SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

PROGRAMMING FOR PROBLEM SOLVING

I YEAR - II SEM

UNIT 5

TOPIC 1 – Structure Introduction

INTRODUCTION

- We have seen that arrays can be used to represent a group of data items that belong to the same type, such as **int or float**.
- However, we cannot use an array if we want to represent a collection of data items of **different types** using a single name.
- Fortunately, C supports a constructed data type known as **structures**, a mechanism for packing data of different types.
- A **structure** is a convenient tool for handling a group of logically related data items.
- For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks.
- The concept of a structure is analogous to that of a ‘record’ in many other languages.

INTRODUCTION

- Examples of such structures are:
- time : seconds, minutes, hours
- date : day, month, year
- book : author, title, price, year
- city : name, country, population
- address : name, door-number, street, city
- inventory : item, stock, value
- customer : name, telephone, city, category

- Structures help to organize complex data in a more meaningful way.
- It is a powerful concept that we may often need to use in our program design.

DEFINING A STRUCTURE

- Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables.
- Let us use an example to illustrate the process of structure definition and the creation of structure variables.
- Consider a book database consisting of book name, author, number of pages, and price.
- We can define a structure to hold this information as follows:

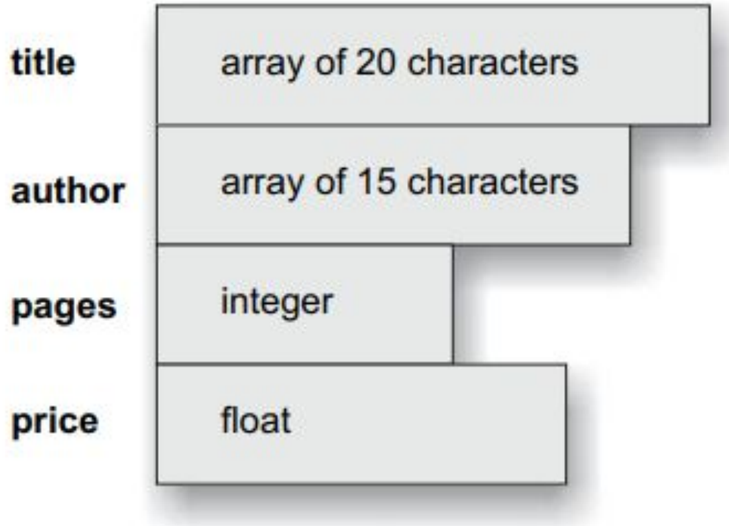
```
struct book_bank
{
 char title[20];
 char author[15];
 int pages;
 float price;
};
```

DEFINING A STRUCTURE

- The keyword **struct** declares a structure to hold the details of four data fields, namely title, author, pages, and price.
- These fields are called structure **elements or members**.
- Each member may belong to a different type of data.
- **book_bank** is the name of the structure and is called the structure **tag**.
- The **tag name** may be used subsequently to declare variables that have the tag's structure.
- Note that the above definition has not declared any variables.
- It simply describes a format called template to represent information as shown in below Fig.

DEFINING A STRUCTURE

```
struct book_bank
 {
   char title[20];
   char author[15];
   int pages;
   float price;
 };
```



The general format of a structure definition is as follows:

```
struct          tag_name
{
     data_type      member1;
     data_type      member2;
        ----        ----
        ----        ----
};
```

DEFINING A STRUCTURE

- In defining a structure you may note the following syntax:
- 1. The template is terminated with a semicolon.
- 2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
- 3. The tag name such as book_bank can be used to declare structure variables of its type, later in the program.

Arrays vs Structures

- Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner.
- But they differ in a number of ways which are as follows:
- 1. An array is a collection of related data elements of same type.
  - Structure can have elements of different types.
- 2. An array is derived data type whereas a structure is a programmer-defined one.
- 3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it.

- But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

DECLARING STRUCTURE VARIABLES

- After defining a structure format we can declare variables of that type.
- A structure variable declaration is similar to the declaration of variables of any other data types.
- It includes the following elements:
  - 1. The keyword struct.
  - 2. The structure tag name.
  - 3. List of variable names separated by commas.
  - 4. A terminating semicolon.
- For example, the statement
- struct book_bank, book1, book2, book3;
- declares book1, book2, and book3 as variables of type struct book_bank.

DECLARING STRUCTURE VARIABLES

- Each one of these variables has four members as specified by the template.
- The complete declaration might look like this:

```
struct book_bank
{
 char title[20];
 char author[15];
 int pages;
 float price;
};
struct book_bank book1, book2, book3;
```

DECLARING STRUCTURE VARIABLES

- The declaration is valid.

```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
} book1, book2, book3;
```

- The use of tag name is optional here. For example:

```
struct
{ ........
........
........
} book1, book2, book3;
```

- declares book1, book2, and book3 as structure variables representing three books, but does not include a tag name

STRUCTURE INITIALIZATION

- Like any other data type, a structure variable can be initialized at compile time.

```
main()
{
 struct
 {
    int weight;
    fl oat height;
 }
 student = {60, 180.75};
 .....
 .....
 }
```

- This assigns the value 60 to **student. weight** and 180.75 to **student. height**.
- There is a one-to-one correspondence between the members and their initializing values.

STRUCTURE INITIALIZATION

- A lot of variation is possible in initializing a structure.
- The following statements initialize two structure variables.
- Here, it is essential to use a tag name.

```
main()
{
 struct st_record
  {
     int weight;
     fl oat height;
  };
 struct st_record student1 = { 60, 180.75 };
 struct st_record student2 = { 53, 170.60 };
  .....
  .....
}
```

STRUCTURE INITIALIZATION

- Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record
{
 int weight;
 fl oat height;
} student1 = {60, 180.75};
main()
{
 struct st_record student2 = {53, 170.60};
 .....
  .....
}
```

RULES FOR INITIALIZING STRUCTURES

- There are a few rules to keep in mind while initializing structure variables at compile-time which are as follows:
- 1. We cannot initialize individual members inside the structure template.
- 2. The order of values enclosed in braces must match the order of members in the structure definition.
- 3. It is permitted to have a partial initialization.
  - We can initialize only the first few members and leave the remaining blank.
  - The uninitialized members should be only at the end of the list.
- 4. The uninitialized members will be assigned default values as follows:
  - Zero for integer and fl oating point numbers.
  - ‘\0’ for characters and strings.

COPYING AND COMPARING STRUCTURE VARIABLES

- Two variables of the same structure type can be copied the same way as ordinary variables.
- If person1 and person2 belong to the same structure, then the following statements are valid:
  - person1 = person2;
  - person2 = person1;
- However, the statements such as
  - person1 == person2
  - person1 != person2
- are not permitted.
- C does not permit any logical operations on structure variables.
- In case, we need to compare them, we may do so by comparing members individually.

COPYING A E VARIABLES

Program

```
struct class
{
     int  number;
     char name[20];
     float marks;
};

main()
{
     int  x;
     struct class student1 = {111,"Rao",72.50};
     struct class student2 = {222,"Reddy", 67.00};
     struct class student3;

     student3 = student2;

     x = ((student3.number ==  student2.number) &&
         (student3.marks  ==  student2.marks)) ? 1 : 0;

if(x == 1)
{
   printf("\nstudent2 and student3 are same\n\n");
   printf("%d %s %f\n", student3.number,
                        student3.name,
                        student3.marks);
   }
   else
        printf("\nstudent2 and student3 are different\n\n");

}
```

Output

```
student2 and student3 are same

222 Reddy 67.000000
```

Ways to Access Members

- We have used the **dot operator** to access the members of structure variables.
- In fact, there are two other ways.
- Consider the following structure:

```
typedef struct
{
 int x;
  int y;
} VECTOR;
VECTOR v, *ptr;
ptr = & v;
```

- The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable n.
- Now, the members can be accessed in the following three ways:
  - 1. using dot notation : v.x
  - 2. using indirection notation : (*ptr).x
  - 3. using selection notation : ptr –> x

ARRAYS OF STRUCTURES

- We use structures to describe the **format of a number** of related variables.
- For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables.
- In such cases, we may declare an array of structures, each element of the array representing a structure variable.
- For example:
  - struct class student[100];
- defines an **array** called student, that consists of 100 elements.
- Each element is defined to be of the type struct class.

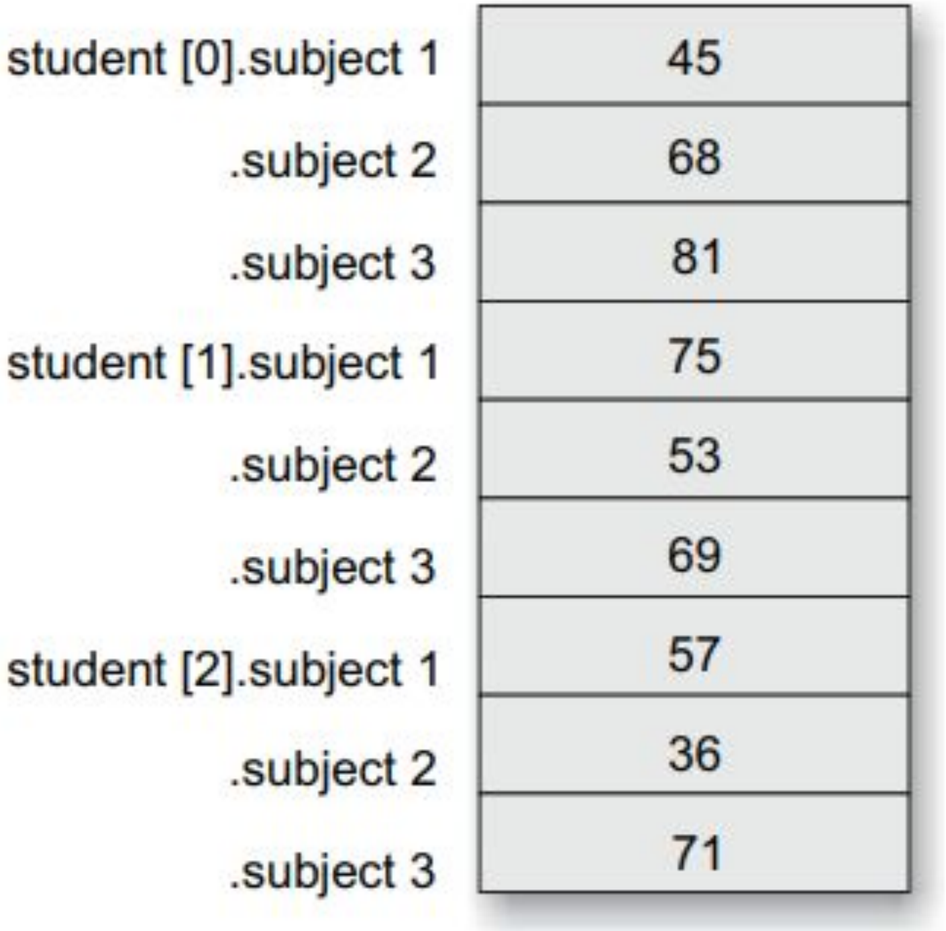
ARRAYS OF STRUCTURES

```
struct marks
 {
      int subject1;
      int subject2;
      int subject3;
 };
 main()
 {
 struct marks student[3] =
 {{45,68,81}, {75,53,69}, {57,36,71}};
```

- This declares the student as an array of three elements student[0], student[1], and student[2] and initializes their members as follows:

```
 student[0].subject1 = 45;
 student[0].subject2 = 65;
 ....
 ....
 student[2].subject3 = 71;
```

ARRAYS OF STRUCTURES



The array student inside memory

ARRAYS WITHIN STRUCTURES

- C permits the use of arrays as structure members.
- We have already used arrays of characters inside a structure.
- Similarly, we can use single-dimensional or multi-dimensional arrays of type int or float.
- For example, the following structure declaration is valid:

```
struct marks
{
int number;
float subject[3];
} student[2];
```

- Here, the member subject contains three elements, **subject[0], subject[1], and subject[2]**.
- These elements can be accessed using appropriate subscripts.
- For example, the name **student[1].subject[2];**
- would refer to the marks obtained in the third subject by the second student

STRUCTURES WITHIN STRUCTURES

- Structures within a structure means nesting of structures.
- Nesting of structures is permitted in C.
- Let us consider the following structure defined to store information about the salary of employees:

```
struct salary
{
    char name;
    char department;
    int basic_pay;
    int dearness_allowance;
    int house_rent_allowance;
    int city_allowance;
}
employee;
```

STRUCTURES WITHIN STRUCTURES

- This structure defi nes name, department, basic pay and three kinds of allowances. We can group all the
- items related to allowance together and declare them under a substructure as shown below:

```
struct salary
{
   char name;
    char department;
struct
{
   int dearness;
    int house_rent;
    int city;
  }
  allowance;
}
employee;
```

STRUCTURES AND FUNCTIONS

- We know that the main philosophy of C language is the use of functions.
- And therefore, it is natural that C supports the passing of structure **values as arguments to functions**.
- There are **three methods** by which the values of a structure can be transferred from one function to another.
- 1. The first method is to pass each member of the structure as an **actual argument** of the function call.
- The actual arguments are then treated independently like ordinary variables.
- This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.

STRUCTURES AND FUNCTIONS

- 2. **The second method** involves passing of a copy of the entire structure to the called function.
- Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function).
- It is, therefore, necessary for the function to return the entire structure back to the calling function.
- **All compilers may not support this method** of passing the entire structure as a parameter.

- **3. The third approach** employs a concept called **pointers** to pass the structure as an argument.
- In this case, **the address location** of the structure is passed to the called function.
- The function can access indirectly the entire structure and work on it.
- This is similar to the way arrays are passed to function.
- This method is more efficient as compared to the second one

STRUCTURES AND FUNCTIONS

- The general format of sending a copy of a structure to the called function is:

  function_name (structure_variable_name);

- The called function takes the following form:

```
data_type function_name(struct_type st_name)
{
.....
.....
return(expression);
}
```

UNIONS AND STRUCTURES

- Unions are a concept borrowed from structures and therefore follow the same syntax as structures.
- However, there is major distinction between them in terms of storage.
- In structures, each member has its own storage location, whereas all the members of a union use the same location.
- This implies that, although a union may contain many members of different types, it can handle only one member at a time.
- Like structures, a union can be declared using the keyword union as follows:

```
union item
{
    int m;
    float x;
   char c;
} code;
```

- This declares a variable code of type union item.

UNIONS AND STRUCTURES

- The union contains **three** members, each with a different data type.
- However, we can use only one of them at a time.
- This is due to the fact that only one location is allocated for a union variable, irrespective of its size.
- The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.
- In the declaration above, the member x requires 4 bytes which is the largest among the members.
- To access a union member, we can use the same syntax that we use for structure members.
- That is,

```
code.m
code.x
code.c
```

- are all valid member variables.
- For example, the statements such as

```
code.m = 379;
code.x = 7859.36;
printf("%d", code.m);
```

SIZE OF STRUCTURES

- We normally use structures, unions, and arrays to create variables of large sizes.
- The actual size of these variables in terms of bytes may change from machine to machine.
- We may use the unary operator **sizeof** to tell us the size of a structure (or any variable).
- The expression sizeof(struct x) will evaluate the number of bytes required to hold all the members of the **structure x**.
- If y is a simple structure variable of type struct x, then the expression sizeof(y) would also give the same answer.
- However, if y is an array variable of type struct x, then sizeof(y) would give the total number of bytes the array y requires.