

Functions

HoangND1

- **Explain the use of functions**
- **Explain the structure of a function**
- **Explain function declaration and function prototypes**
- **Explain the different types of variables**
- **Explain how to call functions**
- **Call by Value**
- **Call by Reference**
- **Explain the scope rules for a function**
- **Explain functions in multi-file programs**
- **Explain Storage classes**
- **Explain function pointers**

- A function is a self-contained program segment that carries out a specific, well-defined task
- Functions are generally used as abbreviations for a series of instructions that are to be executed more than once
- Functions are easy to write and understand
- Debugging the program becomes easier as the structure of the program is more apparent, due to its modular form
- Programs containing functions are also easier to maintain, because modifications, if required, are confined to certain functions within the program

The Function Structure

- The general syntax of a function in C is :

```
type_specifier function_name (arguments)
{
    body of the function
}
```

- The `type_specifier` specifies the data type of the value, which the function will return.
- A valid function name is to be assigned to identify the function
- Arguments appearing in parentheses are also termed as formal parameters.

Arguments of a function

```
#include <stdio.h>
main()
{
    int i;
    for (i =1; i <=10; i++)
        printf ("\nSquare of %d is %d ", i, squarer (i));
}

squarer (int x)
/* int x; */
{
    int j;
    j = x * x;
    return (j);
}
```

Diagram illustrating the flow of arguments:

- A red arrow points from the `(i)` in the `printf` statement to the `(int x)` in the `squarer` function definition.
- A green arrow points from the `(i)` in the `printf` statement to the text "Actual Arguments".
- A horizontal arrow points from the `(int x)` in the `squarer` function definition to the text "Formal Arguments".

- The program calculates the square of numbers from 1 to 10
- The function works on data using arguments
- The data is passed from the `main()` to the `squarer()` function

Returning from the function

```
squarer (int x)
/* int x; */
{
    int j;
    j = x * x;
    return (j);
}
```

- It transfers the control from the function back to the calling program immediately.
- Whatever is inside the parentheses following the return statement is returned as a value to the calling program.

Data Type of a Function

```
type_specifier function_name (arguments)
{
    body of the function
}
```

- The type_specifier is not written prior to the function squarer(), because squarer() returns an integer type value
- The type_specifier is not compulsory if an integer type of value is returned or if no value is returned
- However, to avoid inconsistencies, a data type should be specified

Invoking a Function

- A semicolon is used at the end of the statement when a function is called, but not after the function definition
- Parentheses are compulsory after the function name, irrespective of whether the function has arguments or not
- Only one value can be returned by a function
- The program can have more than one function
- The function that calls another function is known as the calling function/routine
- The function being called is known as the called function/routine

Function Declaration

- Declaring a function becomes compulsory when the function is being used before its definition
- The `address()` function is called before it is defined
- Some C compilers return an error, if the function is not declared before calling
- This is sometimes referred to as Implicit declaration

```
#include <stdio.h>
main()
{
    .
    .
    address()
    .
}

address()
{
    .
    .
    .
}
```

- ❑ Specifies the data types of the arguments

```
char abc(int x, nt y);
```

Advantage :

Any illegal type conversions between the arguments used to call a function and the type definition of its parameters is reported

```
char noparam (void);
```

❑ Local Variables

- ✓ Declared inside a function
- ✓ Created upon entry into a block and destroyed upon exit from the block

❑ Formal Parameters

- ✓ Declared in the definition of function as parameters
- ✓ Act like any local variable inside a function

❑ Global Variables

- ✓ Declared outside all functions
- ✓ Holds value throughout the execution of the program

- Every C variable has a characteristic called as a storage class
- The storage class defines two characteristics of the variable:
 - **Lifetime** – The lifetime of a variable is the length of time it retains a particular value
 - **Visibility** – The visibility of a variable defines the parts of a program that will be able to recognize the variable

- ❑ **automatic**
- ❑ **external**
- ❑ **static**
- ❑ **register**

Function Scope rules

- ❑ Scope Rules - Rules that govern whether one piece of code knows about or has access to another piece of code or data
- ❑ The code within a function is private or local to that function
- ❑ Two functions have different scopes
- ❑ Two Functions are at the same scope level
- ❑ One function cannot be defined within another function

Calling The Functions

- Call by value
- Call by reference

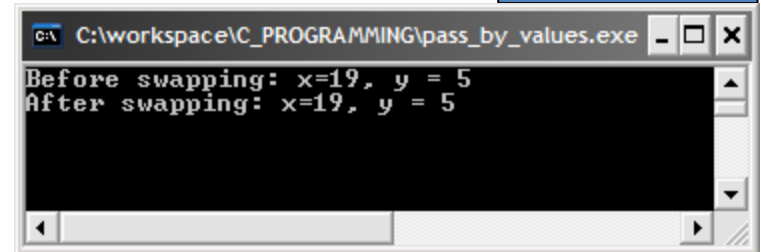
Calling By Value

- In C, by default, all function arguments are passed by value
- When arguments are passed to the called function, the values are passed through temporary variables
- All manipulations are done on these temporary variables only
- The arguments are said to be passed by value when the value of the variable are passed to the called function and any alteration on this value has no effect on the original value of the passed variable

Calling By Value - Example

```
/* Pass-by-Value example */
#include <stdio.h>
int swap (int a, int b);
int main ()
{
    int x = 19, y = 5;
    printf("Before swapping: x=%d, y = %d\n",x,y);
    swap(x, y);
    printf("After swapping: x=%d, y = %d",x,y);
    return 0;
}
int swap (int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Output



C:\workspace\C_PROGRAMMING\pass_by_values.exe

```
Before swapping: x=19, y = 5
After swapping: x=19, y = 5
```

Calling By Reference

- In call by reference, the function is allowed access to the actual memory location of the argument and therefore can change the value of the arguments of the calling routine

- Definition

```
getstr(char *ptr_str, int *ptr_int);
```

- Call

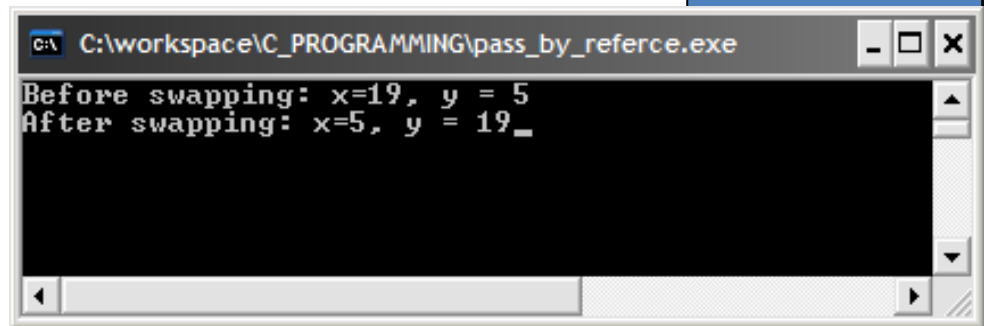
```
getstr(ptr, &var);
```

Calling By Reference - Example

```
/* Pass-by-Reference example */
#include <stdio.h>
int swap (int *a, int *b);
int main ()
{
    int x = 19, y = 5;
    printf("Before swapping: x=%d, y = %d\n",x,y);
    swap(&x, &y);
    printf("After swapping: x=%d, y = %d",x,y);
    return 0;
}

int swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Output



C:\workspace\C_PROGRAMMING\pass_by_referce.exe

```
Before swapping: x=19, y = 5
After swapping: x=5, y = 19_
```

Nesting Function Calls

```
main()  
{  
    .  
    .  
    palindrome();  
    .  
    .  
}
```

```
palindrome()  
{  
    .  
    .  
    getstr();  
    reverse();  
    cmp();  
    .  
    .  
}
```

Functions in Multifile Programs

- Functions can also be defined as **static** or **external**
- Static functions are recognized only within the program file and their scope does not extend outside the program file

```
static fn _type fn_name (argument list);
```

- External function are recognized through all the files of the program

```
extern fn_type fn_name (argument list);
```