PART 1

### Programming

Programming is a required skill today, just like English and Mathematics. It teaches a variety of skills that are important in all kinds of professions such as critical reading, analytical thinking, creative synthesis, and attention to detail.

Programming also provides immediate feedback, leading to exploration, experimentation, and self-evaluation.

### What is programming?

Process of coming up with computer programs.

But what is a program?

A PROGRAM is a set of instructions that a computer can follow to accomplish specific task.

Programming is the process of formulating instructions that a computer can follow to accomplish specific task.

### 1.2. Program Development Process

The program development process can be viewed in a number of steps as follows:

* *Understanding the Specifications:* The specification for the problem must be clear and unambiguous.
* *Role of the System Analyst:*  The system analyst determines the nature of the inputs and outputs needed, and provides the input and output layouts along with the processing requirements for the programmers.
* *Problem Analysis:* The program to be developed might be simple or complex, in accordance with the nature of the task.
* *Solution Design:* Involves testing the various conditions, taking alternate paths in the procedure depending on the outcome of the test, and determining the sequence of the steps. There are two ways of accomplishing this task:
	+ *Algorithm:* A detailed description of the sequence of steps to be followed.
	+ *Flowchart:* A pictorial representation of the program, which is easier to visualize due to its graphical organization.
* *Testing and Debugging:* A computer program may not run properly the first time due to the possibility of logical and/or syntactic errors. These errors are referred as bugs. The process of removing these bugs is called debugging. The errors in the program are checked by testing it at various stages.
	+ *Use of Test Data:* Test data should be prepared in such a way that the programs run by using this data. It should be prepared so as to include all the conditions that the program is expected to test.
	+ *Use of Diagnostic Tools:* There are diagnostic aids, which programmers may deploy to detect errors when a program fails to run correctly. The diagnostic procedures may vary from one language to another. The diagnostic tools provide, in general, a method for testing the execution of a program at each step or each time the program follows a particular path. Breakpoints may also be used in code for debugging purposes.
* *Installation:* The tested software is installed at the client end and finally tested with real-time data.

### 1.3. Algorithms

An algorithm is a well-defined computational procedure (sequence of computational steps). It takes a set of values as its input and produces a set of values as its output. It can be viewed as a utility for solving a computational problem.

There are a number of important features that must be satisfied by any algorithm:

* *Finiteness:* This imposes that the algorithm must terminate after executing a finite number of steps, i.e., it cannot run infinitely.
* *Definiteness:* An algorithm. For a computational procedure to qualify as an algorithm, each step in the procedure must have a precise definition.
* *Input and Output:* An algorithm has a domain of values, which initialize the procedures. These are called the input values to the algorithm. It must also generate a set of result values called output values. Thus, an algorithm can be viewed as a function or a transformation that maps input values to output values.
* *Effectiveness:* The operations specified in an algorithm must be basic enough in nature to be performed by someone using a pen and a paper.

**Example 1.**

Design an algorithm for adding the test scores given below

26, 49, 98, 87, 62, 75, 41, 20, 100, 15

**Algorithm Solution**

1. Start

2. Sum = 0

3. Get the first test score

4. Add first test score to sum

5. Get the second test score

6. Add to sum

7. Get the third test score

8. Add to sum

9. Get the Forth test score

10. Add to sum

11. Get the fifth test score

12. Add to sum

13. Get the sixth test score

14. Add to sum

15. Output the sum

16. Stop

### 1.4. Flowcharts

A flowchart is a graphic/pictorial representation of the steps necessary to solve a problem, accomplish a task, complete a process, or illustrate the components of a system.

The various flowchart symbols are explained as follows:

* *Oval/Rounded Rectangle:* These symbols indicate the start or end of the program as indicated by the text written inside the symbol.
* *Rectangle:* Indicates a processing step where the step performed is indicated inside the rectangle.



##### Fig.1.1. Common Flowchart Symbols

* *Diamond:* Indicates that a condition check is to be performed. The data under test and the test performed are indicated inside the diamond.
* *Parallelogram:* Indicates I/O operations.
* *Arrow:* Indicates direction of program control flow.

A program flowchart or flowchart illustrates the detailed sequence of steps undertaken by a program. It can be used to

* visualize the logic and sequence of steps in an operation,
* experiment with various programming approaches, and
* keep track of all processing steps.



#####

##### Fig. 1.2 Flowchart for the Algorithm adding the test scores

### 1.5. Programming Language Classifications

A language is a medium for communication. The languages we speak are called natural languages. A programming language is a subset of the set of natural languages. It contains all the symbols, characters, and usage rules that permit a human being to communicate with computers. A variety of programming languages have been invented over the years of computer history. However, every programming language must accept certain types of written instructions that enable a computer system to perform a number of familiar operations.

In other words, every programming language must have instructions that fall under the following categories:

* *Input/output Instructions:* A program needs input data from the external world with which it performs operations on the input data, and generates output. Input/output instructions, provide details on the type of input or output operations to be performed, and the storage locations to be used during the operations.
* *Arithmetic Instructions:* A program might be required to perform arithmetic operations on the data in the program. Arithmetic instructions are provided for the requirement. These perform the arithmetic operations of addition, subtraction, multiplication, division, etc.
* *Logical/Comparison Instructions:* They are used to compare two values to check whether the values satisfy a given condition or state.
* *Storage/Retrieval and Movement Instructions:* These are used to store, retrieve, and move data during processing. Data may be copied from one storage location to another and retrieved as required.
* *Control Instructions:* These are selection and loop constructs, which aid in out-of-sequence program flow.

Although all programming languages have an instruction set that permits these familiar operations to be performed, a marked difference is found between the symbols and syntax used in machine languages, assembly languages, and high-level languages.



#### Evolution of programming language

#### 1.5.1. Machine Language

#### Machine language is made up of streams of 1s and 0s. Programs written as series of 1s and 0s is said to be in machine language. Machine language is the only language understood by computer hardware. Writing programs in machine language is difficult and cumbersome –instructions are difficult to remember.

#### Each computer has its own machine language. Machine language is also referred to as First Generation programming language.

####

#### 1.5.2. Assembly Language

Assembly (symbolic) language was developed in 1950s to reduce programmers‟ burden –by Grace Murray Hopper a US naval officer. She developed the concept of a special computer program that would convert programs into machine language. The program mirrored the machine languages using symbols, or mnemonics, to represent the various machine instructions. An assembler translates symbolic code into machine language –thus Assembly Language.

Some applications are developed by coding in *assembly language*. This type of application software is most efficient for processing data. However, since a particular flavour of the assembly language is designed for a particular architecture, the type of assembly language understood by a computer depends upon the underlying architecture of the microprocessor used.

For example, if a Zilog® microprocessor is used, then the machine language understood is Z-80. On the other hand, a flavour of the Intel®assembly language is used for a Pentium® microprocessor which differs from the Z-80. Thus, assembly languages are architecture-dependent.





#### 1.5.3. High-level Languages

Given the inability of assembly language to adapt across different architectural platforms and given the varied nature of real-world problems, a variety of computer languages, called *high-level languages,* were developed, each suited best to a model in a particular class of problems. High-level language programs are architecture-independent, easier to write, and provide easier maintenance and readability than their assembly-language counterparts. However, they require longer execution time compared to assembly-language programs.

Different high-level programming languages were introduced in the late 1950s to reduce the problems that arose in writing code in assembly and machine language. When the first high-level languages were developed, the longer translation and execution time were considered as serious limitations of the technique. Nonetheless, the following factors contributed to the popularity of high-level languages for application development:

* *Savings in Programming Time and Training:* High-level languages were easier to learn and understand. Thus, it took less time and effort to write an error-free program or to make corrections and code-revisions.
* *Increased Speed and Capacity of Hardware:* The third and fourth generations of computer hardware brought about a revolution in access time, memory and disk capacities. This caused the time overhead incurred by the use of high-level languages to be tolerable.
* *Increasing Complexities of Software:* With time, software systems grew more complex. This necessitated use of complex constructs each of which could represent several lines of assembly code and was provided by high-level languages.

Some popular high-level languages include FORTRAN (FORmula TRANslation), COBOL (COmmon Business Oriented Language), Pascal (after Blaise Pascal), C (refer to Section 1.11), and Ada (after Ada Byron, Countess of Lovelace – the world’s first computer programmer).

1.6 Programming Techniques

As stated earlier, the task of coding for a problem in a convenient programming language is performed only after extensive effort in the problem-solving stage. The following sections provide an insight into some commonly adopted programming techniques.

1.6.1. Bottom-up Design

Early programming techniques developed in the 1950s centered on problem-solving by using bottom-up design of the solution in which the extreme details of the programming solution were investigated first, as opposed to beginning with a breakdown by broad objectives. Each program was written in isolation to solve a particular sub-problem. The difficulty arose when the various sub-programs had to work together to produce the desired programs. Program logic was guided by the limitations of primary memory, and programs were designed with the objective of executing them as fast as possible. However, as application programs grew in size, several programmers worked together to solve them. Project teams were set up, consisting of several programmers and a project leader. However, programmers often switch jobs and might leave a company before a project is fully developed, thus require another programmer to continue the unfinished work midstream. This required formulation of a definite summary of how a problem is to be solved. This was not provided by the bottom-up approach to programming. Another approach was required.

1.6.2. Top-down Design

It allowed programs to be written in a more organized manner, i.e., in a structured manner, producing code that is easier to read, analyze, and modify later – if the need arose. With increasing demands for software efficiency and programming standardization, a changed approach saw the programmers examining the problem as a whole and outlining the major steps to solve the problem. Then the process was repeated and the steps thus obtained were broken down in finer details. This is the top-down programming approach and is used in structured programming.

1.6.3. Structured Programming

Structured programming is a methodology that is part of a renewed emphasis on software engineering, which involves the systematic design and development of software and the management of the software development process. Structured approach involves the use of methods such as top-down program design and a limited number of control structures in a program to create tightly structured modules of program code.

A ‘traditional’ flexible and creative environment provided to the programmer often results in complex and difficult-to-read programs requiring much testing before they are error-free. These also become costly to develop and maintain. Structured programming, on the other hand, emphasizes group responsibility for program development. It also brings in a standardization of program-design concepts and methods, which significantly reduces the program complexity.

1.6.4. Modular Design

Structured programming is used as a set of tools to improve program organization, facilitate problem solving, and make code easier to write and analyse. Using structured programming, the solution to the problem is divided into segments called modules.

* A *module* is a logically separable part of a program. It is a unit, discrete and identifiable with respect to compilation and loading. In terms of common programming language constructs, a module can be a macro, a function, a procedure (or subroutine), a process, or a package.
* Each module involves processing of data that are logically related. Modules are functional parts, which aid in processing. Ideally, each module works independent of other modules, although this is sometimes impossible.
* Modules are ranked by hierarchy and organized on the basis of importance. The lower the module on the structure organization plan, more is the detail given to the programming steps involved. The controlling module resides at the top level. It gives the view of the overall structure for the entire program. The system is designed to give more detail at each module level. A module is coded and tested, and then tested with other tested modules. This procedure makes program testing easier, since there is only one entry point and one exit point per module.

The modularization approach involves breaking a problem into a set of sub-problems, followed by breaking each sub-problem into a set of tasks, then breaking each task into a set of actions.

**Example**

Turn on a light bulb

-Sub-problem 1: locate bulb (one task, one action)

-Sub-problem 2: depress switch

**Example**

Given a list of students’ test scores, find the highest and lowest score and the average score.

-Sub-problem 1: read students’ scores

-Sub-problem 2: find highest score

-Sub-problem 3: find lowest score

Sub-problem 1 can be considered as one action and therefore needs no further refinement.

Sub-problems 2 and 3 however can be further divided into a group of actions.

### 1.7. Structured Programming Constructs

It implements modules (procedures / functions), each fulfilling some function and uses basic programming constructs -sequence, selection and iteration.

The most common techniques used in structured programming to solve all problems are called constructs. These constructs are also called *control structures*. Using three basic control structures, it is possible to write standardized programs, which are easy to read and understand.



##### Structured Programming Constructs

* *Sequence Structure:* Sequence refers to an instruction or a series of instructions that perform a required calculation or allow input or output of data. Since these steps are executed one after the other, there is no change in the flow logic. Figure (a) illustrates that program statements in function A will be executed before those for function B. In other words, we say that control ‘flows’ from function A to function B.
* *Selection Structure:* Selection refers to testing for a certain condition for data. There are only two possible answers to questions regarding data – true (yes) or false (no). One selection-technique variation is known as the IF-THEN-ELSE. The instructions that are to be executed when the condition is true follow the IF-THEN alternative. The instructions followed by the ELSE alternative represent what is to be executed when the condition is false. Figure (b) shows that if the condition is true, the control will flow to function B and its statements will be executed; if it is false, function A is executed. Another selection variation is the IF-THEN. It is used when some operation is to be done only when the condition is true.
* *Repetition Structure:* Repetition involves the use of a series of instructions that are repeated until a certain condition is met. Repetition involves the use of two variations – the *while* and the *do-while*. The *while* performs a function as long as a condition is true. On the other hand, *do-while* allows a function to be executed until the given condition is false. Another marked difference is that the *while* first tests the given condition and then executes the function, whereas *do-while* processes the function before checking the condition. These are illustrated in Figure 1.3.

### 1.8. Pseudo Code

Pseudo code is one of the tools that can be used to write a preliminary plan that can be developed into a computer program. Pseudo code is a generic way of describing an algorithm without the use of any specific programming language syntax.

**How to write**

An algorithm can be written in pseudo code using six (6) basic computer operations:

1. **A computer can receive information.**

Typical pseudo code instructions to receive information are:

Read name, Get name, Read number1, number2

1. **A computer can output (print) information.**

Typical pseudo code instructions are:

Print name

Write "The average is", ave

1. **A computer can perform arithmetic operation**

Typical pseudo code instructions:

Add number to total, or

Total = Total + Number

Ave = sum/total

1. **A computer can assign a value to a piece of data:**

e.g. To assign/give data an initial value:

Initialize total to zero

Set count to 0

To assign a computed value:

Total = Price + Tax

1. **A computer can compare two (2) pieces of information and select one of two** alternative actions.

Typical pseudo code e.g.

If number < 0 then

add 1 to neg\_number

else

add one to positive number

end-if

1. **A computer can repeat a group of actions.**

Typical pseudo code e.g.

Repeat until total = 50

read number

write number

add 1 to total

end-repeat

OR

while total < = 50 do:

read number

write number

end-while

**Algorithm Average**

This algorithm reads a list of numbers and computes their average.

Let: SUM be the total of the numbers read

COUNTER be the number of items in the list

AVE be the average of all the numbers

Set SUM to 0,

 COUNTER to 0,

AVE to 0 (i.e. initialize variables)

While there is data do:

Read number

COUNTER = COUNTER + 1

(i.e. add 1 to COUNTER, storing result in COUNTER)

SUM = SUM + number

(i.e. add number to SUM, storing result in SUM)

end-while

if COUNTER = 0 then

AVE = 0

else

AVE = SUM/ COUNTER

Stop.

### 1.9 State Diagram

A state diagram is a type of diagram used to describe the behaviour of systems. This behaviour is analyzed and represented in series of events that could occur in one or more possible states. State diagrams require that the system described is composed of a finite number of states.

**State Diagram Algorithm**

•Draw circles to represent the states given.

•For each of the states, scan across the corresponding row and draw an arrow to the destination state(s). There can be multiple arrows for an input character.

•Designate a state as the START STATE.

•Designate one or more states as ACCEPT STATE. This is also given in the formal definition.

|  |
| --- |
| State Transition Table |
| Input | 1 | 0 |
| S1 | S1 | S2 |
| S2 | S2 | S1 |



**Explanation**

From the state transition table given above, it is easy to see that if the machine is in **S1** and the next input is character **1**, the machine will stay in **S1**. If a character **0** arrives, the machine will transition to **S2** as can be seen from the second column. In the diagram this is denoted by the arrow from **S1** to **S2** labeled with a **0**.

State diagrams versus flowcharts

•A state machine performs actions in response to explicit events. In contrast, the flowchart does not need explicit events but rather transitions from node to node in its graph automatically upon completion of activities

•A state machine is idle when it sits in a state waiting for an event to occur. A flowchart is busy executing activities when it sits in a node.

•A flowchart describes the progression of some task from beginning to end (e.g., transforming source code input into object code output by a compiler). A state machine generally has no notion of such a progression but specifies a particular behaviour, rather than a stage of processing.

* Terminology Checklist

✓ Program

✓ Software

✓ Algorithm

✓ Assembly language

✓ High-level Language

✓ Flowchart

✓ Debugging

✓ Bottom-up design

✓ Top-down design

✓ Structured programming

✓ Modular design

✓Pseudo Code

✓State Diagram

Exercises

|  |  |
| --- | --- |
| 1. | Write an algorithm to find the maximum of three numbers *a*, *b*, and *c*. The output should be stored in ‘max’. |
| 2. | “An operating system never terminates – it is infinite.” Based on this statement, justify whether an operating system is an algorithm. |
| 3. | Trace Euclid’s algorithm for GCD (in Section 1.5) for inputs (32, 14), (12, 9). |
| 4. | Modify Euclid’s algorithm (in Section 1.5) to accommodate a check of the input variables so that the algorithm executes only on positive integer inputs and the divide by zero error is detected. |
| 5. | Write an algorithm that accepts a positive integer number, *n*, and calculate the factorial of the number. Reject inputs less than 1 or greater than 17. |
| 6. | Draw a flowchart to convert miles into kilometers, given that 1 km = 1.609 miles and M is the input value in miles and K is the output value in kilometers. |
| 7. | Write an algorithm that calculates all the elements of rows and columns of a square matrix and calculate the total of primary and secondary diagonal. |
| 8. | Write an algorithm that accepts two positive integer numbers n1 and n2 (n1 < n2), and print the prime numbers in between n1 and n2, both inclusive. |
| 9. | What is validation of an algorithm? |
| 10. | What are the advantages and disadvantages of flowcharts? |

PART 2

### 2.1 History of C Language

C has its origin in the early 1970, a project by AT&T Bell Labs, USA. Dennis Ritchie is credited with defining and creating C. Many others also influenced the development of the language. C is closely related to the UNIX operating system since it was developed along with it. Most of the UNIX commands are written in C. Today, C has been implemented on many different types of hardware and the language is used to tackle a large class of programming problems.

### 2.2. C Language Overview

Several features make C a very attractive programming language for coding solutions. These include:

* *Performance* features such as speed and efficiency of the executable program.
* Features relating to the *portability* of the programs. C is independent of any particular machine architecture – it is not bound to any hardware or system.
* C is a simple and a straightforward language, providing only low-level operations.
* Its *versatility* enables it to be used to solve problems in every application area.
* *Modularity* allows C to be ideal for large projects involving several programmers. It lets us break a large program into small manageable pieces, which can be reused in other programs.

C is both a low- and a high-level language.

It supports the low-level features (i.e., the bit-manipulation facilities of assembly) needed by the operating-system implementers and the compiler writers.

It also supports the high-level features of control and data structures characteristic of a procedural language. C is as flexible as a low-level language. On the other hand, C provides the tools for a structured program, as a high-level language does.

### 2.3 Installing and compiling your First Program

Practical session: refer to the Hello world exercise

#include<stdio.h>

int main()

{

 printf("Hello Student, !\n");

 return(0);

### }

### 2.4 Hello Student Program Description

###

* 1. **#include** is known as a preprocessor directive. What it does is tell the compiler to “include” text from another file, adding it right into your source code. Doing this avoids lots of little, annoying errors that would otherwise occur.
	2. **<stdio.h>**The STDIO.H file contains information about the Standard Input/Output functions required by most C programs. The H means “header”. The whole statement #include <stdio.h> tells the compiler to take text from the file STDIO.H and stick it into your source code before the source code is compiled.
	3. **int main** does two things. First, the int identifies the function main as an integer function, meaning that *main ()* must return an integer value when it’s done. Second, that line names the function main, which also identifies the first and primary function inside the program.
	4. Two empty parentheses follow the function name. Sometimes, items maybe in these parentheses.
	5. **{}** All functions in C have their contents encased by curly braces (**{}**). So, the function name comes first (main in Item 3), and then its contents — or the machine that performs the function’s job — is hugged by the curly braces.
	6. **printf** is the name of a C language function, It is written as printf(). Its job is to display information on the screen. (Because printers predated computer monitors, the commands that display information on the screen are called print commands.
	7. Like all C language functions, printf() has a set of parentheses. In the parentheses, you find text, or a “string” of characters. Everything between the double quote characters (“) is part of printf’s text string.
	8. An interesting part of the text string is \n. That’s the backslash character and a little n. What it represents is the character produced by pressing the Enter key, called a newline in C.
	9. The printf line, or statement, ends with a semicolon. The semicolon is C language punctuation — like a period in English. The semicolon tells the C compiler where one statement ends and another begins. Note that all statements require semicolons in C, even if only one statement is in a program or function.
	10. The second statement in ex\_1.C is the return command. This command sends the value 0 (zero) back to the operating system when the main () function is done. Returning a value is required as part of the main () function. Note that even thought is command is the last one in the program, this statement ends in a semicolon.

Others

* 1. **Scanf():** scanf() is a function like printf(). Its purpose is to read text from the keyboard.
	2. **Comment:** are explanatory remarks made within a program. When used carefully, comments can be very helpful in clarifying what the complete program is about, what a specific group of statements is meant to accomplish, or what one line is intended to do.

// comment

/\* comment \*/

### 2.5 Conversion Control Sequence

Example Sum

#include<stdio.h>

int main()

{

 printf("Sum of student score is %d \n", 6 + 14 +10 + 13 + 30);

 return(0);

}

This statement passes two arguments to the printf () function.

The first argument is the message Sum of student score is %d \n".The second argument is the value of the expression 6 + 14 +10 + 13 + 30.

The first argument passed to printf ( ) must always be a message. A message that also includes a **conversion control sequence**, such as %d. Conversion control sequences have a special meaning to the printf ( ) function. They tell the function what type of value is to be displayed and where to display it. A conversion control sequence always begins with a %symbol and ends with a conversion character (c, d, f, etc.).

Example Average

#include<stdio.h>

int main()

{

 printf("Average of student score is %d \n", (6 + 14 +10 + 13 + 30)/5);

 return (0);

}

### 2.6 Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program.

int myValue; // integer =4 byte

char response; // Character = 1 byte

string name; //

double yCoord; // Double = 8 byte

Example

Calculate area and circumference of a circle

**#include<stdio.h>**

**int main()**

**{**

 **float PI = 3.14159;**

 **float radius, area, circumference;**

 **printf("Enter the radius in cm: ");**

 **scanf\_s("%f", &radius);**

 **area = PI\*radius\*radius;**

 **circumference = 2.0\*PI\*radius;**

 **printf("The area is %f cm square.\n", area);**

 **printf("The circumference is %f cm.\n", circumference);**

 **return 0;**

**}**

**#include<stdio.h>**

**int main()**

**{**

 **int num1, num2, num3;**

 **float average;**

 **printf("Enter the first numbers");**

 **scanf\_s("%d", &num1);**

 **printf("Enter the second numbers");**

 **scanf\_s("%d", &num2);**

 **printf("Enter the third numbers");**

 **scanf\_s("%d", &num3);**

 **average = (num1 + num2 + num3) / 3.0;**

 **printf("Average is %f", average);**

 **return 0;**

**}**

**#include<stdio.h>**

**int main()**

**{**

 **int num1, num2, num3;**

 **float average;**

 **printf("Enter the three numbers");**

 **scanf\_s("%d %d %d", &num1, &num2, &num3);**

 **average = (num1 + num2 + num3) / 3.0;**

 **printf("Average is %f", average);**

 **return 0;**

**}**

Example

Input/output operate to display first name, surname and matric number.

**#include<stdio.h>**

**int main()**

**{**

 **char sname[20];**

 **char fname[20];**

 **char matric[10];**

 **printf("What is your surname?");**

 **scanf\_s("%s", sname);**

 **printf("What is your firstname?");**

 **scanf\_s("%s", fname);**

 **printf("What is your matric no?");**

 **scanf\_s("%s", matric);**

 **printf("My name is %s, surname is %s and matric number %s",sname,fname,matric);**

 **return 0;**

**}**

%s is the string placeholder

%i for integersplaceholder

%f for doubleplaceholder

### 2.7 Program Development

###

**Text Editor -** Text editor is used to edit plain text files, differ from word processors e.g. Microsoft Word. A .doc file in a text editor contains tools to change fonts, margins, and layout. Text editors, however, do not add formatting codes, which makes it easier to compile your code.

**Compiler** is a computer program that transforms human readable source code of another computer program into the machine readable code that a CPU can execute. It reads the instructions stored in the source code file, examines each instruction, and then translates the information into the machine code understood only by the computer’s microprocessor.

**Source Code** is the human readable instructions that a programmer writes.

**Objects** code is a portion of machine code that hasn't yet been linked into a complete program.

**Linker** links several object (and library) files to generate an executable file. Linker is a computer program that takes one or more object files generated by a compiler and combines them into a single executable program.

**Library** is a collection of implementations of behavior, written in terms of a programming language that has a well-defined interface by which the behavior is invoked.

**Executable file** causes a computer "to perform indicated tasks according to encoded instructions.

### 2.8 Data Types and Operations

**Identifiers** are a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore \_ followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, $, and % within identifiers. C is a case sensitive programming language. Thus, ABUAD and abuad are two different identifiers.

Example

mohd zara abc move\_name a\_123myname50 \_temp j a23b9 retVal

**Keywords** These are reserved words and may not be used as constant or variable or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| Auto | Else | Long | switch |
| Break | enum | Register | typedef |
| Case | extern | Return | union |
| Char | Float | Short | unsigned |
| Const | For | Signed | void |
| Continue | Goto | Sizeof | volatile |
| Default | If | Static | while |
| Do | Int | struct | \_Packed |
| Double |   |   |   |

### 2.8.1 Data Types

Data types refer to an extensive system used for declaring variables or functions of different types.

### Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges:

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| Char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| Int | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| Short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| Long | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |

### Floating-Point Types

Following table gives you details about standard floating-point types with storage sizes and value ranges and their precision:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Storage size** | **Value range** | **Precision** |
| Float | 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| Double | 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name. Variable determines how much space it occupies in storage. The type of a Variable Declaration in C:

|  |  |
| --- | --- |
| **Type** | **Description** |
| Char | Typically a single octet (one byte). This is an integer type. |
| Int | The most natural size of integer for the machine. |
| Float | A single-precision floating point value. |
| Double | A double-precision floating point value. |
| Void | Represents the absence of type. |

Example

**#include<stdio.h>**

**int main()**

**{**

 **/\* variable definition: \*/**

 **int a, b;**

 **int c;**

 **float f;**

 **/\* actual initialization \*/**

 **a = 10;**

 **b = 20;**

 **c = a + b;**

 **printf("value of c : %d \n", c);**

 **f = 70.0 / 3.0;**

 **printf("value of f : %f \n", f);**

 **getchar();**

 **return 0;**

**}**

### 2.8.2 Lvalues and Rvalues

There are two kinds of expressions in C. An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment. An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side. Following is a valid statement:

Example

**a = 10; // Valid**

**10 = 20; //Not Valid**

### 2.8.3 Constants and Literals

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*.

### Integer literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Examples

212/\* Legal \*/

215u/\* Legal \*/

0xFeeL/\* Legal \*/

078/\* Illegal: 8 is not an octal digit \*/

032UU/\* Illegal: cannot repeat a suffix \*/

Following are other examples of various type of Integer literals:

85/\* decimal \*/

0213/\* octal \*/

0x4b/\* hexadecimal \*/

30/\* int \*/

30u/\* unsigned int \*/

30l/\* long \*/

30ul/\* unsigned long \*/

### Floating-point literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

3.14159/\* Legal \*/

314159E-5L/\* Legal \*/

510E/\* Illegal: incomplete exponent \*/

210f/\* Illegal: no decimal or exponent \*/

.e55 /\* Illegal: missing integer or fraction \*/

### 2.8.5 Operators

### Arithmetic Operators

Following table shows all the arithmetic operators supported by C language. Assume variable A holds 10 and variable B holds 20 then:

Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| \* | Multiples both operands | A \* B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | Increments operator increases integer value by one | A++ will give 11 |
| -- | Decrements operator decreases integer value by one | A-- will give 9 |

**#include<stdio.h>**

**main()**

**{**

 **int a = 10;**

 **int b = 20;**

 **int c;**

 **c = a + b;**

 **printf("Line 1 - Value of c is %d\n", c);**

 **c = a - b;**

 **printf("Line 2 - Value of c is %d\n", c);**

 **c = a \* b;**

 **printf("Line 3 - Value of c is %d\n", c);**

 **c = a / b;**

 **printf("Line 4 - Value of c is %d\n", c);**

 **c = a % b;**

 **printf("Line 5 - Value of c is %d\n", c);**

 **c = a++;**

 **printf("Line 6 - Value of c is %d\n", c);**

 **c = a--;**

 **printf("Line 7 - Value of c is %d\n", c);**

### }

### Bitwise Operators

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13 then:

Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |

**#include<stdio.h>**

**main()**

**{**

 **unsignedint a = 60; /\* 60 = 0011 1100 \*/**

 **unsignedint b = 13; /\* 13 = 0000 1101 \*/**

 **int c = 0;**

 **c = a & b; /\* 12 = 0000 1100 \*/**

 **printf("Line 1 - Value of c is %d\n", c);**

 **c = a | b; /\* 61 = 0011 1101 \*/**

 **printf("Line 2 - Value of c is %d\n", c);**

 **c = a ^ b; /\* 49 = 0011 0001 \*/**

 **printf("Line 3 - Value of c is %d\n", c);**

 **c = ~a; /\*-61 = 1100 0011 \*/**

 **printf("Line 4 - Value of c is %d\n", c);**

 **c = a << 2; /\* 240 = 1111 0000 \*/**

 **printf("Line 5 - Value of c is %d\n", c);**

 **c = a >> 2; /\* 15 = 0000 1111 \*/**

 **printf("Line 6 - Value of c is %d\n", c);**

### }

### Assignment Operators

There are following assignment operators supported by C language:

Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

**#include<stdio.h>**

**main()**

**{**

 **int a = 21;**

 **int c;**

 **c = a;**

 **printf("Line 1 - = Operator Example, Value of c = %d\n", c);**

 **c += a;**

 **printf("Line 2 - += Operator Example, Value of c = %d\n", c);**

 **c -= a;**

 **printf("Line 3 - -= Operator Example, Value of c = %d\n", c);**

 **c \*= a;**

 **printf("Line 4 - \*= Operator Example, Value of c = %d\n", c);**

 **c /= a;**

 **printf("Line 5 - /= Operator Example, Value of c = %d\n", c);**

 **c = 200;**

 **c %= a;**

 **printf("Line 6 - %= Operator Example, Value of c = %d\n", c);**

 **c <<= 2;**

 **printf("Line 7 - <<= Operator Example, Value of c = %d\n", c);**

 **c >>= 2;**

 **printf("Line 8 - >>= Operator Example, Value of c = %d\n", c);**

 **c &= 2;**

 **printf("Line 9 - &= Operator Example, Value of c = %d\n", c);**

 **c ^= 2;**

 **printf("Line 10 - ^= Operator Example, Value of c = %d\n", c);**

 **c |= 2;**

 **printf("Line 11 - |= Operator Example, Value of c = %d\n", c);**

### }

### Misc Operators = sizeof & ternary

There are few other important operators including sizeof and ? : supported by C Language.

Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof() | Returns the size of an variable. | sizeof(a), where a is integer, will return 4. |
| & | Returns the address of an variable. | &a; will give actual address of the variable. |
| \* | Pointer to a variable. | \*a; will pointer to a variable. |
| ? : | Conditional Expression | If Condition is true ? Then value X : Otherwise value Y |

**#include<stdio.h>**

**main()**

**{**

 **int a = 4;**

 **short b;**

 **double c;**

 **int\* ptr;**

 **/\* example of sizeof operator \*/**

 **printf("Line 1 - Size of variable a = %d\n", sizeof(a));**

 **printf("Line 2 - Size of variable b = %d\n", sizeof(b));**

 **printf("Line 3 - Size of variable c= %d\n", sizeof(c));**

 **/\* example of & and \* operators \*/**

 **ptr = &a; /\* 'ptr' now contains the address of 'a'\*/**

 **printf("value of a is %d\n", a);**

 **printf("\*ptr is %d.\n", \*ptr);**

 **/\* example of ternary operator \*/**

 **a = 10;**

 **b = (a == 1) ? 20 : 30;**

 **printf("Value of b is %d\n", b);**

 **b = (a == 10) ? 20 : 30;**

 **printf("Value of b is %d\n", b);**

### }

### Relational Operators

Following table shows all the relational operators supported by C language. Assume variable A holds 10 and variable B holds 20, then:

Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

**#include<stdio.h>**

**main()**

**{**

 **int a = 10;**

 **int b = 20;**

 **int c;**

 **if (a == b)**

 **{**

 **printf("Line 1 - a is equal to b\n");**

 **}**

 **else**

 **{**

 **printf("Line 1 - a is not equal to b\n");**

 **}**

 **if (a < b)**

 **{**

 **printf("Line 2 - a is less than b\n");**

 **}**

 **else**

 **{**

 **printf("Line 2 - a is not less than b\n");**

 **}**

 **if (a > b)**

 **{**

 **printf("Line 3 - a is greater than b\n");**

 **}**

 **else**

 **{**

 **printf("Line 3 - a is not greater than b\n");**

 **}**

 **/\* Lets change value of a and b \*/**

 **a = 5;**

 **b = 20;**

 **if (a <= b)**

 **{**

 **printf("Line 4 - a is either less than or equal to b\n");**

 **}**

 **if (b >= a)**

 **{**

 **printf("Line 5 - b is either greater than or equal to b\n");**

 **}**

### }

### Operators Precedence

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Examples

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix  | () [] -> . ++ - -   | Left to right  |
| Unary  | + - ! ~ ++ - - (type)\* & sizeof  | Right to left  |
| Multiplicative   | \* / %  | Left to right  |
| Additive   | + -  | Left to right  |
| Shift   | <<>>  | Left to right  |
| Relational   | <<= >>=  | Left to right  |
| Equality   | == !=  | Left to right  |
| Bitwise AND  | &  | Left to right  |
| Bitwise XOR  | ^  | Left to right  |
| Bitwise OR  | |  | Left to right  |
| Logical AND  | &&  | Left to right  |
| Logical OR  | ||  | Left to right  |
| Conditional  | ?:  | Right to left  |
| Assignment  | = += -= \*= /= %=>>= <<= &= ^= |=  | Right to left  |
| Comma  | ,  | Left to right  |

**#include<stdio.h>**

**main()**

**{**

 **int a = 20;**

 **int b = 10;**

 **int c = 15;**

 **int d = 5;**

 **int e;**

 **e = (a + b) \* c / d; // ( 30 \* 15 ) / 5**

 **printf("Value of (a + b) \* c / d is : %d\n", e);**

 **e = ((a + b) \* c) / d; // (30 \* 15 ) / 5**

 **printf("Value of ((a + b) \* c) / d is : %d\n", e);**

 **e = (a + b) \* (c / d); // (30) \* (15/5)**

 **printf("Value of (a + b) \* (c / d) is : %d\n", e);**

 **e = a + (b \* c) / d; // 20 + (150/5)**

 **printf("Value of a + (b \* c) / d is : %d\n", e);**

 **return 0;**

**}**

### Logical Operators

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then:

Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

**#include<stdio.h>**

**main()**

**{**

 **int a = 5;**

 **int b = 20;**

 **int c;**

 **if (a && b)**

 **{**

 **printf("Line 1 - Condition is true\n");**

 **}**

 **if (a || b)**

 **{**

 **printf("Line 2 - Condition is true\n");**

 **}**

 **/\* lets change the value of a and b \*/**

 **a = 0;**

 **b = 10;**

 **if (a && b)**

 **{**

 **printf("Line 3 - Condition is true\n");**

 **}**

 **else**

 **{**

 **printf("Line 3 - Condition is not true\n");**

 **}**

 **if (!(a && b))**

 **{**

 **printf("Line 4 - Condition is true\n");**

 **}**

### }

### 2.9.1 Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| if statement | An **if statement** consists of a boolean expression followed by one or more statements. |
| if...else statement | An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |
| switch statement | A **switch** statement allows a variable to be tested for equality against a list of values. |

### if statement Example

 **/\* local variable definition \*/**

 **int a = 10;**

 **/\* check the boolean condition using if statement \*/**

 **if (a < 20)**

 **{**

 **/\* if condition is true then print the following \*/**

 **printf("a is less than 20\n");**

 **}**

 **printf("value of a is : %d\n", a);**

### if...else statement Example

 **/\* local variable definition \*/**

 **int a = 100;**

 **/\* check the boolean condition \*/**

 **if (a < 200)**

 **{**

 **/\* if condition is true then print the following \*/**

 **printf("a is less than 20\n");**

 **}**

 **else**

 **{**

 **/\* if condition is false then print the following \*/**

 **printf("a is not less than 20\n");**

 **}**

 **printf("value of a is : %d\n", a);**

### C if...else statement

### Switch statement Example

### switch statement in C

 **/\* local variable definition \*/**

 **char grade = 'A';**

 **switch (grade)**

 **{**

 **case'A':**

 **printf("Excellent!\n");**

 **break;**

 **case'B':**

 **case'C':**

 **printf("Well done\n");**

 **break;**

 **case'D':**

 **printf("You passed\n");**

 **break;**

 **case'F':**

 **printf("Better try again\n");**

 **break;**

 **default:**

 **printf("Invalid grade\n");**

 **}**

 **printf("Your grade is %c\n", grade);**

### 2.9.2 Loops

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| while loop | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| for loop | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| do...while loop | Like a while statement, except that it tests the condition at the end of the loop body |

### while loop example

**#include<stdio.h>**

**int main()**

**{**

 **/\* local variable definition \*/**

 **int a = 10;**

 **/\* while loop execution \*/**

 **while (a < 20)**

 **{**

 **printf("value of a: %d\n", a);**

 **a++;**

 **}**

 **return 0;**

**}**

### while loop in C

### for loop example

### for loop in C

 **#include<stdio.h>**

**int main()**

**{**

**/\* for loop execution \*/**

**for (int a = 2; a <10; a++)**

**{**

 **printf("value of a: %d\n", a);**

**}**

 **return 0;**

**}**

### do...while loop example

**#include<stdio.h>**

**int main()**

**{**

 **/\* local variable definition \*/**

 **int a = 10;**

 **/\* do loop execution \*/**

 **do**

 **{**

 **printf("value of a: %d\n", a);**

 **a = a + 1;**

 **} while (a < 20);**

 **return 0;**

**}**

### do...while loop in C

### 2.9.3 Loop Control Statements:

Loop control statements change execution from its normal sequence.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| break statement | Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| continue statement | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| goto statement | Transfers control to the labelled statement. Though it is not advised to use goto statement in your program. |

### break statement example

#**include<stdio.h>**

**int main()**

**{**

 **/\* local variable definition \*/**

 **int a = 10;**

 **/\* while loop execution \*/**

 **while (a < 20)**

 **{**

 **printf("value of a: %d\n", a);**

 **a++;**

 **if (a > 15)**

 **{**

**/\* terminate the loop using break statement \*/**

 **break;**

 **}**

 **}**

 **return 0;**

**}**

### c break statement

### continue statement example

**#include<stdio.h>**

**int main()**

**{**

 **/\* local variable definition \*/**

 **int a = 10;**

 **/\* do loop execution \*/**

 **do**

 **{**

 **if (a == 15)**

 **{**

 **/\* skip the iteration \*/**

 **a = a + 1;**

 **continue;**

 **}**

 **printf("value of a: %d\n", a);**

 **a++;**

 **} while (a < 20);**

 **return 0;**

### C continue statement

### 2.10.1 Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions. Functions can be logically division so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters.

A function **definition** provides the actual body of the function. The function definition consists of a *function header* and a *function body*.

**Return Type**: A **return\_type** is the data type of the value the function returns.

* **Function Name:** The function name and the parameter list together constitute the function signature.
* **Parameters:** A parameter is like a placeholder. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body:** The function body contains a collection of statements that define what the function does.

### Example:

**#include<stdio.h>**

**/\* function declaration \*/**

**int max(int num1, int num2);**

**int main()**

**{**

 **/\* local variable definition \*/**

 **int a = 100;**

 **int b = 200;**

 **int ret;**

 **/\* calling a function to get max value \*/**

 **ret = max(a, b);**

 **printf("Max value is : %d\n", ret);**

 **return 0;**

**}**

**/\* function returning the max between two numbers \*/**

**int max(intnum1, intnum2)**

**{**

 **/\* local variable declaration \*/**

 **int result;**

 **if (num1>num2)**

 **result = num1;**

 **else**

 **result = num2;**

 **return result;**

**}**

### Math Functions

Although addition, subtraction, multiplication, and division are easily accomplished using C's arithmetic operators, no such operators exist for raising a number to a power, finding the square root of a number, or determining trigonometric values. To facilitate the calculation of powers, square roots, trigonometric, logarithmic, and other mathematical calculations frequently required in scientific and engineering programs, C provides standard preprogrammed functions that can be included in a program.

See table for more details

### Square root example

3.0 \* sqrt (5 \* 33 - 13. 71) / 5

|  |  |
| --- | --- |
| Step | Result |
| 1.Performmultiplicationin argument | 3.0\* sqrt(165 - 13.71)/ 5 |
| 2.Completeargumentcalculation | 3.0\* sqrt(151.290000) / 5 |
| 3.Returna functionvalue | 3.0\* 12.300000/ 5 |
| 4.Performthe multiplication | 36.900000/ 5 |
| 5.Performthe division | 7.380000 |

### 2.10.2 Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed. There are three places where variables can be declared in C programming language:

1. Inside a function or a block which is called **local** variables,
2. Outside of all functions which is called **global** variables.
3. In the definition of function parameters which is called **formal** parameters.

### Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables.

### Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

### Formal Parameters

### Formal parameters, are treated as local variables within that function and they will take preference over the global variable.

### 2.10.3 Arrays

An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.



### Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [ array Size ];

This is called a *single-dimensional* array. The **array Size** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

double balance[10];

Now, *balance* is a variable array which is sufficient to hold up to 10 double numbers.

### Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

double balance[5]={1000.0,2.0,3.4,17.0,50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[]={1000.0,2.0,3.4,17.0,50.0};

You will create exactly the same array as you did in the previous example.

balance[4]=50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:



### Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

#include<stdio.h>

int main()

{

 int n[10]; /\* n is an array of 10 integers \*/

 int i, j;

 /\* initialize elements of array n to 0 \*/

 for (i = 0; i < 10; i++)

 {

 n[i] = i + 100; /\* set element at location i to i + 100 \*/

 }

 /\* output each array element's value \*/

 for (j = 0; j < 10; j++)

 {

 printf("Element[%d] = %d\n", j, n[j]);

 }

 return 0;

}

When the above code is compiled and executed, it produces the following result:

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

### 2.10.4 Pointers

A **pointer** is a variable, whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk \* you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

int\*ip;/\* pointer to an integer \*/

double\*dp;/\* pointer to a double \*/

float\*fp;/\* pointer to a float \*/

char\*ch /\* pointer to a character \*/

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

### How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator **\*** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

#include<stdio.h>

int main()

{

 int var = 20; /\* actual variable declaration \*/

 int \*ip; /\* pointer variable declaration \*/

 ip = &var; /\* store address of var in pointer variable\*/

 printf("Address of var variable: %x\n", &var);

 /\* address stored in pointer variable \*/

 printf("Address stored in ip variable: %x\n", ip);

 /\* access the value using the pointer \*/

 printf("Value of \*ip variable: %d\n", \*ip);

 return 0;

}

When the above code is compiled and executed, it produces result something as follows:

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

### 2.10.5 Strings

The string in C programming language is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6]={'H','e','l','l','o','\0'};

If you follow the rule of array initialization then you can write the above statement as follows:

char greeting[]="Hello";

Following is the memory presentation of above-defined string in C/C++:



The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above mentioned string:

#include<stdio.h>

int main()

{

 char greeting[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };

 printf("Greeting message: %s\n", greeting);

 return 0;

}

When the above code is compiled and executed, it produces result something as follows:

Greeting message: Hello