CHAPTER 1

NUMERICAL ALGORITHMS AND ERRORS

SUPPLEMENT No. 1a:

C TUTORIAL

prepared for

NUMERICAL METHODS FOR SCIENTISTS AND ENGINEERS With Pseudocodes

By Zekeriya ALTAÇ

October 2024

Supplement No. 1a: The C Tutorial

LEARNING OBJECTIVES

The objective of this C programming language tutorial is to
- present a short summary of the basics of C programming language;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs;
- explain how to prepare functions or subprograms.

The textbook "*Numerical Methods for Scientists and Engineers: With Pseudocodes*" focuses on implementing the methods in science and engineering applications. Supplemental course materials and resources, including C/C++, Fortran, Visual Basic, Python, Matlab®, and Mathematica®, are provided to assist the instructors in their teaching activities outside the class.

The aim of this short tutorial is to enable students to acquire the knowledge and skills to convert the pseudocodes given in the text into running C programming languages. It is not intended to be a "complete language reference document." The author assumes that the reader is familiar with programming concepts in general and may also be familiar with the C programming language at the elementary level. In this regard, this tutorial illustrates the conversion and implementation of pseudocode statements (such as formatted/unformatted input/output statements, loops, accumulators, control and conditional constructs, creating and using functions, and subprograms, etc.) to the C programming language.

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 semicolon (`;`), newlines are not significant except in preprocessor controls, and the blank lines are ignored. All function names, including the main program `main()`, which is also a function, are always followed by `( )` brackets. Curly braces `{ }` contain a group of statements.

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

1.2 GLOBAL DATA DEFINITIONS

Global data definitions refer to variables and constants that are declared outside of any function, making them accessible from any function within the same file or across multiple files. They are stored in the data section of the program memory. Global variables exist for the entire duration of the program execution, and they should be used judiciously to avoid issues related to maintainability, debugging, and concurrency.

Table 1.1: Some of the C header files and functions.

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.

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 has (at least) one function, called `main`.

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.

1.5 A C CODE EXAMPLE

The basic features of a C program (`example.c`) are 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.

Lines 7, 12, 15, and 18 in `example.c` are the blank lines of the code, which are used to break up long sections of a code, making it easier to read and understand. The blank lines are also used to visually separate different logical sections, such as variable declarations, function definitions, major code blocks, etc.

In line 5, the header file (in this case `<stdio.h>`) is executed first. `<stdio.h>` (STandarD Input- Output) is a library file that provides functions and declarations related to input and output operations. It includes functions 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 feature of C language, known as *macro definition*, is used. A macro definition is a way to define a fragment of code or a constant value that can be reused throughout your program. Macros are created using the preprocessor directive `#define`. They are processed by the preprocessor before the actual compilation of the code. Here, a macro or global data (`PI`) is defined as `#define PI 3.14159`. The constant `PI` now becomes accessible by the main and (if present) auxiliary functions.

In lines 9-20, the code commands, statements, or expressions are placed. 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 local variables `radius` and `area` are type-declared 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 assigned the value 12.0.

In line 14, the area ($A = \pi r^2$) is calculated by the given expression.

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

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

In C, the syntax must be correct. Otherwise, the compiler will generate error messages and will not produce executable code.

2 VARIABLES, CONSTANTS, AND INITIALIZATION

2.1 IDENTIFIERS AND DATA TYPES

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

The C supports 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).

The C language supports 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, -1223, or 140. C provides a rich set of integer types suitable for various applications. Understanding their sizes and ranges helps in selecting the appropriate type for your specific needs. The 4-byte integer, **`int`** or **`long`** type, is the most commonly used integer type and ranges from -2,147,483,648 to 2,147,483,647. A **`short`** integer, usually 2 bytes, ranges from -32,768 to 32,767.

Character (`char`): Character variables are used to store single characters and are represented by the **`char`** data type. The **`char`** data type is typically used to represent a single character, such as letters, digits, punctuation marks, etc. A single character of data holds an information of 1 byte. A sequence of characters can be stored in an array of **`char`**, often used to represent strings in C.

Real numbers (`float`, `double` or `long double`): Real numbers 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. A **`float`** typically represents a single-precision floating-point number. It represents a range of values approximately from 1.2×10^{-38} to 3.4×10^{38} with about 6-7 decimal digits of precision. The suffix **`f`** is used for **`float`** literals. By default, decimal literals are considered **`double`**.

Void (`void`): It is an incomplete type, which implies "*nothing*" or "*no type*". It is used as a (i) function return type, (ii) void pointer, and (iii) function parameter, which indicates that a function does not take parameters.

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 values but has a larger positive range.

2.2 TYPE DECLARATION AND INITIALIZATION OF VARIABLES OR 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 the *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 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.

In C, the identifiers are case-sensitive, where the lower- and uppercase 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).

A *variable definition* or *variable initialization* is carried out by assigning a value to it, typically with the assignment operator **`"="`**. 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, **PI** must be declared and initialized before **zd**.

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 the program may crash in some compilers.

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 functions available in the standard library **<stdio.h>**, which are used to read the initial values of variables from a file (or console) into the program or to write the intermediate or final values of variables to a file (or the console).

Before using the input/output functions, the standard input/output header must be included as follows:

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

This statement ensures that the required functions and definitions are available for input/output operations.

In the C language, the **printf()** and **scanf()** are the most important and useful functions to display and read data on or from the console. The **scanf()** is a function commonly used to supply a set of input data from the user, commonly *via* a 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 console. This function can also be used to print any combination of data.

The basic syntax for the **nmaviprintf** and **scanf** functions is as follows:

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

where **"format string"** specified with double quotation marks specifies the expected data types and their order. It includes not only ordinary characters but also *format specifications*, which begin with the % character, followed by a letter representing the desired data type.

The **scanf()** function scans the input from the standard input stream, adhering to the provided **format string**. It reads characters and sequentially matches them with the format specifiers. Note that, when reading a variable into the program, the address-of operator (&) is used before the variable name (except for arrays) to pass the address where **scanf** will store the input.

The following C code illustrates entering integer, character, float, and string variables into a program.

```
#include <stdio.h>
int main()  { // C program to show input and output
    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 (output) are unformatted, i.e., default format settings are used. However, formatted output according to the user's preferences can be accomplished using format specifications.

3.1 FORMAT SPECIFICATIONS

A *format specification* is a placeholder denoting a value to be filled in during the printing. It gives the user control over the appearance of the output while making the I/O statements complicated and hard to read. The simplest format specifications for **int** (integer) and **float** variables (as float or scientific or exponential format) can be expressed as follows:

```
int                  %wd
float (or double)    %w.pf     or     %w.pF
float (or double)    %w.pe     or     %w.pE
```

where **w** denotes the width (i.e., number of digits) and **p** is the 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 is provided in **Table 1.2**. Be cautious when using **%s** with **scanf** as it can result in *buffer overflows*.

Consider the following code segment. It reserves **w=3** and **w=8** characters (**%3d** and **%8d**) for the integer variable **num**. The real number **x** is displayed as *float* and *exponential*. As for the floating number, the variable **x** is displayed with two different *float* format specifiers with character widths of **w=10** and **w=8** and the number of digits of **d=2** and **d=4**, respectively. The variable **x** is also displayed with two different *scientific* format specifiers with character widths of **w=10** and **w=14** and the number of digits of **p=3** and **p=5**, respectively.

```
        float x=123456.789;
        int num=123;
          printf("n= %3d  %8d\n",num,2*num);
          printf("x= %10.2f  %8.3f\n",x,3*x);
          printf("x= %10.3e  %14.5e\n",x,5*x);
```

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`

The output is

```
         1         2         3
12345678901234567890123456789
n= 123       246
x=  123456.79  370370.375
x=  1.235e+05     6.17284e+05
```

where the first two lines represent column numbers, which do not in reality appear in the output device or file. Notice that all of the format strings have a blank space following the '=' sign and two blank spaces between two format specifiers. The format strings end with `\n`, i.e., linefeed return. The character spacing of `w` is reserved for all numbers. In floats, the `p` number of digits is reserved after the decimal point. Also note that expressions such as `2*num`, `3*x`, and `5*x` are allowed in `printf()`.

Forgetting to implement the & symbol in the `scanf` function is a very common user error! Some compilers 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.

4 SEQUENTIAL STATEMENTS

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

```
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
```

5 ASSIGNMENT OPERATORS AND OPERATIONS

Arithmetic operations involve plus (`+`), minus (`-`), multiplication (`*`), division (`/`), and 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`.

Assignment operations are used to assign values or computed results of an expression to variables. A simple *assignment* denoted by ← (a left-arrow) in the pseudocode notation is replaced by the "`=`" sign, syntax as shown below:

```
variable_name = value ;      or
variable_name = expression ;
```

The *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* (i.e., `variable_name`) on the left-hand side. Any recently computed value of a *variable* replaces its previous value in memory.

In C, several other operators for more complex assignments are also provided. These operators help simplify a code by combining an operation with an assignment, reducing redundancy. For instance, a variable may appear on both the left- and right-hand sides of an expression, as in `x = x + y` or `x = x - y`. These expression can be compressed as `x += y` or `x -= y`, where the operators `+=` and `-=` are referred to as *compound assignment operators*. In this regard, compound assignment operators combine arithmetic operators (`+`, `-`, `*`, `/`, and `%`) with the assignment operator (`=`) to get `+=`, `-=`, `*=`, `/=`) and `%=` (*see* **Table 1.3**).

Another set of important arithmetic operators are the increment `++` and decrement `--` operators, respectively, which can be used as prefix (`++x`) or postfix (twx++) with different characteristics. They are applicable to integer and floating-point type variables. Prefix increment, `++x`, increases the value of the variable `x` by 1, and then returns the updated value. But postfix increment `x++` returns the current value of the variable `x` by 1, and then increases the value of `x`. The decrement operator is employed in a similar manner. These operators are commonly used in loops and repetitive constructions to efficiently modify the variable values.

Consider the C code segment given 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
```

Here, in this code segment, the comments on each line describe the tasks performed in that line. In line 1, `X` is defined as an integer and initialized to zero at the same time. In line 2, the memory value of `X` is substituted in the rhs, which updates the value of `X` as 1 but displays zero. In lines 3 and 4, `X` is incremented by 2 and 4, respectively. In lines 5 and 6, `X` is decremented by 1 and 3, respectively.

In C, *exponentiation operator* (raising a number to a power) is not directly supported by a built-in operator as in some other programming languages. The operation is generally carried out with the `exp(a*ln(b))` or `pow(base, exponent)` function of the `<math.h>` library.

6 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. **Table 1.3** summarizes the arithmetic, relational, and logical operators with illustrative examples.

The *relational operators* (`<, >, <=, >=, ==, !=`) are used to compare two values or expressions and give a boolean value: `true` or `false`. These operators are commonly used in conditional and control constructions to control the flow of a program based on comparisons. The relational operators can be used with various data types, including integers, floats, and characters. However, comparing different types may lead to implicit type conversion. While relational operators do not short-circuit like logical operators, they can be combined in logical expressions for more complex conditions.

The *logical operators*, logical AND (`&&`), logical OR (`||`), logical NOT (`!`), are used to combine or negate boolean expressions. They are essential in controlling the flow of programs, especially in conditional statements and loops. These operators are frequently used in conditional constructions, such as `if`-constructs, to evaluate multiple conditions or control the flow of loops based on complex logical expressions.

A *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 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 L_1 and L_2 be two logical prepositions. In order for L_1`&&`L_2 to be true, both L_1 and L_2 must be true. In order for L_1`||`L_2 to be true, it is sufficient to have either L_1 or 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 the following logical expressions:

```
(x < y && y < a && a > x)              //  1, i.e., true
(x < y && y > a && a >= b)             //  0, i.e., false
((x < y && y < a) || a < b)            //  1, i.e., true
((x > y || y > a) || a < b)            //  0, i.e., false
(!(x > 6) && !(y < 5)) && a > x)       //  1, i.e., true
```

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

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

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

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

Operator	Description	Example
ARITHMETIC OPERATORS		
±	Addition and subtractions	`a ± b`
*	Multiplication	`a * b`
/	Division	`a / b`
%	finding the remainder (modulo).	`5 % 2`
RELATIONAL OPERATORS		
==	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`
LOGICAL OPERATORS		
&&	Logical AND operator	`a && b`
||	Logical OR operator	`a || b`
!	Logical NOT operator	`!(a)`
COMPOUND OPERATORS		
±=	Addition assignment (`p = p ± q`)	`p ±= q`
*=	Multiplication assignment (`p = p * q`)	`p *= q`
/=	Division assignment (`p = p /q`)	`p /= q`
++	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`

7.1 if, if-else, if-else-if CONSTRUCTIONS

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. It is equivalent to **If-Then** pseudocode construction.

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

A more general `if-else` construct (presented below) provides an alternative block for the statements to be executed in the case the *condition* is false. It is equivalent to the **If-Then-Else** structure of the pseudocode convention. The **if-else** construct has the following form:

```
if (condition)
    { STATEMENTS-t }      // if condition is true
else
    { STATEMENTS-f }      // if condition is false
```

Note that the *condition* is enclosed with (). The `if` construct can also command multiple statements; by wrapping them in {}, the block of statements is made syntactically equivalent to a single statement.

One may chain multiple conditions using **else if** to test multiple possibilities. A more complicated **if-else if** construct can be devised as follows:

```
if (condition1)
   { STATEMENTS-1 }      // if condition1 is true
else if (condition2)
   { STATEMENTS-2 }      // if condition2 is true
else if (condition3)
   { STATEMENTS-3 }      // if condition3 is true
else
   { STATEMENTS-4 }      // if condition3 is false
```

This chain can be continued indefinitely by repeating the last statement with another **if-else** statement. Also, nesting **if** statements within other **if** statements is allowed.

Nested **if-else** statements permit checking multiple conditions within another **if** or **else** block, which may be useful for more complex decision-making processes. But excessive nesting can also make code harder to read. It is best to keep nesting manageable and make sure to handle all potential conditions to avoid unexpected behavior.

7.2 TERNARY CONDITIONAL OPERATORS

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 given as

```
condition ? val_if_true : val_if_false
```

This statement evaluates the <*condition*> first. If *condition* is *true*, **val_if_true** is executed; otherwise, **val_if_false** is evaluated. Note that **val_if_true** and **val_if_false** must be of the same type, and they must be simple expressions rather than full statements. The following example, which determines the maximum of a pair of integers, illustrates the use of a 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**.

7.3 switch CONSTRUCTION

The **switch** construction is an alternative to the **if-else-if** ladder. A **switch** construction allows a multi-decision case to be executed based on the value of a switch variable (**int**, **char**, or **enum**) or expression. It is a cleaner alternative to using multiple **if** statements when you have many conditions based on a single variable.

Using **switch** can improve the clarity and maintainability of the code when dealing with multiple conditions based on a single variable. The general form of the **switch** statement is as follows:

```
switch (expression) {

    case value1:          //  case of expression = value1
       { STATEMENTS-1 }   //  statements to be executed
       break;             //  exit the construction

    case value2:
       { STATEMENTS-2 }
       break;
        ....
    default:
       { STATEMENTS }
}
```

where **value1**, **value2**, and so on are integers. The expression in the switch must evaluate to an **int**, **char**, or **enum**. Upon evaluating **expression** (the value or variable to be evaluated), 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 **expression** must be an integer or an enumeration constant.

The following C code segment uses **year** to execute **switch** construct. For the case of **year=1**, **Freshman** are displayed; for **year=2**, **year=3**, and **year=4**, **Sophomore**, **Junior**, and **senior** is displayed, respectively. If **year** corresponds to 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");
}
```

The **switch** construct should be used in cases where there are many distinct sets of values to be checked.

Invoking **break** at the end of each case block exits the **switch** construction. If omitted, the execution will "fall through" to the next case, which can be useful in certain scenarios.

8 CONTROL CONSTRUCTIONS

Control (or 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 of pseudocode convention), and `for`-constructs.

8.1 `while` CONSTRUCTION

A `while`-construct has the following form:

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

In `while` construction, *condition* is evaluated before the block statements. If *condition* is `true`, the block of statements inside the loop will be executed. If *condition* is initially false, the statement block will be skipped. For successful implementation of `while` constructions, make sure that (i) the loop variable is initialized before the loop starts; (ii) the loop variable is updated within the loop or an infinite loop might occur; and (iii) be cautious of conditions that might always evaluate to true, leading to infinite loops.

8.2 `do-while` CONSTRUCTION

The `do-while` construction is equivalent to **Repeat-Until** pseudocode construction. The test *condition* is at the bottom of the loop. It is similar to `while`-construction in that a code block (statements) 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 is executed at least once, even if *condition* is `false` from the beginning. This loop is useful when the initial execution of the block statements is necessary, such as prompting for user input.

8.3 `for` CONSTRUCTION

The `for`-construction is a control flow statement used when a block of code needs to be executed a specified number of times. It is equivalent to **For**-loop in the text pseudocode. It is particularly useful when the number of iterations is known beforehand. A `for` construction has the form

```
for (init; condition; update)
   {  STATEMENT(s)  }  // block statements are executed
```

where *init* denotes declaration and initialization of the loop index or loop counter, *condition* is the loop-continuation condition, and *update* denotes updating the loop index by incrementing or decrementing it. The most common way to update the loop index in unity steps is by using increment (`++`) or decrement (`--`) operators.

For instance, consider the following loops:

```
int i, j, k, m; float x;
   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 j=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 m=6, 4, 2
    }
    for (i=5; i<=20 ; i+=5) {
        STATEMENT(s);    // The block is executedfor i=5, 10, 15, 20
    }
    for (x = 0; x <= 1; x+=0.2) {
        STATEMENT(s);    // The block is executedfor x=0, 0.25, 0.5, 0.75, 1
    }
```

The values that the loop index takes are specified in the comment lines above. Here, we can also see that incrementing or decrementing the loop index using values other than one is accomplished through the use of compound assignments, as in `j+=2`, `m-=2`, or `i+=5`. In these examples, the part of the `for`-loop that updates the loop variable, `i += n` or `i -= n`, allows one to specify how much to change the index with each iteration. The value of `n` can be any integer (or even a floating-point number) to control the loop's behavior as desired. Additionally, multiple variables can be declared in the initialization section, separated by commas, for example: `for (int i = 2, j = 99; i < j; i++, j-)`.

A `for`-construction 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.

8.4 `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 realize that a `break` will only branch out of the inner-most 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 the 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
```

8.5 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**. When **break** is encountered, the loop terminates immediately, and control is transferred to the statement following the loop.

Using the **return** statement inside a loop will exit the loop and return control to the calling function (or exit the program if it's in **main**). This is typically used to terminate the loop when a certain condition is met and you want to exit the function altogether. On the other hand, the **goto** statement can be used to jump to a labeled statement, effectively allowing one to exit loops. However, it's generally discouraged because it can lead to less readable and harder-to-maintain code.

In the **stdlib** library, the **exit()** function provides the user to *terminate* or *exit* the calling process immediately. The **exit** function can be used anywhere in the program, subprograms, or loops. When the **exit()** function is invoked, it closes all open file descriptors belonging to the process and any children of the process inherited. Following illustrates the conditional loop exit.

```
    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);
```

9 ARRAYS

In C, an array, as a special case of a variable, is a collection of elements of the same type stored in contiguous memory locations. Arrays can have one, two, or more dimensions, i.e., a_1, a_2, a_3, $b_{1,1}$, $b_{2,1}$, $c_{1,1,3}$, and so on. 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. Initial values are assigned by using a comma-separated list enclosed by curly braces. For example, in the following declarations, the first expression creates a one-dimensional array `arr`, having six integer elements. Each element can be accessed or referred to by a subscript in "[]" brackets; that is, `a[0]`, `a[1]`, ..., `a[5]`. The second expression creates a two-dimensional array `b` having four elements: `b[0][0]`, `b[0][1]`, `b[1][0]`, and `b[1][1]`.

```
int arr[6];       // one-dimensional integer array of length 6
float b[2][2];    // two-dimensional float array of length 4 (2x2)
int c[2][3][4];   // three-dimensional integer array of length 24 (2x3x4)
```

Arrays can be initialized when they are declared as follows:

```
type  array-name [size] = { ',' separated list } ;
```

Consider the following array initializations:

```
int a[6] ={5, 2, -4, 1, 7, -8};          // initialization of 1-d array
float b[2][2]={{4., 3.},{1.9, 8.}};      // 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
int matrix[3][3] = {{1, 2, 3},           // initialization of 3x3 matrix
                    {4, 5, 6},
                    {7, 8, 9}};
int arrB[4] = {8, 3};                    // yields {8, 3, 0, 0}
```

Here, the array `a` is initialized as `a[0]`=5, `a[2]`=2, `a[3]`=−4,....., `a[5]`=−8. The two-dimensional array `b` is initialized as follows: `b[0][0]`=4., `b[0][1]`=3., `b[1][0]`=1.9, and `b[1][1]`=8. But the size of array `c` is not specified, in which case the compiler automatically assigns a size equal to the number of elements with which the array is initialized. The character type array **`car`** has a length of 3 and is initialized as `car[0]='a'`, `car[1]='b'`, and `car[2]='c'`. The initialization of array `d` to zero is carried out by matching the type and size of the array, i.e., creating 32 zeros. The 3×3 matrix is initialized *row-wise*. In the integer array **`arrB`**, the first two elements are initialized as **`arrB[0]=8`** and **`arrB[1]=3`**; others are set to zero.

Subscripts in arrays start from *zero*, i.e., *lower bound*=0 and *upper bound*=$n-1$, where n is the *size* of the array. In order to access the last element of the array, index **`n-1`** is used.

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*) can be omitted; all other dimensions must be specified. The compiler determines the size of the omitted dimension based on the initializer list. Accessing elements is done using multiple indices. For instance, accessing the second diagonal element of **`matE`** is shown below:

```
int dia2 = matE[2,2];  // accesses the 2nd diagonal element of matrix matE
matE[3,2] = 93;        // changes the element at (3,2) position to 93
```

The C language does not support *whole array operations*, like FORTRAN 90-95, MATLAB, R, or PYTHON with **NumPy**. However, the pseudocode convention adopts *whole array arithmetic*, and array names are treated as if they were scalars. To adapt it to C programming, **For** loops need to be used for such operations.

10 FUNCTION CONSTRUCTIONS

Declaring and using functions is a fundamental part of writing structured and modular code. Functions help organize code into reusable blocks, making it easier to read, maintain, and debug. A small sample of the C math library functions are presented in **Table 1.4**, where the variables x and y are of type **double**. Always refer to the official documentation or resources specific to your compiler for any additional functions or variations that might be available.

A C program itself is a function (named **main**) designed to perform a specific task (i.e., the aim of the program). A program often requires repetitive tasks. On the other hand, in practice, it is impractical to write a complete *program* from A to Z that includes everything within a **main** program. Such programs would not only be too long but also too complicated. To avoid repetition, **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, strings, characters, input/output manipulations, etc. On the other hand, the programmer does also need to prepare *user-defined functions* to perform specific tasks that are not available in standard libraries, which may be used once or numerous times in the program.

Make sure that you are familiar with the available functions in the C standard library, and instead of writing your own functions, use these functions, which are efficient, well-tested, and portable.

10.1 FUNCTION DECLARATION

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

```
1       return-type function-name (p1, p2, ..., pn)
2           {   Declarations
3               Statement(s)
4               return value;  }
```

Here, line 1 is the *function prototype* (also called a *function declaration*). The **function-name** is any valid identifier that should be descriptive of what the function really does. The **return-type** is the data type of the result (**int**, **float**, **double**, etc.) 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*.

The **p1**, **p2**, ... , **pn** are the input parameters given in the *parameter list*, separated with commas. This list specifies the arguments of the function when it is called. If a function does not have any argument, then the parameter list is **void**. A **type** must be stated explicitly for each function parameter. The **value** is the returned function value.

Table 1.4: Some of the available `<math.h>` library functions.

Functiom	Description
`fabs(x)`	absolute value of x
`sqrt(x)`	square root of x
`round(x)`	rounds x to nearest integer
`trunc(x)`	truncates x to nearest integer part
`ceil(x)`	the smallest integer greater than or equal to x
`floor(x)`	the largest integer less than or equal to x
`fmod(x, y)`	the remainder of x divided by y
`pow(x, a)`	`x` raised to the power of `a`
`exp(x)`	the exponential value of e^x
`log(x)`	the natural logarithm (base e) of x
`log10(x)`	the logarithm (base 10) of x
`log2(x)`	the logarithm (base 2) of x
`sin(x)`	the sine of x (angle in radians)
`asin(x)`	the arcsine of x (result in radians)
`atan(x)`	the arctangent of x (result in radians)
`atan2(x)`	the arctangent of x, taking into account the signs of both arguments
`sinh(x)`	the hyperbolic sine of x
`cbrt(x)`	the cube root of x

The braces following the function prototype form the *function body*, which is also a *block*. The body of a C function, *code block*, contains the statements that define commands and expressions to be executed to perform a specific task when the program is called. This is where the logic of the function is implemented. 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 then it can be accessed and executed from the `main` function or any other `function` whenever needed. The local variables should also be declared before they can be used.

There are three ways to return control from a called function to the point at which a function was invoked.

```
return;                 // in void return-type functions
return expression;      // in return-type functions
return 0;               // indicate successful main() execution
```

In a void function, which does not need to return a result, the control is simply returned when the function-terminating right parenthesis is reached. If a function is the return-type function, the value of the last *expression* in the statements is returned to the calling function after completing the specific task it is supposed to perform. However, the `main(void)` function or `main()` returns zero (i.e., `return 0;`), and the word `void` as argument (or no argument) indicates that the `main` has no input parameters.

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

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

The following (return-type) function finds the maximum of the two **float** numbers, **a** and **b**.

```
float findmax( float a, float b)   // float function findmax
 {
  if (a > b)
      return a;         // set return value to 'a'
  else
      return b;         // set return value to 'b'
   }
```

Note that this **float**-type function has two **float** type input parameters and returns a **float** value under any circumstances.

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 **function** 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.

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() {   // uses "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, the function **findmax** is invoked in lines 6, 7, and 8. In line 6, the variable **s1** is set to the maximum of **(a,b)**. In line 7, the variable **s2** is set to the maximum of **(c,d)**. In line 8, the maximum of (**s1**,**s2**) is directly evaluated inside **printf** function. In this code, the function **findmax** is placed before **main**. So when the compiler encounters **findmax**, it *does* not have any information about **findmax**, its arguments, and so on. However, if **function findmax** is placed after the main code, the compiler will *not* know what the term **findmax** is. To avoid such problems, each function should be placed before first use.

In most computer 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 the variable. In C, all arguments are 'passed by value'. Through the use of pointers, it is possible to achieve pass-by-reference.

As a general and better practice, the function prototypes are placed at the top of the program before `main()`, and function definitions after the `main()`. This provides the compiler with a brief glimpse of a function whose full definition will appear later, as shown below:

```
return-type function-name-1( p11, p12, ..., p1n ) ;
return-type function-name-2( p21, p22, ..., p2n ) ;
       ...

int main(void) {    // begining of the main
    Declearations and Statements
     }              // end of the main

return-type function-name-1( p11, p12, ..., p1n )
  { Declearations and Statements
    return value-1;
     }              // end of function-name-1
return-type function-name-2( p21, p22, ..., p2n )
  { Declearations and Statements
    return value-2;
     }              // end of function-name-2
     ...
```

In the following code segment, the function prototype or declaration (*the first line of the function*) is placed on line 3, i.e., before the `main()`, which is placed on lines 5-13. The complete function, involving comparison of two numbers, is placed on lines 15-20 after the `main()` function. The function `findmax` is called on lines 9, 10, and 11 by simply using it in an assignment or list context.

```
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
```

10.2 The exit() FUNCTION

Since **main()** is a function, it has to have a return type. Normally, the return type of **main()** is integer, 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
```

10.3 RECURSIVE FUNCTION

In C, 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. This approach is often used in cases where a task can be broken down into smaller, similar sub-problems or navigating data structures like trees.

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

```
1  #include <stdio.h>
2
3  float FX(int n, float x) {   // define recursive function
4      float f;
5      if(n == 0)
6          f = x;
7      else
8          f = (float)n + x * FX(n - 1, x);
9      return f;  }
10
11 int main() {
12     int n = 5;
13     float x = 1.1;
14     printf("FX of %d is %f\n", n, FX(n, x));
15     return 0; }
```

Here, a local intermediate variable defined as **float f** is used in lines 6 and 8 to store intermediate results. The self-calling takes place in line 8 as **FX(n - 1, x)**. In line 9, on the function name **FX** takes the value of **f** on return.

10.4 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 needs 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;  }             // 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 the **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 by the size of the single integer data type, i.e., **sizeof(int)**.

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

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

10.5 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.

11 POINTERS

Pointers are variables whose values are *memory addresses*. As we know, a variable holds a specific value of a 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*.

11.1 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.

```
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:

```
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'
```

11.2 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 that 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 a 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`.

11.3 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 original variable in the 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 the callee function, the result of this modification is transferred to the original variable in the 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 to 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 the **`&`** symbol when passing arguments (i.e., implementing the call as **`swap(a, b)`**), the values 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 the & 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, pass functions between functions, and create and manipulate dynamic data structures.

Bibliography

[1] BIKSHALU., K., *A Textbook of Basics of C-Language Programming*. Educreation Publishing, 2018.

[2] COPES, F., *C HANDBOOK*, flaviocopes.com.

[3] DAWSON, R. *Programming in ANSI C*. Third Ed. Group D Publications, 2000.

[4] DEITEL, P., DEITEL, H. C., *C How to Program: with an introduction to C++*, 8th Ed.. Pearson Education, 2016.

[5] KERNIGHAN, B. W., RITCHIE, D. *C Programming Language.* Second Ed., Prentice Hall Software Series, 1988.

[6] KING, K. N. *C Programming: A Modern Approach*. W.W. Norton, 2008.

[7] KOCHAN, S. *Programming in C*. Pearson Education, 2005.

[8] REDDY, R., ZIEGLER, C. *C Programming for Scientists and Engineers with Applications*. Jones & Bartlett Learning, 2010.

[9] SZUHAY, J. *Learn C Programming: A Beginner's Guide to Learning C Programming the Easy and Disciplined Way*. Packt Publishing, 2020.

[10] WARREN, T. *C Programming for Beginners: The Simple Guide to Learning C Programming Language Fast!*. Ingram Publishing, 2020.

[11] www.tutorialspoint.com

[12] www.w3schools.com

[13] https://www.learn-c.org

[14] www.programiz.com

[15] www.codechef.com

[16] GNU C library

[17] ISO C Standards