CONTENTS

1. STRINGS

The string in C programming language is a one-dimensional array of characters which is terminated by a **null** character '\0'.

Declaration:

The C does not support string datatype, instead we declare a character array into which we read characters.

<u>Syntax :</u> char string_name[size]; // size is the number of characters in string_name and it should be maximum number of characters plus 1, i.e. for \0.

Example: char s[5];

s[0]	s[1]	s[2]	s[3]	s[4]

Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the \0 character.

Initialization:

- `char str[9]= “I Like C”;`
- `char str[9]={‘I’,’ ’,’L’,’i’,’k’,’e’,’ ’,’C’,’\0’};`
- `char str[]={‘I’,’ ’,’L’,’i’,’k’,’e’,’ ’,’C’,’\0’};`
- `char str[ ]=”I Like C”;`

I		L	i	k	e		C	\0
str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]	str[7]	str[8]

Illegal declarations:

- `char s[3]=”Hello”;` // shows the error too many initializers.
- `char s[5];`
  `s[5]=”Hello”` // can’t separate initialization from declaration.

-
  ```
  char s[4]=”abc”;
  char s1[4];
  s1=s;
  ```

Printing Strings (writing strings to screen)

A) The conversion type ‘s’ can be used for output of strings using printf().

example:
```
char str[ ]=”Hello, World”;

printf(“%s”,str);
```
Output: `Hello, World`

- Width and precision specifications may be used with %s conversion specifier. The width specifies the minimum output field width. The precision specifies the maximum number of characters to be displayed. i.e. **%w.d**
  example:

```
char s[]=”Hello, World“;      Output
printf(“$$%s$$”,s);           $$Hello, World$$
printf(“>>%20s<<”,s);        >>        Hello, World<<  (8 spaces)
printf(“>>%-20s<<”,s);       >>Hello, World        <<
printf(“++%.4s++”,s);        ++Hell++
printf(“!!-20.4s<<”,s);      !!Hell                <<
printf(“>>%20.4s<<”,s);      >>                Hell<<
printf(“>>%10s<<”,s);        >>Hello, World<<
printf(“>>%12.0s<<”,s);      >>            <<
```

- When field width is less than length of the string, entire string is printed.
- Integer value on right side implies number of characters to be printed.
- If d=0, nothing is printed.
- Minus sign means left justified and positive sign means right justified.

B) Using putchar and puts function.

- `putchar()` : A character handling function to output the value of character variables.

example:
```
char c=’A’;

    putchar(c);    //output will be A
```
(Equivalent to `printf(“%c”,c);`)

example:
```
char c[6]=”Paris”;

    for(i=0;i<6;i++)

    putchar(c[i]);
```

- `puts()` : puts() writes all of the characters (not including the \0) to the standard output file, followed by an newline character ('\n'). That makes puts(s) identical to: `printf("%s\n", s);`

example: `puts("welcome");`

Reading Strings (Entering strings or string input)

A) Using scanf() with control string %s

- Strings may be read in using the %s conversion with the function scanf() but there are some restrictions. It can't read white spaces (blank, tab, newline etc).
  example:
  ```
  char a[10];
         scanf("%s",a);
  ```
  input: NEW YORK
  output: NEW
- '&' is not required before the variable name. (Array names are pointers to the 1st element).

B) Using scanset.

- The scanset conversion facility provided by scanf() allows the programmer to specify the set of characters that are acceptable as part of the string. A scanset conversion consists of a list of acceptable characters enclosed within square brackets. A range of characters may be specified using notations such as 'a-z' meaning all characters within this range. While processing scanset, scanf will process only those characters which are part of scanset. Please note that the scansets are case-sensitive.

Example:
```
char str[128];
printf("Enter a string: ");
scanf("%[A-Z]s",str);
printf("You entered: %s\n", str);
```

Sample output:
```
Enter the string: HELLo world
You entered: HELL
```

- If first character of scanset is '^', then the specifier will stop reading after first occurrence of that character. For example, given below scanset will read all characters but stops after first occurrence of 'o'.

Example:
```
char str[128];
printf("Enter a string: ");
scanf("%[^o]s", str);
printf("You entered: %s\n", str);
```

Sample output:
```
Enter the string: Landmarks are of
You entered: Landmarks are
```

- To read anything except "enter" key.

Example:

```
char str[128];
```

```
        printf("Enter a string with spaces: ");
        scanf("%[^\n]s", str);
        printf("You entered: %s\n", str);
```

Sample Output:

```
Enter a string with spaces: I like C Programming
You Entered: I like C Programming
```

C) Using getchar()

- getchar() function is the function which is used to accept the single character from the user.

  ```
  char ch = getchar();
  ```

  The getchar() function will accept the single character. After accepting character control remains on the same line. When user presses the enter key then getchar() function will read the character and that character is assigned to the variable 'ch'.

Example:

```
         char c;
        printf("Enter character: ");
        c = getchar();
        printf("Character entered: ");
        putchar(c);
```

Sample Output:

```
       Enter character: B
       Character Entered: B
```

D) Using gets()

It is used to scan a line of text from a standard input device. The gets() function will be terminated by a newline character. The string may include white space characters.

Example:

```
       void main()
       {
        char name[20];
        printf("\nEnter the Name : ");
        gets(name);
        printf("The entered name is:");
        puts(name);
       }
```

- Whenever gets() statement encounters then the characters entered by user (the string with spaces) will be copied into the variable.
- If user starts entering characters and if new line character appears then the newline character (\n) will not be copied into the string variable (i.e. name).
- A terminating null character is automatically appended after the characters copied to string variable (i.e. name)

- Major drawback of gets() is that it does not allow to specify a maximum size for string variable (which can lead to buffer overflows, i.e. it does not know how big the array is and will continue accepting data and stores into the memory it doesn't own).

E) Using scanf() with control string %c

An alternative method for the input of strings is to use scanf() with %c conversion which may have a count associated with it. The count specifies the number number of characters to be read in. Unlike the %s and %[ ]s conversions, the %c conversion does not automatically generate the string terminating NULL, therefore we may get wrong output if wrong number of character is supplied.

Example:

```
char str[10];
printf("enter the string of 9 characters:");
scanf("%10c",str);
str[9]='\0';
printf(" the entered string is: %s",str);
```

Sample Output:

```
enter the string of 9 characters:abcd fg i
the entered string is: abcd fg i
```

Character Manipulation in String

Header file "ctype.h" includes numerous standard library functions to handle characters (especially test characters). Except for toupper() and tolower() functions, all these functions return values indicating true or false i.e. true is any non zero number and false is zero.

isalnum( )	Test for Alphanumeric
isalpha( )	Test for Alphabetic
iscntrl( )	Test for Control Character
isdigit( )	Test for Digit
isgraph( )	Test for Graphical Character (does not include a space)
islower( )	Test for Lowercase Letter
isprint( )	Test for Printing Character (does include a space)
ispunct( )	Test for Punctuation Character
isspace( )	Test for White-Space Character
isupper( )	Test for Uppercase Letter

isxdigit( ) Test for Hexadecimal Digit
tolower( ) Convert to Lowercase
toupper( ) Convert to Uppercase

1. isalnum() - isalnum( ) function checks whether a character is an alphanumeric chraracter(a-z or A-Z or 0-9) or not. If a character passed to isalnum( ) is alphanumeric(either alphabet or number), it returns non-zero integer if not it returns 0.
2. islapha() - function checks whether a character is an alphabet(a to z and A-Z) or not. If a character passed to isalpha() is an alphabet, it returns non-zero integer if not it returns 0.
3. iscntrl()- checks whether a character is a control character or not. If character passed to iscntrl() funtion is a control character, it returns non-zero integer and if not it returns 0. Those characters which cannot be printed are known as control characters. For example: backspace, Escape, newline etc.
4. isdigit() - checks whether a character is numeric character(0-9) or not. If a character passed to isdigit() is a digit, it returns non-zero integer and if not it returns 0.
5. isgraph() - checks whether a character is graphic character or not. The characters with graphical representations are all those characters that can be printed except for whitespace characters (like ' '), which is not considered as isgraph characters. If the argument passed to isgraph() is a graphic character, it returns non-zero integer if not it returns 0.
6. islower() - checks whether a character is lowercase alphabet (a-z) or not. If the argument passed to islower() is a lowercase alphabet, it returns non-zero integer if not, it returns zero.
7. isprint()- checks whether a character is printable character or not. Those characters that occupy printing space are known as printable characters. If a character passed to isprint() is printable character, it returns non-zero integer, if not it returns 0. Printable characters are just opposite of control character.
8. ispunct() checks whether a character is a punctuation or not. If the argument passed to ispunct() is a punctuation, it returns non-zero integer if not, it returns zero. A punctuation character is any graphic character (as in isgraph) that is not alphanumeric (as in isalnum).
9. isspace() checks whether a character is white-space character or not. If a character passed to isspace() is white-space character, it returns non-zero integer if not it returns 0. White space characters include space, horizontal tab(\t), vertical tab (\v), newline (\n).
10. isupper() checks whether a character is an uppercase alphabet (A-Z) or not. If the argument passed to isupper() is a uppercase alphabet, it returns non-zero integer if not, it returns zero.

11. isxdigit() - checks whether the passed character is a hexadecimal digit or not. To know more about how to represent hexadecimal number in C, go to the last page.
12. tolower() converts an uppercase alphabet to a lowercase alphabet if the argument passed is an uppercase alphabet. If the argument passed is other than an uppercase alphabet, it returns the same character passed to tolower().
13. toupper() converts a lowercase alphabet to an uppercase alphabet if the argument passed is an uppercase alphabet. If the argument passed is other than an lowercase alphabet, it returns the same character passed to toupper().

Example:

```
#include<stdio.h>
#include<ctype.h>
void main()
{

 char c1;
 printf("enter a character:");
 scanf("%c",&c1);
 if(isalnum(c1)==0)
 printf(" %c is not an alphanumeric character",c);
 else
 printf("%c is an alphanumeric character",c);

 }
```

String Manipulation

In **C** programming language, there are a lot of functions that can be used to manipulate strings. The strings manipulations involve combining strings, copying the content of a string to another string, comparing two strings, calculating the length of a string, etc.

The header file <string.h> must be used in order to access these string functions.

1. strlen(): It calculates the length of string. It takes only one argument, i.e, string name. It calculates the length of string before null character.
   Syntax: `variable = strlen(string_name);`
   Example:

```
       #include <stdio.h>
         #include <string.h>
         void main()
```

strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase
strncat()	Concatenates portion of one string at the end of another string
strchr()	Returns pointer to the first occurrence of the character in a given string.
strstr()	Returns pointer to the first occurrence of the string in a given string.

```
         {
          char a[20]="Program";
          printf("Length of string=%d \n",strlen(a));
         }
```

   output: `Length of string=7`

2. strcpy():copies the content of one string to the content of another string. It takes two arguments.
   Syntax:
```
strcpy(destination,source);
     or
strcpy(s1,s2);
```
(source and destination are both the name of the string. This statement, copies the content of string source to the content of string destination or string s2 to string s1).

Example:
```
#include <stdio.h>
#include <string.h>
void main()
{
     char a[10],b[10];
     printf("Enter string:");
     gets(a);
     strcpy(b,a);        //Content of string a is copied to
     printf("Copied string: ");   //string b.
     puts(b);
}
```

Output:

```
Enter string: Hello World
Copied string: Hello World
```

3. strcat() concatenates(joins) two strings. It takes two arguments, i.e., two strings and resultant string is stored in the first string specified in the argument. As we know, each string in C is ended up with null character ('\0'). In strcat( ) operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strcat( ) operation.

   Syntax:

   ```
   strcat(first_string,second_string);
   ```

   Example:
   ```
   #include <stdio.h>
   #include <string.h>
   void main()
   {
          char str1[]="This is ", str2[]="a good place";
          strcat(str1,str2);   //concatenates str1 and str2 and
          puts(str1);     // resultant string is stored in str1
   }
   ```

Output

```
This is a good place
```

4. strcmp( ): it returns a value less than zero if the first is lexicographically (alphabetically) less than the second, a value greater than zero if the first is

lexicographically greater than the second or zero if the two strings are equal. The comparison is done by comparing the coded (ascii) value of the characters, character by character.

Syntax:

```
        temp_varaible=strcmp(string1,string2);
```

Example

```
        #include <stdio.h>
        #include <string.h>
        void main( )
        {
                char str1[ ] = "fresh" ;
                char str2[ ] = "refresh" ;
                int i, j, k ;
                i = strcmp ( str1, "fresh" ) ;
                j = strcmp ( str1, str2 ) ;
                k = strcmp ( str1, "f" ) ;
                if(strcmp(str1,str2)==0)
                        printf("Both strings are equal");
                else
                        printf("Strings are unequal");

                printf ( "\n%d %d %d", i, j, k ) ;

        }
```

Output:

```
Strings are unequal

0 -1 1
```

5. strlwr(): This function converts all the uppercase characters in that string to lowercase characters. The resultant from strlwr() is stored in the same string.

Syntax :

```
strlwr(string_name);
```

Example

```
        #include <stdio.h>
        #include <string.h>
        int main()
        {
                char str1[]="LOWer Case";
                puts(strlwr(str1));
        }
```

Output:

```
lower case
```

6. strupr(): This function converts all the lowercase characters in that string to uppercase characters. The resultant from strupr() is stored in the same string.

   Syntax

   ```
   strupr(string_name);
   ```

   Example

   ```
   #include <stdio.h>
   #include <string.h>
   void main()
   {
          char str1[]="UppEr Case";
           puts(strupr(str1));
   }
   ```

   Output:

   ```
   UPPER CASE
   ```

7. strncat( ): This function in C language concatenates (appends or joins) portion of one string at the end of another string.

   Syntax:

   ```
   strncat ( str2, str1, 3 );
   ```

   (First 3 characters of str1 is concatenated at the end of str2)

   ```
   strncat ( str1, str2, 3 );
   ```

   (First 3 characters of str2 is concatenated at the end of str1.)

   In strncat( ) operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strncat( ) operation.

   Example:

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

   void main( )
   {
    char source[ ] = " says hello";
   ```

```
         char target[ ]= " Amaan";

         printf ( "\nSource string = %s", source );
         printf ( "\nTarget string = %s", target );

         strncat ( target, source, 5 );

         printf ( "\nTarget string after strncat( ) = %s",
                                        target);

        }
```

Output:

```
        source string = says hello
        Target string = Amaan
        Target string after strcat( ) = Amaan says
```

8. The strchr() function returns a pointer to the first occurrence of the character c in the string s.

   Syntax:

   ```
   char *strchr(const char *str, int character);
   ```

   Example:

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

   void main ()
   {
   const char str[] = "hello.world.com";
   const char ch = '.';
   char *ret;

   ret = strchr(str, ch);

   printf("String after %c is : %s\n", ch, ret);

   }
   ```

   Output:

   ```
   String after . is : world.com
   ```

9. strstr( ): This function returns pointer to the first occurrence of the string in a given string.

   Syntax:

```
char *strstr(const char *str1, const char *str2);
```

Example:

```
#include <stdio.h>
#include <string.h>
void main()
{
const char str1[20] = "Harry Potter";
const char str2[10] = "Potter";
char *ret;
ret = strstr(str1, str2);
printf("The substring is: %s\n", ret);
}
```

Output:

```
The substring is: Potter
```

NO NEED TO WRITE THIS (INSIDE []) IN NOTE BOOK

[HEX and OCT numbers

In our number system we have digits ranging from 0 to 9.

86

this translate into 8 * 10 + 6 * 1.

Then

475

4 * 100 + 7 * 10 + 5 * 1

you'll see a pattern here,

note '^' is 'the power of'

9 = 9 * 10 ^ 0

86 = 8 * 10 ^ 1 + 6 * 10 ^ 0

475 = 4 * 10 ^ 2 + 7 * 10 ^ 1 + 5 * 10 ^ 0

So it from left to right, using the base (10) the power raises by 1 and you times that by the digit.

In oct we have 8 digits 0 - 7
and to translate octal numbers into decimal numbers

46 = 4 * 8 ^ 1 + 6 * 8 ^ 0
456 = 4 * 8 ^ 2 + 5 * 8 ^ 1 + 6 * 8 ^ 0

Then hex, we have the digits 0 - 9, with the additional digits from A - F

A - 10
B - 11
C - 12
D - 13
E - 14
F - 15

exactly the same procedure

47A = 4 * 16 ^ 2 + 7 * 16 ^ 1 + 10 * 16 ^ 0
AFD = 10 * 16 ^ 2 + 15 * 16 ^ 1 + 13 * 16 ^ 0

Now why do we use these number systems? Well, they easily translate into binary since they are a multiple of 2.

In octal, each digit will translate into 3 binary digits

0 - 000
1 - 001
2 - 010

3 - 011
4 - 100
5 - 101
6 - 110
7 - 111

463 - 100 110 011

142 - 001100010

and hex is the same accept each digit represents 4 binary digits

0 - 0000
1 - 0001
2 - 0010
3 - 0011
4 - 0100
5 - 0101
6 - 0110
7 - 0111
8 - 1000
9 - 1001
A - 1010
B - 1011
C - 1100
D - 1101
E - 1110
F - 1111

473 - 0100 0111 0011
6FA - 0110 1111 1010

So you see why it's very good to use hex and octal in programming, sometimes we aren't worried about the value of the decimal number, using bit wise operations, we are only interested in the locations of the 1's and 0's in a binary number

Octal we just have a 0 infront of the number

0345 - octal number 345
0153 - octal number 153

And in Hex we put 0x infront of the number

0x3F7 - hex number 3F7
0x2H7 - hex number 2H7

To have printf print out in hex and oct
you have the operators to do so

%0x - hex

%o – oct]

2. FUNCTIONS

Functions can be broadly categorized into two categories:

1. Inbuild or System defined or Library functions.
2. User Defined Functions

1. **Inbuild or System Defined or Library Functions:** These are the functions whose definition is already defined and stored in respective header files of C Library. These functions can be easily incorporated into programs by including the header files. Examples of this type of functions are printf(), scanf(), clrscr(), gets(), puts() etc. We have already seen how to use these type of library functions.

2. User Defined Functions: As there is no limit to the problems that can be solved by C language, the library cannot contain functions to solve all types of problems. C allows us to create functions to suit to our requirements. The functions, which are created by the user to perform the desired tsk, are called user-defined functions. We can design user-defined functions according to our need and there may be a number of user-defined functions in a program. For example, you can create a user defined function to calculate the net salary of and employee.
Advantages of User-Defined Functions
User defined functions provide the following benefits :

- **Redundancy :** Complex programs generally contain some piece of code at more than one location in a program. When we create a function of repeated code, the redundancy of code is reduced, size of program is reduced and it becomes easy to design and debug.
- **Universal Use :** Some functionality may be needed in more than one programs. There are some common tasks in programs of similar type. When we create a function and make it a part of the library, the function is universally available to every program.
- **Modularity :** Modularity is the process of dividing a big problem into multiple smaller problems. Modularity is the main benefit provided by functions. The

benefit of modularity is parallel development and easy understanding of programs.

- **Teamwork :** When we divide a problem into a number of functions, the coding can be done in parallel, by many people. This reduces the development time of program and increases the spirit of teamwork in developers.

Declaring a user defined function in C is very easy. Following is the syntax of declaring a function in C language:

```
Return_Type Function_Name(Parameter_List)
{
Statement Block;
return (expression)
}
```

Here is an explanation of each element:

- **Return Type:** Return type is the data type of the value returned by the function. void is used when the function does not return any value, int float are used when function returns integer of floating point values respectively.
- **Function Name**: Function name is the name of the function. The function name is required to differentiate a function from the other functions. Every function must have a unique name of its own in a program.
- **Parameter List:** Parameter List contains variables that carry information from the main function to this function. We can leave this Parameter list blank if we have no parameters that carry information to this function.
- **Body of Function**: Body of the function contains the sequence of statements that will be used only inside the function. It may contain its own variables and other statements of C language.
- **Return (expression):** This is used to send a value through the function to the main function. If the return type of function is void, you need not use this line. If it is int of float, you have to use this return expression

C allows us to define user-defined functions in various ways. The types of functions are different from each other on their nature of arguments and return type. We can define the functions in following ways:

1. Functions with no arguments and no return type.
2. Functions with no arguments and a return type.
3. Functions with arguments and with no return type.
4. Functions with arguments and with return type.

1. Functions with no arguments and no return type

These are the functions, which do not take any data item from calling function, in the form of parameters and do not return anything to calling function, after execution. As these functions do not contain any parameters, a pair of empty braces follows the name of the function. The following program describes the use of functions with no parameters and no return type.

```
/*Progam showing function with no arguments & no return type*/
#include<stdio.h>
#include<conio.h>
void main()
{
     void sum();
     printf("Control now going to sum function\n");
     sum();
     getch();
}
void sum()
{
     int a,b;    //local variables of function
     printf("Enter two numbers : ");
     scanf("%d %d",&a,&b);
     printf("Sum of entered numbers is %d",a+b);
}
```

Output :

```
Control now going to sum function
Enter two numbers : 5 10
Sum of entered numbers is 15
```

The main() function of this program initially prints the statement in it, then makes a call to the sum() function. sum() function declares two local variables a and b. These variables can only be used inside the function. The values are input by the user into these variables and the sum of these values is printed. sum() function does not return anything to main as it's return type is void.

2. Functions with no arguments and a return type

This type of functions does not take any value from the calling function. After execution, they return some value to the main function. The data type of returned value determines the return type of function. For example, if a function named "void func()" returns an integer value, its syntax of declaration would be : "int func();" . This means function "func" will return an integer value to the function from where it has been called. Following is a program that adds two numbers using return type:

```
/*Progam showing function with no arguments & a return type*/
#include<stdio.h>
#include<conio.h>
void main()
{
     int sum();        //function prototyping
     int result;
     clrscr();
     printf("Control now going to sum function\n");
     result=sum();        //call to function sum
```

```
               //result will contain the value returned by function
          printf("Sum of entered numbers is : %d\n",result);
          getch();
}
int sum()        //body of the function
{
          int a,b;     //local variables of function
          printf("Enter two numbers : ");
          scanf("%d %d",&a,&b);
          return(a+b);       //returning value to calling function
}
```

Output :

```
Control now going to sum function
Enter two numbers : 5 10
Sum of entered numbers is 15
```

The function inputs two numbers and returns the addition to main() function. In this example, no value is passed from the main() function to the sum function. Both of the variables are scanned inside the sum() function and the result after adding them is retuned to main() function, using return statement.

3. Functions with arguments and no return type

This type of functions take same value from the calling function (usually main()) to the called function. However, the functions of this type do not take any value from the called function to the calling function as we did in previous case. The values, which are passed with the function call, are called as "parameters" or "arguments". Syntax for passing arguments or parameters with function call is:

```
Return_Type Function_Name(Parameter1, ... ,ParameterN)
```

Following is a program that adds two numbers using parameters and the sum is displayed from the called function:

```
/*Progam showing function with no arguments & a return type*/
#include<stdio.h>
#include<conio.h>
void main()
{
          void sum(int ,int);       //function prototyping
          int a,b;
          clrscr();
          printf("\nEnter two numbers : ");
          scanf("%d %d",&a,&b);
          printf("Control now going to sum function\n");
          sum(a,b);    //call to function sum(), the values 'a' and 'b' are
                                                         passed to sum()
          getch();
}
void sum(int x,int y)       //body of the function
{
          int result;    //local variable of function
          result=x+y;    //the value of 'a' is stored in 'x' and 'b' in 'y'
```

```
     printf("Sum of entered numbers is : %d\n",result);
}
```

Output :

```
Enter two numbers : 5 10
Control now going to sum function
Sum of entered numbers is 15
```

In this program, the function sum() does not return any value to the main() function but, it takes two values from main() function in the form of parameters 'a' and 'b'. The values stored in argument variables are copied to the variable names listed in the function body i.e. 'x' and 'y'. The value of 'a' will go into 'x' and 'b' will go into 'y'. The calculation is performed with 'x' and 'y' and the result is assigned to the variable 'result'. The sum is printed using printf within the function sum().

4. **Functions with arguments and with return type**

This type of functions does not take any value from the calling function. After execution, they return some value to the main function. The data type of returned value determines the return type of function. For example, if a function named "void func()" returns an integer value, its syntax of declaration would be : "int func();" . This means function "func" will return an integer value to the function from where it has been called. Following is a program that adds two numbers using return type:

```
/*Progam showing function with arguments & with return type*/
#include<stdio.h>
#include<conio.h>
void main()
{
     int sum(int , int);       //function prototyping
     int a,b,answer;
     clrscr();
     printf("\nEnter two numbers : ");
     scanf("%d %d",&a,&b);
     printf("Control now going to sum function\n");
     answer=sum(a,b);       //call to function sum, the values 'a' and
                              'b' are passed to sum()
     printf("Sum of entered numbers is : %d\n",answer);
     getch();
}
int sum(int x,int y)       //body of the function
{
     int result;    //local variables of function
     result=x+y;
     return(result);       //returning value to calling function
}
```

Output:

```
Enter two numbers: 5 10
Control now going to sum function
Sum of entered numbers is 15
```

This program defines a function sum(), having integer return type. This function takes two arguments os integer type and returns an integer value after adding both the arguments. The returned value is used in the main() function. The main() function inputs two values from the use and stores them in the variable a & b. These values are passed to the function sum() as parameters. The values of these variables are copied to variables 'x' and 'y' and the result is calculated in a local variable 'result' of function sum(). This value is returned to main() function and the value gets stored in variable 'answer' of main() function. The sum is the printed using the printf statement and 'answer' variable.

Actual and Formal Parameters:

The variables used to pass information from calling function to the called function are called actual parameters and the variable created in the function body to hold the values of actual parameters are called formal parameters.

- Number of actual and formal parameters must be same.
- Can declare formal parameters with the same or different names as that or actual parameters.
- The data types and the order of declaration of formal and actual parameters must be the same.

Call by Value Method:

In call by value method a copy of actual arguments is created and passed to formal arguments in the function definition. Thus for each actual argument, a copy of value is created and passed to the formal argument. That is why it is called "Call by Value" method. With this method the changes made to the formal arguments in the called functions have no effect on the values of actual arguments in the calling function. Consider the following example in which arguments are passed by Call by Value:

```
#include<stdio.h>
void swap(int, int);
void main()
{
      int a=10, b=20;
      swap(a,b);
      printf("a = %d b = %d", a,b);
}
void swap(int x, int y)
{
      int t;
      t=x;
      x=y;
```

```
        y=t;
        printf("x = %d y = %d",x,y);
}

OUTPUT
x = 20 y = 10
a = 10 b = 20
```

A copy of actual parameters 'a' and 'b' is created and passed to formal parameters 'x' and 'y' in the function body. Note that values of 'a' and 'b' remain unchanged even after exchanging the values of 'x' and 'y'.

Call by Reference Method
In this method of calling a function, the address of actual arguments is passed, instead of creating the copy of actual arguments. The called function makes changes directly to the actual parameters and no copies of parameters are created.

```
#include<stdio.h>
void swap(int *, int *);
void main()
{
        int a=10, b=20;
        swap(&a,&b);
        printf("a = %d b = %d", a,b);
}
void swap(int *x, int *y)
{
        int t;
        t=*x;
        *x=*y;
        *y=t;
        printf("x = %d y = %d",x,y);
}

OUTPUT
x = 20 y = 10
a = 20 b = 10
```

This program manages to exchange the values of 'a' and 'b' using their addresses stored in 'x' and 'y'.

Why use Call by Reference??

Using this we can make a function return more than one value at a time. For example:

```
#include<stdio.h>
void circle(int, float *, float *);
void main()
{
        int r;
        float area,circum;
        printf("enter the radius of circle:");
        scanf("%d",&r);
        circle(r,&area,&circum);
        printf("Area = %f  Circumference = %f",area,circum);
}
void circle(int r, float *a, float *c)
{
        *a = 3.14*r*;
        *c = 2*3.14*r;
}
```

Here we are passing the' value' of radius, but 'address' of area and circumference and since we are passing the addresses, any change that we make in values stored at addresses contained in the variables 'a' and 'c' would make effective change at the main function.

Recursion

Recursion is a technique that involves a function calling itself from its body. Recursive function is a function that calls itself from its own body. The function keeps on calling itself till a particular condition holds true. This way, it creates a controlled chain of execution. Recursive functions should contain some kind of condition checking mechanism so that they should terminate after performing 'n' iterations.

```
/*Progam to calculate factorial using recursive function*/
#include<stdio.h>
void main()
{
     int factorial(int );       //function prototyping
     int num,result;
     printf("\nEnter a number : ");
     scanf("%d",&num);
     result=factorial(num);    /*call to function, value 'num'
                                is passed and value returned
                                by function will be stored in
                                'result'*/

     printf("\nFactorial is : %d",result);
     getch();
}
int factorial(int n)
{
     int ans;
     if( (n==0) || (n==1) )
       return(1);
     else
       ans = n*factorial(n-1);       //call to itself
     return(ans);
}
```

NO NEED TO WRITE THIS (INSIDE []) IN NOTE BOOK

[This program contains a recursive function named "factorial", which is having a return type as "int" and takes one parameter of integer type.

- First of all you entered a number and it is stored in variable 'num'.
- "result=factorial(num);" statement calls the function with 'num' as parameter and whatever integer value the function will return, it will be stored in variable 'result'.
- The value entered by user (num) comes in 'n'. Suppose you entered 5 so 'num' will be 5 and therefore 'n' will be 5.
- The first *if* condition is checked, 'n' is neither 0 nor 1, so the *else* loop would work.
- The statement "ans = n*fact(n-1);" would execute. It is actually to a call to itself. The variable 'n' is 5, so the statement becomes "**ans = 5*fact(4)**",
  which again calls the function with 'n' as 4. The *if* evaluates to false and *else* gets executed. fact(4) will be evaluated to 4*fact(3) and the statement now becomes "**ans = 5*4*fact(3)**",
  which again calls the function with 'n' as 3. The *if* evaluates to false and *else* gets executed. fact(3) will be evaluated to 3*fact(2) and the statement now becomes "**ans = 5*4*3*fact(2)**",
  which again calls the function with 'n' as 2. The *if* evaluates to false and *else* gets executed. fact(2) will be evaluated to 2*fact(1) and the statement now becomes "**ans = 5*4*3*2*fact(1)**",
  which again calls the function with 'n' as 1. Now the *if* condition is evaluated to True and it executes. "fact(1)" returns 1 and the statement now becomes "**ans = 5*4*3*2*1**".
- Therefore, 120 is returned to the calling function where it gets printed via printf() statement. Like this factorial of every number entered by user is calculated.]

3. STRUCTURES

The syntax of declaring a structure is as follows:

```
struct name
{
Data_type variable_name;
Data_type variable_name;
..........
..........
Data_type variable_name;
};
```

where, "struct" is the keyword to declare the structure, "name" is the name of the structure and the "variable_name" is the name of member variable and the "data_type" is the data type of the member variable.

The following lines of code declares a structure to store the details of an employee.

```
struct emp
{
int empno;
char name[10];
float salary;
int deptno;
};
```

All the members of the structure are declared as per the standard variable naming conventions of C language. Structures are generally defined before the main() body of the program.

Declaring Structure Variable

You can declare any number of structure variables, after you have defined the body of the structure. The syntax of defining a structure variable is same as the syntax of defining variables of built-in data types of C language. The only difference is that, here a "struct" keyword is used before the name of structure variable. The following line of code declares the variable, "e1" of "emp" structure.

```
struct emp e1;
```

The other method of declaring the structure variable is at the end of the structure body. The following code illustrates the method.

```
struct emp
{
int empno;
char name[10];
float salary;
int deptno;
}e1;
```

The variable, "e1", further contains 4 variables. Each of these four variables inside "e1" will store the empno, salary and deptno respectively. The members of structure are stored in contiguous memory locations in order they have been defined in the structure body.

Initialization

You can initialize the members of individual structure variable one by one or you can initialize all the members of a structure variable in a single statement. The following methods can be used to initialize the structure variable "e1", created in the example in previous topic.

1. In the first method of initialization, the structure variable is initialized by writing all the values of individual members of structure variables.

   ```
   struct emp e1={12,"Archit",1200,10};
   ```

2. The second method can be used to assign individual values to structure member variables in different steps. The following code illustrates this method.

   ```
   e1.empno=12;
   e1.ename="Archit";
   e1.salary=1500;
   e1.deptno=10;
   ```

Here, the individual elements of a structure variable are accessed with a dot operator (.). In the example, "e1" is the variable of structure type and "empno" is a member variable. The member variable is initialized by using a dot i.e. structure variable name.member variable name.

Accessing Structure Members

The structure members cannot be directly accessed in the expressions. They are accessed by using the name of the structure variable, followed by a dot and then the name of the member variable. This means, to access the employee number associated with structure variable "e1", e1.empno, will be used. The following program declares a structure to store the details of a student and prints the stored details.

```
#include<stdio.h>
#include<conio.h>
struct student
{
int rollno;
char name[10];
char course[5];
float fees;
};
void main()
{
     struct student s1;
     printf("\nEnter the rollno of the student : ");
     scanf("%d",&s1.rollno);
     printf("\nEnter the name of the student : ");
     scanf("%s",&s1.name);
     printf("\nEnter the course of the student : ");
     scanf("%s",&s1.course);
     printf("\nEnter the fees of the student : ");
     scanf("%f",&s1.fees);
     printf("\nThe rollno of student is : %d",s1.rollno);
     printf("\nThe name of student is : %s",s1.name);
     printf("\nThe course of student is : %s",s1.course);
     printf("\nThe fees of student is : %f",s1.fees);
     getch();
}
```

Output :

```
Enter the rollno of the student : 5
Enter the name of the student : Archit
Enter the course of the student : BTech
Enter the fees of the student : 20000
```

```
The rollno of student is : 5
The name of student is : Archit
The course of student is : BTech
The fees of student is : 20000
```

The above program declares a structure "student". It has member variables to store the roll number, name, course and fees of a student. A structure variable named "s1" has been declared to store the details of three different students. The member variable has been accessed by using the pattern of Structure_variable_name.Member_variable_name. The program stores and displays the details by using a variable.

Operations on Individual Members

You can perform all the arithmetic operations on the structure data members in similar way, as we did with normal C variables. The following program performs arithmetic operations on individual structure data members.

```
#include<stdio.h>
#include<conio.h>
struct student
{
int rollno;
char name[10];
float m1;
float m2;
float m3;
float total;
};
void main()
{
     struct student s;
     printf("\nEnter the rollno of the student : ");
     scanf("%d",&s.rollno);
     printf("\nEnter the name of the student : ");
     scanf("%s",&s.name);
     printf("\nEnter the marks in 3 subjects : ");
     scanf("%f %f %f",&s.m1,&s.m2,&s.m3);
     s.total=s.m1+s.m2+s.m3;
     printf("\nThe name is : %s",s.name);
     printf("\nThe total marks of student are : %f",s.total);
     getch();
}
```

Output :

```
Enter the rollno of the student : 5
Enter the name of the student : Archit
Enter the marks in 3 subjects : 70 80 90
The name is : Archit
The total marks of student are : 240.000000
```

In the above program, the three member variables of a structure type variable, have been added to the fourth member variable "total".

Array of structures

C language allows you to create an array of variables of structure type. The mechanism to define and access the array elements of the array of structure is same as that of accessing the members of in integer or character array. The following program illustrates the creation and usage of array of structures.

```
#include<stdio.h>
#include<conio.h>
struct student
{
int rollno;
char name[10];
float marks;
};
void main()
{
     struct student s[5];
     int i;
     for(i=0;i<5;i++)
     {
       printf("\nEnter the marks of student : ");
       scanf("%f",&s[i].marks);
       printf("\nEnter the name of the student : ");
       scanf("%s",&s[i].name);
       printf("\nEnter the rollno of the student : ");
       scanf("%d",&s[i].rollno);
       printf("\n");
     }
     for(i=0;i<5;i++)
     {
       printf("\nThe rollno of student is : %d",s[i].rollno);
       printf("\nThe marks of student is : %f",s[i].marks);
       printf("\nThe name of student is : %s",s[i].name);
       printf("\n");
     }
     getch();
}
```

Output :

```
Enter the marks of student : 65
Enter the name of the student : Ram
Enter the rollno of the student : 1

Enter the marks of student : 49
Enter the name of the student : Shyam
Enter the rollno of the student : 2

Enter the marks of student : 70
Enter the name of the student : Geeta
Enter the rollno of the student : 3
Enter the marks of student : 61
Enter the name of the student : Ravi
Enter the rollno of the student : 4

Enter the marks of student : 88
Enter the name of the student : Lakshay
Enter the rollno of the student : 5

The rollno of student is : 1
The marks of student is : 65.00000
The name of student is : Ram

The rollno of student is : 2
The marks of student is : 49.0000
The name of student is : Shyam

The rollno of student is : 3
The marks of student is : 70.0000
The name of student is : Geeta

The rollno of student is : 4
The marks of student is : 61.0000
The name of student is : Ravi

The rollno of student is : 5
The marks of student is : 88.0000
The name of student is : Lakshay
```

The above program declares an array of structures with a name "s". It is capable of storing 5 structure variables. The program uses a for loop to input and display the roll number, name and marks of 5 students and prints them back

Structure within a Structure

```
struct parent
{
Data_type variable_name;
Data_type variable_name;
    nested structure
    {
    Data_type variable_name;
    Data_type variable_name;
    }nested_structure_variable;
};
```

Example:

```
struct date
```

```
{
 int day;
 char month[20];
 int year;
}
struct stud
{
 int rno;
 char name[30];
 struct date bday;
};
main()
{struct stud s1;
 puts("input roll number,name,DOB of student");
 scanf("%d",&s1.rno);
 gets(s1.name);
 scanf("%d %s %d",&s1.bday.day,&s1.bday.month,&s1.bday.year);
 printf(" %d %s %d %s %d",
s1.rno,s1.name,s1.bday.date,s1.bday.month.,s1.bday.year);
}
```

4. UNION

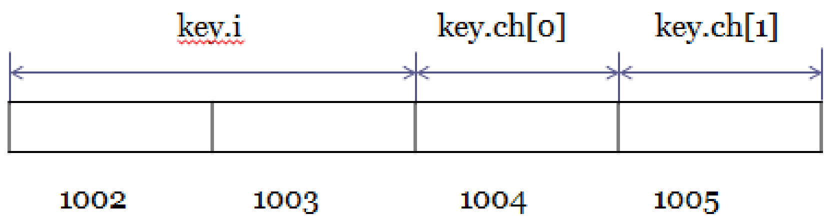
- In structure, all the data members have separate storage locations in the memory.
- In union, a single large sized place of memory equals to the size of largest element of union.

- All the members of a union use the same memory location.
- At one time, the memory location is treated as a variable of one type & at the other time, the memory location is treated as a variable of other type.

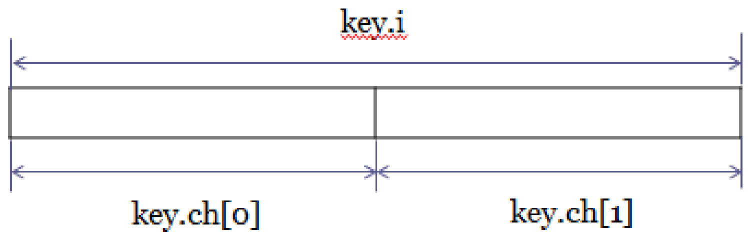
Example:

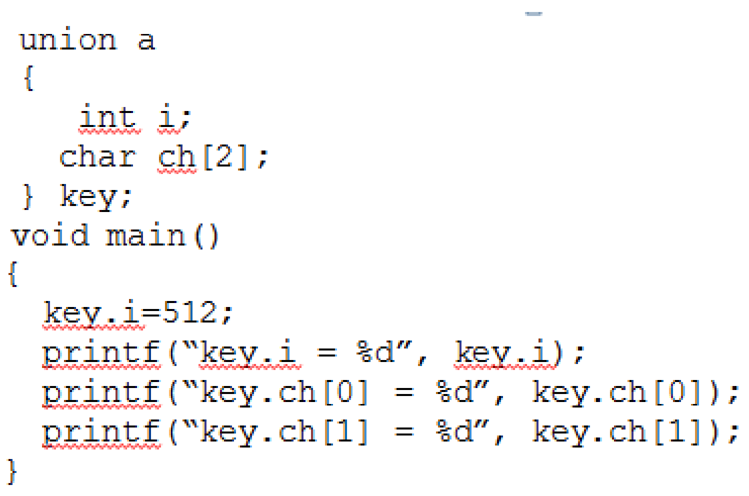
```
union a
{
 int i;
  char ch[2];
} key;
```

In structure:



In union:





```
 union a
 {
    int i;
   char ch[2];
 } key;
void main()
{
  key.i=512;
  printf("key.i = %d", key.i);
  printf("key.ch[0] = %d", key.ch[0]);
  printf("key.ch[1] = %d", key.ch[1]);
}
```

Example:

If i want a structure to store information about employees like:
Name
Grade
Age
if grade = HSK (highly skilled)
        hobby name
        credit card num
if grade = SSK (semi skilled)
        vehicle num
        debit card num
Here, though there is no wrong in using structure, some fields are not used. For any employee, depending upon his grade either the fields hobby and credit card number or the fields vehicle number and debit card number would get used. Both sets of field would never get used. This would lead to wastage of memory with every structure variable that we create, since structure would have all four fields apart from name, grade and age. This can be avoided by creating a union between these sets of fields. This is shown below:

```
sruct info1
{
 char hobby[20];
 int ccnum;
};
struct info2
{
char vehnum[10];
 int dbnum;
};
union info
{
 struct info1 a;
 struct info2 b;
};
```

Another program using Union

```
struct emp
{
 char n[30];
 char grade[4];
 int age;
} emp1;
union student
{
int rollno;
char name[10];
char course[5];
float fees;
};
void main()
{
     struct student s1;
     printf("\nEnter the rollno of the student : ");
     scanf("%d",&s1.rollno);
     printf("\nEnter the name of the student : ");
     scanf("%s",&s1.name);
     printf("\nEnter the course of the student : ");
     scanf("%s",&s1.course);
     printf("\nEnter the fees of the student : ");
     scanf("%f",&s1.fees);
     printf("\nThe rollno of student is : %d",s1.rollno);
     printf("\nThe name of student is : %s",s1.name);
     printf("\nThe course of student is : %s",s1.course);
     printf("\nThe fees of student is : %f",s1.fees);
     getch();
}
```

5. ENUM- Enumerated Data Type

Enumerated Types are a special way of creating your own Type in C. It is a user defined data type. The type is a "list of key words". Enumerated types are used to make a program clearer to the reader/maintainer of the program. Enumeration data type consists of named integer constants as a list.It start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list.

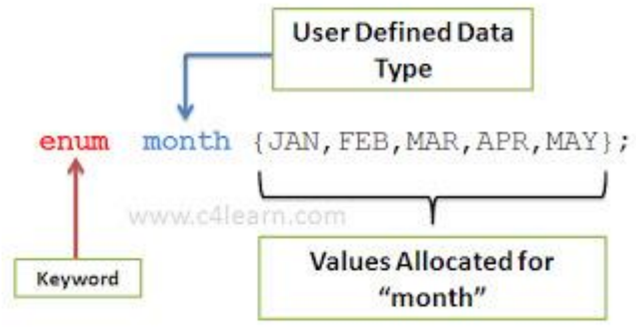
Enum syntax in C:

```
enum tag_name { member1,member2…. }variable1,
                                  variable2..;
```

Enum example in C:

```
enum month { Jan, Feb, Mar }; or
/* Jan, Feb and Mar variables will be assigned to 0, 1 and 2 respectively by default */
enum month { Jan = 1, Feb, Mar };
/* Feb and Mar variables will be assigned to 2 and 3 respectively by default */
enum month { Jan = 20, Feb, Mar };
```

/* Jan is assigned to 20. Feb and Mar variables will be assigned to 21 and 22 respectively by default */


User Defined Data Type
enum month {JAN,FEB,MAR,APR,MAY};
www.c4learn.com
Keyword
Values Allocated for "month"

Another example:

```
enum suit{
    club=0;
    diamonds=10;
    hearts=20;
    spades=3;
};

enum boolean{
    false;
    true;
};
enum boolean check;



#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday,
saturday};
void main()
{
    enum week today;
    today=wednesday;
    printf("%d day",today+1);
}
```

Output: 4 day