ARRAY

2.1 What is an Array?

- Array is an ordered set which consist of fixed number of elements of same data type.
- Array is a linear data structure.
- Array is a collection of variables of same types all of which are referred by a common name.
- A specific element of array is accessed by an index.
- Array is classified into two categories: (1) One dimensional (2) Two dimensional.

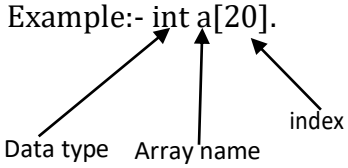
2.2 Characteristics of Array

1. An array holds elements that have the same data type.
2. Array elements are stored in subsequent memory locations.
3. Two-dimensional array elements are stored row by row in subsequent memory locations.
4. Array name represents the address of the starting element.
5. Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.
6. While declaring the 2D array, the number of columns should be specified and it's a mandatory. Whereas for number of rows there is no such rule.
7. An Array index by default starts from 0 and ends with the index no. (N-1).
8. The size of an Array cannot be changed at Run Time.

2.3 One dimensional Array:

- One dimensional array can be declared as

  Example:- int a[20].



- By the above statement we are declaring an array named as **a** and which can store 20 integer elements.
- While declaring array we need to give three piece of information: data type, array name and its size.
- The size specified in [] square brackets (subscript operator). Size must be given when array is declared.
- Array name is an identifier which follows the rules of writing an identifier.

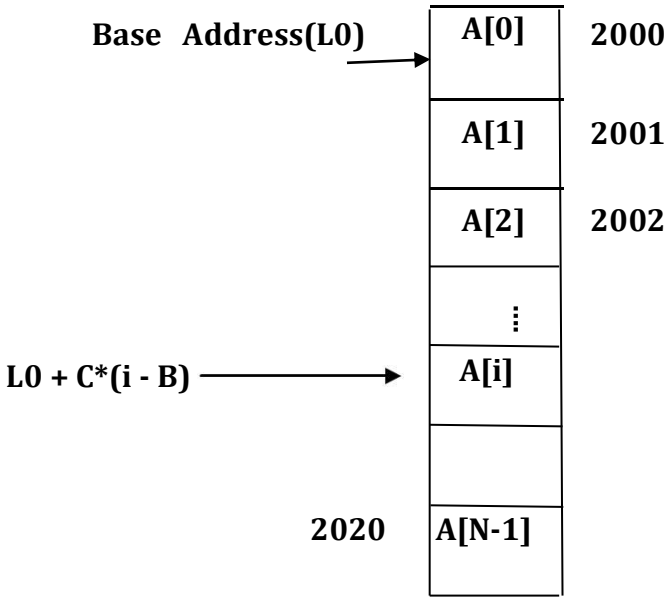
- Each integer takes 2 bytes so this array declaration reserves 40 bytes of memory for the array when the array element is initializing.
- The first elements of array starts at index 0(means a[0]) and index of last element is n-1.

- The starting index for the array i.e. 0 here is known as Lower Bound (LB) and higher index n-1 is known as Upper Bound (UB).
- Number of elements in array is given by the following formula:

  N=UB-LB+1

Storage Representation of One Dimensional Array:

- In one dimensional array the number of memory location is sequentially allocated to the array. So it is also known as sequential list.
- If each element requires one word in memory then n elements array occupies n consecutive words in memory.
- Since the size of an array is fixed it requires fixed number of memory location.
- Representation of an array in memory location is shown below:



- L0 is the starting(Base) address or address of first element in the array.
- C is the number of words allocated to each element
- B is the starting index of an array.
- Thus location(Address) of A[i] in the array is calculated as:

  Address of A [i]=Loc(A[i]) = L0 + C*(i - B)

  If we want to find the address of A [2] then it is calculated as:

  Loc (A [2]) = L0 + C*(i - B)

  = 2000 + 2

  = 2002

2.4 Operations on One dimensional Array:

We can perform operation such as Traversal, Insert, Delete, Search, and Sorting and Find Big or Small Number of Array

1. **Traversal:** This operation is used to traverse one dimensional array for inserting or displaying elements of one dimensional array.

Algorithm: TRAVERSE (A, LB, UB)

- **A is an array**
- **LB is lower limit of an array**
- **UB is Upper limit of an array**

1. [Initialize] COUNT ← LB
2. [Perform Traversal and Increment counter]

   While (COUNT ≤ UB)

   (1) Write A [COUNT] or Read A [COUNT]

   (2) COUNT ← COUNT+1
3. [Finished]

   Exit

2. **Insertion:** This operation is used to insert an element into one dimensional array.

Algorithm: INSERT (A, POS, N, VALUE)

- **A is an array**
- **POS indicates position at which you want to insert element.**
- **N indicates number of elements in an array.**
- **VALUE indicates value(element) to be inserted.**

1. [Initialize] TEMP ← N
2. [Move the elements one position down]

   Repeat While (TEMP ≥ POS)

   (1) A [TEMP + 1] ← A[TEMP]

   (2) TEMP ← TEMP-1
3. [Insert element]

   A [POS] ← VALUE
4. [Increase size of array]

   N ← N + 1
5. [Finished]

Exit

3. <u>Deletion</u>: This operation is used to delete an element from one dimensional array.

Algorithm: DELETE (A, POS, N, VALUE)

- **A is an array**
- **POS indicates position at which we want to insert element.**
- **N indicates number of elements in an array.**
- **VALUE indicates value(element) to be inserted.**

1. [Initialization]

   TEMP ← POS

2. [Move the elements one position up]

   (1) A[TEMP] ← A[TEMP + 1]

   (2) TEMP ← TEMP+1

3. [Decrease size of array]

4. [Finished]

   Exit

4. <u>Searching</u>: This operation is used to search particular element in one dimensional array.

Algorithm: SEARCH(A, N, X)

- **These function searches the list A consist of N elements for the value of X.**

1. [Initialize search]

I ← 1

K [N+1] ← X

2. [Search the Vector]

Repeat while K [I] ≠ X

I ← I+1

3. [Successful Search?] If

I = N+1 then

Write "Unsuccessful Search"

Return 0

Else

Write "Successful Search"

Return 1

5. **Sorting:** This operation is used to sort the elements of array in ascending order or descending order.

Algorithm: SORT(A,N)

- **These function sort elements of array A consist of N elements.**
- **PASS denotes the pass index.**
- **LAST denotes the last unsorted element.**
- **EXCHS counts the total number of exchange made during pass.**

1. [Initialize]
   LAST←N
2. [Loop on pass index]
   Repeat thru step 5 for PASS = 1, 2, …., N-1
3. [Initialize exchange counter for this pass]
   EXCHS← 0
4. [Perform pair wise comparisons on unsorted elements]
   Repeat for I = 1, 2… LAST-1
   If K[I] > K[I+1] then
   K [I] K [I+1]
   EXCHS← EXCHS+1
5. [Exchange made on this pass?]
   If EXCHS = 0 then
   Write "Mission
   Complete" Else
   LAST←LAST-1
6. [Finished]
   Return (Minimum number of pass required)

2.5 Two Dimensional Array (Storage Representation)

(1) Row-major Arrays

- One method of representing a two dimensional array in memory is the row major order representation.
- In this representation the first row of the array occupies the first set of memory locations reserved for the array, the second row occupies the next set and so on.
- An array consisting of n rows and m columns can be stored sequentially in row major order as :

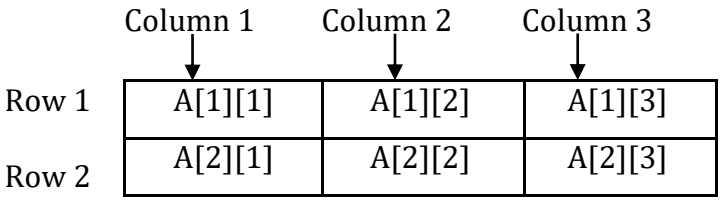
  Row1 A[1,1] A[1,2] A[1,3] A[1,4]……..A[1,m]
  Row2 A[2,1] A[2,2] A[2,3] A[2,4]……..A[2,m]

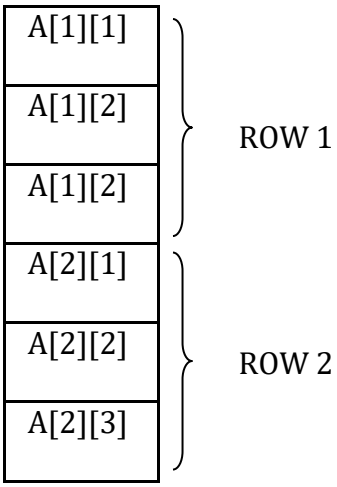
……

Row n A[n,1] A[n,2] A[n,3] A[n,4]……..A[n,m]

- **Example:**

  A two dimensional array consist of two rows and three columns is stored sequentially in row major order as:





- The address of element A[i, j] can be obtained by evaluating expression:

  Loc (A [i, j]) = L_0 + (i - 1) * m + (j - 1)

- Where L_0 is the address of the first element(Base Address) in the array.

  i is the row index.

  j is the column index.

  m is the no. of columns.

- for example the address of element A[1,2] is calculated as:

  A [1, 2] = L_0 + (i - 1) * m + (j - 1)

  Here, m=3, n=2, i=1, j=2

  =L_0 + (1 - 1) * 3 + (2 - 1)

  =L0 + 0 +1

  =L0 + 1

(2) <u>Column-major Arrays</u>

- One method of representing a two dimensional array in memory is the column major order representation.

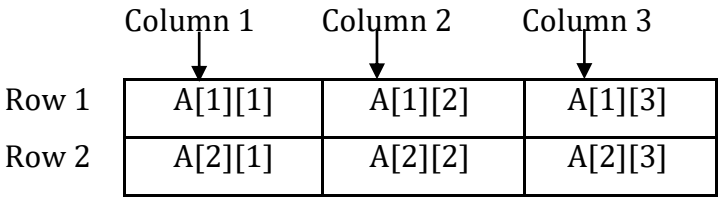
- In this representation the first column of the array occupies the first set of memory locations reserved for the array, the second column occupies the next set and so on.
- An array consisting of n rows and m columns can be stored sequentially in column major order as :

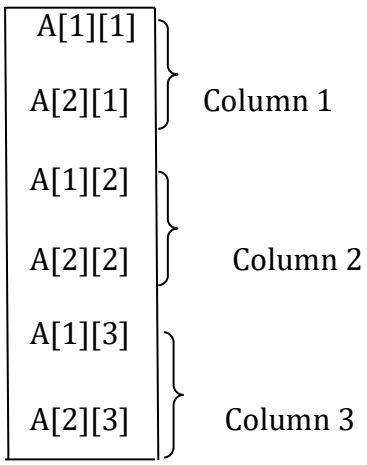
  Column 1 A[1,1] A[2,1] A[3,1] A[4,1]……..A[n,1]

  Column 2 A[1,2] A[2,2] A[3,2] A[4,2]…….. A[n,2]

  ……

  Column m A[1,m] A[2,m] A[3,m] A[4,m]……..A[n,m]
- **Example:** A two dimensional array consist of two rows and three columns is stored sequentially in column major order as:





- The address of element A[i, j] can be obtained by evaluating expression:

  Loc (A [i, j]) = L_0 + (j - 1) * n + (i - 1)
- Where L_0 is the address of the first element(Base Address) in the array.

  i is the row index

  j is the column index
- for example the address of element A[1,2] is calculated as:

  A[1,2] = L_0 + (j - 1) * n + (i - 1)

  Here, m=3, n=2 , i=1, j=2

  =L_0 + (2 - 1) * 2 + (1 - 1)

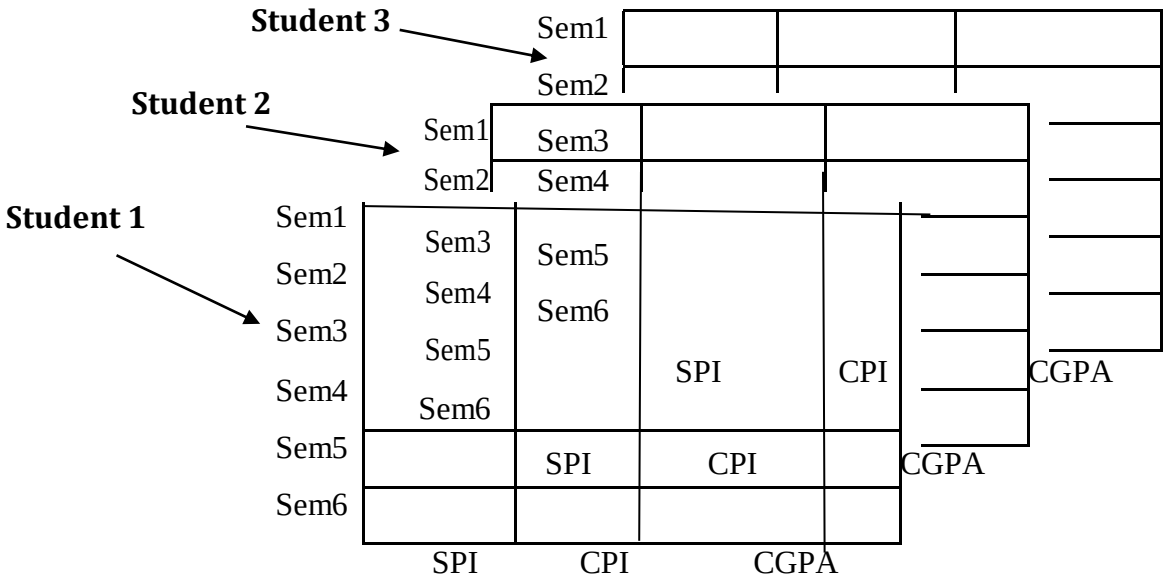
  =L_0 + 2 + 0

  =L_0 + 2

2.6 Three or Multidimensional Arrays

- The C also allows the use of three or more dimensional arrays which are known as multi dimensional arrays.
- The two dimensional are declared as follows:

  Data type array_name[d1][d2][d3]...[dn].
- If I want to display student result in form of SPI, CPI, CGPA from sem1 to sem 6. So we have to use three dimensional array.
- In three dimensional array first dimensional is student, second dimensional is sem and third dimensional is result.



2.7 Strings

- String is a group of characters.
- The storage structure for strings is same as characters as string is a sequence of characters.
- A string is stored in computer memory as series of ASCII codes of the characters in a string.
- Each string in C is ended with the special character called null character and denoted as '\0' or NULL.
- **Character set of string:**
  1. Alphabets (a-z, A-Z)
  2. Numeric (0-9)
  3. Special Character (+,/,*,$,@,#,..etc)
- For example:

String "GOOD" is stored as shown in fig

'G'	'O'	'O'	'D'	'\0'

- In the above string four characters need five bytes of storage.
- **String Functions:**
  1. Computing length of string.
  2. Copy one string into another.
  3. Reverse String
  4. Concate two strings.
  5. Compare two strings.
  6. Find substring.

2.8 Array of structure *OR* Structure and Array

- Many times we have to store information regarding real word entities like employees in a company or details of students in a particular class.
- Employees or the students in this case are entities which cannot be described by the single data item.

  For example:

  Student is described by its id number, name, address etc.
- Consider the above example, first we need to define the structure for student. It defined as below

```
struct Student
{
        int id_no; char
        name[20]; char
        address[50];
}s;
```

- The above declaration contains information of one student but if we want to store information about 120 students then we use concepts of Array of Structure.

```
struct Student
{
        int id_no;

        char name[20];
        char address[50];
}s[120];
```

- C use the dot (.) operator to access the members of the structure.
- Each and every member is accessed by structure variable like s[i].id_no, s[i].name, s[i].address, where 0 < i < 120.

2.9 Drawbacks of linear Arrays

- The major drawback of the linear or one dimensional array is the insertion and deletion operation is complicated and time consuming.
- If we declare array with size 50 and we create array of 20 elements so rest of 30 elements are occupies the memory space, which will be west the memory space.
- The size of array must be known first before it declared, otherwise it create overflow or underflow error.
- The array is best when the size of data or number of elements remains fixed.

2.10 Insertion and deletion operation are difficult and time consuming with array.

Insertion with Array:

- Suppose there are N elements in an array and we want to insert an element between first and second element.
- We have to move last N-1 elements down

Deletion with Array:

- Suppose there are N elements in an array and we want to delete an element from array.
- We have to move the elements up to take vacant space after deleting element.

Thus insertion and deletion operation are very difficult and time consuming with array.

2.11 Pointer and Array

- Pointer and array are related with each other.
- The name of array itself is address of array i.e. address of first element (a=&a[0]).

For example: int a[5];

a[0]	a
a[1]	
a[2]	
a[3]	
a[4]	

*(a+0)	a
*(a+1)	a+1
*(a+2)	a+2
*(a+3)	a+3
*(a+4)	a+4

- As a is a first element of array and it also defines base address.
- The only limitation to use array name a as a pointer is that it cannot be modified because array name is address of first element and is always constant.
- So here subscript of array(a[i]) is internally converted into *(a+i).
- So a[0]=*(a+0),a[1]=*(a+1)….etc. here array adds **base address+subscript*size of data type.**
- If base address is 2000 then address of a[2]=2000+2*2=2004.

2.12 Pointers and strings

- A pointer which pointing to an array which content is string, is known as pointer to array of strings.

- **Example: w.a.p to find reverse string of given string.**

```
#include   <stdio.h>
#include <string.h>
void main()
{
        char str1[] = "Pointers are fun and hard to use";
        char str2[80], *p1, *p2;
        /* make p point to end of str1 */
        p1 = str1 + strlen(str1) - 1;
         p2 = str2;
        while(p1 >= str1)
                *p2++ = *p1--;
        /* null terminate str2
         */ *p2 = '\0';
printf("%s %s", str1, str2); getch();}
```

POINTER

Introduction to Pointer

A pointer is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type ``pointer to int,`` it might point to the int variable i, or to the third cell of the int array a. Given a pointer variable, we can ask questions like, ``What's the value of the variable that this pointer points to?''

Addresses, the Address Operator, and printing Addresses

- All information accessible to a running computer program must be stored somewhere in the computer's memory. (RAM chips.)
- Particular locations in memory are identified by their address.
- Memory addresses on most modern computers are either 32-bit or 64-bit unsigned integers, though this may vary with particular computer architectures. (Ignoring segmentation and paging and other low-level memory details beyond the scope of this course.)
- Those memory addresses can be thought of as "house numbers", in a very long linear city where everyone lives on the same street.
- The address of a variable can be determined by applying the address operator, &.
- Addresses can be printed using the %p format specifier
- So for example, the code:

```
int i = 42;
printf( "The variable i has value %d, and is located at
        0x%p\n", i, &i );
```

might produce the following result:

```
The variable i has value 42, and is located at 0x0022FF44
```

Pointer Variables

Declaring Pointer Variables

- Since addresses are numeric values, they can be stored in variables, just like any other kind of data.
- Pointer variables must specify what kind of data they point to, i.e. the type of data for which they hold the address. This becomes very important when the pointer variables are used.
- When declaring variables, the asterisk, *, indicates that a particular variable is a pointer type as opposed to a basic type.
- So for example, in the following declaration:

```
int i, j, *iptr, k, *numPtr, *next;
```

variables i, j,and k are of type (int), and variables iptr, numPtr, and next are of type (pointer to int). Note that regular ints and int pointers can be mixed on a single declaration line.

Initializing Pointer Variables
Pointer variables can be initialized at the time that they are declared, just like any other variable. This is normally done using the address operator, &, applied to any previously declared variable (that is defined in the current scope.)
For example:

```
int iGlobal = 0;
int main( void ) {
int i = 1;
int j = 2, * iGlobalptr = & iGlobal, *iptr = &i, *jptr = &j;
int *illegal = & number; // This line causes a compiler error
int number = 42;
. . .
} // main
```

NULL Pointers

- Uninitilized pointers start out with random unknown values, just like any other variable type.
- Accidentally using a pointer containing a random address is one of the most common errors encountered when using pointers, and potentially one of the hardest to diagnose, since the errors encountered are generally not repeatable.
- Therefore, POINTER VARIABLES SHOULD ALWAYS, ALWAYS, ALWAYS BE INITIALIZED WHEN THEY ARE DECLARED.
- If you don't have any better value to use to initialize your variables, use the special macro NULL, (which is essentially a zero formatted as an address.)
- NULL pointers are safer than uninitialized pointers because the computer will stop the program immediately if you try to use one, as opposed to blindly letting you follow an uninitialized pointer off to la la land.

Example:

```
double d, *dptr = NULL, e;
```

Assigning Pointer Variables

- Pointers can be assigned values as the program runs, just like any other variable type. For example:

```
int i = 42, j = 100;
int *ptr1 = NULL, *ptr2 = NULL;     // ptrs initially point nowhere.

ptr1 = &i;                  // Now ptr1 points to i.

ptr2 = ptr1;                 // Now both pointers point to i.  Note no & operator

ptr1 = &j;                  // ptr1 changed to point to j.

ptr2 = NULL;                 // ptr2 back to pointing nowhere.
```

Pointers and Arrays
Pointers to Array Elements
A pointer may be made to point to an element of an array by use of the address operator:

```
int nums[ 10 ], iptr = NULL;

iptr = & nums[ 3 ];             // iptr now points to the fourth element

*iptr = 42;                                 // Same as nums[ 3 ] = 42;
```

Interchangeability of Pointers and Arrays

Because of the pointer arithmetic works, and knowing that the name of an array used without subscripts is actually the address where the beginning of the arrays is located, and assuming the following declarations:

int nums[10], *iptr = nums;

Then the following statements are equivalent:

nums[3] = 42;

*(iptr + 3) = 42;

What may come as more of a surprise is that the following two statements are also legal, and equivalent to the first two:

iptr[3] = 42;

*(nums + 3) = 42;

Basically since nums and iptr are both addresses of where ints are stored, the computer treats them identically when interpreting the array element operator, [ ], and the dereference operator, *. (The compiler will generate the exact same machine instructions for all four of the lines given above.) The only difference is that nums is a fixed address determined by the compiler, that cannot be changed while the program is running, whereas iptr is a variable, that can be changed to point to other locations. (iptr refers to a memory location on the stack that holds an address,whereas nums is a constant inserted into the instructions.)

Looping Through Arrays Using Pointers

The following code will print the characters in the array passed to the function, until a null byte is found, and will then print a new line character. (It will make more sense after character arrays are covered.)

```
void printString( const char array[ ] ) { // Same as printString( const char * array )

        char *p = array;

        while( *p )
                printf( "%c", *p++ );
        printf( "\n" );

        return;
}
```

Combinations of * and ++

- *p++ accesses the thing pointed to by p and increments p
- (*p)++ accesses the thing pointed to by p and increments the thing pointed to by p
- *++p increments p first, and then accesses the thing pointed to by p
- ++*p increments the thing pointed to by p first, and then uses it in a larger expression.

Arrays of Pointers

Arrays can hold any data type, including pointers. So the declaration:

int * ipointers[10];

would create an array of 10 pointers, each of which points to an int.

Pointer to pointer

- Addition of pointer variable stored in some other variable is called pointer to pointer variable.

  Or

- Pointer within another pointer is called pointer to pointer.

Syntax:-

```
Data type **p;
int x=22;
int *p=&x;
int **p1=&p;
printf("value of x=%d",x);
printf("value of x=%d",*p);
printf("value of x=%d",*&x);
printf("value of x=%d",**p1);
printf("value of p=%u",&p);
printf("address of p=%u",p1);
printf("address of x=%u",p);
printf("address of p1=%u",&p1);
printf("value of p=%u",p);
printf("value of p=%u",&x);
```

Precedence of dereference (*) Operator and increment operator and decrement operator

The precedence level of difference operator increment or decrement operator is same and their associatively from right to left.

Example :-

int x=25;
int *p=&x;
Let us calculate int y=*p++;
Equivalent to *(p++)
Since the operator associate from right to left, increment operator will applied to the pointer p.

i) int y=*p++; equivalent to *(p++)
p =p++ or p=p+1

ii) *++p;→*(++p)→p=p+1
y=*p

iii) int y=++*p
equivalent to ++(*p)
p=p+1 then *p

iv) y=(*p)++→equivalent to *p++
y=*p then
P=p+1 ;

Since it is postfix increment the value of p.

Structure

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

Structure declaration

```
struct
tagname
{
Data type member1;
Data type member2;
Data type member3;
……
……
Data type member n;
};
OR
struct
{
Data type member1;
Data type member2;

Data type member3;
……
……
Data type member n;
};
OR
struct tagname
{
struct element 1;
struct element 2;
struct element 3;
……
……
struct element n;
};
Structure variable declaration;
struct student
{
int age;
char name[20];
char branch[20];

}; struct student s;
```

Initialization of structure variable

Like primary variables structure variables can also be initialized when they are declared. Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program.

We cant initialize structure members while defining the structure

```
struct student
{
int age=20;
char name[20]=”sona”;
}s1;
```

The above is invalid.
A structure can be initialized as

```
struct student
{
int age,roll;
char name[20];
} struct student s1={16,101,”sona”};
 struct student s2={17,102,”rupa”};
```

If initialiser is less than no.of structure variable, automatically rest values are taken as zero.

Accessing structure elements

Dot operator is used to access the structure elements. Its associativety is from left to right.

```
structure variable ;
s1.name[];
s1.roll;
s1.age;
```

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int roll, age;
char branch;
} s1,s2;
printf(“\n enter roll, age, branch=”);
scanf(“%d %d %c”, &s1.roll, &s1.age, &s1.branch);
s2.roll=s1.roll;
printf(“ students details=\n”);
printf(“%d %d %c”, s1.roll, s1.age, s1.branch);
printf(“%d”, s2.roll);

}
```

Unary, relational, arithmetic, bitwise operators are not allowed within structure variables.

Size of structure

Size of structure can be found out using sizeof() operator with structure variable name or tag name with keyword.

sizeof(struct student); or
sizeof(s1);
sizeof(s2);

Size of structure is different in different machines. So size of whole structure may not be equal to sum of size of its members.

Array of structures

When database of any element is used in huge amount, we prefer Array of structures.

Example:

suppose we want to maintain data base of 200 students, Array of

structures is used.

```
#include<stdio.h>
#include<string.h>
struct student
{

char name[30];
char branch[25];
int roll;
};
void main()
{
struct student s[200];
int i;
s[i].roll=i+1;
printf("\nEnter information of students:");
for(i=0;i<200;i++)
{
printf("\nEnter the roll no:%d\n",s[i].roll);
printf("\nEnter the name:");
scanf("%s",s[i].name);
printf("\nEnter the branch:");
scanf("%s",s[i].branch);
printf("\n");
}
printf("\nDisplaying information of students:\n\n");
for(i=0;i<200;i++)
{
printf("\n\nInformation for roll no%d:\n",i+1);
printf("\nName:");
puts(s[i].name);
printf("\nBranch:");
puts(s[i].branch);
{
}
```

Array within structures

```
struct student
{
char name[30];
int roll,age,marks[5];
}; struct student s[200];
```

We can also initialize using same syntax as in array.

Nested structure

When a structure is within another structure, it is called Nested structure. A structure variable can be a member of another structure and it is represented as

```
struct student
{
element 1;
element 2;
……
……
struct student1
{
member 1;
member 2;
}variable 1;
……..
……..
element n;
}variable 2;
```

It is possible to define structure outside & declare its variable inside other structure.

```
struct date
{
int date,month;
};
struct student
{
char nm[20];
int roll;
struct date d;
}; struct student s1;
 struct student s2,s3;
```

Nested structure may also be initialized at the time of declaration like in above example.

```
struct student s={“name”,200, {date, month}};
 {“ram”,201, {12,11}};
```

Passing entire structure to function

```
#include<stdio.h>
#include<string.h>
struct student
{
char name[30];
int age,roll;
};
display(struct student); //passing entire structure
void main()
{
 struct student s1={”sona”,16,101 };
 struct student s2={”rupa”,17,102 };
display(s1);
display(s2);
}
display(struct student s)
{
printf(“\n name=%s, \n age=%d ,\n roll=%d”, s.name, s.age, s.roll);
}
```

Output: name=sona
roll=16

UNION

Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is

Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

Syntax of union:

union student
{
datatype member1;
datatype member2;
};
Like structure variable, union variable can be declared with definition or separately such as
union union name
{
Datatype member1;
}var1;

Example:- union student s;
Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:-

struct student
struct student
{
int i;
char ch[10];
};struct student s;
Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

Difference between Array and Structure

Arrays	Structure
1. An array is a collection of related data elements of the same type.	1. Structure can have elements of different types
2. An array is a derived data type	2. A structure is a programmer-defined data

	type
3. Any array behaves like a built-in data types. All we have to do is to declare an array variable and use it.	3. But in the case of structure, first, we have to design and declare a data structure before the variable of that type are declared and used.
4. Array allocates static memory and uses index/subscript for accessing elements of the array.	4. Structures allocate dynamic memory and uses (.) operator for accessing the member of a structure.
5. An array is a pointer to the first element of it	5. Structure is not a pointer
6. Element access takes relatively less time.	6. Property access takes relatively large time.

Difference between Structure and Union

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.