CONTENTS

CHAPTER – 10

FUNCTIONS

INTRODUCTION

Almost all programming languages have some equivalent of the function. You may have met them under the alternative names subroutine or procedure.

Some languages distinguish between functions which return variables and those which don't. C assumes that every function will return a value. If the programmer wants a return value, this is achieved using the return statement. If no return value is required, none should be used when calling the function.

Each function should be limited to performing a single, well-defined task, and the function name should effectively express that task. This facilities abstraction and promotes software reusability.

TYPE SIGNATURE

A ***type signature*** defines the inputs and outputs for a function or method. A **type signature** includes at least the function name and the number of its parameters. In some programming languages, it may also specify the function's return type or the types of its parameters.

POINTERS AND FUNCTIONS

The pointers are very much used in a function declaration. Sometimes only with a pointer a complex function can be easily represented and success. The usage of the pointers in a function definition may be classified into two groups.

1. Call by reference
2. Call by value

Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.

When C passes arguments to functions it passes them by value.

There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. Other languages do this (*e.g.* var parameters in PASCAL). C uses pointers explicitly to do this. Other languages mask the fact that pointers also underpin the implementation of this.

The best way to study this is to look at an example where we must be able to receive changed parameters.

The usual function *call*:

swap (a, b) WON'T WORK.

Pointers provide the solution: *Pass the address of the variables to the functions and access address of function.*

Thus our function call in our program would look like this:

swap (&a, &b)

The Code to swap is fairly straightforward:

```
void swap (int *px, int *py)

{
        int temp;
        temp = *px;
        /* contents of pointer */
        *px = *py;
        *py = temp;
}
```

MODIFYING FUNCTION ARGUMENTS

Some functions work by modifying the values of their arguments. This may be done to pass more than one value back to the calling routine, or because the return value is already being used in some way. C requires special arrangements for arguments whose values will be changed.

You can treat the arguments of a function as variables, however direct manipulation of these arguments won't change the values of the arguments in the calling function. The value passed to the function is a copy of the calling value. This value is stored like a local variable, it disappears on return from the function.

There is a way to change the values of variables declared outside the function. It is done by passing the addresses of variables to the function. These addresses, or pointers, behave a bit like integer types, except that only a limited number of arithmetic operators can be applied to them. They are declared differently to normal types, and we are rarely interested in the value of a pointer