

Семинар #4: Функции. Решения.

Часть 1: Функции без возвращаемого значения

Функция `max` из примера ниже вычисляет максимум 2-х чисел и возвращает его.

Помимо функций с возвращаемым значением можно написать и функцию, которая ничего не возвращает. Например, `print_n_times` – печатает число `n` раз и ничего не возвращает. У функций, которые ничего не возвращают на месте возвращаемого типа стоит ключевое слово `void`. Такие функции не требуют оператора `return`, однако его всё равно можно использовать для выхода из функции.

```
#include <stdio.h>
int max(int a, int b) {
    if (a > b)
        return a;
    return b;
}

void print_n_times(int a, int n) {
    for (int i = 0; i < n; ++i) {
        printf("%i ", a);
    }
}

int main() {
    printf("%i\n", max(5, 7));
    print_n_times(7, 3);
}
```

Задачи:

1. Вызовите функцию `print_n_times` из функции `main`, чтобы 10 раз напечатать на экран число 123.
2. Напишите функцию `void print_even_numbers(int a, int b)`, которая будет печатать все четные числа от `a` до `b`. Вызовите эту функцию из функции `main`.
3. Напишите функцию `void print_rectangle(int a, int b)`, которая будет печатать прямоугольник из звёздочек `*`. Например, если эта функция будет вызвана с аргументами 4 и 3, то функция должна напечатать:

```
****
****
****
```

Вызовите эту функцию из функции `main` с различными аргументами.

4. Напишите функцию `void multi(int type, int a, int b)`, которая, в зависимости от переменной `type`, должна делать различные вещи. При `type == 1`, она должна вызывать функцию `print_even_numbers`. При `type == 2`, она должна вызывать функцию `print_n_times`. При `type == 3`, она должна вызывать функцию `print_rectangle`. При ином другом значении `type`, она должна просто печатать **Error!**. Протестируйте вашу функцию.

```
#include <stdio.h>

void print_n_times(int a, int n) {
    for (int i = 0; i < n; ++i) {
        printf("%i ", a);
    }
    printf("\n");
}

void print_even_numbers(int a, int b) {
    if (a % 2)
        a++;
    for (int i = a; i <= b; i += 2)
        printf("%i ", i);
    printf("\n");
}

void print_rectangle(int a, int b) {
    for (int i = 0; i < a; ++i) {
        for (int j = 0; j < b; ++j)
            printf("*");
        printf("\n");
    }
}

void multi(int type, int a, int b) {
    if (type == 1)
        print_even_numbers(a, b);
    else if (type == 2)
        print_n_times(a, b);
    else if (type == 3)
        print_rectangle(a, b);
    else
        printf("Error!\n");
}

int main() {
    multi(1, 2, 8);
    multi(2, 2, 8);
    multi(3, 2, 8);
}
```

Часть 2: Рекурсия

Рекурсивная функция – это функция, которая вызывает саму себя. В примере ниже функция `counter` – рекурсивная. Если этой функции передать, скажем, число 5, то она напечатает это число и вызовет функцию `counter`, передав ей число 4. Так будет продолжаться пока число не дойдёт до 0.

```
#include <stdio.h>
void counter(int n) {
    if (n <= 0) {
        return;
    }
    printf("%i ", n);
    counter(n - 1);
}
int main() {
    counter(10);
}
```

Задачи:

- Немного измените функцию `counter` чтобы она печатала числа с шагом 2.

```
void counter(int n) {
    if (n <= 0) {
        return;
    }
    printf("%i ", n);
    counter(n - 2);
}
```

- Немного измените функцию `counter` чтобы она делила число `n` нацело на 2 при каждом рекурсивном вызове.

```
void counter(int n) {
    if (n <= 0) {
        return;
    }
    printf("%i ", n);
    counter(n / 2);
}
```

- Немного измените изначальную функцию `counter` чтобы она печатала числа по возрастанию.

```
void counter(int n) {
    if (n <= 0) {
        return;
    }
    counter(n - 1);
    printf("%i ", n);
}
```

Для лучшего понятия, что происходит в этой функциях `counter` рассмотрим на схемы вызовов рекурсивной функции. Для изначальной функции `counter` она выглядит следующим образом (предположим, что вызываем `counter(4)`):

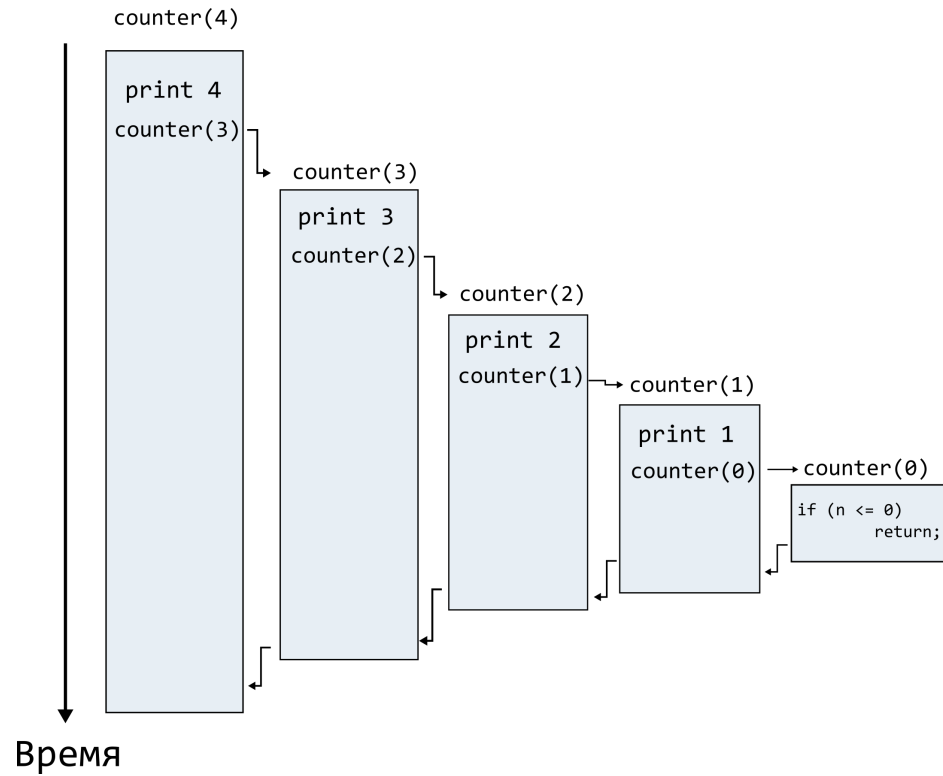
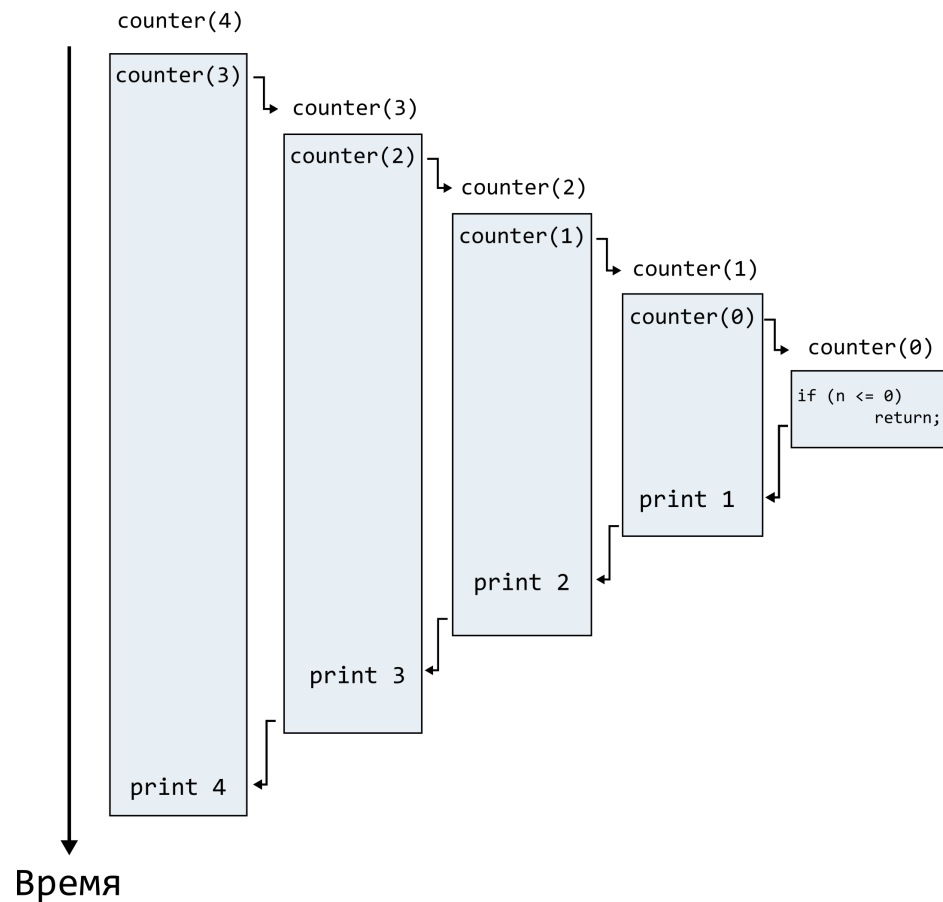


Схема для функции `counter`, которая печатает числа по возрастанию выглядит так:



- **Факториал:** Напишите функцию `int fact(int n)`, которая рекурсивно вычисляет факториал числа `n`. Вызовите эту функцию из `main`.

```
#include <stdio.h>
int fact(int n) {
    if (n == 0)
        return 1;
    return n * fact(n - 1);
}
int main() {
    printf("%i\n", fact(5));
}
```

- **Возведение в целую степень:** Напишите функцию `double power(double a, int n)`, которая рекурсивно возводит число `a` в целую степень `n` по формуле:

$$a^n = a \cdot (a^{n-1})$$

```
#include <stdio.h>
double power(double a, int n) {
    if (n == 0)
        return 1;
    return a * power(a, n - 1);
}
int main() {
    printf("%lf\n", power(2, 4));
}
```

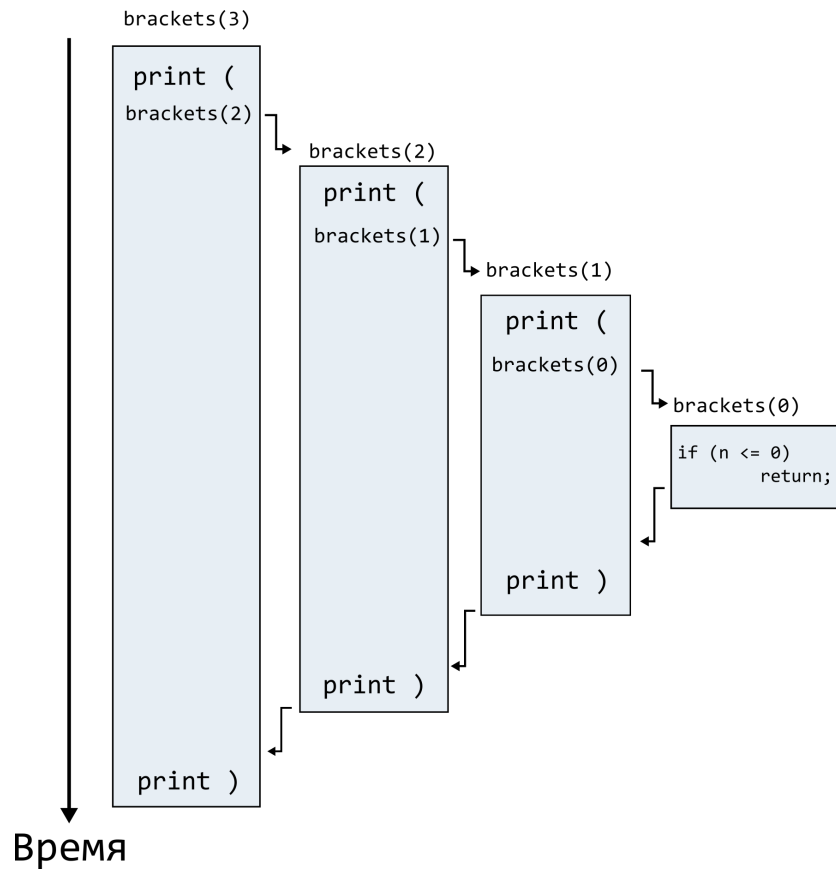
- **Скобочки:** Напишите рекурсивную функцию `brackets`, которая будет печатать некоторую скобочную последовательность. `brackets(n)` должна сначала печатать `n` открывающихся скобочек, а затем `n` закрывающихся. Например, вызов `bracket(4)` должен напечатать `((((()))`.

Чтобы это сделать рекурсивно нужно сделать следующее:

- Напечатать открывающуюся скобку
- Напечатать `n-1` открывающихся и `n-1` закрывающихся скобок вызовом рекурсивной функции
- Напечатать закрывающуюся скобку

```
#include <stdio.h>
void brackets(int n) {
    if (n == 0)
        return;
    printf("(");
    brackets(n - 1);
    printf(")");
}
int main() {
    brackets(5);
}
```

Схема вызовов для этой функции(вызываем `brackets(3)`):



- **Фибоначчи:** Для вычисления числа Фибоначчи было написано 2 функции. Функция `fib` вычисляет число Фибоначчи итеративно(то есть с помощью цикла), а функция `fibrec` вычисляет то же самое рекурсивно. Так как число фибоначчи может быть большим, то в качестве типа данных используется `long long` вместо `int`. Переменных типа `long long` могут хранить числа до примерно 10^{19} . Числа Фибоначчи до 93-го номера могут храниться в таких переменных.

```
#include <stdio.h>

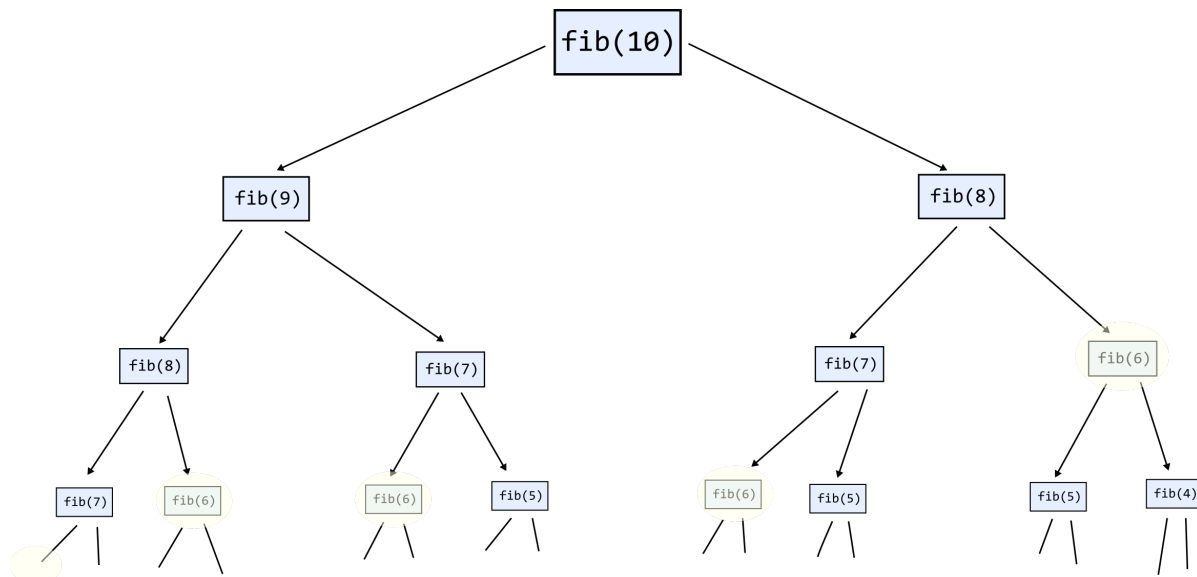
long long fib(int n) {
    long long a = 0, b = 1;
    for (int i = 1; i < n; ++i) {
        long long temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}

long long fibrec(int n) {
    if (n < 2) {
        return n;
    }
    return fibrec(n - 1) + fibrec(n - 2);
}

int main() {
    printf("%lli\n", fib(20));
    printf("%lli\n", fibrec(20));
}
```

Кажется, что обе функции работают, но если посчитать числа Фибоначчи не от 20, а от 60-ти, то рекурсивная функция перестает работать. Почему это происходит?

Рекурсивная функция вычисления чисел Фибоначчи работает так медленно потому что она по многу раз вычисляет одни и те же значения. На изображении ниже представлена схема вызовов этой функции для вычисления 10-го числа Фибоначчи.



Видно что промежуточные числа Фибоначчи вычисляются заново по многу раз. Например, `fib(6)` вызывается 5 раз в процессе вычисления `fib(10)`, `fib(5)` вызывается 8 раз, а меньшие числа Фибоначчи вычисляются ещё чаще. Получается, что количество операций для вычисления одного числа Фибоначчи рекурсивным способом экспоненциально большое.

Рекурсивную функцию можно исправить если в процессе выполнения запоминать все промежуточные вычисления и не делать их заново. Для этого заведём глобальный массив `data` в котором будем хранить все промежуточные числа Фибоначчи. В начале все элементы этого массива равны нулю. Теперь мы продолжаем рекурсию только если мы не вычисляли данное число ранее и `data[n] == 0`.

```
#include <stdio.h>
long long data[1000];

long long fibrec(int n) {
    if (n < 2) {
        return n;
    }
    if (data[n] == 0) {
        data[n] = fibrec(n - 1) + fibrec(n - 2);
    }
    return data[n];
}

int main() {
    for (int i = 0; i < 1000; ++i)
        data[i] = 0;
    printf("%lli\n", fibrec(90));
}
```

- **Бинарное возведение в целую степень:** Напишите функцию `double binpow(double a, int n)`, которая рекурсивно возводит число `a` в целую степень `n` по формуле:

$$a^n = \begin{cases} a \cdot a^{n-1}, & \text{если } n - \text{нечётное} \\ a^{n/2} \cdot a^{n/2}, & \text{если } n - \text{чётное} \end{cases}$$

Протестируйте функции `power` и `binpow` на следующих тестах:

ВХОД	ВЫХОД
2 4	16
1.05 100	131.501
1.00001 1000000	22025.364
1.00000001 2000000000	485165075.539

Какая из этих функций более эффективна?

```
#include <stdio.h>

double binpow(double a, int n) {
    if (n == 0) {
        return 1;
    }
    if (n % 2) {
        return a * binpow(a, n - 1);
    }
    else {
        double temp = binpow(a, n / 2);
        return temp * temp;
    }
}

int main() {
    printf("%lf\n", binpow(1.05, 100));
}
```

В этой функции важно не сделать ту же ошибку, что встретилась в задаче вычисления чисел Фибоначчи, и не вызвать `binpow(a, n/2)` два раза.

Для вычисления `n`-й степени функции `power` требуется порядка n операции умножения, а функции `binpow` для того же самого требуется порядка $\log_2(n)$ операций. Конечно, функция `binpow` намного быстрее.

Часть 3: Передача массива в функцию

Массивы можно передавать в функции. Однако, передача массива в функцию в языке C устроена таким образом, что узнать размер массива внутри функции нельзя. Поэтому размер массива нужно тоже передавать в функцию вместе с массивом.

Пример двух функций: одна печатает массив на экран, другая прибавляет ко всем элементам массива 1.

```
#include <stdio.h>
void print_array(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%i ", array[i]);
    }
    printf("\n");
}
void inc(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        array[i] += 1;
    }
}
int main() {
    int a[10] = {4, 8, 15, 16, 23, 42};
    print_array(a, 6);
    inc(a, 6);
    print_array(a, 6);
}
```

Задачи:

Все эти функции нужно писать в одном файле. Также всех их нужно протестировать, вызвав из функции `main`. Все эти функции не должны ничего печатать, а должны просто изменять массив или возвращать значения. Вся печать должно совершаться посредством вызова функции `print_array` из функции `main`.

- Напишите функцию `mult2`, которая принимает на вход массив и его размер, а затем умножает каждый элемент массива на 2. Протестируйте эту функцию в функции `main`. Используйте `print_array` для печати массива.
- Напишите функцию `sqr`, которая принимает на вход массив и его размер, а затем возводит каждый элемент массива в квадрат.
- Напишите функцию `add_x`, которая принимает на вход массив, его размер и некоторое число `x`, а затем прибавляет ко всем элементам массива это число.
- Напишите функцию `sum`, которая принимает на вход массив и его размер и возвращает сумму элементов этого массива.
- Напишите функцию `max`, которая возвращает максимальный элемент массива.
- Напишите функцию `int sum_subarray(int array[], int l, int r)`, которая принимает на вход массив и границы подмассива и возвращает сумму элементов данного подмассива. Например, сумма подмассива `a[1, 4]` равна 39.
- Напишите функцию `void add_array(int C[], int A[], int B[], int size)`. Эта функция должна складывать массивы `A` и `B` и записывать результат в массив `C`.

```

#include <stdio.h>
void print_array(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%i ", array[i]);
    }
    printf("\n");
}
void inc(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        array[i] += 1;
    }
}
void mult2(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        array[i] *= 2;
    }
}
void sqr(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        array[i] *= array[i];
    }
}
void add_x(int array[], int size, int x) {
    for (int i = 0; i < size; ++i) {
        array[i] += x;
    }
}
int sum_array(int array[], int size) {
    int result = 0;
    for (int i = 0; i < size; ++i) {
        result += array[i];
    }
    return result;
}
int sum_subarray(int array[], int l, int r) {
    int result = 0;
    for (int i = l; i < r; ++i) {
        result += array[i];
    }
    return result;
}

int max(int a, int b) {
    return a > b ? a : b;
}
int max_array(int array[], int size) {
    int result = array[0];
    for (int i = 0; i < size; ++i)
        result = max(result, array[i]);
    return result;
}

```

```

void add_array(int C[], int A[], int B[], int size) {
    for (int i = 0; i < size; ++i) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int a[10] = {4, 8, 15, 16, 23, 42};
    print_array(a, 6);
    printf("Sum      %i\n", sum_array(a, 6));
    printf("Sum[1,4]  %i\n", sum_subarray(a, 1, 4));
    printf("Max      %i\n", max_array(a, 6));

    printf("Square:\n");
    sqr(a, 6);
    print_array(a, 6);

    printf("\nAdd arrays:\n");
    int A[6] = {4, 8, 15, 16, 23, 42};
    int B[6] = {5, 9, 1, 55, 90, 20};
    int C[6];
    add_array(C, A, B, 6);
    print_array(C, 6);
}

```

Часть 4: Рекурсия на подотрезках

Рекурсию можно применять и на массивах. В примере ниже написана функция `sumrec`, которая рекурсивно вычисляет сумму подмассива. Для этого она делает следующее:

- Если в подмассиве 1 элемент, то нужно просто вернуть этот элемент.
- Вычисляет середину подмассива
- Рекурсивно вызывает себя для вычисления сумм левой половины подмассива и правой половины.
- Возвращает сумму левой и правой половины

```

#include <stdio.h>

int sumrec(int array[], int l, int r) {
    if (r - l == 1) {
        return array[l];
    }
    int mid = l + (r - l) / 2;
    return sumrec(array, l, mid) + sumrec(array, mid, r);
}

int main() {
    int a[6] = {4, 8, 15, 16, 23, 42};
    printf("%i\n", sumrec(a, 0, 6));
}

```

Все задачи можно решить как с помощью цикла, так и с помощью рекурсии. Чаще решение с помощью цикла является более понятным и простым, но бывают задачи, решение которых проще сделать с помощью рекурсии.

Задачи:

- Напишите рекурсивную функцию, которая вычисляет максимум на подмассиве.

```
#include <stdio.h>
int max(int a, int b) {
    return a > b ? a : b;
}

int maxrec(int array[], int l, int r) {
    if (r - l == 1) {
        return array[l];
    }
    int mid = l + (r - l) / 2;
    return max(maxrec(array, l, mid), maxrec(array, mid, r));
}

int main() {
    int a[6] = {4, 8, 95, 16, 23, 42};
    printf("%i\n", maxrec(a, 0, 6));
}
```

- Функция сортировки выбором выглядит так (нужно добавить функцию `print_array`):

```
#include <stdio.h>
void selection_sort(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        int min_index = i;
        for (int j = i + 1; j < size; ++j)
            if (array[j] < array[min_index])
                min_index = j;

        int temp = array[i];
        array[i] = array[min_index];
        array[min_index] = temp;
    }
}

int main() {
    int a[10] = {15, 11, 54, 8, 1, 6, 5, 97, 1, 22};
    print_array(a, 10);
    selection_sort(a, 10);
    print_array(a, 10);
}
```

Напишите функцию `selection_sort_rec`, которая будет сортировать выбором, но рекурсивно.

```

#include <stdio.h>
void print_array(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%i ", array[i]);
    }
    printf("\n");
}
void selection_sort_rec(int array[], int l, int r) {
    if (r - l <= 1) {
        return;
    }
    int min_index = l;
    for (int j = l + 1; j < r; ++j)
        if (array[j] < array[min_index])
            min_index = j;

    int temp = array[l];
    array[l] = array[min_index];
    array[min_index] = temp;

    selection_sort_rec(array, l + 1, r);
}
int main() {
    int a[10] = {15, 11, 54, 8, 1, 6, 5, 97, 1, 22};
    print_array(a, 10);
    selection_sort_rec(a, 0, 10);
    print_array(a, 10);
}

```

Часть 5: Передача по указателю

Отличие массивов от других переменных заключается в том, что при их изменении внутри функции, они меняются и вне функций. С обычными переменными это не работает. Для того чтобы можно было менять переменные внутри функций нужно использовать указатели. Пример программы, которая создаёт указатель `p`. Этот указатель хранит адрес переменной `a`.

```
#include <stdio.h>
int main() {
    int a = 100;
    int* p = &a;

    *p = 321;
    printf("%d\n", a);
}
```

`int*` – это тип переменной `p` – указатель на `int`.

`*p` – в этой строки символ `*` имеет другой смысл. Она означает, что нужно пойти по адресу, который хранится в `p` и воспринимать эту область памяти как число `int`.

- Создайте переменную `b` типа `float` и присвойте ей какое-либо значение. Создайте переменную `p` типа указатель на `float` (`p` – это сокращение от `pointer` – указатель) и присвойте ей значение – адрес переменной `b`. Измените переменную `b`, используя только переменную `p`.

```
#include <stdio.h>
int main() {
    float b = 123.45;
    float* p = &b;
    *p /= 2;
    printf("%f\n", b);
}
```

- Создайте переменную `pp` и присвойте ей значение – адрес переменной `p`. Измените переменную `b`, используя только переменную `pp`.

```
#include <stdio.h>
int main() {
    float b = 123.45;
    float* p = &b;
    float** pp = &p;
    **pp /= 2;
    printf("%f\n", b);
}
```

Пример передачи в функцию с помощью указателей. Пытаемся увеличить переменную на 10.

```
#include <stdio.h>
void add10_wrong(int a){
    a += 10;
}
void add10_right(int* p){
    *p += 10;
}
int main() {
    int a = 80;
    add10_wrong(a);
    printf("%d\n", a);

    add10_right(&a);
    printf("%d\n", a);
}
```

- Написать функцию `void mult2(int* p)`, которая удваивает число, поступающее на вход, используя указатель на эту переменную. Протестируйте эту функцию в функции `main`.

```
#include <stdio.h>
void mult2(int* p) {
    *p *= 2;
}
int main() {
    int a = 10;
    mult2(&a);
    printf("%i\n", a);
}
```

- Написать функцию `swap`, которая меняет значения 2-х переменных типа `int` местами. Используйте эту функцию в функции `main`.

```
#include <stdio.h>
void swap(int* pa, int* pb) {
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
int main() {
    int a = 10, b = 20;
    printf("%i %i\n", a, b);
    swap(&a, &b);
    printf("%i %i\n", a, b);
}
```