C
The Programming Language
Course Material

Written by:
Dalal BARDOU

Associate Professor at
Computer Science Department, Faculty of Sciences and Technology,
University of Abbes Laghrour, Khenchela-2021

This course is designed for undergraduate students. Precisely, for first-year students in the material sciences department, Faculty of Sciences and Technology. This course covers programming in the C language. Students will learn how to translate a problem already solved with an algorithm to an executable program written in C language.

CONTENTS

LIST OF FIGURES

CHAPTER

1

INTRODUCTION TO PROGRAMMING LANGUAGE C

1.1 Introduction

Computers are dumb machines because they only do what they're told to do. They operate at a very primitive level. For example, they know how to add one to a number or how to test if a number is equal to zero. The sophistication of these basic operations does not go much further than that. These basic operations form what are called computer instructions.

To solve a problem with a computer, one must express the solution in terms of instructions. So a computer program is just a set of written instructions utilized to solve a specific problem. The approach or method used to solve a problem is called an "Algorithm". For example, if we want to develop a program that tests whether a number is even or odd, we must first express the solution in terms of an algorithm and then we develop the program that implements this algorithm with a programming language like C [**?**], C++, Java, ...etc.

A programming language is an interface between human beings and machines. It makes it possible to formulate algorithms in the form of source code that will be analyzed by a machine and to produce computer programs that apply these algorithms. A language used to write code is called "high-level language". This code can be then compiled into a "low-level language", which is recognized directly by computer hardware.

1.2 A Brief history of C language

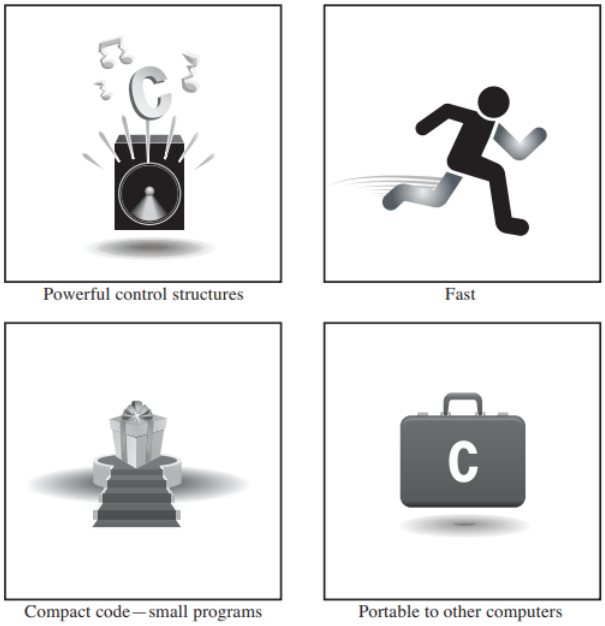
The C programming language already has a long history in the computer world. It is the culmination of two languages: BPCL developed by Richards and B developed by Thompson. It is one of the most powerful advanced languages. It was invented by the two engineers Dennis Ritchie

and Brian Kernighan in the 1970s to design a portable, UNIX operating system in which more than 90 % of the kernel is written in C [**?**]. It is a language widely used in industry because it combines the advantages of a high-level language (portability, modularity, etc.) and those of assembly languages close to hardware but more difficult to use for large projects.

1.3 Why C?

During the past four decades, C has become one of the most important and popular programming languages. It has grown because people try it and like it. In the past decade or two, many have moved from C to languages such as C++, Objective C, and Java, but C is still an important language in its own right, as well a migration path to these others (see Figure 1.1). Let's preview a few of them now:

Figure 1.1: The virtues of C [**?**].



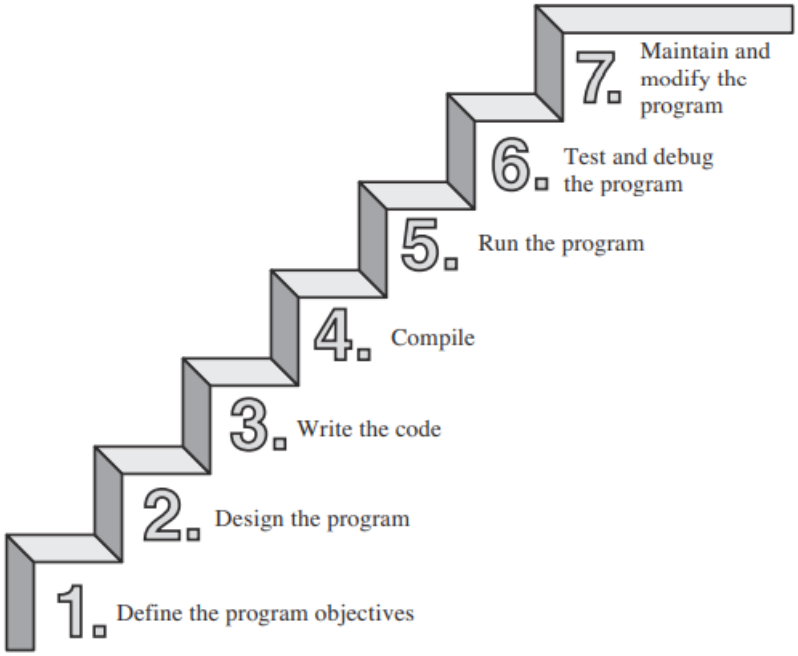
- **Efficiency:** C is an efficient language. Its design takes advantage of the capabilities of current computers. C programs tend to be compact and to run quickly. In fact, C exhibits some of the fine control usually associated with an assembly language. (An assembly language is a mnemonic representation of the set of internal instructions used by a particular central processing unit design.
- **Portability:** C is a portable language, which means that C programs written on one system can be run on other systems with little or no modification.
- **Power and flexibility:** C is powerful and flexible (two favorite words in computer literature). For example, most of the powerful, flexible Unix operating system was written in C. Many compilers and interpreters for other languages such as FORTRAN, Perl, Python, Pascal, LISP, Logo, and BASIC have been written in C. As a result, when you use FORTRAN

on a Unix machine, ultimately a C program has done the work of producing the final executable program.

1.4 Using C: Seven steps

Writing a C program mainly comprises seven steps (see Figure 1.2).

Figure 1.2: The seven steps of programming in C [**?**].



- **Define the objectives of the program:** In this step, you should start with a clear idea of what you want the program to do. Think about the information your program needs, the calculations and manipulation it needs to do, and the information the program should give you. At this level of planning, you should think in general terms, not in terms of certain programming languages.
- **Design the program:** Once you have an idea of what your program should do, you need to decide how the program will go about it. What should the user interface look like? How should the program be organized? Who will be the target user?
- **Write the code:** Now that you have a clear design for your program, you can start implementing it by writing the code. That is, you translate your program design into the C language.
- **Compile the program:** The next step is to compile the source code with a compiler. A compiler is a program whose job is to convert source code into executable code. The executable code is the native language or machine language. The compiler also checks that your program is valid in C. If the compiler finds any errors, it reports them and does not produce an executable file.

- **Run the program:** After successfully compiling the program, you can now run it.
- **Test and debug the program:** The fact that your program is working is a good sign, but it is possible that it could work incorrectly. Therefore, you need to verify that your program is doing what it is supposed to do. You will find that some of your programs have errors-bugs. So debugging helps to detect and correct these errors.
- **Maintain and modify the program:** As things progress, you'll probably find reasons to make changes your program.

CHAPTER

2

THE BASICS OF C PROGRAMMING

2.1 The backbone of a C program

A C program can be defined as follows:

1. # directives addressed to the preprocessor.
2. Declarations (objects to be manipulated)
3. int main ()
4. {
5. Declarations (objects to be manipulated)
6. Instructions (the operations to be performed)
7. }

Example

The following program displays the message "Hello":

```
1. /* A program that displays hello. */
2. #include<stdio.h>
3. int main ()
4. {
5. printf ("Hello!\ n");
6. return 0;
7. }
```

- The first line of this program is a comment, intended for the human reader only: the compiler ignores everything in between * et */.

- The third line is a preprocessor directive and tells the compiler that we will be using interfaces from the standard input/output library (the standard input/output library, or stdio).
- The rest of the program consists of the function definition, a series of instructions framed by braces " { " et " } " and having a name (here main).
- The body of the main function of the program contains two instructions. The first instruction prints on the standard output the string "Hello" followed by the character of line end (" \ n " for newline). Like any instruction in C, it is terminated with a semicolon ";". The last statement ends the execution of the main function.

2.2 Basics elements of C program

2.2.1 Variables

A variable defines a location name where we can put value and we can use this value whenever required in the program. The value of a variable can be changed at different times of executions and it may be chosen by the programmer in a meaningful way. Its name also should be unique. It is important to give meaningful names for identifiers as it is self-documenting and easy for other programmers to understand. A variable has:

1. **Name or identifier:** A variable that appears in programming should have a name. Its name always starts with an alphabetic character. The other characters can be alphabetic or numeric or the underscore. A name must consist of 1 to 31 characters with no embedded blank spaces. Special characters are not allowed in the name.
2. **Type:** Each variable should have a type and it is used to distinguish variables from each other. We use different types of variables such as integers, reals, character, string, etc.
3. **Value:** Once the variable is defined, a value is assigned to it.
4. **Address:** The address is the location of the variable in the computer memory where its value is stored.

Example

Examples of valid and invalid names of variables:

1. **Valid names:** test, pgm, highest_value, total_count, xs567.
2. **Invalid names:**
   (a) 5test: First character cannot be a digit.
   (b) pg*m: Illegal character "*".
   (c) one,value: Illegal character ",".

2.2.2 Variable types in C

In C language, as in other languages, it is always necessary, by precise instructions, to declare (define) the name and type of a variable, before you can assign a value to it which must be com-

patible with the declared type. If subsequently another value is assigned to a variable, the previous one is lost.

Notes

In C language, the main types of variables are as follows:

1. **Boolean**
   - Set of definition: {True, False}.
   - Algorithmic declaration: **a,b: Boolean;**
   - Declaration in C: **bool a,b;**

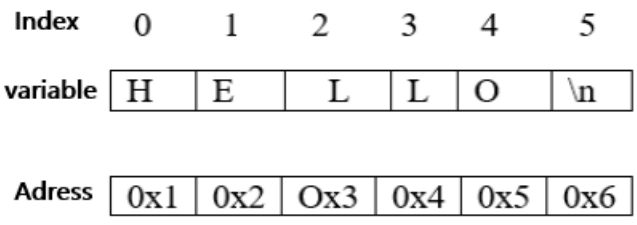
2. **Integer**
   - Algorithmic declaration: **i,j: integer;**
   - Declaration in C: **int i,j;**

3. **Float**
   - Algorithmic declaration: **x,y: float;**
   - Declaration in C: **float x,r; or double x,y;**

4. **Character**
   - Algorithmic declaration: **c,g: char;**
   - Declaration in C: **char c,g;**

5. **String:** In C, strings are one-dimensional arrays of characters terminated by a null character.
   - Algorithmic declaration: **c: string;**
   - Declaration in C: **char msg [size];**, where size is the number of characters in the string. Figure 2.1 shows an example of a string variable.

Figure 2.1: An example of String declaration in C.



2.2.3 Constants

If you want to define a variable whose value cannot be changed, you can use the keyword **const**. This will create a constant. Example: **const double PI = 3.14;**

2.2.4 Comments

In C, you can place comments in your source code that are not executed as part of the program. Comments provide clarity to the source code allowing others to better understand what the code was intended to accomplish and greatly helping in debugging the code. They are important in large projects containing hundreds or thousands of lines of source code or in projects in which many contributors are working on the source code.

A comment starts with a slash asterisk /* and ends with an asterisk slash */ and can be anywhere in your program. Comments can span several lines within your C program. They are typically added directly above the related C source code.

Syntax

The syntax of a comment is: **/* Add your comment here */** Or:

```
/*
* Add your comment here
*/
```

for multiple lines comment.

2.2.5 Arithmetic and logical operators in C

Engineering and scientific computing involves the use of arithmetic and logical operators. Arithmetic operators can be performed on numeric data of type int, float, and double, while logical operators involve the evaluation of logical expressions that produce either true or false.

Arithmetic operators

Operator	Algorithmic notation	C notation
Addtion +	a+b	a+b
substraction -	a-b	a-b
Unary minus -	-a	-a
Multiplication *	a*b	a*b
Division /	a/b	a/b

Logical operators

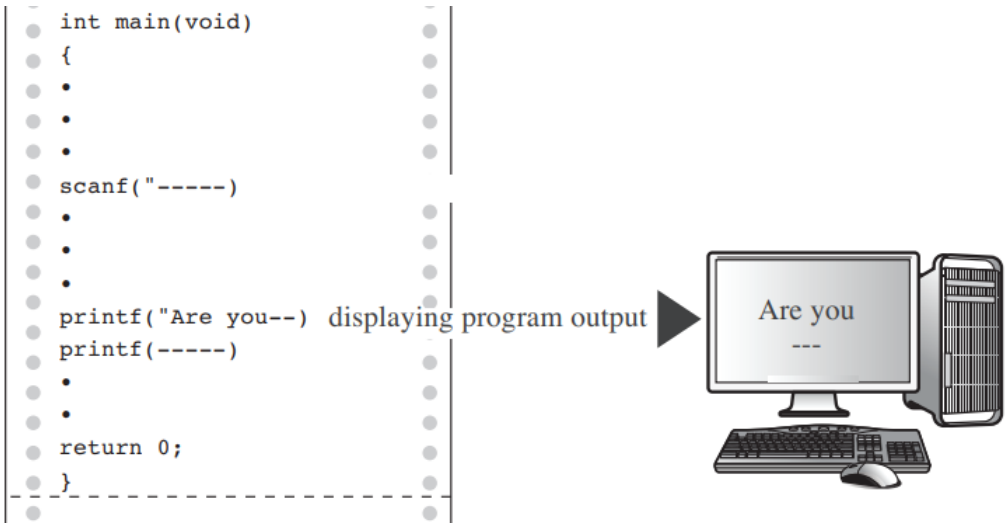
Operator	Algorithmic notation	C notation
Negation	not(a)	! a
Or	a or b	a$\|\|\|b$
And	a and b	a && b

2.2.6 Printf function

The printf function is used to transfer text, variable values, or expression results to the stdout standard output file (the screen by default). Printf function has the following format: **printf**

("<format>", <Exp1>,<Exp2>,...) where "<format>" comprises the format specifiers indicating how the values of expressions <Expr1..N> are printed. The "<format>" part contains exactly one format specifier for each <Expr1..N> expression. Format specifiers always start with the % symbol and end with one or two characters that indicate the print format. <Expr1>,... are the variables and expressions where the values should be represented.

Figure 2.2: The Printf function at work.



The format specifiers for printf are listed below:

1. **%d or %i**: To display a signed integer in decimal format (int).
2. **%u**: To display an unsigned integer in decimal format.
3. **%f**: To display a float or double.
4. **%c**: To display a character.
5. **%s**: To display a string.

Example

Here are two examples of using printf:

1. printf ("Hello World !") ;
2. printf ("%d kilogram is equivalent to %d grams", 1, 1000) ;

The argument of the printf function named format is a string that determines what will be display by printf and in what form. In Example 1, it's "Hello World!" and in example 2 is "% d kilogram is equivalent to% d grams". This string consists of "normal" text and control sequences for including variables in the output. Thus, in example 2, when displaying the first% d will be replaced by the value 1 and the second% d by the value 1000. The result is the following display: "1 kilogram is equivalent to 1000 grams".

Example

The following instructions:

1. int A = 1234 ;
2. int B = 567;
3. printf ("%i multiplied by %i is equal to %li \ n", A, B, (long)A*B);

Will display on screen: 1234 multiplied by 567 is equal to 699678.
The arguments of printf are:

- The format part "% i multiplied by % i is % li"
- The variable A.
- The variable B.
- The expression (long)A*B.

The 1st specifier (% i) indicates that the value of A will be printed as an integer (1234).
The 2nd specifier (% i) indicates that the value of B will be printed as an integer (567).
The 3rd specifier (% li) indicates that the value of (long) A * B will be printed as a long integer (69967).

Example

The following instructions:

1. char B = 'A';
2. printf ("The character %c has the code %i ! \ n", B, B);

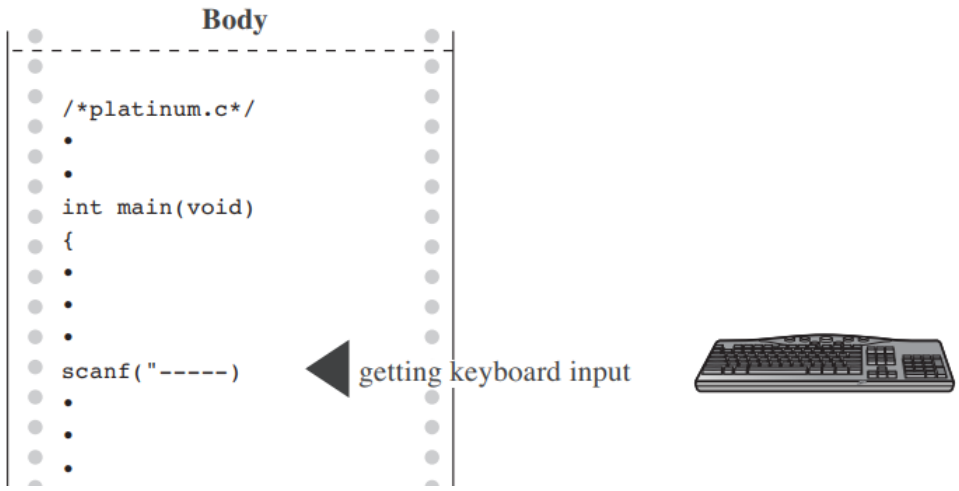
Will display on screen: The character A has the code 65!
The value of B is therefore displayed in two different formats:

- %c as a character: A.
- %i as an integer: 65.

2.2.7 Scanf function

This function allows reading the so-called formatted data from the standard input (keyboard). Scanf uses the same formats as printf but we precede the name of the variable with the character ampersand ("&"). For example, the syntax of a variable i of type int is :**scanf ("%d", &i);**

Figure 2.3: The Scanf function at work.



Example

```
1. int main(int argc, char *argv[]
2. {
3. char name[100]
4. printf("What is your name?");
5. scanf("%s", name);
6. printf("Hello %s, Nice to meet you !", name);
7. return 0;
8. }
```

Example

```
1. #include <stdio.h>
2. int main () {
3. int nb1; float nb2;
4. printf("Please, provide a value for nb1 : ") ;
5. scanf("%d",nb1) ;
6. printf("Please, provide a value for nb2 : ") ;
7. scanf("%f", nb2) ;
8. printf("nb1 is %d ; nb2 is %f \ n", nb1, nb2) ;
9. return 0;
10. }
```

2.2.8 Assignment operators

In C, assignment is an operator in its own right. It is symbolized by the sign =. Its syntax is the following: **variable = expression**.

The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type of the variable on the left side otherwise the compiler will raise an error. There are many types of assignment operators (given below).

1. **"="**: This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left. For example:
   - x=4;
   - y= 67;
   - mychar= 'B';

2. **"+="**: This operator is the combination of '+' and '=' operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left. For example: **(a += b)** is equivalent to **(a = a + b)**. If a is equal to 2 and b is equal to 3 then a= 5 after evaluating the expression.
3. **"-="**: This operator is the combination of '-' and '=' operators. This operator first subtracts the current value of the variable on left from the value on the right and then assigns the result to the variable on the left. For example:**(a -= b)** is equivalent to **(a = a - b)**. If a is equal to 2 and b is equal to 3 then a= -1 after evaluating the expression.
4. **"*="**: This operator is the combination of '*' and '=' operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left. For example: **(a *= b)** is equivalent to **(a = a * b)**. If a is equal to 2 and b is equal to 3 then a= 6 after evaluating the expression.
5. **"/="**: This operator is the combination of '/' and '=' operators. This operator first divides the current value of the variable on left to the value on the right and then assigns the result to the variable on the left. For example: **(a /= b)** is equivalent to **(a = a / b)**. If a is equal to 4 and b is equal to 2 then a= 2 after evaluating the expression.

2.2.9 Increment/Decrement operator

Increment (++) and decrement (--) operators are two special unary operators used to increment and decrement the value of a variable by 1 respectively. These two operators can be used only with variables. They can't be used with constants or expressions. Increment/Decrement operators are of two types:

1. **Prefix increment/decrement operator**: The prefix increment/decrement operator firstly increases or decreases the current value of the variable by 1 and then this value is used in the expression. For example: **y = ++x;**. Here, the current value of x is firstly incremented by 1. Then, the new value of x is assigned to y. Similarly, in the statement: **y = --x;**, the current value of x is decremented by 1 and then the new value of x is assigned to y.
2. **Postfix increment/decrement operator**: The postfix increment/decrement operator causes the current value of the variable to be used in the expression, then the value is incremented or decremented by 1. For example:**y = x++;**. Here, the current value of x is assigned to y then x is incremented. Similarly, in the statement:**y = x--;**, the current value of x is as-

signed to y then x is decremented.

Example

Example of Prefix increment/decrement operator:

```
1. #include <stdio.h>
2. int main () {
3. int x=12, y=1;
4. printf("Initial value of x = %d \ n", x); // print the initial value of x.
5. printf("Initial value of y = %d \ n", y); // print the initial value of y.
6. y = ++x; // increment the value of x by 1 then assign this new value to y.
7. printf("After incrementing by 1: x = %d \ n", x);
8. printf("y = %d \ n", y);
9. y = --x; // decrement the value of x by 1 then assign this new value to y.
10. printf("After decrementing by 1: x = %d \ n", x);
11. printf("y = %d \ n", y);
12. return 0;
13. }
```

The output after the evaluation is:

Initial value of x = 12

Initial value of y = 1

After incrementing by 1: x = 13

y = 13

After decrementing by 1: x = 12

y = 12

Example

Example of Postfix increment/decrement operator:

```
1. #include <stdio.h>
2. int main () {
3. int x=12, y=1;
4. printf("Initial value of x = %d \ n", x); // print the initial value of x.
5. printf("Initial value of y = %d \ n", y); // print the initial value of y.
6. y = x++; // use the current value of x then increment it by 1.
7. printf("After incrementing by 1: x = %d \ n", x);
8. printf("y = %d \ n", y);
9. y = x--; // use the current value of x then decrement it by 1.
10. printf("After decrementing by 1: x = %d \ n", x);
11. printf("y = %d \ n", y);
```

12. return 0;
13. }

The output after the evaluation is:

Initial value of x = 12

Initial value of y = 1

After incrementing by 1: x = 13

y = 12

After decrementing by 1: x = 12

y = 13

2.2.10 Comma operator

An expression can be made up of a series of expressions separated by commas: expression-1, expression-2, ..., expression-n. This expression is then evaluated from left to right. Its value will be the value of the expression on the right. For example, the program given below displays **b=5**.

```
1. main () {
2. int a,b;
3. b = ((a = 3), (a + 2));
4. printf (" b = %d \ n", b);}
```

2.2.11 Type conversion operator

The type conversion operator, called cast, allows to explicitly modify the type of an object. We write **(type) object**. Type indicated the data type to which the final result is converted.

Advantages of Type Conversion:

1. This is done to take advantage of certain features of type hierarchies or type representations.
2. It helps us to compute expressions containing variables of different data types.

Example

The program given below returns after evaluation 1.5.

```
1. main () {
2. int i=3, j=2;
3. printf ("%f \ n",(float) i/j);
```

Example

The program given below returns after evaluation 2.

```
1. #include<stdio.h>
2. main () {
3. double x = 1.2;
4. // Explicit conversion from double to int
5. int sum = (int)x + 1;
6. printf (""sum = %d", sum);
7. return 0;
8. }
```

2.2.12 Modulus operator

The modulus operator is used in integer arithmetic. It gives the remainder that results when the integer to its left is divided by the integer to its right. For example, 18 % 5 (read as "18 modulo 5") has the value 3, because 5 goes into 18 thrice, with a remainder of 3. This operator only works with integers and does not work with floating-point numbers.

Return Possibilities of the Modulus Operator:

- If the variable a is completely divisible by the second number (b), it returns zero (0), or the remainder becomes 0.
- If the variable a is not completely divisible by the second number (b), it returns a numeric value in the range [1, a - 1]. Or we can say it returns the remainder to a non-zero integer value.
- If the first number (a) is non-zero and the second number is 0, it gives an error at compile time.

Example

The program given below converts seconds to minutes and seconds.

```
1. #include<stdio.h>
2. main () {
3. int sec, min, left;
4. printf("Convert seconds to minutes and seconds!");
5. printf("Enter the number of seconds:");
6. scanf("%d", &sec); // read number of seconds
7. min = sec / 60; // truncated number of minute
8. left = sec % 60; // number of seconds left over
9. printf("%d seconds is %d minutes, %d seconds.", sec, min, left);
10. return 0;
```

```
11. }
```

Here is some sample output:
Convert seconds to minutes and seconds!
Enter the number of seconds :
154
154 seconds is 2 minutes, 34 seconds.

2.3 Exercises

2.3.1 Exercise 1

Write a C program that reads two real numbers, and then displays their product, with a precision of three numbers after the comma.

Solution

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main () {
4. float x, y, p;
5. printf("Enter two reals: \ n");
6. scanf("%f %f",&x, &y);
7. p=x*y;
8. printf("The product of %f and %f is: %.3f \ n", x, y, p);
9. system ("pause");
10. return 0;
11. }
```

2.3.2 Exercise 2

Write a C program that allows swapping the content of two variables of type integers, passing through a third auxiliary variable. Then, display the content of the two variables before and after permutation.

Solution

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main () {
4. int a, b, tmp;
5. printf("Enter two integers a and b: \ n");
6. scanf("%d %d",&a, &b);
7. printf("Before permutation: a= %d and b = %d \ n", a, b);
8. tmp=a;
9. a=b;
10. b=tmp;
11. printf("After permutation: a= %d and b = %d \ n", a, b);
12. system ("pause");
13. return 0;
14. }
```

2.3.3 Exercise 3

Write a C program that reads three integers as input and displays their average with a precision of two numbers after the comma.

Solution

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main () {
4. int a, b,c;
5. float avg;
6. printf("Enter three integers: \ n");
7. scanf("%d %d %d ",&a, &b, &c);
8. avg= (float)(a+b+c)/3;
9. printf("The average of these three integers is: %.2f \ n", avg);
10. system ("pause");
11. return 0;
12. }
```

2.3.4 Exercise 4

Write a C program that reads as input an alphabetic character between a and y (The character can be either uppercase or lowercase) and display the letter that just comes after it in the alphabetical order.

Solution

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main () {
4. char letter;
5. printf("Enter a character between a and y: \ n");
6. scanf("%c",&letter);
7. letter++;
8. printf("The next character is: %c \ n", letter);
9. system ("pause");
10. return 0;
11. }
```

2.3.5 Exercise 5

Write a C program that reads two integers and then displays the max among them. This exercise aims to recall the formula for calculating the max of two numbers. Given a and b, any two

numbers, their max is given by: max(a,b) = (a + b + absolute (a - b))/2 where absolute (a - b) is the absolute value of the difference (a-b).

Solution

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main () {
4. int a,b, max;
5. printf("Enter two integers: \ n");
6. scanf("%d %d ",&a,&b);
7. max=(a+b+abs(a-b))/2;
8. printf("The max of %d and %d is: %d. \ n", max);
9. system ("pause");
10. return 0;
11. }
```

CHAPTER

3

DECISION MAKING AND SELECTION STATEMENTS

3.1 Decision Making Statement: The if-else Statement

The if-else statement is used to carry out a logical test and then take one of two possible actions depending on the outcome of the test (ie, whether the outcome is true or false).

The if statement is written in its basic form as follows:

```
1. if (condition)
2. {
3. instruction1;
4. }
```

The instruction instruction1 is executed only if the condition in parenthesis is true. It can be also extended and be written as follows:

```
1. if (condition)
2. {
3. instruction1;
4. }
5. else
6. {
7. instruction2;
8. }
```

Figure 3.1: The flowchart of the If statement execution.

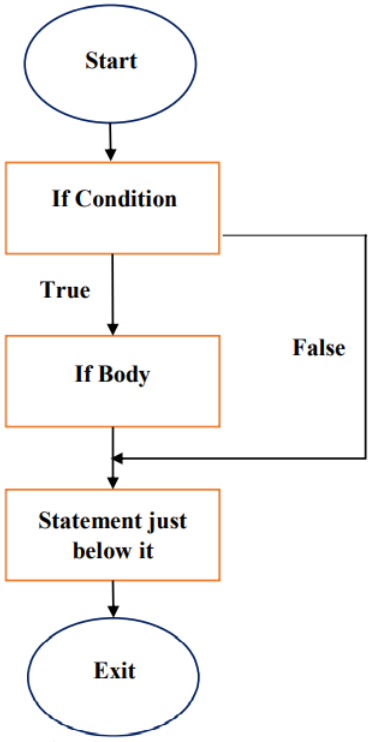
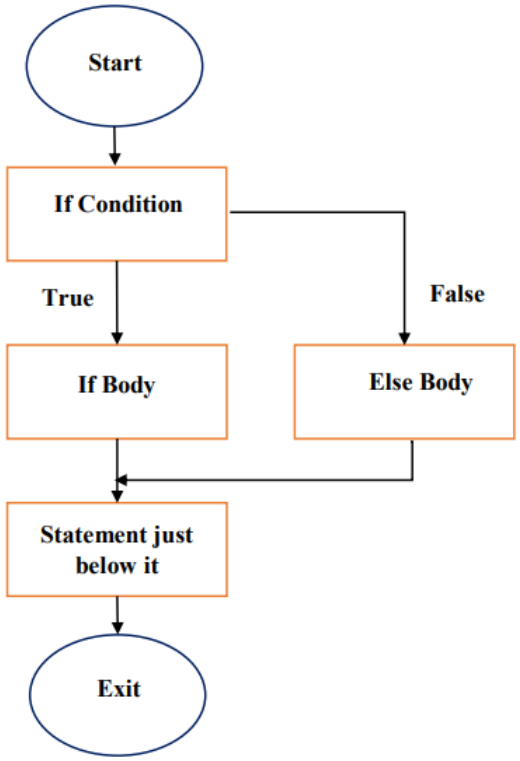


Figure 3.2: The flowchart of the If-else statement execution.

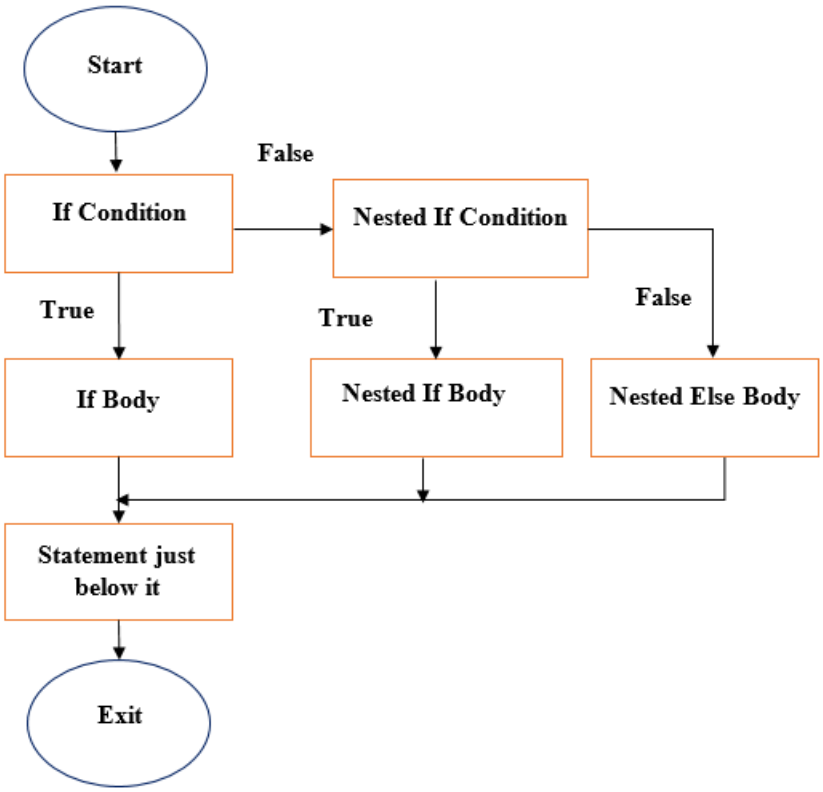


We execute the instruction instruction1 if the condition in parenthesis is verified, otherwise, we execute the instruction instruction2. We can also write more complex versions with multiple

conditions as follows (called nested if-else statements):

```
1. if (condition1)
2. {
3. instruction1;
4. }
5. else if (condition2)
6. {
7. instruction2;
8. }
9. else if (condition3)
10. {
11. instruction3;
12. }
13. else
14. {
15. by default instruction;
16. }
```

Figure 3.3: The flowchart of the nested If statement execution.



If condition1 is verified, we execute instruction1, If condition2 is verified, we execute instruction2, if condition3 is verified, we execute instruction3, otherwise, if none of the above conditions are true, the instruction by default is executed.

Before going ahead with some examples, we need to first present the operators allowing to perform tests, most often these are logical operators:

Operator	Signification
==	This is the equivalence test
!=	different (non-equivalence)
>	superior
>=	greater than or equal to
<	inferior
>=	lower than or equal
& &	logical and
||	logical or
!	logical not

Example

```
1. if (a ==b)
2. {
3. printf ("a is equal to b \ n");
4. }
```

If the value of a is equal to the value of b then we display "a is equal to b".
ATTENTION: Do not confuse a==b with a=b. a==b is a comparison that verifies if a and b are equal, while a=b assigns the value of b to a.

Example

```
1. if (!(a ==b))
2. {
3. printf ("a is not equal to b \ n");
4. }
```

Because of the NOT (!) Which reverses the result of the expression (a == b), we display the sentence "the variables a and b are not equal "if the expression a == b is not true, in other words if a and b are not equal.

Example

```
1. if ((a ==b) or (a<b))
2. {
3. printf ("a is not greater than b \ n");
4. }
```

If a is equal to b OR a is less than b, we display the sentence "a is not greater than b.

Example

```
1. if (a ==b)
2. {
3. printf ("a is equal to b \ n");
4. }
5. else
6. {
7. printf ("a is not equal to b \ n");
8. }
```

If a is equal to b, we display the sentence "a is equal to b" otherwise we display "a is not equal to b".

Example

```
1. if (a > b)
2. {
3. printf ("a is greater than b \ n");
4. }
5. else if (a == b)
6. {
7. printf ("a is equal to b \ n");
8. }
9. else
10. {
11. printf ("a is lower than b \ n");
12. }
```

If a is superior to b, we display "a is greater than b". If a is equal to b, we display the sentence "a is equal to b" otherwise we display "a is not equal to b".

3.2 Selection Statement: The switch-case Statement

To avoid the nesting of if statements, C has a selection statement that allows exploring several cases at the same time: this is the switch-case statement. Its syntax is as follows:

```
1. switch (variable)
2. case value1:
3. instruction1-0;
4. instruction1-1;
5. break;
6. case value2:
7. instruction2-0;
8. instruction2-1;
9. break;
10. ...
11. default: by default instructions;
```

If the "variable" takes the value value1 then we execute the instructions instruction1-0 and instruction1-1, if it takes the value value2, the instructions instruction2-0 and instruction12-1 are executed, etc. If none of the above values match the variable), then we execute by default instructions.

ATTENTION: The break statement should be present at the end of every case. If it were not so, the execution would continue into the code segment of the next case without even checking it. There is no break after default because after the default case, the control will get transferred to the next statement immediately after the switch.

Example

A C program to print the days of the week.

```
1. #include<stdio.h>
2. int main( )
3. {
4. int day;
5. printf ("nEnter the number of the day:");
6. scanf("%d",& day);
7. switch(day)
8. {
9. case 1:
10. printf("Sunday");
11. break;
12. case 2:
13. printf("Monday");
14. break;
15. case 3:
16. printf("Tuesday");
17. break;
```

```
18. case 4:
19. printf("Wednesday");
20. break;
21. case 5:
22. printf("Thursday");
23. break;
24. case 6:
25. printf("Friday");
26. break;
27. default:
28. printf("Invalid choice");
29. }
30. return 0;
31. }
```

3.3 Exercises

3.3.1 Exercise 1

Write a C program that allows checking if an entered number is positive or negative.

Solution

```
1. #include<stdio.h>
2. int main( )
3. {
4. int a;
5. printf("Enter a number:");
6. scanf("% d", & a);
7. if(a>0)
8. {
9. printf( "The number %d is positive.",a);
10. }
11. else
12. {
13. printf("The number %d is negative.",a);
14. }
15. return 0;
16. }
```

3.3.2 Exercise 2

Write a C program that:

1. Initializes 2 variables a and b (or ask the user to enter the values).
2. Declares an integer variable named nbmin.
3. Stores the minimum value of a and b in nbmin using if-else statement.
4. Displays the value of nbmin.

Solution

```
1. #include<stdio.h>
2. int main( )
3. {
4. int a=18;
5. int b=42;
6. int nbmin;
7. if(a <= b)
8. {
9. nbmin= a;
10. }
```

```
11. else
12. {
13. nbmin= b;
14. }
15. printf (" The min is %d:" nbmin);
16. return 0;
17. }
```

3.3.3 Exercise 3

Write a C program that displays a minimum of three numbers.

Solution

```
1. #include<stdio.h>
2. int main( )
3. {
4. int a=18;
5. int b=42;
6. int c=10;
7. int nbmin;
8. if(a <= b)
9. if(a <= c)
10. {
11. nbmin= a;
12. }
13. else
14. {
15. nbmin= c;
16. }
17. } else
18. {
19. if(b <= c)
20. {
21. nbmin= b;
22. }
23. else
24. {
25. nbmin= c;
26. }
27. }
28. printf (" The min is %d:" nbmin);
29. return 0;
30. }
```

3.3.4 Exercise 4

Write a C program that compares two strings and check whether they are equal or not.

Solution

```
1. #include<stdio.h>
2. #include <string.h>
3. int main( )
4. {
5. char a[20] , b[20];
6. printf("Enter the first string:");
7. scanf("% s",a);
8. printf("Enter the second string:");
9. scanf("% s",b);
10. if((strcmp(a,b)==0))
11. {
12. printf("Strings are the same");
13. }
14. else
15. {
16. printf("Strings are different");
17. }
18. return 0;
19. }
```

3.3.5 Exercise 5

Use a switch-case statement to write a C program that asks the user to type a number and then writes it in a full letter on the screen. For example, if the user types the number 9, the program will display nine. Note: you will only deal with digits and not with numbers outside the interval [0–9].

Solution

```
1. #include<stdio.h>
2. int main( )
3. {
4. int nb;
5. printf ("Enter a digit between 0 and 9:");
6. scanf("%d",& nb);
7. switch(nb)
8. {
9. case 0:
10. printf("Zero");
11. break;
```

```
12. case 1:
13. printf("One");
14. break;
15. case 2:
16. printf("Two");
17. break;
18. case 3:
19. printf("Three");
20. break;
21. case 4:
22. printf("Four");
23. break;
24. case 5:
25. printf("Five");
26. break;
27. case 6:
28. printf("Six");
29. break;
30. case 7:
31. printf("Seven");
32. break;
33. case 8:
34. printf("Eight");
35. break;
36. case 9:
37. printf("Nine");
38. break;
39. default:
40. printf("Error, you did not type a digit");
41. }
42. return 0;
43. }
```

3.3.6 Exercise 6

Write a C program to input electricity unit charge and calculate the total electricity bill according to the given condition:

- For first 50 units Rs. 0.50/unit.
- For next 100 units Rs. 0.75/unit.
- For next 100 units Rs. 1.20/unit.
- For unit above 250 Rs. 1.50/unit.
- An additional surcharge of 20% is added to the bill.

Solution

```
#include<stdio.h>
int main( )
{
int unit;
float amt, total_amt, sur_charge;
printf("Enter total units consumed: ");
scanf("%d",&unit);
if(unit <= 50)
{
amt = unit * 0.50;
}
else if(unit <= 150)
{
amt = 25 + ((unit-50) * 0.75);
}
else if(unit <= 250)
{
amt = 100 + ((unit-150) * 1.20);
}
else
{
amt = 220 + ((unit-250) * 1.50);
}
sur_charge = amt * 0.20;
total_amt = amt + sur_charge;
}
printf("Electricity Bill = Rs. %.2f", total_amt);
return 0;
}
```

CHAPTER

4

ITERATION STATEMENTS

An iteration Statement, or loop, is a structure that allows repeating the same instructions several times. Similar to the condition structure, there are several ways to make loops. Ultimately, this is like doing the same thing: repeating the same instructions several times. There are three common types of C iteration Statements:

- While
- Do...while
- For

In all cases, the diagram is the same (Figure 4.1).

Figure 4.1: The Execution Process of Iteration Statements .

Here's what happens in order during iteration statements Execution :

- The computer reads the instructions from top to bottom (as usual).

- Then, once arrived at the end of the loop, it starts again at the first instruction.
- It then resumes reading the instructions from top to bottom ...
- ... and it starts again at the beginning of the loop.

The problem with this statement is that if you don't stop it, the computer can repeat the endless instructions! The computer does what it is told to do ... It could very well get stuck in an infinite loop, it is also one of the many fears of programmers.
And that's where we find ... the conditions! When we create a loop, we always indicate a condition. This condition will mean "Repeat the loop as long as this condition is true".

4.1 While

The while statement evaluates a control expression before each execution of the loop body. If the control expression is true, the loop body is executed. If the control expression is false, the while statement terminates. The while statement has the following syntax:

```
1. while (/* Condition */)
2. {
3. // Instructions to repeat
4. }
```

Example

In this example, we do a simple test: we will ask the user to type in the number 47. As long as he has not typed the number 47, we ask him again for the number. The program will only be able to stop if the user types the number 47.

```
1. int enterednumber = 0 ;
2. While (enterednumber ! = 47)
3. {
4. printf ("Enter the number 47! \ n");
5. Scanf ("%d,&enterednumber");
6. }
```

Now here is the test we did. Note that we are on purpose to make mistakes 3 times before typing the right number:

- Enter the number 47! 10
- Enter the number 47! 27
- Enter the number 47! 40
- Enter the number 47! 47

The program stopped after typing the number 47.

Now let's try to do something more interesting: we want our loop to be repeated a number of times. To do this, we are going to create a counter variable that will be equal to 0 at the start of the program and that we will increment as we go. We use here the increment

operator which consists of adding 1 to the variable by doing variable ++.

Example

```
1. Int counter = 0 ;
2. While (counter < 5)
3. {
4. printf ("Hello Newbie! \ n");
5. counter++;
6. }
```

The result after execution is:

- Hello newbie!
- Hello newbie!
- Hello newbie!
- Hello newbie!
- Hello newbie

This code repeats the display of "Hello Newbie!" 5 times".

- At the start, we have a counter variable initialized to 0. It is therefore equal to 0 at the start of the program.
- We then go to the while loop. As the counter is 0 at the start, we enter the loop.
- We display the sentence "Hello Newbie!" " Via a printf.
- The value of the counter variable is incremented, using counter ++. The counter was 0, it is now 1.
- We arrive at the end of the loop (closing brace): we therefore start again at the beginning, at the level of the while.
- We repeat the while test: "Is the counter still less than 5?" ". The answer is yes, the counter is equal to 1! So we start the loop instructions again.

4.2 Do...while

This type of loop is very similar to while, although a little less commonly used. The only thing that changes from while is the position of the condition. Instead of being at the start of the loop, the condition is at the end. Its syntax is given below:

```
1. do
2. {
3. // Instructions to repeat
4. } while (condition)
```

Example

```
1. Int counter = 0 ;
2. do
3. {
4. printf ("Hello Newbie! \ n");
5. counter++;
6. } while (counter<5)
```

What does it change ? It's very simple: the while loop could never be executed if the condition is false from the start. For example, if we had initialized the counter to 10, the condition would have been false from the start and we would never have entered the loop.
For the do... while loop, it's different: this loop will always execute at least once. Indeed, the test is done at the end as you can see. If we initialize the counter to 10, the loop will run once.

4.3 For

In theory, the while loop allows us to perform all the loops we want. However, just like the switch for the conditions, it is in some cases useful to have another more "condensed" loop system, faster to write. For loops are widely used in programming and they're just another way to do a while loop. The syntax of the loop For is given below:

- for (expression-1(opt) ; expression-2(opt) ; expression-3(opt))
- {
- Instructions;
- }

Example

Here is the equivalence of the code written in section 4.1 with the for loop

```
1. Int counter; for ( counter=0 ; counter<5 ; counter++)
2. {
3. printf ("Hello Newbie! \ n");
4. }
```

What are the differences between the while and for loop?

- You will notice that we did not initialize the counter variable to 0 as soon as it was declared (but we could have done it).
- There are many expressions in the parentheses after the for.
- There is no longer a counter++; in the loop.

For what is in the parentheses, where there lies the whole point of the for a loop. There are three condensed instructions, each separated by a semicolon.

- The first is **initialization**: this first instruction is used to prepare the counter variable. In this case, it is initialized with 0.
- The second is the **condition**: as for the while loop, it is the condition that says if the loop must be repeated or not. As long as the condition is true, the for loop continues.
- Finally, there is the **increment**: this last instruction is executed at the end of each loop turn to update the counter variable. Most of the time we will do an incrementation, but we can also do a decrementation (variable--) or any other operation (variable + = 2; to advance from 2 to 2 for example).

4.4 Exercises

4.4.1 Exercise 1

Testing the WHILE statement: Write a C program that:

1. Loops 10 times by displaying the i value incremented at each iteration.
2. Display the value of i after the last loop.

Solution

```
1. #include <stdio.h>
2. int main ()
3. {
4. int i = 0;
5. while (i < 10) {
6. printf ("iteration %d \ n", i);
7. i = i + 1;
8. }
9. printf ("The value i after the loop : %d \ n", i) ;
10. return 0 ;
11. }
```

4.4.2 Exercise 2

Testing the Do...while statement: with the same exercise in 4.4.1:

Solution

```
1. #include <stdio.h>
2. int main ()
3. {
4. int i = 0;
5. do {
6. printf ("iteration %d \ n", i);
7. i = i + 1;
8. }
9. while (i < 10)
10. printf ("The value i after the loop : %d \ n", i) ;
11. return 0 ;
12. }
```

4.4.3 Exercise 3

Write a C program to calculate the factorial of a number.

Solution

```
1. #include <stdio.h>
2. int main ()
3. {
4. int nbr, i, f = 1;
5. printf ("Enter a number you want to calculate its factorial %d \ n");
6. scanf(("%d", nbr));
7. for (i = 1; i <= nbr; i++)
8. f = f * i;
9. printf ("Factorielle de %d = %d \ n", nbr, f);
10. return 0 ;
11. }
```

4.4.4 Exercise 4

Write a C program to find the sum of the first 10 natural numbers.
Example:
Expected Output :
The first 10 natural numbers are :
1 2 3 4 5 6 7 8 9 10.
The Sum is : 55

Solution

```
1. #include <stdio.h>
2. int main ()
3. {
4. int i;
5. printf("The first 10 natural numbers are:\ n");
6. for (i=1;i<=10;i++)
7. {
8. printf("%d ",i);
9. }
10. return 0 ;
11. }
```

4.4.5 Exercise 5

Write a C program to print all odd numbers from 1 to n.
Example:
Input:
Input upper limit: 10
Output:
Odd numbers between 1 to 10: 1, 3, 5, 7, 9

Solution

```
1. #include <stdio.h>
2. int main ()
3. {
4. int i, n;
5. printf("Print odd numbers till: ");
6. scanf("%d", n);
7. printf("All odd numbers from 1 to %d are: \ n", n);
8. for(i=1; i<=n; i++)
9. {
10. if(i%2!=0)
11. {
12. printf("
13. }
14. }
15. return 0 ;
16. }
```

4.4.6 Exercise 6

Write a C program to find all factors of a number.
Exemple:
Input:
Input number: 12
Output:
Factors of 12: 1, 2, 3, 4, 6, 12.

Solution

```
1. #include <stdio.h>
2. int main ()
3. {
4. int i, num;
5. printf("Enter any number to find its factor: ");
6. scanf("%d", &num);
7. printf("All factors of %d are: \ n", num);
8. for(i=1; i<=num; i++)
9. {
10. if(num % i == 0)
11. {
12. printf("%d, ",i);
13. }
14. }
15. return 0 ;
16. }
```

4.4.7 Exercise 7

Write a C program to check whether a number is a prime number or not.
Exemple:
Input:
Input any number: 17
Output:
17 is prime number.

Solution

```
1. #include <stdio.h>
2. int main ()
3. {
4. int i, num, isPrime;
5. isPrime = 1;
6. printf("Enter any number to check prime: ");
7. scanf("%d", &num);
8. for(i=2; i<=num/2; i++)
9. {
10. if(num%i==0)
11. {
12. isPrime = 0;
13. break;
14. }
15. }
16. if(isPrime == 1  num > 1)
17. {
18. printf("%d is prime number", num);
19. }
20. else
21. {
22. printf("%d is composite number", num);
23. }
24. return 0 ;
25. }
```

CHAPTER

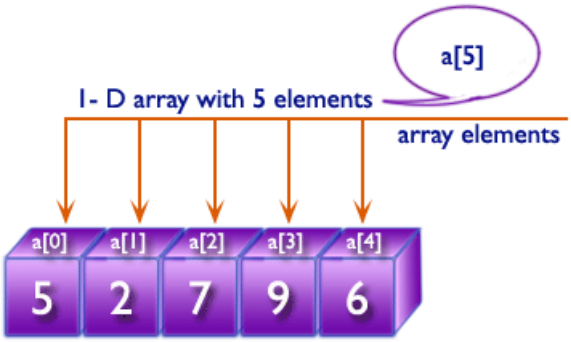
5

ARRAYS

5.1 Introduction

An array (also named a 1-D array)is series of elements of one data type. During the declaration of an array, the number of elements, what the type is for, these elements are set. Array elements can have the same types as ordinary variables. The declaration of an array has the following form:
Type [] arrayName; where:
Type: is the type of the elements of the array to which the variable we are declaring will refer.
arrayName: is the name of the array variable we are declaring.

Figure 5.1: An example of a 1-D array .



Consider the following example of array declarations:

```
1. /* some array declarations */
2. int main(void)
3. {
4. float candy[365]; /* array of 365 floats */
5. char code[12]; /* array of 12 chars */
6. int states[50]; /* array of 50 ints */
7. ...
8. }
```

The brackets ([]) identify candy, code and states as arrays, and the number enclosed in the brackets indicates the number of elements in the array. To access elements in an array, an index is used. The index numbering starts with 0. Thus, candy[0] is the first element of the candy array, and candy[364] is the 365th and last element.

5.2 Array Initialization

Arrays are often used to store data needed for a program. For example, a 12-element array can store the number of days in each month. An array needs to be initialized before further computations.
The syntax of declaration of an array is given below for an example:

- int main(void)
- {
- int powers[8] = { 1,2,4,6,8,16,32,64 };
- }

The array is initialized using a comma-separated list of values enclosed in braces. The first element (powers[0]) is assigned the value 1, and so on.

Example

This example prints the number of days per month. :

```
1. #include<stdio.h>
2. int main ()
3. {
4. int days [12] = {31,28,31,30,31,30,31,31,30,31,30,31 };
5. int index;
6. for (index = 0; index < 12; index++)
7. printf("Month %d has %2d days.", index +1, days[index]);
8. return 0;
9. }
```

The output of the program looks like this:
Month 1 has 31 days.
Month 2 has 28 days.

```
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

5.3 Assigning Array Values

After an array has been declared, values can be assigned to array members by using an array index. For example, the following fragment assigns even numbers to an array:

Example

```
1. #include<stdio.h>
2. int main ()
3. {
4. int counter, evens[50];;
5. for (counter = 0; counter < 50; counter++)
6. evens[counter] = 2 * counter;
7. }
```

5.4 Array Bounds

An important thing to consider durig using arrays is that the array indices are within bounds; that is, you have to make sure they have values valid for the array. For example, suppose we have the following declaration:

```
int doofi[20];
```

Then you should make sure that the program uses indices only in the range 0 through 19, because the compiler is not required to check for you. Not checking bounds in C programs is a part of its philosophy that involves trusting the programmer. It allows the program to run faster. The compiler can not necessarily catch all index errors because the value of an index might not be determined until after the resulting program begins execution. Therefore, to be safe, the compiler would have to add extra code to check the value of each index during runtime, and that would slow things down.

5.5 Specifying an Array Size

There is a specified way to define an array size in C language. When declaring an array, a constant integer expression between the brackets should be used. A constant integer expression is one formed from integer constants. The value of the expression must be greater than 0.

Example

1. float a1[5]; // yes
2. float a2[5*2 + 1]; // yes
3. float a4[-4]; // no, size must be > 0
4. float a5[0]; // no, size must be > 0
5. float a6[2.5]; // no, size must be an integer
6. float a7[(int)2.5]; // yes, typecast float to int constant

5.6 Multidimensional Arrays

5.6.1 Introduction

A multidimensional array can be defined in simple words as an array of arrays. Data in multidimensional arrays are stored in tabular form (in row-major order). The general form of declaring N-dimensional arrays is given below:
data_type array_name [size1][size2]....[sizeN]; where:
data_type: Type of data to be stored in the array.
array_name : Name of the array.
size1, size2,... ,sizeN: Sizes of the dimensions.

Example

Two dimensional array:

- int two_d[10][20];

Three dimensional array:

- int three_d[10][20][30];

5.6.2 Two-Dimensional Arrays

The two dimensional (2D) array also known as matrix, is the simplest form of a multidimensional array. It can be represented as a table of rows and columns. The basic form of declaring a two-dimensional array of size x, y is as follows:

data_type array_name[x][y];
where:
array_name is the name of the array.
data_type is the type of data to be stored.

Figure 5.2: An example of a 2-D array.

5.6.3 Initializing a Two-Dimensional Array

Initializing a two-dimensional array builds on the technique for initializing a one-dimensional array. For example:

- int x[3][4] = { {0,1,2,3 }, {4,5,6,7 }, {8,9,10,11 } };

Here:

- int is the type of the elements stored in the 2-D array named x.
- 3 is the number of rows in the 2-D array.
- 4 is the number of columns in the 2-D array.
- The first element of the 2-D array is x[0][0] with the value 0.
- The second element of the 2-D array is x[0][1] with the value 1.

5.6.4 Assigning 2-D Array Values

The assigning of values to a 2-D array is similar to the one of a 1-D array. The only difference is that here we use two array indexes instead of one.

Example

```
1. int main()
2. {
3. int disp[2][3];
4. int i, j;
5. for(i=0; i<2; i++)
6. {
7. for(j=0;j<3;j++)
```

```
8. {
9. printf("Enter value for disp[%d][%d]:", i, j);
10. scanf("%d", &disp[i][j]);
11. }
12. }
13. return 0;
14. }
```

The output:

- Enter value for disp[0][0]:1
- Enter value for disp[0][1]:2
- Enter value for disp[0][2]:3
- Enter value for disp[1][0]:4
- Enter value for disp[1][1]:5
- Enter value for disp[1][2]:6

5.7 Exercises

5.7.1 Exercise 1

Write a program in C to store elements in an array and print it.
Test Data :
Input 10 elements in the array :
element - 0 : 1
element - 1 : 1
element - 2 : 2 .....
Expected Output :
Elements in array are: 1 1 2 3 4 5 6 7 8 9.

Solution

```
1. #include <stdio.h>
2. int void main()
3. {
4. int arr[10];
5. int i;
6. printf("Read and Print elements of an array:");
7. printf("-------------------------------------");
8. printf("Input 10 elements in the array :");
9. for(i=0; i<10; i++)
10. {
11. printf("element - %d : ",i);
12. scanf("%d", &arr[i]);
13. }
14. printf("Elements in array are: ");
15. for(i=0; i<10; i++)
16. {
17. printf("%d ", arr[i]);
18. }
19. }
```

5.7.2 Exercise 2

Write a program in C to find the sum of all elements of the array.
Test Data :
Input the number of elements to be stored in the array :3
Input 3 elements in the array :
element - 0 : 2
element - 1 : 5
element - 2 : 8
Expected Output :
Sum of all elements stored in the array is : 15

Solution

```
1. #include <stdio.h>
2. int main()
3. {
4. int a[100];
5. int i, n, sum=0;
6. printf("Find sum of all elements of array:");
7. printf("--------------------------------------");
8. printf("Input the number of elements to be stored in the array :");
9. scanf("%d",&n);
10. printf("Input %d elements in the array :",n);
11. for(i=0;i<n;i++)
12. {
13. printf("element - %d : ",i);
14. scanf("%d",&a[i]);
15. }
16. for(i=0; i<n; i++)
17. {
18. sum += a[i];
19. }
20. printf("Sum of all elements stored in the array is : %d", sum);
21. }
```

5.7.3 Exercise 3

Write a program in C to read n number of values in an array and display it in reverse order.
Test Data :
Input the number of elements to store in the array :3
Input 3 number of elements in the array :
element - 0 : 2
element - 1 : 5
element - 2 : 7
Expected Output :
The values stored into the array are :
2 5 7
The values stored into the array in reverse are :
7 5 2

Solution

```
1. #include <stdio.h>
2. int main()
3. {
4. int i, n, a[100];
5. printf("Read n number of values in an array and display it in reverse order:");
```

```
printf("------------------------------------------------------------------------");
printf("Input the number of elements to store in the array :");
scanf("%d",&n);
printf("Input %d number of elements in the array :",n);
for(i=0;i<n;i++)
{
printf("element - %d : ",i);
scanf("%d",&a[i]);
}
printf("values store into the array are :");
for(i=0;i<n;i++)
{
printf("% 5d",i]);
}
printf("The values store into the array in reverse are :");
for(i=n-1;i>=0;i--)
{
printf("% 5d",a[i]);
}
}
```

Solution

```
#include <stdio.h>
int main()
{
int arr1[100], arr2[100];
int i, n;
printf("Copy the elements one array into another array :");
printf("-----------------------------------------------");
printf("Input the number of elements to be stored in the array :");
scanf("%d",&n);
printf("Input %d elements in the array :",n);
for(i=0;i<n;i++)
{
printf("element - %d : ",i);
scanf("%d",&arr1[i]);
}
for(i=0; i<n; i++)
{
arr2[i] = arr1[i];
}
printf("The elements copied into the second array are :");
for(i=0; i<n; i++)
{
printf("% 5d", arr2[i]);
```

```
}
}
```

5.7.4 Exercise 5

Write a program in C to count a total number of duplicate elements in an array.
Test Data :
Input the number of elements to be stored in the array :3
Input 3 elements in the array :
element - 0 : 5
element - 1 : 1
element - 2 : 1
Expected Output :
Total number of duplicate elements found in the array is : 1

Solution

```
#include <stdio.h>
int main()
{
int arr1[100];
int arr2[100];
int arr3[100];
int n,mm=1,ctr=0;
int i, j;
printf("Count total number of duplicate elements in an array:");
printf("------------------------------------------------------");
printf("Input the number of elements to be stored in the array :");
scanf("%d",&n);
printf("Input %d elements in the array :",n);
for(i=0;i<n;i++)
{
printf("element - %d : ",i);
scanf("%d",&arr1[i]);
}
/*----------------- copy in other array ----------------------*/
for(i=0;i<n; i++)
{
arr2[i]=arr1[i];
arr3[i]=0;
}
/*---------------- mark the elements are duplicate ----------*/
for(i=0;i<n; i++)
{
for(j=0;j<n;j++)
{
```

```
30. if(arr1[i]==arr2[j])
31. {
32. arr3[j]=mm;
33. mm++;
34. }
35. }
36. mm=1;
37. }
38. /*---------- Prints the array ----------*/
39. for(i=0; i<n; i++)
40. {
41. if(arr3[i]==2)ctr++;
42. }
43. printf("The total number of duplicate elements found in the array is: %d", ctr);
44. }
```