
# C Programming

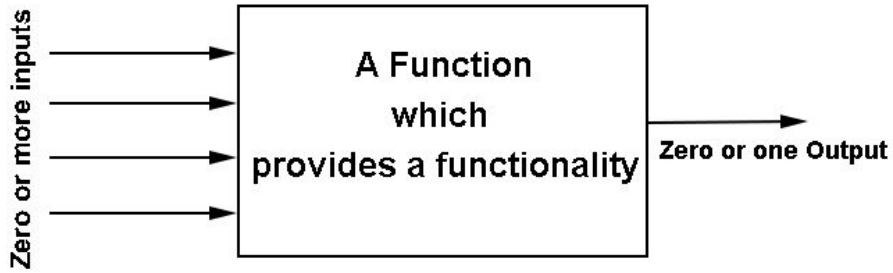
## Functions

# Functions in C

- A function is a block of organized and reusable code, used to perform a task.
- Functions provide better modularity for programs and a high degree of code reusing.
- Functions are the main building block of a C program. E.g. main, printf, sqrt and etc.
- A function is declared as

  ***return_type function_name(parameter_declarations);***

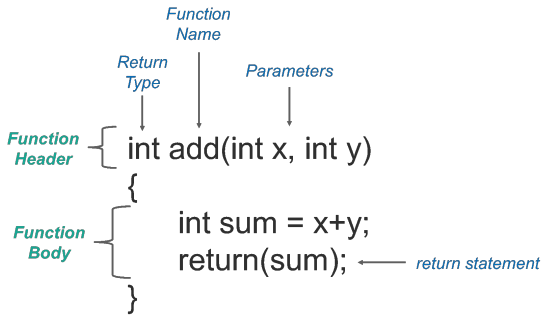
  - This is also called the function prototype
  - Functions shall be declared before using




- A function is defined (implemented) as

  *return_type function_name(parameter_declarations) // The Function head*

  *{ /* Declarations and statements */ } // The function body*




- A function can be declared and defined at once.

# Functions in C

- The ***return_type*** can be void or any object type except **function** and **array** types
  - But we can return a pointer to any type. E.g. char *get_input(int length);
    - E.g. An array is a pointer; int array[10]; array is a **pointer** to int.
- The ***return_type*** can have a storage class of static or extern
  - E.g. static int add(int x, int y); or extern int add(int x, int y);
  - By default a function defined in a C file is extern
    - It means that the scope of such a function is global and it can be used in any file in the program
  - When a function is defined as static, its scope is limited to the file the function is defined in
  - You can not use extern for a static defined function
- ***function_name*** is an identifier and follows the rules of identifiers and scopes in C
- **parameter_declarations** are a comma-separated list of the parameter declarations

# Functions in C

- If a function has no parameters, we shall use void as the **parameter_declarations**
- The parameters of a function are ordinary **local** variables.
  - Their scope is the function block.
- A function can change the value of a parameter without affecting the value of the variable used in the function call
  - But we shall avoid modifying parameters of functions
- We can only use register as the storage class in declaring function parameters
- To declare an array as a function parameter you can generally do it in two ways
  - As ***type parameter_name[ ]***. E.g. void func(int length, int array[length]);
    - We can specify the length of the array in the declaration. E.g. void func(int array[**5**], ...);
  - As a pointer ***type *parameter_name.*** E.g. void func(int *array, int length);

# Functions in C

- In C it is possible to use const, volatile, restrict and static with an integer constant inside the square brackets of an array declaration as a parameter.
  - E.g. int func(int array[const static 5], ...);
  - **We shall avoid using them.**
- When we specify the size in the declaration of an array as a parameter
  - E.g. int func(int array[**5**], ...);
  - We shall pass an array with same size to the function.
- A function to handle an array of any size
  - **int func(int length, int array[length]);** or
  - int func(int length, int array[]); or
  - **int func(int *array, int length);**
  - ***func*** can handle arrays of any size

# Functions in C

- To declare a multidimensional array as a parameter, we shall specify size of the dimensions and only size of the first dimension can be omitted.
  - E.g. **void func(int rows, int columns, int array[rows][columns]);** or void func(int rows, int array[ ][**5**]);
- In C it is possible to pass arguments to the main function
  - int main(int argc, char *argv[ ]);
    - **argc** is the number of the arguments; name of the program is the first argument.
    - **argv** is an array of the arguments as strings (char *)
    - E.g. run a program like: $ ./program 12 hello test => you can get 12, hello and test in the main function
- In a function declaration (prototype) it is possible to omit the parameter names
  - E.g. void func(int [ ][**5**], int, int);
  - **But we shall specify name of parameters**

# Functions in C

- In C, it is possible to have functions with variable number of arguments.
  - Such functions are called **variadic** functions. E.g. printf, scanf and etc.
  - Such functions must have at least **one mandatory** argument
  - The types of the optional arguments can also vary
- To declare a function with variable number of arguments, **…** operator is used.
  - void func(int x, ...); // To get access to the optional arguments, *macros in stdarg.h* can be used
  - **We should avoid using variadic functions**
- A function can call itself, directly or indirectly.
  - Such a function is called a recursive function.
  - E.g. int factorial(int n) { return (n == 0) ? 1 : (n * factorial(n-1)); }
    - Factorial of n = n! = 1 * 2 * 3 * 4 … n = n*(n - 1)!

# Functions in C

- We shall avoid using recursive functions
- In C, it is possible to have inline functions
  - E.g. static inline int max(int x, int y) { return (x > y) ? x : y; }
  - An inline function shall be declared with the static storage class
  - During compilation the machine codes of inline functions are inserted where the functions are called. Unlike calls to function-like macros which are replaced during preprocessing.
  - Inline functions improve the performance and usually used for small blocks of code
  - The keyword inline is a request to the compiler and the compiler does not garaunti it.
    - For example recursive functions are not compiled as inline
  - inline functions are preferred to function-like macros
- A function with non-void return type shall have an explicit return statement with an expression

# Functions in C

- The address of an object with auto storage class shall not be copied to another object that persists after the auto object does not exist
  - The local variable gets invalid after returning from the function

```
int *func(void)
{
    int local = 0;

    return &local;
}
```

- Functions and objects should not be defined with external linkage if they are referenced in only one file. We should use static storage class
- All declarations of an object or function shall use the same names and type qualifiers
  - int div(int **m**, int **n**); and int div(int **n**, int **m**) { return (n / m); } // Not OK! look at the order of n and m
- A function should have a single point of exit at the end
- The value returned by a function having non-void type shall be used or discarded explicitly using void type casting. E.g. (void)printf("Hello World!\n");

# Functions Pointers in C

- Name of a function in C is the address of where the function starts
  - Example: printf("Address of main is %p or %p\n", main, &main);  // We can omit the & operator
- In C we can have pointers to functions.
  - Unlike normal pointers, a function pointer points to code, not data.
  - Function pointers are used to pass a function to another function (a callback function).
  - A function pointer is declared as **return_type (*function_pointer_name)(list_of_param_types);**
    - E.g. *int (*func)(int, int);* is a function pointer which can point to any function has int as its return type and two arguments of type int. **For example: int f(int a, int b); func = &f; // or f. But we can omit &**
  - We can even use typedef to make new types of function pointers
    - E.g. typedef int (*func_t)(int, int); func_t my_func = func;
      - Then we can call the func using the function pointer as **my_func(20, 30);**
  - Like normal pointers, we can have an array of function pointers. E.g. func_t farr[2] = {add, divide};

# Functions Pointers in C

- We can have function pointers as parameters of functions.

```c
typedef void (*func_t)(void);

//Callback function
void funcA(void) { printf("This is function A!\n"); }

//Function using callback
void funcB(func_t fptr)
{
    printf("B is calling...\n");
    fptr(); // callback to A
}

int main(void)
{
    printf("Lets start...\n");
    func_t temp = &funcA; // Even we can omit the & operator

    //Calling a function with a callback function as argument
    funcB(temp);

    return 0;
}
```

```c
#include <stdio.h>
#define ARRAY_SIZE 5

typedef int array_t[5];
typedef void (*func_t)(int);

void func1(int a)
{
    printf("The value is %d\n", a);
}

int main(void)
{
    array_t arr = {1, 2, 3, 4, 5};
    array_t *ptr = &arr;

    printf("Address of main(void) is %p or %p\n", main, &main);

    for (int i = 0; i < ARRAY_SIZE; ++i)
    {
        printf("arr[%d] = %d\n", i, (*ptr)[i]);
    }

    func_t print_integer = NULL;
    print_integer = func1; // &func1
    print_integer(20);     // (*print_integer)(20);

    return 0;
}
```