

# Приемы программирования на языках C и C++

## Содержание

1	Ресурсы по языкам C и C++	1
2	Указатели на функции	1
3	Коротко о стеке и куче	3
4	Динамическое выделение памяти	3
5	Функции в C	4
5.1	Параметры по ссылке и по значению . . . . .	4
5.2	Передача массива в качестве параметра . . . . .	5
	Список литературы	6
	Список листингов	6

## 1. Ресурсы по языкам C и C++

<https://learn.c.info/c/>

## 2. Указатели на функции

*Указатель* – это переменная, содержащая *адрес* другой переменной. Если одна переменная содержит адрес другой переменной, то говорят, что она *указывает* на ту переменную [1, стр. 95].

Синтаксис объявления указателя

```
тип *имя_указателя;
```

тип – это тип переменной, на которую будет ссылаться указатель.

Указатель, не ссылающийся на конкретную ячейку памяти, должен быть равен *нулю*. Использование нулевого указателя – это всего лишь общепринятое соглашение.

```
// объявляем указатель на целочисленную переменную и инициализируем его с помощью NULL
int *p = NULL;
```

---

### Замечание

Указатель можно сравнивать с нулем или с NULL, но нельзя NULL сравнивать с переменной целого типа или типа с плавающей точкой [1, стр. 103]

---

Несмотря на то, что функция не является переменной, она располагается в памяти, и, следовательно, ее адрес можно *присваивать указателю*. Этот адрес считается точкой входа в функцию. Именно он используется при вызове. Поскольку *указатель может ссылаться на функцию*, ее

можно вызывать с помощью этого указателя. Это позволяет также передавать функцию другим функциям в качестве аргументов [1].

Адрес функции задается ее именем, указанным без скобок и аргументов.

Пример

```
// Подключение заголовочных файлов с помощью инструкции препроцессора #include
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// main - это точка входа приложения
int main() {
    char s1[10], s2[10]; // Объявление строк, как массива символов
    int (*p) (const char *, const char *); // Объявление указателя на функцию
    p = strcmp; // Инициализация указателя; указателю присваивается адрес функции strcmp

    printf("Enter first string: ");
    scanf("%s", &s1);

    printf("Enter second string: ");
    scanf("%s", &s2);

    void test(
        char *x, // Указатель на символьную переменную; ожидает получить адрес
        char *y, // Указатель на символьную переменную; ожидает получить адрес
        int (*cmp) (const char *, const char *) // Указатель на функцию!
    ) {
        // Сравнивает две строки
        printf("Comparison ... \n");
        if (!((*cmp)(x, y))) { // <==NB вызов функции strcmp как (*cmp)(x, y)
            printf("=> Equal!");
        } else {
            printf("=> Not equal.");
        }
    }

    // s1 и s2 это указатели на первый символ массива
    test(s1, s2, p);

    return 0;
}
```

Важный момент: имя массива является указателем на его первый элемент [1, стр. 78]. То есть

```
double *p; // Объявление указателя на вещественную переменную
double total[50]; // Объявление на массив вещественных чисел двойной точности
p = total; // Указателю p присваивается адрес первого элемента массива total
```

Поэтому в функцию `test` передаются не явные адреса `&s1` и `&s2`, а просто имена переменных `s1` и `s2`. Ведь имена переменных `s1` и `s2` связаны со строками (по сути массивами символов), а значит имена указывают на свои первые символы и таким образом в функцию на самом деле передаются адреса первых символов этих строк.

### 3. Коротко о стеке и куче

Стек – это область памяти, которую вы, как программист, не контролируете никоим образом. В нее записываются переменные и информация, которые создаются в результате вызова любых функций. Когда функция заканчивает работу, то вся информация о ее вызове и ее переменные удаляются из стека автоматически.

Куча – это область памяти, которую контролируют непосредственно программисты.

### 4. Динамическое выделение памяти

Благодаря *динамическому выделению памяти* (dynamic allocation) программа может получать необходимую ей память в ходе выполнения, а не на этапе компиляции.

В языке С есть две функции динамического выделения памяти – `malloc()` и `calloc()`. И одна функцию освобождения памяти – `free()`.

Память, выделяемая функциями динамического распределения, находится в куче (heap), которая представляет собой область свободной памяти, расположенную между кодом программы, сегментом данных и стеком.

Прототип функции `malloc()`

```
void *malloc(size_t количество_байтов)
```

Функция `malloc()` возвращает указатель типа `void *`. Это означает, что его можно присваивать указателю любого типа. В случае успеха функция `malloc()` возвращает *указатель на первый байт памяти* (или другими словами, адрес зарезервированного участка памяти на куче), в противном случае (т.е. если размера кучи не достаточно для успешного выделения памяти) – нулевой указатель (`NULL`).

Также полезна функция `calloc()`, позволяющая выделять память под данные конкретного типа данных

```
void *calloc(size_t num, size_t size)
```

Размер выделенной памяти будет равен величине `num * size`, где `size` задается в байтах.

Следующий пример выделяет 2000 байт непрерывной памяти

```
char *s; // объявление указателя на символьную переменную
s = malloc(2000);
```

После этого указатель `s` будет ссылаться на первый из 2000 байт выделенной памяти.

Пример. Пусть требуется вычислить максимум в массиве, размер которого мы заранее не знаем

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

int main() {
    int i, num, size;
    float *data = NULL; // объявляем указатель на вещественную переменную

    printf("Enter number of elems: ");
    scanf("%d", &num);

    size = num * sizeof(float);
```

```

// выделяем память
if ((data = (float *) malloc(size)) != NULL) {
    printf("Allocated %d bytes of memory\n", size);
    // теперь с указателем data можно работать как с обычным массивом
    for (i = 0; i < num; ++i) {
        printf("Enter elem data[%d]=", i);
        scanf("%f", data + i); // 'data + i' адресная арифметика
        /*
        Здесь можно было бы использовать нотацию []
        для обращения к элементам массива data, т.е.
        ...
        scanf("%f", &value);
        data[i] = value;
        ...
        но тогда бы пришлось объявлять промежуточную переменную value;
        адресная арифметика позволяет передавать значения
        элементам массива без создания лишних переменных
        */
    }
} else {
    printf("Oops ...");
    exit(1);
}

for (i = 0; i < num; ++i) {
    if (*data < *(data + i)) { // значение первого элемента сравнивается с data[i]
        *data = *(data + i); // присваиваем новое значение первому элементу массива
    }
}

printf("Max(data)=%.2f", *data);
}

```

NB: оператор \* справа от символа присваивания это оператор разименования указателя (просто по адресу переменной получаем значение). А оператор \* слева от символа присваивания означает, что будет изменено значение соответствующей переменной.

## 5. Функции в C

Синтаксис объявления функции выглядит так

```

<тип> имя_функции (список_аргументов) {
    тело_функции;
}

```

Функция может возвращать переменные любого типа, кроме массива! [1, стр. 109]

### 5.1. Параметры по ссылке и по значению

Параметры можно передавать *по ссылке* (call by reference) и *по значению* (call by value) [1, стр. 112]. В случае *передачи параметра по значению* формальному параметру функции присваивается копия значения аргумента и любые изменения копии никак не отражаются на оригинале.

Пример

Передача параметра по ЗНАЧЕНИЮ

```

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

int main() {
    int n = 1;
    printf("n from global: %d\n", n); // n from global: 1

    int change(int a) {
        // присваивание значения переменной 'a'
        // никак не отражается на значении глобальной переменной 'n'
        a = 2;
        printf("Change = %d\n", a); // Change = 2
        return a;
    }

    change(n);
    printf("n from global: %d", n); // n from global: 1
}

```

При передаче параметра по ссылке у функции есть возможность изменить значение переданного аргумента, поскольку в функцию передается не значение, а *адрес* аргумента. Внутри функции этот *адрес открывает доступ* к фактическому аргументу и все изменения будут отражены на аргументе.

Если нужно передавать параметры по ссылке, то указывают \* перед именем параметра. А при вызове функции нужно указать & перед именем параметра.

Пример

#### Передача параметра по ССЫЛКЕ

```

int n = 10;

printf("n from global: %d", n); // n from global: 10

// здесь функция change ожидает получить адрес переменной!
int change(int *a) { // a -> указатель на целочисленную переменную
    // присваивание нового значения
    // по сути изменение значения переменной 'n' в глобальной области видимости
    *a = 2; // запись
    printf("Change = %d\n", *a); // чтение; Change = 2
    return a;
}

change(&n); // в функцию передается АДРЕС переменной 'n'
printf("n from global: %d", n); // n from global: 2

```

Внутри функции нужно использовать оператор \* при каждом обращении к аргументу – независимо от того, хотите ли вы *прочитать* или *записать* его значение [1, стр. 113]. А в основной программе используйте & для передачи аргументов функции.

## 5.2. Передача массива в качестве параметра

Если аргументов функции является массив, то ей передается *адрес массива*. Эта ситуация представляет собой исключение из общепринятого правила передачи параметров по значению. Функция, получившая массив, получает доступ ко всем его элементам и может его модифицировать [1, стр. 114].

```

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

int main() {
    int i, t[5]; // объявляем целочисленную переменную и массив целых чисел емкостью 5

    // при вызове этой функции будет передан адрес первого элемента массива 't',
    // формальный параметр следует описывать так, будто речь идет о передаче параметра по значению
    void printArray(int arr[5]) {
        for (i = 0; i < 5; ++i) {
            printf("arr[%d]=%d\n", i, arr[i]);
        }
    }

    for (i = 0; i < 5; ++i) {
        t[i] = i * i + rand() % 10;
    }
    printArray(t); // printArray(&t[0]);
}

```

## Список литературы

1. *Кольцов Д.М.* Си на примерах. Практика, практика и только практика. – СПб.: Наука и Техника, 2019. – 288 с.

## Листинги