**Day2(24-11-2016)- Anil**

**DATA TYPES:**

To describe type of variable will use the data type.

The Basic data types in C:-

**1. char - Character data type:**

Character types are used to store characters value.

In C programming, keyword *char* is used for declaring Character variable.For example:

i) char var;

The size of character data type in C is: 1Byte

## **2. int - Integer data type:**

Integers are whole numbers that can have both positive and negative values: 0, -5, 10

In C programming, keyword *int* is used for declaring integer variable. For example:

i) int v;

ii) int n;

The size of int is either 2 bytes or 4 bytes. It depends upon compiller and os:

|  |  |  |
| --- | --- | --- |
| complier | os | int-sige |
| 16 | 16 | 2 bytes |
| 16 | 32 | 2 bytes |
| 32 | 16 | 2 bytes |
| 32 | 32 | 4 b ytes |

## 3. float - Floating type:

Floating types are used to store real numbers. such as: 2.34, -9.382, 5.0 etc. You can declare a floating point variable in C by using *float.* For example:

i) float accountBalance;

In C, floating values can be represented in exponential form as well. For example:

ii) float normalizationFactor = 22.442e2;

The size of float data type in C is: 4 Bytes

**4. double- Double type:**

double data types are used to sote real numbers in double precession.

In C programming, keyword *double* is used for declaring Double variable. for example:

i) double bookPrice;

The size of float data type in C is: 4 Bytes

**Size Qualifiers:**

There are two size qualifiers, *long* and *short*. For example:

i) long int i;

ii) short int i;

The size of int varibale is 4 bytes,However if long int then the size is 8 bytes or else if short int the size is 2 bytes.

### **Sign qualifiers:**

There are two sign qualifiers, *unsigned* and *signed*.The default one is signed. For Example

i) unsigned int v;

Unsigned variables should starts with 0 only,it will not store -ve numbers.

*Signed variable* can hold both negative and positive.

Range of unsigned variable is ---> 0 to 2n-1

Range of signed variables is ------> -2 n-1 to 2n-1-1

Char – unsigned -------------------> 0 to 28-1 ==== 0 to 255

signed -----------------------> -28-1 to 28-1-1 === -128 to 127

**ASCII:**

ASCII stands for American Standard Code for Information Interchange.

To get more ascii. Type, man ascii in linux

**Operators:**

An operator is a symbol that tells the compiler to perform specific mathematical or logical function.

There are 10 operators in C:

i) Arithmetic Operators

ii) Assignment Operators

iii) Increment/ Decrement Operators

iv) Relational Operators

v) Logical Operators

vi) Ternary or Conditional

vii) Size of Operators

viii) comma Operators

ix) implecite/ explecite Operators

x) Bitwise Operators.

**1. Arithemetic operators:**

Arithemetic operators used to perform the mathematical calculations like addition, subtraction, multiplication, division and modulus. For example

int i;

int v;

int c;

i = 10;

v = 20;

c= i+v

c= i\*v

c = i/v

printf(“%d\n”,c)

**2. Assignment operators:**

These are used to assign the values for the variables. For example

int v1,v2;

int res;

v1 = 10;

v2 = 3;

res = v1/v2;

printf(“%d\n”,res)

**3. Increment / Decrement operators:**

These operators are used to either increase or decrease the value of the variable by one.

int A = 0;

int b = ++A;

int c = --A

printf(“%d\n”,c)

**4. Relational operators:**

These operators are used to compare the value of two variables.

int v1,v2;

int res;

v1 = 10;

v2 = 5;

res = (v1<v2);

printf(“%d\n”, res);

**5. Logical operators:**

These operators are used to perform logical operations on the given two variables.

int v1,v2, v3;

int res;

v1 = 10;

v2 = 5;

v3 = 20;

res = (v1<v2)&&(v2<v3);

printf(“%d\n”,res);

**6. Ternary / Conditional operators:**

Conditional operators return one value if condition is true and returns another value is condition is false.

int v1,v2;

int res;

v1 = 10;

v2= 20;

res = (v1>v2)?v1:v2;

printf(“%d\n”,res)

**7. Size of operators:**

The sizeof operator which returns the size of data.

int v1;

printf(“%d\n”,sizeof(v1))

**8. Comma operators:**

Comma operators are used to link related expressions together

int v1,v2 res;

res = (v1=10) , (v2 = 20), (v1+v2)

printf(“%d\n”,sizeof(v1))

Day 3(25-11-2016) - Anil

**Statement** : A statement is the smallest element that expresses some action to be carried out.

Statements are of 3 types.

1)Simple statement:consists of one statement.

2)compound statement:consists of 2 or more statements.

3)Null statement:consists of 0 statement.

Ex: (v1<v2)----->this is expression

res= (v1<v2)----->this is statement

**9.Implicit and Explicit operators:**

**Implicit:**Evaluation of expression will take the default datatype.

Ex:If we doing some division operation the default datatype is

int/int = int

int/float = float

float/int = float

float/float = float

**Explicit:**To overwrite the default datatype of implicit functionality.

**Ex**: int v1,v2;

float v3;

v1 =10;

v2 = 3;

v3 = (float)v1/v2

printf(“%f”,v3)

**10.Bitwise operators:**Bit-wise operators are used to perform operations on bit-level.

1)Bitwise-OR-------> |

2)Bitwise-AND-----> &

3)Bitwise-XOR------> ^

4)Bitwise-negation----> ~

5)leftshift----------------> <<

6)Rightshift-------------> >>

**AND Truth Table:**

a b output

0 0 0

0 1 0

1 0 0

1 1 1

**OR Truth Table:**

a b output

0 0 0

0 1 1

1 0 1

1 1 1

**XOR Truth Table:**

a b output

0 0 0

0 1 1

1 0 1

1 1 0

**Negation Truth Table:**

a output

0 1

1 0

OR Example:

a = 0xAB79

b = 0x979C

0xAB79 1010 1011 0111 1001

0x979C 1001 0111 1001 1100

o/p 0xBFFD 1011 1111 1111 1101

AND Example:

a = 0xAB79

b = 0x979C

0xAB79 1010 1011 0111 1001

0x979C 1001 0111 1001 1100

o/p 0x8318 1000 0011 0001 1000

XOR Example:

a = 0xAB79

b = 0x979C

0xAB79 1010 1011 0111 1001

0x979C 1001 0111 1001 1100

o/p 0x3CE5 0011 1100 1110 0101

Negation Example:

a = 0xAB79

i/p 0xAB79 1010 1011 0111 1001

o/p 0x5486 0101 0100 1000 0110

Left-Shift:Shifting the bits ntimes to the left

a= 0xAB79<<1

0xAB79 1010 1011 0111 1001

o/p 0x56F2 0101 0110 1111 0010

Right-Shift:Shifting the bits ntimes to the right

a= 0xAB79>>1

i/p 0xAB79 1010 1011 0111 1001

o/p 0x55BC 0101 0101 1011 1100

**30/NOV/2016 - Namrata**

**CLASS-4**

**Priority And Association**

priority – different precedence.

Association – same precedence (same priority operation).

The operators precedence are :-

. ,(),[],->

unary operator ( ++ , -- , +, - sizeof, !)

\* / %

+ -

<< >>

< <= > >=

== !=

&

^

|

&&

| |

?

\*= += -= /=

,

Unary , assignment and comma operators has right to left associativity.

**Conditional Statement.**

1. if

2. if else

3. nested if

4. nested if else (else if ladder)

5. switch case.

If condition – To check the condition.

Eg :-

int number;

if ( number > 0)

{

printf(“Number is greater”);

printf(“Number is positive also”);

} // we need braces for compound statment in if block

If else – to check the two conditions, if first is not true then else block will excute.

int number;

if ( number > 0)

{

printf(“Number is greater”);

}

else

printf(“Number is not valid”);

Nested if – to check miltiple conditions.

int number;

if ( number > =0)

{

printf(“Number is greater”);

if (number == 0 )

{

printf(“Number is zero”);

}

if (number > 0)

{

printf(“Number is positive”);

}

}

Nested if else -

int number;

if ( number > =0)

{

printf(“Number is greater”);

if (number == 0 )

{

printf(“Number is zero”);

}

else {

printf(“Number is positive”);

}

}

Switch Case – To excute particular block of statment based on choice.

Eg

int number;

switch (number)

{

case 0 :

printf (“number is zero ”);

break; // break is mandotary for each block

case 1:

printf (“number is postive”);

break;

default :

printf (“the number is %d”, number);

}

// case switches work only for character and integers

//Default statment is optional.

//multiplse cases can also be possible in a single switch case.

**Class-5**

**Loops**

Loops are useful for excuting a particular block for number of times.

1. for(intialization; condition; conditional updater) // use when number of ittration are known.

2. while (condition) // use when number of iteratuion are unknown.

The worst case execution is 0.

3. do while (condition with semicolon at the end).

The worst case execution is 1.

**examples :-**

1 for //

int number = 20 , counter = 0;

for (counter = 0; counter < number ; counter ++ )

{

printf(“%d”, counter);

}

2. while

int number;

while (number <10)

{

printf(“%d number \n”, number);

number ++;

}

3. do while

int number = 0;

do

{

printf(“%d”,number);

number ++;

}while (number <10);

printf(“%d”,number);

01/Dec/2016-Santhosh

**Class-6**

**Loop Control Statements:**

It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

In such cases, Break and continue statements are used.

* Break
  + Break statement is used to terminate the while loops, switch case loops and for loops from the subsequent execution.
  + Syntax: break;
* Continue
  + Continue statement is used to continue the next iteration of for loop, while loop and do-while loops. So, the remaining statements are skipped within the loop for that particular iteration.
  + Syntax : continue;

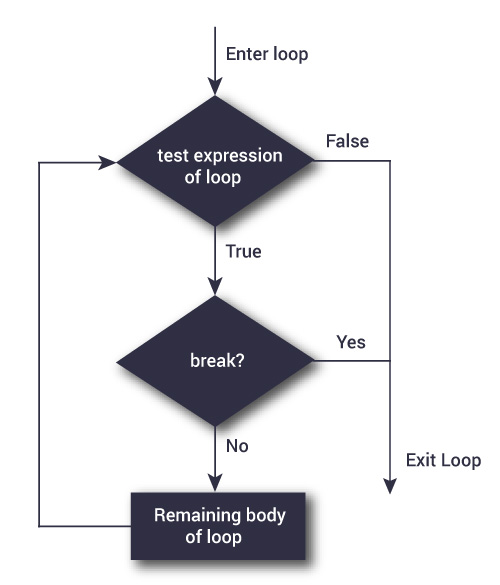
**Break Statement:**

The **break** statement in C programming has the following usages −

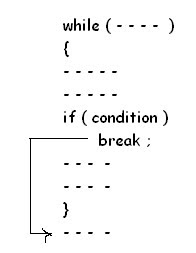
* When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

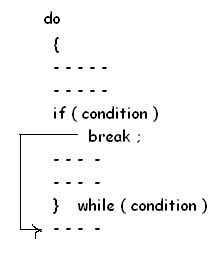
**Flow Diagram:**



Break statement in while:



Break statement in do-while:



**Example:**

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* while loop execution \*/

while( a < 20 ) {

printf("value of a: %d\n", a);

a++;

if( a > 15) {

/\* terminate the loop using break statement \*/

break;

}

}

return 0;

}

The above program produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

**Class-7**

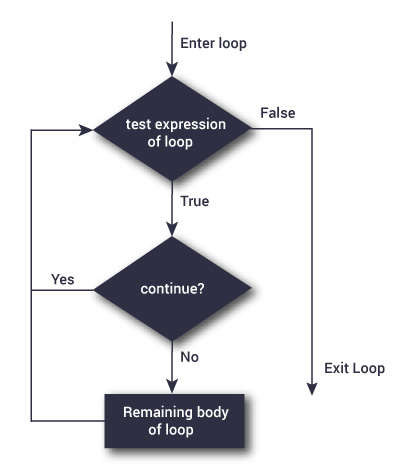
**Continue Statements:**

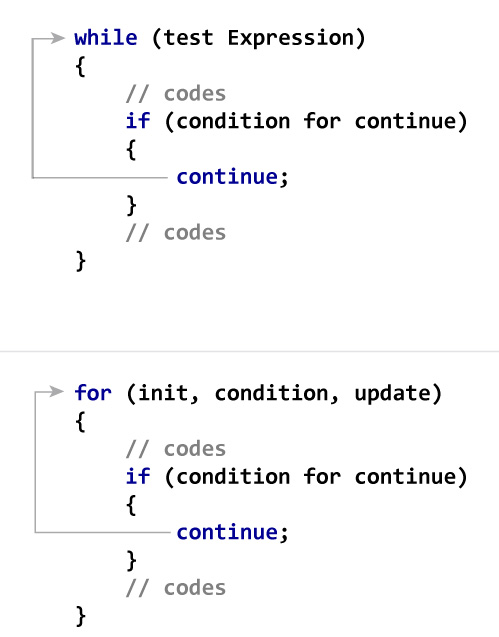
The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else.

**Note :**

* It is used for Skipping part of Loop.
* Continue causes the remaining code inside a loop block to be skipped and causes execution to jump to the top of the loop block

**Flowchart:**





**Example:**

// Program to calculate sum of maximum of 10 numbers

// Negative numbers are skipped from calculation

# include <stdio.h>

int main()

{

int i;

double number, sum = 0.0;

for(i=1; i <= 10; ++i)

{

printf("Enter a n%d: ",i);

scanf("%lf",&number);

// If user enters negative number, loop is terminated

if(number < 0.0)

{

continue;

}

sum += number; // sum = sum + number;

}

printf("Sum = %.2lf",sum);

return 0;

}

**Output**

Enter a n1: 1.1

Enter a n2: 2.2

Enter a n3: 5.5

Enter a n4: 4.4

Enter a n5: -3.4

Enter a n6: -45.5

Enter a n7: 34.5

Enter a n8: -4.2

Enter a n9: -1000

Enter a n10: 12

Sum = 59.70

In the program, when the user enters positive number, the sum is calculated using sum += number; statement.

When the user enters negative number, the continue statement is executed and skips the negative number from calculation.

**Nested Loops:**

A loop inside another loop is called a nested loop. The depth of nested loop depends on the complexity of a problem. We can have any number of nested loops as required.

Types Of Nested Loops:

1.Nested for loop

2.Nested while loop

3.Nested do-while loop

4. Mixed Nested loop

## Nested while loop

A while loop inside another while loop is called nested while loop.

### Syntax of Nested while loop

while (condition1)

{

statement(s);

while (condition2)

{

statement(s);

... ... ...

}

... ... ...

}

## Nested do-while loop

A do-while loop inside another do-while loop is called nested do-while loop.

### Syntax of Nested do-while loop

do

{

statement(s);

do

{

statement(s);

... ... ...

}while (condition2);

... ... ...

}while (condition1);

## Nested for loop

A for loop inside another for loop is called nested for loop.

### Syntax of Nested for loop

for (initialization; condition; increment/decrement)

{

statement(s);

for (initialization; condition; increment/decrement)

{

statement(s);

... ... ...

}

... ... ...

}

**Mixed Nested Loop:**

**1. for – while**

for (initialization; condition; increment/decrement)

{

statement(s);

while (condition2)

{

statement(s);

... ... ...

}

... ... ...

}

**2. for – do-while**

for (initialization; condition; increment/decrement)

{

statement(s);

do

{

statement(s);

... ... ...

}while (condition2);

... ... ...

}

**3. while – for**

while (condition1)

{

statement(s);

for (initialization; condition; increment/decrement)

{

statement(s);

... ... ...

}

... ... ...

}

**4. while – do-while**

while (condition1)

{

statement(s);

do

{

statement(s);

... ... ...

}while (condition2);

... ... ...

}

**5. do-while – for**

do

{

statement(s);

for (initialization; condition; increment/decrement)

{

statement(s);

... ... ...

}

... ... ...

}while (condition1);

**6. do-while – while**

do

{

statement(s);

while (condition2)

{

statement(s);

... ... ...

}

... ... ...

}while (condition1);

06/Dec/2016 - Divya

**Class 8**

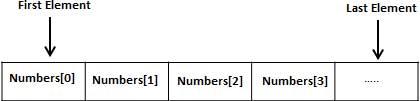
### Arrays

Array is a collection of variables belonging to the same data type.( i.e. storing group of data of same data type in an array). The arrays are reffered to as structured data types. An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

* Array might be belonging to any of the data types
* Array size must be a constant value.
* Adjacent memory locations are used to store array elements in memory.

Instead of declaring individual variables, as number0, number1, ..., and number99. We can declare one array variable, as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

The lowest address corresponds to the first element and the highest address to the last element.



**Array declaration syntax:**  
data\_type arr\_name [arr\_size];

**Array initialization syntax:**  
data\_type arr\_name [arr\_size]=(value1, value2, value3,….);

**Array accessing syntax:**  
arr\_name[index];

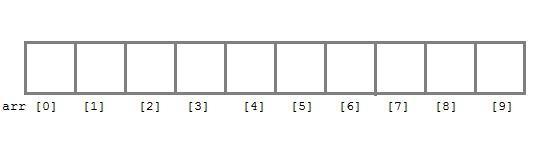
#### Declaring an Array

Like any other variable, arrays must be declared before they are used. General form of array declaration is,

data-type variable-name[size]; //Allocating memory and filling at later point of time when required

Example :

* char b[10]; // character array i.e. string
* int arr[10]; // integer array



Here **int** is the data type, **arr** is the name of the array and 10 is the number elements of an array. It means array **arr** can only contain 10 elements of **int** type.

**Index** of an array starts from 0 to size-1

**i.e.** first element of **arr** array will be stored at arr[0] address and last element will occupy arr[9].

#### Initialization of an Array

After an array is declared it must be initialized. Otherwise, it will contain **garbage** value(any random value).

An array can be initialized at either **compile time** or at **runtime**.

#### i) Compile time Array initialization

Compile time initializtion of array elements is same as ordinary variable initialization. The general form of initialization of array is,

*data-type* **array-name[size] = { list of values };** //filling values during memory allocation

int marks[4]={ 67, 87, 56, 77 }; *//integer array initialization*

float area[5]={ 23.4, 6.8, 5.5 }; *//float array initialization*

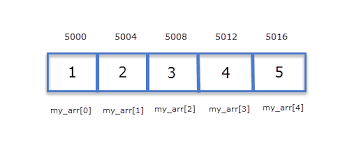
int marks[4]={ 67, 87, 56, 77, 59 }; //Compile time error

int arr[5]={0}; //To initialize all array elements to 0

One important things to remember is that when you will give more initializer than declared array size than the **compiler** will give an error.

Memory representation of an array:

Example : int my\_arr[5] = {1, 2, 3, 4, 5}



**NOTE: Size of an array = total elements \* size of datatype**

sizeof(my\_arr) = 5 \* sizeof(int) => 5\*4 => 20 bytes

Example:

#include<stdio.h>

void main()

{

int i;

// int arr1[]; //Compile time error, as arr1[] is declared and not filled even at later point of time

int arr[]={2, 3, 4}; //Compile time array initialization

printf("size of array: %lu bytes\n", sizeof(arr));

for(i=0 ; i<3 ; i++)

{

printf("arr[%d] is %d \t", i, arr[i]); //Accessing array elements

printf("address is %p \n", &arr[i]); //Accessing address of the array elements

}

printf("\n");

}

Output:

size of array: 12 bytes

arr[0] is 2 address is 0x7fffabb3a220

arr[1] is 3 address is 0x7fffabb3a224

arr[2] is 4 address is 0x7fffabb3a228

#### ii) Runtime Array initialization

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large array, or to initialize array with user specified values.

Example:

#include<stdio.h>

void main()

{

int arr[4];

int i, j;

printf("Enter array elements: ");

for(i=0;i<4;i++)

scanf("%d",&arr[i]); //Run time array initialization

printf("Given array elements are: ");

for(j=0;j<4;j++)

printf("%d\n",arr[j]); //Accessing array elements

}

Output:

Enter array elements:

1

2

3

4

Given array elements are:

1

2

3

4

1-Dimentional Arrays are used to store list of numbers or characters etc.

**13-12-2016:**

**vamsi krishna reddy**

**Multidimensional Array**

**Initializing Two-Dimensional Arrays:**

Multidimensional arrays may be initialized by specifying bracketed values for each row.

Following is an array with 3 rows and each row has 4 columns.

int a[3][4] = {

{0, 1, 2, 3} ,

{4, 5, 6, 7} ,

{8, 9, 10, 11}

};

The following initialization is equivalent to the previous example −

int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

**Accessing Two-Dimensional Array Elements:**

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array.

For example −

int val = a[2][3];

The above statement will take the 4th element from the 3rd row of the array.

**Let us check the following program.**

#include <stdio.h>

int main ()

{

/\* an array with 5 rows and 2 columns\*/

int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

int i, j;

/\* output each array element's value \*/

for ( i = 0; i < 5; i++ )

for ( j = 0; j < 2; j++ )

printf("a[%d][%d] = %d\n", i,j, a[i][j] );

}

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

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

#### Things which you must consider while initializing 2D array:

You must remember that when we give values during one dimensional array declaration, we don’t need to mention dimension. But that’s not the case with 2D array; you must specify the second dimension even if you are giving values during the declaration. Let’s understand this with the help of few examples –

/\* Valid declaration\*/

int abc[2][2] = {1, 2, 3 ,4 };

/\* Valid declaration\*/

int abc[][2] = {1, 2, 3 ,4 };

/\* Invalid declaration – you must specify second dimension\*/

int abc[][] = {1, 2, 3 ,4 };

/\* Invalid because of the same reason mentioned above\*/

int abc[2][] = {1, 2, 3 ,4 };

**Memory Represenation :**

Memory will be stored in continuesly.

\* if array type is int so each element would use 4 bytes.

\* if array type is char so each element would use 1 byte.

**14-12-2016**

**vamsi krishna reddy**

**STRINGS**

**STRINGS:**

Strings are actually one-dimensional array of characters terminated by a null character

'\0'.

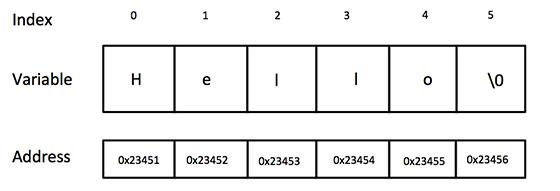
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C:



Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

**Ex Program:**

#include <stdio.h>

int main ()

{

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

printf("Greeting message: %s\n", greeting );

return 0;

}

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

Greeting message: Hello.

**C supports a wide range of string library functions :**

|  |  |
| --- | --- |
| S.N. | Function & Purpose |
| 1 | strcpy(s1, s2);  Copies string s2 into string s1. |
| 2 | strcat(s1, s2);  Concatenates string s2 onto the end of string s1. |
| 3 | strlen(s1);  Returns the length of string s1. |
| 4 | strcmp(s1, s2);  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |

**Note:** if we are using this functions in our program we must include below header file.

#include<string.h>

**Ex Program on Above functions:**

#include <stdio.h>

#include <string.h>

int main ()

{

char str1[20] = "votary";

char str2[10] = "tech";

char str3[10];

int len ;

/\* copy str1 into str3 \*/

strcpy(str3, str1);

printf("%s\n", str3 ); ---------------------- -----------------> o/p : votary.

/\* concatenates str1 and str2 \*/

strcat( str1, str2);

printf("%s\n", str1); ------------------------- --------------> o/p : votarytech.

/\* total lenghth of str1 after concatenation \*/

len = strlen(str1);

printf("strlen(“%d\n", len ); ------------------- --------------->o/p : 10.

return 0;

}.

15 – 12 – 2016

JUTURI SAI KRISHNA

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **FUNCTIONS** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Function** is a set of statements which is meant to accomplish a specific task.

**Types of Functions :**

1) Predefined functions

2) User defined functions :

I/P -> input O/P -> output

a) No I/P, No O/P

b) No I/P, with O/P

c) with I/P, No, with O/P

d) with I/P, with O/P

**Functions has**:

\* proto type (or) declaration

\* definition

\* declaration

To use a function we have to call it.

**1) No input, No output example**

void fun\_ni\_no(void); // Function declaration

int main(void)

{

fun\_ni\_no(); //Function call

return 0;

}

void fun\_ni\_no( void ) // No arguments so no input, No return type so no output

{

printf(“Say Hi\n”);

} // Function definition

**2) No input,with output example**

int ret\_0to100(void);

/\* this function returns values ranging from 0 to 100 without expecting any arguments\*/

int main(void)

{

int retval;

retval = ret\_0t0100();  
 printf (“You got %d\n”, retval);

return 0;

}

int ret\_0to100(void)

{

int limit = 100 +1 ; //As we want numbers upto 100

return (rand() % limit);

}

**NOTE : main is a user defined function declared by compiler and called by OS**

**................................ continued in next page**

**3) With input,no output example**

void print\_fibo(int);

int main(void)

{

int count;

printf(“enter how many numbers you want to print in a fibonacci series\n”);

scanf(“%d”,&count);

print\_fibo(count);

}

void print\_fibo(int num)

{

/\* We have to take care for validation of input \*/

/\* The validation can be done with in the function or outside of the function before function Function Call \*/

/\* Now in this function validation is done inside the function \*/

if ( num <= 0 ) {

return;

}

int res,first,next;

res = first = 0, next = 1;

while ( num-- ) {

printf(“%d ”,res);

res = first + next;

first = next;

next = res;

}

printf(“\n”);

}

............................... continued in next page

**4) With input,With output example**

int add\_nums(int,int);

int main(void)

{

int num1,num2,res;

printf(“enter any 2 numbers to get their sum\n”);

scanf(“%d %d”,&num1,&num2);

res = add\_nums(num1,num2);

printf(“sum is %d \n”,res);

}

int add\_nums ( int var1, int var2 )

{

int cal\_res;

cal\_res = var1 + var2 ;

return cal\_res;

}

**Storage classes**

Praveen B

Emp.No:- 817

Date: 16 December 2016

* Storage class define the scope(visibilty) and life time of variables within a cprogram or file.

Syntax for variable defination:-

**storage\_class\_keyword datatype variable\_name;**

We have four different storage classes which are:

1.Automatic storage class

2.Register storage class

3.Static storage class

4.External storage class

1. Automatic storage class

a. This is default storage class means All variables declared are of type Auto by default.

b. To define variable in Explicitly we use “auto” keyword.

c. Scope of variable is local to block and default value is garbage.

d. life time of variable is local to block.

example:- **auto int v1;**

void main()

{

auto int v1=1;

{

int v1=2;

printf(“V1=%d\n”,v1);

}

printf(“V1=%d\n”,v1);

return 0;

}

output:- V1=1

V1=2

2. Register storage class

a. To define variable use “register” keyword.

b. register variable are stored in register instead of **RAM** and life and scope are same as auto variable.

c. This is generally used for **faster access**.

d. unary operator [&] is not associated with it because Value is not stored in RAM instead

it is stored in Register.

Example:-  **register int v1;**

3. Static storage class

a.To define variable use “static” keyword.

b. local and global static variables are different and default value is zero.

c. Scope of variable is local to block or function.

d. life time of variable is value will be persist between different fucntion calls.

Example:-  **static int v1;**

void main()

{

hi();

hi();

hi();

}

void hi()

{

static int v1=5;

int v2=5;

v1++;

v2++;

printf(“V1=%d\tV2=%d\n”,v1,v2);

}

output:-V1=6 V2=6

V1=7 V2=6

V1=8 V2=6

4. External storage class

a. To define variable use “extern” keyword.

b. External variables are define again in the function and default value is zero..

c. Scope of variable is total program.

d. life time of variable is as long as the programs execution doesn't come to end.

Example:-  **extern int v1;**

V1=1

V1=2

2. Register storage class

a. To define variable use “register” keyword.

b. register variable are stored in register instead of **RAM** and life and scope are same as auto variable.

c. This is generally used for **faster access**.

d. unary operator [&] is not associated with it because Value is not stored in RAM instead

it is stored in Register.

Example:-  **register int v1;**

3. Static storage class

a.To define variable use “static” keyword.

b. local and global static variables are different and default value is zero.

c. Scope of variable is local to block or function.

d. life time of variable is value will be persist between different fucntion calls.

Example:-  **static int v1;**

void main()

{

hi();

hi();

hi();

}

void hi()

{

static int v1=5;

int v2=5;

v1++;

v2++;

printf(“V1=%d\tV2=%d\n”,v1,v2);

}

output:-V1=6 V2=6

V1=7 V2=6

V1=8 V2=6

4. External storage class

a. To define variable use “extern” keyword.

b. External variables are define again in the function and default value is zero..

c. Scope of variable is total program.

d. life time of variable is as long as the programs execution doesn't come to end.

Example:-  **extern int v1;**

**RECURSIVE FUNCTIONS**

--G.Divya Preethika

EMP.ID.-814

Date:19 December2016

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

void recurse()

{

... .. ...

recurse();

... .. ...

}

int main()

{

... .. ...

recurse();

... .. ...

}

A recursive function must consists of two main things.

1. Base or terminating condition

2. Iterative function

### How recursion works?



void recurse()

{

... .. ...

recurse();

... .. ...

}



int main()

{

... .. ...

recurse();

... .. ...

}

The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, [i](https://www.programiz.com/c-programming/c-if-else-statement)f...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

### Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow. Instead, you can use [loop](https://www.programiz.com/c-programming/c-for-loop).

**Factorial of a number using recursive functions:**

int fact(int num)

{

int res;

if((num == 0) || (num == 1))

return 1;

res = num\*fact(num-1);

return res;

}

int main()

{

int num = 5, res;

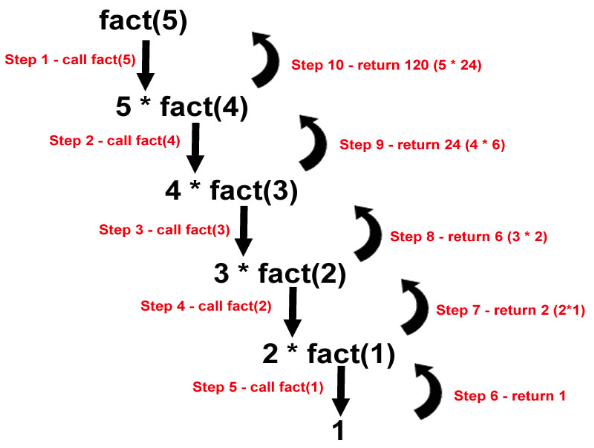
res = fact(num);

printf(“Factorial of %d is %d\n”, num,res);

}

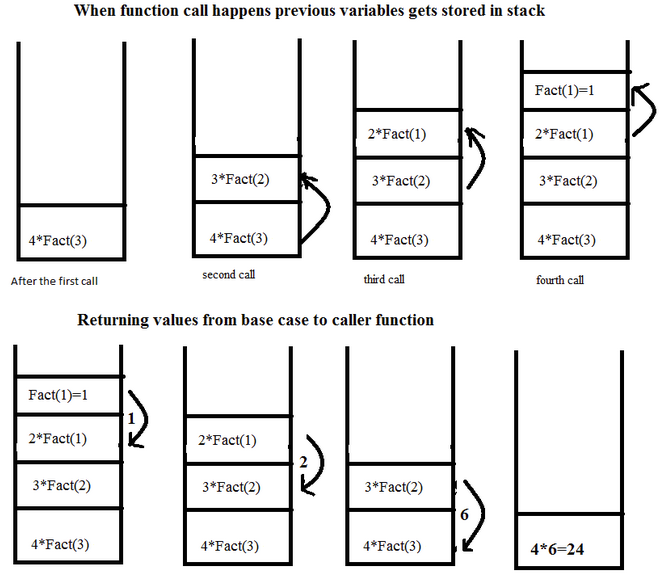
Output:

Factorial of 5 is 120



**Stack** is a LIFO ( Last In First Out) data structure. coming to **recursion**, **Recursion** is technique of solving any problem by calling same **function** again and again until some breaking (base) condition where **recursion** stops and it starts calculating the solution from there on.

Stack also strores the local variables in that function



**By using recursive functions find the sum of digits of a number.**

int sum(int num)

{

if(num==0)

return 0;

else

{

int res, rem;

rem = num%10;

res = rem + sum(num/10);

return res;

}

}

int main()

{

int num;

num = 1234;

res = sum(num);

printf(“the sum of digits of the given no. %d is %d”, num, res);

return 0;

}

output:

the sum of digits of the given no. 1234 is 10

**Printing fibbonacci series using recursive functions:**

void fibb(int val1, int val2, int num)

{

int temp = val2;

printf(“%d”, val1);

val2 = val2+val1;

val1 = temp;

if(num == 1)

return ;

fibb(val1,val2, --num);

}

int main()

{

int num;

scanf(“%d”, &num); // assume num = 5

fibb(0, 1, num);

}

Output:

0 1 1 2 3

**POINTERS**

P.Deepika

Emp no:816 Date:27December2016

Pointer is a variable used to store the address of variable.Pointers can stores the address of variable or address of array or address of function or address of address pointer.

(i) An address operator or reference operator '&' ,which returns the address of variable when placed before it. This operator can be read as “ the address of ”.

Example:

int \*iptr,age=30;

double \*dptr,x=1500.50;

iptr=&age;

dptr=&x;

iptr contains the address of variable age i.e it points to variable age, similarly dptr points to variable x.

(ii) By placing dereference operator or indirection operator before a pointer variable,we can access the variable whose address is stored in the pointer.Indirection operator can be read as 'value at the address'.

Example:

int v1=87;

float v2=4.5;

int \*p1=&v1;

float \*p2=&v2;

If we place ' \* ' before p1 then we can access the variable whose address is stored in p1. Since p1 contains the address of variable ' a ', we can access the variable ' a ' by writing \*p1. Similarly we can access variable ' b ' by writing \*p2.

**Example programs on pointers:**

**(1).**#include<stdio.h>

#include<math.h>

int main( )

{

int arr[5]={1,2,3,4,5};

int index,size;

int \*iptr;

iptr=arr;

size=sizeof(arr) /sizeof(int);

for(index=0;index<size;index++)

{

printf(“%p %d\n”,iptr,\*iptr);

iptr++;

}

return 0;

}

output:

Note:Assume array address starts at 1000

1000 1

1004 2

1008 3

1012 4

1016 5

**(2)**.**#**include<stdio.h>

#include<math.h>

int main( )

{

char arr[6]=”hello”;

int index,size;

char \*chptr;

chptr=arr;

size=sizeof(arr) /sizeof(char);

for(index=0;index<size;index++)

{

printf(“%p %c\n”,chptr,\*chptr);

chptr++;

}

return 0;

}

output:

Note:Assume array address starts at 1000

1000 h

1001 e

1002 l

1003 l

1004 o

1005

**(3).**#include<stdio.h>

int main( )

{

int arr[5];

int index;

int \*iptr;

iptr=arr;

for(index=0;index<5;index++)

{

printf(“enter the value %d:”,index+1);

scanf(“%d”,&arr[index]);

}

for(index=0;index<5;index++)

{

printf(“%p %d\n”,iptr,\*iptr);

iptr++;

}

return 0;

}

output:

Enter the value 1:1

Enter the value 2:2

Enter the value 3:3

Enter the value 4:4

Enter the value 5:5

Note:Assume array address starts at 1000

1000 1

1004 2

1008 3

1012 4

1016 5

**POINTERS**

Notes prepared by-

G. Divya Preethika

Emp Id. 814

Date:28 December2016

1. **Accept array elements and print the values using pointer**

#include<stdio.h>

int main()

{

int arr[10], \*ptr, size, cntr;

ptr=arr; // or ptr = &arr[0];

printf("Enter how many elements?:");

scanf("%d", &size);

for(cntr=0; cntr<size; cntr++)

{

printf("Enter num %d: ",cntr+1);

scanf("%d", &arr[cntr]);

}

printf("The elements are::\n");

for(cntr=0; cntr<size; cntr++)

{

printf("%p---%d\n",ptr,\*ptr);

ptr++;

}

return 0;

}

**Output:**

Enter how many elements?:5

Enter num 1: 10

Enter num 2: 20

Enter num 3: 30

Enter num 4: 40

Enter num 5: 50

The elements are::

0x7ffd99258c10---10

0x7ffd99258c14---20

0x7ffd99258c18---30

0x7ffd99258c1c---40

0x7ffd99258c20---50

1. **Write a program to accept numbers from user, if user enters 0 then stop accepting inputs and display array elements using pointers.**

#include<stdio.h>

int main()

{

int arr[10], \*ptr, size, cntr, index=0;

do{

printf("Enter num %d: ",index+1);

scanf("%d", &arr[index]);

if(arr[index] == 0)

break;

index++;

}while(arr[index-1]!=0);

ptr=arr; // or ptr = &arr[0];

printf("The elements are::\n");

for(cntr=0; cntr<index; cntr++)

{

printf("%p---%d\n",ptr,\*ptr);

ptr++;

}

return 0;

}

**OUTPUT:**

Enter num 1: 10

Enter num 2: 20

Enter num 3: 30

Enter num 4: 40

Enter num 5: 50

Enter num 6: 0

The elements are::

0x7fff76dd06f0---10

0x7fff76dd06f4---20

0x7fff76dd06f8---30

0x7fff76dd06fc---40

0x7fff76dd0700---50

1. **Write a program to show the arithmetic operations on pointers**

#include<stdio.h>

int main()

{

int arr[]={10,20,30,40,50,60,70,80,90,100};

int \*ptr, index;

ptr=arr;

printf("The elements of the array are:\n");

for(index=0; index<10; index++)

printf("%d\t",arr[index]);

printf("\nptr=%p------\*ptr=%d\n", ptr,\*ptr);

ptr++;

printf("After ptr++\nptr=%p------\*ptr=%d\n", ptr,\*ptr);

ptr = ptr+6;

printf("After ptr+6\nptr=%p------\*ptr=%d\n",ptr,\*ptr);

ptr = ptr-5;

printf("After ptr-5\nptr=%p------\*ptr=%d\n",ptr,\*ptr);

--ptr;

printf("after --ptr\nptr=%p------\*ptr=%d\n",ptr,\*ptr);

return 0;

}

**OUTPUT:**

The elements of the array are:

10 20 30 40 50 60 70 80 90 100

ptr=0x7fff3f38e410------\*ptr=10

After ptr++

ptr++=0x7fff3f38e414------\*ptr=20

After ptr+6

ptr=0x7fff3f38e42c------\*ptr=80

After ptr-5

ptr=0x7fff3f38e418------\*ptr=30

after --ptr

ptr=0x7fff3f38e414------\*ptr=20

1. **Write a program to perform operations on pointers**

#include<stdio.h>

int main()

{

int arr[20]={10, 20}, \*ptr, index, res;

ptr = arr;

printf("arr[0]=%d-----&arr[0]=%p\narr[1]=%d-------&arr[1]=%p\n",arr[0],&arr[0],arr[1],&arr[1]);

res = \*ptr++; //(internally first res=\*ptr;

// \*ptr=ptr+1;)

printf("\nafter res = \*ptr++\n%p-----%d\n", ptr, res);

ptr = arr;

res = ++\*ptr; //(internally first \*ptr=ptr+1;

//res=\*ptr;)

printf("\nafter res = ++\*ptr\n%p-----%d\n", ptr, res);

ptr = arr;

res = \*++ptr; //(internally first ptr++;

// res = \*ptr;)

printf("\nafter res = \*++ptr\n%p-----%d\n", ptr, res);

ptr = arr;

res = (\*ptr)++; //(internally first res = \*ptr;

// \*ptr = \*ptr+1;)

printf("\nafter res = (\*ptr)++\n%p-----%d\n", ptr, res);

return 0;

}

**OUTPUT:**

arr[0]=10-----&arr[0]=0x7ffea70ac970

arr[1]=20-------&arr[1]=0x7ffea70ac974

after res = \*ptr++

0x7ffea70ac974-----10

after res = ++\*ptr

0x7ffea70ac970-----11

after res = \*++ptr

0x7ffea70ac974-----20

after res = (\*ptr)++

0x7ffea70ac970-----11

**POINTER ARITHMETIC**

Pointer arithmetic is very important to understand, if you want to have complete knowledge of pointer. In this topic we will study how the memory addresses change when you increment a pointer.

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer

ptr++;

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

**Valid pointer operations:**

1. **Incrementing Pointer:**

#include<stdio.h>

**int** main(){

**int** \*ptr, myarr[5];

ptr=myarr;

printf(“Value of ptr : %u”, ptr);

ptr=ptr+1;

printf("New Value of ptr : %u",ptr);

**return** 0;

}

**Output :**

Value of ptr : 1000

New Value of ptr : 1004

1. **Decrementing Pointer**

decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below

#include <stdio.h>

int MAX = 3;

int main ()

{

int var[] = {10, 100, 200};

int cntr, \*ptr;

/\* let us have array address in pointer \*/

ptr = &var[MAX-1];

for ( cntr = MAX; cntr > 0; cntr--) {

printf("Address of var[%d] = %x\n", cntr-1, ptr );

printf("Value of var[%d] = %d\n", cntr-1, \*ptr );

/\* move to the previous location \*/

ptr--;

}

return 0;

}

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

Address of var[2] = bfedbcd8

Value of var[2] = 200

Address of var[1] = bfedbcd4

Value of var[1] = 100

Address of var[0] = bfedbcd0

Value of var[0] = 10

1. **Addition of Pointer and Number**
2. **Subtraction of Pointer and Number**

**Write a program to perform increment and decrement operations on pointers.**

#include<stdio.h>

int main()

{

int arr[10]={1,2,3,4,5,6,7,8,9,10}, \*iptr1, \*iptr2, value;

iptr1 = &arr[2];

iptr2 = &arr[5];

printf("arr[2]=%d----&arr[2]=%p\n",arr[2],&arr[2]);

printf("arr[5]=%d----&arr[5]=%p\n",arr[5],&arr[5]);

value= iptr2-iptr1;

printf("value= iptr2-iptr1\nvalue=%d\n",value);

value= iptr1-iptr2;

printf("value= iptr1-iptr2\nvalue=%d\n", value);

value=\*iptr1++;

printf("value= iptr1++\nvalue=%d----iptr1=%p\n", value, iptr1);

value=\*iptr2--;

printf("value= iptr2--\nvalue=%d------iptr2=%p\n", value,iptr2);

iptr1 = &arr[0];

printf("After iptr1=&arr[0]\niptr1++ = %p\n", iptr1++);

iptr1 = &arr[1];

printf("After iptr1=&arr[1]\niptr1-- = %p\n", --iptr1);

return 0;

}

**OUTPUT:**

arr[2]=3----&arr[2]=0x7ffe7b5dcff8

arr[5]=6----&arr[5]=0x7ffe7b5dd004

value= iptr2-iptr1

value=3

value= iptr1-iptr2

value=-3

value= iptr1++

value=3----iptr1=0x7ffe7b5dcffc

value= iptr2--

value=6------iptr2=0x7ffe7b5dd000

After iptr1=&arr[0]

iptr1++ = 0x7ffe7b5dcff0

After iptr1=&arr[1]

iptr1-- = 0x7ffe7b5dcff0

1. **Differencing between two pointers**

#include<stdio.h>

**int** main(){

int arr[] = {10,20,30,40,50};

**int** \*ptr1=&arr[1]; // assume arr[0] address is 1004

**int** \*ptr2=&arr[3]; // assume arr[0] address is 1012

printf("\nDifference : %d",ptr2-ptr1); //internally (1012-1004)/sizeof(int)=> 8/4 => 2

**return** 0;

}

**Output :**

Difference : 2

**Invalid pointer operations:**

1. addition of two pointers
2. product of pointer and a numerical
3. pointer divided by numerical
4. sum of pointer and float value

Date:29/12/2016 ISHAQUE

* CALL BY VALUE AND CALL BY REFERENCE:

Call by Value:

If we want to pass only one value then we have to use call by value.

Call By Reference:

If we want to return one value or more than one value then we can use call by reference.

Example:

void call\_by\_ref\_sample(int ,int \*);

int main()

{

int val1,val2;

val=10;

val2=20;

call\_by\_ref\_sample(val1,&val2);

printf(“%p %d\n”,&val2,val2);

return 0;

}

void call\_by\_ref\_sample(int v1,int \*v2)

{

v1=v1\*2;

\*v1=(\*v2)\*2;

printf(“%p %d\n”,&v1,v1);

printf(“%p %d\n”,v2,\*v2);

}

Assignment question: 1) accept an array paas as argument.remove the even no. From array and print

the array which contain only odd no. By call by reference.

print the fibonaci series by call by reference.

* write a programme reverse a array print it into main function by using call by reference.

void myreverse(int \*,int );

int main()

{

int myarr[5]={3,4,5,6,8},size=5;

myreverse(myarr,size);

for(index=0;index<size;index++)

{

printf(“%d”,myarr[index]);

}

}

void myreverse(int myarr[],int size)

{  
 int i,j,temp;

for(i=0;j=size-1;i<j;i++,j++)

{

temp=myarr[j];

myarr[j]=myarr[i];

myarr[i]=temp;

}

}

* make your own stringcpy function

int main()

{  
 char dest[100];

char scr[]=”votarytech”;

int \*p;

p=mystrcpy(dest,scr);

printf(“%p:,%d”,p,p);

}

char \*mystrcpy(char dest[],const char scr[])

{

int i;

for(i=0;scr[i]!='\0';i++)

{  
 dest[i]=scr[i];

}

dest[i]='\0';

return dest;

}

**DYNAMIC MEMORY ALLOCATION:03:01:2017**(G.SANDEEP REDDY)

It means allocating memory dynamically which means at run time.DMA is used for effective memory usage.It user heap section for memory allocation.

We can allocate memory using three ways,

**1.malloc:**

syntax:void \*malloc(size\_t size);

It is a predefined function which is used to allocate memory,on successfull allocation it return the starting address of allocated memory,and on falure it returns NULL.

By default allocated address is initialised with garbage values.

Eg:

int \*ptr;

ptr=(int \*)malloc(4); //here it allocates 4 bytes to the ptr

ptr=(int \*)malloc(4\*sizeof(int)); //here it allocates sizeof 4 ints i.e 16 bytes

int main()

{

int \*ptr;

int bytes,index;

printf(“entere no.of byte to be allocated to ptr:” );

scanf(“%d”,&bytes);

ptr=(int \*)malloc(bytes\*sizeof(int));

printf(“enter elements into ptr\n”);

for(index=0;index<bytes;index++)

scanf(“%d”,ptr[index]);

for(index=0;index<bytes;index++)

printf(“%d”,ptr[index]);

free(ptr);

return 0;

}

**2.calloc:**

syntax:void \*calloc(size\_t nmemb,size\_t size);

It is a predefined function which is used to allocate memory block wise,on successfull allocation it return the starting address of allocated memory,and on falure it returns NULL.

By default allocated address is initialised with NULL.

Eg:

int \*ptr;

ptr=(int \*)calloc(1,4); //here it allocates 1 block of 4 bytes i.e allocates 4 bytes

ptr=(int \*)calloc(4,sizeof(int)); //it allocates 4 blocks,each block is 4bytes i.e toal 16bytes

int main()

{

int \*ptr;

int bytes,index;

printf(“entere no.of byte to be allocated to ptr:” );

scanf(“%d”,&byte);

ptr=(int \*)calloc(bytes,sizeof(int));

printf(“enter elements into ptr\n”);

for(index=0;index<bytes;index++)

scanf(“%d”,ptr[index]);

for(index=0;index<bytes;index++)

printf(“%d”,ptr[index]);

free(ptr);

return 0;

}

**3.realloc:**

syntax:void \*realloc(void \*ptr,size\_t size);

The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

on successfull allocation it return the starting address of allocated memory,and on falure it returns NULL.

Eg:

int \*ptr;

ptr=(int \*)realloc(ptr,4); //it allocates 4 bytes to ptr

ptr=(int \*)realloc(ptr,10); //it allocates 10 bytes continuous memory,contents upto 4 bytes are doesn't change

int main()

{

int \*ptr=NULL;

int choice,index,cnt=0;

printf(“do you want to enter an element into ptr,if yes press 1,else press otherthan 1 ” );

scanf(“%d”,&choice);

while(choice==1)

{

ptr=(int \*)realloc(ptr,(cnt+1)\*sizeof(int));

printf(“enter an element into ptr\n”);

scanf(“%d”,ptr[cnt]);

cnt++;

printf(“do you want to enter an element into ptr,if yes press 1,else press otherthan 1” );

scanf(“%d”,&choice);

}

for(index=0;index<cnt;index++)

printf(“%d”,ptr[index]);

free(ptr);

return 0;

}

**4.free:**

Syntax:free(void \*ptr);

It used to free the memory allocated by ptr,if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

Eg:

int \*ptr;

ptr=(int \*)malloc(4);

free(ptr);//free the 4 bytes of memory

**STRUCTURE**

**Notes pepared by**

**Yasir NL**

**Emp id : 830**

**date : 04/01/2017**

Arrays allow to define type of variables that can hold several data items of the same kind. **Similarly structure is another user defined data type available in C that allows to combine data items of different kinds.**

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

**struct <structure tag> {**

**member definition;**

**member definition;**

**...**

**member definition;**

**} <Variables>;**

## Defining a Structure

**To define a structure, you must use the struct keyword** e struct statement defines a new data type, with more than one member. The format of the struct statement is as follows .

structure data member's are stored in contiguous memory location......

structure data member's are accessed by using .(dot) operater

Example on structure template :

struct lc\_engr {

char name[50];

int id;

int test\_marks;

} ;

**There is two way to declaration of structure variable**

1) With structure declaration

2) Using the structure tag

**With structure declaration**

struct lc\_engr {

char name[50];

int id;

int test\_marks;

}**eng1,eng2** ;

**Using the structure tag**

struct lc\_engr {

char name[50];

int id;

int test\_marks;

**};**

int main()

**{**

struct lc\_engr eng1;

struct lc\_engr eng2 , eng3;

printf(“Size of Structure : %u”,sizeof(eng1)); // 52

printf(“Size of Structure : %u”,sizeof(struct lc\_engr)); //52

}

**Padding**

Padding is the concept of alignment,for the issue of computer efficiency and the speed of the access of the data,aligned data are perfectly accessed

with the fetching cycle of the processor from the addresses where the data are stored,it doesn't mean that with out alignment processor doesn't work it only meant for the speed access of the memory,for the integer data type it is 4 byte

alignment is done by the compiler to access the data more efficiently by the processor.

**Structure Padding**

struct lc\_engr {

char name[5];

int id;

int test\_marks;

}; // memory allocated for this structure is 16 byte 3 byte will be wested here

to avoid this problem we need to use the #pragma

**initialization of structure varibles**

struct lc\_engr {

char name[50] = “Yasir”; // Invalid

int id;

int test\_marks = 89; // Invalid

}; // We cannot initialize structure varibles inside the structure

struct lc\_engr {

char name[50];

int id;

int test\_marks;

**}**eng = {“Votary”,12,100}**;**

#include<stdio.h>

struct lc\_engr {

char name[50];

int id;

int test\_marks;

};

int main()

{

struct lc\_engr eng1 = {"VotaryTech",14,600};

struct lc\_engr eng2 , eng3;

}

**Program to accessing the indivisual variable of structure and aasigning a value to indivisual varible of structure**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

struct lc\_engr {

char name[50];

int id;

int test\_marks;

};

int main()

{

struct lc\_engr eng1 = {"VotaryTech",14,600};

struct lc\_engr eng2 , eng3;

strcpy(eng2.name,"Well come");

eng2.id = 40;

eng2.test\_marks = 100;

**eng3 = eng2** // **assign a structure vasriable to the another struture variable**

**if(eng3 == eng2 ) //** this is not possible in structures

printf(“both strutures are equal”);

printf("%s\t%d\t%d \n",eng1.name,eng1.id,eng1.test\_marks);

printf("%s\t%d\t%d",eng2.name,eng2.id,eng2.test\_marks);

}

**Bit-Filds in structured**

struct lc\_engr {

char name[4];

int id : 4; //Four bits of memory is allocated

int test\_marks : 5; //Five bits of memory is allocated

};

#include<stdio.h>

struct Lceng

{

int id;

int \*arry\_ptr;

};

int main()

{

int index;

int arr[]={1,2,3,4,5};

struct Lceng st;

st.id=10;

st.arry\_ptr =arr;

for(index=0;index<5;index++)

printf("%d ",st.arry\_ptr [index]);

}

**Dynamic memory allocation for struture variable**

struct Lceng

{

char\*name;

int id;

};

int main()

{

struct Lceng myvar;

myvar.name=(char\*)malloc(sizeof(char)\*10);

if(myvar.name==NULL)

{

printf("memory allocation not successful:\n");

exit(0);

}

else

{

strcpy(myvar.name,"Votary");

id=78;

printf("%s\t%d\n",name,id);

}

}

**array of strutures**

struct student

{

char name[10];

int sid;

float per;

};

int main()

{

struct student s1[2];

int arr[2];

struct student \*sptr=&s1[0];

int i;

printf("%d\n",sizeof(s1[0]));

for(i=0;i<2;i++)

{

scanf("%d%s%f",&s1[i].sid,s1[i].name,&s1[i].per);

printf("%d\t%s\t%f\n",s1[i].sid,s1[i].name,s1[i].per);

}

for(i=0;i<2;i++)

{

scanf("%d%s%f",&sptr->sid,sptr->name,&sptr->per);

printf("%d\t%s\t%f\n",sptr->sid,sptr->name,sptr->per);

sptr++;

}

}

**Nesting of structures**

struct one

{

struct two

{

int iva;

}inr\_st;

float fva;

double dva;

};

int main()

{

struct one nst\_st;

nst\_st.inr\_st.iva=89;

nst\_st.fva= 12.3;

nst\_st.dva= 12.4;

printf("%d %f %lf",nst\_st.inr\_st.iva , nst\_st.fva, nst\_st.dva);

}

**passing a structure to a function--> by value, by address**

struct product\_db

{

char pname[10];

int pro\_id;

float pro\_price;

}t1={"accessories",123,897.3};

void passbyvalue(struct product\_db);

void passbyaddress(struct product\_db\*);

int main()

{

struct product\_db p1,p2,p3;

struct product\_db s[2];

int i;

printf("enter the structure datamember value:\n");

scanf("%s%d%f",p1.pname,&p1.pro\_id,&p1.pro\_price);

passbyvalue(p1);

p3=p1;

printf("%s\t%d\t%f\n",p3.pname,p3.pro\_id,p3.pro\_price);

printf("pro\_id=%d\n",p1.pro\_id);

passbyaddress(&p1);

printf("pname=%s\n",p1.pname);

for(i=0;i<2;i++)

printf("enter the structure datamember value:\n");

scanf("%s%d%f",s[i].pname,&s[i].pro\_id,&s[i].pro\_price);

for(i=0;i<2;i++)

printf("%s%d%f",s[i].pname,s[i].pro\_id,s[i].pro\_price);

struct product\_db \*sptr=&s[0]; sptr=s;

printf("%s%d%f",sptr->pname,(sptr->pro\_id),(sptr->pro\_price)); //accessing s[0]

printf("sptr=%p\n",sptr);

sptr++;

printf("sptr=%p\n",sptr);

printf("%s%d%f",sptr->pname,(sptr->pro\_id),(sptr->pro\_price)); //accessing s[1]

}

void passbyvalue(struct product\_db t)

{

t.pro\_id=67;

return;

}

void passbyaddress(struct product\_db\*ptr)

{

strcpy(ptr->pname,"keyboard");

return;

}

**UNIONS**

**prepared by:** Lakshmi Kiran J

**Notes:05/01/2017**

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

## Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

union [union tag] {

member definition;

member definition;

...

member definition;

} [one or more union variables];

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

Unions are quite similar to structures in C. Like structures, unions are also derived types.

Defining a union is as easy as replacing the keyword **struct** with the keyword **union**.

union car

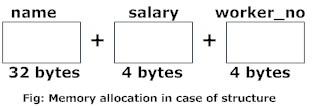
{

char name[50];

int price;

};

The amount of memory required to store a structure variable is the sum of memory size of all members.



But, the memory required to store a union variable is memory required for the largest element of an union.



* Storing the data in union is done in two different ways.

1. Different Data, same purpose

2. Same Data, Different ways

In Case 1: (Different Data,same purpose)

**program:**

#include<stdio.h>

union ranking {

int avg;

char grade[4];

}

main()

{

union ranking emp1;

emp.avg=80;

printf(“%p%d\n”,&emp1.avg,emp1.avg);

strcpy(emp1.grade,”T10”);

}

Output:

address of avg=0x7ffd8557ef60

emp avg=80

emp1 grade=T10

In Case2 (Same Data, Different ways):

**Program:**

#include<stdio.h>

union twobytes {

char bytes[2];

short int si;

}

main()

{

union twobytes bytes2;

bytes2.si=0x1234;

printf(“%p%x\n”,&bytes2.si,bytes2.si);

printf(“%p%x\n”,&bytes2.byte[0],bytes2.byte[0]);

printf(“%p%x\n”,&bytes2.byte[1],bytes2.byte[1]);

}

Output:

address of bytes2.si=0x7ffca01b59c0

value of bytes.si=1234

address of bytes2.byte[0]=0x7ffca01b59c0

value of bytes2.byte[0]=34

address of bytes2.byte[1]=0x7ffca01b59c1

value of bytes2.byte[1]=12

Bit Fields:

The declaration of a bit-field has the following form inside a structure −

struct {

type [member\_name] : width ;

};

The following table describes the variable elements of a bit field −

|  |  |
| --- | --- |
| Elements | Description |
| type | An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int. |
| member\_name | The name of the bit-field. |
| width | The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. |

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

struct {

unsigned int age : 3;

} Age;

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example −

#include <stdio.h>

#include <string.h>

struct {

unsigned int age : 3;

} Age;

int main( ) {

Age.age = 4;

printf( "Sizeof( Age ) : %d\n", sizeof(Age) );

printf( "Age.age : %d\n", Age.age );

Age.age = 7;

printf( "Age.age : %d\n", Age.age );

Age.age = 8;

printf( "Age.age : %d\n", Age.age );

return 0;

}

When the above code is compiled it will compile with a warning and when executed, it produces the following result −

Sizeof( Age ) : 4

Age.age : 4

Age.age : 7

Age.age : 0

**FILES**

In C programming, file is a place on your physical disk where information is stored.

## Why files are needed?

* When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
* If you have to enter a large number of data, it will take a lot of time to enter them all.  
  However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
* You can easily move your data from one computer to another without any changes.

**Types of Files**

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

**1. Text Files:**

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

**2.Binary Files:**

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

## File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Need to open anew file
2. Read the file/write/append
3. Traversal
4. Closing a file

## Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE \*fptr;

## Opening a file - for creation and edit

Opening a file is performed using the library function in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

ptr = fopen("fileopen","mode")

For Example:

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

* Let's suppose the file new program.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.  
  The writing mode allows you to create and edit (overwrite) the contents of the file.
* Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'.  
  The reading mode only allows you to read the file, you cannot write into the file.

**Program 1**

#include<stdio.h>

int main()

{

char ch;

FILE \*fp;

fp=fopen("read\_chars.txt","r");

if(fp=NULL)

{

printf("File is not present");

}

ch=fgetc(fp);

while(ch!=EOF)

{

printf("%c",ch);

ch=fgetc(fp);

}

}

**Output:**

In read\_chars.txt file whatever is there it will print

ex: asfghjk

**Program 2**

#include<stdio.h>

main()

{  
FILE \*fp;

char ch;

fp=fopen(“write\_chars.txt”,”w”);

if(fp==NULL)

{

printf(“File is not present”);

}

printf(“Enter the character\n”);

ch=getchar();

while(ch!=EOF)

{

fputc(fp);

ch=getchar();

}

fclose(fp);

return 0;

}

**Output:**

It will write particular thing into the file.

Ex: afasgsdgvbh