Functions

Function in general is nothing but a specific task. In C programming functions are defined as the set of instructions which are named under a specific name and do specific operations. There can be any number of user defined functions written apart from the main function. Main function has to be written only once and that will be called by the operating system. Other user defined functions can be called anywhere according to the requirements. Some of the built-in functions are printf, scanf, etc.

If we have to be writing a function to perform some task, the function definition should first have the data type of the value it might return which is called the return type followed by the name of the function and then the input types along with the names of the variables called the formal arguments.

Syntax: return_type func_name(Formal argument type name1, …)

{

}

Eg: write a function to add two numbers.

The function will return an integer, it might take two inputs x and y. The function definition is as follows.

```
int add(int x, int y)
{
        int res;
        res = x + y;
        return res;
}
```

To get this function executed, the function needs to be called somewhere in the program as shown below.

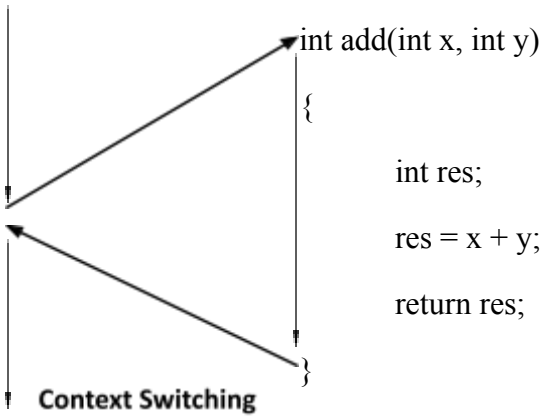
```
int main()
{
        int num1 = 10,  num2 = 20, sum;
        sum = add(num1, num2);
        printf("Sum = %d\n", sum);
```

```
        return 0;
    }
```

With respect to the add function here, the main function is called the caller function and the add function is called the called function.

Whenever a function call is done, control moves from function call to function definition, gets all the instructions executed and comes back to the place where the function was called. This is called context switching.

```
int main()
{
    int num1 = 10,  num2 = 20, sum;
    sum = add(num1, num2);
    printf("Sum = %d\n", sum);
    return 0;
}
```



Advantages of writing a function are,

*To avoid repetitions i.e. to reuse the code whenever necessary.

*To achieve divide and conquer methodology.

*To achieve modularity and abstraction

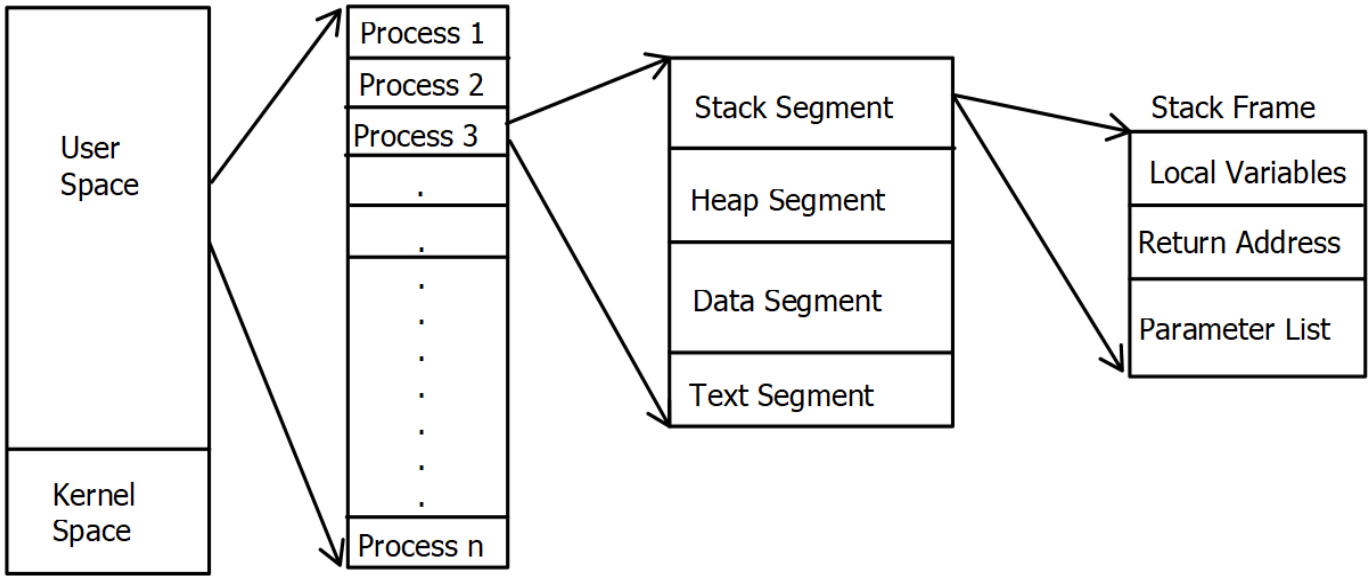
*For easy debugging and testing

The major changes when a function is called are seen in the memory segment. The memory in any operating system is divided into 2 sections, the user space and kernel space. Any normal user cannot access the kernel space, doing so will lead to segmentation fault. All the normal users like us can only access user space. There can be 'n' number of processes being executed and each of then gets its turn for execution depending on the scheduler in the OS. If ./a.out process is getting executed, it gets certain bytes of memory allocated which is further divided into 4 segments. They are text segment, data segment, heap segment and stack segment.

Whenever a function is called, a stack frame gets created in the stack segment. Stack frame has 3 sections, local variables, return address and parameter list. Any variables which are declared or defined inside a function without any storage class gets the memory allocated in the local variables section. Return address section has the address of the caller function which helps the called function to get back to the place where it was called. Parameter list is that section

where the formal arguments i.e. the arguments which are written in ( ) gets the memory allocated.

Any stack frame created will at least have the return address in it. Once the statements of the function are executed and control is brought back to the place where it was called, the stack frame gets destroyed.

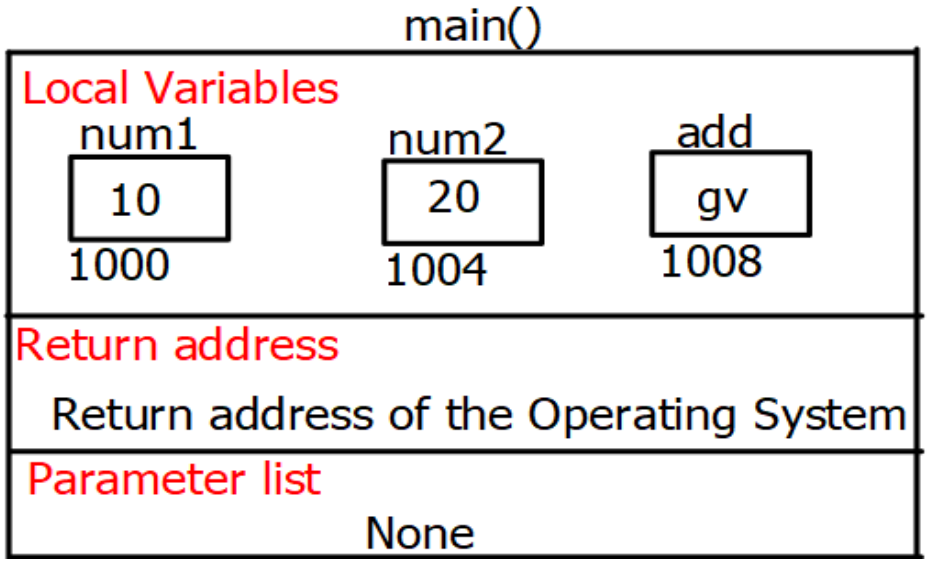


Consider an example shown below.

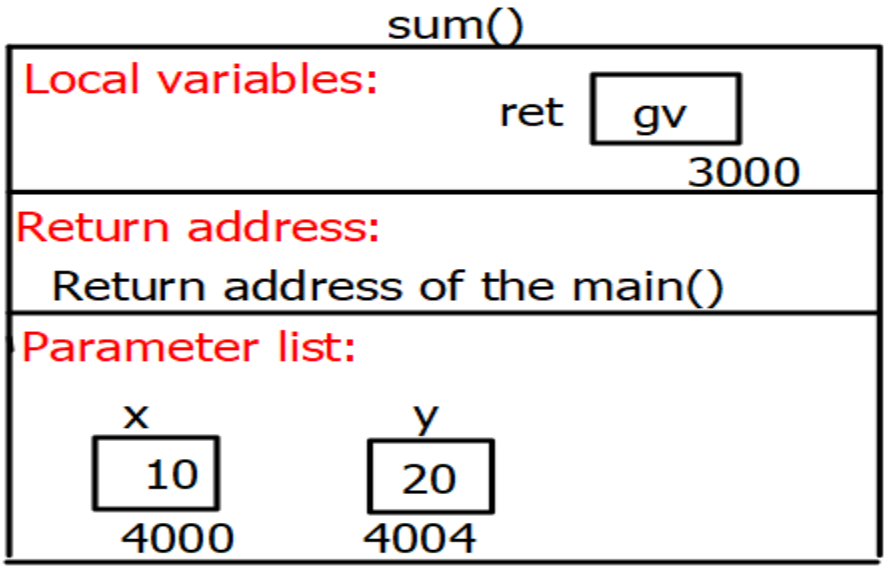
```
int sum(int x, int y)
{
        int ret;
        ret = x + y;
        return ret;
}
int main()
{
        int num1 = 10, num2 = 20, add;
        add = sum(num1, num2);
        printf("sum of %d and %d is %d\n", num1, num2, sum);
```

```
    return 0;
}
```

In the above given example, num1 and num2 in the main function are the actual arguments and x and y in the sum function are formal arguments. First after compiling the program, when the process gets to execute, the OS calls the main function. The moment main function is called, the corresponding stack frame gets created. Num1, num2 and add are the 3 local variables, return address will be the address of OS and there is no parameter list in this function.



After the stack frame is created, the instructions of the function start to execute. During the function call to the sum function, the values of num1 and num2 are passed which are then collected by the formal arguments x and y. Because the function is called, now a stack frame gets created for the function with ret in the local variable, return address of main function in return address and x, y in the parameter list.



Now the statements of the sum function are executed and ret get updated to 30. This value will be returned to the place where the sum function was called and the stack frame of the function gets destroyed. Once all the statements of the main function are executed, the stack frame of the main function also gets destroyed.

In the above example, the function call is done by passing the values of variables. This method of function call is called pass by value. There are some of the disadvantages of this method for which we need to do pass by reference method of function call.

Disadvantages of pass by value:

*Returning is compulsory to see the change done in the function.

*Returning more than one value from a function is not possible.

*Returning is compulsory to see the change done in the function.

Eg: void modify (int x)

```
{
    x = x + 1;
}
int main()
{
    int x= 10;
```

```
    printf("Before modification, x = %d\n", x);
    modify (x);
    printf("After  modification, x = %d\n", x);
    return 0;
}
```

In the above example, x in the main function is 10 which is passed to the modify function. In the modify function, it is received by x and is changed to 11. After coming back to the main function, when x is printed, it still shows 10. This is because the change which was done in the modify function is to the copy of the variable which needs to be returned from the function. As the return was not done from the modify function. we do not get to see the change in the main function. Note that x in the main function and modify function are different. In the main function, it is a local variable and in the modify function, it is in the parameter list. Both are in different stack frames.

*Returning more than one value from a function is not possible.

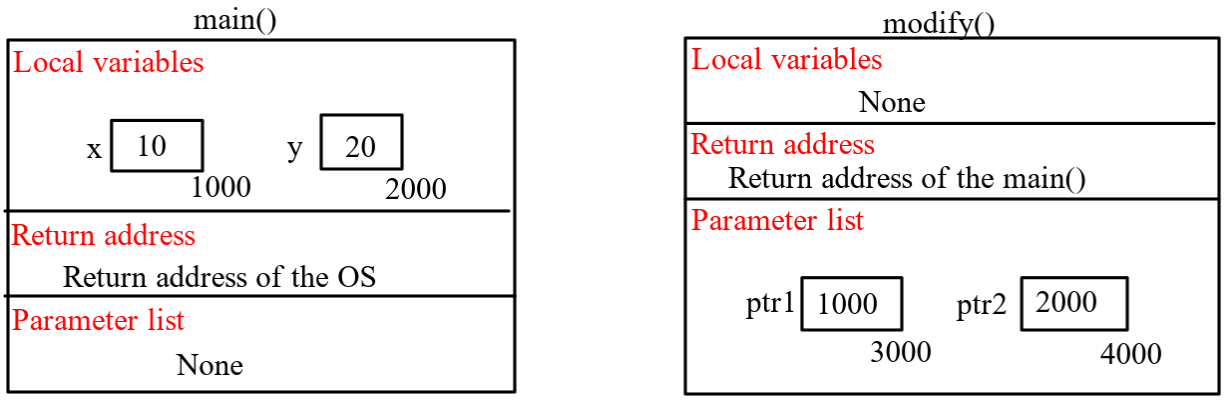
```
    int modify(int x, int y)
    {
        x = x + 1;
        y = y + 1;
        return x, y;
    }
    int main()
    {
        int x = 10, y = 20;
        printf("Before modification:  x = %d, y = %d\n", x, y);
        x , y = modify (x, y);
        printf("After  modification: x = %d, y = %d\n", x, y);
        return 0;
    }
```

In the above example, the modify function is trying to return more than one value from the function. But return x, y or return (x, y) both will return the value of y. In the main function,

the value is received by y only . In the statement x, y = modify (x, y); it will be seen by the compiler as x, (y = modify(x, y)); Hence the value will be received by y only.

In order to see the change in more than one value from a function, we need to go with the pass by reference method of function call.

```
int modify(int *ptr1, int *ptr2)
{
        *ptr1 = *ptr1 + 1;
        *ptr2 = *ptr2 + 1;
}
int main()
{
        int x = 10, y = 20;
        printf("Before modification:  x = %d, y = %d\n", x, y);
         modify (&x, &y);
        printf("After  modification: x = %d, y = %d\n", x, y);
        return 0;
}
```



In the above example, when the modify function is called, the addresses of x and y are passed. Hence any change done using ptr will directly affect x and y.

Whenever the user defined functions are written after the main function, the compiler either throws a warning or an error. The compiler should always be made aware of the function's existence before the function call. This is done either by writing the function's prototype or by writing the function definition before the function call. The discrepancy is created because of the implicit int rule. If the compiler is not made aware of the function's existence, it assumes that the function takes inputs of type integer and returns the output of type integer. If the user defined function's definition matched with what compiler has assumed, it throws a warning saying implicit declaration ppf the function else if the function is accepting inputs of some other type or returning an output of some other type which happens to be a mismatch of what compiler has assumed, it throws an error.

Eg:

```
int main()
{
        int num;
        num = fun();
        printf("%d\n", num);
        return 0;
}
int fun()
{
        int x;
        return x + 10;
}
```

In the above example, the compiler will throw a warning as it is not aware of the function's existence before the function call. Compiler will assume that the function takes integer inputs and returns an integer. To avoid the warning, it is necessary to have the function prototype written.

Syntax of writing prototype is:

return_type func_name(input arguments type);

In the above example, function's prototype should look as follows,

int fun();

If a function is taking two arguments of type int and returning an output of type int, the prototype should be as follows:

```
int sum(int, int);
```

or

```
int sum(int x, int y);
```

Note that the ; termination is compulsory, arguments names are optional and can be anything.

Necessity of writing function's prototype are:

*to make the compile aware of function's existence

*to make the compiler know the number of inputs and its type.

*to make the compiler know the output type