1 Structures and Pointers

Significant Learning Outcomes

After the completion of this chapter, the learner

- identifies the need of user-defined data types and uses structures to represent grouped data.
- creates structure data types and accesses elements to refer to the data items.
- uses nested structures to represent data consisting of elementary data items and grouped data items.
- develops C++ programs using structure data types for solving real life problems.
- explains the concept of pointer and uses pointer with the operators & and *.
- compares the two types of memory allocations and uses dynamic operators `new` and `delete`.
- illustrates the operations on pointers and predicts the outputs.
- establishes the relationship between pointer and array.
- uses pointers to handle strings.
- explains the concept of self referential structures.

We started writing C++ programs in Class XI for solving problems. Almost all problems are related to the processing of different types of data. Last year we came across only elementary data items such as integers, fractional numbers, characters and strings. We used variables to refer to these data and the variables are declared using basic data types of C++. We know that all data is not of fundamental (basic) types; rather many of them may be composed of elementary data items. No programming language can provide data types for all kinds of data. So programming languages provide facility to define new data types, as desired by the users. In this chapter, we will discuss such a user-defined data type named structure. This chapter also discusses a new kind of variable known as pointer. The concept of pointers is a typical feature of languages like C, C++. It helps to access memory locations by specifying the memory addresses directly, which makes execution faster. A good understanding of this concept will help us to design data structure applications and system level programs.

Last year you might have used GNU Compiler Collection (GCC) with Geany IDE or Turbo C++ IDE for developing C++ programs.

GCC differs from Turbo C++ IDE in the structure of source code, the way of including header files, the size of int and short, etc. In this chapter the concepts are presented based on GCC.

1.1 Structures

Now-a-days students, employees, professionals, etc. wear identity cards issued by institutions or organisations. Figure 1.1 shows the identity card of a student. The first column of Table 1.1 contains some of the data printed on the card. You try to fill up the second column with the appropriate C++ data types discussed in Class XI.

Fig.1.1: ID card of a student

Data	C++ data type
12345 Sneha S. Raj 20/02/1997 O+ve Snehanilayam, Gandhi Nagar, Chemmanavattom, Pin 685 531	

Table 1.1: Data and C++ data types

You may use `short` or `int` for admission number (12345), `char` array for name (Sneha S. Raj), blood group (O +ve) and even address. Sometimes you may not be able to identify the most appropriate data types for date of birth and address. Let us consider the data 20/02/1997 and analyse the composition of this data. It is composed of three data items namely day number (10), month number (02) and year (1997). In some cases, the name of the month may be used instead of month number. Address can also be viewed as a composition of data items such as house number/name, place, district, state and PIN code. Even the entire details on the identity card can be considered as a single unit of data. Such data is known as grouped data (or aggregate data or compound data). C++ provides facility to define new data types by which such aggregate or grouped data can be represented. The data types defined by user to represent data of aggregate nature are generally known as user-defined data types.

Structure is a user-defined data type of C++ to represent a collection of logically related data items, which may be of different types, under a common name. We learnt array in Class XI, to refer to a collection of data of the same type. But structure

can represent a group of different types of data under a common name. Let us discuss how a structure is defined in C++ and elements are referenced.

1.1.1 Structure definition

While solving problems, the data to be processed may be of grouped type as mentioned above. We have to define a suitable structure to represent such grouped data. For that, first we have to identify the elementary data items that constitute the grouped data. Then we have to adopt the following syntax to define the structure.

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

In the above syntax, **`struct`** is the keyword to define a structure, `structure_tag` (or `structure_name`) is an identifier and `variable1`, `variable2`, ..., `variableN` are identifiers to represent the data items constituting the grouped data. The identifier used as the structure tag or structure name is the new user-defined data type. It has a size like any other data types and it can be used to declare variables. This data type can also be used to specify the arguments of functions and as the return type of functions. The variables specified within the pair of braces are known as elements of the structure. The data types preceded by these elements may be basic data types or user-defined data types. These data types determine the size of the structure.

Now let us define a structure to represent dates of the format 20/02/1997 (as seen in the ID card). We can see that this format of date is constituted by three integers which can be represented by `int` data type of C++. The following is the structure definition for this format of date:

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

Here, `date` is the structure tag (structure name), and `dd`, `mm` and `yy` are the elements of the structure `date`, all of them are of `int` type. If we want to specify the month as a string (like January instead of 1), the definition can be modified as:

```
struct strdate
{
    int day;
    char month[10];      // name of month is a string
    int year;
};
```

While writing programs for solving problems, some of the data involved may be logically related in one way or the other. In such cases, the concept of structure data type can be utilised effectively to combine the data under a common name to represent data compactly. For example, the student details such as admission number, name, group, fee, etc. are logically related and hence a structure can be defined as follows:

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

Let us do

Now, try to define separate structures yourself to represent **address** and **blood group**. Blood group consists of group name and rh value.

We know that the details of an employee may consist of employee code, name, gender, designation and salary. Define a suitable structure to represent these details.

We have discussed the way of defining a structure type data. Can we now store data in it? No, it is simply a data type definition. Using this data type we should declare a variable to store data.

1.1.2 Variable declaration and memory allocation

As in the case of basic data items, a variable is required to refer to a group of data. Once we have defined a structure data type, variable is declared using the following syntax:

```
struct structure_tag var1, var2, ..., varN;
```

OR

```
structure_tag var1, var2, ..., varN;
```

In the syntax, `structure_tag` is the name of the structure and `var1`, `var2`, ..., `varN` are the structure variables. Let us declare variables to store some dates using the structure `date` and `fulldate`.

```
date dob, today;          OR    struct date dob, today;
strdate adm_date, join_date;
```

We know that variable declaration statement causes memory allocation as per the size of the data type. What will be the size of a structure data type? Since it is user-defined, the size depends upon the definition of the structure. The definition of `date` shows that its variables require 12 bytes each because it contains three `int` type elements (size of `int` in GCC is 4 bytes). The memory allocation for the variable `join_date` of `strdate` type is shown in Figure 1.2.

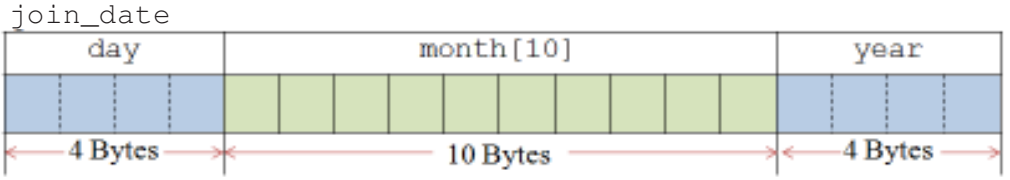

Fig. 1.2: Memory allocation for a structure variable

The size of `int` type in GCC is 4 bytes and in Turbo IDE, it is 2 bytes. In Figure 1.2, the elements day and year are provided with 4 bytes of memory since we follow GCC. Since this much memory is not required for storing values in these elements, it is better to replace int with `short`.

The variable `join_date` consists of three elements `day`, `month` and `year`. These elements require 4 bytes, 10 bytes and 4 bytes, respectively and hence the memory space for `join_date` is 18 bytes.

Let us do

Now you find the size of the structure `student` defined earlier. Also write the C++ statement to declare a variable to refer to the details of a student and draw the layout of memory allocated to this variable.

A structure variable can be declared along with the definition also, as shown below:

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

This structure, named `complex` can represent complex numbers. The identifiers `c1` and `c2` are two structure variables, each of which can be used to refer to a

complex number. If we declare structure variables along with the definition, structure tag (or structure name) can be avoided. The following statement declares structure variables along with the definition.

```
struct
{
    int a, b, c;
}eqn_1, eqn_2;
```

There is a limitation in this type of definition cum declaration. If we want to declare variables, to define functions, or to specify arguments using this structure later in the program, it is not possible since there is no tag to refer. The above structure also shows that if the elements (or members) of the structure are of the same type, they can be specified in a single statement.

Variable initialisation

During the declaration of variables, they can be assigned with some values. This is true in the case of structure variables also. When we declare a structure variable, it can be initialised as follows:

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

For example, the details of a student can be stored in a variable during its declaration itself as shown below:

```
student s={3452, "Vaishakh", "Science", 270.00};
```

The values will be assigned to the elements of structure variable `s` in the order of their position in the definition. So, care should be given to the order of appearance of the values. The above statement allocates 38 bytes of memory space for variable `s`, and assigns the values 3452, "Vaishakh", "Science" and 270.00 to the elements `adm_no`, `name`, `group` and `fee` of `s`, respectively.

If we do not provide values for all the elements, the given values will be assigned to the elements on First Come First Served (FCFS) basis. The remaining elements will be assigned with 0 (zero) or '\0' (null character) depending on numeric or string.

A structure variable can be assigned with the values of another structure variable. But both of them should be of the same structure type. The following is a valid assignment:

```
student st = s;
```

This statement initialises the variable `st` with the values available in `s`. Figure 1.3 shows this assignment:

s

adm_no	name	group	fee
3452	Vaishakh	Science	270.00

st

adm_no	name	group	fee
3452	Vaishakh	Science	270.00

Fig. 1.3: Structure assignment

While defining a structure, the elements specified in it cannot be assigned with initial values. Though the elements are specified using the syntax of variable declaration, memory is not allocated for the structure definition statement. Hence, values cannot be assigned to them. The structure definition can be considered as the blue print of a house. It shows the number of rooms, each with a specific name and size. But nothing can be stored in these rooms. The total space of the building will be the sum of the spaces of all rooms in the house. Any number of houses can be constructed based on this plan. All of them will be the same in terms of number of rooms and space, but each house will be given different name. Structure definition is the blue print and structure variables are the realisation of the blue print (similar to the construction of houses based on the blue print). Each variable can store data in its elements (similar to the placing of furniture, house-hold items and residents in rooms).

1.1.3 Accessing elements of structure

We know that array is a collection of elements and these elements are accessed using the subscripts. Structure is also a collection of elements and C++ allows the accessibility to the elements individually. The period symbol (`.`) is provided as the operator for this purpose and it is named ***dot operator***. The dot operator (`.`) connects a structure variable and its element using the following syntax:

```
structure_variable.element_name
```

In programs, the operations on the structure data can be expressed only by referring the elements of the structure. The following are some examples for accessing the elements:

```
today.dd = 10;
strcpy(adm_date.month, "June");
cin >> s1.adm_no;
cout << c1.real + c2.real;
```

But the expression `c1+c2` is not possible, since the operator + can be used with numeric data types only.

Let us discuss an interesting fact about assignment operation on structure variables. Two structures are defined as follows:

```
struct test_1
{
    int a;
    float b;
}t1={3, 2.5};
```

```
struct test_2
{
    int a;
    float b;
}t2;
```

The elements of both the structures are the same in number, name and type. The structure variable `t1` of type `test_1` is initialised with 3 and 2.5 for its `a` and `b`. But the assignment statement: `t2=t1;` is invalid, because `t1` and `t2` are of different types, i.e., `test_1` and `test_2`, respectively. But if we want to copy the values of `t1` into `t2`, the following method can be adopted:

```
t2.a = t1.a;        t2.b = t1.b;
```

It is possible because we are assigning an `int` type data into another `int` type variable.

Now let us write a program to implement the concepts discussed so far. We define a structure `student` to represent register number, name and scores awarded in continuous evaluation (CE), practical evaluation (PE) and term-end evaluation (TE). The details are input and the total score as part of Continuous and Comprehensive Evaluation (CCE) is displayed.

Program 1.1: To find the total score of a student

```
#include <iostream>
#include <cstdio>     //To use gets() function
using namespace std;
struct student  //structure definition begins
{
    int reg_no;  //Register number may exceed 32767, so int
    char name[20];
    short ce;//int takes 4 bytes, but ce score is a small number
    short pe;
    short te;
};                    //end of structure definition
int main()
{
    student s;        //structure variable
    int tot_score;
    cout<<"Enter register number: ";
    cin>>s.reg_no;
```

```
    fflush(stdin);    //To clear the keyboard buffer
    cout<<"Enter name: ";
    gets(s.name);
    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;
}
```

A sample output window of Program 1.1 is given below:

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
```

In Program 1.1, the structure is defined outside `main()` function. It may be defined inside `main()` also. The position of the definition determines the scope and life of the structure. Recollect the concept of local and global scope of variables and functions that we discussed in Chapter 10 of Class XI. If the definition is inside the `main()`, the structure can be used to declare variables within the `main()` function only. On the other hand, if the definition has a global scope, it allows declaration of structure variables in any function in the program.

Program 1.1 uses `fflush()` function before the `gets()` function. It is required in programs where an input of string facilitated by `gets()` function is followed by any other input. When we press <Enter> key as the delimiter for the former input, the `'\n'` character corresponding to the <Enter> key available in the keyboard buffer will be taken as the input for the string variable. This character will be considered as the delimiter for the string variable and the program control goes to the next statement in the program. In effect, we will not be able to input the actual string. So, we used `fflush()` function before inputting the name.

In Program 1.1, only one structure variable is used and hence the data of only one student can be referenced by the program at a time. If we have to deal with the details of a group of students, we will use an array of structures. So, let us write a program to illustrate the concept of array of structures. Program 1.2 accepts the details of a group of salesmen, each of which includes salesman code, name and amount of sales in 12 months. The program displays the entered details along with the average sales of all the salesmen. We can also see an array of floating point numbers as one of the elements of the structure.

Program 1.2: To find the average sales by salesmen

```
#include <iostream>
#include <cstdio>
#include <iomanip>    //To use setw() function
using namespace std;
struct sales_data
{
    int code;
    char name[15];
    float amt[12]; //To store the amount of sales in 12 months
    float avg;
};
int main()
{
    sales_data s[20]; //array of structure
    short n,i,j; //short is to minimise the amount of memory
    float sum;
    cout<<"Enter the number of salesmen: ";
    cin>>n;
    for(i=0; i<n; i++)
    {
        cout<<"Enter details of Salesman "<<i+1;
        cout<<"\nSalesman Code: ";
        cin>>s[i].code;
        fflush(stdin);
        cout<<"Name: ";
        gets(s[i].name);
        cout<<"Amount of sales in 12 months: ";
        for(sum=0,j=0; j<12; j++)
        {
            cin>>s[i].amt[j];
            sum=sum+s[i].amt[j];
        }
```

```
        s[i].avg=sum/12;
    }
    cout<<"\t\tDetails of Sales\n";
    cout<<"Code\t\tName\t\tAverage Sales\n";
    for(i=0;i<n;i++)
    {
        cout<<setw(4)<<s[i].code<<setw(15)<<s[i].name;
        for (j=0;j<12;j++)
            cout<<setw(4)<<s[i].amt[j];
        cout<<s[i].avg<<'\n';
    }
    return 0;
}
```

You may try out this program in the lab and see the output. In program 1.2, we used a floating point array as one of the elements of the structure. It uses array of structures for handling the details of different salesmen. Note that the variables `n`, `i` and `j` are declared using `short`. It allocates only 2 bytes for each of these variables. If `int` would have been used, 4 bytes would be used.

1.1.4 Nested structure

An element of a structure may itself be another structure. Such a structure is known as ***nested structure***. The concept of nesting enables the building of powerful data structures. If we want to include date of admission as an element in the structure `student`, any of the definitions given in Table 1.2 can cater to the need.

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;`
	`};`

Table 1.2: Two styles of nesting

Definition A of Table 1.2 contains the two structures defined separately. The second structure, `student`, contains structure variable `dt_adm` of `date` type as an element. Here we have to make sure that the inner structure is defined before making it nested. But in definition B we can see that structure `date` is defined inside the structure `student`. If this style is followed, the scope of `date` is only within `student` structure and hence a variable of type `date` cannot be declared outside `student`. Since the variable declaration of the inner structure is essential, its tag may be avoided in the definition. The following statements illustrate how a nested structure variable is initialised and the elements are accessed:

```
student s = {4325, "Vishal", {10, 11, 1997}, 575};
cout<<s.adm_no<<s.name;
cout<<s.dt_adm.day<<"/"<<s.dt_adm.month<<"/"<<s.dt_adm.year;
```

Let us do

Define a structure `employee` with the details employee code, name, date of joining, designation and basic pay.
Draw the layout of memory location allocated to a variable of `employee` type and find its size.

Note that the format for accessing the inner structure element is:

```
outer_structure_varaiable.inner_structure_variable.element
```

Array Vs Structure

We discussed arrays and structures as data types to refer to a collection of data under a common name. But they differ in some aspects. Table 1.3 shows a 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

Table 1.3: Comparison between arrays and structures

Know your progress

1. What is structure?
2. Structure combines different types of data under a single unit. State whether this is true or false.
3. Which of the following is true for accessing an element of a structure?
   a. `struct.element`
   b. `structure_tag.element`
   c. `structure_variable.element`
   d. `structure_tag.structure_variable`
4. What is nested structure? Write an example.
5. As subscript is for array, _____ is associated with structure.

1.2 Pointers

Suppose we have to prepare an assignment paper on 'Advances in Computing'. We may need suitable books for collecting the material. Obviously we may search for the books in the library. We may not be able to locate the book in the library. The librarian or our Computer Science teacher can help us to access the book. Let us think of the role of the librarian or the teacher. He/she is always a reference. He/she can provide us with the actual data (book) that is stored somewhere in the library. Figure 1.4 illustrates this example.

Fig. 1.4: Example for reference

Pointer is something like the librarian or teacher in the above example. It is a kind of reference. Consider the following C++ statement:

```
int num=25;
```

We know that it is a variable initialisation statement, in which `num` is a variable that is assigned with the value 25. Naturally, this statement causes memory allocation as shown in Figure 1.5.

In the figure, we can see that a variable has three attributes - its name, address and data type. Here, the name of the variable is `num` and the content 25 shows the data type. What about the address? Is it 1001, 1002, 1003 or 1004? It is 1001. Variable `num`

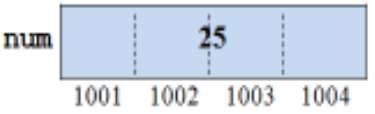

Fig.1.5: Memory allocation

being `int` type, 4 bytes (in GCC) are allocated. We know that each cell in RAM is of one byte size and each cell is identified by its unique address. But, when more than one cell constitute a single storage location (known as *memory word*), the address of the first cell will be the address of that storage location. That is how 1001 becomes the address of `num`. In class XI, we learnt that a variable is associated with two values: L-value and R-value, where ***L-value*** is the address of the variable and ***R-value*** is its content. Figure 1.5 shows that L-value of `num` is 1001 and R-value is 25.

Suppose we want to store the L-value (address) of a variable in another memory location. A variable is needed for this and it is known as pointer variable. Thus we can define ***pointer*** as a variable that can hold the address of a memory location. Pointer is primitive since it contains memory address which is atomic in nature. So we will say that pointer is a variable that points to a memory location (or data).

Harold Lawson (born 1937), a software engineer, computer architect and systems engineer is credited with the 1964 invention of the pointer. In 2000, Lawson was presented the Computer Pioneer Award by the IEEE for his invention.

As you know, computers use their memory for storing the instructions of a program, as well as the values of the variables that are associated with it. The memory is a sequential collection of 'storage cells' as shown in Figure 1.6. Each cell, commonly known as a byte, has a number called address associated with it. Typically, the addresses are numbered consecutively, starting from 0 (zero). The address of the last cell depends on the memory size. A computer memory having 64 K (64 x 1024 = 65536 Bytes) memory will have its last address as 65,535.

Fig.1.6: Memory organisation

Whenever we declare a variable in a program, a location is allocated somewhere in the memory to hold the R-value of the variable. Since every byte has a unique number as its address, this location will have its own address. Nowadays, the size of RAM is in terms of GBs and the address of memory location is expressed in hexadecimal number. It is because hexadecimal system can express larger values with lesser number of digits compared to decimal system.

1.2.1 Declaration of pointer variable

Pointer is a derived data type and hence a variable of pointer type is to be declared prior to its use in the program. The following syntax is used to declare pointer variable:

```
data_type * variable;
```

The `data_type` can be fundamental or user-defined and variable is an identifier. Note that an asterisk (*****) is used in between the data type and the variable. The following are examples of pointer declaration:

```
int *ptr1;
float *ptr2;
struct student *ptr3;
```

As usual memory will be allocated for these pointers. Do you think that the amount of memory for these variables is dependent on the data types used? We know that memory addresses are unsigned integer numbers. But it does not mean that pointers are always declared using `unsigned int`. Then, what is the criterion for determining the data type for a pointer? The data type of a pointer should be the same as that of the data pointed to by it. In the above examples, `ptr1` can contain the address of an integer location, `ptr2` can point to a location containing floating point number, and `ptr3` can hold the address of location whose R-value is `student` type data. So what will be the size of a pointer variable? The memory space for a pointer depends upon the addressing scheme of the computer. Usually, the size of a pointer in C++ is 2 to 4 bytes. As far as a programmer is concerned, there is no need to bother about the size of pointer while solving problems.

1.2.2 The operators & and *

Once a pointer is declared, memory address of a location of the same data type can be stored in it. When a variable is referenced in a C++ statement, actually its R-value is referred to. How can we retrieve its address (L-value)? C++ provides an operator named ***address of operator*** (**&**), to get the address of a variable. If `num` is an integer variable, its address can be stored in pointer `ptr1` by the following statement:

```
ptr1 = &num;
```

The statement, on execution, establishes a link between two memory locations as shown in Figure 1.7.

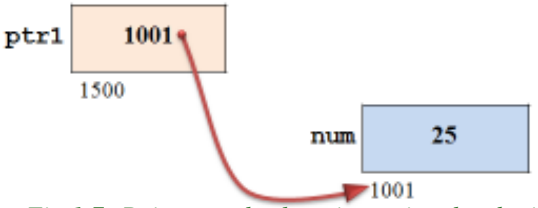

Fig.1.7: Pointer and a location pointed to by it

We have discussed that pointer is a kind of reference. Since a pointer references

a data stored somewhere in the memory, by dereferencing the pointer we get the data. C++ provides this dereferencing facility by an operator named ***indirection*** or ***dereference operator*** (*). The following statement retrieves the value pointed to by the pointer `ptr1` and displays on the screen.

```
cout << *ptr1;
```

It is clear that this statement is equivalent to the statement: `cout << num;`

Since the operator * retrieves the value at the location pointed to by the pointer, the * operator is also known as ***value at operator***.

Note that 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.

Considering the variables used in Figure 1.7, the following statements illustrate the operations performed by these operators:

```
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
```

The last statement is invalid. An error will be reported during compilation, because `num` is not a pointer and the content 25 is not a memory address. The indirection operator (*) should be used only with pointers.

1.3 Methods of memory allocation

We know that variable declaration statements initiate memory allocation. The required memory is allocated when the program is loaded in RAM. The execution of the program begins only after this memory allocation. The amount of memory allocated depends upon the number and data type of variables used in the program. This amount is static, i.e., it will not increase or decrease during the program run. The memory allocation that takes place before the execution of the program is known as ***static memory allocation***. It is due to the variable declaration statements in the program. There is another kind of memory allocation, called ***dynamic memory allocation***. In this case, memory is allocated during the execution of the program. It is facilitated by an operator, named **`new`**. As complementary to this operator, C++ provides another operator, named **`delete`** to de-allocate the memory.

1.3.1 Dynamic operators - new and delete

The operator `new` is a keyword in C++ and it triggers the allocation of memory during run-time (execution). It is a unary operator and the required operand is either a fundamental or user-defined data type. Dynamic memory allocation being an operation, the operator `new` and the operand data type constitute an expression. 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. The following syntax is used for dynamic memory allocation:

```
pointer_variable = new data_type;
```

Note that a pointer variable is used to hold the address returned by the `new` operator. So, it should be declared earlier with the same data type specified after `new` operator. The following are examples for dynamic memory allocation:

```
short * si_ptr;
float * fl_ptr;
struct complex * cx_ptr;
si_ptr = new short;
fl_ptr = new float;
cx_ptr = new complex;
```

The memory allocations are shown in Figure 1.8.

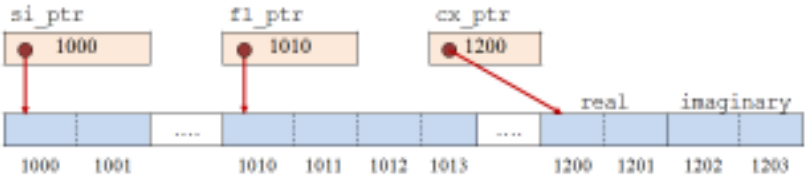

Fig. 1.8: Layout of dynamic memory allocation

Figure 1.8 shows that 2 bytes of location for `short` type data is allocated at the address 1000 and it is stored in `si_ptr`. Similarly 4 bytes from the address 1010 for `float` type data is allocated and this address is stored in `fl_ptr`. Earlier we discussed a structure named `complex` that consists of two `short` type elements. The pointer `cx_ptr` holds the address 1200 that is allocated for a `complex` type data of size 4 bytes (2 bytes each for `short real` and `short imaginary`). Note that, the dynamically allocated memory locations cannot be referred to by ordinary variables. Rather these are accessed using indirection (dereferencing) operator only as shown in the following examples:

```
*si_ptr = 247;
cin >> *fl_ptr;
```

We have a structure pointer `cx_ptr`, but the data pointed to by this pointer cannot be accessed in this format. We will discuss the accessing method later in this chapter.

As in the case of variable initialisation during static memory allocation, dynamically allocated memory locations can also be initialised using the following syntax:

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

The following examples show initialisation along with dynamic memory allocation:

```
si_ptr = new short(0);
fl_ptr = new float(3.14);
```

In the case of `cx_ptr`, this kind of initialisation is not possible.

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 purpose. In the case of static memory allocation, operating system itself allocates and releases memory depending on the scope and life of the variables. But in the case of dynamic memory allocation, the program should have an explicit statement to release (or free) the memory. For that, `delete` operator is used with the following syntax:

```
delete pointer_variable;
```

The following are valid examples:

```
delete si_ptr;
delete fl_ptr, cx_ptr;
```

1.3.2 Memory leak

If the memory allocated using `new` operator is not freed using `delete`, that memory is said to be an orphaned memory block - a block of memory that is left unused, but not released for further allocation. This memory block is allocated on each execution of the program and the size of the orphaned block is increased. Thus a part of the memory seems to disappear on every run of the program, and eventually the amount of memory consumed has an unfavorable effect. This situation is known as ***memory leak***.

The following are the reasons for memory leak:

- Forgetting to delete the memory that has been allocated dynamically (using `new`).
- 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 is to ensure that the memory allocated through `new` is properly de-allocated through `delete`. Memory leak takes place only in the case of dynamic memory allocation. But in case of static memory allocation, the Operating System takes the responsibility of allocation and deallocation without user's instruction. So there is no chance of memory leak in static memory allocation

Let us do

Now let us compare static memory allocation and dynamic memory allocation. Table 1.4 may be used for comparison. Some of the entries are left for you to complete using proper points.

Static memory allocation	Dynamic memory allocation
i. Takes place before the execution of the program.	
ii.	`new` operator is required
iii.	Pointer is essential
iv. Data is referenced using variables	
v. No statement is needed for de-allocation	

Table 1.4: Static Vs Dynamic memory allocation

Know your progress

1. What is pointer?
2. What is the criterion for determining the data type of a pointer?
3. If `mks` is an integer variable, write C++ statements to store its address in a pointer.
4. If `ptr` is an integer pointer, write C++ statement to allocate memory for an integer number and initialise it with 12.
5. Consider the statements: `int *p, a=5; p=&a; cout<<*p+a;` What is the output?

1.4 Operations on pointers

We have discussed that indirection (*) and address of (&) operators can be used with pointers. In Class XI, we used arithmetic, relational and logical operators. In this section, we will have a look at the operators that can be used with pointers and how these operations are performed.

1.4.1 Arithmetic operations on pointers

We have seen that memory address is numeric in nature. Hence some of the arithmetic operations can be performed on pointers. Let us consider the pointers `si_ptr` and `fl_ptr` declared in section 1.3.1 (Refer Figure 1.8). Now, observe the following statements:

```
cout << si_ptr + 1;
cout << fl_ptr + 1;
```

What will be the output? Do you think that it will be 1001 and 1011?

Adding 1 to a pointer is not the same as adding 1 to an `int` or `float` type data. When we add 1 to a `short int` pointer, the expression returns the address of the next location of `short int` type. The cells with addresses 1000 and 1001 constitute a single storage location for an integer data of `short` type. Hence the address of the next addressable short integer location is 1002. 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. So the expression `fl_ptr+1` returns 1014. So, it is clear that the expression `si_ptr+5` returns 1010 (1000+5×2) and `fl_ptr+3` returns 1022 (1010+3×4). Similarly, subtraction operation can also be performed on pointers.

Let us do

Find the values returned by the following arithmetic expressions:

```
si_ptr + 10        fl_ptr + 7
si_ptr - 5         fl_ptr - 10
```

Note that this kind of operation is practically wrong. Because we are trying to access locations that are not allocated for authorised use. These locations might have been used by some other variables. Sometimes these locations might not have been accessible due to the violation of access rights.

No other arithmetic operations are performed on pointers. So we can conclude that pointers are only incremented or decremented. The following statements illustrate various operations on pointers:

```
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 1004
    cin>> *ptr2;        /*   Reads an integer (say 12) and
                        stores it in location 1004 */
    cout<< *ptr1 + 1; //   Displays 6 (5 + 1)
    cout<< *(ptr1 + 2);//  Displays 12, the value at 1004
    ptr1--;             //   Same as ptr1 = ptr1 - 1
```

Let us write a program to demonstrate the operations on pointers. Program 1.3 gives the average height of a group of students.

Program 1.3: To find the average height of students

```
#include <iostream>
using namespace std;
int main()
{
    int *ht_ptr, n, s=0;
    float avg_ht;
    ht_ptr = new int;       //dynamic memory allocation
    cout<<"Enter the number of students: ";
    cin>>n;
    for (int i=0; i<n; i++)
    {
        cout<<"Enter the height of student "<<i+1<<" - ";
        cin>>*(ht_ptr+i);//to get the address of the next location
        s = s + *(ht_ptr+i);
    }
    avg_ht = (float)s/n;
    cout<<"Average height of students in the class = "<<avg_ht;
    return 0;
}
```

In program 1.3, an integer location is dynamically allocated and the address is stored in the pointer `ht_ptr`. When the body of the loop is executed for the first time, 0 is added to this address and it does not make any change. The input data is stored in this location. During the second execution of the loop-body, 1 is added to this address and the next integer location is referenced for the input. This process is continued for entering the heights of n students. Sum of these heights is calculated along with the input and after the completion of the loop, average is calculated.

Here explicit type conversion is used to get the accurate result. A sample output is shown below:

```
Enter the number of students: 5
Enter the height of student 1 - 170
Enter the height of student 2 - 169
Enter the height of student 3 - 175
Enter the height of student 4 - 165
Enter the height of student 5 - 177
Average height of students in the class = 171.199997
```

Program 1.3 also shows that a collection of the same type of data can be handled by utilising pointer arithmetic. Last year, we used arrays in such a situation. But the size should be specified during the array declaration. This may cause wastage or insufficiency of memory space. Pointer and its arithmetic overcome this drawback.

But there is a problem in this kind of memory usage. It is not sure that Program 1.3 will always run with any value of `n`. GCC may not give any output for the `avg_ht`. Though there is no problem theoretically, unexpected results may occur during execution. As mentioned earlier, pointer `ht_ptr` is initialised with the address of only one location. The memory locations accessed using pointer arithmetic on `ht_ptr` are unauthorised, since these locations are not allocated by the OS. This may lead to unexpected termination of the program or loss of some data that already reside in those locations. We can overcome these issues by the facility of dynamic arrays, which we discuss in Section 1.5 of this chapter.

1.4.2 Relational operations on pointers

Among the six relational operators, only == (equality) and != (non-equality) operators are used with pointers. Memory address is simply a unique number to identify each memory location. If `p` and `q` are two pointers, they may contain the address of the same integer location or different memory locations. This can be verified with the expressions `p==q` or `p!=q`.

Know your progress

1. Dynamic memory allocation operator in C++ is _____.
2. What happens when the following statement is executed?
   ```
   int *p = new int(5);
   ```
3. What is orphaned memory block?
4. If p is an integer pointer, which of the following are invalid?
   a. `cout<<&p;` b. `p=p*5;` c. `p>0`
   d. `p++;` e. `p=1500;` f. `cout<<*p * 2;`

1.5 Pointer and array

We learnt that an array can contain a collection of homogeneous type of data under a common name. This data is stored in contiguous memory locations. Figure 1.9 shows the memory allocation of an array `ar[10]` of `int` type with 10 numbers.

	ar
1000	34
1004	12
1008	8
1012	18
1016	24
1020	38
1024	43
1028	14
1032	7
1036	19

Fig. 1.9: Memory allocation for array `ar`

It is assumed that the array begins at location 1000 and each location consists of 4 bytes (as per GCC). We know that any element of this array can be referenced by specifying the subscript along with the array name. For example, `ar[0]` returns 34, `ar[1]` returns 12, and at last `ar[9]` returns 19.

Let us do

Write C++ statement to display all the 10 elements of this array.

How can we store the address of the first location of this array into a pointer?

If `ptr` is an integer pointer, the address of the first location of array `ar[10]` can be stored in it with the following statement:

```
ptr = &ar[0];
```

Now let us see the output of the expressions used in the following statements:

```
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]
```

Can you predict the output of the statement: `cout<<ar;`?

The output will be 1000, which is the address of the first location of the array. This address is known as base address of the array. We have seen that a variable that contains the address of a memory location is called pointer. In that sense, 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];
```

The following C++ statement displays all the elements of this array:

```
for (int i=0; i<10; i++)
    cout<<*(ar+i)<<'\t';
```

There is a difference between an ordinary pointer and an array-name. The statement `ptr++;` is valid and is equivalent to `ptr=ptr+1;`. After the execution of this statement `ptr` will point to the location of `ar[1]`. That is, `ptr` will contain the address of `ar[1]`. But the statement `ar++;` is invalid, because array-name always contains the base address of the array, and it cannot be changed.

Dynamic array

In C++, array helps to handle a collection of same type of data. But, if the number of data items is not known in advance, there is a problem in declaring the array. As we know, size of array is to be specified in the declaration statement, and it should be an integer constant. How can we declare an array to store the percent of pass obtained by the schools in any district in the Higher Secondary examination? Neither `float pass[n];` nor `float pass[];` is valid. We have to mention an integer constant as size of the array and it may cause insufficiency or wastage of memory space. The district, and hence the number of schools are unknown while writing the program. So, the program should provide the facility to allocate the required locations as per the user's input. The solution in such a situation is dynamic array.

Dynamic array is created during run time using the dynamic memory allocation operator `new`. The syntax is:

```
pointer = new data_type[size];
```

Here, the `size` can be a constant, a variable or an integer expression. Program 1.4 illustrates the concept of dynamic array. It can store the percent of pass secured by the schools. The number of schools will be decided by the user only at the time of execution of the program.

Program 1.4: 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
```

Program 1.4 uses dynamic array to store the data. Memory is allocated only during execution and five locations, each with 4 bytes, are reserved for the array `pass`. Elements of this array are accessed using subscript as well as pointer arithmetic operation.

Let us do

Read the following statements and write the difference between them:

```
int *ptr = new int(10);
int *ptr = new int[10];
```

1.6 Pointer and string

In Class XI, we learnt that string data can be referenced by character array and the array-name can be considered as string variable. In the previous section, we saw that array-name contains the base address of the array, and hence it can be considered as a pointer. Let us discuss how these two aspects are combined to refer to strings using pointer. The following statements illustrate how character pointer differs from the other pointers:

```
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
```

A string contained in an array cannot be copied into another character array using assignment operator (=) (*we used* `strcpy()` *function last year*). But, the assignment is possible with character pointers. The statements `sp=str;` and `cout<<sp;` show this fact. It proves that a character pointer can be used to store a string and this pointer can be considered as a string variable. That is, as we use character array name to refer to string data, character pointer can also serve the same.

Another interesting aspect is that, the statement `cout<<&str[0];` also displays the entire string, instead of the address of the first location (base address). That means, if we access the address of a string data, we get the string itself. But `str[0]` and `*sp` gives the first character of the string.

Advantages of character pointer

The use of character pointer for storing string offers the following advantages over character array:

- Since there is no size specification, a string of any number of characters can be stored. There is no wastage or insufficiency of memory space. But it should be done with initialization. (e.g., `char *str = "Program";`)
- 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

Suppose, we want to store the names of days in a week. A character array or character pointer can be used to store only one name at a time. Here we need to refer to a collection of strings ("Sunday", "Monday", ..., "Saturday"). Obviously we should use an array of character arrays (2D array of `char` type) or an array of character pointers. The following statement declares an array of character pointers to handle this case:

```
char  *name[7];
```

This array can contain a maximum of 7 strings, where each string can contain any number of characters. But we should make sure that the pointer array is initilaised. It may be as follows:

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

Figure 1.10 shows the optimal use of memory locations. Only the shaded portion will be allocated.

Let us do

Write C++ statements to sort these names using any of the sorting techniques we discussed in Class XI. Since we use character pointer, strings can be copied using assignment operator. Check the correctness of your code during your lab work.

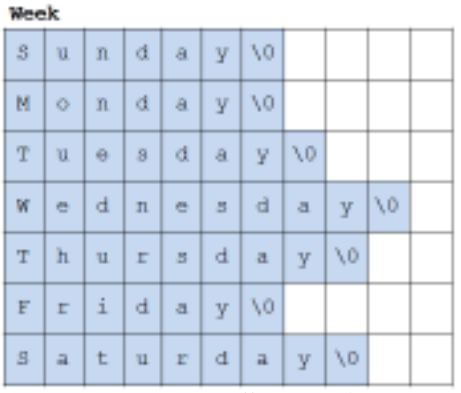

Fig. 1.10: Memory allocation for strings

The following statement illustrates the accessing of these strings:

```
for (i=0; i<7; i++)
    cout<<name[i];
```

An array of strings can be handled using a 2D character array as given below:

```
char name[10][20];
```

This array can contain 10 names, each of which can have a maximum of 19 characters. One byte is reserved for null character (`'\0'`). Each string is referred to by the expression `name[i]`, where the subscript `i` can take values from 0 to 9. In this case `strcpy()` function should be used for copying the strings into variables.

Know your progress

1. What is dynamic array?
2. Address of the first location of an array is known as _____.
3. If `arr` is an integer array, which of the following are invalid?
   a. `cout<<arr;` b. `arr++;` c. `cout<<*(arr+1);`
   d. `cin>>arr;` e. `arr=1500;` f. `cout<<*arr * 2;`
4. Write a declaration statement in C++ to refer to the names of 10 books using pointers.
5. Write a statement to declare a pointer and initialise it with your name.

1.7 Pointer and structure

Earlier in this chapter, we discussed structure data type and its applications. This section discusses how structures are accessed by pointers. A structure is defined to represent the details of employees as follows:

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

Now, observe the following declaration statement:

```
employee *eptr;
```

It is clear that `eptr` is a pointer that can hold the address of `employee` type data. The statement:

```
eptr = new employee;
```

allocates 23 bytes of memory and its address is stored in the pointer `eptr`. Figure 1.11 illustrates the effect of this statement.

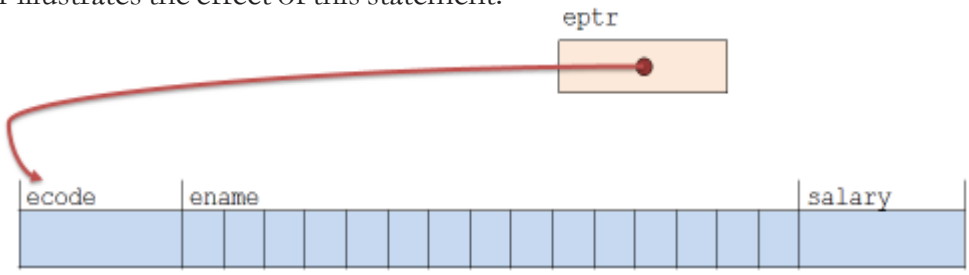

Fig. 1.11: Dynamic memory allocation for `employee` *type data*

We learnt that a structure is accessed in terms of its elements with the following format:

```
structure_variable.element_name
```

Here, we do not have a structure variable to access the elements `ecode`, `ename` and `salary`. So we have to use the pointer `eptr`. The syntax for accessing the elements of a structure is as follows:

```
structure_pointer->element_name
```

Note that structure pointer and an element is connected using arrow operator (**->**). It is constituted by a hyphen (**-**) followed by greater than symbol (**>**). The following statements are examples for accessing the elements of the structure shown in Figure1.11.

```
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 do

Earlier in this chapter, in Section 1.3.1, we mentioned a pointer `cx_ptr` of `complex` type structure. Write C++ statements to input a complex number and display in its actual format.

Let us modify the structure `employee` by adding an element as follows:

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

Obviously, the element `ip` can contain the address of an integer location. The following statements illustrate the use of pointer `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
```

Observe that 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 its element. Even it may be of the same structure data type as follows:

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

The element ep is a pointer of employee data type

Now, the structure employee is known as self referential structure. Let us discuss more on this type of structures and their applications.

Self referential structure

Self referential structure is a structure in which one of the elements is a pointer to the same structure. A location of this type contains data and the address of another location of the same type. This location can again contain data and address of yet another location of the same structure type. It can be extended as per the requirement. Figure 1.12 shows this concept.

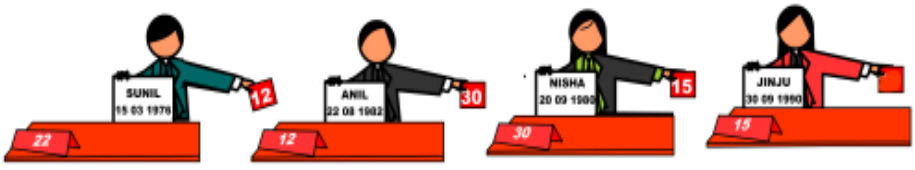

Fig. 1.12: An employee of structure type points to another employee

An employee named "Sunil" points to the next employee whose table number is 12. The employee at table number 12 is "Anil" and he points to the next employee "Nisha" and so on.

Self referential structure is a powerful tool of C and C++ languages that helps to develop dynamic data structures like linked list, tree, etc. Dynamic data structure means a collection of data for which memory will be allocated during run-time. The memory locations are scattered, but there will be a link from one location to another. More about the data structure linked list will be discussed in Chapter 3.

Let us conclude

We have discussed more advanced data types in C++ in this chapter. Structure data type is introduced to represent grouped or aggregate data under single name. Accessing of elements with dot operator (.) is discussed. Pointer is presented as a special type of data. Operations associated with pointers are illustrated with the help of expressions. The concept of dynamic memory allocation and the required

operators, and its advantages are discussed. The relationship between array and pointer is illustrated, and string data are handled using pointer. A good understanding of the concepts dealt with in this chapter will help you to attain the learning outcomes specified in Chapter 3 and equip you for higher studies.

Let us practice

1. Define a structure to represent the details of telephone subscribers which include name of the subscriber and telephone number. Write a menu driven program to store the details of some subscribers with options for searching the name for a given number, and the number for a given name.
2. Define a structure to represent the details of customers in a bank. The details include account number, name, date of opening the account and balance amount. Write a menu driven program to input the details of a customer and provide options to deposit, withdraw and view the details. During deposit and withdrawal, proper update is to be made in the balance amount. A minimum balance of Rs. 1000/- is a must in the account.
3. Write a program to input the TE scores obtained by a group of students in Computer Science and display them in the descending order using pointers.
4. Write a program to input a string and check whether it is palindrome or not using character pointer.
5. Write a program to input the names of students in a class using pointers and create a roll list in which the names are listed in alphabetical order with roll number starting from 1.
6. Define a structure student with the details register number, name and CE marks of six subjects. Using a structure pointer, input the details of a student and display register number, name and total CE score.

Let us assess

1. Compare array and structure in C++.
2. Identify the errors in the following structure definition and write the reason for each:

```
struct
{
    int roll, age;
    float fee=1000;
};
```

3. Read the following structure definition and answer the following questions:

```
struct Book
{
    int book_no;
    char bk_name[20];
    struct
    {
        short dd;
        short mm;
        short yy;
    }dt_of_purchase;
    float price;
};
```

   a. Write a C++ statement to declare a variable to refer to the details of a book. What is the memory requirement of this variable? Justify your answer.

   b. Write a C++ statement to initialise this variable with the details of your Computer Science text book.

   c. Write C++ statement(s) to display the details of the book.

   d. The missing of structure tag in the inner structure does not cause any error. State whether this is true or false. Give reason.

4. "Structure is a user-defined data type". Justify this statement with the help of an example.

5. Read the following statements:

   i. While defining a structure in C++, tag may be omitted.

   ii. The data contained in a structure variable can be copied into another variable only if both of them are declared using the same structure tag.

   iii. Elements of a structure is referenced by structure_name.element

   iv. A structure can contain another structure.

   Now, choose the correct option from the following:

   a. Statements (i) and (ii) are true
   b. Statements (ii) and (iv) are true
   c. Statements (i), (ii) and (iv) are true
   d. Statements (i) and (iii) are true

6. Read the following C++ statements:

```
int * p, a=5;
p=&a;
```

   a. What is the speciality of the variable p?

   b. What will be the content of p after the execution of the second statement?

   c. How do the expressions `*p+1` and `*(p+1)` differ?

7. Identify the errors in the following C++ code segment and give the reason for each.

```
int *p,*q, a=5;
float b=2;
p=&a;
q=&b;
cout<<p<<*p<<*a;
if (p<q)cout<<p;
    cout<<*p * a;
```

8. While writing a program, the concept of dynamic memory allocation is applied. But the program does not contain a statement with `delete` operator and it creates a problem. Explain the problem.

9. Read the C++ statements given below and answer the following questions:

```
int ar[] = {34, 12, 25, 56, 38};
int *p = ar;
```

   a. What will be the content of p?

   b. What is the output of the expression: `*p + *(ar+2)`?

   c. The statement `ar++;` is invalid. Why? How does it differ from `p++;`?

10. Explain the working of the following code segment and predict the output:

```
char *str = "Tobacco Kills";
for (int i=0; str[i]!='\0'; i++)
    if (i>8)
        *(str+i) = toupper(*(str+i));
cout<<str;
```

11. Observe the following C++ statements:

```
int ar[] = {14, 29, 32, 63, 30};
```

One of following expressions cannot be used to access the element 32. Which is that?

a. `ar[2]` b. `ar[*ar%3]` c. `*ar+2` d. `*(ar+2)`

12. Explain the operations performed by the operaors `new` and `delete` with the help of examples.

13. What is meant by memory leak? What are the reasons for it? How can we avoid such a situation?

14. Compare the following two statements.

```
int a=5;
int *a=new int(5);
```

15. Read the structure definition given below and answer the following questions:

```
struct sample
{
    int num;
    char *str;
}   *sptr;
```

a. Write C++ statements to dynamically allocate a location for `sample` type data and store its address in sptr.

b. Write C++ statements to input data into the location pointed to by `sptr`.

c. Modify this structure into a self referential structure.