Chapter 1

Supplement No. 1(A): Fundamentals of the C Programming Language

LEARNING OBJECTIVES

The objective of this supplement is to
- present a short summary of basic concepts of C programming language;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs in C language;
- explain how to apply pseudofunction or pseudocodes into functions or subprograms in C programming language.

The textbook focuses on computational engineering applications. Thus, supplementary course materials involving programs in C/C++, Fortran, Visual Basic, Python, Matlab® and Wolfram Mathematica® languages or software are used. This supplement is a reference document for C computer language and illustrates how pseudocode statements can be converted to actual programming languages. It is assumed that the reader is familiar with programming concepts in general and may also be familiar with the C programming language.

The algorithms presented in this book are presented in a form that requires very little time and effort to digest. The motivations for using pseudocodes have been stated in the preface and Chapter 1. The aim of this appendix is to present the basic syntax and constructions with simple and concise examples.

1.1 A C PROGRAM STRUCTURE

A C program consists of preprocessor directives, definitions, global declarations, comments, the `main()` function, and additional (if required) functions.

Every program statement ends in a semi-colon (;), newlines are not significant except in preprocessor controls, and the blank lines are ignored. All function names, including the main program, which is a function (`main()`) is always followed by () brackets. Braces `{ }` contain a group of statements.

1.1.1 Preprocessor Directives

Every C program begins with at least one preprocessor directive. The first thing the compiler processes is the preprocessor directive, which provides control instructions from a code referred to as *header file*. The header files, typically having the "`.h`" extension, are an important part of C programs, which files serve as a mean to import predefined standard library functions, data types, macros, and other features into the main programs. A list of some of the frequently used C libraries and the header files is given in **Table 1.1**. The header files

Table 1.1: Some of the C libraries list and the header files.

Header file	Library functions
`stdio.h`	printf(), scanf(), fgets(), fopen(), fclose(), fprintf(), fscanf(), sprintf(), snprintf(), fseek(), fread(), fwrite()
`stdlib.h`	malloc(), calloc(), realloc(), free(), srand(), rand(), atoi(), atof()
`string.h`	strlen(), strcpy(), strcat(), strcmp(), strstr(), strtok(), memset(), memcpy(), memmove()
`math.h`	sqrt(), sin(), cos(), pow(), ceil(), floor(), abs(), rand(), srand(), etc.
`time.h`	time(), localtime(), strftime(), difftime(), etc.
`stdbool.h`	bool, true, false, etc.

are imported into the code by `#include` preprocessor directive (Click to see all available C libraries).

1.1.2 Global Data Definitions

Global variables are externally defined functions and are accessible by the main main and auxiliary functions. They are stored in the data section of the program's memory.

1.1.3 Function Definitions

Function definitions contain both data definitions and code instructions to be executed when the program runs. All program executable statements are contained within function definitions. Every C program should have one function called `main`.

1.1.4 Commenting

Commenting is done to allow *human-readable* descriptions detailing the purpose of some of the statement(s) and/or to create *in situ* documentation. A double forward slash (//) is used for single-line comments; a series of multi-line comments is enclosed within /* and */.

A *single-line* comment is applied to describe an expression (or statement) on the corresponding line. A *block* of comments generally at the beginning of each module (*Header Comments*) is used to describe the purpose of the module, its variables, exceptions, other modules used, etc.

1.1.5 A sample C Example

The basic features of a C program (`example.c`) is illustrated in **C code 1.1**:

Lines 1-4, 8, 13, and 16 are reserved entirely for comments. Note that in this example explanatory comments are dispersed throughout the program. The other lines also include comments after the end of the statements.

In line 5, the header file (in this case `<stdio.h>`) is executed first. `<stdio.h>` (STandarD Input- Output) is a library file, which provides functions and declarations related to input and output operations. It includes such as `printf()`, `scanf()`, and other functions for communicating the input and output data. Without it, it is impossible to input and output data to a program.

In line 6, a global data (`PI`) is defined: `#define PI 3.14159`. The constant PI becomes accessible by the main and auxiliary functions, if present. This feature is known as *macro definition*.

In lines 9-20, the example code takes place. The `main`, which is a function, is the starting point of the program. It is defined as `int main (void)` or also expressed as `int main ()` or simply `main ()`. This representation indicates that the main program (function) does not take an argument.

In line 10, the type of the local variables `radius` and `area` are identified as `float`.

C Code 1.1

```
1  /* ================================================================
2   * Description : An example C program calculating the area of a circle.
3   * Written by  : Z. Altac
4   * ================================================================ */
5   #include <stdio.h>     // Preprocessor directive
6   #define PI 3.14159      /* Global data definition */
7
8   /* The main program is contained within { } marks */
9   int main(void) {        /* The main (void) program starts here */
10      float radius, area; // Local variable definitions
11      radius = 12.0;      // Local data (radius) definitions
12
13      // Statements
14      area = pi * radius * radius; // Calculate the area of circle
15
16      /* Display the result (area) on the screen */
17      printf("Area= %f", area); // printf is a <stdio.h> library function
18
19      return 0;           // Program is terminated
20  }
```

In line 11, the local variable `radius` is defined as `12.0`.

In line 14, where the computation statement is introduced, the area is calculated by the formula $A = \pi r^2$.

In line 17, the result (calculated `area`) is printed (on screen) using the `printf()` function of `<stdio.h>` library.

In line 19, the program is terminated with `return 0`. The return value of zero signifies successful execution.

1.2 VARIABLES, CONSTANTS, AND INITIALIZATION

1.2.1 Identifiers and Data Types

Identifiers (i.e., symbolic names for variables, functions, and so on) are represented symbolically with letters or combinations of letters and numbers (`a`, `b`, `ax`, `xy`, `a1`, `tol`, ...). The first character of an identifier must be a letter, which includes underscore (_). The C language has 32 keywords (such as `int`, `do`, `while`, etc)) reserved for specific tasks and cannot be adopted as identifiers.

The C language supports the a wide variety of built-in data types: signed integer type (`int`, `short`, `long`), real types (`float`, `double`, `long double`), and void type (*see* full listing).

Integer (`int`): Integers are whole numbers that can hold positive, or negative whole values; e. g., 0, -23, or 140.

Character (`char`): A single character data holds an information of 1.

Real numbers (`float`, `double` or `long double`): Real numbers (or floats) with a decimal point can be represented; e. g., -2.3, 0.14, 1.2e5, and so on. `float`, `double`, and `long double` require 32, 64, and 80 bits.

Void (`void`): It is an incomplete type, which implies "nothing" or "no type".

Signed/Unsigned (`signed/unsigned`): These are type modifiers. As `signed` allows the storage of both positive and negative numbers, the `unsigned` can store only positive numbers.

1.2.2 Type Declaration or Initialization of Variables and Constants

A *variable* is a symbolic representation of the address in memory space where values are stored, accessed, and updated. A *variable declaration* is to specify its *type* of the variable (i.e., the *identifier*) as `char`, `int`, `float` and so on, but a value to the variable has yet to be assigned. The value of a variable changes during the execution of a computer program, while the value of a `constant` does not. All variables (or constants) used in a program or subprogram must be declared before they are used in any statement.

The type of every constant or variable name must be declared in the program before it is used. The type declaration syntax for variables or constants is as follows:

```
type  identifier-1, identifier-2, ..., identifier-n ;
type  constant-identifier = type-compatible value ;
```

`<stdlib.h>` (STandarD LIBrary) provides functions for memory management and general utilities, which includes declarations for functions like `malloc()`, `free()`, and other functions for dynamic memory allocation and manipulation, as well as various utility functions.

A *variable definition* or *variable initialization* is carried out by assigning a value to it, typically with the assignment operator `"="`.

In C, the identifiers are cases sensitive, where the lower-case and upper case letters are treated as different. For example, `Name`, `NAME`, or `name` denote three different variables. In general, lower-case letters are used for variables, and upper case letters are used for constants (i.e., macro definitions).

The `type` declaration is carried out at the beginning of the `main()` function. A variable can be initialized while being declared. The following are a set of example declarations.

```
int iter, maxit, i, j; // Declare integer variables
float radius, area;    // Declare float (real) variables
int p=0, m = 99;       // p and m are declared and initialized
float PI=3.14159;      // pi is declared and initialized
double yd, zd=2.0*PI   // zd is declared and initialized
char one, s='*', no;   // s is declared and initialized
char lines[72];        // a 72-character-long string
```

In this example, `m`, `pi`, `zd`, and `s` are not only type-declared but also initialized at the same time. The order of declaration can be important. For instance, in order to initialize `zd`, `PI` must be declared and initialized beforehand.

In general, a variable does not have a default value. Using the value of an uninitialized variable in operations may lead to unpredictable results or in some compilers the program may crash.

Table 1.2: Symbol and format specifiers for I/O

Data Type	Format Specifier
Integer (`int`)	`%d`
Character (`char`)	`%c`
Single precision floating point (`float`)	`%f` or `%F`
Double precision floating point (`double`)	`%lf`
Floating point value in exponential form (`float`)	`%e` or `%E`
Double precision floating point (`double`)	`%lf`
Addresses in memory (pointers)	%p
Specifier Task	Specifier symbol
Horizontal tab spacing (of 8 spaces)	`\t`
Vertical tab spacing	`\v`
Linefeed return (makes a newline)	`\n`

1.3 INPUT/OUTPUT (I/O) FUNCTIONS

An inevitable element in any program is the communication of the input and output data with the main program or its sub-programs. This is done through suitable `stdio.h` standard library functions, which are used for reading initial values of variables into the program from a file (or console) or writing out intermediate or final values of variables to a file (or a console).

In C programming, the `printf()` and `scanf()` are the most important and useful functions to display and read data on or from input and output devices. The `scanf()` is a function commonly used to supply a set of input data from the user *via* keyboard. It can be used to enter any combination of formatted or unformatted input data. On the other hand, the `printf()` function is used to display the results of a set of intermediate or final operations on the monitor. This function also can be used to print any combination of data.

The syntax for the `nmaviprintf` and `scanf` functions are as follows:

```
printf("format string", argument-1, argument-2, ..., argument-n);
scanf ("format string", &variable-1, &variable-2, ...,&variable-n);
```

where `format string` may involve not only ordinary characters but also *format specifications*, which begin with the % character.

1.3.1 Format Specifications

A *format specification* is a placeholder denoting a value to be filled in during the printing. It gives the user a control over the appearance of the output, while making the I/O statements complicated and hard to read.

The simplest format specifications for `int` and `float` variables can be expressed as follows:

```
For   int (integer)        %wd
For   float (or double)    %w.pf
```

where `w` denotes the width (i.e. , number of digits) and `p` is number of digits to be displayed after the decimal point, after rounding off if necessary. The width `w` in the case of a `float` includes the decimal point as well. A short list of format symbols and specifiers are provided in **Table 1.2**.

For example, the following code segment reserves 5 characters (`w=5`) for the integer and 8 characters (`w=8`

with p=5) for the float variable. In the following output, the spaces between the letter 'n' and '%' symbol are depicted by '_', and the blank spaces are noted by 'b'.

```
float fval=1.2345;
int nval=12;
  printf("nval % 5d \nfval % 8.5f",nval,fval);
```

The output is

```
nval_bbb12
fval_b1.23450
```

We use the `printf` and `fscanf` functions to write data to and read from a disk, respectively. For instance, a call to `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

Forgetting to implement the & symbol in the `scanf` function is a very common user error! Some compiler may not spot this error. Since the task that this symbol is to perform cannot be realized, the variable whose value is read is not saved in the memory field reserved for it. Thus, the variable retains its (perhaps meaningless) value in memory.

```c
/* C program to show input and output                              */
#include <stdio.h>
int main()
{
    int int_val;
    char ch_val;
    float f_val;
    char strng[10];

    // Read an integer value
    printf("Enter an integer value : \n");
    scanf(" %d", &int_val);

    // Read a float value
    printf("Enter a float value : \n");
    scanf(" %f", &f_val);

    // Read a character value
    printf("Enter a character value : \n");
    scanf(" %c", &ch_val);

    // Read a 10-character long string value
    printf("Enter a string value : \n");
    scanf(" %s", &strng);

    // Print the input values
    printf("\nInteger input value is: %d", int_val);
```

```
        printf("\nFloat input value is: %f", f_val);
        printf("\nCharacter input value is: %c", ch_val);
        printf("\nString input value is: %s", strng);

        return 0;
    }
```

The printed results are unformatted. The results can be formatted according to the user desires.

1.4 ASSIGNMENT OPERATION

Expressions involving the arithmetic (or bitwise) operators often involve the assignment operator =, e. g., `z = x + y`. In some cases, a variable is observed on the left- and right-hand-sides as `sum = sum + x` or `sum = sum - x`. These expression can be compressed as `x += y`, where the operator += now denotes an assignment operator. The arithmetic operators (+, -, *, /) and %) have a corresponding assignment operators: +=, -=, *=, /=) and %= (*see* **Table 1.3**).

In C, an *expression* on the right-hand side of the "=" sign is evaluated first, and its value is placed at the allocated memory location of the *variable* on the left-hand side. Any recently computed value of *variable* replaces its previous value. An *assignment* denoted by ← (a left-arrow) in pseudocode notation is replaced with '=' sign.

Consider C code segment below:

```
1   int X = 0;  // X is initialized by zero
2   X ++;       // Equivalent to X = X +1, increment X by 1, X becomes 1
3   X +=2;      // Equivalent to X = X +2, increment X by 2, X becomes 3
4   X = X + 4;  // Increment X by 4                        , X becomes 7
5   X --;       // Equivalent to X = X -1, decrement X by 1, X becomes 6
6   X -=3;      // Equivalent to X = X -3, decrement X by 3, X becomes 3
```

In line 1, `X` is declared as integer and initialized at the same time. The memory value of `X` becomes zero. In line 2, the memory value of `X` is substituted in the rhs, which updates the value of `X` as 4. Finally, in line 3, the rhs is evaluated first (4+6=10), and the result is placed in the memory location of `X`.

1.5 SEQUENTIAL STATEMENTS

Frequently, a sequence of statements (with no imposed conditions) are used to perform a specific task. These statements will be executed in the order they are specified in the program. By using a semicolon (;) as a separator, multiple expressions can be placed in a single line. For instance, the following C code segment illustrates how the first and second statements are presented on a single line with the use of a semicolon.

```
a = b * b + c * c ;  d = sqrt(a) // Statement-1 and Statement-2
x1 = d / ( b + c )               // Statement-3
x2 = d / ( b - c )               // Statement-4
y = x1 + x2                      // Statement-5
```

1.6 ARITHMETIC OPERATIONS

Arithmetic operations involve plus (+), minus (-), multiplication (*), division (/), and the modulus operators (%). These operations (excluding the modulus operator) can be used with integer or floating-point types. The

modulus operator involving an integer division truncates any fractional part, e.g., `15/3`, `16/3`, and `17/3` all yield 5. The modulus operator(`x%y`) produces the remainder from the division `x/y`, e.g., `15% 3=0`, `16%3=1`, and `17%3=2`.

An important set of arithmetic operators are the increment `++` and decrement `-` operators, which add `1` to a variable and subtract `1` from a variable, respectively. In other words, the expression `x++` performs `x = x + 1` operation. The increment and decremant operators can be used as prefix or postfix (`++x`, `x++`, `-x`, and `x-`) with different characteristics.

1.7 RELATIONAL AND LOGICAL OPERATORS

Branching in a computer program causes a computer to execute a different block of instructions, deviating from its default behavior of executing instructions sequentially.

Logical calculations are carried out with an assignment statement:

```
Logical_variable = Logical_expression ;
```

Logical_expression can be a combination of logical constants, logical variables, and logical operators. A logical operator is defined as an operator on numeric, character, or logical data that yields a logical result. There are two basic types of logical operators: relational operators (`<, >, <=, >=, ==, !=`) and combinational (logical) operators (`&&, ||, !`). Branching is carried out using and . In **Table 1.3**, the arithmetic, relational, and logical operators are summarized.

Branching structures are controlled by *logical variables* and *logical operations*. Logical operators evaluate relational expressions to either 1 (`True`) or 0 (`False`). Logical operators are typically used with Boolean operands. The logical AND operator (&&) and the logical OR operator (||) are both binary in nature (require two operands). The logical NOT operator (!) negates the value of a Boolean operand, and it is a unary operator.

Logical operators are used in a program together with relational operators to control the flow of the program. The `&&` and `||` operators connect pairs of conditional expressions. Let $\mathbf{L}_1$ and $\mathbf{L}_2$ be two logical prepositions. In order for $\mathbf{L}_1$ `&&` $\mathbf{L}_2$ to be True, both $\mathbf{L}_1$ and $\mathbf{L}_2$ must be True. In order for $\mathbf{L}_1$ `||` $\mathbf{L}_2$ to be True, it is sufficient to have either $\mathbf{L}_1$ or $\mathbf{L}_2$ to be True. When using `!`, a unary operator, in any logical statement, the logic value is changed to True when it is False or changed to False when it is True. These operators can be used to combine multiple expressions. For given `x=5`, `y=9`, `a=18`, and `b=3`, we can construct following logical expressions:

```
1    (x < y && y < a && a > x)     // TRUE
2    (x < y && y > a && a >= b)    // FALSE
3    ((x < y && y < a) || a < b)   // TRUE
4    ((x > y || y > a) || a < b)   // FALSE
```

The order of evaluation of `&&` and `||` is from left to right.

In C, there is no *exponentiation operator*. The operation is generally carried out with the `exp (a * ln(b) )` or `pow (base, exponent)` function of `<math.h>` library.

1.8 CONDITIONAL CONSTRUCTIONS

In C, branching control and conditional structures allow the execution flow to jump to a different part of the program. *Conditional* statements create branches in the execution path based on the evaluation of a condition.

Table 1.3: Arithmetic, equality/relational, logical and assignment operators in C.

Operator	Description	Example
`+, -`	Addition and subrations	`a + b` or `a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	finding the remainder (modulo).	`5 % 2`
`==`	compares the operands to determine equality	`a == b`
`!=`	compares the operands to determine unequality	`a != b`
`>`	determines if first operand greater	`a > b`
`<`	determines if first operand smaller	`a > b`
`<=`	determines if first operand smaller than or equal to	`a > b`
`>=`	determines if first operand greater and equal to	`a > b`
`++`	Increment operator, which increments the value of the operand by 1	`i++` or `++i`
`--`	Decrement operator, which decrements the value of the operand by 1	`i--` or `--i`
`&&`	Logical AND operator	`a && b`
`||`	Logical OR operator	`a || b`
`!`	Logical NOT operator	`!(a)`
$\pm$`=`	Addition assignment	`p` $\pm$`= q` is equivalent to `p = p` $\pm$ `q`
`*=`	Multiplication assignment	`p *= q` is equivalent to `p = p * q`
`/=`	Division assignment	`p /= q` is equivalent to `p = p /q`

When a control statement is reached, the condition is evaluated, and a path is selected according to the result of the condition.

In C language, we use `if`, `if-else`, `if-else-if`, and `switch` constructions allow one to check a condition and execute certain parts of the code if *condition* (logical expression) is True.

1.8.1 `if`, `if-else`, `if-else-if` Construction

The `if` construct shown below is the most common and simplest form of the conditional constructs. It executes a block of statements *if and only if* a logical expression (i.e., *condition*) is True.

```
if (condition)
    { STATEMENT(s) }     // if condition is true
```

Note that the *condition* is enclosed with round brackets. The `if` construct can also command multiple statements. By wrapping them in braces, the block of statements is made syntactically equivalent to a single statement.

A more general `if-else` construct (presented below) contains another block for the statements to be executed in the case the *condition* is False.

The **if-else** construct has the following form:

```
if (condition)
    { STATEMENT(s) }     // if condition is true
else
    { STATEMENT(s) }     // if condition is false
```

A more complicated `if-else-if` construct can be devised as follows:

```
if (<condition-1>)
   {                      // if <condition-1> is true
       if (<condition-2>)
            { STATEMENT(s) }     // if <condition-2> is true
        else
            { STATEMENT(s) }     // if <condition-2> is false
    }
else
   {                      // if <condition-1> is false
        if (<condition-3>)
             { STATEMENT(s) }     // if <condition-3> is true
        else
            { STATEMENT(s) }     // if <condition-3> is false
    }
```

1.8.2 ?: Conditional Expression

A conditional expression can be constructed with the ternary operator `"?:"`, which provides an alternate way to write `if-else` or similar conditional constructs. The syntax is he following expression

```
<condition> ? <value_if_true> : <value_if_false> ;
```

This statement evaluates the `condition` first. If the condition is *true*, `<value_if_true>`; otherwise, `<value_if_false>` is evaluated. Note that `<value_if_true>` and `<value_if_false>` must be of the same type, and they must be simple expressions rather than full statements.

The example below (determining the maximum of a pair of integers) illustrates the use of ternary operator:

```
int a = 10, b = 20, c, d;

c = (a > b) ? a : b;      // c = max(a, b)
printf("%d", c);          // c becomes 20

d = (a >6 ? (b<= 25 ? 13 : 25) : 100);
printf ("%d\n", d);       // d becomes 13
```

In evaluating `c`, the condition `(a > b)` is evaluated, and since `a<b`, the value of `c` is set to `b`, i.e., 20. In evaluating `d`, the condition `(b <= 25)` is evaluated, which yields `13` since the condition is `true`. Then `(a > 6)` is evaluated to give `13` since this condition is also `true`.

1.8.3 switch Construction

The switch construction is an alternative to the if-else-if ladder. A switch construction allows a multi-decision cases to be executed based on the value of an integer switch variable.

The general form of the switch statement is as follows:

```
switch (expression) {

    case value1:
       { STATEMENT(s) }

    case value2:
       { STATEMENT(s) }
    ....
    default:
       { STATEMENT(s) }
}
```

where value1, value2, and so on are integers. Upon evaluating expression, if and when it matches one of the available cases, the switch branches to the matching case and executes the statements from that point onwards. Otherwise, it branches to default, which is optional, and executes its statements.

The following C code segment uses year to execute switch construct. For the case of year=1, Freshman is displayed; for year=2, year=3, and year=4, Sophomore, Junior, and senior is displayed, respectively. If year corresponds none of the above, the message Graduated is displayed.

```
int year=3;        // year is initialized
switch (year) {
    case 1:
        printf("Freshman");
        break;
    case 2:
        printf("Sophomore");
        break;
    case 3:
        printf("Junior");      // Output is  Junior
        break;
    case 4:
        printf("Senior");
        break;
    default:
        printf("Graduated");
}
```

An Introduction to An Introduction to the C Programming Language and Software Design, Tim Bailey

If the break statement is not used at the end of each case, all statements after the matching label will also be executed.

1.9 CONTROL CONSTRUCTIONS

Control (loop) constructions are used when a program needs to execute a block of instructions repeatedly until a <*condition*> is met, at which time the loop is terminated. There are three control statements in most programming languages that behave in the same way: `while`, `do-while` (equivalent to Repeat-Until loop), and `for`-constructs.

1.9.1 `while` CONSTRUCTS

A `while`-construct has the following form

```
while ( <condition> )
    { STATEMENT(s) }   // if <condition> is true
```

In the `while` construct, the `<condition>` is evaluated before the statement block. If the `<condition>` is `true`, the block of statement(s) will be executed. If the condition is initially `false`, the statement block will be skipped.

1.9.2 `do-while` CONSTRUCTS

The `do-while` construct is equivalent to Repeat-Until construct. The test `<condition>` is at the bottom of the loop. It is similar to `while`-construct in that the statement-block is executed as long as the `<condition>` is `False`.

The `do-while` construct has the form

```
do
    { STATEMENT(s)  }   // if <condition> is false
while ( <condition> )
```

Note that the block will be executed at least once.

1.9.3 `for`- CONSTRUCT

The `for`-construct is used when a block of STATEMENT(S) is to be executed a specified number of times. A `for`-construct has the form

```
for ( <init>; <condition>; <increment> )
    {  STATEMENT(s);  }  // statements in the block are executed
```

where `<init>` initializes the loop-control variable, the <condition> is the loop-continuation condition, and the <increment> increments (or decrements) the control variable.

For instance, consider the following loops:

```
int i, j, k, m;
   for (i=2; i<10 ; i++) {
        STATEMENT(s);    // The block is executed for i=2, 3,..., 9
   }
   for (j=1; j<9 ; j+=2) {
```

```
        STATEMENT(s);    // The block is executed for i=1, 3, 5, 7
    }
    for (k=5; k>2 ; k--) {
        STATEMENT(s);    // The block is executed for k=5, 4, 3
    }
    for (m=6; m>0 ; m-=2) {
        STATEMENT(s);    // The block is executedfor k=6, 4, 2
    }
```

where <*init*> denotes declaration and initialization of the loop index or loop counter, i_1 and i_2 are the initial and terminal values of the index, and an optional parameter Δi denotes increments $(\Delta i > 0)$ or decrements $(\Delta i < 0)$; if it is omitted, it is 1 by default. The total number of iterations is $(i_2 - i_1 + \Delta i)/\Delta i$. To break out of a **For**-loop (**Exit**) before it reaches n_2, a *condition* within the STATEMENT(S) block needs to be specified.

A flowchart for a **For**-construct is illustrated in Fig. . The initial step is executed first and only once; the next $i \leqslant i_2$ condition is evaluated. If this condition is True, then the STATEMENT(S) in the iteration loop are executed. Otherwise, (if $i \leqslant i_2$ is False) the STATEMENT(S) are not executed, and the flow of control jumps to the next statement just after **End For**.

A **For**-construct is used when the number of iterations to be performed is known beforehand. It is easy to use in nested loop settings (with arrays) due to having clearly identified loop indexes.

1.9.4 `break` and `continue` Statements

As covered earlier, `break` was used to branch out of a `switch`-statement. Likewise, it could also be used to branch out of any of the control constructs. Therefore, a `break` can be used to terminate the execution of a `switch`, `while`, `do-while`, or `for`. It is important to realise that a `break` will only branch out of an innermost enclosing block, and transfers program-flow to the first statement following the block. For example, consider a nested `do`-loops

```
int i, d;
for (d=2; d<4; d++) {                    // Output is
    for (i=d; i<= 6; i++) {              //  d=2 i=2
        printf ("d=%d i=%d\n", d,i);     //  d=2 i=3
        if (i==4) {                      //  d=2 i=4
            break;                       //  d=3 i=3
        };                               //  d=3 i=4
    };
}
```

The `break` branch out of innermost `for`-loop when `i` becomes `i=4` as the outermost `for`-loop continues to run over all available index values.

The `continue`-statement operates on `while`, `do-while`, or `for` loops but not on `switch`. In `while` and `do-while` constructs, the loop-continuation test is evaluated immediately after the `continue` statement executes.

In the `for` loop of the following code segment, the loop control variable is initialized at `n=1`. Then the loop-condition (`n<=5`?) is evaluated. When the loop variable becomes 3 (i.e., `n=3`), the `continue` statement skips the rest of the statements and `printf` and takes the iteration to top of the loop.

```
    for (int n = 1; n <=5; ++n) {
       // skip if n=3
       if (n == 3) {
           continue; // skip remaining code in loop body
       }
       int n2=n*n;
       int n3=n*n2;
       printf("n= %d %d %d\n", n,n2,n3);
       }
}
```

The code output is as follows:

```
n= 1 1 1
n= 2 4 8
n= 4 16 64
n= 5 25 125
```

1.9.5 Exiting a loop

It is sometimes required to exit a loop other than by the `<condition>` at the top or bottom. The `break` statement provides an early exit from `for`, `while`, and `do-while`, just as from `switch`.

The C `stdlib` library `exit()` function provides the user to *terminate* or *exit* the calling process immediately. The `exit` statement is used anywhere in the program, subprograms, or loops. The `exit()` function is invoked, it closes all open file descriptors belonging to the process and any children of the process inherited.

```
       int i, d;
       for (d=2; d<4; d++) {                    // Output is
           for (i=d; i<= 6; i++) {              //  d=2 i=2
               printf ("d=%d i=%d\n", d,i);     //  d=2 i=3
               if (i==4) {                      //  d=2 i=4
                   exit(0);                     //  program is terminated
                   };
           };
       };            //  All statements after the 'exit' are not executed
       printf ("d=%d i=%d\n", d,i);
```

1.10 ARRAYS

An array is a special case of a variable representing a set of data (variables) under one group name. Arrays can have one, two or more dimensions. The subscripts are always *integers*. Arrays must always be explicitly declared at the beginning of each program because the range and length of an array are critical in programming.

To create an array, the name of the array followed by square brackets `[]` is specified after defining the data type. To initial values are assigned by using a comma-separated list enclosed by curly braces.

Consider the following example:

```
       int arr[6];       // one-dimensional integer array of length 6
       float b[2][2];    // two-dimensional float array of length 4
```

The first statement creates a one dimensional array `arr`, having six integer elements. Each element can be accessed or referred to by an appropriate subscript in "[]" brackets; that is, `a[0]`, `a[1]`, ..., `a[5]`. The second statement creates a two-dimensional array named `b` having four elements: `b[0][0]`, `b[0][1]`, `b[1][0]`, and `b[1][1]`.

Subscripts in arrays start from *zero*. Thus, if the size of an array is `n`, to access the last element (i.e., `n-1`) index is used.

Arrays can be initialized when they are declared as follows:

```
type  array-name [size] = { list of values } ;
```

In the following example,

```
int a[6] ={5, 2, -4, 1, 7, -8};          // initialization of 1-d array
float b[2][2]={{4.2, 3.7},{1.9, 8.3}}; // initialization of 2-d array
int c[] ={8, 3, 6, 1, 7, 1, 4};          // initialization of 1-d array
char car[3] = {'a', 'b', 'c'};           // initialization of 1-d array
float d[31] = {0};                       // initialization with zeros
```

The initialization of `a` is performed as `a[0]`=5, `a[2]`=2, `a[3]`=-4,....., `a[5]`=-8. The twp-dimensional array `b` is initialized as follows: `b[0][0]`=4.2, `b[0][1]`=3.7, `b[1][0]`=1.9, and `b[1][1]`=8.3. In the initialization of `c`, the size of an array is not specified. This array is automatically be allocated memory to match the number of elements in the list. Initialization of `d` is carried out by matching the type and size of the array. We often need to initialize a large array with a value. This can simply done as `d[size]=value`.

As for one dimensional arrays, multi-dimensional arrays may be defined without a specific size. However, only the left-most subscript (i.e., the number of rows) is free, and the other dimensions must be given a definite value

This pseudocode convention also adopts *whole array arithmetic*. Under certain circumstances, whole array arithmetic is used in order to keep the pseudocodes as short as possible without sacrificing the intended operation. If two arrays have the *same size*, then they are used in arithmetic operations where the operation is carried out on an element-by-element basis. The whole array arithmetic expressions do not require **For**-loops and operate with the array names as if they were scalars.

A brief summary of whole array operations (using conformable arrays) is presented in the pseudocode segment below:

Declare: a_n, b_n, c_n, $e_{10,10}$, $f_{10,10}$, $g_{10,10}$ \ Declaring array variables
c←**a**+**b** \ Performs **c=a+b**
c← 4**a**+(−3)**b** \ Performs c_i=4*a_i+(−3)*b_i for all i
c← **a***b \ Performs $c_i = a_i * b_i$ for all i
. . .
g← **e**+5***f** \ Performs g_{ij}=e_{ij}+5*f_{ij} for all i, j
Write: c \ Prints c_i sequentially for all i
. . .

In the pseudocodes in this text, vectors (i.e., one-dimensional arrays) and matrices (i.e., two- or multi-dimensional arrays) are denoted in lowercase and uppercase bold typeface, respectively. For instance, **a** denotes $a_1, a_2, \cdots, a_n$ and **E** denotes $e_{1,1}, e_{1,2}, \cdots, e_{n,m}$ and they should be declared as

Declare: $a_n,\ e_{n,m}$.

1.11 FUNCTION CONSTRUCTIONS

A C program is a (main) function designed to perform a specific task. In general, it is impractical to write a complete *program* from A to Z that includes everything. Such programs would not only be too long but also too complicated to be practical. When writing a program, it is often necessary to repeat the same set of tasks multiple times within the same program.

In C, to avoid repetition in coding, `functions` are utilized to perform specific tasks. In this regard, functions can be viewed in two categories: (i) those provided with the standard C library and (ii) those prepared by the user (i.e., user-defined functions). The C standard library provides a collection of functions to perform common mathematical calculations, string manipulations, character manipulations, input/output, etc. On the other hand, the programmer can also write user-defined functions to do specific tasks, such as inverting a matrix, finding the real roots of a polynomial, and so on, that may be used numerous times in a program.

1.11.1 Function Declaration

The general structure of a `function` in C is given below:

```
return-type function-name ( p1, p2, ..., pn )
{     Declarations
      Statements
}
```

where p1, p2,... , pn are the input argument list. The `function-name` is a valid identifier. The *`return-type`* is the data type of the result returned to the caller. The `void` return-type indicates that a function does not return a value. The return-type, function-name, and argument-list together are often referred to as the function header.

A function can also have its own internal (*local*) variables that are accessible only internally, i.e., its content is invisible to other functions. A function is only prepared once, and it is accessed and executed from the `main` function or any other `functions` whenever needed. Once a return-type function completes the specific task it is supposed to perform, at least one output value is returned to the calling function. There is no restriction on the number of functions. A function can be used with any other relevant programs and can be invoked or accessed many times in the same program.

A *statement* is a command that is to be executed when the program runs. The local variables should also be declared before they can be used. The declarations and statements within braces form the function block. There are two ways to return (as illustrated below) from a called function to the point at which it was invoked.

```
return expression;   // in return-type functions
return;               // in void-return-type functions
```

A void return-type function does not return a value while a return-type function returns a result computed from the *expression*. The return-type of a function is the type of value that the function returns. The main(void) function (or main()) however returns zero (i.e., return 0;), and the word void as argument (or no argument) indicates that the main has no arguments.

The following is an example of a void function that generates no computed value or return data.

```
void             // this is the 'return type'
print_welcome( ) // by the function!
{
   printf("Welcome to the game!");
   return;       // A void function can be terminated by 'return'
}
```

The following (return-value) function finds the maximum of two real numbers a and b.

```
float                  // this is the 'return type'
findmax( float a, float b) // function name 'max' is followed by parameters
{ if (a > b)
    {
      return a;        // 'a' and return value are 'float' type
    }
  else
    {
      return b;
    }
}
```

Note that this float-type function returns a float value under any circumstances.

Functions are called by naming the function along with their entire argument list (i.e., referred to as function call) to which information is passed. Consider the following main program involving the use of findmax.

```
1  #include <stdio.h>
2      int main() {   // makes use of function findmax
3          float a, b, c, d, s1, s2;
4             printf("Enter four numbers\n");
5             scanf("%f %f %f %f", &a, &b, &c, &d);
6             s1 = findmax(a, b);
7             s2 = findmax(c, d);
8             printf("The max. value is %.3f\n", findmax(s1, s2));
9          return 0;
10     }
```

In this program findmax is invoked in lines 6, 7, and 8. In line 6, s1 is set to the maximum of (a,b). In line 7, s2 is set to the maximum of (c,d). In line 8, the maximum of (s1,s2) is directly evaluated inside printf function.

In many programming languages, there are two ways to pass arguments—*pass-by-value* and *pass-by-reference*. When arguments are passed by value, changes to the argument do not affect an original value of the variable in the caller. When an argument is passed by reference, the called function can modify the original value of variable. In C, all arguments are passed by value. Through the use of pointers, it is possible to achieve pass-by-reference.

When the compiler encounters the first call of `findmax` at line 6, it does not have any information about `findmax`, its arguments, and so on. To avoid such problems, the function should be placed before it is used. However, as a general and better practice, the function declarations are inserted before `main()`, and function definitions after `main()`. This provides the compiler with a brief glimpse of a function whose full definition will appear later; that is,

```
return-type function-name-1 ( p11, p12, ..., p1n ) ;
return-type function-name-2 ( p21, p22, ..., p2n ) ;
  ...
int main(void) {
       Declearations and Statements
}
return-type function-name-1 ( p11, p12, ..., p1n ) {
       Declearations and Statements
}
  ...
```

In following program, the function declaration (the first line of a function) is placed in line 3 before the `main()` starts at line 5. The complete function definition is placed after the `main()` in lines 15-20.

```
1  #include <stdio.h>
2
3     float findmax( float a, float b); // function declaration
4
5     int main()  {  // main function starts here
6         float a, b, c, d, s1, s2;
7             printf("Enter four numbers\n");
8             scanf("%f %f %f %f", &a, &b, &c, &d);
9             s1 = findmax(a, b);
10            s2 = findmax(c, d);
11            printf("The max. value is %.3f\n", findmax(s1, s2));
12        return 0;
13    }
14    // complete function definition placed after 'main'
15    float  findmax( float a, float b) {
16         if (a > b)  {
17            return a; }         // return-value
18         else {
19            return b; }         // return-value
20    }     // findmax ends here
```

An efficient program generally consists of one or more independent modules. The modular programming approach is also easier to conceptualize and write a program as a whole.

1.11.2 The `exit()` function

Since `main()` is a function, it has to have a return type. Normally, the return type of `main()` is `int`, which is why the main program has been defined in the following way:

```
int main()
    {
        STATEMENT(s)
     }
```

The use of `main()` or `main(void)` is not illegal, but it is best to avoid this practice.

The `exit` function (of `<stdlib.h>`) returns the termination status.

```
exit(0);    // normal termination
exit(1);    // abnormal termination
```

1.11.3 Recursive `function`

In C language, a function in a program is allowed to call itself multiple times. A recursive function is a function that can call itself repeatedly until a certain condition or task is met. That is why, a conditional construction is required to prevent the call go into an infinite loop to terminate the task.

Consider the recursive function $f_n(x) \leftarrow n + x * f_{n-1}(x)$ with $f_0(x) = x$. This function can be made a recursive function as follows:

```
1  float FX(int n, float x) {
2     float f;
3     if(n==0) {
4          f = x;
5      }
6      else {
7          f = (float)n + x * FX( n - 1, x)
8      }
9      return f;
10 }
```

A local intermediate variable is defined (`float f`) and used in lines 4 and 7 to store intermediate result. The self-calling takes place in line 7 as `FX( n - 1, x)`. On function name `FX` takes the value of `f` on return.

1.11.4 Array Arguments

Arguments can often be arrays. When an argument is a one-dimensional array, the length of the argument may not need to be specified.

There are basically two ways to pass an array to a function: *call by value* or *call by reference*. When using the call by value method, the argument need to be an initialized array, or an array of fixed size equal to the size of the array to be passed. In the call by reference method, the argument is a pointer to the array.

In the following code, the `main()` function has an array of integers. A user-defined function `max_of_array` is called by passing array `arr` to it. The `max_of_array` function receives the array, and searches for the largest element using a `for` loop. After `maks` is set to `arr[0]`, whenever an array element with a value greater than `maks` is found, it is set to `maks`. By the time the end of the loop is reached, `maks` gives the largest value in the array `arr`.

```
1  #include <stdio.h>
2
3  int max_of_array(int arr[5]);   // declare function 'max_of_array'
4
5  int main(){                     //  main funtion
6     int arr[] = {21, 47, 93, 38, 25}; //initialize array
7
8     printf("Max value is %d", max_of_array(arr)); // find and print the maximum
9  }                               // end of main
10
11 int max_of_array(int arr[5]) {  //  define function
12    int k;
13    int maks = arr[0];
14    for (k=1; k<5; k++){
15       if( arr[k] > maks ) {
16           maks = arr[k];
17       };
18    }
19    return maks;
20 }                              // declare function 'max_of_array'
```

Note that the length of `arr` is specified in lines 3 and 11 as 5 in the declaration and definition of the function `max_of_array`. In lines 6 and 14, the length of `arr` is compatible with the data.

In the following version, the `max_of_array` function is defined with two arguments, an uninitialized array without any size specification. The length of the array declared in `main()` function is obtained by using the `sizeof` function, which gives the size (in bytes) that a data type occupies in the computer's memory. To find the number of integer values, the total memory size of the array (`sizeof(arr)`) is divided to the size of the single integer data type, i.e, `sizeof(int)`.

```c
#include <stdio.h>

int max_of_array(int n, int arr[]); // decleration of max_of_array

int main(){
   int arr[] = {21, 47, 93, 38, 25}; //initialize array
   int n = sizeof(arr)/sizeof(int);
   printf("Max value is %d", max_of_array(n, arr)); // find and print the maximum
}
int max_of_array(int n, int arr[]) {  //  define the actual function
   int k;
   int maks = arr[0];
   for (k=1; k<n; k++){
      if( arr[k] > maks ) {
          maks = arr[k];
      };
   }
   return maks;
}
```

This version of the function is flexible, allowing it to handle array variables of different lengths.

1.11.5 Pass array with call by reference

To use this approach, we should understand that elements in an array are of similar data type, stored in continuous memory locations, and the array size depends on the data type. Also, the address of the 0'th element is the pointer to the array.

1.12 POINTERS

Pointers are variables whose values are *memory addresses*. As we know, a variable holds a specific value of specified data type. A pointer, however, holds the address rather than the specific value of a variable. In other words, a pointer *points to* that other variable by holding a copy of its address indirectly references a value.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the *address*, and the address points to the contained *value*; i.e., *pointer* and the *value pointed to*.

1.12.1 Declaring a pointer

Pointers, like all variables, must be defined before they can be used. A pointer is declared using * (asterisk symbol). It is also known as *indirection pointer* used to reference a pointer.

```c
datatype * pointer_variableName;
```

The asterisk is allowed to be anywhere between the base type and the variable name. An integer and character pointer variable declaration examples are presented below:

```c
int  *arr;     // declare an integer pointer variable 'arr'
char *letter;  // declare a character pointer variable 'letter'
```

```
float pi=3.14; // initializing float variable 'pi'
float *c=&pi   // declaring and initializing a pointer variable 'c'
```

1.12.2 The Address (&) and Dereferencing Operators (*)

The & (reference) operator is used to store the memory address of a variable and to assign it to the pointer. The & operator computes a pointer to the argument to its right. The argument can be any variable which takes up space in the stack.

The dereferencing operator, also called *indirection operator*, returns the value of the variable to which a pointer points. The following example illustrates the use of a typical pointer:

```
1  #include <stdio.h>
2
3  int main() {
4      int a, n;
5      int *p;    // pointer to an integer
6      p = &a;
7      *p = 5;    // value of the variable to pointer points
8      n = a;
9      printf("% d %d %d \n", a, n, *p);
10     return 0;
11 }
```

In line 4, a and n are declared as integers, while in line 5 p is declared as pointer variable, i.e., *p indicates that a pointer is being declared. In line 6, the variable p points to an integer variable a. In line 6, p = &a statement uses & symbol, which is referred to as the address operator. The expression &a refers to the *memory address of the variable* a. In other words, p = &a assigns the address of a to p. In line 7, *p=5 indicates that the location pointed to by p should be set to 5. Since the location *p is also a, a also takes on the value 5. In line 8, setting n = a assigns the value of n to 5, and consequently, the printf statement produces 5 5 5.

1.12.3 Using Pointers for Function Arguments

There are two ways to pass arguments to a function: *pass-by-value* and *pass-by-reference*. In pass-by-value, a copy of the argument value (of a function) is created in memory, and the caller and callee functions, though having two independent variables, share the same value. If the value of an argument is modified by the callee function, the result of this modification is not transferred to the the original variable in caller function. In C, all arguments are passed by value. In pass-by-reference, the caller and the callee use the same variable memory for the argument. If the value of an argument is modified by callee function, the result of this modification is transferred to the original variable in caller function.

In C, we use pointers and the dereferencing operator to accomplish pass-by-reference. To understand how this works, consider implementing the swap(a,b) presented below, which interchanges the values of the a and b integer variables.

```
#include <stdio.h>

void swap(int a, int b);  // Declare function swap

int main() {
```

```
    int a = 9, b = 5;
    printf("Before swap:a =%d, b=%d\n", a, b);

    swap(a, b);           // swap a and b
    printf("After swap:a=%d, b=%d\n", a, b);
}

void swap(int a, int b) { // Define swap
    int temp = a;
    a = b;
    b = temp;
}
```

When this code is executed, it will be seen that the desired swapping operation is not realized although the values of `a` and `b` are correctly passed to `swap` and they are indeed interchanged in the `swap` function. But still `swap` does not return the outcome realized in the caller function (i.e., `main()`).

To make this function work properly as designed, one must use pointers, as illustrated in the example below:

```
1  #include <stdio.h>
2
3  void swap(int *a, int *b);  // Declare function swap
4
5  int main() {
6      int a = 9, b = 5;
7      printf("Before swap:a =%d, b=%d\n", a, b);
8
9      swap(&a, &b);           // swap a and b
10     printf("After swap:a=%d, b=%d\n", a, b);
11 }
12
13 void swap(int *a, int *b) { // Define swap
14     int temp = *a;
15     *a = *b;
16     *b = temp;
17 }
```

To be able[**knuth:1984**] swap the two values, we will make use of pointers. In line 13, the function `swap` is declared as `void` since it does not return new computed data. The arguments, `a` and `b`, are declared as integer pointers: `int *a` and `int *b`. We then use an integer (non-pointer) temporary variable (`temp`) to implement the swapping algorithm. The values of the pointers in lines 14-16 are accessed as `*a` and `*b`. In line 9, the `swap` function is called and the memory addresses of the arguments `a` and `b` are passed as `&a` and `&b`.

If you leave out `&` symbol when passing arguments (i.e., implementing the call as `swap(a, b)`), the value of the arguments are passed instead of the address, which leads to a `segmentation fault`.

When calling a function with arguments that should be modified, the addresses of the arguments are passed. This is normally accomplished by applying the address operator (&) to the variable (in the caller)

whose value will be modified. Arrays are not passed using operator & because C automatically passes the starting location in memory of the array (the name of an array is equivalent to &arrayName[0]). When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory. Pointers enable programs to accomplish pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures.