Imperative Programming (FY.IT-Semester-1)

Unit 1

Introduction:

1 TYPES OF PROGRAMMING LANGUAGES

There are many different languages can be used to program a computer. The most basic of these is machine language—a collection of very detailed, cryptic instructions that control the computer’s internal circuitry. This is the natural dialect of the computer. Very few computer programs are actually written in machine language, however, for two significant reasons: First, because machine language is very cumbersome to work with and second, because every different type of computer has its own unique instruction set. Thus, a machine-language program written for one type of computer cannot be run on another type of computer without significant alterations. Usually, a computer program will be written in some high-level language, whose instruction set is more compatible with human languages and human thought processes. Most of these are general-purpose languages such as C. (Some other popular general-purpose languages are Pascal, Fortran and BASIC.) There are also various special-purpose languages that are specifically designed for some particular type of application. Some common examples are CSMP and SIMAN, which are special-purpose simulation languages, and LISP, which is a special-purpose language that is widely used for artificial intelligence applications. As a rule, a single instruction in a high-level language will be equivalent to several instructions in machine language. This greatly simplifies the task of writing complete, correct programs. Furthermore, the rules for programming in a particular high-level language are much the same for all computers, so that a program written for one computer can generally be run on many different computers with little or no alteration. Thus, we see that a high-level language offers three significant advantages over machine language: simplicity, uniformity and portability (i.e., machine independence). A program that is written in a high-level language must, however, be translated into machine language.
before it can be executed. This is known as *compilation* or *interpretation*, depending on how it is carried out. (Compilers translate the entire program into machine language before executing any of the instructions. Interpreters, on the other hand, proceed through a program by translating and then executing singleinstructions or small groups of instructions.) In either case, the translation is carried out automatically within the computer. In fact, inexperienced programmers may not even be aware that this process is taking place, since they typically see only their original high-level program, the input data, and the calculated results. Most implementations of C operate as compilers. A compiler or interpreter is itself a computer program. It accepts a program written in a high-level language (e.g., C) as input, and generates a corresponding machine-language program as output. The original high-level program is called the *source* program, and the resulting machine-language program is called the *object* program. Every computer must have its own compiler or interpreter for a particular high-level language. It is generally more convenient to develop a new program using an interpreter rather than a compiler. Once an error-free program has been developed, however, a compiled version will normally execute much faster than an interpreted version. The reasons for this are beyond the scope of our present discussion.

2 INTRODUCTION TO C

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English *keywords* such as if, else, for, do and while. In this respect C resembles other high-level structured programming languages such as Pascal and Fortran. C also contains certain additional features, however, that allow it to be used at a lower level, thus bridging the gap between machine language and the more conventional high-level languages. This flexibility allows C to be used for *systems programming* (e.g., for writing operating systems) as well as for *applications programming* (e.g., for writing a program to solve a complicated system of mathematical equations, or for writing a program to bill customers). C is characterized by the ability to write very concise source programs, due in part to the large number of
operators included within the language. It has a relatively small instruction set, though actual implementations include extensive library functions which enhance the basic instructions. Furthermore, the language encourages users to write additional library functions of their own. Thus the features and capabilities of the language can easily be extended by the user. C compilers are commonly available for computers of all sizes, and C interpreters are becoming increasingly common. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages. The interpreters are less efficient, though they are easier to use when developing a new program. Many programmers begin with an interpreter, and then switch to a compiler once the program has been debugged (i.e., once all of the programming errors have been removed). Another important characteristic of C is that its programs are highly portable, even more so than with other high-level languages. The reason for this is that C relegates most computer-dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for the particular characteristics of the host computer. These library functions are relatively standardized, however, and each individual library function is generally accessed in the same manner from one version of C to another. Therefore, most C programs can be processed on many different computers with little or no alteration.

History of C

C was originally developed in the 1970s by Dennis Ritchie at Bell Telephone Laboratories, Inc. (now a part of AT&T). It is an outgrowth of two earlier languages, called BCPL and B, which were also developed at Bell Laboratories. C was largely confined to use within Bell Laboratories until 1978, when Brian Kernighan and Ritchie published a definitive description of the language.* The Kernighan and Ritchie description is commonly referred to as “K&R C.” Following the publication of the K&R description, computer professionals, impressed with C’s many desirable features, began to promote the use of the language. By the mid 1980s, the popularity of C had become widespread. Numerous C compilers and
interpreters had been written for computers of all sizes, and many commercial application programs had been developed. Moreover, many commercial software products that were originally written in other languages were rewritten in C in order to take advantage of its efficiency and its portability. Early commercial implementations of C differed somewhat from Kernighan and Ritchie’s original definition, resulting in minor incompatibilities between different implementations of the language. These differences diminished the portability that the language attempted to provide. Consequently, the American National Standards Institute** (ANSI committee X3J11) has developed a standardized definition of the C language. Virtually all commercial C compilers and interpreters now adhere to the ANSI standard. Many also provide additional features of their own. In the early 1980s, another high-level programming language, called C++, was developed by Bjarne Stroustrup*** at the Bell Laboratories. C++ is built upon C, and hence all standard C features are available within C++. However, C++ is not merely an extension of C. Rather, it incorporates several new fundamental concepts that form a basis for object-oriented programming—a new programming paradigm that is of interest to professional programmers. We will not describe C++ in this book, except to mention that a knowledge of C is an excellent starting point for learning C++. This book describes the features of C that are included in the ANSI standard and are supported by commercial C compilers and interpreters. The reader who has mastered this material should have no difficulty in customizing a C program to any particular implementation of the language.

Structure of a C Program

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

1. A function heading, which consists of the function name, followed by an optional list of arguments, enclosed in parentheses.
2. A list of argument declarations, if arguments are included in the heading.
3. A compound statement, which comprises the remainder of the function. The arguments are symbols that represent information being passed between the function and other parts of the program. (Arguments are also referred to as parameters.) Each compound statement is enclosed within a pair of braces, i.e., { }. The braces may contain one or more elementary statements (called expression statements) and other compound statements. Thus compound statements may be nested, one within another. Each expression statement must end with a semicolon (;). Comments (remarks) may appear anywhere within a program, as long as they are placed within the delimiters / * and */ (e.g., /* t h i s is a comment */). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features. These program components will be discussed in much greater detail later in this book. For now, the reader should be concerned only with an overview of the basic features that characterize most C programs.

EXAMPLE 1.6 Area of a Circle Here is an elementary C program that reads in the radius of a circle, calculates its area and then writes the calculated result.

```c
#include <stdio.h> /* LIBRARY FILE ACCESS */
main() /* FUNCTION HEADING */
float radius, area; /* VARIABLE DECLARATIONS */
printf("Radius = ? /* OUTPUT STATEMENT (PROMPT) */I");
scanf("%f", &radius); /* INPUT STATEMENT */
area = 3.14159 * radius * radius; /* ASSIGNMENT STATEMENT */
printf("Area = %f", area); /* OUTPUT STATEMENT */
```

The comments at the end of each line have been added in order to emphasize the overall program organization. Normally a C program will not look like this. Rather, it might appear as shown below.
The following features should be pointed out in this last program.

1. The program is typed in lowercase. Either upper- or lowercase can be used, though it is customary to type ordinary instructions in lowercase. Most comments are also typed in lowercase, though comments are sometimes typed in uppercase for emphasis, or to distinguish certain comments from the instructions.

2. The first line is a comment that identifies the purpose of the program.

3. The second line contains a reference to a special file (called stdio.h) which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler.

4. The third line is a heading for the function main. The empty parentheses following the name of the function indicate that this function does not include any arguments.

5. The remaining five lines of the program are indented and enclosed within a pair of braces. These five lines comprise the compound statement within main.

6. The first indented line is a variable declaration. It establishes the symbolic names radius and area as floating-point variables (more about this in the next chapter).

7. The remaining four indented lines are expression statements. The second indented line (printf) generates a request for information (namely, a value for the radius). This value is entered into the computer via the third indented line (scanf).

8. The fourth indented line is a particular type of expression statement called an assignment statement. This statement causes the area to be
calculated from the given value of the radius. Within this statement the asterisks (*) represent multiplication signs.

9. The last indented line (printf) causes the calculated value for the area to be displayed. The numerical value will be preceded by a brief label.

10. Notice that each expression statement within the compound statement ends with a semicolon. This is required of all expression statements.

1 Finally, notice the liberal use of spacing and indentation, creating whitespace within the program. The blank lines separate different parts of the program into logically identifiable components, and the indentation indicates subordinate relationships among the various instructions. These features are not grammatically essential, but their presence is strongly encouraged as a matter of good programming practice.

Execution of the program results in an interactive dialog such as that shown below. The user's response is underlined, for clarity.

Radius = 7.3
Area = 28.274309

Understanding the Program Development Cycle
A programmer’s job involves writing instructions (such as those in the doubling program in the preceding section), but a professional programmer usually does not just sit down at a computer keyboard and start typing. Figure 1-1 illustrates the program development cycle, which can be broken down into at least seven steps:

1. Understand the problem.
2. Plan the logic.
3. Code the program.
4. Use software (a compiler or interpreter) to translate the program into machine language.
5. Test the program.
6. Put the program into production.
1 Understanding the Problem

Professional computer programmers write programs to satisfy the needs of others, called users or end users. Examples could include a Human Resources department that needs a printed list of all employees, a Billing department that wants a list of clients who are 30 or more days overdue on their payments, and an Order department that needs a Web site to provide buyers with an online shopping cart in which to gather their orders. Because programmers are providing a service to these users, programmers must first understand what the users want. Although when a program runs, you usually think of the logic as a cycle of input-processing-output operations; when you plan a program, you think of the output first. After you understand what the desired result is, you can plan what to input and process to achieve it. Suppose the director of Human Resources says to a programmer, “Our department needs a list of all employees who have been here over five years, because we want to invite them to a special thank-you dinner.” On the surface, this seems like a simple request. An experienced
Programmer, however, will know that the request is incomplete. For example, you might not know the answers to the following questions about which employees to include:

• Does the director want a list of full-time employees only, or a list of full- and part-time employees together?
• Does she want people who have worked for the company on a Month-to-month contractual basis over the past five years, or only Regular, permanent employees?
• Do the listed employees need to have worked for the organization for five years as of today, as of the date of the dinner, or as of some other cutoff date?
• What about an employee who, for example, worked three years, took a two-year leave of absence, and has been back for three years? The programmer cannot make any of these decisions; the user (in this case, the Human Resources director) must address these questions.

More decisions still might be required. For example:
• What data should be included for each listed employee? Should the list contain both first and last names? Social Security numbers? Phone numbers? Addresses?
• Should the list be in alphabetical order? Employee ID number order? Length-of-service order? Some other order?
• Should the employees be grouped by any criteria, such as department number or years of service?

Several pieces of documentation are often provided to help the programmer understand the problem. Documentation consists of all the supporting paperwork for a program; it might include items such as original requests for the program from users, sample output, and descriptions of the data items available for input. Really understanding the problem may be one of the most difficult aspects of programming. On any job, the description of what the user needs may be vague—worse yet, users may not really know what they want, and users who think they know frequently change their minds.
after seeing sample output. A good programmer is often part counselor, part detective!

2 Planning the Logic

The heart of the programming process lies in planning the program’s logic. During this phase of the process, the programmer plans the steps of the program, deciding what steps to include and how to order them. You can plan the solution to a problem in many ways. The two most common planning tools are flowcharts and pseudocode. Both tools involve writing the steps of the program in English, much as you would plan a trip on paper before getting into the car or plan a party theme before shopping for food and favors. The programmer shouldn’t worry about the syntax of any particular language at this point, but should focus on figuring out what sequence of events will lead from the available input to the desired output. Planning the logic includes thinking carefully about all the possible data values a program might encounter and how you want the program to handle each scenario. The process of walking through a program’s logic on paper before you actually write the program is called desk-checking. You will learn more about planning the logic throughout this book; in fact, the book focuses on this crucial step almost exclusively.

3 Coding the Program

After the logic is developed, only then can the programmer write the program. Hundreds of programming languages are available. Programmers choose particular languages because some have built-in capabilities that make them more efficient than others at handling certain types of operations. Despite their differences, programming languages are quite alike in their basic capabilities—each can handle input operations, arithmetic processing, output operations, and other standard functions. The logic developed to solve a programming problem can be executed using any number of languages. Only after choosing a language must the programmer be concerned with proper
punctuation and the correct spelling of commands—in other words, using the correct syntax. Some very experienced programmers can successfully combine logic planning and program coding in one step. This may work for The Program planning and writing a very simple program, just as you can plan and write a postcard to a friend using one step. A good term paper or a Hollywood screenplay, however, needs planning before writing—and so do most programs. Which step is harder: planning the logic or coding the program? Right now, it may seem to you that writing in a programming language is a very difficult task, considering all the spelling and syntax rules you must learn. However, the planning step is actually more difficult. Which is more difficult: thinking up the twists and turns to the plot of a best-selling mystery novel, or writing a translation of an existing novel from English to Spanish? And who do you think gets paid more, the writer who creates the plot or the translator? (Try asking friends to name any famous translator!)

3 Using Software to Translate the Program into Machine Language

Even though there are many programming languages, each computer knows only one language: its machine language, which consists of 1s and 0s. Computers understand machine language because they are made up of thousands of tiny electrical switches, each of which can be set in either the on or off state, which is represented by a 1 or 0, respectively. Languages like Java or Visual Basic are available for programmers because someone has written a translator program (a compiler or interpreter) that changes the programmer’s English-like high-level programming language into the low-level machine language that the computer understands. If you write a programming language statement incorrectly (for example, by misspelling a word, using a word that doesn’t exist in the language, or using “illegal” grammar), the translator program doesn’t know how to proceed and issues an error message identifying a syntax error, which is a misuse of a language’s grammar rules. Although making errors is never desirable, syntax errors are not a major concern to programmers, because the compiler or interpreter catches every syntax error and displays a message that notifies you of the problem. The computer will not execute a
program that contains even one syntax error. Typically, a programmer develops a program’s logic, writes the code, and compiles the program, receiving a list of syntax errors. The programmer then corrects the syntax errors and compiles the program again. Correcting the first set of errors frequently reveals new errors that originally were not apparent to the compiler. For example, if you could use an English compiler and submit the sentence “The dg chase the cat,” the compiler at first might point out only one syntax error.

The second word, “dg,” is illegal because it is not part of the English language. Only after you corrected the word to “dog” would the compiler find another syntax error on the third word, “chase,” because it is the wrong verb form for the subject “dog.” This doesn’t mean “chase” is necessarily the wrong word. Maybe “dog” is wrong; perhaps the subject should be “dogs,” in which case “chase” is right. Compilers don’t always know exactly what you mean, nor do they know what the proper correction should be, but they do know when something is wrong with your syntax.

When writing a program, a programmer might need to recompile the code several times. An executable program is created only when the code is free of syntax errors. When you run an executable program, it typically also might require input data. Figure 1-2 shows a diagram of this entire process.
4 Testing the Program

A program that is free of syntax errors is not necessarily free of logical errors. A logical error results when you use a syntactically correct statement but use the wrong one for the current context. For example, the English sentence “The dog chases the cat,” although syntactically perfect, is not logically correct if the dog chases a ball or the cat is the aggressor. Once a program is free of syntax errors, the programmer can test it—that is, execute it with some sample data to see whether the results are logically correct.

Recall the number-doubling program:

```
input myNumber
set myAnswer = myNumber * 2
output myAnswer
```

If you execute the program, provide the value 2 as input to the program, and the answer 4 is displayed, you have executed one successful test run of the program. However, if the answer 40 is displayed, maybe the program contains a logical error. Maybe the second line of code was mistyped with an extra zero, so that the program reads:

```
input myNumber
set myAnswer = myNumber * 20
output myAnswer
```

Placing 20 instead of 2 in the multiplication statement caused a logical error. Notice that nothing is syntactically wrong with this second program—it is just as reasonable to multiply a number by 20 as by 2—but if the programmer intends only to double myNumber, then a logical error has occurred. Programs should be tested with many sets of data. For example, if you write the program to double a number, then enter 2 and get an output value of 4, that doesn’t necessarily mean you have a correct program. Perhaps you have typed this program by mistake:
input myNumber
set myAnswer = myNumber + 2
output myAnswer

An input of 2 results in an answer of 4, but that doesn’t mean your program doubles numbers—it actually only adds 2 to them. If you test your program with additional data and get the wrong answer—for example, if you enter 7 and get an answer of 9—you know there is a problem with your code. Selecting test data is somewhat of an art in itself, and it should be done carefully. If the Human Resources department wants a list of the names of five-year employees, it would be a mistake to test the program with a small sample file of only long-term employees. If no newer employees are part of the data being used for testing, you do not really know if the program would have eliminated them from the five-year list. Many companies do not know that their software has a problem until an unusual circumstance occurs—for example, the first time an employee has more than nine dependents, the first time a customer orders more than 999 items at a time, or when (as well-documented in the popular press) a new century begins. Putting the Program into Production Once the program is tested adequately, it is ready for the organization to use. Putting the program into production might mean simply running the program once, if it was written to satisfy a user’s request for a special list. However, the process might take months if the program will be run on a regular basis, or if it is one of a large system of programs being developed. Perhaps data-entry people must be trained to prepare the input for the new program; users must be trained to understand the output; or existing data in the company must be changed to an entirely new format to accommodate this program. Conversion, the entire set of actions an organization must take to switch over to using a new program or set of programs, can sometimes take months or years to accomplish.

5 Maintaining the Program

After programs are put into production, making necessary changes is called maintenance. Maintenance can be required for many reasons: new tax rates are legislated, the format of an input file is altered, or the end user requires additional information not included in the original output specifications, to
name a few. Frequently, your first programming job will require maintaining previously written programs. When you maintain the programs others have written, you will appreciate the effort the original programmer put into writing clear code, using reasonable variable names, and documenting his or her work. When you make changes to existing programs, you repeat the development cycle. That is, you must understand the changes, then plan, code, translate, and test them before putting them into production. If a substantial number of program changes are required, the original program might be retired, and the program development cycle might be started for a new program.

Using Pseudocode Statements and Flowchart Symbols
When programmers plan the logic for a solution to a programming problem, they often use one of two tools: pseudocode (pronounced “sue-doe-code”) or flowcharts. Pseudocode is an English-like representation of the logical steps it takes to solve a problem. A flowchart is a pictorial representation of the same thing. Pseudo is a prefix that means “false,” and to code a program means to put it in a programming language; therefore, pseudocode simply means “false code,” or sentences that appear to have been written in a computer programming language but do not necessarily follow all the syntax rules of any specific language.

Writing Pseudocode
You have already seen examples of statements that represent pseudocode earlier in this chapter, and there is nothing mysterious about them. The following five statements constitute a pseudocode representation of a number-doubling problem:

start
input myNumber
set myAnswer = myNumber * 2
output myAnswer
stop
Using pseudocode involves writing down all the steps you will use in a program. Usually, programmers preface their pseudocode with a beginning statement like start and end it with a terminating statement like stop. The statements between start and stop look like English and are indented slightly so that start and stop stand out. Most programmers do not bother with punctuation such as period sat the end of pseudocode statements, although it would not be wrong to use them if you prefer that style. Similarly, there is no need to capitalize the first word in a sentence, although you might choose to do so. This book follows the conventions of using lowercase letters for verbs that begin pseudocode statements and omitting periods at the end of statements. Pseudocode is fairly flexible because it is a planning tool, and not the final product. Therefore, for example, you might prefer any of the following:

• Instead of start and stop, some pseudocode developers would use the terms begin and end.
• Instead of writing input myNumber, some developers would write get myNumber or read myNumber.
• Instead of writing set myAnswer = myNumber * 2, some developers would write calculate myAnswer = myNumber times 2 or compute myAnswer as myNumber doubled.
• Instead of writing output myAnswer, many pseudocode developers would write display myAnswer, print myAnswer, or write myAnswer.

The point is, the pseudocode statements are instructions to retrieve an original number from an input device and store it in memory where it can be used in a calculation, and then to get the calculated answer from memory and send it to an output device so a person can see it. When you eventually convert your pseudocode to a specific programming language, you do not have such flexibility because specific syntax will be required. For example, if you use the C# programming language and write the statement to output the answer, you will code the following: Console. Write (myAnswer);

The exact use of words, capitalization, and punctuation are important in the C# statement, but not in the pseudocode statement.

Algorithms and Flowchart
**Algorithms**

1. A sequential solution of any program that written in human language, called algorithm.
2. Algorithm is first step of the solution process, after the analysis of problem, programmer writes the algorithm of that problem.
3. Example of Algorithms:

Q. Write a algorithm to find out number is odd or even?
Ans.

step 1 : start
step 2 : input number
step 3 : rem=number mod 2
step 4 : if rem=0 then
print "number even"
else
print "number odd"
endif
step 5 : stop

**Flowchart**

1. Graphical representation of any program is called flowchart.
2. There are some standard graphics that are used in flowchart as following:

![Flowchart Diagram]

- Start / Stop
- Input / Output
- Process / Instruction

Lines or arrows represent the direction of the flow of control.
Algorithm for calculate factorial value of a number:
[algorithm to calculate the factorial of a number]
step 1. Start
step 2. Read the number n
step 3. [Initialize]
  i=1, fact=1
step 4. Repeat step 4 through 6 until i=n
step 5. fact=fact*i
step 6. i=i+1
step 7. Print fact
step 8. Stop
[process finish of calculate the factorial value of a number]
Flowchart for calculate factorial value of a number:
Hello World Example

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```c
#include int main()
{
    /* my first program in C */
    printf("Hello, World! \n");
    return 0;
}
```

Let us take a look at the various parts of the above program:

1. The first line of the program `#include` is a preprocessor command, which tells a C compiler to include `stdio.h` file before going to actual compilation.
2. The next line `int main()` is the main function where the program execution begins.
3. The next line `/*...*/` will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
4. The next line `printf(...)` is another function available in C which causes the message "Hello, World!" to be displayed on the screen. 5. The next line `return 0;` terminates the main() function and returns the value 0.

**Compile and Execute C Program**
Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps:
1. Open a text editor and add the above-mentioned code.
2. Save the file as hello.c
3. Open a command prompt and go to the directory where you have saved the file.
4. Type gcc hello.c and press enter to compile your code.
5. If there are no errors in your code, the command prompt will take you to the next line and would generate a.out executable file.
6. Now, type a.out to execute your program.
7. You will see the output "Hello World" printed on the screen.

$ gcc hello.c
$ ./a.out
Hello, World!

Make sure the gcc compiler is in your path and that you are running it in the directory containing the source file hello.c.

THE C CHARACTER SET

C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as building blocks to form basic program elements (e.g., constants, variables, operators, expressions, etc.). The special characters are listed below.

+   -   *   /   =   %   &   #
!   ?   ^   "   '   ~   \   |
<   >   (   )   [   ]   {   }   
:   ;   .   ,   _

(blank space)
Most versions of the language also allow certain other characters, such as @ and $, to be included within strings and comments.

IDENTIFIERS AND KEYWORDS

Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consist of letters and digits, in any order, except that the first character must be a letter. Both upper- and lowercase letters are permitted, though common usage favors the use of lowercase letters for most types of identifiers. Upper- and lowercase letters are not interchangeable (i.e., an uppercase letter is not equivalent to the corresponding lowercase letter.) The underscore character (-) can also be included, and is considered to be a letter. An underscore is often used in the middle of an identifier. An identifier may also begin with an underscore, though this is rarely done in practice.

EXAMPLE

The following names are valid identifiers.

X Y12 sum-1 _temperature
Names area tax-rate

TABLE Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables 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
Data types in C Language

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we use in our program. These datatypes have different storage capacities.

C language supports 2 different type of data types,

**Primary data types**

These are fundamental data types in C namely integer(int), floating(float), charater(char) and void.

**Derived data types**

Derived data types are like arrays, functions, stuctures and pointers. These are discussed in detail later.
**Integer type**

Integers are used to store whole numbers.

**Size and range of Integer type on 16-bit machine**

<table>
<thead>
<tr>
<th>Type</th>
<th>Size(bytes)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>int or signed int</td>
<td>2</td>
<td>-32,768 to 32767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>short int or signed short int</td>
<td>1</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>long int or signed long int</td>
<td>4</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>4</td>
<td>0 to 4,294,967,295</td>
</tr>
</tbody>
</table>

**Floating type**

Floating types are used to store real numbers.

**Size and range of Integer type on 16-bit machine**

<table>
<thead>
<tr>
<th>Type</th>
<th>Size(bytes)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>4</td>
<td>3.4E-38 to 3.4E+38</td>
</tr>
<tr>
<td>Type</td>
<td>Size (bytes)</td>
<td>Range</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>char or signed char</td>
<td>1</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>0 to 255</td>
</tr>
</tbody>
</table>

**void type**

void type means no value. This is usually used to specify the type of functions.

**Constants**: Constants are of fixed values that do not change during the execution of a program. There are various types of constants. The types are illustrated in the following figure.
**Integer constants:** An integer constant refers to a sequence of digits. There are three types of integer constants, namely, decimal integer, octal integer and hexadecimal integer.

**Decimal integer** consists of a set of digits from 0 to 9, preceded by an optional + or – sign. Examples, 123 -321 0 64932

**Octal integer** consists of a set of digits from 0 to 7, with a leading 0. Examples, 037 0 0437 0551

A sequence of digits preceded by 0x or 0X is considered as **hexadecimal integer**. They may also includes letters from A to F or from a to f. The letters represents the numbers from 10 to 15. Examples, 0X2 0x9F 0Xbcd 0x

**Real constants:** Real constants are used to represent quantities that are very continuously, such as distances, temperature etc. These quantities are
represented by numbers containing fractional parts. Examples, 0.00832 -0.75 33.337

**Single character constants:** A single character constants contains a single character enclosed within a pair of single quote marks. Example, ‘5’ ‘X’ ‘;’

**String constants:** A string constant contains a string of characters enclosed within a pair of double quote marks. Examples, “Hello!” “1987” “?....!”

**VARIABLES**
A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Typically a single octet (one byte). This is an integer type.</td>
</tr>
<tr>
<td>int</td>
<td>The most natural size of integer for the machine.</td>
</tr>
<tr>
<td>float</td>
<td>A single-precision floating point value.</td>
</tr>
<tr>
<td>double</td>
<td>A double-precision floating point value</td>
</tr>
<tr>
<td>void</td>
<td>Represents the absence of type. C</td>
</tr>
</tbody>
</table>
**Variable Definition in C**: A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```c
type variable_list;
```

Here, `type` must be a valid C data type including `char`, `w_char`, `int`, `float`, `double`, `bool`, or any user-defined object; and `variable_list` may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```c
int i, j, k;
char c, ch;
float f salary;
double d;
```

The line `int i, j, k;` declares and defines the variables `i`, `j`, and `k`; which instruct the compiler to create variables named `i`, `j`, and `k` of type `int`. Variables can be initialized (assigned an initial value) in their declaration. The initialize consists of an equal sign followed by a constant expression as follows:

```c
type variable_name = value;
```

Some examples are:

```c
Extern int d = 3, f = 5;       // declaration of d and f.
int d = 3, f = 5;              // definition and initializing d and f.
byte z = 22;                   // definition and initializes z.
char x = 'x';                  // the variable x has the value 'x'.
```