Chapter: 1

Introduction to C

Basics of programming in C

1.1 The components of a C program

- All C programs consist of one or more functions, each of which contains one or more statements.
- In C, a function is a named subroutine that can be called by other parts of the program. Functions are the building blocks of C.
- A statement specifies an action to be performed by the program. All C statements end with a semicolon. One may place two or more statements on one line.

1.1.1 C function

- The general form of a C function is shown here:

```
ret-type function-name(param-list){  statement sequence
                                                    }
```

  Here, ***ret-type*** specifies the type of data returned by the function. It is possible for a function to return a value. The **function-name** is the name of the function. Information can be passed to a function through its parameters, which are specified in the function's parameter list, *param-list*. The statement sequence may be one or more statements. (Technically, a function can contain no statements.).
- with few exceptions, a function can be called by any name. It must be composed of only the upper- and lowercase letters of the alphabet, the digits 0-9, and the underscore. ***A digit cannot start a function name***.
- C is case-sensitive. For example **Myfunc** and **myfunc** are entirely different names.

1.1.2 The main( ) function

- Although a C program may contain several functions, the only function that it must have is **main( )**. The **main( )** function is where execution of a program begins. That is, when a program begins running, it starts executing the statements inside **main( )**, beginning With the First statement after the opening curly brace and ends when **main( )**'s closing curly brace is reached.

1.1.3 The library function

- Another important component of all C programs is ***library functions***. The ANSI C standard specifies a set of library functions to be supplied by all C compilers, which is usually referred to as the C standard library. The standard library contains functions to perform *disk I/O (input/ output), string manipulations, mathematical computations*, and much more. When a program is compiled, the code for each library function used by the program is automatically included. Library functions can be enhanced and expanded as needed to accommodate changing circumstances.
- One of the most common library functions is printf( ). This is C's general-purpose output function. The ***printf()*** function is quite versatile, allowing many variations. Its simplest form is shown here:

```
Printf("string-to-output");
```

  The ***printf()*** function outputs the characters that are contained between the beginning and ending double quotes to the screen. (The double quotes are not displayed on the screen.) In C, one or more characters enclosed between double quotes is called a string. The quoted string between ***printf()*** s parentheses is said to be an argument to ***printf()***. In general, *information passed to a function is called an argument*. In C, calling a library function is a statement; therefore, it must end with a semicolon.
- To call a function, you specify its name followed by a parenthesized list of arguments that you will be passing to it. If the function does not require any arguments, no arguments will be specified—and the parenthesized list will be empty. If there is more than one argument, the arguments must be separated by commas.

```
/* function definition to swap the values */
void swap(int *x, int *y) {    int temp;
                               temp = *x;   /* save the value at address x */
                               *x = *y;     /* put y into x */
                               *y = temp;   /* put temp into y */
                               return; }
```

```
#include <stdio.h>
void swap(int *x, int *y);         /* function declaration */

int main(){   int a = 100;         /* local variable definition */
              int b = 200;

printf("Before swap, value of a : %d  b : %d \n", a, b );
      /*     calling a function to swap the values.
             &a indicates pointer to a   ie. address of variable a and
             &b indicates pointer to b   ie. address of variable b.      */
swap(&a, &b);

printf("After swap, value of a : %d  b : %d \n", a, b);
return 0; }
```

1.1.4 The header file

☞ Another component common to most C programs is the header file. In C, information about the standard library functions is found in various files supplied with compiler. These files all end with a **'.h'** extension. The C compiler uses the information in these files to handle the library functions properly. To add these files to a program ***#include*** is used (pre-processor directive).
(Preprocessor directives are not actually part of the C language, but rather instructions from you to the compiler. The ***#include*** directive tells the preprocessor to read in another file and include it with your program.)
The most commonly required header file is called **STDIO.H**. Here is the directive that includes this file:

```
#include <stdio.h>
```

*(Notice that the **#include** directive does not end with a semicolon. The reason for this is that **#include** is not a C keyword that can define a statement. Instead, it is an instruction to the C compiler itself.)*

One last point: With few exceptions, C ignores spaces. That is, it doesn't care where on a line a statement, curly brace, or function name occurs. If you like, you can even put two or more of these items on the same line. The blank lines separate different parts of the program into logically identifiable components, and the indentation indicates subordinate relationships among the various instructions. These features are not grammatically essential, but their presence is strongly encouraged as a matter of good programming practice.

1.1.5 Structure of a C program

Every C program consists of one or more modules called **functions**. One of the functions must be called **main**. The program will always begin by executing the **main** function, which may access other functions. Any other function definitions must be defined separately, either *ahead of or after* **main** .

Each function must contain:

1. A function *heading*, which consists of the function name, followed by an optional list of *arguments*, enclosed in parentheses.
2. A list of argument *declarations*, if arguments are included in the heading.
3. A *compound statement*, which comprises the remainder of the function.

Arguments : The arguments are symbols that represent information being passed between the function and other parts of the program. (Arguments are also referred to as parameters.)

NOTE :

- Each ***compound statement*** is enclosed within a pair of braces, i.e., **{ }**. The braces may contain one or more elementary statements (called expression statements) and other compound statements. Thus compound statements may be nested, one within another. Each expression statement must end with a semicolon (;).
- ***Comments*** (remarks) may appear anywhere within a program, as long as they are placed within the delimiters/* and */ (e.g., I* this is a comment */). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.
- The empty parentheses following the name of the function **main( )** indicate that this function does not include any arguments.
- The lines of a program are indented and enclosed within a **{}**. These lines comprise the compound statement within **main**.
- The first indented line is usually a variable declaration inside **main**.
- The remaining lines which are not variable declaration are expression statements.
- The assignment statement causes the calculation stuffs inside a function of a program.
- Each expression statement within the compound statement ends with a semicolon.

❒ Example of a C program

```
/* program to calculate the area of a circle */      /* TITLE (COMMENT) */

#include <stdio.h>                                   /* LIBRARY FILE ACCESS */
int main(void)                                       /* FUNCTION HEADING */
{
      float r, area;                                 /* VARIABLE DECLARATIONS */
      printf("Enter the value of radious r= ");      /* OUTPUT STATEMENT (PROMPT) */
      scanf("%f", &r);                               /* INPUT STATEMENT */
      area=3.14159*(r*r);                            /* ASSIGNMENT STATEMENT */
      printf("Radious = %f \n Area = %f",r,area);    /* OUTPUT STATEMENT */
      return 0;
}
```

NOTE :

- ***#include <stdio.h>*** : It causes the file ***STDIO.H*** to be read by the C compiler and to be included with the program This file contains information related to ***printf( ), scanf( )*** and other
- ***int main(void)*** : begins the ***main( )*** function. This is where program execution begins. The int specifies that ***main( )*** returns an integer value. The void tells the compiler that ***main( )*** does not have any parameters.
- After ***main( )*** an opening curly brace marks the beginning of statements that make up the function.
- ***return 0;*** : The last line causes ***main( )*** to return the value zero. In this case, the value is returned to the calling process, which is usually the operating system. *[By convention, a return value of zero from **main( )** indicates normal program termination. Any other value represents an error. The operating system can test this value to determine whether the program ran successfully or experienced an error, return is one of C's keywords .]*
- **File Extension :** Do not give the program files the ".cpp" extension instead of ".c" extension. It may occur error because ".cpp" extension tells compiler to compile it as a C++ program since C is the basis of C++ - not all C programs are valid C program. So a C program must be compiled as a C program not as C++ program.
- **Source(.c), Object(.o), Executable(.exe) :** The file contains the C program called the "Source file". The compiled file is called the "Objet file". The file which is executable contain ".exe" extension is called the "Executable file".

1.2 DECLARE VARIABLES AND ASSIGN VALUES

1.2.1 Variable Types in C

Type	*Keyword*
character data	char
signed whole numbers	int
floating-point numbers	float
double-precision floating-point numbers	double
valueless	void

- ***variable :*** A variable is a named memory location that can hold various values. In C all variables must be declared before they can be used. A variable's declaration tells the compiler what type of variable is being used. C supports five different basic data types, as shown in Table along with the C keywords that represent them.
- ***char type of variable :*** A variable of type char is 8 bits long and is most commonly used to hold a single character. A variable of type char can also be used as a "little integer".
- ***Integer variables :*** Integer variables (int) may hold signed whole numbers (numbers with no frictional part). For 16-bit environments, such as DOS or Windows 3.1, integers are usually 16 bits long and may hold values in the range -32,768 to 32,767. In 32-bit environments, integers are typically 32 bits in length. In this case, they may store values in the range -2,147,483.648 to 2.147,483,647.
- ***Float & Double variables :*** Variables of types float and double hold signed floating-point values, which may have fractional components. One difference between float and double is that double provides about twice the precision (number of significant digits) as does float. Also, for most uses of C, a variable of type double is capable of storing values with absolute magnitudes larger than those stored by variables of type float. Of course, in all cases, variables of types float and double can hold very large values.
- ***General form to declare a variable***

```
type var-name
```

where **type** is a C data type and **var-name** is the name of the variable. For example, this declares **counter** to be of type **int**

```
int counter;
```

- ***void :*** void is a special-purpose data type. The void type has no values. This is usually used to specify the type of functions. The type of a function is said to be void when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.
- ***Glabal and Local variables :*** Variables declared outside all functions are called global variables and they may be accessed by any function in a program. Global variables exist the entire time when the program is executing.
- Variables declared inside a function are called local variables. A local variable is known to—and may be accessed by—only the function in which it is declared. It is common practice to declare all local variables used by a function at the start of the function, after the opening curly brace.
- ***Constants :*** A constant is a fixed value used in your program Constants are often used to initialize variables at the beginning of a program's execution. A character constant is specified by placing the character between single quotes. For example to specify the letter "A" you would use 'A'. Integers are specified as whole numbers. Floating-point values must include a decimal point. For example, to tell the compiler that 100 is a floating-point number, use 100.0

1.2.2 Assignment Statement

The general form of an assignment statement is:

```
variable-name = value;
```

For example, to assign an integer variable named num the value 100, we can use this statement;

```
num = 100;
```

1.2.3 Format Specifiers and printf(), scanf()

- ***printf( ) :*** The operation of the function is as follows. The first argument is a quoted string that may contain either normal characters or format specifiers that begin with the percent '%' sign. Normal characters are simply displayed as-is on the screen in the order in which they are encountered in the string (reading left to right). Eg :

```
printf("This prints the number %d",99);
```

A format specifier, also called a format code , informs printf( ) that a different type item is to be displayed. In this case, the %d means that an integer is to be output in decimal format. The value to be displayed is found in the second argument. This value is then output to the screen at the point where the format specifier is found in the string.

- ***format specifiers :*** format specifiers also called a format code which begin with the percent '%' sign. the value associated with a format code is displayed at the point where that format code is encountered in the string.
To understand the relationship between the normal characters and the format codes, examine this statement:

```
printf("This displays %d, too", 99);
```

Now the call to ***printf( )*** displays **"This displays 99, too"**.

  - **%d** is the format specifier to specify a integer value,. To specify a character value, the format specifier is **%c**. To specify' a floating-point value, use **%f**. The **%f** works for both float and double.

- ***Scanf( ) :*** To use **scanf( )** to read an integer value from the keyboard, call it using the general form

```
Scanf('%d", &int-var-name);
```

where **int-var-name** is the name of the integer variable which receive the value. The first argument to **scanf( )** is a string that determines how the second argument will be treated. In this case, the **%d** specifies that the second argument will be receiving an integer value entered in decimal format. for example,

```
int num;
scanf("%d", &num);
```

reads an integer entered from the keyboard.
The "**&**" preceding the variable name is essential to the operation of **scanf( )**. "**&**" allows a function to place a value into one of its arguments.
To read a floating-point number from the keyboard, call **scanf( )** using the general form

```
scanf("%f", &float-var-name);
```

where **float-var-name** is the name of a variable that is declared as being of type float. To input to a double variable, use the **%lf** specifier.

NOTES :

- In C. a variable declaration is a statement and it must end in a semicolon.
- There are two places where variables are declared: inside a function or outside all functions.
- The local variables in one function have no relationship to the local variables in another function (they may have same name or not).
- local variables are created when a function is called, and they are destroyed when the function is exited. Therefore, local variables do not maintain their values between function calls.
- You can declare more than one variable of the same type by using a comma-separated list. For example, this declares three floating-point variables, x, y, and z:

```
float x, y, z;
```

- A digit may not start a variable name.
- In general, when there is more than one argument to a function, the arguments are separated from each other by commas.

```
printf("This prints the number %d",99);
```

arguments are separated from each other by a comma. Eg: `"This prints the number %d"` and `'99'` are two arguments separated by a comma in `printf().`

- The values matched with the ***format specifier*** need not be constants; they may be variables, too.
- Notice that the format specifiers for scanf( ) are similar to those used for printf( ) for the corresponding data types except that %lf is used to read a double.
- When you enter a number at the keyboard, you are simply typing a string of digits. The **scanf( )** function waits until you have pressed **ENTER** before it converts the string into the internal binary format used by the computer.
- **printf( )** and **scanf ( )** are complementary functions.
- In case j,k are ***int*** type and a,b are ***float*** type: to print the ***float*** result of arithmetic operation of j,k

```
printf("\nThe area of the Ractangle = %f", 1.0*j*k);/*'int' type data to 'float' transform*/
also a*j, a+(j+k) gives float result.
```

Mixing with ***float*** & ***int*** type data cause severe problem. Proper way is : `a+((j+k)*1.0)`
Not using proper specifire may cause problem. `printf("%f", j*k)` doesn't work , proper way is `printf("%f", 1.0*j*k)` [transform int to float]. `printf("%d", a*b);` doesn't works because ***a*** & ***b*** are ***float*** type. There is no direct way. The way is

```
int l; l= a*b; printf("%d", l);
```

1.2.4 Arithmetic Expression

An expression is a combination of operators and operands. C expressions follow the rules of algebra. C defines these five arithmetic operators : "+" as addition, "–" as subtraction, "*" as multiplication, "/" as division, "%" as modulus.

NOTES :

- The +,-,/, and * operators may be used with any of the basic data types.
- The modulus operator % may be used with integer types only. The modulus operator produces the remainder of an integer division. This has no meaning when applied to floating-point types.
- The - has two meanings.
  - First, it is the subtraction operator.
  - Second, it can be used as a unary minus to reverse the sign of a number. A unary operator uses only one operand.
- The *, /, and % are higher in precedence than the + and the - .
- A C expression may contain variables, constants, or both.

1.2.5 Comments

General form of c comment is `/*this is a C comment */`

NOTES:

- Comment can extended over several lines.
- Comments can go anywhere in C program except in the middle of any C keyword, function name , or variable name.
- Comments can not be nested .

1.3 Properties of function and C keywords

Defining functions : The general form of a C program that has multiple functions is shown here:

```
/* include header files here */
/* function prototypes here */

int main(void)
{
/*...*/
}

ret-type f1{param-list)
{
/*...*/
}

ret-type f2(param-lisQ
{
/*...*/
}
.
.
.
ret-type fN(param-list)
{{
/*...*/
}
```

Functions can be called by different names. Here, `ret-type` specifies the type of data returned by the function.

NOTE

- If a function does not return a value, then its return type should be void.
- If a function does not use parameters, then its param-list should contain the keyword void.

function prototype : A function prototype declares a function before it is used and prior to its definition. A prototype consists of a function's name, its return type, and its parameter list.
It is terminated by a semicolon. The compiler needs to know this information in order for it to properly execute a call to the function.
For example, given this simple function:

```
void myfunc(void)
{
printf("This is a test.");
}
```

Its prototype is `void myfunc(void);`

NOTE

- The only function that does not need a prototype is main( ) since it is predefined by the C language.
- Traditionally, main( ) is not called by any other function, but there is no technical restriction to this effect.

Calling a function : When a function is called, execution transfers to that function. When the end of that function is reached, execution returns to a point immediately after the place at which the function was called.
Any function inside a program may call any other function within the same program.

those that that do not return values and do not use parameters. The skeletal form of such a function is shown here:

```
void FuncName(void) {     /* body of function here */
                      }
```

Of course, the name of the function will vary. Because the function does not return a value, its return type is `void`. Because the function does not have parameters, its parameter list is `void`.

Example of calling a function

Example 1

```
/*Program with three functions*/
#include <stdio.h>

void func1(void);
int main(void)
{
    printf("I ");
    func1(); /*Calling the function*/
    printf("C");
    return 0;
}

void func1(void)
{
    printf("Love ");
}
```

This program displays ***I like C***. on the screen. Here is how it works. In ***main( )***, the first call to ***printf( )*** executes, printing the ***I.*** Next, ***funcl()*** is called. This causes the ***printf( )*** inside ***funcl( )*** to execute, displaying ***like***. Since this is the only statement inside ***funcl( ),*** the function returns. This causes execution to resume inside ***main( )*** and the ***C*** is printed.
*(Notice that the statement that calls **funcl( )** ends with a semicolon. (Remember a function call is a statement.)*

Example 2

```
/*Program with three functions*/
#include <stdio.h>

void func1(void);
void func2(void);
int main(void)
{
    /*Calling the function*/
    func2();
    printf(" 3 ");
    return 0;
}
/*function calls another function*/
void func2(void)
{
    func1();
    printf(" 2 ");
}
void func1(void)
{
    printf(" 1 ");
}
```

In this program, ***main( )*** first calls ***func2( )***, which then calls ***funcl( )***. Next, ***funcl( )*** displays ***1*** and then returns to ***func2( ),*** which prints ***2*** and then returns to ***main( ),*** which prints ***3***.

1.4 USE FUNCTIONS TO RETURN VALUES

For your program to obtain the return value, you must put the function on the right side of an assignment statement. For example, this program prints the square root of 10:

```
#include <stdio.h>
#include <math.h>          /*Needed by sqrt() library function*/
int main(void){  double answer;
                 answer=sqrt(10.0);
                 printf("Answer = %f", answer);
                 return 0;}
```

This program calls ***sqrt( )*** and assigns its return value to answer Notice that ***sqrt( )*** uses the ***MATH.H*** header file.

NOTE :

- In C, a function may return a value to the calling routine. For example, another of C's standard library functions is ***sqrt( ),*** which returns the square root of its argument.
- Actually, the assignment statement in the preceding program is not technically necessary because ***sqrt( )*** could simply be used as an argument to ***printf( )***, as shown here:

```
#include <stdio.h>
#include <math.h>            /*Needed by sqrt() library function*/
int main(void){printf("%f", sqrt(10.0));   return 0;}
```

  The reason this works is that C will automatically call ***sqrt( )*** and obtain its return value before calling ***printf( ).*** The return value then becomes the second argument to ***printf( ).***

- The ***sqrt( )*** function requires a *floating-point* value for its argument, and the value it returns is of type *double*. We must match the type of value a function returns with the variable that the value will be assigned to.

Return a value to the calling routine using the return statement : When writing our own functions, we can return a value to the calling routine using the return statement. The return statement takes the general form

```
return value;
```

where ***value*** is the value to be returned. For example, this program prints 10 on the screen:

```
#include <stdio.h>
int func(void); /* prototype */
int main(void)
{
      int num;
      num = func();
      printf ("%d', num) ;
      return 0;
}

int func(void)
{
      return 10;
}
```

In this example, func( ) returns an integer value and its return type is specified as int.

NOTE :

1. Although we can create functions that return any type of data, functions that return values of type ***int*** are quite common.
2. Functions that are declared as ***void*** may not return values.
3. If a function does not explicitly specify a return type, it is assumed to return an ***integer by default.*** For example, func( ) could have been coded like this: `func(void){ return 10; }`
   In this case, the ***int*** is implied. The use of the "default to int" rule is very common in older C code. However, recently there has been a move away from using the integer default.
4. When the return statement is encountered, the ***function returns immediately***. No statements after it will be executed. Thus, a return statement causes a function to return ***before its closing curly brace*** is reached.
5. The value associated with the return statement need not be a constant. It can be ***any valid C expression***.

6. A return statement can also be used by itself, without a return value. This form of return looks like this:

```
return ;
```

It is used mostly by void functions (i.e., functions that have a ***void return type***) to cause the function ***to return immediately***, before the function's closing curly brace is reached.
[While not recommended, you can also use this form of return in functions that are supposed to return values. However, doing so makes the returned value undefined.]

7. There can be more than one return in a function.

[Even though a function returns a value, you don't necessarily have to assign that value to anything. If the return value of a function is not used, it-is lost, but no harm is done.]

1.5 Arguments of functions and their use

Argument : A function's argument is a value that is passed to the function when the function is called. A function in C can have from zero to several arguments. (The upper limit is determined by the compiler you are using, but the ANSI C standard limits up to 31 arguments).

Parameters : For a function to be able to take arguments, special variables to receive argument values must be declared. These are called the formal parameters of the function. The parameters are declared between the parentheses that follow the function's name.

General form of declaring functions with multiple arguments

```
ret-type func_name(type ver1, type ver2, . . . , type ver31);
```

For example, the function listed below prints the sum of the two integer arguments used to call it.

```
void sum(int x, int y){    printf("%d ", x + y);      }
```

Each time sum( ) is called, it will sum the value passed to x with the value passed to y.

Consider the following short program, which illustrates how to cal

```
#include<stdio.h>
/*Function prototype with arguments & parameters*/
void sum(int x, int y);
int main(void)
{
    sum(1,20);
    sum(9,6);
    sum(81,9);
    return 0;
}

void sum(int x, int y)
{
    printf(" sum = %d ", x+y);
}
```

This program will print 21. 15. and 90 on the screen. When ***sum( )*** is called, the value of each argument is copied into its matching parameter. That is, in the first call to ***sum( )*** 1 is copied into x and 20 is copied into y. In the second call, 9 is copied into x and 6 into y In the third call. 81 is copied into x and 9 into y.

Parameterized functions : Argument refers to the value that is passed to a function. Functions that take arguments are called parameterized functions. The variable that receives the value of the argument inside the function is the formal parameter of the function. If a variable is used as an argument to a function, it has nothing to do with the formal parameter that receives its value. i.e. `sum(u,v)` do nothing to **`void`** `sum(int x, int y).`

NOTES :

- In C functions, arguments are always separated by commas.
- You must specify the type and name of each parameter and, for more than one parameter, you must use a comma to separate them.
- Functions that do not have parameters should use the keyword void in their parameter list.

1.6 REMEMBER THE C KEYWORDS

ANSI C standard has 32 keywords that may not be used as variable or function names. These words, combined with the formal C syntax, form the C programming language.

The lowercase lettering of the keywords is significant. C requires that all keywords be in lowercase form. For example, RETURN will not be recognized as the keyword return Also, no keyword may be used as a variable or function name.

auto	***double***	***int***	***struct***
break	***else***	***long***	***switch***
case	***enum***	***register***	***typedef***
char	***extern***	***return***	***union***
const	***float***	***short***	***unsigned***
continue	***for***	***signed***	***void***
default	***goto***	***sizeof***	***volatile***
do	***if***	***static***	***while***