Chapter - 4

Structured Data Types : Arrays and Structures.

Objectives :

- **to understand the meaning of structure datatypes and its availability in C++.**
- **To appreciate the use and importance of Arrays in C++**
- **to differentiate between the use and implementation of different types of Arrays**
- **To use structures as User Defined data type to write programs.**
- **To understand and use typedef**

Structured Data types :

Students till now whatever data type we have used are just primitive data types like int , char , float , etc. All these datatypes are defined within the C++ compiler and that is why they are also called as primitive. We can define a length using int , a weight using float , a name using characters etc. but suppose I tell you “please define a fruit for me in program” ,then your mind starts wondering, as you can't define a fruit just with any one datatype as you did with length , weight etc.

A Fruit itself is a composite entity having following attributes :

color	:	can be described with a name i.e. char [ ]
taste	:	can be described with a name i.e. char[ ]
season	:	can be described with int i.e. 1 for summer , 2 for winter …
price	:	can be described as float

ans so on...

This means that to describe a fruit we need to have a collection of data-types bundled togeather so that all the attributes of the fruit can be captured. This is true for any real world thing around you say student , mobile , plant etc. So when we bundle many primitive data types together to define a real world thing then it is known as **derived data type or structured data type or User defined data types.**

In this chapter we would look onto two important structured data types , one is **Array** and the other one is **Structure**.

Sometimes we need to have variables in very large quantities , that too of same data type i.e. suppose we want 200 integer variables. In this situation will you declare 200 individual variables ? Absolutely not. This is because it will :

a) wastage of time as well time taking task
b) we won't be able to manage these 200 variables in our program , it will be difficult to remember the names of variable every now and then during programming.

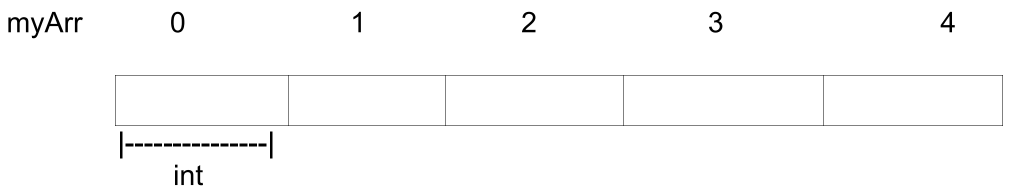
So there exist special structured data type in C++ to tackle this situation. C++ allows a programmer to bundle together all these same type of 200 variable under a same tag name called as Arrays.

So we are observing that **structuring data type means bundling primitive data type in some or other way so that it solves some special programming situations.**

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

For example, an array to contain 5 integer values of type int called myArr could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type int. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

Syntax :
```
<datatype>  array_name [elements];
```

where datatype is a valid type (like int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets [ ]), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called myArr as the one shown in the above diagram it is as simple as:

***int* myArr [5];**

NOTE: The elements field within brackets [ ] which represents the number of elements the array is going to hold, must be a **constant** value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

Initializing arrays.

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example:

***int* myArr [5] = { 16, 2, 77, 40, 12071 };**

This declaration would have created an array like this:

The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [ ]. For example, in the example of array myArr we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

```
int myArr [ ] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array myArr would be 5 ints long, since we have provided 5 initialization values.

Accessing the values of an array.

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

Syntax: `array_name[index]`

Following the previous examples in which myArr had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:

For example, to store the value 75 in the third element of myArr, we could write the following statement:

```
myArr[2] = 75;
```

and, for example, to pass the value of the third element of myArr to a variable called a, we could write:

```
a = myArr[2];
```

Therefore, the expression myArr[2] is for all purposes like a variable of type int.

Notice that the third element of myArr is specified myArr[2], since the first one is myArr[0], the second one is myArr[1], and therefore, the third one is myArr[2]. By this same reason, its last element is myArr[4]. Therefore, if we write myArr[5], we would be accessing the sixth element of myArr and therefore exceeding the size of the array.

In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets [ ] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets [ ] with arrays.

```
int myArr[5];        // declaration of a new array
myArr[2] = 75;       // access to an element of the array.
```

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

```
myArr[0] = a;
myArr[a] = 75;
b = myArr [a+2];
myArr[myArr[a]] = myArr[2] + 5;
```

program 4.1

```
// adding all the elements of an array

#include <iostream>
using namespace std;

int myArr [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    result += myArr[n];
  }
cout << result;
return 0;
}
```

Output : 12206

Dynamic Initialization:
Arrays can also be initialized during runtime. The following program shows how to input values into arrays during rum time :

program 4.2

```
// Inputting value in an array during run-time
#include<iostream.h>
main()
{
        int my_arr[5];      // name of array.
        cout<<"\nEnter values at: ";
        for(int  i = 0 ; i < 5; i++)
        {
                cout<<"\n"<<i+1<<" :";
                cin>>my_arr[ i ];                  //stores value at ith index.
        }
}
```

//program 4.3

// program to store 10 integers and show them.

```
#include<iostream.h>
main()
{
        int my_arr[5];      // name of array.
```

```
                        cout<<"\n"<<i+1<<" :";
                        cin>>my_arr[ i ];                        //stores value at ith index.
                }
                for(int  i = 0 ; i < 10; i++)
                {
                        cout<<"\Number at "<<i+1<<" :"<<my_arr[ i ];    //show value at  ith index.
                }

        }
```

```
//program 4.4
// program to search a particular value out of linear array :
        #include<iostream.h>
        #include<stdio.h>
        #include<process.h>
        main( )
        {
                int a[10] , val = 0;
                cout<<"Input ten integers : ";
                        // inputting array value
                for(int i = 0 ; i<=9 ; i++)
                {
                    cin>> a[i];
                }
                // searching element
                for(int i = 0 ; i<=9 ; i++)
                {
                    if(a[i] == val)
                    {
                        cout<<"Element found at index location : " << i;
                        getch();
                        exit();  // if element is found no need to further search, terminate program
                }
                // if control flow reaches here that means if( ) in the loop was never satisfied
                cout<<"Element not found";
        }
```

```
// program 4.5
// program to find the maximum and minimum of an array :
        #include<iostream.h>
        main( )
        {
                int arr[ ] = {10, 6 , -9 , 17 , 34 , 20 , 34 ,-2 ,92 ,22 };
                int max = min = arr[0];
                for(int i = 0 ; i<= 9 ; i++)
                {
                        if(arr[i] < min)
                          min= arr[i];
                        if(arr[i] > max)
                          max = arr[i];
                }
```

Check your progress :

1. Write a program to store 10 elements and increase its value by 5 and show the array.
2. Write the program to divide each of the array element by 3 and show the array.
3. Write a program to find the average of all elements of an array of size 20.
4. Write a program to find second minimum value out of an array containing 10 elements.

4.2 Strings : Array of characters

Strings are in fact sequences of characters, we can represent them also as plain arrays of char elements terminated by a '\0' character.

For example, the following array:

char myStr [20];

is an array that can store up to 20 elements of type char. It can be represented as:

myStr

'H'	'e'	'l'	'l'	'o'	'\0'														
0	1	2	3	4	…				…				…						19

Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, myStr could store at some point in a program either the sequence "Hello" or the sequence "Merry christmas", since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the null character, whose literal constant can be written as '\0' (backslash, zero).

Our array of 20 elements of type char, called myStr, can be represented storing the characters sequence "Merry Christmas" as:

myStr

'M'	'e'	'r'	'r'	'y'		'C'	'h'	'r'	'i'	's'	't'	'm'	'a'	's'	'\0'	▨	▨	▨	▨
0	1	2	3	4	…				…				…						19

Notice how after the valid content a null character ('\0') has been included in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:

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

In this case we would have declared an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character '\0' at the end.

In the expressions we have used in some examples in previous chapters, constants that represent entire strings of characters have already showed up several times. These are specified enclosing the text to become a string literal between double quotes ("). For example:
"the result is: "   is a constant string literal that we have probably used already.

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

```
        char myword [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
        char myword [ ] = "Hello";
```

In both cases the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming mystext is a char[] variable, expressions within a source code like:

```
                mystext = "Hello";
                mystext[] = "Hello";
```

would **not** be valid, like neither would be:

```
        mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.
Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. In fact, regular string literals have this type (char[]) and can also be used in most cases.

For example, cin and cout support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cin or to insert them into cout. For example:

```
//program 4.6
//null-terminated sequences of characters

        #include <iostream.h>
        int main ()
        {
                char question[ ] = "Please, enter your first name: ";
                char greeting[ ] = "Hello, ";
                char yourname [80];
                cout << question;
        }

        output :  Please, enter your first name: John
                     Hello, John!
```

As you can see, we have declared three arrays of char elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to specify the size of the array: in the first two (question and greeting) the size was implicitly defined by the length of the literal constant they were initialized to. While for yourname we have explicitly specified that it has a size of 80 chars.

```
//Program 4. 7
// program to count total number of vowels present in a string  :
#include<iostream.h>
#include<stdio.h>
int main( )
{
        char string[35]; int count = 0;
        cout<<"Input a string";
        gets(string);               // library function in stdio.h to input a string
        for(int i = 0 ; string[i] != '/0' ; i++)
        {
          if( string[i] == 'a' || string[i] == 'e' || string[i] == 'o' || string[i] == 'u' || string[i] == 'i'
                || string[i] == 'A' ||  string[i] == 'E' || string[i] == 'O' || string[i] == 'U' || string[i] == 'I' )
          {
                count++;
          }
        }
}
```

In the above program the loop is iterated till the character at ith location matches with null character, because after that there is no character left to be scanned. So scanning of characters from first to last character is done using for loop and if vowels are found count keeps incrementing.

String related library functions :

There are few string related library functions in header file string.h which are very useful when we work with strings. They are :

i) **strlen(char[ ]) :** it accepts a string as parameter and returns the length of the string i.e. number of characters within the string. For example strlen("Hurray") will return 6.

ii) **strcpy(char[ ] , char[ ]) :** it accepts two strings as input parameters and then copies the second into the first. For example :

```
char mstr[ ] = "Suresh";
char ystr[20];
strcpy(ystr , mstr);  // copies content of mstr into ystr
puts(mstr);  // prints Suresh
```

iii) **strrev(char[ ]) :** it accepts a string , reverses its content and stores it back into the same string. For example :

```
char myName[ ] = "Kamal";
strrev(myName);
puts(myName) ;    // prints the reversed string as "lamaK"
```

This function returns the value as integer. The integer can be :

0	:	if the two strings are equal
+val	:	if the first string is bigger than the second
-ve	:	if the second string is bigger than the first.

The above comparison is case sensitive , if we want to perform case insensitive comparison then we have to take another version of the function called as strcmpi( )

Example :

```
if( strcmpi("Kamal" , "kamal") != 0);
    cout<<"Not equal";
else
    cout<<"equal";
```

4.3 Multidimensional arrays

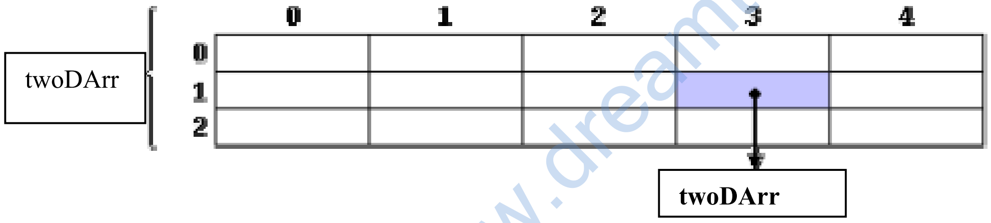
Multidimensional arrays can be described as "arrays of arrays". For example, a two dimensional array can be imagined as a bidimensional table made of elements, all of them of a same uniform data type.

twoDArr represents a bidimensional array of 3 per 5 elements of type int. The way to declare this array in C++ would be:

int twoDArr [3][5];

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

twoDArr[1][3]



(remember that array indices always begin by zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example:

char alpha [100][365][24][60][60];

declares an array with a char element for each second in a alpha, that is more than 3 billion chars. So this declaration would consume more than 3 gigabytes of memory!

Program 4.2
// program to show use of a 2D array

```
#define WIDTH 5
#define HEIGHT 3
```

```
int main ()
{
        for (n=0;n<HEIGHT;n++)
        for (m=0;m<WIDTH;m++)
        {
                twoDArr[n][m]=(n+1)*(m+1);
        }
        return 0;
}
```

We have used "defined constants" (#define macro) to simplify possible future modifications of the program. For example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done simply by changing the line:

```
        #define HEIGHT 3
to:
        #define HEIGHT 4
```

with no need to make any other modifications to the program.

Arrays as parameters
At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function:

```
        void procedure (int arg[])
```

accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

```
        int myarray [40];
```

it would be enough to write a call like this:

```
        procedure (myarray);
```

4.2 Structures In C++

4.2.1 : Why Structures???

To understand the basic need let us proceed with a sample programming scenario.
Problem Definition :

Mr. Chamanlal is Human resource manager in a construction company. He has different responsibilities related to manpower and its utilities. One of his main duties is to keep the record of data related to the Workers who are working in the construction company.

> Name
> Sex
> Age
> Rate per day.

Data members

Chamanlal wants to write a C++ program which can keep track of the information related to all Workers working in the construction company.

<u>Solution :</u> The above problem could be solved in C++ using 3 ways. They are :

> Using Simple variables to store all information of each employees
> Using Arrays to store related information of each employees
> Using C++ Structures

Let us explore all these three techniques to store information of each Worker of the company. Assuming that there are total 20 Workers.

Using Variables :

As we have learned in class XI that any type of data could be stored in a C++ variable having a suitable data type. For example in the given problem the following data types must be used to store information related to a Worker :

Data	Data Type	C++ Declaration	Total Size in bytes
Name	char	char name[45] ;	45 x 1 = 45 bytes
Sex	char	char sex = 'M'	1 x 1 = 1 byte
Age	int	int age = 15 ;	1 x 2 = 2 bytes
Rate per Day	float	float rate = 100.00 ;	1 x 4 = 4 bytes.

So to store the information of 20 Workers one has to declare a total of :
4 x 20 = 80 variables

The variable declarations in the program would be like :
char name1[45] , name2[45] , name3[45] , … , name20[45] ;

char gender1 , gender2 , gender3 , … , gender4 ;

int age1 , age2 , age 3 , … , age20 ;

float rate1 , rate2 , rate3 , … , rate20 ;

Conclusion :

Using the above methodology , we find that as the number of worker increases the number of variables also increases , and thus it becomes difficult task to manage these huge number of variables.

The drawbacks of the previous method of information storage could be managed up to some extent , if instead of taking separate variables for storing related information of Workers (say Age) , we could have used an array of Age , which would store the age of all 20 employees.
Though this method does not provides a economic solution in terms of memory , it provides a better management of Memory locations , where the information about the employees would be stored.
So, four different arrays would be required to keep the related information of Workers i,e :

1. An array to keep the Names of the Workers
2. An array to keep the Gender of the Workers
3. An array to keep the Age of the Workers
4. An array to keep the Rate of the Workers.

1.Array to keep Names :

"Ramu"	"Hari"	"kajri"	…	"Bajrangi"
Name[0]	name[1]	name[2]	…	name[19]

Declaration : **char name[20]45] ;**

2. Array to keep Gender:

'M'	'M'	'F'	…	'M'
Gender[0]	gender[1]	gender[2]	…..	gender[19]

Declaration : **char gender[20];**

3. Array to keep Age:

18	22	24	…	19
age[0]	age[1]	age[2]	…	age[19]

Declaration : **int age[20];**

4. Array to keep Rate:

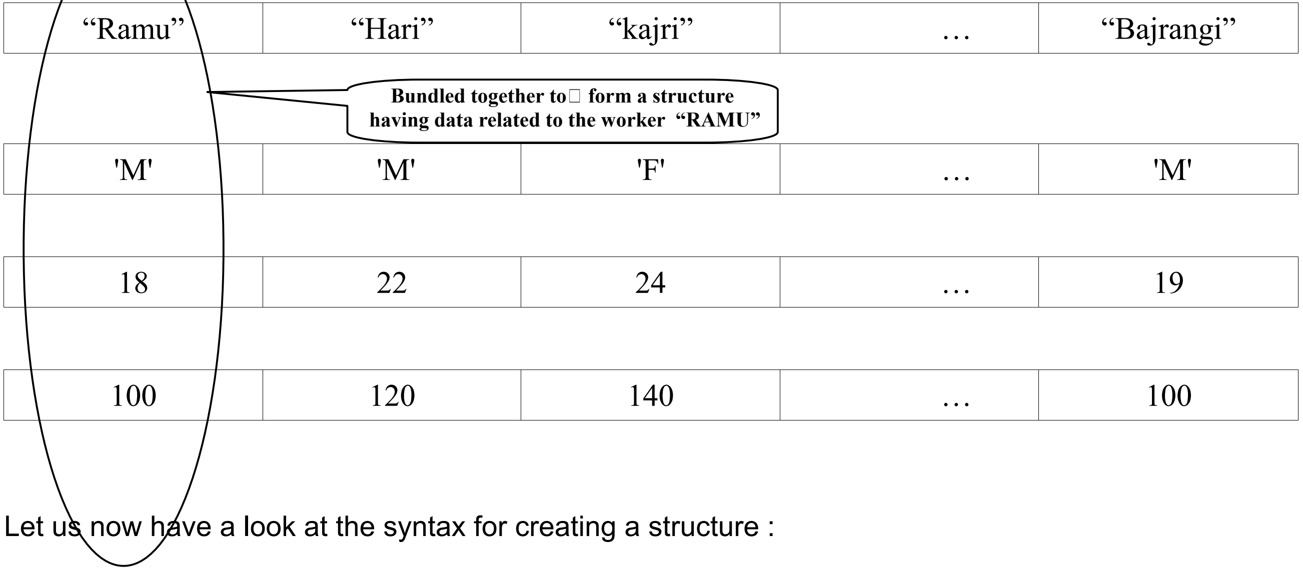
100	120	140	…	100
rate[0]	rate[1]	rate[2]	…	rate[19]

Declaration : **float rate[20];**

Using C++ Structures :

The previous method of storing related data of Workers in different Arrays is a better a solution than storing them in several separate variables. Though it provides better management of information but as the information related to a Worker increases (Say , along with Name , Age , Sex , and Rate , we also want to store Date Of Join , Category , SSN etc.) the need for extra Arrays arises , thus Array management would be another cumbersome issue , if we use Arrays to store information of a Worker.
Thus C++ provides us one of its most fascinating programming construct to handle this situation of storing huge amount of related information about similar entities like WORKER.

To solve the given problem using C++ Structure one has to bundle all the information related to a single WORKER under one single tag name. **This Bundle of information related to an entity,with a Tag Name is called Structure.**


Bundled together to form a structure having data related to the worker "RAMU"

"Ramu"	"Hari"	"kajri"	…	"Bajrangi"
'M'	'M'	'F'	…	'M'
18	22	24	…	19
100	120	140	…	100

Let us now have a look at the syntax for creating a structure :

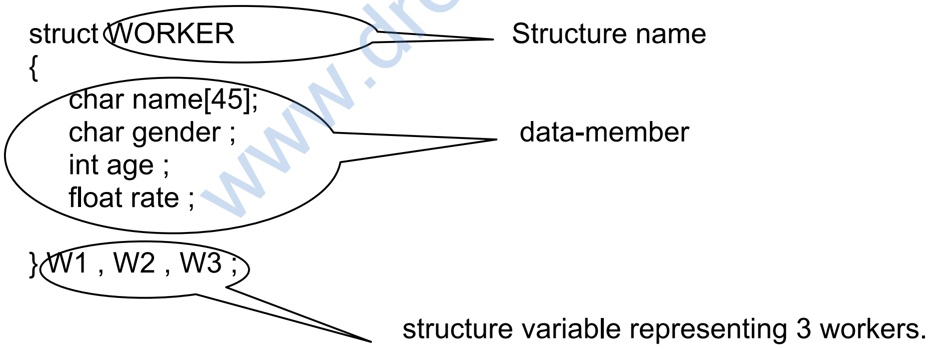
A Structure in C++ is created by using the Keyword struct. The General Syntax for creating a Structure is :

Syntax :

```
struct < Name of Structure >
{
   < datatype > < data-member 1>;
   < datatype > < data-member 2>;
   < datatype > < data-member 3>;
                 …
                 …
   < datatype > < data-member n>;
} [ <variable list > ] ;
```

Example :

A Proper example following the previous syntax could be :



```
struct WORKER
{
   char name[45];
   char gender ;
   int age ;
   float rate ;

} W1 , W2 , W3 ;
```

Points to remember :

1. A structure can declared locally (inside main( )) or globally (Outside main( )).

```
//local declaration
main( )
{
            } W1 , W2 , W3;
                ...
                ...
        }
```

In this case we can create structure variable only within main( )

```
        //Global declaration
        struct WORKER
        {
            char name[45];char gender;...
        };
        main( )
        {
                WORKER W1 , W2 , W3 ;
        ...... }
```

In this case we can create structure variable anywhere in the program

If declared globally , the structure variables could be declared both from inside main( ) and any other place outside main( ) function including any other user defined functions.

If declared locally , the structure variables could be declared only within the scope in which the structure has been defined.

2. No data member should be initialized with any value within the structure declaration. i,e the following type of structure declaration is incorrect and cause error :

```
        struct WORKER
        {
                char name[45];
                gender = 'M';    ←——    Invalid Initialization within structure scope
                age = 16;
                rate = 100.00;
        };
```

4.2.2 : Structure Variable Initialization

Structure variables could be initialized by two ways :

- Static Initialization (During design time)
- Dynamic Initialization (Passing values during Runtime)

Static Initialization :

```
WORKER w1 ; // structure variable declaration

w1.name = "Ramu";
w1.gender = 'M'; // static initializations
w1.age = 17;
w1.rate = 100.00;
```

The (.) operator here is known as component operator, used to access the data members composing the structure variable.

```
WORKER w1 = { “Ramu” , ‘M’ , 17 , 100.00 };
```

Warning : The declaration as well as initialization should be in the same line.

Dynamic Initialization :

```
main( )
{
        WORKER w1 ;
        cout<< “Input Worker’s Name”;cin.getline(w1.name , 45);
        cout<< “Input Worker’s Gender”;cin >> w1.gender ;
        …
}
```

4.2.3 Structure variable assignments

We know that every variable in C++ can be assigned any other variable of same data type i,e :

```
if int a = 7 ; b = 3;
we can write :
        a = b ; // assigning the value of b to variable a
```

Similar is the case with Structure variables also , i,e :

```
if WORKER w1 = {“Ramu” , ‘M’ , 17 , 100.00};
WORKER w2;
we can write :
w2 = w1 ; // assigning the corresponding individual // data member values of w1 to Worker w2;
```

or

```
WORKER w2 = w1;
```

Note : Both structure variables must be of same type i,e WORKER.

There is a member wise copying of member-wise copying from one structure variable into other variable when we are using assignment operator between them.

So, it is concluded that :

```
Writing : w1. name = w1.name ;
w1.gender = w2.gender ;
w1.age = w2.age ;
w1.rate = w2.rate ;
```

is same as :

```
w1 = w2 ;
```

Though we can copy the contents of similar types of structure variables , two dissimilar types of structure variables can’t be assigned to each other, even though they may have same types of constituent data members. Assigning values into dissimilar structure variable would cause a incompatible data type error

Consider the two different structures Student , and Worker :

The Student structure can be written as :

```
struct Student
```

```
          int age ;
          float height;
        };
```

Compared to Worker Structure , we find that , the data types and sequence of all the data members of structure student is same as that of structure Worker, but still we can never write :

Worker w = {“Ramu” , ‘M’ , 20 , 5.5};

So the following assignments are invalid assignments :

```
        Student s = w ;    // Invalid assignment as s and w both are two different structures
or
        Student s ;
        s = w;               //Invalid
```

4.2.4 Array of Structure :

Just like we can make array of integers , floats , chars we can also make array of user defined data types like structures. The syntax to make such an array is :

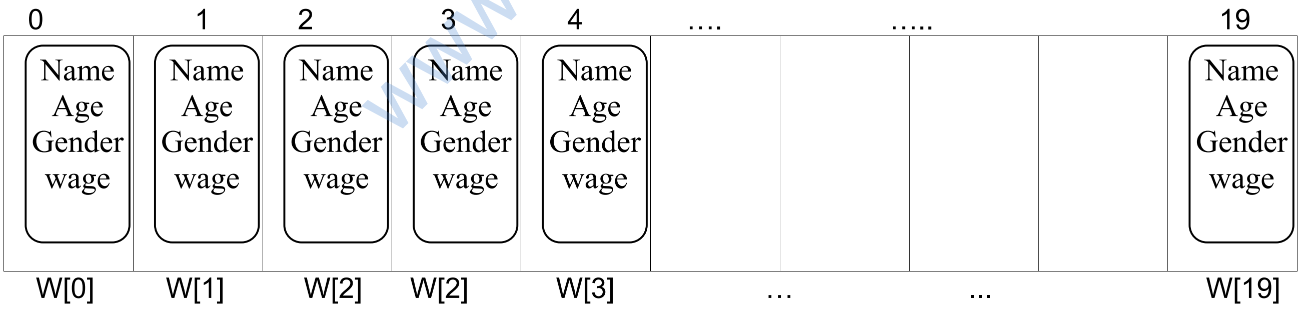
```
                <structure_name> <array_name>[ size ];
```

where : structure_name is the name of the structure which you have created.
array_name is any valid identifier
size is a positive integer constant.

Example : to create array of 20 Workers we can have :

```
                Worker W[20];
```

The above structure could be visualized as :



Each of the elements of the array is itself a structure hence each of them have all the four components.

Initialization of Structure Array :
Array of structures like any other array can be initialized statically and dynamically (runtime).

Static initialization : The above array of Workers W[ ] can be statically initialized as :

```
                    { "Ramu" , 'M' , 17 , 100.00 },
                    { "Hari" , 'M' , 22 , 120.00 },
                    { "kajri" , 'F' , 18 , 100.00 },

                      …              //values of other 16 workers.
                      …
                    {"Bajrangi" , 'M' , 24  , 140.00 }
            };
```

Note : The data-member values must be supplied in the same order of the data-member described in structure definition , otherwise datatype mismatch error may cause.

Dynamic Initialization :

```
#include<iostream.h>
#include<stdio.h>
main( )
{
        Worker  W[20] ;       // consider the Worker structure declared earlier.
[redacted]
}
```

4.2.5 : Function with Structure variable:

Just like we can pass various primitive datatypes to a function we can also pass a structure variable as function parameter / arguments. The benefit is that the structure carries a bundled information to the structure. The prototype of such a function would be :

```
<return_type> function_name(<structure_name> <var> , … , … );
```

Let us understand the concept with following program :

```
//program  4.8
// program to write a function which increases the wage of a female worker by passed percent

        void increaseWage(   Worker  & w ,  float incr )
        {
                if( w.gender == 'F' || w.gender == 'f' )
                {
```

}
Look at the highlighted parameter, we have passed formal structure variable as reference, so that the increment is reflected back in actual parameter.

Similarly , if we don't want to pass our structure variable as reference we can do so , but then we have to return the modified structure variable back. This can be achieved in above function, if we take return type as structure name. Let us modify the function so that it returns a structure :

```
Worker increaseWage( Worker w , float incr)
{
        if( w.gender == 'F' || w.gender == 'f' )
        {
                w.wage + = w.wage* (incr /100) ;
        }
  return w ;
}
```

4.2.6 : Use of typedef statement :

A typedef can be used to indicate how a variable represents something. This means that we are defining another name for an existing datatype. Let us see the following example to understand this.

Suppose we are going to write a program in C++ where we need lot of strings of size 30 say. This means that every time we have to declare character array of size 30 , like char arr[30] or so. Instead of declaring each time with size specification if we have a declaration like :

string arr ;

which automatically instructs the compiler that arr will be a character array of size 30 , then this will give a good look to our program. We can achieve this by using typedef statement as done in the program given below :

```
#include<iostream.h>
typedef char  string[30] ;
main( )
{
        string name ;
        cout<<"Input your name";
        gets(name);
        cout<<"You name has  "<< strlen(name)<<"ciharacters";
}
```

The second line defines a new datatype named as string which is nothing but array of 30 characters.

Check your Progress :

Q1.   consider the structure distance given below :

```
struct Distance
{
        int feets , inches; };
```

ii) declare a variable of type Distance and initialize it statically.
iii) Can we use = operator between two Distance type variables. If yes, what does it infers?
iv) write a function which takes two Distance variable as parameters and then returns their sum. For example say D1 = {9 , 6 } and D2 = {5 , 8 } , then D1 + D2 = 15 feet 2 inches

v) Declare an array representing 10 Distances , and then write a program to initialize all these 10 distances and then increase all the distances by 5 feets.

Q2. Consider the structure called Point as :

```
struct Point
{
        int xCoord;
        int yCoord;
};
```

Write a function which accepts three Points as parameter and then checks whether the three points are collinear.

Q3. Consider the following structure called Element and Compound :

```
struct Element
{
        int atomic_no;
        float atomic_mass;
        char symbol[2];
        int valency;
};
struct Compound
{
        Element Elements[ 5 ];  // array to keep all elements present in the compound.
        float molecular_mass;
};
```

Write a program to initialize a compound and show its details.

Summary :

1. Structured data types are used in special programming situations like arrays are used when there is bulk of similar type of variables to deal with, whereas structures models a real life entity having some attribute.

2. Arrays can have both primitive as well user defined data types. There are basically two types of arrays , one dimensional and multidimensional (2 dimensional)

3. Functions can be passed structured datatypes as its parameter as well as they can return them.

4. typedef is a special keyword to define a another name for a variable definition.