Chapter 1

Structures and Pointers

Structures

Structure is a user defined data type to represent logically related data items, which may be of different types, under a common name.

Structure definition

Syntax

```
struct  structure_tag
{
        data_type   variable1;
        data_type   variable2;
        ……..;
        ……..;
        data_type   variableN;
};
```

Here struct is a keyword to define a structure, structure_tag is an identifier and variable1,variable2,…variableN are identifiers to represent data items

The identifier used as the structure_tag or structure_name is the new user defined data type.It has a size and can be used to declare variables.

The data represented by a structure is known as grouped data or aggregate data or compound data. The data types of data items inside a structure may be basic data types or user defined data types.

Eg 1.

```
struct date
{
    int dd;
    int mm;
    int yy;
};
```

Eg 2.

```
struct strdate
{
    int day;
    char month[10];
    int year;
};
```

Eg 3.

```
struct student
{
    int adm_no;
    char name[20];
    char group[10];
    float fee;
};
```

Variable declaration and memory allocation

Syntax

Struct structure_tag var1,var2,…,varN;

OR

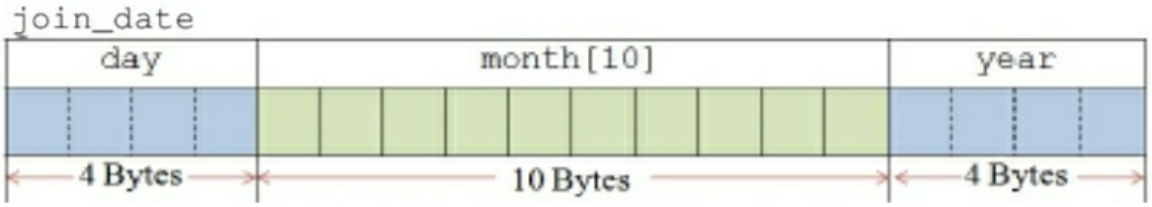
structure_tag var1,var2,….,varN;

Eg.

date dob,today; OR struct date dob,today;

strdate adm_date, join_date;

Variable declaration causes memory allocation. The size of structure data type will be the sum of the sizes of data types of each elements of the structure.



The memory allocation of the structure variable requires 18 Bytes of memory

A structure variable can be declared along with the definition also

Eg.

```
struct complex
{
    short real;
    short imaginary;
} c1, c2;
```

If we declare structure variables along with the definition, structure tag (or structure name) can be avoided

```
struct
{
    int a,b,c;
} eqn1, eqn2;
```

Variable Initialisation

A structure variable can be initialized as follows

Syntax

```
structure_tag  variable  = {value1, value2,……,valueN};
```

Eg. student s = {3452,”Vaishakh”,”Science”,270.00};

A structure variable can be assigned with the values of another structure variable. But both of them should be of the same structure type.

Eg. student st = s;

s

adm_no	name	group	fee
3452	Vaishakh	Science	270.00

st

adm_no	name	group	fee
3452	Vaishakh	Science	270.00

Accessing elements of structure

The period symbol (**.**), named as dot operator is used for accessing elements of a structure variable.

Syntax

structure_variable.element_name

Eg.

today.dd = 10;

strcpy(adm_date.month, “June”);

cin>>s1.admno;

cout<<c1.real + c2.real;

Write a program to find the total score of a student

one.cpp

```cpp
#include<iostream>
using namespace std;
struct student
{
    int reg_no;
    char name[20];
    short ce;
    short pe;
    short te;
};
int main()
{
    student s;
    int tot_score;
    cout<<"Enter register number:";
    cin>>s.reg_no;
    cout<<"Enter name:";
    cin>>ws;
    cin.getline(s.name,20);
    cout<<"Enter scores in CE, PE and TE:";
    cin>>s.ce>>s.pe>>s.te;
    tot_score=s.ce+s.pe+s.te;
    cout<<"\nRegister Number:"<<s.reg_no;
    cout<<"\nName of Student:"<<s.name;
    cout<<"\nCE Score:"<<s.ce<<"\tPE Score:"<<s.pe<<"\tTE Score:"<<s.te;
    cout<<"\nTotal Score:"<<tot_score;
    return 0;
}
```

Output window:

```
Enter register number: 23545
Enter name: Deepika Vijay
Enter scores in CE, PE and TE: 19  38  54

Register Number: 23545
Name of Student: Deepika Vijay
CE Score: 19 PE Score: 38    TE Score: 54
Total Score    : 111
```

To store the details of more than one student, array of structures is used.

Nested Structure

When an element of a structure may itself be another structure, Such a structure is known as nested structure

Definition A	Definition B

struct date
{
short day;
short month;
short year;
};
struct student
{
int adm_no;
char name[20];
date dt_adm;
float fee;
};
	struct student
	{
	int adm_no;
	char name[20];
	struct date
	{
	short day;
	short month;
	short year;
	} dt_adm;
	float fee;
	};

Example for nested structure variable Initialisation and accessing of elements of nested structure variable.

student s = {4325,"Vishal",{10,11,1997},575};

cout<<s.admno<<s.name

cout<<s.dt_adm.day<<"/"<<s.dt_adm.month<<"/"<<s.dt_adm.year;

format for accessing inner structure element is

outer_structure_variable.inner_structure_variable.element

Array Vs Structure

The following are the comparison between these two data types

Arrays	Structures
• It is a derived data type. • A collection of same type of data. • Elements of an array are referenced using the corresponding subscripts. • When an element of an array becomes another array, multi-dimensional array is formed. • Array of structures is possible.	• It is a user-defined data type • A collection of different types of data. • Elements of structure are referenced using dot operator (.) • When an element of a structure becomes another structure, nested structure is formed. • Structure can contain arrays as elements

Pointers

Pointer is a variable that can hold the address of a memory location

Other definitions :

Pointer is a variable that points to a memory location. Pointer is a derived data type.

When more than one cell constitute a single storage location, the address of the first cell will be the address of the storage location

(base address).

Declaration of Pointer Variable

Syntax

```
data_type  *variable;
```

The data_type can be fundamental or user-defined and variable is an identifier.

Eg. int *ptr1;

```
float  *ptr2;

struct student  *ptr3;
```

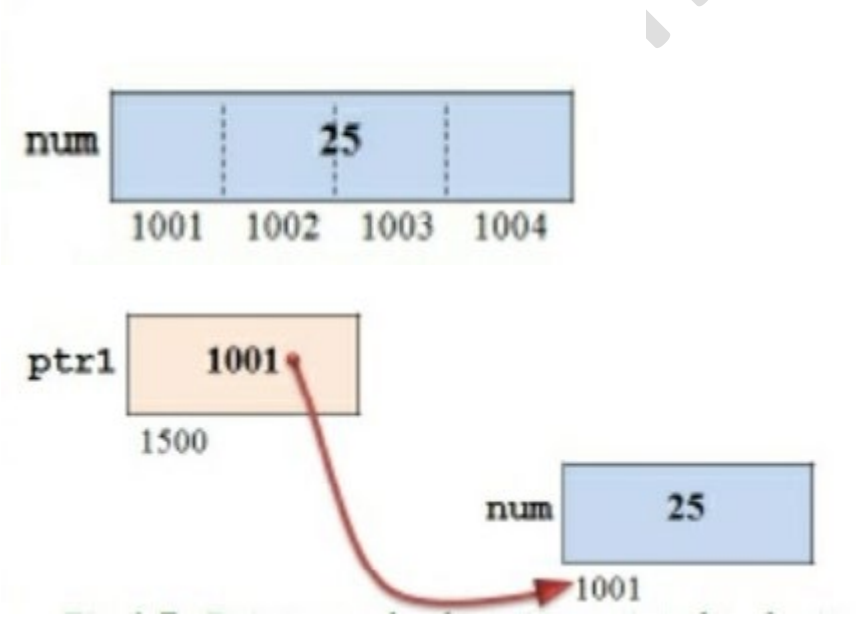
The data type of a pointer should be the same as that of the data pointed to by it. Usually, the size of a pointer in C++ is 2 to 4 bytes.

The operators & and *

address of operator (&) is used to get the address of a variable.

```
int  num =25;

ptr1 = &num;
```

The following is the picture representation of memory allocation of the variable num and the working of address of operator (&).Here the address 1001 is stored in the pointer ptr1.



Value at operator (*) is used to get the value at the address, stored in the pointer variable. It is also known as **indirection or dereference operator** .

Eg. cout <<*ptr1; the result of this statement will be same as

the result of cout<<num;

The operators address of (&) and indirection (*) are unary operators. The & operator can be used with any kind of variable since every variable is associated with a memory address. But, the * operator can be used only with pointers.

```
cout<< &num; // 1001 (address of num) will be the output
cout<< ptr1; // 1001 (content of ptr1) will be the output
cout<< num;  // 25 (content of num) will be the output
cout<< *ptr1;/* 25 (value in the location pointed to by
                    ptr1) will be the output */
cout<< &ptr1;// 1500 (address of ptr1) will be the output
cout<< *num; // Error!! num is not a pointer
```

Methods of memory allocation

There are two types of memory allocation

Static memory allocation

In this memory allocation the memory allocation takes place before the execution of the program. The amount of memory to be allocated is known in advance.

Eg. Variable declaration statement

Dynamic memory allocation

In this memory allocation the memory allocation takes place during the execution of the program. The amount of memory to be allocated is not known in advance.

In C++ dynamic memory allocation is facilitated by an operator, named new. de-allocation is facilitated by delete operator.

Dynamic operators – new and delete

The operator new is a keyword. It is a unary operator and the required operand is either a fundamental or user-defined data type.

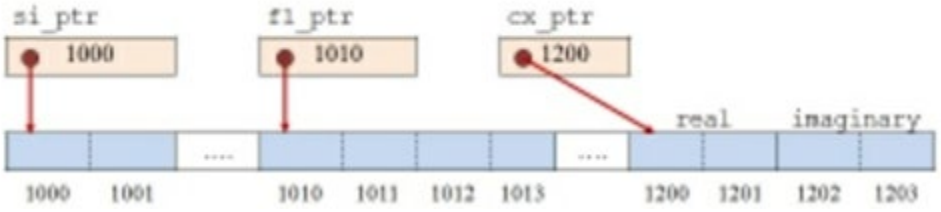
Naturally it returns a value and this value will be the address of a location. The size of this location will be the same as that of the data type used as the operand.

Syntax

```
pointer_variable  =  new data_type;
```

Eg.

```
short  *si_ptr;

float   *fi_ptr;

struct complex  *cx_ptr;

si_ptr = new short;

fi_ptr  = new float;

cx_ptr = new complex;
```



Dynamically allocated memory cannot be referred by an ordinary variable. It can be accessed using value at (*) operator.

```
*si_ptr  =  247;

 cin>>*fi_ptr;
```

Dynamically allocated memory location can also be initialized using the following syntax also

```
pointer_variable  =  new data_type(value);
```

Eg.
```
si_ptr = new short(0);

fi_ptr  = new float(3.14);
```

Once memory is allocated dynamically using new operator, it should be de-allocated or released before exiting the program. C++ provides delete operator for this.

Synatx

```
delete pointer_variable;
```

Eg.
```
delete  si_ptr;

delete  fi_ptr;
```

Memory Leak

If the memory allocated using new operator is not freed using delete, then a block of memory that is left unused but not released for further allocation. Thus a part of memory seems to disappear on every run of the program. This situation is known as memory leak.

Reasons for memory leak :-

Forgetting to delete the memory that has been allocated dynamically

Failing to execute the delete statement due to poor logic of the program code

Assigning the address returned by new operator to a pointer that was already pointing to an allocated object

Remedy for memory leak :-

Ensure that memory allocated through new is properly de-allocated through delete

Operations on Pointers

Arithmetic Operations on Pointers

cout<<si_ptr + 1;

cout <<fi_ptr + 1;

Here the result will be 1002 and 1014 respectively , So when 1 is added to a short int type pointer, actually its size (i.e. 2) is to be added to the address contained in the pointer variable. Similarly , to add 1 to float type pointer, its size (i.e., 4) is to be added to the address.

```
int *ptr1, *ptr2; // Declaration of two integer pointers
ptr1 = new int(5); /*  Dynamic memory allocation (let the
             address be 1000)and initialisation with 5*/
ptr2 = ptr1 + 1;  /*   ptr2 will point to the very next
             integer location with the address 1004 */
++ptr2;           //   Same as ptr2 = ptr2 + 1
```

```
cout<< ptr1;        //    Displays 1000
cout<< *ptr1;       //    Displays 5

cout<< ptr2;        //    Displays 1008

cin >> *ptr2;       /*    Reads an integer (say 12) and stores it in
                          location 1008  */

cout<< *ptr1 + 1;   //    Displays 6 (5 + 1)

cout<< *(ptr1 + 2); //    Displays 12, the value at 1008

ptr1- - ;           //    Same as ptr1 = ptr1 – 1;
```

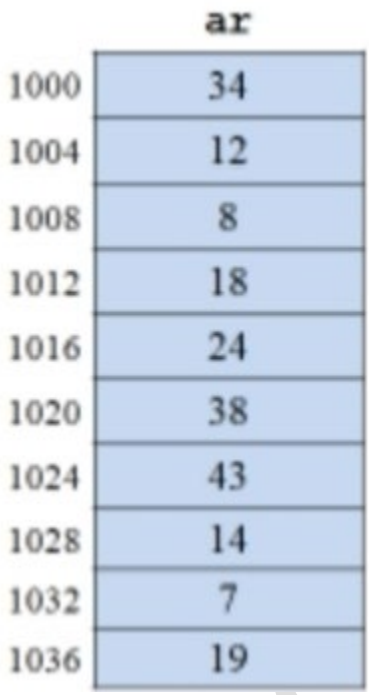
Pointers are only incremented or decremented

Relational operations on pointers

Only == (equality) and != (non-equality) operators are used with pointers

Pointers and Array

An array can contain a collection of homogeneous type of data under a common name. This data is stored in contiguous memory locations .



ptr = &ar[0];

```
cout<<ptr;    //Displays 1000, the address of ar[0]
cout<<*ptr;   //Displays 34, the value of ar[0]
cout<<(ptr+1);  //Displays 1004, the address of ar[1]
cout<<*(ptr+1); //Displays 12, the value of ar[1]
cout<<(ptr+9);  //Displays 1036, the address of ar[9]
cout<<*(ptr+9); //Displays 19, the value of ar[9]
```

Array name ar can be considered as a pointer. So the following statements are also valid.

```
cout<<ar;     //Displays 1000, the address of ar[0]
ptr=ar;       //same as ptr=&ar[0];
cout<<*ar;    //Displays 34, and is same as cout<<ar[0];
cout<<(ar+1); //Displays 1004, the address of ar[1];
cout<<*(ar+1);  //Displays 34, and is same as cout<<ar[1];
```

Array name always contains the base address of the array, and it can not be changed

Dynamic Array

It is created during run time using the dynamic memory allocation operator new.

Syntax

pointer = new data_type [size];

Here, the size can be a constant, a variable or an integer expression.

Write a program to find the highest percent of pass in schools

```
#include <iostream>
using namespace std;
int main()
{
    float *pass, max;
    int i, n;
    cout<<"Enter the number of schools: ";
    cin>>n;   //To input number of schools
    pass = new float[n];//dynamic array having n elements
```

```
    for (i=0; i<n; i++)
    {
        cout<<"Percent of pass by school "<<i+1<<": ";
        cin>>pass[i]; //Concept of subscripted variable
    }
    max=pass[0];
    for (i=1; i<n; i++)
        if (pass[i]>max) max = *(pass+i);
    /* Elements are accessed using subscript and pointer
    arithmetic operation */
    cout<<"Highest percent is "<<max;
    return 0;
}
```

Output:

```
Enter the number of schools: 5
Percent of pass by school 1: 75.6
Percent of pass by school 2: 66.5
Percent of pass by school 3: 89.3
Percent of pass by school 4: 71
Percent of pass by school 5: 70.6
Highest percent is 89.3
```

Pointer and String

```
char str[20];      //character array declaration
char *sp;          //character pointer declaration
cin>>str;          //To input a string, say "Program"
cout<<str;         //Displays the string "Program"
sp=str;   //Content of str is copied into the pointer sp
cout<<sp; //Displays the string "Program"
cout<<&str[0];     //Displays the string "Program"
cout<<sp+1;        //Displays the string "rogram"
cout<<&str+1;      //Displays the string "rogram"
/* The two statements given above display the substring
starting from 2nd character onwards */
cout<<str[0];      //Displays the character 'P'
cout<<*sp;         //Displays the character 'P'
cout<<&str; //Displays the base address of the array str
cout<<&sp;  //Displays the address of the pointer sp
```

From the above statements, It proves that

A character pointer can be used to store a string and this pointer can be considered as a string variable.

Another aspect is, the statement cout<<&str[0]; will display the entire string instead of the address of the first location. i.e., if we access the address of a string data, we get the string itself.

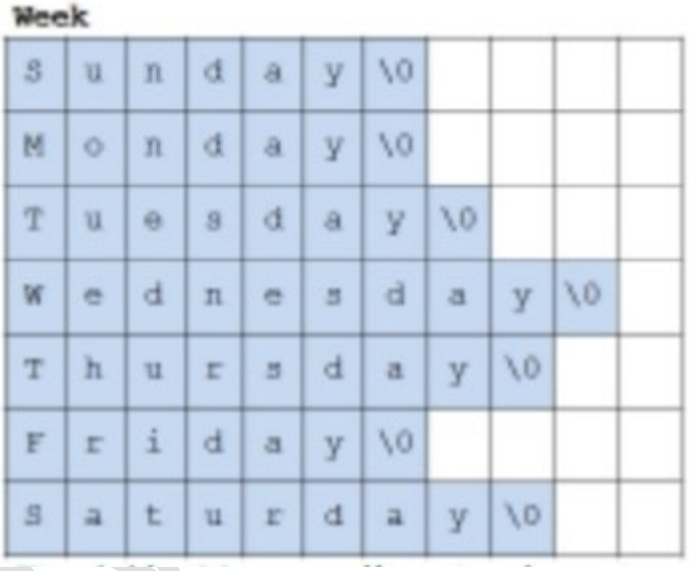
Advantages of character pointer

- There is no memory wastage
- Assignment operator (=) can be used to copy strings
- Any character in the string can be referenced using the concept of pointer arithmetic which makes access faster
- Array of strings can be managed with optimal use of memory space.

Array of Strings.

An array of character pointers is used to store Array of Strings.

```
char  * name [ 7 ] ;

char  * week [ 7 ] = {“Sunday”, “Monday”, “Tuesday”,
                      “Wednesday”, “Thursday”,
                      “Friday”, “Saturday”};
```



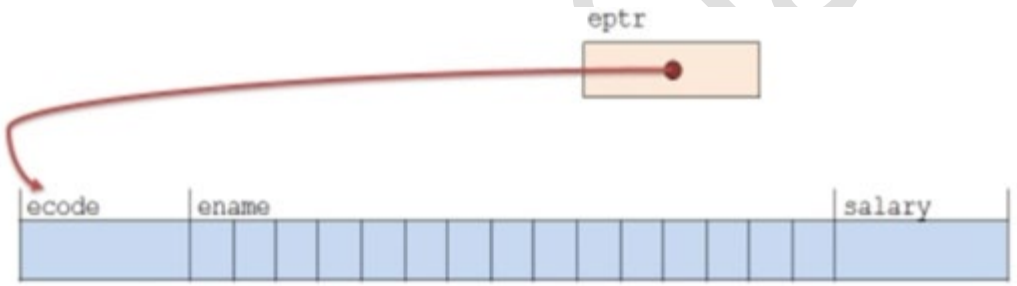

```
for ( i=0; i<7; i++)

cout<<name[ i ];
```

Pointer and Structure

```
struct  employee

{

    int  ecode;
```

```
    char ename[15];
    float  salary;
};
employee  *eptr;
eptr  =  new employee;
```

In the above code a structure employee is defined with the data items ecode, ename and salary and a structure pointer eptr, of data type employee is declared and a memory of 23 bytes is dynamically allocated and its base address is stored in eptr.



Structure elements can be accessed through structure pointers using arrow operator (->). It is constituted by a hyphen (-) followed by greater than symbol (>).

```
eptr->ecode = 657346;   //Assigns an employee code
gets(eptr->ename);      //inputs the name of an employee
cin>> eptr->salaray;    //inputs the salary of an employee
cout<< eptr->salary * 0.12; //Displays 12% of the salary
```

Let us modify the structure employee as

```
struct  employee
{
```

```
    int  ecode;
    char ename[15];
    float  salary;
    int  *ip;
};
```

```
eptr->ip = new int(5); /* Dynamic allocation for integer
                          and initialiastion with 5 */
cout << *(eptr->ip);   // Displaying the value 5
int n = eptr->*ip+1;   // Adding 1 to 5 and stores it in n
```

The value pointed to by ip can be referenced in two ways *(eptr -> ip) and eptr ->*ip. A structure can contain pointer of any data type as an element. Even it may be of the same structure data type.

```
struct  employee
{
    int  ecode;
    char ename[15];
    float  salary;
    employee  *ep;
};
```

This type of structure is known as self referential structure.

Self referential structure

A structure in which one of the element is a pointer to the same structure is known as self referential structure. It helps to develop dynamic data structures like linked list, tree etc in C++.