AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

Functions

19CSE102 Computer Programming

Functions in C Programming

- A function is a **block of statements** that performs a specific task.

Why do we need functions in C?

- To improve the readability of code.
- Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
- Debugging of the code would be easier if you use functions, as errors are easy to be traced.
- Reduces the size of the code, duplicate set of statements are replaced by function calls.

Types of functions

1. **Predefined standard library functions:**

   - Standard library functions are also known as **built-in functions**.
   - Functions such as puts(), gets(), printf(), scanf() etc. are standard library functions.
   - These functions are already defined in header files (files with .h extensions are called header files such as stdio.h).
   - So we just call them whenever there is a need to use them.

2. **User Defined functions:**

   - A function created by user is known as user defined function.

Syntax of a function

```
return_type function_name (argument list)
{
    Set of statements - Block of code
}
```

- **return_type:** Return type can be of any data type such as int, double, char, void, short etc.
- **function_name:** It is advised to have a meaningful name for the functions.
- **argument list:** Argument list contains variables names along with their data types. These arguments are kind of inputs for the function.
- **block of code:** Set of C statements, which will be executed whenever a call will be made to the function.

Example – To create a function to add two integer variables.

- Function will add the two numbers, so it should have some meaningful name like sum, addition, etc.
- If we take the name addition for this function, then the syntax becomes,

```
return_type addition(argument list)
```

- This function addition adds two integer variables. So needs two integer parameters in the function signature.
- The function signature would be –

```
return_type addition(int num1, int num2)
```

- The result of the sum of two integers would be integer only. Hence function should return an integer value.

- Thus the syntax becomes,

```
int addition(int num1, int num2);
```

- Thus the above is the complete function prototype or signature.

How to call a function in C?

```c
#include <stdio.h>
int addition(int num1, int num2)
{
     int sum;
     /* Arguments are used here*/
     sum = num1+num2;

     return sum;
}

int main()
{
     int var1, var2;
     printf("Enter number 1: ");
     scanf("%d",&var1);
     printf("Enter number 2: ");
     scanf("%d",&var2);

     int res = addition(var1, var2);
     printf ("Output: %d", res);

     return 0;
}
```

Output:

```
Enter number 1: 100
Enter number 2: 120
Output: 220
```

Creating a void user defined function that doesn’t return anything

```c
#include <stdio.h>
/* function return type is void and it doesn't have parameters*/
void introduction()
{
    printf("Hi\n");
    printf("My name is Chaitanya\n");
    printf("How are you?");
    /* There is no return statement inside this function, since its
     * return type is void
     */
}

int main()
{
     /*calling function*/
     introduction();
     return 0;
}
```

Output:

```
Hi
My name is Chaitanya
How are you?
```

Few Points to Note regarding functions in C:

1) main() in C program is also a function.
2) Each C program must have at least one function, which is main().
3) There is no limit on number of functions; A C program can have any number of functions.
4) A function can call itself and it is known as "**Recursion**".

C Functions Terminologies

return type: Data type of returned value. It can be void also, in such case function doesn't return any value.

Actual parameters: The parameters that appear in function calls.

Formal parameters: The parameters that appear in function declarations.

For example: If we have a function declaration like this:

```
int sum(int a, int b);
```

If we are calling the function like this:

```
int s = sum(10, 20); //Here 10 and 20 are actual parameters
or
int s = sum(n1, n2); //Here n1 and n2 are actual parameters
```

Note: Example: If function return type is **char,** then function should return a value of char type and while calling this function the main() function should have a variable of char data type to store the returned value.

Structure would look like –

```
char abc(char ch1, char ch2)
{
   char ch3;
   …
   …
   return ch3;
}

int main()
{
   …
   char c1 = abc('a', 'x');
   …
}
```

More Topics on Functions in C

1) **Function – Call by value method** – In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

2) **Function – Call by reference method** – Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

Function call by value in C programming

Function call by value is the default way of calling a function in C programming.

Actual parameters: The parameters that appear in function calls.

Formal parameters: The parameters that appear in function declarations.

```c
#include <stdio.h>
int sum(int a, int b)
{
     int c=a+b;
     return c;
}

int main(
{
    int var1 =10;
    int var2 = 20;
    int var3 = sum(var1, var2);
    printf("%d", var3);

    return 0;
}
```

- In the above example variable a and b are the formal parameters (or formal arguments).
- Variable var1 and var2 are the actual arguments (or actual parameters).
- The actual parameters can also be the values.
- When we pass the actual parameters while calling a function then this is known as function call by value.
- In this case the values of actual parameters are copied to the formal parameters.

Another Example

```c
#include <stdio.h>
void swapnum( int var1, int var2 )
{
   int tempnum ;
   /*Copying var1 value into temporary variable */
   tempnum = var1 ;

   /* Copying var2 value into var1*/
   var1 = var2 ;

   /*Copying temporary variable value into var2 */
   var2 = tempnum ;

}
int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);

    /*calling swap function*/
    swapnum(num1, num2);
    printf("\nAfter swapping: %d, %d", num1, num2);
}
```

Output:

```
Before swapping: 35, 45
After swapping: 35, 45
```

Why variables remain unchanged even after the swap?

The reason is same – function is called by value for num1 & num2. So actually var1 and var2 gets swapped (not num1 & num2). As in call by value actual parameters are just copied into the formal parameters.

Function call by reference in C Programming

- Calling a function by passing the addresses of actual parameters.
- The operation performed on formal parameters, affects the value of actual parameters.
- This is because all the operations are performed on the value stored in the address of actual parameters.

1. Example of Function call by Reference

```c
#include <stdio.h>
void increment(int  *var)
{
    *var = *var+1;
}
int main()
{
     int num=20;
     increment(&num);
     printf("Value of num is: %d", num);
   return 0;
}
```

Output:

```
Value of num is: 21
```

Example 2: Function Call by Reference – Swapping numbers

```
#include
void swapnum ( int *var1, int *var2 )
{
   int tempnum ;
   tempnum = *var1 ;
   *var1 = *var2 ;
   *var2 = tempnum ;
}
int main( )
{
   int num1 = 35, num2 = 45 ;
   printf("Before swapping:");
   printf("\nnum1 value is %d", num1);
   printf("\nnum2 value is %d", num2);

   /*calling swap function*/
   swapnum( &num1, &num2 );

   printf("\nAfter swapping:");
   printf("\nnum1 value is %d", num1);
   printf("\nnum2 value is %d", num2);
   return 0;
}
```

Output:

```
Before swapping:
num1 value is 35
num2 value is 45
After swapping:
num1 value is 45
num2 value is 35
```

Passing array to function in C programming

1. Passing array to function using call by value method

```c
#include <stdio.h>
void disp( char ch)
{
   printf("%c ", ch);
}
int main()
{
   char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
   for (int x=0; x<10; x++)
   {
       /* I'm passing each element one by one using subscript*/
       disp (arr[x]);
   }

   return 0;
}
```

Output:

```
a b c d e f g h i j
```

2. Passing array to function using call by reference.

```c
#include <stdio.h>
void disp( int *num)
{
    printf("%d ", *num);
}

int main()
{
     int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
     for (int i=0; i<10; i++)
     {
         /* Passing addresses of array elements*/
         disp (&arr[i]);
     }

     return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 0
```

How to pass an entire array to a function as an argument?

- We can also pass an entire array to a function like this:

```c
#include <stdio.h>
void myfuncn( int *var1, int var2)
{
    for(int x=0; x<var2; x++)
    {
        printf("Value of var_arr[%d] is: %d \n", x, *var1);
        /*increment pointer for next element fetch*/
        var1++;
    }
}

int main()
{
     int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
     myfuncn(var_arr, 7);
     return 0;
}
```

Output:

```
Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77
```

C – Scope Rules

- A scope is a region of the program where a defined variable can have its existence.
- There are two places where variables can be declared in C programming language.
  - Inside a function or a block which is called local variables.
  - Outside of all functions which is called global variables.

Local Variables

- Variables that are declared inside a function or block are called local variables.
- They can be used only by statements that are inside that function or block of code.

Local Scope: Example.

```c
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program.
- They can be accessed inside any of the functions defined for the program.

```c
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

- A program can have same name for local and global variables but the value of local variable inside a function will take preference.
- Example:

```c
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

```
value of g = 10
```

Initializing Local and Global Variables

- When a local variable is defined, it is not initialized by the system, you must initialize it yourself.
- Global variables are initialized automatically by the system when you define them, as follows:

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

Storage Classes in C

- Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable.

- There are four types of storage classes in C:

  i.Automatic.

  ii.External.

  iii.Static.

  iv.Register.

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

```c
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
return 0;
}
```

```
garbage garbage garbage
```

Example 2

```c
#include <stdio.h>
int main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
}
}
printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

Output:

```
11 20 20 20 11
```

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

```c
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
}
```

Output:

```
0 0 0.000000 (null)
```

Example 2:

```
#include<stdio.h>
void sum()
{
static int a = 10;
static int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i< 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

Output:

```
10 24
11 25
12 26
```

Register

- The variables defined as the register is allocated the memory in the CPU registers, depending upon the size of the memory remaining in the CPU.
- It is compiler's choice whether or not the variables can be stored in the register.
- We cannot dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

```c
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
printf("%d",a);
}
```

Output:

```
0
```

Example 2

```c
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
}
```

Output:

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~
```

External

- Difference between declaration and definition:
  i. **Declaration** of a variable or function simply declares that the variable or function exists somewhere in the program, but the memory is not allocated for them.

  ii. Coming to the **definition**, when we *define* a variable or function, in addition to everything that a declaration does, it also allocates memory for that variable or function.

  iii. A variable or function can be *declared* any number of times, but it can be *defined* only once.

- How would you *declare* a variable without *defining* it? You would do like this:
  extern int var;
- Here, an integer type variable called var has been declared.
- It hasn't been defined yet, so no memory allocation for var so far. And we can do this declaration as many times as we want.

- Now, how would you *define* var? You would do this:

  int var;
- In this line, an integer type variable called var has been both declared **and** defined.
- Since this is a definition, the memory for var is also allocated.
- We need to include the extern keyword explicitly when we want to declare variables without defining them.
- Also, the extern keyword extends the visibility to the whole program.
- By using the extern keyword with a variable, we can use the variable anywhere in the program provided we include its declaration.
- Extern keyword is generally used to share variables across source files.

Example 1:

```
int var;
int main(void)
{
   var = 10;
   return 0;
}
```

This program compiles successfully. var is defined (and declared implicitly) globally.

Example 2:

```
extern int var;
int main(void)
{
  return 0;
}
```

Here var is declared only. Notice var is never used so no problems arise.

Example 3:

```
extern int var;
int main(void)
{
  var = 10;
  return 0;
}
```

- This program throws an error in the compilation(during the linking phase) because var is declared but not defined anywhere.

Example 4:

```
#include "somefile.h"
extern int var;
int main(void)
{
 var = 10;
 return 0;
}
```

Assuming that somefile.h contains the definition of var, this program will compile successfully.