(3)

VARIABLES AND CONSTANTS

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.

Rules for Constructing Variable Names

1. The first letter in a variable must be an alphabet.
2. No comments or blanks are allowed within a variable name.
3. No special symbol other than an underscore can be used in a variable name.
4. A variable name is any combination of alphabets, digits or underscore.

Variable Declaration

All variables must be declared before we use them in C program, although certain declarations can be made implicitly by content. A declaration specifies a type, and contains a list of one or more variables of that type as follows:

type variable_list;

type must be a valid C data type including char, int, float, double, or any user defined data type etc., and **variable_list** may consist of one or more identifier names separated by commas.

Examples of declaration

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

You can initialize a variable at the time of declaration as follows:

int *i = 100;*

An **extern** declaration is not a definition and does not allocate storage. In effect, it claims that a definition of the variable exists some where else in the program. A variable can be declared

multiple times in a program, but it must be defined only once. Following is the declaration of a variable with **extern** keyword:

```
extern int   i;
```

Though you can declare a variable multiple times in C program but it can be declared only once in a file, a function or a block of code.

Variable Initialization in C

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

```
variable_name = value;
```

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Examples

```
int d = 3, f = 5;          /* initializing d and f. */
byte z = 22;               /* initializes z. */
double pi = 3.14159;       /* declares an approximation of pi. */
char x = 'x';              /* the variable x has the value 'x'. */
```

It is a good programming practice to initialize variables properly otherwise, sometime program would produce unexpected result.

Example of declaration in a program

```
#include <stdio.h>

int main ()
   {
      /* variable declaration: */
      int a, b;
      int c;
      float f;

      /* actual initialization */
```

```
   a = 10;
   b = 20;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);

   return 0;
}
```

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

value of c : 30
value of f : 23.333334

VARIABLE SCOPE

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

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

Local Variables

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

Example

The following is an example using local variables. Here all the variables a, b and c are local to main() function.

```
#include <stdio.h>

int main ()
```

```
{
    /* local variable declaration */
    int a, b;
    int c;

    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;

    printf("value of a = %d, b = %d and c = %d\n", a, b, c);

    return 0;
}
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

Example

The following is an example of using global and local variables:

```
#include <stdio.h>

/* global variable declaration */
int g;

int main ()
{
 /* local variable declaration */
 int a, b;

 /* actual initialization */
 a = 10;
 b = 20;
 g = a + b;
```

```
 printf("value of a = %d, b = %d and g = %d\n", a, b, g);

 return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference.

Example

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main ()
  {
    /* local variable declaration */
    int g = 10;

    printf ("value of g = %d\n",  g);

    return 0;
  }
```

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

```
value of g = 10
```

Formal Parameters

A function parameters, formal parameters, are treated as local variables with-in that function and they will take preference over the global variables.

Example

```
#include <stdio.h>

/* global variable declaration */
int a = 20;
```

```
int main ()
  {
   /* local variable declaration in main function */
      int a = 10;
      int b = 20;
      int c = 0;

      printf ("value of a in main() = %d\n",  a);
      c = sum( a, b);
      printf("value of c in main() = %d\n",  c);

      return 0;
  }

 /* function to add two integers */
int sum(int a, int b)
  {
   printf("value of a in sum() = %d\n",  a);
   printf("value of b in sum() = %d\n",  b);

   return a + b;
  }
```

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

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

CONSTANT

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well. The **constants** are treated just like regular variables except that their values cannot be modified after their definition.

Integer literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal. An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various type of Integer literals:

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
```

Floating-point literals

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

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

Here are some examples of floating-point literals:

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character constants

Character literals are enclosed in single quotes e.g., 'x' and can be stored in a simple variable of **char** type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

String literals

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters. You can break a long lines into multiple lines using string literals and separating them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

*"hello, *

dear"

"hello, " "d" "ear"

DEFINING CONSTANTS

There are two simple ways in C to define constants:

1. Using **#define** preprocessor.
2. Using **const** keyword.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant:

#define identifier value

Example

```
#include <stdio.h>
#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'
```

```
int main()
 {

  int area;

  area = LENGTH * WIDTH;
  printf("value of area : %d", area);
  printf("%c", NEWLINE);

  return 0;
 }
```

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

value of area : 50

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Example

```
#include <stdio.h>

int main()
 {
  const int  LENGTH = 10;
  const int  WIDTH  = 5;
  const char NEWLINE = '\n';
  int area;

  area = LENGTH * WIDTH;
  printf("value of area : %d", area);
  printf("%c", NEWLINE);

  return 0;
}
```

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

value of area : 50

Note that it is a good programming practice to define constants in CAPITALS.

OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C language is rich in built-in operators and provides following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators

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

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

Relational Operators

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

Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then	(A == B) is not true.

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

Logical Operators

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

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

Bitwise Operators

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

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one	(A ^ B) will give 49 which is 0011

	operand but not both.	0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by C language:

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

Misc Operators ↦ sizeof & ternary

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

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

OPERATORS PRECEDENCE

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedence than + so it first get multiplied with 3*2 and then adds into 7.

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

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

Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right