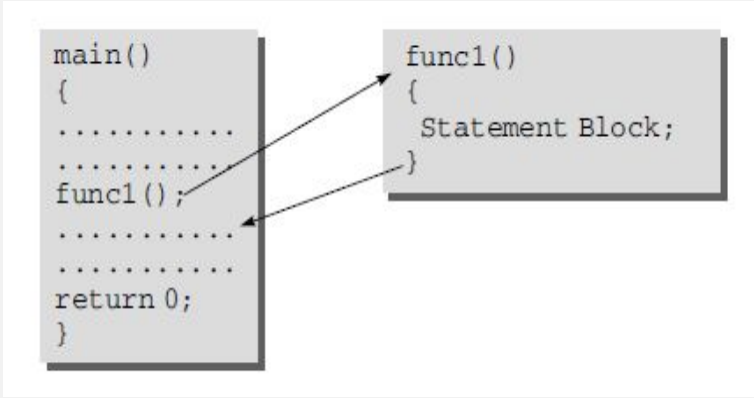
Functions

Chapter 4/7

Introduction

- C enables its programmers to break up a program into segments commonly known as functions, each of which can be written more or less independently of the others.
- Every function in the program is supposed to perform a well defined task. Therefore, the program code of one function is completely insulated from that of other functions.
- Every function has a name which acts as an interface to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it.



- **In the fig, main() calls another function, func1() to perform a well defined task.**

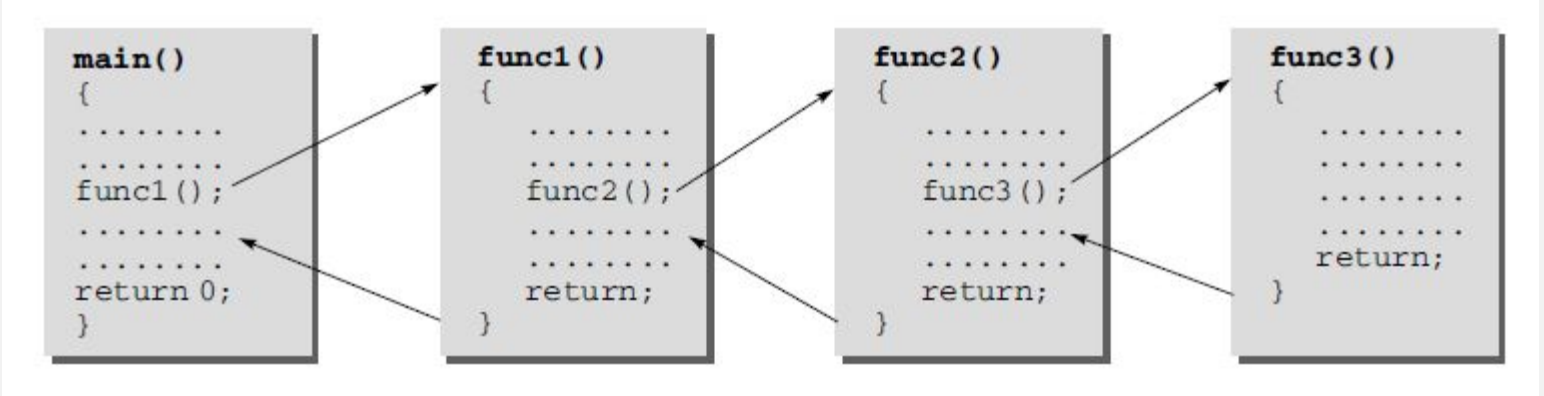
Introduction

- main() is known as the calling function and func1() is known as the called function.
- When the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function.
- After the called function is executed, the control

  is returned back to the calling program.

Introduction

- It is not necessary that the main() can call only one function, it can call as many functions as it wants and as many times as it wants. For example, a function call placed within a for loop, while loop or do-while loop may call the same function multiple times until the condition holds true.
- It is not that only the main() can call another functions. Any function can call any other function. In the fig. one function calls another, and the other function in turn calls some other function.



Why do we need Functions?

- Dividing the program into separate well defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work.
- Understanding, coding and testing multiple separate functions are far easier than doing the same for one huge function.
- If a big program has to be developed without the use of any function (except main()), then there will be countless lines in the main() .
- All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been prewritten and pre-tested, so the programmers use them without worrying about their code details. This speeds up program development.

Terminology of Functions

- A function, *f* that uses another function *g*, is known as the *calling function* and g is known as the *called function*.
- The inputs that the function takes are known as *arguments.*
- When a called function returns some result back to the calling function, it is said to *return* that result.
- The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- Main() is the function that is called by the operating system and therefore, it is supposed to return the result of its processing to the operating system.

Function Declaration

- *Function declaration* is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.
- The general format for declaring a function that accepts some arguments and returns some value as result can be given as:

  return_data_type function_name(data_type variable1, data_type variable2,..);

- No function can be declared within the body of another function.

  Example, float avg (int a, int b);

Function Definition

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.
- When a function defined, space is allocated for that function in the memory.
- The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
 ..............
 statements
 ..............
 return( variable);
}
```

- The no. and the order of arguments in the function header must be same as that given in function declaration statement.

Function Call

- The function call statement invokes the function.
- When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function.
- Once the called function is executed, the program control passes back to the calling function.
- Function call statement has the following syntax.

function_name(variable1, variable2, …);

Function Call

Points to remember while calling the function:

- Function name and the number and type of arguments in the function call must be same as that given in the function declaration and function header of the function definition
- Names (and not the types) of variables in function declaration, function call and header of function definition may vary
- Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.
- If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.

```
variable_name = function_name(variable1, variable2, ...);
```

Program that uses Function

```
#include<stdio.h>
int sum(int a, int b);    // FUNCTION DECLARATION
int main()
{
        int num1, num2, total = 0;
        printf("\n Enter the first number : ");
        scanf("%d", &num1);
        printf("\n Enter the second number : ");
        scanf("%d", &num2);
        total = sum(num1, num2);          // FUNCTION CALL
        printf("\n Total = %d", total);
        return 0;
}
```

Program that uses Function

```
// FUNCTION DEFNITION

int sum ( int a, int b)        // FUNCTION HEADER

{                      //  FUNCTION BODY

     return (a + b);

}
```

Return Statement

- The **return** statement is used to terminate the execution of a function and return control to the calling function. When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- **Programming Tip:** It is an error to use a return statement in a function that has void as its return type.
- A **return** statement may or may not return a value to the calling function. The syntax of return statement can be given as

  return <*expression***> ;**

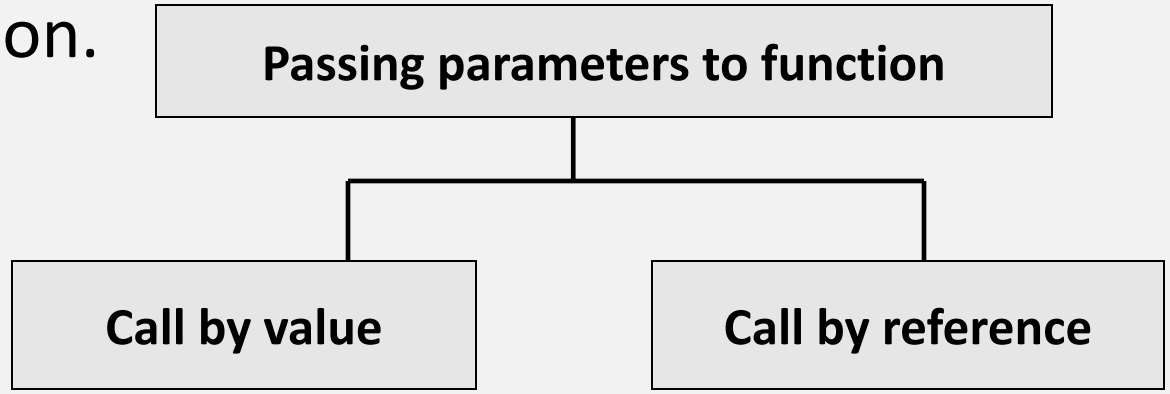
- Here expression is placed in between angular brackets because specifying an expression is optional. The value of *expression*, if present, is returned to the calling function. However, in case *expression* is omitted, the return value of the function is undefined.

Return Statement

- Programmer may or may not place the *expression* within parentheses.
- By default, the return type of a function is *int.*
- For functions that has no **return** statement, the control automatically returns to the calling function after the last statement of the called function is executed.

Passing Parameters to the Function

- There are two ways in which arguments or parameters can be passed to the called function.
- Call by value in which values of the variables are passed by the calling function to the called function.
- Call by reference in which address of the variables are passed by the calling function to the called function.



Call by Value

- In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

```
#include<stdio.h>
void add( int n);
int main()
{
    int num = 2;
    printf("\n The value of num before calling the function = %d", num);
    add(num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
}
```

Call by Value

```
void add(int n)
{
    n = n + 10;
    printf("\n The value of num in the called function = %d", n);
}
```

The output of this program is:

The value of num before calling the function = 2

The value of num in the called function = 12

The value of num after calling the function = 12

Call by Reference

- When the calling function passes arguments to the called function using call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it received are visible by the calling program.
- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to that parameter in the called function body will then be reflected in its value in the calling program.

Program Illustrating Call by Reference Technique

```
#include<stdio.h>
void add( int &n);
int main()
{
    int num = 2;
    printf("\n The value of num before calling the function = %d", num);
    add(num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
}
void add( int &n)
{
    n = n + 10;
    printf("\n The value of num in the called function = %d", n);
}
```

Program Illustrating Call by Reference Technique

- The output of this program is:
- The value of num before calling the function = 2
- The value of num in the called function = 12
- The value of num after calling the function = 12

Built-in-Functions

<ctype.h>	
int isalnum(int c);	// to check if c is an alphabet or a digit
int isalpha(int c);	// to check if c is an alphabet
int iscntrl(int c);	// to check if c is a control character
int isdigit(int c);	// to check if c is a decimal digit
int isgraph(int c);	// to check if c is a printing character other than space
int islower(int c);	// to check if c is a lower-case character
int isprint(int c);	// to check if c is a printing character including space
int ispunct(int c);	// to check if c is a printing character other than space, letter, or digit
int isspace(int c);	// to check is c if a space, formfeed, newline, carriage return, tab, or vertical tab
int isupper(int c);	// to check if c is an upper-case character
int isxdigit(int c);	// to check if c is a hexadecimal digit
int tolower(int c);	//to convert c into its lower-case equivalent
int toupper(int c);	// to convert c into its upper-case equivalent
<errno.h>	
errno	// object to which certain library functions assign specific positive values when an error occurs
EDOM	// code used for domain errors
ERANGE	// code used for range errors
<limits.h>	
CHAR_BIT	// Specifies the number of bits in a char
CHAR_MAX	// Specifies the maximum value of type char

Thank You!