**Функция** — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция может содержаться как в основном файле, может быть определена в заголовочном файле и включена в основной.

**Составные части функции:**

**Сигнатура** функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

**Семантика** функции определяет способ реализации функции. Обычно представляет собой тело функции.

**Определение функции**

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

* тип возвращаемого значения;
* имя функции;
* информация о формальных аргументах;
* тело функции.

Определение функции имеет следующий синтаксис:

ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)

{

   ТелоФункции;

  …

  return(ВозвращаемоеЗначение);

}

Определение функции может быть написано как перед управляющей функцией main(Или перед любой другой функцией из которой происходит вызов), так и после нее, но в таком случае нужно перед функцией main объявить ее, то есть написать ее прототип:

ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов);

int main()

{

  return 0;

}

Здесь вместо списка параметров достаточно списка типов параметров в нужном порядке.

Формальные аргументы (*или параметры*) функции — это переменные в вызываемой функции, они описывают данные, передаваемые функции:

* Их количество;
* Тип дынных;
* Имя, по которому можно обратиться к параметру в теле функции.

Фактические аргументы( *или просто аргументы*) функции – фактические данные, передаваемые функции, при вызове присваиваются параметрам. Аргумент может быть константой, переменной или выражением. Если фактический аргумент представлен в виде выражения, то его значение сначала вычисляется, а затем передается в вызываемую функцию.

При несоответствии количества передаваемых аргументов количеству параметров выведется ошибка и код не скомпилируются.

При несоответствии типов аргумента и параметра тип аргумента неявно приведется к типу параметра, и функция отработает не так, как задумывалась. Приведение типов происходит неявно и только тогда, когда это возможно. Если функция получает число в качестве аргумента, то нельзя ей передать переменную строку, например "20" и т.д. Вообще, лучше всегда использовать верный тип или явно приводить тип к нужному.

1. #include <stdio.h>
2. int sum(float \*n, int m)
3. {
4. return n+m;
5. }
6. int main()
7. {
8. int x = 3;
9. int y = 5;
10. printf("%d\n", sum(x, y));//23
11. return 0;
12. }

## Возврат в вызывающую функцию

По окончании выполнения вызываемой функции осуществляется возврат значения в точку ее вызова. Это значение присваивается переменной, тип которой должен соответствовать типу возвращаемого значения функции. Возвращаемое значение может являться аргументом другой функции, например printf.

Функция может передать в вызывающую программу только одно значение. Для передачи возвращаемого значения в вызывающую функцию используется оператор return.

Оператор return также завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Оператор return не обязательно должен находиться в конце тела функции.

Функции могут и не возвращать значения, а просто выполнять некоторые вычисления. В этом случае указывается пустой тип возвращаемого значения void, а оператор return может либо отсутствовать, либо не возвращать никакого значения:

Функция может вернуть только одно значение. В случае если требуется вернуть более одного значения, остальные переменные, значения которых должны быть возвращены, передаются как указатели или ссылки.

Пример:

void swap(int\*a, int \*b)

{

  int tmp = \*a;

  \*a = \*b;

  \*b = tmp;

}

Так как при вызове функции в ней создаются локальные переменные(копии аргументов), которые после завершения ее работы затрутся другими данными, мы не можем возвращать из функции созданный в ее теле массив, или какие-либо другие указатели на данные, находящиеся в стеке этой функции, т.к. получим непредсказуемое поведение.

Пример:

void change(int x){

//функция для изменения значения, но она его не меняет

  x = 100;

  printf("%d", x);100

}

int main()

{

    int x = 3;

    change(x);

    printf("x = %d\n", x);//3

}

Работа с массивами

Как я и писал выше при вызове в функции создаются копии аргументов,

Для того чтоб передать массив в функцию, нужно передать в качестве аргументов указатель на массив(его имя) и его длину. Для того чтоб передать строку будет достаточно только указателя на ее начало.

Пример.

**void** printArray(***int*** *\*arr*, **unsigned** size) {

**unsigned** i;

**for** (i = 0; i < size; i++) {

**printf**("%d ", arr[i]);

    }

}

//или

**void** printArray(***int*** *arr[]*, **unsigned** size) {

**unsigned** i;

**for** (i = 0; i < size; i++) {

**printf**("%d ", arr[i]);

    }

}

При передаче двумерного массива нужно указывать размерности(хотя бы последнюю):

**void** printArray(***int*** *arr[][5]*, **unsigned** size) {

**unsigned** i, j;

**for** (i = 0; i < size; i++) {

**for** (j = 0; j < 5; j++) {

**printf**("%d ", arr[i][j]);

        }

**printf**("\n");

    }

}

//Или

**void** printArray(***int*** *(\*arr)[5]*, **unsigned** size);

//int (\*arr)[5] – указатель на массив целых длины 5

Если двумерный массив создан динамически, то можно передавать указатель на указатель. Например функция, которая получает массив слов и возвращает массив целых, равных длине каждого слова:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #include <conio.h>  #include <stdio.h>  #include <string.h>  #include <stdlib.h>    #define SIZE 10    **unsigned**\* getLengths(**const** **char** \*\*words, **unsigned** size) {  **unsigned** \*lengths = NULL;  **unsigned** i;      lengths = (**unsigned**\*) **malloc**(size \* **sizeof**(**unsigned**));  **for** (i = 0; i < size; i++) {          lengths[i] = **strlen**(words[i]);      }  **return** lengths;  }    **void** main() {  **char** \*\*words = NULL;  **char** buffer[128];  **unsigned** i;  **unsigned** \*len = NULL;      words = (**char**\*\*) **malloc**(SIZE \* **sizeof**(**char**\*));    **for** (i = 0; i < SIZE; i++) {  **printf**("%d. ", i);  **scanf**("%127s", buffer);          words[i] = (**char**\*) **malloc**(128);  **strcpy**(words[i], buffer);      }        len = getLengths(words, SIZE);  **for** (i = 0; i < SIZE; i++) {  **printf**("%d ", len[i]);  **free**(words[i]);      }  **free**(words);  **free**(len);}  //или  **void** getLengths(**const** **char** \*\*words, **unsigned** size, **unsigned** \*out) {  **unsigned** i;  **for** (i = 0; i < size; i++) {          out[i] = **strlen**(words[i]);      }  } |
|  |  |