Contents

Data type, constant, and variables

C has different types of variables which is stored in memory during compilation. These variables or identifiers or operands help to develop program based on algorithm. C has following types of data types to work with a program.

Types of data type:

1. Primary or Fundamental Data types
   Example :) Integer, float, double, character, and character string.
2. Derived Data types
   Example:) Array, Function, and Pointer
3. User defined data types
   Example:) Structure, Union, and Enumerated.

Rules of declaring variable or identifier:

1. The variable must start with letter or underscore (ex: int _x, char fruit)
2. The variable can have numbers (ex: int res2)
3. The variable cannot have key word as identifier (ex: float do) because do is key word.
4. The variable name must be unique in their location local or global. That is variable cannot have same name in the local declaration part or in the global declaration part of any data type.
5. Give meaningful name to the variable for easy identification (ex: int gi_total), here prefix g stands for global, and i for integer, and the name.
6. The variable cannot have special characters like %, $, # etc. (ex: int rf_$s is invalid)
7. No space between letters (ex: int ap ple; is invalid declaration)
8. Length of the variable name 31 characters.

Declaring Primary data type variables:

Example: int a; float b, double c, char d, and char s[5];

User defined data type declaration:

typedef int marks; // typedef is keyword marks is variable of type integer
marks, phy, chem, bio; //new integer variables can be declared by referring marks

Assigning value to variables:

```
Example: int a; char d;
         a = 10;
         d = 'A'
```

Initializing value to variables:

```
Example: float b = 123.456;
          char s[6] = "apple";
```

It is good practice to initialize value for integer (0), character (' '), and character string (" ")data type, because default values for these data type are garbage. The garbage value may create logical error in the program.

TOP

Enumeration Data type:

It is a user-defined data type. There two types

1. typedef
2. enum

Type declaration:

Syntax is typedef data_type variable_name. Example:

typedef int marks;
marks phy,chem,mat,tot[3];

From the above example the variables work just like primary data type. The variables phy, chem, mat, tot[3] are of data type integer because marks is defined as integer data type in type declaration.

Syntax for enumeration data type is eum variable_name {value1,…….valuen}; Example:

enum day {Monday, Tuesday,……Sunday}; In this example default value for Monday will be assigned 0 and increased by one to other values. To change the default the syntax is:

enum day {Monday=1, Tuesday,……Sunday};

Declaring local and global variables or storage class:

A variable can be local or global variables depending on the place of declaration. Consider the following example

```
int y = 30, x=10; // global variables
void main( )
{
     {int y =20; // local variable
       printf (“%d”, y); // out put will be 20
      }
      printf (“%d”, x); // out put will be 10
      printf (“%d”, y); // out put will be 30
}
```

Local variable within the braces will work only within that braces. That variable cannot be used outside the braces. To use a variable in the entire program the variable must be a global variable. Global variables are declared above the void main( ) part of the program.

Types of Constants:

1. Numeric constants
2. Character constants
3. Backslash constants
4. Symbolic constants
5. Volatile constants

When a variable is declared as a constant it cannot be changed anywhere in the program. When a variable is declared as a constant the value must be assigned at the place of declaration.

TOP

Declaring constants:

Numeric constants and character constants are declared as follows.

const int x =10;
const char = ’x’;

Backslash constants are declared in printf function for better control over the output.

printf(“This is first line \n and this is second line”);

Example: \n for next line
\t for tab control
\a beep sound

Symbolic constants is a preprocessor it is declared as global constant variable.

#define PI 3.14

It improves the readability of the program

Volatile constants are used; to change the value of the variable from outside the program.

volatile time t; The value of t can be changed by external program only.

volatile constant int x =10; to modify x by local and external program

TOP

Operators, Precedence and Associativity

The C operators help to write program to solve a given problem. The operators and the built-in C functions help the programmer to find efficient way to solve the given problem. The operators are of following types.

Types of C operators

1. Arithmetic operators
   It is used to work with two or more variables and constants
   Example: Add +, Subtract -, Multiply *, divide /, Modulus or remainder %
2. Relational operators
   It is used to compare two variables
   Example: x >10; x == 10; x>=100; y !=10;
3. Logical operators
   It is used to compare two relational operations
   Example: (x >10 && y > 100)
   (x >10 || y >100)
4. Assigning Operator
   It is used to assign a value to a variable
   Example: x = 10;
5. Conditional Operator
   It is used to check the condition of a relational operation
   Example: y = (x==‘p’)?40:30;
   Here if x has a value p then y will be assigned 40 otherwise it will be assigned 30.
   The same statement can be written in if-else statement.

```
if (x==‘p’)
  { y=40;
   }
else
 { y=30;
 }
```

6. Increment or Decrement operator
   It is used to increase or decrease the value by one. Assume x =1;
   Example: ++x; // now x will be 2
   x++; // now x will be 3
   --x; // now x will be 2
   x--; // now x will be 1

   There are two types prefix and postfix when assigning the increment variable to another variable. Assume x =1;
   Example: y = ++x; // now x and y is 2
   y = x++ // now x is 2 and y is 1.
7. Shorthand operator
   It is used to reduce the statement length.
   Example: y = y +1, x = x*y can be written as y += 1 and x *= y.

TOP

8. Bitwise operators
   It is used to work with binary number
   Example: AND &, OR |, and Exclusive OR ^, left shift <<, right shift >>
9. Special operators
   It is used for completing a meaningful syntax
   Example: int x,y;
   Comma, pointer, member selection operator, size of operator.

Precedence and Associativity:

Precedence is a priority given to the operators. Each operators has different priorities. In an expression priority is given based on the operators used. Therefore care must be taken to avoid logical error when using complex expression with operators.

Example: x = (s+10) * (r+t)
The processing will be as follows
Step1: s+10 will be processed
Step2: r+t will be processed
Step3: results of step1 and step2 will be multiplied
Step4: the final result value will be assigned to x

From the above example it is clear that the following operators (), *, +, and = are used. The first operation was (s+10), the second operation was (r+t), and the third operation was * and the last operation was =. Therefore for the above example the highest priority was given to expression within ( ) and last priority was given to = (assigning operator). This process of flow of operation among the operators is called precedence.

The direction at which the operators are processed is called associativity. The above example showed that ( ) moved from left to right, and operator equal (=) from right to left.

For more detail see page 70

TOP

Managing Input and Output

Most programs are executed by reading the data from the input through keyboard and displaying the result on the monitor (output) after going through some process written in the program. The processes of input and output can be controlled or managed with the help of C functions.

Managing Input

The input data are managed with the help of following C functions.

1. getchar( )
2. gets( )
3. scanf( )

Managing Output

The output data are managed with the help of following C functions.

1. putchar( )
2. puts( )
3. printf( )

The getchar( ) and putchar( ) are used only for character data type.

The gets( ) and puts( ) are used only for string or character string data type.

The scanf( ) and printf( ) function can be used be for any primary data types in mixed or any form depending on the type of input or output data.

Commonly used scanf and printf format codes:

%c read or print single character
%d read or print decimal integer
%e read or print float
%f read or print float
%g read or print float
%i read or print signed decimal, octal, hexadecimal, integer
%o read or print octal integer
%s read or print string
%u read or print unsigned decimal integer
%x read or print hexadecimal integer
%[..] read a string of words

Other prefix for format codes:

h for short integer
l for long integers or double
L for long double

Managing input and output of character data type:

The function getchar( ) or scanf( ) can be used to read the character from the keyboard. The character data type will read only one character from the keyboard. The character can be alphabet or number or any other valid character. Since character data type can handle only one character. The data

TOP

cannot be controlled. Therefore it reads the first character found by these functions.

Example:

```
char t;
t = getchar( ); // is same as scanf("%c", &t);
```

Both getchar( ) and scanf read only one character from the keyboard as input value.

Similarly the function putchar( ) or printf ( ) can be used to display the output result in the monitor.

Example:

```
char t;
t = getchar( ); // is same as scanf("%c", &t);
putchar(t);
printf("%c", t);
```

Since t is of type character it will display only one character from putchar( ) and printf ( ).

Managing input and output of integer data type:

The syntax is %wd; where w is width of the field and d is format of decimal integer.

Input:

Example 1: scanf("%3d", &x);

Assume user enter 24578 in the keyboard. Here the scanf will read only the first three digits that is it will read 245. Therefore x will be assigned 245. It is clear from this statement that the %3d can read maximum of 3 digits.

Example 2: scanf("%3d %2d", &x, &y);

Assume user enter 245789. Here x will be 245 and y will be 78.

Example 3: scanf("%d-%3d", &x,&y)

Assume user has typed 123-64254; now x will be 123 and y will be 642. In this method the width of the first value must equal to the width of the value entered.

Example 4: scanf("%3d-%3d", &x, &y);

Assume user has type 2341-456. In this example the value of x will be 234 and y will try to read the next 3 digits starting from 1 since it finds - as second digit there will be a run time error. Therefore to avoid this error the first variable must be read in full like defined in example 3.

Output:

When managing output for numeric data types like integer, float, double, etc. The output value cannot be reduced or cut by specifying the width. If the width specified is less than the width of the value the compiler will read the entire value, because to maintain the data accuracy or not loss any data.

Example: printf("%3d", y);

Assume y = 23456; here the output will be 23456 not 234 because cut the number may affect the data accuracy. Therefore specifying width in the printf will help for better presentation of output value. There are other methods to generate output. Consider y = 9876. Refer Page 95

TOP

1. printf(“%d”, y); - width is equal to value of width (4) output is right justify. 9876
2. printf(“%2d”, y); - width is equal to value of width (4) output is right justify. 9876
3. printf(“%6d”, y); - width is 6 output is right justify. 9876
4. printf(“%06d”, y); - width is 6 output is right justify with leading zero. 009876
5. printf(“%-6d”, y); - width is 8 output is left justify.9876

Note for left justify type minus sign (-) before the field width as shown in example 5.

Managing input and output of float or double data type:

This is minimum field specification for float or double w >= p +7. Where w is width and p is the precision. The default precision (p) is 6. Therefore as the precision increases the width increases.

Input:

Example 1:   scanf(“%f”, &x);

Assume user enter 245.4566 in the keyboard. Data type of float or double the compiler will read the entire input value. Therefore for float or double the width cannot be specified for input.

Output:

Consider the example for 98.7654. Refer page 97

1. printf(“%f”, y); - width is equal to value of width (7) output is right justify.
2. printf(“%7.4f”, y); - width is (7) precision (4) output is right justify.
3. printf(“%7.2f”, y); - width is (7) precision (2) output is right justify 98.77.
4. printf(“%-7.2f”, y); - width is (7) precision (2) output is left justify.
5. printf(“%10.2e”, y); - width is (10) precision (2) output is right justify. 9.88e+01
6. printf(“%-10.2e”, y); - width is (10) precision (2) output is left justify. 9.88e+01
7. printf(“%e”, y); - width is (12) precision (6) output is right justify. 9.88e+01

Note for left justify type minus sign (-) before the field width as shown in example 5.

The width and precision can also be managed using the below method

printf(“%*.*f”, -7,4,y); is same as method 4 in the output.

Managing input and output of character string or string:

Character string value or data can be controlled in both input and output. Therefore required input and output data can be managed.

Input:

The syntax for input is %ws or %wc. Consider the following example.

Example 1: scanf(“%10s, t);

Assume user typed “Einstein is Great“. The variable t will be assigned only “Einstein” because as soon as the compiler read the space it stops assigning to t or it may be assigned another variable. Consider the following example.

Example 2: scanf(“%10s%5s”, t,v).

Assume user typed “Einstein is Great“. The variable t will be assigned only “Einstein” and variable v will be assigned “is”.

Therefore to read the entire used input including the space, the syntax is as follows.

Example 3: scanf(“%[^\n]”, t);

TOP

Assume user typed “Einstein is Great”. The variable t will be assigned “Einstein is Great”. In this example the compiler will read all but the next line character.

Example 4: scanf(“%[A-Z a-z 0-9]”, t)

In this example the compiler will read only alphabets and numbers.

Output:

The syntax for output is %w.ps. Where w is width and p is data to be read. Consider the example for “NEW DELHI 110001“. There are 16 characters including blank space. Refer page 99

1. printf(“%s”, y); - width is equal to value of width (16) output is right justify.
2. printf(“%20s”, y); - width is (20) output is right justify.
3. printf(“%20.10s”, y); - width is (20) reads first 10 character output is right justify . NEW DELHI
4. printf(“%.5s”, y); - width is (5) reads first 5 characters right justify. NEW D
5. printf(“%-20.10s”, y); - width is (20) reads first 10 character output is left justify. NEW DELHI
7. printf(“%5s”, y); - width is (16) reads all characters because no precision given left justify.

Managing Mixed data for input and output:

Multiple primary data type can be managed using scanf( ) and printf() functions for input and output respectively.

Input:

Consider the following example.

scanf(“%3d %c %f %5s”, &a, &b, &c, d);

In this example the variable a must be integer data type, variable b must be character data type, variable x must be float data type, and variable d must character string data type. In the scanf( ) character string does not require & sign before the variable name d. Therefore it is clear that format code order must match data type order or vice versa.

Output:

Consider the following example.

printf(“Integer %6d \n character %c, \n float %9.2f \n, string %5.2”,a,b,c,d)

In the printf( ) function format code must match the data type just like in scnaf( ). The printf( ) can also have following features in the control string

1. Text to be displayed
2. Backslash constants
3. Format code

In the above example format string is data enclosed within quotes, which is “Integer %6d \n character %c, \n float %9.2f \n, string %5.2”. Thus in this format string it has text, backslash constant, and format code. The format string can be anyone for those or in any combination.

Therefore the data can be managed to generate better output for presentation. It can be seen that only character string value can be managed at both input and output. Numeric output reads the entire data to avoid losing data at the same time precision for float can be managed in the output. Similarly only required value can be read for integer in the input only, whereas in the output for the integer the data is not changed it is only arranged in the required form.

TOP

TOP

Decision Making, Branching and Looping

A program can be controlled based on its algorithm by using control structures. C has following control structures for decision making. They are if-else, switch, while, do-while, and for control structures. While, do-while, and for; are the process known as iteration that is the process will be repeated until the condition is true.

If statement:

Consider the example 1:

```
if (x < 10)
  { printf("Value of x is less than 10"); //Statement-1 (True Block)
  }
```

In the above statement if the value of x is less than 10. It will execute the True block.

Consider the example 2:

```
if (x < 10)
  { printf("Value of x is less than 10"); // Statement-1 (True Block)
  }
else
  {printf("Value of x is greater than 9."); Statement-2 (False Block)
   }
```

In the above statement when the value of x is greater than 9. It will print the Statement-2. If value of x is less than 10; it will print Statement-1.

Consider the example 3:

```
if (x < 10)
  { printf("Value of x is less than 10"); // Statement-1 (True Block)
  }
else
  {printf("Value of x is greater than 9."); Statement-2 (False Block)
   }
  printf("End of the program."); //Statement-3 (Independent statement)
```

In the above statement when the value of x is greater than 9. It will print the Statement-2 and Statement-3. If value of x is less than 10; it will print Statement-1 and Statement-3.

In the example1 there is only true block therefore the only true block statements will be executed. In example2 and example3 there is true and false block, therefore at least one block will be executed either true or false block. The satement-3 will be executed at all time during execution because it not within the true or false block. A error free block is achieved only by using the braces { } correctly.

Consider the example 4:

```
if (x < 10)
  { printf("Value of x is less than 10"); // Statement-1 (True Block)
  }
```

TOP

```
        else
          printf(“Value of x is greater than 9.”); Statement-2 (False Block)
          printf(“End of the program.”); //Statement-3 (Independent statement)
```

The example4 work exactly as example3 because no { } was given to the false block, therefore the C compiler assume the first statement that is statement-2; as the false block and the following statement-3 as the independent statement.

Consider the example 5:

```
        if (x < 10)
          { printf(“Value of x is less than 10“); // Statement-1 (True Block)
          }
        else
          {printf(“Value of x is greater than 9.”); Statement-2 (False Block)
            printf(“End of the program.”); //Statement-3
          }
```

The example5 work exactly as example2 except the False Block has more than one statements. That is when the condition is false, the compiler will executed both the statements in the false block because they are written within { }. Therefore true and false block can have any number of statements within the braces { }.

Consider the example 6:

```
        if (x < 10);
          { printf(“Value of x is less than 10“); // Statement-1 (Independent Statement)
          }
```

Example6 may look like example1 but they are not, because immediately after the condition (x<10) there is semicolon(;) or statement terminator. This denotes the end of the statement block. Therefore from the next line of program or statement will be independent statements. So in the above example for any value of x the Statement-1 will be executed.

Consider the example 7:

```
        if (x < 10)
          { printf(“Value of x is less than 10“); // Statement-1 (True Block-1)
          }
        else if (x <100)
          {printf(“Value of x is less than 100.”); Statement-2 (True Block-2)
           }
        else
          {printf(“Value of x is greater than99.”); //Statement-3 (False Block)
          }
```

When a program is required to check for more than one condition an else if condition block can be included in the control structure. This is like having more than one true block, but only one true block will be executed. In the example7 if the value of x is less than 10 only True Block-1 will be executed. If the value of x is less than 100 only True Block-2 will be executed. If the value of x is

TOP

greater than 99 false block will be executed. Therefore it is clear that only one block will be executed it can be anyone of the True Blocks or false block. Therefore there can be more than one True block in the control structure, but there can be only one false block. Similarly there can be only True Blocks in the control structure without a false block. Consider the following example

Consider the example 8:

```
if (x < 10)
  { printf("Value of x is less than 10"); // Statement-1 (True Block-1)
  }
else if (x <100)
  {printf("Value of x is less than 100."); Statement-2 (True Block-2)
   }
else if (x < 1000)
  {printf("Value of x is greater than99."); //Statement-3 (True Block-3)
  }
```

This control structures has only True Block and these are valid statements.

Consider the example 9:

```
if (x < 10)
  { printf("Value of x is less than 10"); // Statement-1 (True Block-1)
  }
if (x <100)
  {printf("Value of x is less than 100."); Statement-2 (True Block-2)
   }
if (x < 1000)
  {printf("Value of x is greater than99."); //Statement-3 (True Block-3)
  }
```

Example8 can be rewritten in the form of example9, thus the example8 and example9 is exactly the same.

Nesting:

Nesting is used when a program requires to satisfy two or more additional conditions. Care must be taken when using nesting statements because complex nesting some time may result in logical error when braces { } are not assigned correctly. A nesting can have if, while, do-while, next, and switch in any form or combination.

Consider the example10:

```
if (x <10 && y<100)
   {printf("x is less than 10 and y is less than 100.");
   }
```

The example10 can be written in nested form as shown exampl11.

Consider the example11:

```
if (x <10)
  {if (y<100)
```

```
                {printf(“x is less than 10 and y is less than 100.”);  --True Block 1
                }
            }
```

An if within an if is called nesting. The compiler first checks the condition (x<10), if that condition is true then it checks the condition (y<100), if y is less than 100 then True Block-1 is executed. Therefore for the example11 it is clear that for the if condition (x<10) true block is;

```
                {if (y<100)
                    {printf(“x is less than 10 and y is less than 100.”);  --True Block 1
                    }
                }
```

Similarly for (y<100) true block is

```
        {printf(“x is less than 10 and y is less than 100.”);  --True Block 1
         }
```

Switch Control Structures:

The switch control can be used just like if control, but the switch value must be a value of type integer or character only. It can be used only for simple operation. Consider the following examples.

```
switch (x) -- x is integer data type
{ case 1:
          {printf(“The value of x is 1”);
           break;
           }
  case 2:
          {printf(“The value of x is 2”);
           break;
           }
   default:
          {printf(“The value of x is greater than 2”);
           break;
           }
```

In above example if the value of x is 1 then this block will be executed

```
          {printf(“The value of x is 1”);
           break;
           }
```

In above example if the value of x is 2 then this block will be executed

```
          {printf(“The value of x is 2”);
           break;
           }
```

In above example if the value of x is not 1 orr 2 then this block will be executed

```
          {printf(“The value of x is greater than 2”);
           break;
           }
```

TOP

Similarly for the following example

```
switch (ch) -- ch is character data type
{ case ‘a‘:
        {printf(“The value of ch is a”);
         break;
        }
  Case ‘b‘:
        {printf(“The value of ch is b”);
         break;
        }
   default:
        {printf(“The value of ch is not a and b”);
         break;
        }
```

The above switch statements can be rewritten using “if” control structure. Default part of switch statement works just like else statement.

```
if (x==1)
        {printf(“The value of x is 1”);
         break;
        }
else if (x==2)
        {printf(“The value of x is 2”);
         break;
        }
else
        {printf(“The value of x is greater than 2”);
         break;
        }
```

Consider the following if control, but this control cannot be written in switch control form.

```
if (x < 10)
  { printf(“Value of x is less than 10“); // Statement-1 (True Block-1)
  }
else if (x <100)
  {printf(“Value of x is less than 100.”); Statement-2 (True Block-2)
   }
else
  {printf(“Value of x is greater than99.”); //Statement-3 (False Block)
  }
```

Rules of switch:

1. Case value must be unique that is it same case value cannot be repeated in the switch
2. Default is optional and it can be placed any where in switch but it is preferred last.
3. Break is optional

TOP

4. Case value must be of type integer or character.
5. Case value is value derived from the switch expression.

The “if” and “switch” controls execute a process only once when the condition is true or false. When same true block to be repeated again and again until the condition is turned to false through certain process a new control structure called while, do-while, or next is used.

While Control Structure:

Consider the following example:

```
if (x != ‘a‘)
  { printf(“Value of x is not a“); //Statement-1 (True Block)
  }
```

In the above example the true block is executed only once when x is not ‘a‘. To repeat the process again and again the above example can be rewritten as shown below

```
while (x != ‘a‘)
  {x = getchar( );
   printf(“Value of x is not a“); //Statement-1 (True Block)
  }
```

Like if and there is no false block in while or do-while control structure. In above example the true block is

```
  {x = getchar( );
   printf(“Value of x is not a“); //Statement-1 (True Block)
  }
```

This block will be executed repeatedly when x is not equal to ‘a’ because the condition x!=‘a’. To finish the process the value of x must be equal to ‘a’

Do-while control structure:

The only difference between while and do-while is, the while will first check the condition and then if the condition is true it will execute the true block. Where as in the do-while statement the true block is executed first and then it checks the condition, only if the condition is true the process is repeated again. Therefore in the do-while control-structure the true block is executed at lease once.

Consider the following example.

```
x = ‘a’;
do
{ x = getchar( );
  printf(“Value of x is not a”);
} while (x!=‘a’)
```

In this example the true block will executed even if the value x is set as ‘a‘, because as said earlier in do-while control structure the true block executed first and then checks the condition.

The while and do-while controlled loop are called sentinel loop because number of iteration is unknown. The process can be repeated forever as long as x is not equal to ‘a’. If the number of iteration

TOP

is known "for" control structure is preferred. All the "for" control structure can be written in while control structure form whereas not all while control structure can be written in for control structure.

For control structure:

This control structure is preferred when the number of iteration is known. Consider the following example.

```
for (x=1; x<=10; x++)
  { printf"(The value of x is %d", x);
  }
```

x=1 is initial value of x, x<=10 is the condition for iteration, and x++ is the increment of x. The above example is fixed looping the process is repeated for 10 times, this is also static iteration. The iteration can be made dynamic by declaring variables for the initial value and the condition. Consider the same is variable form

```
scanf("%d %d", &x,&y);
for (x; x<=y; x++)
  { printf"(The value of x is %d", x);
  }
```

In above example the process is repeated y minus x times. Assume x=5 and y=10, here initial value of x is 5,starting condition is 5<=10 (x<=20), the increment is by one x++. The value of x can be increased in any form of expression. If value x to be increased by two the same program must be as follows.

```
scanf("%d %d", &x,&y);
for (x; x<=y; x=x+2)
  { printf"(The value of x is %d", x);
  }
```

The "for" control structure can be written using while statement as shown below

```
for (x=1; x<=10; x++)
  { printf"(The value of x is %d", x);
  }

x =1;
while (x<=10)
    {printf"(The value of x is %d", x);
     ++x;
    }
```

Consider the following example

```
while (x != 'a')
   {x = getchar( );
    printf("Value of x is not a"); //Statement-1 (True Block)
```

TOP

```
            }
```

The above program cannot be written in “for” control structure form.

A programmer may prefer while or for depending on the style of the program because of which the end result will not be affected. Therefore it is clear that to check the condition once “if” or “switch” may be used; to repeat it again and again until condition fail “while” or “for” is preferred. When working with array “while” or “for” loop will improve the program efficiency significantly.

TOP

Arrays

Array

In the early chapters the values are stored in a variable of primary data type. The disadvantage is only one value can be stored in a variable at a time. Therefore to store more than one value, then more than one variable have to be declared. An additional variable can be declared to store or read two or three values. Imagine storing the register numbers of students of a college in several variables. If there are thousand students in a college then the program requires 1000 different variable names to store the register number of the students. Then the program becomes unreadable and complex. To simplify that array can be used by just declaring one variable of type array. Consider the following example of data type int reg[1000], this means the variable name “reg” can store up to 1000 values. The values can be read, element by element. Therefore to store a large set of data of same type array is the choice. Array is a derived data type. It is very effective when working with large number of data of same data type or different data types. Array reduces the programming size or coding.

Consider the following example:

```
int x, y, z;
x=10; y=20; z=30;
```

The above statements can be rewritten using array form

```
int x[3];
x[0]=10; x[1]=20; x[2]=30;
```

When the number of variable is less the statement may not look confusing or difficult. Now consider the following example.

```
int y[60];
```

The array y can store up to 60 values in its elements. The structure and location value in the array is similar to matrix. Now it is clear that it is not good practice to declare 60 variables for array y.

Arrays are classified a follows.

1. One dimensional array
2. Two dimensional array
3. Multidimensional array
4. Dynamic array

One Dimensional Array:

It is similar to matrix with only one column but multiple rows. This type of array is useful when working with only one set of data at a time.

Declaring One Dimensional Array:

It is similar to declaring any other primary data type except, the number element must specified for an array in bracket [ ].

```
int z[5]; float s[10]; double w[6];
char t[4];
```

The arrays declared as shown. The numbers inside the bracket is the number of elements for that variable. Therefore z can store 5 values, s 10 values, w 6 values, and t 4 values. The only difference for

TOP

character array t; is that the last element in the array must be allotted for NULL value, therefore remaining 3 elements of t can be any character including NULL.

Assigning One Dimensional Array:

```
int z[5]
z[0]=10;
z[1]=20;
z[2]=30;
z[3]=40;
z[4]=50;
```

The values are assigned element by element in the array.

Note: The array element start with 0 NOT 1

```
float s[10];
z[2]=11.11;
z[5]=22.22
z[8]=88.88;
```

In the above the element 2, 5, and 8 are assigned values but the other elements are not assigned any values. In such case the remaining element will be assigned zero. Therefore for numeric array when the value is assigned to fewer elements the remaining elements will be assigned zero by the compiler.

```
char t[4];
t[0]='a';
t[1]='b';
t[2]='c';
t[3]='d';
```

The last element t[3] is assigned ‘d’, as we discussed earlier the last element for character array must be provided for NULL value. Here‘d’ is assigned to the last element the compiler will not give error during compilation but there will be error in the program during run time.

Now consider this example

```
char t[4];
t[0]='a';
t[1]='b';
```

In this example the 3rd element and 4th element will be NULL. Therefore when the value is assigned to fewer elements the element will be assigned NULL by the compiler for character array whereas for numeric array zero will be assigned.

TOP

Initializing One Dimensional Array

Method 1:

```
int z[5] = {11,22,33,44,55};
char t[6] ={‘a’,’p’,’p’,’l’,’e’};
```

All the array elements are assigned values. The last element for the character arrays must be provided for NULL character.

Method 2:

```
int z[5] = {11,22,33};
char t[6] ={‘a’,’p’,’p’};
```

Only first three elements are assigned values the remaining elements will be assigned zero for integer data type. The remaining element for character array will be assigned NULL.

Method 3:

```
int z[5] = {0};
char t[6] = “apple”;
```

All the array elements are assigned zero. The character array can also be initialized as above.

Method 4:

```
int z[ ] = {11,22,33};
char t[ ] = “apple”;
```

Since array size is not defined. The array must be initialized, therefore the size of the array will be same the number of values assigned. Therefore the integer array size for this example is three since three values are provided in the initialization. The array size for character array t is 6, five for the character and one for the NULL terminator.

To assign zero to a specific element the location of the elements must be specified. Thus the array values are assigned sequentially. To assign value element by element then assigning method must be adopted.

```
z[5] = {0,22,33,44,55};
z[5] = {0,22,33,0,55};
```

Assigning value to array by program:

```
int x[5],i;
for (i=0; i<5; ++i)
   { scanf(“%d”, &x[i]);
   }
```

Reading value to array by program:

```
int x[5],i;
for (i=0; i<5; ++i)
   { printf(“%d”, x[i]);
```

TOP

```
}
```

Remember, the initial value for array must start with zero and the condition part must be (<) less than sign, and the value must be the size of array, and the increment must by one, to assign value element by element.

Caution exceeding array boundary:

int x[3];

x[4]=10; In this example a value has been assigned to the fifth element which exceeds the array size x, which is 3. The compiler will not give error during compilation. Therefore care must be taken to avoid exceeding the array boundary when assigning value to element.

Two Dimensional Arrays:

It is similar to matrix with multiple rows and columns. This type of array is useful when working with more than one set of data at a time.

Declaring Two Dimensional Arrays:

int marks[10][3];

The array has 10 rows and 3 columns. The values in the array are located by row and column, just like in matrix. Similar to one dimensional array the row and column elements start from zero not one.

Assigning values to Two Dimensional Arrays:

Consider the values in the following matrix.

	C0	C1	C2
R0	11	22	33
R1	44	0	55
R2	0	2	4

The above example has 3 rows and 3 columns. The value of the matrix can be assigned to the array as follows.

```
int rc[3][3];
rc[0][0]=11;   rc[0][1]=22;   rc[0][2]=33;
rc[1][0]=44;   rc[1][1]=0;    rc[1][2]=55;
rc[2][0]=0;    rc[2][1]=2;    rc[2][2]=4;
```

Initializing Two Dimensional Arrays:

Method 1:

Consider the following example for the matrix to be initialized:

int rc[3][3] = {11,22,33,44,0,55,0,2,4};

In this example the first row will be 11, 22, and 33; the second row 44, 0, and 55; and the third row 0, 2, and 4. Therefore it is clear that the values will be assigned in row wise.

TOP

Method 2:

The method 1 can be written as follows to get the same result.

```
int rc[3][3] = {{11,22,33},{44,0,55},{0,2,4}};
```

Here each braces { } represents each row, starting from the first row. The method two can be written to look like matrix without any change in the syntax

```
int rc[3][3] = {{11,22,33},
                {44,0,55},
                {0,2,4}
                };
```

Method 3:

The method 1 can be written as follows to get the same result.

```
int rc[ ][3] = {{11,22,33},{44,0,55},{0,2,4}};
```

In this example the row will be set to 3 since row size is not specified the number of rows will be decided by the initial values.

Method 4:

```
int rc[3][3] = {{33},{0,55,0},{0}};
```

In this method the value in the array will be as follows

```
rc[0][0]=33;   rc[0][1]=0;    rc[0][2]=0;
rc[1][0]=0;    rc[1][1]=55;   rc[1][2]=0;
rc[2][0]=0;    rc[2][1]=0;    rc[2][2]=0;
```

Method 5:

```
int rc[3][3] = {{0},{0},{0}};
int rc[3][3] = {0,0,0};
```

In this method the value in the entire array element will be zero.

```
rc[0][0]=0;    rc[0][1]=0;    rc[0][2]=0;
rc[1][0]=0;    rc[1][1]=0;    rc[1][2]=0;
rc[2][0]=0;    rc[2][1]=0;    rc[2][2]=0;
```

Assigning value to array elements by program:

```
int rc[3][3],i,j;
for (i=0; i<3; ++i)
  {for (j=0;j<3;++j)
      { scanf("%d", &rc[i][j]);
      }
  }
```

Reading value from array elements by program:

```
int rc[3][3],i,j;
for (i=0; i<3; ++i)
  {for (j=0;j<3;++j)
      { printf("%d", rc[i][j]);
```

TOP

```
            }
        }
```

Remember for both row and column, the initial value for array must start with zero and the condition part must be less than sign (<) and the value must be the size of array, and the increment must by one, to assign value element by element.

Multidimensional Array:

```
int rcm[3][3][3];
float table [3][4][5][6];
```

This type of array used when breaking down the reports to different or several levels. Therefore three or more "for" control structures are required to locate an element.

Dynamic Array:

So for we have seen array of fixed size or limit. Some time a program may require array of varying limit, here limit is unknown or changes frequently. In such situation one cannot fix the largest possible size, because creating a large fixed array the program will require large memory and the process time will be more even when array element is small. To avoid this complexity dynamic array can be preferred. Normally in heavy sorting, and ordering in the database dynamic array is chosen.

The dynamic array can be created during runtime using following C function.

1. malloc
2. calloc
3. realloc

These files are located in <stdlib.h.> file. It is extensively used in linked lists, stacks, and queues in the data structures.

Advantages of Array:

1. It reduces the programming length significantly.
2. It has better control over program for modification.
3. Coding is simple, efficient, and clear.
4. It can handle large set of data with simple declaration.

Rules of Array:

1. It does not check the boundary.
2. Processing time will increase when working with large data because of increased memory.
3. The array element start with zero not 1.
4. Character array size must be one element greater than data for NULL value.
5. One for control structure is required to assign and read value to one dimensional array.
6. Two for control structures are required to assign and read value to two dimensional arrays.

Simple programs:

Example1 : Multiplication Table

```
void main()
{
int row=0,col=0,x,rc[12][12],v;
clrscr();
```

TOP

```
for (row=0; row<12; ++row)
  { for (col=0;col<12; ++col)
    { rc[row][col] = (row+1)*(col+1); //assigning value to array element
      printf("%4d",rc[row][col]);//reading value for array element
    }
  printf("\n");
  }
getch();
}
```

Example 2: Find the Maximum

```
void main()
{
int i=0,x,a[5] ={32,44,11,3,6};
clrscr();
x=a[0];
for (i=1; i<5; ++i)
 { if (x<a[i])
       x=a[i];
  }
  printf("\nThe value is %d.", x);
getch();}
```

Example 3: Sorting

```
void main()
{
int i=0,j=0,x,a[5] ={32,44,11,3,6};
clrscr();

for (i=1; i<5; ++i)
 { for(j=1; j<5; j++)
      {if(a[j-1] >= a[j])
            {x=a[j-1];
             a[j-1]=a[j];
             a[j]=x;
            }
       }
  }
  for(i=0;i<5;i++)
    {printf("\nThe value of element %d is %d.", i, a[i]);
    }
getch();
}
```

Example 4: Prime Number

```
//Wilson formula r=(n-1)!%n, if the r is equal to (n-1)
```

TOP

```
//then n is prime number otherwise n is not a prime number.

int r,fact=1,i,n;
void main()
{
clrscr();

printf("Print Prime Numbers of:");
scanf("%d", &n);
n=n-1;
for(i=1;i<=n;i++)
  {fact=fact*i;
  }
printf("Factorial of %d is %d\n", n,fact);
n=n+1; //set to initial value of n
r=fact%n;
printf("Remainder is %d\n", r);
if (r==n-1)
        printf("%d is a prime number.",n);
else
        printf("%d is not a prime number",n);
getch();
}
```

Example 5: Fibbonaci Number

```
//It is number added to previous number
#include <stdio.h>
#include <conio.h>

int x, y, a,b, c, sum;
void main()
{
clrscr();
x=0, a=1,b=1,c=0;
printf("Print Fibonacci Number of:");
scanf("%d", &y);

for(x=0; x<=y;++x)
  { a=b;// a is 1
    b=c; // b is 0
    c=a+b;// c is 1
    printf("%d\n", c);
  }
  printf("Fibonacci numbers from 1 to %d\n", y);
```

TOP

```
getch();

}
```

Example 6: Armstrong Number

```
#include <stdio.h>
#include <conio.h>

// Armstrong number is sum of the cube of the each digit in the number
// Example 153; 1^3+5^3+3^3 = 153
int a,b,c,d,y,ans;
void main()
{
clrscr();
printf("Print Armstrong Number: \n");
scanf("%d", &y);
for (a=1; a<=y; a++)
{       b = a%10; //remainder assigned to b
        c = (a%100)/10; // remainder divided by 10 and the integer value only assigned to c
        d = a/100; //integer value assigned to d; fraction ignored
        ans = (b*b*b)+(c*c*c)+(d*d*d);
        if (a==ans)
          printf("Armstrong number is %d\n", a);
}
getch();
}
```

Character Array and String

So for array of numeric data types such as integer, float, and double was discussed. Array of character data type can also be declared. Previously data type of just character was discussed, such data type can store only one character at a time, to store more than one character a data type of character array or string must be declared. String is similar to character but the array size is not mentioned or specified. It is a pointer variable because when declaring a string variable the variable name must be prefixed with point sign (*).

Declaring Character, Character Array and string:

1. Character: char x;
2. Character Array: char t[6], char[10][10];
3. String: char *s

Declaring and Initializing String Variable

Declaring:

```
    char city[15];
```

Assigning:

```
    city[0]='c';
    city[1]='h';
```

TOP

city[2]=’e’;
city[3]=’n’;
city[4]=’n’;
city[5]=’a’;
city[6]=’i’;

Declaring and Assigning is called initialization:

char city[15] = {‘c’,’h’,’e’,’n’,’n’,’a’,’i’};
char name[15]=”Einstein”;
char dept[ ]= “Computer Science”;

String values must be assigned either element by element or in wholly in the initialization. The following statements are invalid.

char city[3]=”Chennai; -- Because element value exceeded array limit
char name[10];
name = “Armstrong”; -- String value cannot be initialized separately.
name = city; – Cannot one assign value of one string variable to other string variable.

Handling Input and Output of Character Array and String:

It is already discussed in Managing Input and Output, refer to unit II.

String Comparison:

The string variable or character cannot be compared with relational operator ==. The comparison can be done only through the function strcmp or it can be compared character by character through a program.

Therefore if(s1==s2) is invalid for character array.

Built-in String functions:

There are several string functions to work with string variables and its values. These functions are available C header file called string.h.

Consider the following example:

char string1[15]=”Einstein ”
char string2[5]=”College”
1. Concatenation String

strcat(string1,string2);

This function appends or adds the value from string2 to string1. Now the string1 will be “Einstein College”. To add the string2 to string1 the size of the string1 must be sufficient enough to fit the string1 and string2. Now string1 = “Einstein College” and string2 = “College”.

2. Copying String

srtcpy(string1,string2)

TOP

This function copy's the value of string2 to string1. Now the string1 will be "College". To add the string2 to string1 the size of the string1 must be sufficient enough to fit the value of string2. This function will replace the existing value string1 with string2. Now string1 and string2 are "College"

3. String comparison

    strcmp(string1,string2)

This function compares the value from string2 with string1. If both the string1 and string2 are exactly the same then the function will return zero or else it will return some positive or negative value. For the above example the function will return negative of positive value. Here string1 and string2 will not change.

4. Concatenation String

    strncat(string1,string2,n);

This function appends or adds the first n value from string2 to string1. Assume n =3, now the string1 = "Einstein Col" and string2 = "College"

5. Copying String

    srtncpy(string1,string2,n)

This function copy's the first n value of string2 to string1. Assume n =4 now the string1 = "Coll" and string2 = "College".

6. String comparison

    strncmp(str1,str2,n)

This function compares the first n value from string2 with string1. Assume n=5, now string1 = "Einstein " and it reads "Colle" from string2 now it compares the string1 and string2. Therefore string1 is not as same string2. If string1 and string2 are same the function will return 0.

7. Find a value in string

    strstr(string1, string2);

This function will find the value of string2 in string1. Assume string1 as "Apple" and string2 as "Ap", now the function will return position of first occurrence of "Ap", since "Ap" is found in "Apple".

8. Find the first occurrence of character

    strchr(string1, 'i');

In the above example 'i' the first occurrence of 'i' is the second character of the word "Einstein", the function return 2.

9. Find the last occurrence of character

    strrchr(string1, 'i');

In the above example; the last occurrence of 'i' is the seventh character of the word "Einstein", the function return 7.

10. Reversing a string

    strrev(string1); This function reverse the data of string1 and stores it in string1.

TOP

11. Length of String

strlen(string1); This function will return length of the string. For the above example it returns 8.

Character Arrays as Table

Just like two dimensional array for numbers the character array can also be declared in two dimensional form or table.

```
char name[3][15] = {  “Apple”
                      “Bannana”
                      “Pineapple”
                     }
```

Example 6: Palindrome of string data

```
//To check whether the data is Palindrome
#include <stdio.h>
#include <conio.h>
#include <string.h>
int r;
char s1[15], s2[15];
void main()
{
clrscr();
printf("Enter anything :");
scanf("%s", s1);

        strcpy(s2,s1);//copy's s1 to another variable s2
        strrev(s2);   //reverse the value of s2
        printf("%s\n", s1);
        printf("%s\n", s2);
        r= strcmp(s1,s2);
        if (r==0)
                printf("It is a Palindrome %s\n", s1);
        else
                printf("It is not a Palindrome %s\n", s2);
getch();
}
```

TOP

Functions

Functions are two types Built-in functions and user-defined functions. Functions are created when the same process or an algorithm to be repeated several times in various places in the program. By creating function the source code is reduced significantly. Function is a derived data type.

Functions are two types:

1. Built-in / Library function
   ex:) printf, scanf, strcmp etc.
2. User defined function.

Advantage of User defined function:

1. Reduce the source code
2. Easy to maintain and modify
3. It can be called any where in the program.

Body of user defined function:

```
return_type f_name (argument1,rgument 2)
        { local variables;
          statements;
          return_type;
        }
```

The body of user-defined shows that an user-defined functions has following characteristics.

1. Return type
2. Function name
3. Parameter list of arguments
4. Local variables
5. Statements
6. Return value.

Note: The function with return value must be the data type, that is return type and return value must be of same data type.

User defined function has three parts:

1. Declaration part
   ret_type f_name (arguments);
2. Calling part
   f_name(arguments);
3. Definition part
   As discussed earlier in the body of user-defined function

Categories of User defined function:

1. Function without return value and without argument.
2. Function without return value and with argument.
3. Function with return value and with argument.
4. Function with return value and without argument
5. Function with more than one return value.

TOP

Note: Function with more than one return value will not have return type and return statement in the function definition.

Consider the following example to multiply two numbers:

```
void main( )
 { int x,y,z;
     scanf(“%d%d”, &x,&y);
     z=x* y;
    printf(“The result is %d”, z);
}
```

1. Function without return value and without argument

```
void f_name (void)
            { local variables;
              statements;
            }
```

The above example can be rewritten by creating function

```
void f_mult(void )
 { int x,y,z;
     scanf(“%d%d”, &x,&y);
     z=x* y;
    printf(“The result is %d”, z);
}
```

2. Function without return value and with argument

```
void f_name (int x, int y)
            { local variables;
              statements;
            }
```

The above example can be rewritten by creating function

```
void f_mult(int x, int y )
 { int z;
     z=x* y;
    printf(“The result is %d”, z);
}
```

3. Function with return value and without argument

```
int f_name (void)
            { local variables;
              statements;
              return int;
            }
```

The above example can be rewritten by creating function

```
int f_mult(void )
 { int x,y,z;
```

TOP

```
            scanf(“%d%d”, &x,&y);
            z=x* y;
           printf(“The result is %d”, z);
           return(z);
       }
```

4. Function with return value and with argument

```
int f_name (int x, int y)
        { local variables;
          statements;
          return int;
        }
```

The above example can be rewritten by creating function

```
int f_mult(int x, int y)
 { int z;
     z=x* y;
    printf(“The result is %d”, z);
    return(z);
}
```

5. Function with multiple return values

```
void f_name (int *x, int *y)
        { local variables;
          statements;
        }
void f_mult(int *x, int *y)
 { int a=10, b=20;
    *x = a;
    *y = b;
}
```

The values to the argument are passed through address variable (&v). Function with more than one return value will not have return type and return statement in the function definition.

In the above five types an example for function definition of each type is shown. After defining a function it can be called in the main program only after it is declared in the program. Just like any other primary data type a data type has to be declared before it can be used. The calling and declaration function is shown below.

Declaring and Calling a defined function:

```
int f_mult(int r, int s); // Function declaration
void main( )
{ int a,b,c;
  a = f_mult(b,c); //Function calling
}
```

TOP

```
int f_mult(int x, int y) //Function definition or sub program
 { int z;
      scanf("%d%d", &x,&y);
      z=x* y;
     printf("The result is %d", z);
     return(z);
}
```

Calling function with multiple return values:

```
void f_mult(int a, int b, int *x, int *y);
void main ( )
{int r=10,s=20 p, q;
f_mult(r,s,&p, &q);
printf("%d%d", p,q); // p will be 10 and q will be 20
}
void f_mult(int a, int b, int *x, int *y)
 { *x = a;
    *y = b;
}
```

Recursion:

Recursion is calling function by itself in the definition.

```
int f_name (int x)
{ local variables;
 f_name(y); // this is recursion
  statements;
}

int factorial(int n)
{ int fact;
if (n==1)
   return(1);
else
   fact = n*factorial(n-1);
return(fact);
}
```

Nesting of functions:

When a function is called within other function or same function; outside the definition area is called nesting. Except defined function; all other function called in the definition section is also nesting. Function can be nested with any functions either built-in or user defined. When nesting a function the data type of the argument of one function must match with return type of other function.

A function can be called inside

1. Built-in C-functions

TOP

2. Other user defined functions in definition
3. Other user defined functions in function calling

Example:

```
int f_mult(int r, int s); // Function declaration
void main( )
{ int a,b,c;
  a = f_mult(f_mult(b,c),c); //Nesting of function
  printf("%d", f_mut(a,b)); //Nesting of function
}
int f_mult(int x, int y) //Function definition or sub program
 { int z;
    scanf("%d%d", &x,&y);
    z=x* y;
   printf("The result is %d", z);
   return(z);       }
```

Passing array to function for both numeric and character:

```
int f_array(int a[], int e);
void main()
{       int a[5], int e=5;
        clrscr();
        f_array(a, e); //passing array to function
        getch();
}
int f_array(int a[], int e)
{ int i=0;
  for (i=0; x<e; ++i)
 { a[i] = i*i; //assigning value to array element
   printf("\nThe value of element %d is %d.", i, a[i]);
 }
 return(0);
}                {
```

Note: Array and string cannot be passed by value, it is passed through variable.
}

Passing values to function

1. Pass by reference (Pointer argument) Example: pointer as argument
2. Pass by value through variable Example: Other argument
3. Pass by value through constant

Storage Class and its scope:

1. Automatic variables Ex: local variables
2. Static variables Ex: It acts like local variables till end of the function in a loop
3. External variables Ex: Global variable
4. Register Variables Ex: memory is placed in the register for a variable to work faster.

TOP

STRUCTURES & UNIONS

Structure is a user-defined data type. Structures help to work with set of data of different or same data type. The difference between a structure and an array is array can store only data of one data type; whereas structure can store multiple data type at a time.

Declaration of structure:

Since structure is a data type it has to be defined or declared before it can be used effectively. Structure can be declared in three methods. The variables or data type inside the structure are called members of the structure. The data type can also be array.

Method 1:

Defining a structure:

Syntax:- struct tag_name {data type1, data type2……..};

```
struct str_learn1{      int i,x;
                        float t;
                        char a;
                };

struct str_learn1 str_grp;
```

Method 2:

```
struct str_learn2{      int i,x;
                        float t;
                        char a;
                } str_name, str_names;
```

Method 3:

```
struct {int i,x;
        float t;
        char a;
       } str_learn3, str_learn33;
```

In real work environment the third method is not followed because the structure does not have a tag name. Therefore the structure cannot be declared in the later stage of the program.

Assigning values to structure member:

Method 1: Define, declare, and Assign:

```
struct str_learn1{      int i,x;
                        float t;
                        char a;
                };
struct str_learn1 str_grp;
str_grp.i=1;
```

TOP

```
        str_grp.x=10;
        str_grp.t=12.345;
        str_grp.a='g';
```

Method 2: Initializing is Define, Declare & Assign:

```
        struct str_initial{     int i,x;
                                float t;
                                char a;
                        } str_ini ={1,2,5,'b'}, str_ini2={1,2};
```

Method 3: Define, Declare and Assign.

```
        struct str_initial{     int i,x;
                                float t;
                                char a;
                        }
struct str_initial str_ini3={1,2,5.66,'z'};
```

Members of structures:

Consider the following example:

```
                struct str_initial{     int i,x;
                                        float t;
                                        char a;
                                }str_ini;
```

In the above example the variables i,x,t, and a are called members of the structure str_initial;

Accessing Members:

The members are accessed as follows.

```
                structure_name.member;
                str_in.x=10;
```

Copying and Comparing Structure Variables:

Consider the following example

```
        struct str_initial{     int i,x;
                                float t;
                                char a;
                        } str_ini1 ={1,2,5.23,'b'};
        struct str_intial str_in2;
```

For the str_in2, the values of str_in1 can be copied to the variables of str_in2 by following statement.

```
        str_in2 = str_in1;
```

To copy from variables values of one structure to another structure through above statement, the order of the data type must be exactly the same in other words both the structure must be identical in all aspects.

TOP

Member values from one structure can be copied to member of another structure; but the structures cannot be compared; only the members of the structures can be compared.

Example:

```
str_in1 == str_in2;
```

The above statement is invalid because structures cannot be compared only its member can be compared.

```
str_ini1.x == str_in2.x;
```

This is valid statement.

Structure as Array:

```
struct str_initial{     int i,x;
                        float t;
                        char a;
                   };
```

The above structure can be declared as array.

```
struct str_initial str_ini[10];
```

Array inside structure:

```
struct str_initial{     int x[10];
                        float t;
                        char a;
                   }str_ary[5];
```

The above structure can be declared as array.

```
struct str_initial str_inia[10];
struct str_initial str_ini;
```

Structure inside a structure or sub-structure:

Example 1:

```
struct str_initial{     int i,x;
                        float t;
                        char a;
                        struct { int y;
                                float z;
                                char w;
                                } str_grand_child1, str_grand_child2;
                   }str_parent;
```

To access a substructure the syntax is

```
str_parent.str_grand_child1.y;
str_parent.str_grand_child2.z;
```

TOP

Example 2:

```
struct str_learn1{      int x;
                        float t;
                        char a;
                        struct { int sx;
                                 float st;
                                 char sa;
                               }grand_child1, grand_child2;
                 };
```

Declaring a sub-structure:

```
struct str_learn1 str_grp;
```

Assigning values of structure member:

Method 1:

```
str_grp.x=10; str_grp.t=12.345; str_grp.a='g';
str_grp.grand_child1.sx=20;
str_grp.grand_child1.st=22.345;
str_grp.grand_child1.sa='h';
```

Member values cannot be assigned from structure to sub-structure or vice versa even when the members are same. The following statements are invalid.

```
str_grp.grand_child1=str_grp;
str_grp=str_grp.grand_child1;
```

Method 2 Initializing:

```
struct str_learn2{      int x;
                        float t;
                        char a;
                        struct {  int sx;
                                  float st;
                                  char sa;
                               }grand_child1, grand_child2;
                 }str_met2={1,2,'a',3,4,'b',5,6.7,'y'};
```

Method3

```
struct str_learn2 str_ini3={1,5.66,'z',6,7,'m',7,9.9,'r'};
```

Structure and Function:

Remember, a function can have an argument and return type. Previously in the function we have seen function passing and returning primary data type. Similarly structure can be passed and returned in a function. Consider the following structure.

```
struct str_fun {int x;
                float y;
                char z;
                }str_funpass;
```

Now, a function can be defined with structure as argument and return type.

```
struct str_fun f_structure (struct str_fun str_arg)
```

TOP

```
{       str_arg.x=10;
        str_arg.y=12.34;
        str_arg.x='Y';
        return (str_arg);
}
```

Note: The return value must have structure tag name not declaration name.

The above function can be called as follows. Since function returns a structure, a structure has to be declared similar to the structure argument. Therefore to call this function follow; these steps

```
struct str_fun f_structure (struct str_fun str_arg);
void main( )
{
struct str_fun str_return;
str_return = f_structure(str_funpass);
printf("%d\t%f\t%c",str_return.x,str_return.y,str_return.z);
}
```

The output will be 10 12.34 Y.

Union:

The union is defined and declared just like the structure. The disadvantage in using the union is, it can hold only one data or value at a time. That is it can hold value only for one member.

```
union  u_var { int x;
               float y;
               char z;
             } u_declare;
```

The above union will be provided memory space of 4 bytes since float is the largest data type in the union; and it allocates float size as the memory. To store all the values of the member in the union, it requires 7 bytes because (int 2 bytes, float 4 bytes, and char 1 byte). Because of this only one member can be assigned or read at a time.

Only the first member can be initialized in union.

```
union u_var u_assign ={100};
```

Since only the first member can be initialized the statement is invalid because trying to assign value to two members.

```
union u_var u_assign ={100,23.45};
```

Since the first member of the union is integer the initialization fails because the data type of assigning value is float.

```
union u_var u_assign ={23.45};
```

```
u_declare.x=10;
u_declare.y=12.34;
printf("%d",x); This statement will be ineffective
printf(%f",y); Output will be12.34
```

TOP

From the above statement it is clear that the union stores only one member data at a time because of that union is rarely used.

The difference between structure and union is; memory space is provided for all the members in the structure, whereas in union memory space is provided only for the largest member of the union. The members work exactly like the primary data type, the only rule is the member must be identified through the structure. Remember parent-child relation for structure and grandparent-parent-child relation for substructure.

TOP

POINTERS

Pointer is a variable which stores the address of another variable. A pointer can also store the address of another pointer which is called chain of pointer. The values of variable are read through the pointer. Since the pointer locates the address immediately the process time is considerably reduced. The data type of the pointer variable must be same as the data type of variable. Remember pointer is a variable so it has also address.

Declaration of pointer:

Method 1: int *p;
Method 2: int* p;
Method 3: int * p;

Pointer can be declared in these three methods. Let us follow the first method which is widely adopted.

Assigning value to pointer:

It is not necessary to assign value to pointer. Only zero (0) and NULL can be assigned to a pointer no other number can be assigned to a pointer. Consider the following examples;

int *p=0;
int *p=NULL; The above two assignments are valid.

int *p=1000; This statement is invalid.

Assigning variable to a pointer:

int x; *p;

p = &x; This is nothing but a pointer variable p is assigned the address of the variable x. The address of the variables will be different every time the program is executed.

Reading value through pointer:

int x=123; *p;
p = &x; Here the pointer variable p is assigned the address of variable x.
printf(“%d”, *p); will display value of x 123. This is reading value through pointer
printf(“%d”, p); will display the address of the variable x.
printf(“%d”, &p); will display the address of the pointer variable p.
printf(“%d”,x); will display the value of x 123.
printf(“%d”, &x); will display the address of the variable x.

Note: It is always a good practice to assign pointer to a variable rather than 0 or NULL.

Chain of pointers:

A pointer can point to the address of another pointer. Consider the following example.

int x=456, *p1, **p2;
p1 = &x;
p2 = &p1;

TOP

printf(“%d”, *p1); will display value of x 456.

printf(“%d”, *p2); will also display value of x 456. This is because p2 point p1, and p1 points x. Therefore p2 reads the value of x through pointer p1. Since one pointer is points towards another pointer it is called chain pointer. Chain pointer must be declared with ** as in **p2.

Pointers and expressions:

An expression can be written using pointer variable. Consider the following example.

int x =100, y =50, *p1, *p2, z;
p1 = &x;
p2 = &y;
z =x +y; The value of x is obtained without using pointer.
z = *p1 + *p2; The value of x is obtained using pointer.

Similar it can be done for all the operators we have seen earlier, but care must be taken when using division operator and pointer. Consider the following example

z = *p1/*p2. In this statement the value of z will be 100 not 2 because the compiler reads /*p2 as comment not as operators. To avoid such error the above statement must be written as follows;

z = *p1/ *p2. There must be a space between / and *p2. Now z is 2.

Pointers increments and Scale factor & Pointer and Array:

Pointer variable can be assigned to an array. The address of each element is increased by one factor depending upon the type of data type. The factor depends on the type of pointer variable defined. If it is integer the factor is increased by 2. Consider the following example.

int x[5]={11,22,33,44,55}, *p;

p = x; Remember, earlier the pointer variable is assigned with address (&) operator. When working with array the pointer variable can be assigned as above or as shown below.

p = &x[0]; Therefore the address operator is required only when assigning the array with element.

Assume the address on x[0] is 1000 then the address of other elements will be as follows

x[1] = 1002
x[2] = 1004
x[3] = 1006
x[4] = 1008 The address of each element increase by factor of 2. Since the size of the integer is 2 bytes the memory address is increased by 2 bytes, therefore if it is float it will be increase 4 bytes, and for double by 8 bytes. This uniform increase is called scale factor.

p = &x[0]; Now the value of pointer variable p is 1000 which is the address of array element x[0]. To find the address of the array element x[1] just write the following statement.

p = p + 1; Now the value of the pointer variable p is 1002 not 1001 because since p is pointer variable the increment of will increase to the scale factor of the variable, since it is integer it increases by 2. The p = p + 1; can be written using increment or decrement operator ++p;

The values in the array element can be read using increment or decrement operator in the pointer variable using scale factor. Consider the above example.

TOP

printf(“%d”, *(p+0)); will display value of array element x[0] which is 11.
printf(“%d”, *(p+1)); will display value of array element x[1] which is 22.
printf(“%d”, *(p+2)); will display value of array element x[2] which is 33.
printf(“%d”, *(p+3)); will display value of array element x[3] which is 44.
printf(“%d”, *(p+4)); will display value of array element x[4] which is 55.

Method 1: Pointer point to array

```
void main()
{       //pointers and array one D
        {int x[2], *p;
        p=x; //or &x[0];
        x[0]=10;x[1]=20;
        printf("%d\n%d", *(p),*(p+1));
        }
        {//pointers and array two D
        int x[2][2], *p;
        p=&x[0][0];
        x[0][0]=10;x[0][1]=20;
        x[1][0]=30;x[1][1]=40;      //*(p+(2xr)+c))
        printf("\n%d\t%d\n%d\t%d", *(p),*(p+1),*(p+2),*(p+3));
        }
}
```

Method 2: Pointer as Array

```
void main()
{       //pointers and array one D
        {int (*p)[2],a[2]; //(*p) create pointers for two rows
        p=&a;
        a[0]=10;a[1]=20;
        printf("%d\n%d",*(*p), *(*p+1));
        }
        {//pointers and array two D
        int a[2][3]={{1,2,3},{4,5,6}};
        int (*p)[3]; //(*p) create pointers for two rows
        p=a;
        printf("\n%d\t%d\t%d\n%d\t%d\t%d",
        *((*p+0)+0),*((*p+0)+1),*((*p+0)+2),*(*(p+1)+0),*(*(p+1)+1),*(*(p+1)+2));
        }
}
```

Method 3: Array of pointers

```
void main()
{       //pointers and array one D
        {int *p[2],a[2];
        p[0]=&a[0];
```

TOP

```
        a[0]=10;a[1]=20;
        printf("%d\n%d",*(*p), *(*p+1));
        }
        {//pointers and array two D
        int a[2][3]={{1,2,3},{4,5,6}};
        int *p[3]; //*p pointing to array
        p[0]=&a[0][0];
        p[1]=&a[1][0];
        printf("\n%d\t%d\t%d\n%d\t%d\t%d",
        *((*p+0)+0),*((*p+0)+1),*((*p+0)+2),*(*(p+1)+0),*(*(p+1)+1),*(*(p+1)+2));
        }
}
```

Pointers and character string:

A pointer can be declared as character string. Consider the following example.

char name[25]; In this statement the variable name is assigned 25 bytes to hold 25 characters. The size is constant therefore even if the variable name is assigned 5 characters it allocates 25 bytes when only 5 bytes is required. This can be avoided by declaring character array as character string. Consider the following example.

char *names; The size of the variable names will be adjusted to the size of the data. In other word if the data is 5 characters it will assign 5 bytes and if the data is 30 characters it will assign 30 bytes.

```
char name[25] = “Character array is OK”;
char *names;
names = “Character String is BEST”;
```

Character string allows the values to be assigned later, whereas character cannot be assigned as a whole it has to be assigned one by one character or element by element. It can be assigned as a whole only if it is initialized.

Array of pointers:

```
char *name[3] = {“Tamil Nadu”, “Tirunelveli”, “I BE ECE”};
```

In the above example the first row is “Tamil Nadu”, second row “Tirunelveli”, and the third row is “I BE ECE”, the column size is different for each this is called **“Ragged Arrays”.**

```
name[0] = “Tamil Nadu”;
name[1] = “Tirunelveli”
name[2] = “I BE ECE”
```

Pointers and functions:

Function returning pointer:

```
int *f_ret_ptr (int *y);
void main( )
{int b=20, *p;
  p = f_ret_ptr(&b);
 printf("%d", *p);
 getch();
```

TOP

```
}
int *f_ret_ptr(int *s)
{return (s);
}
```
The value of a will be *p is 20.

Pointer as function arguments:

As stated in the function, a function can have pointer variable as argument. Value is passed to the pointer argument in the function by reference or address. Consider the following example

```
void f_ptr_arg (int *x, int *y);
void main( )
{int a, b;
  f_ptr_arg(&a, &b);
 printf("%d\t%d", a, b)
}
void f_ptr_arg(int *r, int *s)
{*r = 10; *s = 20;}
```
The function will return 10 and b 20 through address of a and b.

Pointers to function:

A pointer can be declared as function to point to a function. A pointer pointing to function must have the same return type and argument in the pointer variable. Earlier a pointer was declared as a primary data type.

```
int x, *p;
p = &x;
```

A pointer can be declared to point to a function as follow

```
data_type (*fptr) ( );
```

Consider the following example

```
int f_ptr(int x);
void main()
{       clrscr();
        int (*ptr)(int x);
        int a = 10,r=0;
        ptr=f_ptr;
        r=(*ptr)(a);
        printf("%d",r);
        r = f_ptr(a);
        printf("%d",r);
}
int f_ptr(int y)
{ return (y+10);
}
```

TOP

Pointers and Structure:

```
struct str_ptr_tag
{int x;
 float y;
 char z;
};
void main()
    {clrscr();
    struct str_ptr_tag str_data, *str_ptr;
    int a = 10;
    float b=12.34;
    char d = 'H';
    str_ptr=&str_data;
    str_data.x=a;
    str_data.y=b;
    str_data.z=d;
    printf("%d",str_ptr->x);
    printf("%f",str_ptr->y);
    printf("%c",str_ptr->z);
}
```

THE PREPROCESSOR

The C preprocessor is a tool to make program easy to read, modify, portable, and efficient. It improves the readability of the program and the speed. The preprocessor directives are divided into three categories.

1. Macro substitution directives
2. File inclusion directives
3. Compiler control directives.

Rules of defining preprocessor:

1. No space between # and define
2. No space between identifiers (ex: NOT EQUAL)
3. The string value (10, 1, 3.14, “TAMIL NADU”) will take appropriate data type.
4. String can also be an expression.
5. Identifier is represented by capital letter, it is not a rule. But it will be easy to find the directives.
6. Directives are constant therefore the string value cannot be changed later in the program.
7. If the macro is define with semicolon then the macro is called without semicolon at the end, otherwise use semicolon to end the statement

Macro Substitution directives:

1. **Simple Macro:**

```
#define     COUNT       10
#define     TRUE        1
#define     PI          3.1415956
```

TOP

```
#define     STATE          "TAMIL NADU"
#define     EQUALS         = =
#define     AND            &&
#define     OR             ||
#define     NOT_EQUAL      !=
#define     START          main( ) {
#define     END            }
#define     NEXT_LINE      printf("\n");
```

Example

if (x = = y && y = = z) can be rewritten as if (x EQUALS y AND y EQUALS z), thus the statement becomes easy to read and understand.

2. **Macro with expression:**

```
#define     A          (50-40)
#define     D          (20-10)
```

int x; x = A/D; If the strings are not placed in the bracket it will result in logical error.

3. **Macro with Argument:**

```
#define     SQUARE (x)     ((x) * (x))
#define     CUBE(x)        (SQUARE(x) * (x))
```

SQUARE is macro with argument and the CUBE is nesting of macro with SQUARE macro as one of its arguments.

Note: It is always good practice to place bracket ( ) in expression and arguments.

File inclusion directives:

```
#include "filename"   or
#include <filename>
```

Compiler control dicrectives:

Example:

```
#ifndef TEST
        #define TEST
#endif
```

TEST is a file containing macro definitions. If it is not defined (ifndef) in the file header, then the TEST file will be included in the file header.

Example:

```
#ifdef TEST
        #undef TEST
#endif
```

TOP

TEST is a file containing macro definitions. If it is defined (ifdef) in the file header, then the TEST file will be included in the file header.

The syntax #define TEST and #undef TEST cannot be used as a single separated statement, it should be placed within the if block as shown.

Guidelines to 'C' Program

Program Design

1. Problem Analysis.
2. Outlining the program structure.
3. Algorithm Development.
4. Selection of control Structures.

   Program Coding
   1. Internal Documentation.
   2. Construction of statements.
   3. Generality of the program.
   4. Input/output formats.

Common Programming Errors

1. Missing Semicolons.
2. Misuse of Semicolons.
3. Missing Braces.
4. Missing Quotes.
5. Misusing Quotes.
6. Improper Comment Characters.
7. Undeclared variables.
8. Forgetting the precedence of Operators.
9. Ignoring the Order of Evaluation of Increment/Decrement Operators.
10. Forgetting to Declare Function Parameters.
11. Mismatching of Actual and Formal Parameter Types in Function Calls.
12. Non declaration of Functions
13. Missing & Operator in Scanf Parameters.
14. Crossing the Bounds of an Array.
15. Forgetting a Space for Null Character in a string.
16. Using Uninitialized Pointers.
17. Missing Indirection and Address Operators.
18. Missing Parentheses in Pointer Expressions.
19. Omitting Parentheses around Arguments in Macro Definitions.

Some other mistakes that we commonly make are:

- ➢ Wrong indexing of loops.
- ➢ Wrong termination of loops

TOP

- Unending loops
- Use of incorrect relational test
- Failure to consider all possible conditions of a variable
- Trying to divide by zero
- Mismatching of data specifications and variable in scanf and printf statements.
- Forgetting truncation and rounding off errors.

TOP