C Functions

Function Basics

Functions:

- Functions are used to divide a large C program into smaller and less complex pieces.
- A function can be called multiple or several times to provide reusability and modularity to the C program.
- Functions are also called procedures, subroutines, or methods.
- A function is also a piece of code that performs a specific task.

A function is nothing but a group of code put together and given a name, and it can be called anytime without writing the whole code again and again in a program.

I know its syntax is a bit difficult to understand, but don't worry; after reading this whole information about Functions, you will know each and every term or thing related to Functions.

Advantages of Functions

- The use of functions allows us to avoid re-writing the same logic or code over and over again.
- With the help of functions, we can divide the work among the programmers.
- We can easily debug or find bugs in any program using functions.
- They make code readable and less complex.

Aspects of a Function

1. **Declaration**

This is where a function is declared to tell the compiler about its existence.

2. **Definition**

A function is defined to get some task executed. (It means when we define a function, we write the whole code of that function, and this is where the actual implementation of the function is done.)

3. **Call**

This is where a function is called in order to be used.

Types of Functions

1. **Library functions:**

Library functions are pre-defined functions in C Language. These are the functions that are included in C header files prior to any other part of the code in order to be used.

E.g. printf(), scanf(), etc.

2. **User-defined functions**

User-defined functions are functions created by the programmer for the reduction of the complexity of a program. Rather, these are functions that the user creates as per the requirements of a program.

E.g. Any function created by the programmer.

Function Parameters

A parameter or an argument of a function is the information we wish to pass to the function when it is called to be executed. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

Here's the basic syntax of a function

return_type functionName(parameter1, parameter2) { // body of the function}

What is the return_type?

Return type specifies the data type the function should return after its execution. It could also be a void where the function is required to return nothing.

Different ways to define a function

1. **Without arguments and without return value**

In this function, we don't have to give any value to the function (argument/parameters) and even don't have to take any value from it in return.

One example of such functions could be:

```
#include <stdio.h>


void func() {
   printf("This function doesn't return anything.");
}
```

```
int main() {
  func();
}
```

Output:

This function doesn't return anything.

2. **Without arguments and with the return value.**

In these types of functions, we don't have to give any value or argument to the function but in return, we get something from it i.e. some value.

One example of such functions could be:

```
#include <stdio.h>

int func() {
  int a, b;
  scanf("%d", &a);
  scanf("%d", &b);
  return a + b;
}

int main() {
  int ans = func();
  printf("%d", ans);
}
```

Input:

5

6

Output:

11

The function func when called, asks for two integer inputs and returns the sum of them. The function's return type is an integer.

3. **With arguments and without a return value.**

In this type of function, we give arguments or parameters to the function but we don't get any value from it in return.

One example of such functions could be:

```
#include <stdio.h>


void func(int a, int b) {
  printf("%d", a * b);
}


int main() {
  func(2, 3);
}
```

Output:

6

The function's return type is void. And it takes two integers as its parameters and prints the product of them while returning nothing.

4. **With arguments and with the return value**

In these functions, we give arguments to a function and in return, we also get some value from it.

One example of such functions could be:

```
#include <stdio.h>


int func(int a, int b) {
  return a + b;
}
```

```
int main() {
  int ans = func(2, 3);
  printf("%d", ans);
}
```

Output:

5

The function func takes two integers as its parameters and returns an integer which is their sum.

Make sure the return statement and the return_type of the functions are the same.

Functions Declaration

A function consists of two parts:

- Declaration, where we specify the function's name, its return type, and required parameters (if any).
- Definition, which is nothing but the body of the function (code to be executed).

The basic structure of a function is:

return_type functionName() { // declaration // body of the function (definition)}

For code optimization and readability, it is often recommended to separate the declaration and the definition of the function.

And this is the reason why you will often see C programs that have function declarations above the main() function, and function definitions below the main() function. This will make the code better organized and easier to read.

This is how it is done:

```
#include <stdio.h>

// Function declaration
void func();

int main()
{
  func(); // calling the function
```

```
    return 0;
}

// Function definition
void func()
{
    printf("This is the definition part.");
}
```

Output:

This is the definition part.

Recursive Functions

What are recursive functions?

Recursive functions or recursion is a process when a function calls a copy of itself to work on smaller problems.

Recursion is the process in which a function calls itself directly or indirectly. The corresponding function which calls itself is called a recursive function.

- Any function which calls itself is called a recursive function.
- This makes the life of a programmer easy by dividing a complex problem into simple or easier problems.
- A termination condition is imposed on such functions to stop them from executing copies of themselves forever or infinitely. This is also known as the base condition.
- Any problem which can be solved recursively can also be solved iteratively.
- Recursions are used to solve tougher problems like Tower Of Hanoi, Fibonacci Series, Factorial finding, etc., and many more, where solving by intuition becomes tough.

What is a base condition?

The case in which the function doesn't recur is called the base case.

For example, when we try to find the factorial of a number using recursion, the case when our number becomes smaller than 1 is the base case.

Recursive Case

The instances where the function keeps calling itself to perform a subtask, which is generally the same problem with the problem size reduced to many smaller parts, is called the recursive case.

Example of recursion:

```
#include <stdio.h>


int factorial(int num)
{
   if (num > 0)
   {
      return num * factorial(num - 1);
   }
   else
   {
      return 1; // Corrected to return 1 for factorial base case
   }
}


int main()
{
   int ans = factorial(10);
   printf("%d", ans);
   return 0;
}
```

Output:

3628800