Unit - 2

Chapter 4

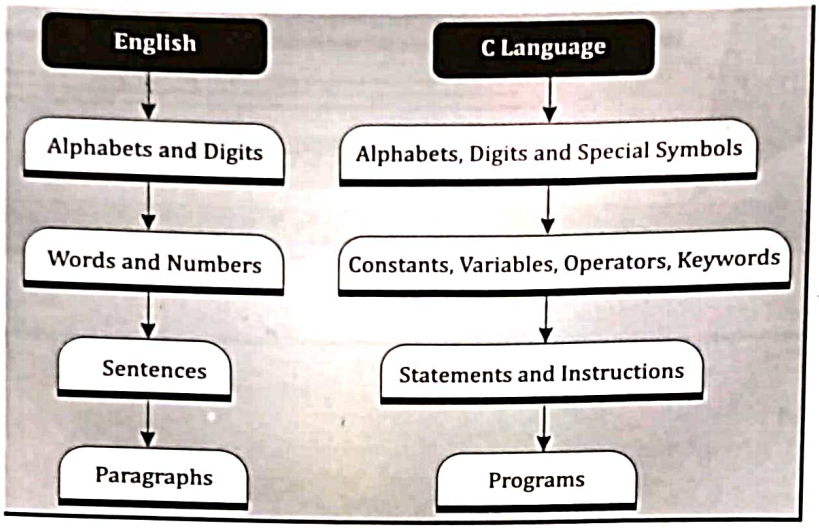
C Programming Basic Concepts

Chapter Outline

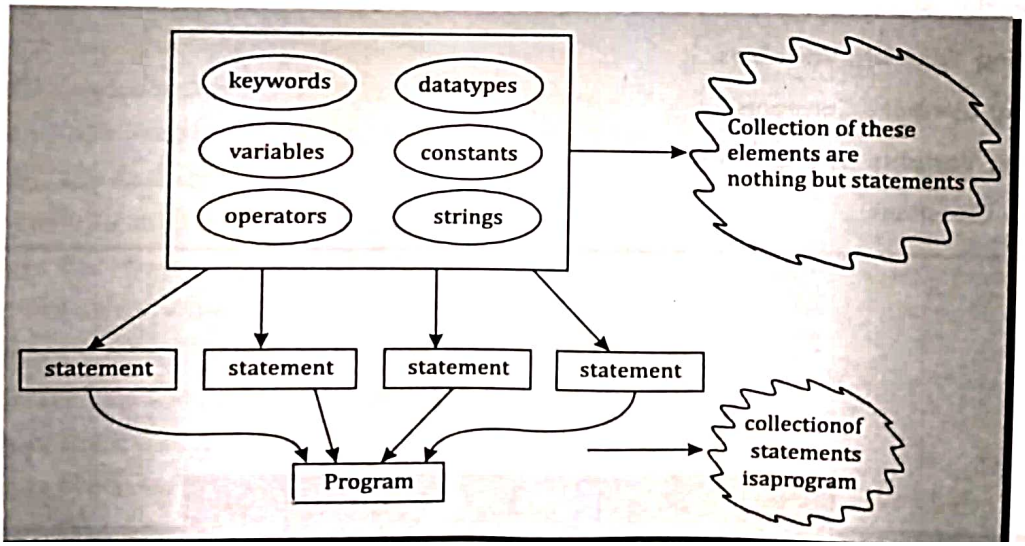
- Introduction
- C Character Set
- C Tokens
- Keywords
- Identifiers
- Constants
- Variables
- Data Types
- Declaration of Variables
- Assigning Values to Variables
- Defining Symbolic Constants
- Defining Variable as a Constant
- Review Questions

4.1 Introduction

All of us would agree that we began our journey of learning English language by first learning alphabets which are the basic building blocks of English language. There alphabets were combined to form words, words combined to form sentences, sentences combined to for paragraphs & paragraphs combined to form a story. Obviously, we could not have written a story without knowing the alphabets and words in English. In a similar way, we can't write a program in C language without knowing the characters and token of C language.



In the previous chapter, we discussed the programming structure of C programs. All programming languages are designed to support certain kind of data, such as numbers, characters, strings etc; to get useful output known as information. A program is a set of statements, which will be executed in a sequential form. These statements are formed using C character set, identifiers, variables, data types, constants, etc,. The main objective of this chapter to introduce these basic elements which are used for C programming.



4.2 C Character Set

Character set of a language is set of all the symbols used to write a program in that language. They have been taken from English language. We can think of the C character set consisting of all the characters one could type on a standard English keyboard: for example, the digits; uppercase and lowercase letters; and special characters such as +, =, <, >, &, and %. The C character set is used to form words, numbers, statements and expressions.

The characters in C are grouped into four categories.

1. Letters / Alphabets
2. Digits
3. Special Symbols
4. Whitespaces

Letters/Alphabets	Uppercase Letters (A, B, C, D ... X, Y, Z) Lowercase Letters (a, b, c, d, ... x, y, z)
Digits	All decimal digits - 0,1,2,3,4,5,6,7,8,9
White Spaces	blank space, horizontal tab, carriage return, new-line, form-feed
Special Symbols	

,	Comma	&	Amphersand
.	Period or dot	^	Caret
;	Semi colon	*	Asterisk
:	Colon	-	Minus
?	Question mark	+	Plus
'	Apostrophe	<	Less than
"	Quotation mark	>	Greater than
!	Exclamation mark	( )	Parenthesis left/right
|	Vertical bar	[ ]	Brackets left/right
/	Slash	{ }	Braces left/right
\	Back slash	%	Percent
~	Tilde	#	Number sign or hash
_	Underscore	=	Equal to
$	Dollor	@	At the rate

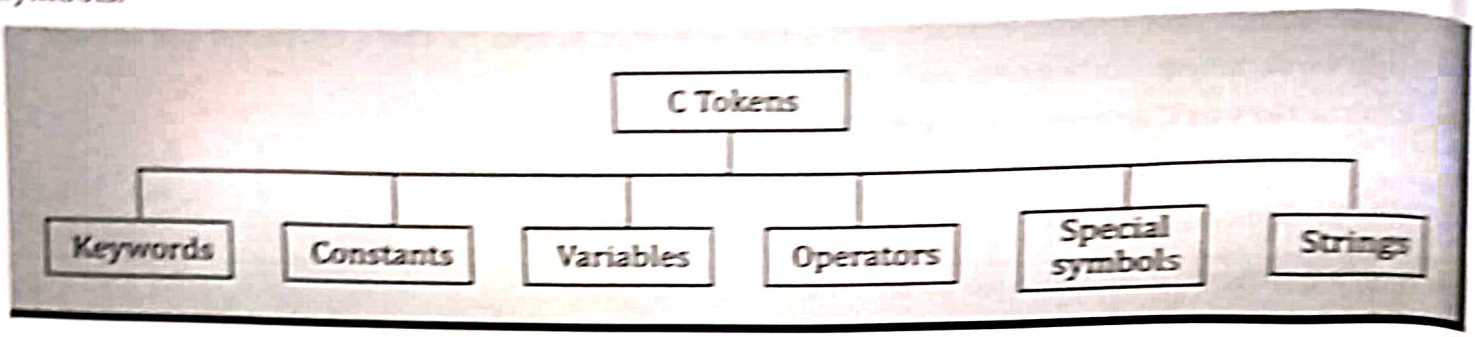
Note

- A – Z and a – z are used for naming variables (identifiers) and keywords that intern are used for building statements.
- The numbers 0 – 9 are used for forming numerical constants and are also used for naming variables.
- Special characters such as +, - * etc are used as arithmetic operators and < , > , ! , &&, ||, != etc are used as relational and logical operators. (more information about the types of operators is available in the next chapter).

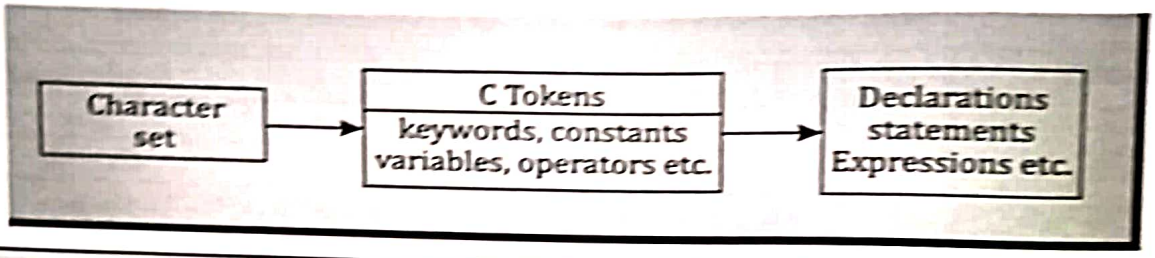
4.3 C Tokens

One or more characters grouped together to form basic elements of C are known as C Tokens.

The C tokens include identifiers, constants, variables, keywords, operators, strings and some special symbols.



We know that one or more characters are grouped to form basic elements of C known as **tokens** which intern are used for building declarations, statements and expressions etc



Note:

The tokens of a language are the basic building blocks that can be put together to construct programs. A token can be a reserved word (such as int or while), an identifier (such as b or sum), a constant (such as 25 or "Programming"), a delimiter (such as } or ;) or an operator (such as + or =).

4.4 Keywords

Every word in C language is classified as either a keyword or an identifier.

The keywords are reserved for specific meanings in C language. .When keywords are combined with the formal C syntax, it forms the C programming language.

The variables should not be named with any of the keywords because all the keywords in C have fixed meaning and they cannot be changed.

Important to Know

- All the keywords are lower case letters. Thus int is a keyword but Int and INT are not.
- Keywords are also called as reserved words.
- All the keywords have pre-defined meanings. They cannot be redefined or used in other contexts.
- Keywords are reserved, that is, we cannot use them as identifiers.

Reserved C keywords

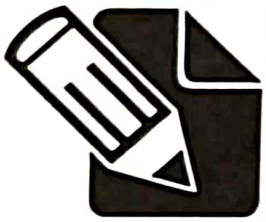
Keyword	Description
asm	Keyword that denotes in line assembly language code.
auto	The default storage class.
break	Command that exits for, while, switch, and do...while statements unconditionally.
case	Command used within the switch statement.
char	The simplest C data type.
const	Data modifier that prevents a variable from being changed. See volatile.
continue	Command that resets a for, while, or do...while statement to the next iteration.
default	Command used within the switch statement to catch any instances not specified with a case statement.
do	Looping command used in conjunction with the while statement. The loop will always execute at least once.
double	Data type that can hold double-precision floating-point values.
else	Statement signaling alternative statements to be executed when an if statement evaluates to FALSE.
enum	Data type that allows variables to be declared that accept only certain values.
extern	Data modifier indicating that a variable will be declared in another area of the program.
float	Data type used for floating-point numbers.
for	Looping command that contains initialization, incrementation, and conditional sections.
goto	Command that causes a jump to a predefined label.
if	Command used to change program flow based on a TRUE/FALSE decision.
int	Data type used to hold integer values.
long	Data type used to hold larger integer values than int.
register	Storage modifier that specifies that a variable should be stored in a register if possible.
return	Command that causes program flow to exit from the current function and return to the calling function. It can also be used to return a single value.
short	Data type used to hold integers. It isn't commonly used, and it's the same size as an int on most computers.
signed	Modifier used to signify that a variable can have both positive and negative values. See unsigned.
sizeof	Operator that returns the size of the item in bytes.
static	Modifier used to signify that the compiler should retain the variable's value between the function calls.
struct	Keyword used to combine C variables of any data type into a group.
switch	Command used to change program flow in multiple directions. Used in conjunction with the case statement.

typedef	Modifier used to create new names for existing variable and function types.
union	Keyword used to allow multiple variables to share the same memory space.
unsigned	Modifier used to signify that a variable will contain only positive values. See signed.
void	Keyword used to signify either that a function doesn't return anything or that a pointer being used is considered generic or able to point to any data type.
volatile	Modifier that signifies that a variable can be changed. See const.
while	Looping statement that executes a section of code as long as a condition remains TRUE.

4.5 Identifiers

A C identifier is a name given to a variable, function, or any other user defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_'followed by zero or more letters, underscores, and digits (0 to 9).

In C, the names of variables, arrays, functions, labels and various user-defined items are called **Identifiers.** The variables, arrays, functions etc can be identified by giving them meaningful names. The names given to them are nothing but **identifiers.**



Rules for Identifier

1. The first character of an identifier must be an alphabet or an underscore (_).
2. Identifiers must consist of only alphabets, digits or underscore. Special symbols are not allowed.
3. The maximum length of an identifier is 31 characters.
4. Keywords should not be used as the identifiers.
5. Identifier should be single word. i.e., No blank space is allowed.
6. Identifiers can be both upper-case and lower-case letters. C is case sensitive, i.e., upper-case letters are different from lower-case letters. **Example: ADD, AdD, add** are different identifiers

Example

Valid Identifiers	Invalid Identifiers	
ADD	3add	(numerics cannot be in the beginning)
Add	add+123	(+ is not allowed)
_Add	for	(it is a keyword)
Student_Name	ABC XYZ	(Blank not allowed)
Name	Student%123	(% is not allowed)
A312B	_Add*5Add	(* is not allowed)

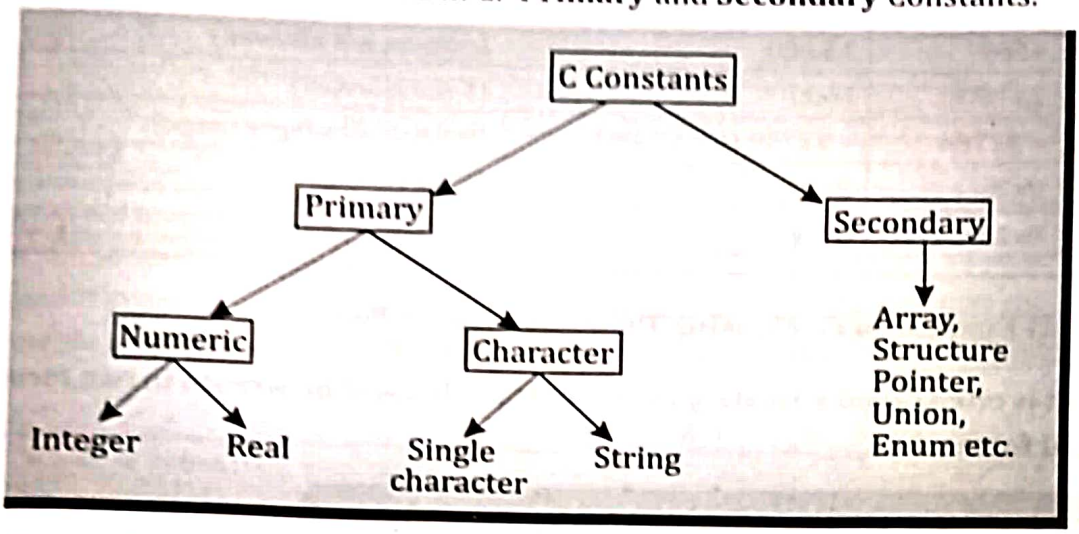
Tip

▲ If we need an identifier that consists of two or more words, use a combination of uppercase and lowercase letters (like sumOfSeries, areaOfTraiangle) or use the underscore to separate the words (like sum_of_series, area_of_triangle). It is preferable to use uppercase/lowercase combination.

4.6 Constants

Constants in C refer to the **fixed values** that do not change during the execution of a program. (Or) A constant is a token with fixed value that does not change.

There are basically two types of constants in C: **Primary** and **Secondary** Constants.



There are two types of numeric constants: *Integer* & *Real constants*. There are two kinds of Character Constants: ***Single character* & *String constants***. At this stage we would restrict our discussion to only Primary Constants namely, **Numeric & Character Constants.**

4.6.1 Integer Constants

An integer constant refers to a *sequence of digits*. There are three types of integers, namely, *decimal*, *octal* and *hexadecimal*.

1. **Decimal Integers :** It consist of a set of digits **0** through **9**.
2. **Octal Integers :** It consist of any combination of digits from **0** through **7**, with a leading **0**.
3. **Hexa Decimal Integers**: It consist of the digits preceded by **'0x'** or **'0X'.** They may include alphabets **A/a** through **F/f,** where the letters **A/a** through **F/f** represent the numbers **10** through **15.**

Rules for constructing Integer Constants other than the above mentioned points:

1. It must not have a decimal point.
2. It could be either positive or negative. If no sign precedes constant, it is assumed to be positive.
3. Embedded spaces/blanks, commas, and non-numeric characters are not permitted.
4. The allowable range for integer constants is -32768 to 32767 for 16-bit machines and -2,147,483,648 to 2,147,483,647 for 32-bit machines. It is also possible to store large integer constants on these machines by appending qualifiers such as U/u, L/l, and UL/ul to the constants.

Example

Valid	Invalid	
678	25 000	(space not allowed)
-333	15.50	(decimal not allowed)
+666	15,000	(comma not allowed)
24700	$600	($ not allowed)
- 32766	- 32769 (for 16-bit)	(outside of integer range)
0x2	2x3	(2x is invalid)
0x2d/0x2D	0x2g	(alphabets a to f is allowed but not g)

4.6.2 Real Constants or Floating Point Constants

A real constant is often called a Floating Point Constant. It could be written in two forms: **Fractional & Exponential forms.**

Rules for constructing Real Constants for Fractional form:

1. It must have at least one digit (**0 to 9**).
2. It must have a decimal point.
3. It could be either positive or negative. Default sign is positive.
4. Embedded spaces, commas are **not** allowed with in a real constant.

Rules for constructing Real Constants for Exponential form:

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large.

Here, the real constant is represented in two parts. The part before **'e'** is called **mantissa**, where as the part following 'e' is called **exponent**.

The general syntax is:

```
mantissa e exponent
```

1. The mantissa part and the exponential part should be separated by letter 'e'.
2. Mantissa part could be either positive or negative. Default sign is positive.
3. Exponent must have at least one digit which must be a positive or negative integer. Default is +ve.
4. Blank is not allowed.
5. Range of Real Constants expressed in Exponential form is $-3.4e^{38}$ to $3.4e^{38}$.

Example

Valid	Invalid	
426.00	666	(No decimal)
-66.6666	$20060	($ not allowed)
+.5	25 000 00	(Space not allowed)
+3.2e-5	+3.2e 5	(space between e and 5 not allowed)
−0.2e+3	−0.2 +3.5	('e' missing)

4.6.3 Character Constants

A character constant consists of a single character, single digit, or a single special symbol enclosed within a pair of single inverted commas. The maximum length of a character constant is one character.

Example

```
'a' is a character constant
'd' is a character constant
'P' is a character constant
'5' is a character constant
'*' is a character constant
```

Note

'5' here is different from number 5. '5' in the above example is a character.

4.6.4 String Constants

A string constant is a sequence of one or more characters enclosed within a pair of double quotes (" "). If a single character is enclosed within a pair of double quotes, it will also be interpreted as a string constant and not a character constant. The characters may be letters, numbers, special characters, and blank space.

Example

```
"Hello"    "1987"    "&...!"    "X"    "5"    "C Programming"    "Srikanth"
```

Note

"X" here is different from character 'X'. "X" in the above example is a String.

Actually, a string is an array of characters terminated by a NULL character. Thus, "a" is a string consisting of two characters, viz. 'a' and NULL('\0').

4.7 Variables

A variable is an entity whose value can change during program execution. A variable can be thought of as a symbolic representation of address of the memory space where values can be stored, accessed and changed. A specific location or address in the memory is allocated for each variable and the value of that variable is stored in that location.

Each variable has a name, data-type, size, and the value it stores. All the variables must have their type indicated so that the compiler can record all the necessary information about them; generate the appropriate code during translation and allocating required space in memory.

Every programming language has its own set of rules that must be observed while writing the names of variables. If the rules are not followed, the compiler reports compilation error.

A variable is a named location in memory that is used to hold a value that can be modified by the program. A variable can be assigned different values at different times during the execution of a program. All variables must be declared before they can be used. The variable declaration tells two things:

1. It tells the compiler the name of the variable.
2. It also tells the type of value the variable will hold.

The general syntax of declaring a variable is

```
datatype variable-list ;
```

- **datatype** refers to the valid type of the data. Datatypes can be int, float, char etc. We will discuss about datatypes in next section.
- **variable-list** refers to one or more identifier names separated by commas.

Example

```
int    i, j, sum;
```

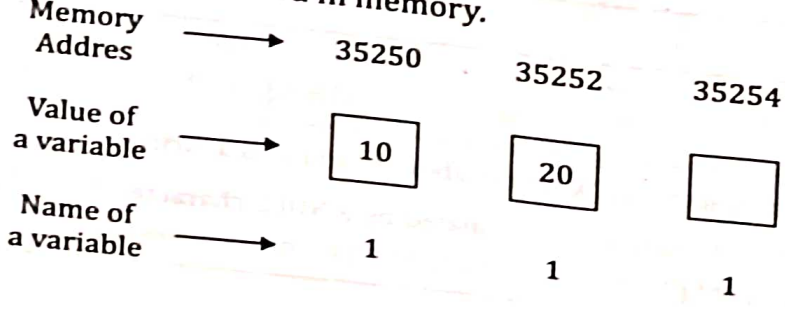
Data type — Variable list

- In the above example **i, j** and **sum** are called variables of type **int.**
- The variable declaration statement must end with a semicolon.
- C also permit us to initialize variable values along with the declaration as follows.

```
int  i = 10,  j = 20, sum;
```

Here *i* and *j* are initialized with values 10 and 20 respectively and sum is just declared but not initialized.

- All the variables declared are stored in memory.



The variables i, j and sum are stored in memory address 35250, 35252 and 35254 respectively.

- A variable name may be declared based on the meaning of the operation. Some meaningful variable names are as follows.

  Example: height, average, sum etc.

- Rules for naming variables are same as identifiers.
- Examples of valid variable names and invalid variable names are shown below:

Valid Variable Names	Invalid Variable Names
Sum	for (reserved word)
Name	123Add (must begin with letter)
_Add	accno%123 (% is not allowed)
average_score	average score (space is not allowed)

- Variables can be declared in either inside the functions or outside of all functions. The variables declared inside the functions are called **local variables** and the variables declared outside of all the functions are called **global variables.**

```
int a, b, c ;              → global variables
main()
{
      int i, j, sum ; → local variables
      ... ...  ..
      ...  ...  . .
}
```

Caution

The number of characters that we can have in a variable name will depend upon compiler. A minimum of 31 characters must be supported by a compiler that conforms to the C language standard, so we can always use names up to this length without any problems. It is suggested that variable names which are longer become cumbersome and make the code harder to follow. Some compilers will truncate names that are too long.

4.8 Data Types

The data types are used to inform the type of value that can be stored in a variable. Before using the variables, each variable must be declared to inform the type of data it can hold, in the beginning of a program. We have already discussed about declaring a variable. The general syntax is

```
datatype var1, var2 ... varn ;
```

Here var_1, var_2, ... var_n are the variable names. The data type informs the type of value that can be stored in variables var_1, var_2, ... var_n.

```
int  First,  Second,  Sum ;
```

Here, the declaration informs that the variables **First, Second** and **Sum** are the variables of the type **integer,** used to hold only integer value.

C has four basic data types. They are **int, char, float** and **double.**

4.8.1 int Data Type

All integer numbers are whole numbers without a decimal point or any other character. All these numbers can be preceded by either '+' or '-' sign. If a number is positive value, then it is not essential to write '+' sign before the number. If a number is negative number, then it is required to write '-' sign before the number.

Example

```
100      55      -250   9987
```

The number of bytes required to store an int variable is system dependent. In a 16 bit machine, int occupies 2 bytes. In a 32 bit machine, int occupies 4 bytes. In most cases an integer variables occupy 4 bytes of space in memory.

The range of int variable is from – 32768 to + 32767 for 16 bit machines.

The size of the integer variable for 32 bit machine = 4 bytes
= 32 bits
= $2^{32} - 1$
= 4,294,967,295

-2,147,483,648 (min) +2,147,483,647 (max)

Example of declaring integer variables:

```
int   sum, i,  j;
int   a = 10, b = 20, result ;
```

Here **sum, i, j, a, b,** and **result** are variables of type **integer.** It can hold the values within the range -2,147,483,648 to +2,147,483,647 .

```
int   a = 3547483647 ; is illegal
```

We can use the above declaration, but 'a' cannot hold the value specified.

4.8.2 char Data Type

This means that a variable is a character. A **char** is single ASCII character. Any symbol enclosed within two single quotes is a **character.** A **char** data type requires a single byte.

Example

'A' 'a' '5' '%' '\n' etc.

Size of a character variable = 1 byte
= 8 bits
= $2^8 - 1$
= 255

-128 (Min) +127 (Max)

The value that can be stored in the character variable should be between –128 to +127. The following declaration declares the character variables.

```
char ch, i, str ;
```

In the above declaration: **ch, i** and **str** are declared as character variables, used to store a single character. We can also assign some initial values to the variables at the time of declaration.

```
char ch = 'A', i = 'B', str = 'C' ;
```

4.8.3 float Data Type

The data type **float** is used to declare the variables to hold real numbers. These are the numbers with decimal points. In C language, a float variable occupies 4 bytes of memory space. The numbers shown below are examples of float data type

10.56 3.14 –1.06 10.0 .5678 etc.

The **float** variable takes a size of 4 bytes, with 6 digits of precision (i.e., 6 decimal places). The minimum and maximum value that can be stored in float variable must be between $3.4*10^{-38}$ to $3.4*10^{+38}$.

Declaring the float variable is as follows

```
float  a, b, c = 3.142 ;
```

The above declarations tells that a, b, c are variables of type float. The variable c is declared as well as initialized with a value.

4.8.4 double Data Type

The data type **double** is used to declare the variables to hold large real numbers. When higher precision numbers are required, instead of using **float, double** is used. The double variable occupies 8 bytes of memory space. The double data type variables takes a size of 8 bytes (64 bits) with 14 digits of precision. This is known as **double precision.**

The minimum and maximum value that can be stored in double variable must be in the range of $1.7*10^{-308}$ to $1.7*10^{+308}$. The below example shows the declaration of double variables.

```
double  d1, d2, d3 = 1.5e55 ;
```

The above declaration tells that **d1, d2** and **d3** are variables of type **double,** which can be used to store the real numbers. The d3 variable is initialized with the value 1.5e55.

2.8.5 Qualifiers / Modifiers

Qualifiers or **Modifiers** are keywords, used to alter the basic data types based on the size and sign. The qualifiers that can be applied to the basic data types are

- signed
- unsigned
- short
- long

signed and **unsigned** are used to specify whether the variables can store both positive and negative numbers or only the positive numbers.

All these modifiers alters the meaning of the basic data type to more precisely fit a specific need. The following general syntax shows the usage of modifiers.

```
<modifier>  <basic data type>  <var1>,<var2>,  …<varn> ;
```

Here, modifier precede the basic data type.

Example

```
short   int  a, b, c ;
long     int d ;
```

4.8.6 Qualifiers for the int Data Type

All the qualifiers, i.e., **signed, unsigned, short** and **long** can be applied to the integer variables. The declaration

```
int  a ;
```

declares the variable **a** of type **int,** which can hold the value ranging from -2,147,483,648 to +2,147,483,647. That means variable a can hold both positive and negative numbers.

(1) signed int

The qualifier signed is not necessary to prefix to **int** keyword. This is because the data type **int** can hold both +ve and – ve numbers. The declaration int a is same as signed int a.

(2) unsigned int

Suppose we want to store only positive numbers, then we use the modifier **unsigned**. The keyword **unsigned** is prefixed to the keyword int. The maximum and minimum value that can be stored in **unsigned int** variable should be in the range 0 to 4,294,967,295.

Example:

```
unsigned int a = 4000000000 ;
```

(3) short int

short int is same as **int,** requires 2 bytes of memory space. Some machines may have **short int** as half the size of **int**. The maximum and minimum value that can be stored in **short int** variable should be in the range – 32768 to + 32767.

```
short int a = 10 ;
```

(4) long int

In 16 bit machines, int is 2 bytes and long int is 4 bytes. But in 32 bit machines, both int and long int are same.

The range of long int is : -2,147,483,648 to +2,147,483,647
The range of unsigned long int is : 0 to 4,294,967,295.

We can also use **long long int** to store numbers larger than int or long int.

The range of long long int is : -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
The range of unsigned long long int is : 0 to 18,446,744,073,709,551,615.

```
int a = 10 ; or long int a=10
long long int b=9999999999999999;
```

4.8.7 Qualifiers for char Data Type

The qualifiers that can be prefixed to **char** data type are **signed** and **unsigned**. The qualifiers **long** and **short** may not be applied to **char** data type. We know that every character is associated with a unique ASCII value. Hence the character variable can be considered as integer variable. The character variable stores any positive or negative integer between the range –128 to +127. The **signed** qualifier need not be preceded with char variable because char variable itself can hold both positive and negative integers.

```
char ch ;        is same as      signed char ch ;
```

The unsigned qualifier allows the variable to store any character in the range 0 to 255.

```
unsigned char ch ;
```

4.8.8 Qualifiers for float and double Data Type

We know that a **float** variable occupies 4 bytes in memory and can range from $3.4*10^{-38}$ to $3.4*10^{+38}$. If it is not sufficient then C offers **double** data type which occupies 8 bytes in memory and can range from $1.7*10^{-308}$ to $1.7*10^{+308}$. If this is also not sufficient then C offers **long double.**

The **long double** variable occupies minimum10 bytes of memory and maximum and minimum value that can be stored in a variable must be in a range from $3.4*10^{-4932}$ to $3.4*10^{+4932}$. In few compilers, the long double variables occupies 12 or 16 bytes of memory. This is compiler or machine dependent.

Examples of declaring long double variables is shown below:

```
long  double d = 1.5e2520;
```

Note

No qualifiers can be applied on float data type.

Size and Range of data types on 32 bit system is shown below.

Data Type	Size (bytes)	Range
char	1	–128 to +127
signed char	1	–128 to +127
unsigned char	1	0 to 255
short int	2	–32,768 to +32,767
int	4	–2,147,483,648 to +2,147,483,647
unsigned int	4	0 to 4,294,967,295
long int	4	–2,147,483,648 to +2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long int	8	0 to 18,446,744,073,709,551,615
float	4	$3.4E^{-38}$ to $3.4E^{+38}$
double	8	$1.7E^{-308}$ to $1.7E^{+308}$
long double	10	$3.4E^{-4932}$ to $3.4E^{+4932}$

4.9 Declaration of Variables

Specifying the data type that can be stored in each variable is done using declaration statements. Declaration statements, in their simplest form, provide a data type and variable name. The general syntax of declaring a variable is

```
datatype variablename;
```

or

```
datatype  var1, var2, ----- varn;
```

Example

(1) Declare an Integer Variable

```
int sum.;
```

Tells the computer to reserve enough space for an integer number

Tells the computer to "Tag" the first byte of reserved storage with the name sum.

(2) Declare Floating point Variable

```
float avg:
```

Tells the computer to reserve enough space for a floating point number.

Tells the computer to "Tag" the first byte of reserved storage with the name **avg**.

(3) Declare Character Variable

```
char ch;
```

(4) Declare Double Precision Variable

```
double d;
```

(5) Declare Multiple Integer Variables in Single Declaration Statement

```
int a1, a2, a3, a4;
```

Note:

It is required that we declare variables to the C compiler before we use them. We do this by providing a list of variables near the beginning of the program. This way, the compiler knows by what names the variables are referred to and their type, i.e. what type of values they can contain.

4.10 Assigning Values to Varinbles

Once we have declared a variable, the next step is to assign a value to that variable. The process of assigning a value to a variable is called as initialization. We can assign a value to a variable in two ways.

(1) Using Assignment Operator

(2) Reading Data from Keyboard

4.10.1 Using Assignment Operator

Variables declared can be assigned (or) initialized using an assignment operator '='. The declaration and initialization can also be done in the same line.

Syntax

```
data_type variable_name = constant;
```

Example

```
int x = 10;
```

where x is an integer variable and 10 is an integer constant

Example

```
int x;
x = 10;
```

In first statement x is declared but not initialized. The second statement assigns value 10 to variable 'x'.

Example

```
int x, y, z;
x = y = z = 10;
```

one (or) more variables can be initialized with one value in a single statement.

Example

```
int x = y = z = 10;
```

This statement is also valid. Declaration of three variables and initializing with one value in single statement.

Example

```
int x=10;
char gender = 'M';
float avg = 0.0;
double factor = 0.21058e-3;
```

Here, x is an integer variable whose initial value is 10, gender is a character variable initialized with 'M', avg is a floating point variable with initial value 0.0, and factor is a double- precision variable whose initial value is 0.21058×10^{-3}

Program Write a Program to Illustrate Declaration and Assignments

```
# include <stdio.h>
void main( )
{
      / * ------ only Declaration ---------------------- * /
      int x , y ;
      float p, q;
      / * ------ Declaration and Assignments ---------------------- * /
      int z = 20;
      float r = 3.142;
      / * ------ only Assignments  ---------------------- * /
      x = y = 10;
      p = q = 1.0;
      / * ------ Display ---------------------- * /
     printf ("x = %d\n",x);
     printf ("y = %d\n",y);
     printf ("z = %d\n",z);
     printf ("p = %f\n",p);
     printf ("q = %f\n",q);
     printf ("r = %f\n",r);
}
```

4.10.2 Reading Data from Keyboard

One more way of assigning values to variables is to input data from keyboard using the scanf function. We will discuss more about scanf function in next chapters. The general format of scanf is as follows.

Syntax

```
scanf (" format string ", &var1, &var2, ------&varn);
```

Example

```
int x;
printf("Enter the Value of x");
scanf ("%d", &x);
```

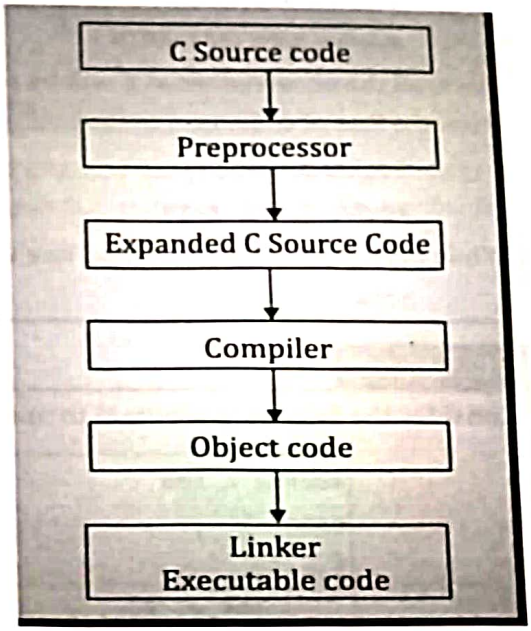
Here, x is an integer variable which is declared but not initialized. The printf function prints the string mentioned in the double quotes. When scanf function is executed, the execution stops and waits for the value of 'x' to be typed in. Once the value is entered through the keyboard and press Enter Key, then the value entered will be assigned to the variable x.

Note

The ampersand symbol & before variable name is an operator that specifies the variable name's address.

4.11 Defining Symbolic Constants

The C language has a unique feature called preprocessor directive, which is not available in many other high-level languages. As the name implies, a **preprocessor** is a program that processes the source code before it passes through the compiler. The preprocessor are used to help the programmer to make the source code easier, more efficient and portable. The C-preprocessor is a program which contains set of directives called preprocessor directives, which are translated into valid C codes by the compiler. Preprocessor operates under the control of preprocessor directives, which follows a special syntax rules different from the normal C syntax rules. In C, each preprocessor directive must start with #and without **semicolon.**



The preprocessor directives are divided into 3 categories. They are

1. Macro substitution directives
2. File inclusion directives
3. Conditional compilation directives.

A macro is essentially a defined name having a replacement text for macro expansion (or) macro substitution. The macro name is replaced with its corresponding text. A preprocessor directive called #define can be used to define macros in a C program.

4.11.1 Simple Macro Substitution

Simple macro substitution will replace all the occurrences of an identifier with the string from the position of definition to the end of a program. This can be achieved with the help of **#define** preprocessor directive.

"#define" is a preprocessor directive and is used to declare a symbolic constants in C program. A symbolic constant is a name given to a constant.

Example : `#define PI 3.14`

The above statement replaces every occurrence of PI (identifier) with 3.14 (value) at the time o preprocessing. The preprocessing is done before the compilation. When we compile our program first it preprocesses and then it compiles.

The general syntax of #define directive is as follows.

```
#define    <identifier>    <substitute-value>
                    (or)
#define    <identifier>(argument1, – argumentn)    <substitute-value>
```

Here identifier is also called as **symbolic name**

Example

```
#define X 100
#define NAME SRIKANTH
```

Here, all the occurrences of X will be replaced by 100 and NAME will be replaced by SRIKANTH, starting from the line of definition to the end of the program.

```
#define X 100
#define TOTAL X+10
```

While preprocessing, the second line will be changed as

```
TOTAL 100+10
```

Example

Consider the following program to understand the symbolic constants.

```
#define X 100
main( )
{
    int a, b, c ;
    a = 10;
    b = 20;
    c = X;
    a = a+X;
    b = X+b;
}
```

after preprocessing

```
#define X 100
main()
{
    int a, b, c ;
    a = 10;
    b = 20;
    c = 100;
    a = a+100;
    b = 100+b;
}
```

The rules to be followed while defining the symbolic constants are

- They are usually defined at the beginning of the program. However it may appear anywhere in the program but before it is referenced in the program.
- #define statements must not end with a semicolon.

  Example: #define PI 3.14; is illegal
- No blank space is allowed between # and define.

  Example: # define PI 3.14 is illegal.
- A blank space is required between #define and identifier.
- Symbolic names or identifiers have the same form as variable names. Traditionally symbolic names are written in upper-case letters to distinguish them from the normal variable names.
- Re-assigning of values to the symbolic names is not allowed.

  Example: #define PI 3.14

  PI = 4.28; is not allowed

4.11.2 Macros with Arguments

The more complicated form of replacement can be made with the help of macros with arguments. The general form is

```
#define identifier(arg1, arg2 .... argn) <substitute-value>
```

During preprocessing in a source program, the subsequent occurrence of macros with arguments is known as **macro call**. When the macro call is made, the preprocessor substitutes the string, replacing the formal arguments (arg_1, arg_2 ... arg_n) with actual arguments.

Example

Consider the **macro with arguments** definition as shown below.

```
#define sum(a, b) a+b
```

If the following statement is included in the program.

```
z = sum(x, y) ;
```

Then the preprocessor would expand the statement to

```
z = x + y ;
```

Example

Consider the following code

```
#define square(x)  x*x;
main( )
{
    int a = 5, b;
    b = square(a);
    printf("b = %d", b);
}
```

In the above code, b = square(a) would expand to

```
b = a*a ;
```

Therefore b will contain the value 5*5 = 25. The b value will be printed in the last printf( ) statement.

Example

Consider the following code

```
#define square(x) x*x
main ( )
{
    int a ;
    a = 16/square(2) ;
    printf(''a = %d", a) ;
}
```

The above code prints a = 16. This is because the macro was expanded to a = 16/2*2 which results in a = 8*2 = 16.

The correct code will be as shown below

```
#define square(x) (x*x)
main ( )00
{
    int a ;
    a = 16/square(2) ;
    printf("a = %d", a) ;
}
```

Now, the macro was expanded to

```
a = 16/ (2*2)
```

which results in

```
a = 16/4
a = 4
```

Example

Consider the following code

```
#define mul(a,b) a*b
main ()
{
    int x = 5, y = 2, result;
    result = mul(x + y, x + y);
    printf("result = %d", result);
}
```

The above code would expand to

```
result = x + y * x + y
```

which results inresult = 5 + 2 * 5 + 2

= 5 + 10 + 2 = 17

This result is not the expected result. Since the preprocessor does the blind text substitution, the parameter x + y is placed in the place of a and b respectively.

The above definition can be made correctly to get the expected result as

```
#define mul(a,b) ((a)*(b))
```

This would result a proper expansion as

```
result = ((x + y)*(x + y)) ;
```

Example

Some more macros with arguments are

```
#define AREA(a) ((a)*(a))
#define LARGEST(a,b) ((a>b) ? (a) : (b))
```

Example

We can also nest one macro with in another macro. That is, one-macro definition can be substituted in another macro definition. Consider the following examples.

```
#define mul(a,b), ((a)*(b))
#define multiply(a, b, c) mul(mul(a, b), (c))
#define large2(a, b) ((a>b) ? (a) : (b))
#define large3(a, b, c) large2(large2(a, b), (c))
```

If the following statement is placed in the program

```
big = large3(x, y, z) ;
```

Then the preprocessor would expand this to

```
big = large2(large2(x, y), (z)) ;
```

4.12 Defining Variable as a Constant

Using the constant type qualifier, we can tell the compiler that the variable value cannot be changed during the execution of the program. Once we declare a variable by using constant type qualifier, we cannot change its value later in the program. Any attempt to change its value would result in an error.

Example

```
int const x=10;
```

Here, x is declared as constant variable and hence its value cannot be changed later in the program.

Example

```
int const x=10
.........
.........
.   .   .   .   .   .   .   .   .   .
x= 20;    // wrong Error
```

Since x is constant variable with initial value as 10, we cannot assign 20 to it again.

Example

```
int const x:
x=10; // Error
```

We need to remember that, a constant variable can be initialized only at the time of declaration.

Note:

- We can declare constant pointers.
- We can use constant in function parameter
- Constant can be applied to other data type variables also