбором – это простейший алгоритм сортировки, который заключается в следующем:
Для каждого подмассива `[j, n]` (где `j` последовательно меняется от `0` до `n - 1`) поменять местами первый и минимальный элементы этого подмассива.

```
for (int j = 0; j < n; ++j)
{
    int min_index = j;
    for (int i = j + 1; i < n; ++i)
    {
        if (a[i] < a[min_index])
            min_index = i;
    }

    int temp = a[j];
    a[j] = a[min_index];
    a[min_index] = temp;
}
```

Сортировка пузырьком

Сортировка пузырьком – это простейший алгоритм сортировки, который заключается в следующем:
Для каждого подмассива `[0, n - j]` (где `j` последовательно меняется от `0` до `n - 1`) мы делаем следующую операцию: пробегаем по этому подмассиву и, если соседние элементы стоят неправильно, то меняем их местами.

```
for (int j = 0; j < n; ++j)
{
    for (int i = 0; i < n - 1 - j; i += 1)
    {
        if (a[i] > a[i + 1])
        {
            int temp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = temp;
        }
    }
}
```

Бинарный поиск на отсортированном массиве

Если известно, что массив уже отсортирован, то многие задачи на таком массиве можно решить гораздо проще и/или эффективней. Например, просто найти минимум, максимум и медианное значение. Одной из задач, которая быстрее решается на отсортированном массиве – это задача поиска элемента в массиве. Если массив отсортирован, то решить эту задачу можно гораздо быстрее чем простой обход всех элементов.

Предположим, что массив отсортирован по возрастанию и надо найти элемент `x` в этом массиве или понять, что такого элемента в массиве не существует. Для этого мы мысленно разделим массив на 2 части:

1. Элементы, которые меньше, чем `x`
2. Элементы, которые больше или равны `x`

Затем введём две переменные-индекса `l` и `r`. В начале работы алгоритма индекс `l` будет хранить индекс фиктивного элемента, находящегося до первого (то есть `l = -1`), а индекс `r` будет хранить индекс фиктивного элемента, находящимся после последнего (то есть `r = n`).

На каждом шаге алгоритма мы будем брать середину между индексами `l` и `r` и передвигать к этой середине или индекс `l` или индекс `r`. При этом при изменении индексов должны соблюдаться условия:

```
a[l] < x
a[r] >= x
```

Алгоритм закончится тогда, когда разница между индексами не станет равным 1, то есть не станет `r == l + 1`. И так как `a[l] < x` и `a[r] >= x`, то если элемент `x` в массиве существует, то его индекс равен `r`.

Код для поиска в отсортированном массиве бинарным поиском:

```
#include <stdio.h>

int main()
{
    int n;
    int a[1000];
    scanf("%i", &n);
    for (int i = 0; i < n; ++i)
        scanf("%i", &a[i]);

    int x;
```

```
    scanf("%i", &x);

    int l = -1, r = n;
    while (r > l + 1)
    {
        int mid = (l + r) / 2;

        if (a[mid] >= x)
            r = mid;
        else
            l = mid;
    }

    if (r < n && a[r] == x)
        printf("Element found! Index = %i\n", r);
    else
        printf("Element not found!");
}
```