INTRODUCTION TO DATA STRUCTURES

Chapter Objectives

At the completion of this chapter, you would have learnt:

? ? to understand the concept of Data Structures

? ? to declare and use an array in C;

? ? to declare and use a structure in C

? ? to declare and use an Union in C

1.1 Introduction

Data structure is a collection of data objects and a set of legal operations to be performed on them. E.g. array, structure and etc.

This chapter is used to teach understand the concept of data structures and also the use of an array, structure and union.

1.2 One-Dimensional Arrays

The array is a data structure in which we store a collection of data item of the SAME data type (e.g. all the exam scores for a class). By using an array, we can associate a single variable name (e.g. Scores) with the entire collection of data.

Example

88	95	33	55	44

We can also access individual items in the array by using the index or subscript number. Each of the item in the array is called a cell.

1.2.1 Definition and Examples

There are 2 ways to declare a structure :-

```
Declaration 1
int Number[10];

Declaration 2
typedef int NumberArray[10];

NumberArray Number;
```

***Typedef* is used to create a new data type (beside the original – int, float and etc) to be used in the program**

Both of the declarations above are declaring an array of integers of 10 cells. The first cell starts with the subscript 0. The advantage of using array compared to single item is that all the items in the array accessed using the same name.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
20	50	30	40	89	100	25	21	85	96

1.2.2 Accessing Array

Each item in the array is accessed by giving the array name first and then followed by the subscript number for each cell.

E.g. Let's say we need to assign 100 to cell number 3

```
Number[3] = 100;
```

You may also ask user to enter data and store data into the array by the following:

```
void main()
{
      int Number[10];
      int a;

      for (a=0;a<10;a++)
      {
            printf("Enter a number:");
            scanf("%d",Number[a]);
      }
}
```

1.2.3 Implementing Arrays

For the declaration

int num[100];

assuming that integer requires 2 bytes for storage, the array num will then require 200 bytes of storage location.

In fact, in the C language an array variable is implemented as a pointer variable. The type of the variable *b* in the above declaration is "pointer to an integer" or *int **. An * does not appear in the declaration because the brackets automatically imply that the variable is a pointer. The difference between declarations *int *b*; and *int b[100]*; is that the latter also reserves 100 integer locations starting at location *b*.

In C, all elements of an array have the same fixed, predetermined size.

1.2.4 Array as Parameters

Since an array variable in C is a pointer, array parameters are automatically passed ***by reference*** rather than pass by value. E.g

```
void input(int Numbers[]);

void main()
{
      int Numbers[10];
      input(Numbers);
}

void input(int Numbers[])
{
      ------
}
```

1.3 Structures

A structure is a group of elements which are of different data type. Each of the element is identified by its own identifier and they are called member. In some language, structures are known as records and the members are known as fields.

1.3.1 Definition and Examples

There are 3 ways to declare a structure :-

1st Way

```
struct      {
      char studentname[30];
      int age;
      float mark;
}sturec1, sturec2;
```

This declaration creates two structure variables, *sturec1* and *sturec2*, each of which contains three members; *studentname, age, mark.*

2nd Way

```
struct  studentrecord{
      char studentname[30];
      int age;
      float mark;
}

struct studentrecord sturec1, sturec2;
```

This declaration creates a structure tag *studentrecord* containing three members. Once a structure tag has been defined, variables *sturec1* and *sturec2* may be declared.

3rd Way

```
typedef  struct {
      char studentname[30];
      int age;
      float mark;
}STUDENTRECORD

STUDENTRECORD sturec1,sturec2;
```

This declaration uses *typedef* definition. Conventionally, *typedef* specifiers are written in uppercase

1.3.2 Accessing Structure

Each members of a C structure variable are referenced by giving first the name of the structure variable, then a period (.), then the name of the member as declared.

Example: To access the name of structure *sturec1*, we use the following statement

printf("%s", sturec.studentname);

You may also ask user to enter data and store data into the structure by the following:

```
void main()
{
      printf("Enter your name:");
      scanf(" %s",&sturec1.studentname);
      printf("Enter your age:");
      scanf(" %d",&sturec1.age);
      printf("Enter your mark:");
      scanf(" %f",&sturec1.mark);
}
```

1.3.3 Implementing Structures

Any type in C may be thought of as a pattern or a template. By this we mean that a type is a method for interpreting a portion of memory. When a variable is declared as being of a certain type, we are saying that the identifier refers to a certain portion of memory and that the contents of that memory are to be interpreted according to the pattern defined by the type. The type specifies both the amount of memory set aside for the variable and the method by which memory is interpreted.

For example, suppose that under a certain C implementation an integer is represented by 4 bytes, a float number by 8, and an array of 10 characters by 10 bytes. Then

`int x;`	means 4 bytes of memory are reserved for x
`float y;`	means 8 bytes are reserved for y
`char z[10];`	means 10 bytes are reserved for z

Lets look at the memory location for a structure shown below:-

```
struct structtype{
      int field1;
      float field2;
      char field3[10];
}

struct structtype r;
```

The amount of memory specified by the structure is the sum of the storage specified by each of its member types. Thus,

Total memory for r = (int + float + char) memory locations
= 4 + 8 + 10
= 22

The first 4 bytes are interpreted as an integer, the next 8 bytes as a float number, and the last 10 as an array of characters.

1.3.4 Structures Declared in a structure

You can also declare structures in a structure.

```
Declaration 1

struct Book_Details
{
      char title[30];
      float price;
};

struct Student_Details
{
      char studentname[20];
      int age;
      struct Book_Details Book_Bought;
};

Declaration 2

struct Student_Details
{
      char studentname[20];
      int age;
      struct
      {
            char title[30];
            float price;
      }Book_Bought;
};

Variable Declaration
struct Student_Details Student_Record;
```

Above shows 2 ways of declaring a structure in another structure. The 1st declaration declares the *Books_Details* structure separately whereas the 2nd declaration puts the books structure in the student structure. For the 2nd declaration, the books structure is not allowed to have a structure tag.

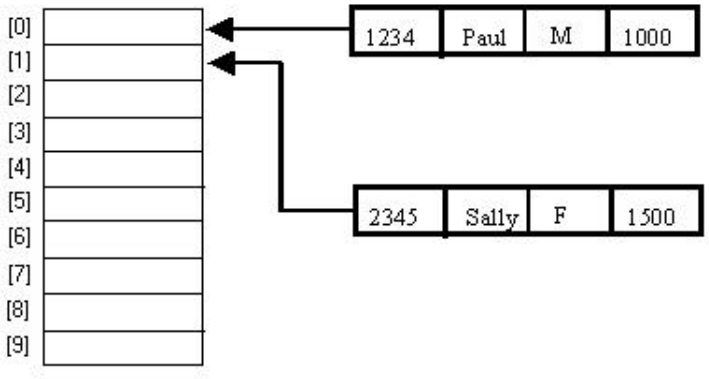
Below is a sample coding to assign the value 120 into price of the book for Student_Record:-

```
Student_Record.Book_Bought.price = 120;
```

1.3.5 Array of Structures

When the two types of addressing(structure and array) may be combined.

From the previous example, we only declare a structure which can only store one record. If we need more than 1, we will need to combine both the data structures and declare array of structures. The figure below shows an array of employee structures.



This model looks exactly like what we would achieve if we were to use a database. It is simply a collection of the employee particulars for a department. And, for each employee, we would need the same information such as the ID, name, Gender and salary.

In order to declare this array of structure,

```
struct EmployeeRecord{
            int ID;
            char Name[20];
            char Gender;
            float Salary;
}

struct EmployeeRecord employee[10];
```

To read data into the array of structure, we would have to use

```
for (a=0;a<10;a++)
{
      printf("Enter your ID:");
      scanf("%d",&emplyee[a].ID);
      printf("Enter your Name:");
      scanf(" %s",&employee[a].Name);
      printf("Enter your Gender:");
      scanf(" %c",&employee[a].Gender);
      printf("Enter your salary:");
      scanf("%f",&employee[a].Salary);
}
```

1.3.6 Structure Parameters

In C, a structure may not be passed to a function by means of a call by value. To pass a structure to a fucntion, we must pass its address to the function, in terms of pass by reference. Refer to an example below

```
#include<stdio.h>

struct nametype{
      char first[10];
      char midinit;
      char last[20];
};

void inputsingle(struct nametype &sname);
void inputmany(struct nametype ename[]);

void main(){
            struct nametype sname,ename[10];
            inputsingle(sname);
            inputmany(ename);
}

void inputsingle(struct nametype &sname)
{
      printf("Enter your first name:");
      scanf(" %s",&sname.first);
      printf("Enter your middle initial name:");
      scanf(" %c",&sname.midinit);
      printf("Enter your last name:");
      scanf(" %s",&sname.last);
}

void inputmany(struct nametype ename[])
{
     int a;
     for(a=0;a<10;a++)
     {
      printf("Enter your first name:");
      scanf(" %s",&ename[a].first);
      printf("Enter your middle initial name:");
      scanf(" %c",&ename[a].midinit);
      printf("Enter your last name:");
      scanf(" %s",&ename[a].last);
     }
}
```

1.4 Unions

A *union* is a data structure that can hold any one of a set of named members. The members of the named set can be of any data type. Members are overlaid in storage. The storage allocated for a union is the storage required for the largest member of the union

1.4.1 Definition and Examples

For example, consider an insurance company that offers three kinds of policies: life, auto, and home. A policy number identifes each insurance policy, of whatever kind. For all three types of insurance, it is necessary to have the policyholder's name, address, the amount of the insurance, and the monthly premium payment. For auto and home insurance policies, a deductible amount is needed. For a life insurance policy, the insured's birth date and beneficiary are needed. For an auto insurance policy, a license number, state, car model, and year are required. For a homeowner's policy, and indication of the age of the house and the presence of any security devices is required. A policy structure type for such a company may be defined as a union.

```
struct addr{
      char street[50];
      char city[10];
      char state[3];
      char zip[6];
};
struct date{
      int month;
      int day;
      int year;
};
struct policy{
      int polnumber;
      char name[30];
      struct addr address;
      int amount;
      float premium;
      int kind;   /*LIFE, AUTO, or HOME*/
      union {
            struct{
                        char beneficiary[30];
                        struct date birthday;
                  }life;
            struct {
                        int autodetect;
                        char license[10];
                        char state[3];
                        char model[15];
                        int year;
                  }auto;
            struct{
                        int homededuct;
                        int yearbuilt;
                  }home;
            }policyinfo;
}

variable declaration
struct policy p;
```

The definition of union consists of two parts; a fixed part and a variable part. The fixed part consists of all member declarations up to the keyword union, while the variable part consists of the remainder of the definition.

From the above declaration, all the members such as polnumber, name, address, amount, premium and kind will have their own memory storage location whereas all the members in the union(policyinfo) will share the same memory storage location.

It is the programmer's responsibility to make sure that the use of a member is consistent with what has been placed into that location. It is a good idea to maintain a separate fixed member in a structure containing a union whose value indicates which alternatively is currently in use. In the above example, the member kind is used for this purpose. If its value is 1, then the structure holds a life insurance policy; if 2, an auto insurance policy; and if 3, a home insurance policy.

```
if (p.kind == 1)
   printf("\n%s %2d//%2d//%4d", p.policyinfo.life.beneficiary,
           p.policyinfo.life.birthday.month,
           p.policyinfo.life.birthday.day,
           p.policyinfo.life.birthday.year);
else
   if (p.kind == 2)
        printf("\n%d %s %s %s %d",p.policyinfo.auto.autodeduct,
                p.policyinfo.auto.license
                p.policyinfo.auto.state
                p.policyinfo.auto.model
                p.policyinfo.auto.year);
   else
        if (p.kind == 3)
                printf("\n%d %d, p.policyinfo.home.homededuct,
                        p.policyinfo.home.yearbuilt);
        else
                printf("\nbad type %d in kind,p.kind);
```

From the above sample, if the value of *p.kind* is 1, *p* currently contains members *beneficiary* and *birthday.* It is invalid to reference *model* or *yearbuilt* while the value of *kind* is *1*.

1.4.2 Implementation of Unions

A structure may be regarded as a road map to an area of memory. It defines how the memory is to be interpreted. A union provides several different road maps for the same area of memory, and it is the responsibility of the programmer to determine which road map is in current use. In practice, the compiler allocates sufficient storage to contain the largest member of the union. It is the road map, however, that determines how that storage is to be interpreted.

```
struct stint{
      int f3, f4;
};
struct stfloat{
      float f5,f6;
};
struct sample{
      int f1;
      float f2;
      int utype;
      union{
            struct stint x;
            struct stfloat y;
      }funion;
};
```

The three fixed members *f1*,*f2* and *utype* occupy 16 bytes. The first member of the union, *x*, requires 8 bytes, while the second member, *y*, requires 16. The memory actually allocated for the union part of such a variable is the maximum of the space needed by any single member. In this case, therefore, 16 bytes are allocated for the union part of *sample*. Added to the 16 bytes needed for the fixed part, 32 bytes are allocated to *sample*.

The different members of a union overlay each other. In the above example, if space for *sample* is allocated starting at location 100, so that *sample* occupies bytes 100 through 131, the fixed members *sample.f1*, *sample.f2* and *sample.utype* occupy bytes 100 through 103, 104 through 111, and 112 through 115, respectively. If the value of the member *utype* is INTEGER (that is 1), bytes 116 through 119 and 120 through 123 are occupied by *sample.funion.x.f3* and *sample.funion.x.f4*, respectively, and bytes 124 through 131 are unused. If the value of *sample.utype* is REAL (that is,2), bytes 116 through 123 are occupied by *sample.funion.y.f5* and bytes 124 through 131 are occupied by *sample.funion.y.f6*. That is why only a single member of a union can exists at a single instant. All the members of the union use the same space, and that space can be used by only one of them at a time.

1.4.3 Differences and similarities between union and structure

Similarities between union and structure

Both of them stores a group of elements of different data type under a single name.

Differences between union and structure

Structure	- Different memory locations are reserved for each member in a structure. - Total memory size for a structure is the added sum of memory bytes for each member - All members can exist in their own reserved memory location at the same time
Union	- All member share the same memory location - The memory size for the union is the size for the member which has needed the largest memory size. - All members cannot exist at the same time because they al share the same memory location

Exercises for Arrays, Structures and Unions

Arrays

1. Given the statements Y = X3 and Y = X[3], supply variable declaration such that these statements are valid. Assume Y is of the type float.
2. What type of data structure can have the expression A[5]. X as a legal representation of one of its members? What does the X represent?
3. Assuming that type integer requires 2 bytes of memory, float requires 8 bytes of memory, char needs 1 byte of memory and type T needs 10 bytes of memory. Specify how many memory bytes needed for the below declaration
    a) char alpha[20];
    b) float salary[10];
    c) int numbers[20];
    d) T sample[40];

Structures & Unions

1. What is the primary difference between a structure and an array? Which would you use to store the catalog description of a course? To store the names of the students in the course?
2. For Employee declared as follows, provide a statement that increase the salary of the Employee by 30%

```
struct EmployeeRecord {
        int ID;
        char name[20];
        float salary;
}
struct EmployeeRecord Employee;
```

3. Declare a record called Subscriber that contains the fields Name, Street_address, MonthlyBill (how much the subscribe owes), and which paper the subscriberbill receives (Morning, Evening, or Both).
4. Assuming that type integer requires 2 bytes of memory, float requires 8 bytes of memory and char needs 1 byte of memory. Calculate the number of memory bytes does the structure Employee and CustomerPolicy requires for question 4 and 5.

```
struct DepartmentRecord{
        int DeptID;
        char DeptName[20];
        char DeptType[30];
}

struct EmployeeRecord{
        int ID;
        char Name[20];
        float salary;
        struct DepartmentRecord Dept;
}

struct EmployeeRecord Employee;
```

5.

```
struct addr{
      char street[50];
      char city[10];
      char state[3];
      char zip[6];
};
struct date{
      int month;
      int day;
      int year;
}
struct policy{
      int polnumber;
      char name[30];
      struct addr address;
      int amount;
      float premium;
      int kind;   /*LIFE, AUTO, or HOME*/
      union {
            struct{
                        char beneficiary[30];
                        struct date birthday;
                  }life;
            struct {
                        int autodetect;
                        char license[10];
                        char state[3];
                        char model[15];
                        int year;
                  }auto;
            struct{
                        int homededuct;
                        int yearbuilt;
                  }home;
            }policyinfo;
}

struct policy CustomerPolicy;
```