

Unit I: Data Types & Operators

Syllabus: Levels of programming language, Introduction to Programming Paradigms, Algorithm, Flowchart, for problem solving with Sequential Logic Structure, Introduction to imperative language; syntax and constructs of a specific language. Types Operator and Expressions: Variable Names, Data Type and Sizes (Little Endian Big Endian), Constants, Declarations, Arithmetic Operators, Relational Operators, Logical Operators, Type Conversion, Increment Decrement Operators, Bitwise, Operators, Assignment Operators and Expressions, Precedence and Order of Evaluation, variable naming, Hungarian Notation.

1.1 Levels of Programming Language

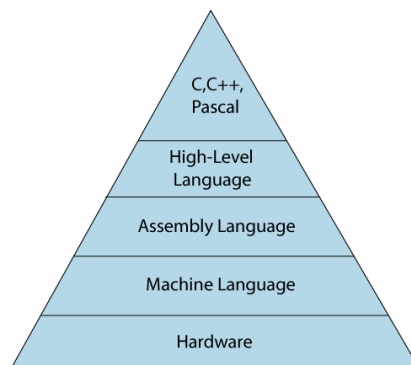
1.1.1 What is a programming language?

A programming language defines a set of instructions that are compiled together to perform a specific task by the CPU (Central Processing Unit). The programming language mainly refers to high-level languages such as C, C++, Pascal, Ada, COBOL, etc.

Each programming language contains a unique set of keywords and syntax, which are used to create a set of instructions. Thousands of programming languages have been developed till now, but each language has its specific purpose. These languages vary in the level of abstraction they provide from the hardware. Some programming languages provide less or no abstraction while some provide higher abstraction. Based on the levels of abstraction, they can be classified into two categories:

- Low-level language
- High-level language

The image which is given below describes the abstraction level from hardware. As we can observe from the below image that the machine language provides no abstraction, assembly language provides less abstraction whereas high-level language provides a higher level of abstraction.



1.1.2 Low-level language

The low-level language is a programming language that provides no abstraction from the hardware, and it is represented in 0 or 1 forms, which are the machine instructions. The languages that come under this category are the Machine level language and Assembly language.

1.1.2.1 Machine-level language

The machine-level language is a language that consists of a set of instructions that are in the binary form 0 or 1. As we know that computers can understand only machine instructions, which are in binary digits, i.e., 0 and 1, so the instructions given to the computer can be only in binary codes. Creating a program in a machine-level language is a very difficult task as it is not easy for the programmers to write the program in machine instructions. It is error-prone as it is not easy to understand, and its maintenance is also very high. A machine-level language is not portable as each computer has its machine instructions, so if we write a program in one computer will no longer be valid in another computer.

The different processor architectures use different machine codes, for example, a PowerPC processor contains RISC architecture, which requires different code than intel x86 processor, which has a CISC architecture.

1.1.2.2 Assembly Language

The assembly language contains some human-readable commands such as mov, add, sub, etc. The problems which we were facing in machine-level language are reduced to some extent by using an extended form of machine-level language known as assembly language. Since assembly language instructions are written in English words like mov, add, sub, so it is easier to write and understand.

As we know that computers can only understand the machine-level instructions, so we require a translator that converts the assembly code into machine code. The translator used for translating the code is known as an assembler.

The assembly language code is not portable because the data is stored in computer registers, and the computer has to know the different sets of registers.

The assembly code is not faster than machine code because the assembly language comes above the machine language in the hierarchy, so it means that assembly language has some abstraction from the hardware while machine language has zero abstraction.

1.1.2.3 Differences between Machine-Level language and Assembly language

The following are the differences between machine-level language and assembly language:

Machine-level language	Assembly language
The machine-level language comes at the lowest level in the hierarchy, so it has zero abstraction level from the hardware.	The assembly language comes above the machine language means that it has less abstraction level from the hardware.
It cannot be easily understood by humans.	It is easy to read, write, and maintain.
The machine-level language is written in binary digits, i.e., 0 and 1.	The assembly language is written in simple English language, so it is easily understandable by the users.
It does not require any translator as the machine code is directly executed by the computer.	In assembly language, the assembler is used to convert the assembly code into machine code.
It is a first-generation programming language.	It is a second-generation programming language.

1.1.3 High-Level Language

The high-level language is a programming language that allows a programmer to write the programs which are independent of a particular type of computer. The high-level languages are considered as high-level because they are closer to human languages than machine-level languages.

When writing a program in a high-level language, then the whole attention needs to be paid to the logic of the problem.

A compiler is required to translate a high-level language into a low-level language.

1.1.3.1 Advantages of a high-level language

- The high-level language is easy to read, write, and maintain as it is written in English like words.
- The high-level languages are designed to overcome the limitation of low-level language, i.e., portability. The high-level language is portable; i.e., these languages are machine-independent.

1.1.4 Differences between Low-Level language and High-Level language

The following are the differences between low-level language and high-level language:

Low-level language	High-level language
It is a machine-friendly language, i.e., the computer understands the machine language, which is represented in 0 or 1.	It is a user-friendly language as this language is written in simple English words, which can be easily understood by humans.
The low-level language takes more time to execute.	It executes at a faster pace.
It requires the assembler to convert the assembly code into machine code.	It requires the compiler to convert the high-level language instructions into machine code.
The machine code cannot run on all machines, so it is not a portable language.	The high-level code can run all the platforms, so it is a portable language.
It is memory efficient.	It is less memory efficient.
Debugging and maintenance are not easier in a low-level language.	Debugging and maintenance are easier in a high-level language.

1.2 Introduction to Programming Paradigms

A programming paradigm is a style, or “way,” of programming.

1.2.1 Some Common Paradigms

- **Imperative:** Programming with an explicit sequence of commands that update state.
- **Declarative:** Programming by specifying the result you want, not how to get it.
- **Structured:** Programming with clean, goto-free, nested control structures.
- **Procedural:** Imperative programming with procedure calls.
- **Functional (Applicative):** Programming with function calls that avoid any global state.
- **Function-Level (Combinator):** Programming with no variables at all.
- **Object-Oriented:** Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces. Object orientation can be:
 - Class-based: Objects get state and behavior based on membership in a class.
 - Prototype-based: Objects get behavior from a prototype object.
- **Event-Driven:** Programming with emitters and listeners of asynchronous actions.
- **Flow-Driven:** Programming processes communicating with each other over predefined channels.
- **Logic (Rule-based):** Programming by specifying a set of facts and rules. An engine infers the answers to questions.
- **Constraint:** Programming by specifying a set of constraints. An engine finds the values that meet the constraints.
- **Aspect-Oriented:** Programming cross-cutting concerns applied transparently.
- **Reflective:** Programming by manipulating the program elements themselves.
- **Array:** Programming with powerful array operators that usually make loops unnecessary.

1.3 Algorithm

An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. An algorithm is a step by step method of solving a problem. It is commonly used for data processing, calculation and other related computer and mathematical operations.

An algorithm is a detailed series of instructions for carrying out an operation or solving a problem. An algorithm is an unambiguous description that makes clear what has to be implemented.

An algorithm expects a defined set of inputs.

An algorithm produces a defined set of outputs. It might output the larger of the two numbers, an all-uppercase version of a word, or a sorted version of the list of numbers.

An algorithm is guaranteed to terminate and produce a result, always stopping after a finite time. If an algorithm could potentially run forever, it wouldn't be very useful because you might never get an answer. If an algorithm imposes a requirement on its inputs (called a *precondition*), that requirement must be met.

1.3.1 Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

An Example Algorithm

Problem – Design an algorithm to add two numbers and display the result.

Step 1 – START
Step 2 – declare three integers **a**, **b** & **c**
Step 3 – define values of **a** & **b**
Step 4 – add values of **a** & **b**
Step 5 – store output of step 4 to **c**
Step 6 – print **c**
Step 7 – STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

Step 1 – START ADD
Step 2 – get values of **a** & **b**
Step 3 – $c \leftarrow a + b$
Step 4 – display **c**
Step 5 – STOP

1.4 Flowchart for problem solving with Sequential Logic Structure




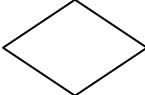
1.4.1 What is a Flowchart?

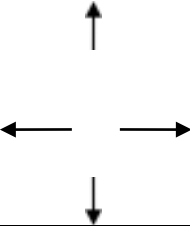



Flowchart as a diagram shows step-by-step progression through a procedure or system especially using connecting lines and a set of conventional symbols.

Flowcharts are graphical representation of steps. It was originated from computer science as a tool for representing algorithms and programming logic, but had extended to use in all other

Kinds of processes. Nowadays, flowcharts play an extremely important role in displaying information and assisting reasoning. They help us visualize complex processes, or make explicit the structure of problems and tasks. A flowchart can also be used to define a process or project to be implemented.

1.4.2 Basic Flowchart Symbols

Symbol Name	Symbol	function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc

Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
		Off Page Connector
		Predefined Process /Function Used to represent a group of statements performing one processing

1.4.3 Advantages of Flowchart:

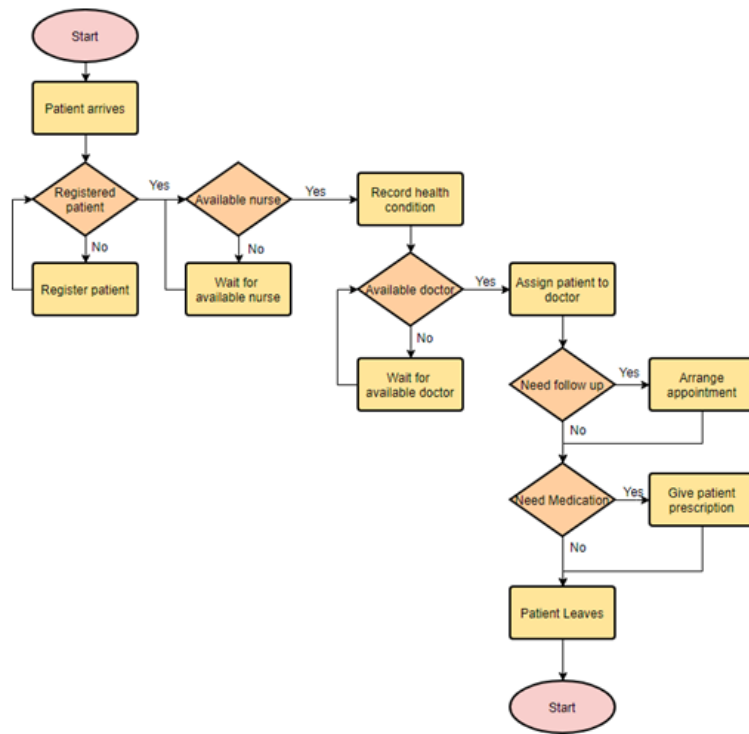
- It helps to clarify complex processes.
- It identifies steps that do not add value to the internal or external customer, including: delays; needless storage and transportation; unnecessary work, duplication, and added expense; breakdowns in communication.
- It helps team members gain a shared understanding of the process and use this knowledge to collect data, identify problems, focus discussions, and identify resources.
- It serves as a basis for designing new processes.

1.4.4 Flowchart examples

Here are several flowchart examples. See how you can apply flowchart practically.

Flowchart Example – Medical Service

This is a hospital flowchart example that shows how clinical cases shall be processed. This flowchart uses decision shapes intensively in representing alternative flows.



1.4.5 Solved examples**Algorithm & Flowchart to find the sum of two numbers****Algorithm**

Step-1 Start

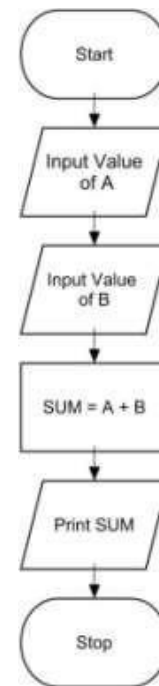
Step-2 Input first numbers say A

Step-3 Input second number say B

Step-4 $SUM = A + B$

Step-5 Display SUM

Step-6 Stop

**Algorithm**

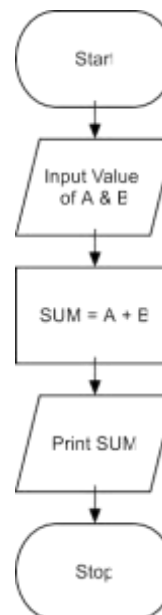
Step-1 Start

Step-2 Input two numbers say A & B

Step-3 $SUM = A + B$

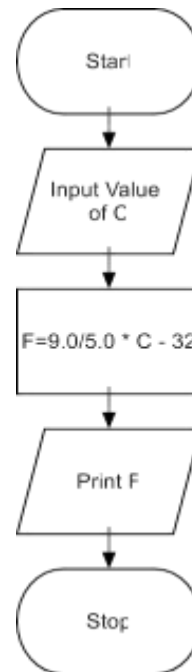
Step-4 Display SUM

Step-5 Stop



Algorithm & Flowchart to convert temperature from Celsius to Fahrenheit**FLOWCHART****Algorithm**

- Step-1 Start
Step-2 Input temperature in Celsius say C
Step-3 $F = (9.0/5.0 \times C) + 32$
Step-4 Display Temperature in Fahrenheit F
Step-5 Stop

**1.5 Introduction to imperative language: Imperative languages**

Imperative programming is a paradigm of computer programming in which the program describes a sequence of steps that change the state of the computer. Unlike declarative programming, which describes "what" a program should accomplish, imperative programming explicitly tells the computer "how" to accomplish it. Programs written this way often compile to binary executables that run more efficiently since all CPU instructions are themselves imperative statements.

Fundamentally, languages can be broken down into two types: **imperative** languages in which you instruct the computer *how* to do a task, and **declarative** languages in which you tell the computer *what* to do. Declarative languages can further be broken down into **functional** languages, in which a program is constructed by composing functions, and **logic** programming languages, in which a program is constructed through a set of logical connections. Imperative languages read more like a list of steps for solving a problem, kind of like a recipe. Imperative languages include C, C++, and Java; functional languages include Haskell; logic programming languages

include Prolog.

Imperative languages are sometimes broken into two subgroups: **procedural** languages like C, and **object-oriented languages**. Object-oriented languages are a bit orthogonal to the groupings, though, as there are object-oriented functional languages (OCaml and Scala being examples).

You can also group languages by typing: **static** and **dynamic**. Statically-typed languages are ones in which typing is checked (and usually enforced) prior to running the program (typically during a compile phase); dynamically-typed languages defer type checking to runtime. C, C++, and Java are statically-typed languages; Python, Ruby, JavaScript, and Objective-C are dynamically-typed languages. There are also **untyped** languages, which include the Forth programming language.

You can also group languages by their typing *discipline*: **weak** typing, which supports implicit type conversions, and **strong** typing, which prohibits implicit type conversions. The lines between the two are a bit blurry: according to some definitions, C is a weakly-typed languages, while others consider it to be strongly-typed. Typing discipline isn't really a useful way to group languages, anyway.

Imperative programming languages

- Ada
- ALGOL
- Assembly language
- BASIC
- Blue
- C
- C#
- C++
- COBOL
- D
- FORTRAN
- Groovy
- Java
- Julia
- MATLAB
- Modula
- Pascal
- Perl
- PHP
- Python
- Ruby

1.5.1 What are Declarative and Imperative Programming?

Declarative programming is a programming paradigm ... that expresses the logic of a computation without describing its control flow.

Imperative programming is a programming paradigm that uses statements that change a program's state.

1.5.2 Imperative vs. declarative programming

Imperative programming contrasts with declarative programming, in which how a problem is solved is not specifically defined, but instead focuses on *what* needs to be solved. Declarative programming provides a constant to check to ensure the problem is solved correctly, but does not provide instructions on how to solve the problem. The exact manner in which the problem is solved is defined by the programming language's implementation through models. Declarative programming is also called model-based programming. Functional, domain-specific (DSL) and logical programming languages fit under declarative programming, such as SQL, HTML, XML and CSS.

A simplified example to distinguish between an imperative and declarative approach is to think of giving driving directions. An imperative approach would provide step by step instructions on how to arrive at a given destination. A declarative approach would provide the address of the destination, without concern about how it's found.

The models from which declarative programming gets its functions are created through imperative programming. As better methods for functions are found through imperative programming, they can be packaged into models to be called upon by declarative programming.

1.6 Syntax and Constructs of a specific language

1.6.1 C Language Introduction

C is a procedural programming language. It was initially developed by Dennis Ritchie in the year 1972. It was mainly developed as a system programming language to write an operating system. The main features of C language include low-level access to memory, a simple set of keywords, and clean style, these features make C language suitable for system programmings like an operating system or compiler development. Many later languages have borrowed syntax/features directly or indirectly from C language. Like syntax of Java, PHP, JavaScript, and many other languages are mainly based on C language. C++ is nearly a superset of C language (There are few programs that may compile in C, but not in C++).

Beginning with C programming:

After the above discussion, we can formally assess the structure of a C program. By structure, it is meant that any program can be written in this structure only. Writing a C program in any other structure will hence lead to a Compilation Error.

1.6.2 The structure of a C program is as follows:

Structure of C Program	
Header	#include <stdio.h>
main()	int main() {
Variable declaration	int a = 10;
Body	printf("%d ", a);
Return	return 0; }

The components of the above structure are:

Header Files Inclusion: The first and foremost component is the inclusion of the Header files in a C program. A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

Some of C Header files:

- stddef.h – Defines several useful types and macros.
- stdint.h – Defines exact width integer types.
- stdio.h – Defines core input and output functions
- stdlib.h – Defines numeric conversion functions, pseudo-random network generator, memory allocation
- string.h – Defines string handling functions
- math.h – Defines common mathematical functions

Syntax to include a header file in C:

#include

Main Method Declaration: The next part of a C program is to declare the main() function. The syntax to declare the main function is:

Syntax to Declare main method:

```
int main()
{
}
```

Variable Declaration: The next part of any C program is the variable declaration. It refers to the variables that are to be used in the function. Please note that in the C program, no variable can be used without being declared. Also in a C program, the variables are to be declared before any operation in the function.

Example:

```
int main()
{
    int a;
    .
    .
}
```

Body: Body of a function in C program, refers to the operations that are performed in the functions. It can be anything like manipulations, searching, sorting, printing, etc.

Example:

```
int main()
{
    int a;

    printf("%d", a);
    .
    .
}
```

Return Statement: The last part in any C program is the return statement. The return statement refers to the returning of the values from a function. This return statement and return value depend upon the return type of the function. For example, if the return type is void, then there will be no return statement. In any other case, there will be a return statement and the return value will be of the type of the specified return type.

Example:

```
int main()
{
    int a;
}
```

```
    printf("%d", a);  
  
    return 0;  
}
```

STEPS FOR EXECUTING THE PROGRAM

Creation of program

Programs should be written in C editor. The file name does not necessarily include extension

C. The default

extension is C.

Compilation of a

program

The source program statements should be translated into object programs which is suitable for execution by the computer. The translation is done after correcting each statement. If there is no error, compilation proceeds and translated program are stored in another file with the same file name with extension—.objl.

Execution of the program

After the compilation the executable object code will be loaded in the computers main memory and the program is executed.

Assignment Symbol (\leftarrow or $=$) is used to assign value to the variable.

The statement $C = A + B$ means that add the value stored in variable A and variable B then assign/store the value in variable C.

The statement $R = R + 1$ means that add 1 to the value stored in variable R and then assign/store the new value in variable R, in other words increase the value of variable R by 1

Keywords in C Programming Language:

1. Keywords are those words whose meaning is already defined by Compiler
2. Cannot be used as **Variable Name**

3. There are **32 Keywords** in C
4. C Keywords are also called as **Reserved words**.

Keywords in C Programming			
auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

1.7 Types Operator and Expressions

1.7.1 Variable Names

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, playerScore is a variable of int type. Here, the variable is assigned an integer value 95.

The value of a variable can be changed, hence the name variable.

```
char ch = 'a';  
// some code
```

```
ch = 'l';
```

1.7.1.1 Rules for naming a variable

1. A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore.
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

C is a strongly typed language. This means that the variable type cannot be changed once it is declared. For example:

```
int number = 5;      // integer variable
number = 5.5;        // error
double number;       // error
```

Here, the type of number variable is int. You cannot assign a floating-point (decimal) value 5.5 to this variable. Also, you cannot redefine the data type of the variable to double. By the way, to store the decimal values in C, you need to declare its type to either double or float.

1.7.2 Data Type and Sizes

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int myVar;
```

Here, `myVar` is a variable of `int` (integer) type. The size of `int` is 4 bytes.

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
<code>int</code>	at least 2, usually 4	<code>%d, %i</code>
<code>char</code>	1	<code>%c</code>
<code>float</code>	4	<code>%f</code>
<code>double</code>	8	<code>%lf</code>
<code>short int</code>	2 usually	<code>%hd</code>
<code>unsigned int</code>	at least 2, usually 4	<code>%u</code>
<code>long int</code>	at least 4, usually 8	<code>%ld, %li</code>
<code>long long int</code>	at least 8	<code>%lld, %lli</code>
<code>unsigned long int</code>	at least 4	<code>%lu</code>
<code>unsigned long long int</code>	at least 8	<code>%llu</code>
<code>signed char</code>	1	<code>%c</code>
<code>unsigned char</code>	1	<code>%c</code>
<code>long double</code>	at least 10, usually 12 or 16	<code>%Lf</code>

1.7.2.1 int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10

We can use `int` for declaring an integer variable.

```
int id;
```

Here, `id` is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of `int` is usually 4 bytes (32 bits). And, it can take 2^{32} distinct states from -2147483648 to 2147483647.

1.7.2.2 float and double

`float` and `double` are used to hold real numbers.

```
float salary;  
double price;
```

In C, floating-point numbers can also be represented in exponential. For example,

```
float normalizationFactor = 22.442e2;
```

What's the difference between `float` and `double`?

The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes.

1.7.2.3 char

Keyword `char` is used for declaring character type variables. For example,

```
char test = 'h';
```

The size of the character variable is 1 byte.

1.7.2.4 void

void is an incomplete type. It means "nothing" or "no type". You can think of void as **absent**.

For example, if a function is not returning anything, its return type should be **void**.

Note that, you cannot create variables of **void** type.

1.7.2.5 short and long

If you need to use a large number, you can use a type specifier **long**. Here's how:

```
long a;  
long long b;  
long double c;
```

Here variables **a** and **b** can store integer values. And, c can store a floating-point number.

If you are sure, only a small integer (**[-32,767, +32,767]** range) will be used, you can use **short**.

```
short d;
```

You can always check the size of a variable using the **sizeof()** operator.

```
#include <stdio.h>  
int main() {  
    short a;  
    long b;  
    long long c;  
    long double d;  
  
    printf("size of short = %d bytes\n", sizeof(a));  
    printf("size of long = %d bytes\n", sizeof(b));  
    printf("size of long long = %d bytes\n", sizeof(c));  
    printf("size of long double= %d bytes\n", sizeof(d));  
    return 0;  
}
```

1.7.2.6 Signed and unsigned

In C, `signed` and `unsigned` are type modifiers. You can alter the data storage of a data type by using them. For example,

```
unsigned int x;  
int y;
```

Here, the variable `x` can hold only zero and positive values because we have used the `unsigned` modifier.

Considering the size of `int` is 4 bytes, variable `y` can hold values from -2^{31} to $2^{31}-1$, whereas variable `x` can hold values from 0 to $2^{32}-1$.

Other data types defined in C programming are:

- `bool` Type
- Enumerated type
- Complex types

1.7.2.7 Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

1.7.3 Little and Big Endian

Little and big endian are two ways of storing multibyte data-types (`int`, `float`, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable `x` with value `0x01234567` will be stored as following.



1.7.3.1 Big endian vs. little endian

Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address). Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first.

For people who use languages that read left-to-right, big endian seems like the natural way to think of storing a string of characters or numbers - in the same order you expect to see it presented to you. Many of us would thus think of big-endian as storing something in forward fashion, just as we read.

Note that within both big-endian and little-endian byte orders, the bits within each byte are bigendian.

That is, there is no attempt to be big- or little-endian about the entire bit stream represented by a given number of stored bytes. For example, whether hexadecimal 4F is put in storage first or last with other bytes in a given storage address range, the bit order within the byte will be:

01001111

Both big endian and little endian are widely used in digital electronics. The endianness in use is typically determined by the CPU.

On the other hand, Intel processors (CPUs) and DEC Alphas and at least some programs that run on them are little-endian.

Language compilers such as that of Java or FORTRAN have to know which way the object code

they develop is going to be stored. Converters can be used to change one kind of endian to the other when necessary.

The ordering of bytes in a 16-bit word differs from the ordering of 16-bit words within a 32-bit word. Bi-endian processors can operate in either little-endian or big-endian mode, and switch between the two.

1.7.4 Constants

If you want to define a variable whose value cannot be changed, you can use the `const` keyword. This will create a constant. For example,

```
const double PI = 3.14;
```

Notice, we have added keyword `const`.

Here, `PI` is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;  
PI = 2.9; //Error
```

1.7.5 Declarations

Before you can use a variable in C, it must be defined in a *declaration statement*.

A variable declaration serves three purposes:

1. It defines the name of the variable.
2. It defines the type of the variable (integer, real, character, etc.).
3. It gives the programmer a description of the variable. The declaration of a variable answer can be:

```
int answer;    /* the result of our expression */
```

The keyword **int** tells C that this variable contains an integer value. (Integers are defined below.) The variable name is answer. The semicolon (;) marks the end of the statement, and the comment

is used to define this variable for the programmer. (The requirement that every C variable declaration be commented is a style rule. C will allow you to omit the comment. Any experienced teacher, manager, or lead engineer will not.)

The general form of a variable declaration is:

```
type name; /*comment */
```

where *type* is one of the C variable types (**int**, **float**, etc.) and *name* is any valid variable name. This declaration explains what the variable is and what it will be used for. Variable declarations appear just before the `main()` line at the top of a program.

1.7.6 Types of Operators

Types of Operators	Description
Arithmetic_operators	These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus
Assignment_operators	These are used to assign the values for the variables in C programs.
Relational operators	These operators are used to compare the value of two variables.
Logical operators	These operators are used to perform logical operations on the given two variables.
Bit wise operators	These operators are used to perform bit operations on given two variables.
Conditional (ternary) operators	Conditional operators return one value if condition is true and returns another value if condition is false.
Increment/decrement operators	These operators are used to either increase or decrease the value of the variable by one.
Special operators	&, *, sizeof() and ternary operators.

1.7.6.1 Arithmetic operators (+, -, *, /, **, %)

Arithmetic operators perform mathematical operations such as addition and subtraction with

operands. There are two types of mathematical operators: unary and binary. Unary operators perform an action with a single operand. Binary operators perform actions with two operands. In

a complex expression, (two or more operands) the order of evaluation depends on precedence rules.

Unary arithmetic operators

Unary operators are arithmetic operators that perform an action on a single operand. The script language recognizes the unary operator negative (-).

The negative unary operator reverses the sign of an expression from positive to negative or vice-versa. The net effect is that of multiplying the number by -1. Example:

Binary arithmetic operators

Insert a space before and after an arithmetic operator. The binary arithmetic operators that are supported are listed below.

Symbol	Operation	Example	Description
+	Addition	a + b	Add the two operands
-	Subtraction	a - b	Subtract the second operand from the first operand
*	Multiplication	a * b	Multiply the two operands
/	Division	a / b	Divide the first operand by the second operand
**	Power	a ** b	Raise the first operand by the power of the second operand
%	Modulo	a % b	Divide the first operand by the second operand and yield the remainder portion
Binary Arithmetic Operators			

1.7.6.2 Operator precedence

Expressions are normally evaluated left to right. Complex expressions are evaluated one at a time. The order in which the expressions are evaluated is determined by the precedence of the operators used. The standard C ordering is followed.

1. negation (-) unary
2. power
3. multiplication, division and modulo
4. addition and subtraction

If an expression contains two or more operators with the same precedence, the operator to the left is evaluated first. For example, $10 / 2 * 5$ will be evaluated as $(10 / 2)$ and the result multiplied by 5.

When a lower precedence operation should be processed first, it should be enclosed within parentheses. For example, $30 / 2 + 8$. This is normally evaluated as 30 divided by 2 then 8 added to the result. If you want to divide by $2 + 8$, it should be written as $30 / (2 + 8)$.

Parentheses can be nested within expressions. Innermost parenthetical expressions are evaluated first.

1.7.6.3 Assignment Operator (=)

Use the assignment operator (=) to copy a constant, literal, variable expression result, or function result to a variable. The script language does not support multiple assignments in a single statement (such as $a=b=c=0$). String lengths are defined based on the size of the string assigned to the variable and can change dynamically at runtime.

1.7.6.4 Logical Operators (AND, OR)

Logical operators allow the combining of more than one relational test in one comparison. Logical operators return a TRUE (1) or FALSE (0) value. Logical operators have a lower precedence than arithmetic operators.

Operator	Example	Meaning
AND	$A < B \text{ AND } B < C$	Result is True if both $A < B$ and $B < C$ are true else false
OR	$A < B \text{ OR } B < C$	Result is True if either $A < B$ or $B < C$ are true else false
NOT	$\text{NOT } (A > B)$	Result is True if $A > B$ is false else true

1.7.6.5 Bitwise Operators

C provides the following binary operators for manipulating individual bits inside of integer operands. These operators all have the same meaning as in ANSI-C.

Bitwise Operators

	bitwise OR
&	Bitwise AND
^	bitwise XOR
<<	shift the left-hand operand left by the number of bits specified by the right-hand operand
>>	shift the left-hand operand right by the number of bits specified by the right-hand operand

The binary & operator is used to clear bits from an integer operand. The binary | operator is used to set bits in an integer operand. The binary ^ operator returns one in each bit position where exactly one of the corresponding operand bits is set.

The shift operators are used to move bits left or right in a given integer operand. Shifting left fills empty bit positions on the right-hand side of the result with zeroes. Shifting right using an unsigned integer operand fills empty bit positions on the left-hand side of the result with zeroes. Shifting right using a signed integer operand fills empty bit positions on the left-hand side with the value of the sign bit, also known as an arithmetic shift operation.

In addition to the binary logical operators, the unary ~ operator may be used to perform a bitwise negation of a single operand: it converts each zero bit in the operand into a one bit, and each one bit in the operand into a zero bit.

1.7.6.6 Relational Operators

Relational operators are as follows:

Symbol	Operation	Example	Description
<	Less than	a < b	True if a is less than b.
>	Greater than	a GT b	True if a is greater than b.
==	Equal	a == b	True if a is equal to b.

!=	Not equal	a NE b	True if a is not equal to b.
<=	Less than or equal	a <= b	True if a is less than or equal to b.
>=	Greater than or equal	a GE b	True if a is greater than or equal to b.
Relational Operators			

Comparisons must be made on like data types—string to string, numbers to numbers. If the data types are different, a run time error will be raised.

1.7.6.7 Assignment Operators

C provides the following binary assignment operators for modifying the variables. You can only modify the variables and arrays. Kernel data objects and constants may not be modified using the the assignment operators. The assignment operators have the same meaning as they do in ANSI-C.

Assignment Operators

=	set the left-hand operand equal to the right-hand expression value
+=	increment the left-hand operand by the right-hand expression value
-=	decrement the left-hand operand by the right-hand expression value
*=	multiply the left-hand operand by the right-hand expression value
/=	divide the left-hand operand by the right-hand expression value
%=	modulo the left-hand operand by the right-hand expression value
=	bitwise OR the left-hand operand with the right-hand expression value
&=	bitwise AND the left-hand operand with the right-hand expression value
^=	bitwise XOR the left-hand operand with the right-hand expression value
<<=	shift the left-hand operand left by the number of bits specified by the right-hand expression value

>>=	shift the left-hand operand right by the number of bits specified by the right-hand expression value
-----	--

Aside from the assignment operator =, the other assignment operators are provided as shorthand for using the = operator with one of the other operators described earlier. For example, the expression `x = x + 1` is equivalent to the expression `x += 1`, except that the expression `x` is evaluated once. These assignment operators obey the same rules for operand types as the binary forms described earlier.

The result of any assignment operator is an expression equal to the new value of the left-hand expression. You can use the assignment operators or any of the operators described so far in

combination to form expressions of arbitrary complexity. You can use parentheses () to group terms in complex expressions.

1.7.6.8 Increment and Decrement Operators

C provides the special unary `++` and `--` operators for incrementing and decrementing pointers and integers. These operators have the same meaning as in ANSI-C. These operators can only be applied to variables, and may be applied either before or after the variable name. If the operator appears before the variable name, the variable is first modified and then the resulting expression is equal to the new value of the variable. For example, the following two expressions produce identical results:

<code>x += 1;</code>	<code>y = ++x;</code>
<code>y = x;</code>	

If the operator appears after the variable name, then the variable is modified after its current value is returned for use in the expression. For example, the following two expressions produce identical results:

<code>y = x;</code>	<code>y = x--;</code>
<code>x -= 1;</code>	

You can use the increment and decrement operators to create new variables without declaring them. If a variable declaration is omitted and the increment or decrement operator is applied to a variable, the variable is implicitly declared to be of type `int64_t`.

The increment and decrement operators can be applied to integer or pointer variables. When applied to integer variables, the operators increment or decrement the corresponding value by one. When applied to pointer variables, the operators increment or decrement the pointer address by the size of the data type referenced by the pointer. Pointers and pointer arithmetic in D are discussed in Pointers and Arrays.

1.7.6.9 Conditional Expressions

In this example, the expression `i == 0` is first evaluated to determine whether it is true or false. If the first expression is true, the second expression is evaluated and the `?:` expression returns its value. If the first expression is false, the third expression is evaluated and the `?:` expression return its value.

As with any C operator, you can use multiple `?:` operators in a single expression to create more complex expressions. For example, the following expression would take

a char variable `c` containing one of the characters 0-9, a-z, or A-Z and return the value of this character when interpreted as a digit in a hexadecimal (base 16) integer:

```
hexval = (c >= '0' && c <= '9') ? c - '0' :
```

```
(c >= 'a' && c <= 'z') ? c + 10 - 'a' : c + 10 - 'A';
```

The first expression used with `?:` must be a pointer or integer in order to be evaluated for its truth value. The second and third expressions may be of any compatible types. You may not construct a conditional expression where, for example, one path returns a string and another path returns an integer. The second and third expressions also may not invoke a tracing function such as `trace` or `printf`. If you want to conditionally trace data, use a predicate instead, as discussed in Introduction.

1.7.6.10 Type Conversions

When expressions are constructed using operands of different but compatible types, type conversions are performed in order to determine the type of the resulting expression. A simple way to describe the conversion rules is as follows: each integer type is ranked in the order `char`, `short`, `int`, `long`, `long long`, with the corresponding unsigned types assigned a rank above its signed equivalent but below the next integer type. When you construct an expression using two integer operands such as `x + y` and the operands are of different integer types, the operand type with the highest rank is used as the result type.

If a conversion is required, the operand of lower rank is first promoted to the type of higher rank. Promotion does not actually change the value of the operand: it simply extends the value to a larger container according to its sign. If an unsigned operand is promoted, the unused high-order bits of the resulting integer are filled with zeroes. If a signed operand is promoted, the unused high-order bits are filled by performing sign

extension. If a signed type is converted to an unsigned type, the signed type is first sign-extended and then assigned the new unsigned type determined by the conversion.

Integers and other types can also be explicitly cast from one type to another. `y = (int)x;`

1.7.6.11 Precedence

The C rules for operator precedence and associativity are described in the following table. The table entries are in order from highest precedence to lowest precedence.

Operator Precedence and Associative

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof stringof offsetof xlate	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
^^	left to right
	left to right
?:	right to left

= += -= *= /= %= &= ^= = <<= >>=	right to left
,	left to right
sizeof	Computes the size of an object (Structs and Unions)
offsetof	Computes the offset of a type member (Structs and Unions)
stringof	Converts the operand to a string (Strings)
xlate	Translates a data type (Translators)
unary &	Computes the address of an object (Pointers and Arrays)
unary *	Dereferences a pointer to an object (Pointers and Arrays)
-> and .	Accesses a member of a structure or union type (Structs and Unions)

1.7.6.12 Hungarian Notation.

Origins of Hungarian notation

Charles Simonya, chief architect at Microsoft is the originator of the Hungarian Notation standard, which is used extensively in Microsoft Windows code. Simonya first used the notation in 1972. The notation was referred to as *Hungarian* originally as a criticism. At first glance, identifiers using Hungarian Notation appear to be gibberish until the pattern is deduced. Friends of Simonya compared Simonya's notation convention to some obscure foreign language and since Simonya is Hungarian, that was the obscure foreign language referred to.

Since its inception in 1972, Hungarian Notation has been adopted by Xerox, Apple, 3Com, and of course Microsoft.

. Hungarian notation is a naming convention in computer programming that indicates either the type of object or the way it should be used. ... There are two variations of Hungarian notation: Systems and Apps. They both involve using a special prefix as part of the

name to indicate an object's nature.

Hungarian notation is a naming convention in computer programming that indicates either the type of object or the way it should be used. It was originally proposed by Charles Simonyi, a programmer at Xerox PARC in the early 1980s. There are two variations of Hungarian notation: Systems and Apps. They both involve using a special prefix as part of the name to indicate an object's nature.

Hungarian notation prefixes

The prefix used is up to the programmer, but standard prefixes include:

- b for boolean
- ch for char
- w for word
- dw for double word
- i for integer
- f or fp for floating-point
- d or db for double-precision floating point
- p for pointer
- u32 for unsigned 32-bit integer
- fn for function

Prefix	Meaning	Example	Notes
p	Pointer	Finger* pRude;	In most cases, <i>p</i> is combined with another prefix; the prefix of the type of object being pointed to. For example: String* psName is a pointer to a string object containing a name.
s str	String	String sName; String strName;	This convention is generally used for first-class string classes.
sz psz	zero-terminated / null-terminated string	char szName[16]; char* pszName;	
h	Handle	HWND hWindow	
c	Character (char)	char cLetter;	Sometimes <i>c</i> is used to denote a counter object.
by y	Byte or Unsigned Char	byte byMouthFull; byte yMouthFull;	
n	Integer (int)	int nSizeOfArray;	

f	Float	float fRootBeer;	
d	Double	double dDecker;	
b	Boolean	boolean bIsTrue; BOOL bIsTrue; int bIsTrue;	An integer can store a boolean value as long as you remember not to assign it a value other than 0 or 1
u	Unsigned...		
w	Word or Unsigned Integer	unsigned int wValue;	
l	Long	long lIdentifier;	Sometimes <i>l</i> is appended to <i>p</i> to denote that the pointer is a long. For example: lpszName is a long pointer to a zero-terminated string.
dw	Unsigned Long Integer		
C or just a capital first letter	Class	Class CObject; Class Object;	C is used heavily in Microsoft's Foundation Classes but using just a capital first letter is emphasized by Microsoft's J++.
I	Interface (usually a struct or class with only pure virtual methods and no member variables)	class IMotion { public: virtual void Fly() = 0; };	Used extensively in COM.
X	Nested Class	class CRocket { public: class XMotion:public IMotion { public: void Fly(); } m_xUnknown; }	Used extensively in COM.
x	Instantiation of a nested class.	class CAirplane { public: class XMotion:public IMotion { public: void Fly(); } m_xUnknown; }	Used extensively in COM.
m_	Class Member Identifiers	class CThing {	

		private: int m_nSize; };	
g_	Global	String* g_psBuffer	Constant globals are usually in all caps. The g_ would denote that a particular global is not a constant.
v	Void (no type)	void* pvObject	In most cases, v will be included with p because it is a common trick to typecast pointers to void pointers.

Hungarian notation is an identifier naming convention in computer programming, in which the name of a variable or function indicates its intention or kind, and in some dialects its type. The original Hungarian Notation uses intention or kind in its naming convention and is sometimes called Apps Hungarian as it became popular in the Microsoft Apps division in the development of Word, Excel and other apps. As the Microsoft Windows division adopted the naming convention, they used the actual data type for naming, and this convention became widely spread through the Windows API; this is sometimes called Systems Hungarian notation. Hungarian notation was designed to be language-independent, and found its first major use with the BCPL programming language. Because BCPL has no data types other than the machine word, nothing in the language itself helps a programmer remember variables' types. Hungarian notation aims to remedy this by providing the programmer with explicit knowledge of each variable's data type.