CPNA Lecture 13 - Structures and Unions

Mridul Sankar Barik

Jadavpur University

2023

Introduction

- A structure is a collection of values, possibly of different types
- A union is same as a structure, except that its members share the same storage

Structures I

- Collections of related variables (aggregates) under one name
- Can contain variables of different data types
- Commonly used to define records to be stored in files
- Combined with pointers, can create linked lists, stacks, queues, and trees
- Structures are also known as records and members as fields
- Arrays vs. Structures
  - All elements of an array has the same type
  - To select an array element we have to specify its position or index
  - Members of a structure need not be of same type
  - Members of a structure can be accessed by its name

Structures II

- Example:

```
struct student{
    char name[40];
    int roll;
    int age;
}
...
printf("%d\n", sizeof(struct student)); /*Output 48*/
```

- Keyword `struct` defines a structure, followed by the name of the structure `student`
- `name`, `roll`, age are members of the structure `student`
- Members of structure are stored in memory in the order in which they are declared
- Each structure defines a new scope

Structures III

  - Names declared in that scope won't conflict with other names in a program
  - Each structure has a separate name space for its members
- A structure cannot contain an instance of itself
- A structure can contain a member that is a pointer to the same structure type
- A structure definition does not reserve space in memory, instead creates a new data type used to declare structure variables

Structures IV

- Structure declarations

```
struct student{
    char name[40];
    int roll;
    int age;
} s1, students[20], *p;
OR
struct student s1, students[20], *p;
```

- Structure initialization

```
struct student s1 = {"ABC", 1234, 18};
...
struct student s2 = s1;
...
struct student s3;
s3.name = "XYZ";
```

Structures V

```
s3.roll = 1234;
s3.age = 20;
```

- Initializers must be constant expressions
- Initializers can have fewer members; Leftover members are given 0 as their initial value (0.0 for float and empty string for string members)
- Designated initializers {`.age=19, .name="PQR", .roll=1024`}
- The combination of period and member name is called a designator
- Order of members is not important

Operations on Structure

- Dot operator: access a member within a structure
- Name of the structure variable, period, name of the member

```
printf("Student name : %s\n", s1.name);
printf("Student Roll : %d\n", s1.roll);
printf("Student age : %d\n", s1.age);
```

- Assignment operator `s1 = s2;`
- Arrays cannot be copied using '=' operator; Solution: Create dummy structures to enclose arrays that will be copied later
- Can be used only with structures of compatible type
- Can not use the == or != operators

Accessing Members of Structures

- Dot operator (.) used with structure variables

```
struct student s1;
printf("%s", s1.name);
```

- Arrow operator (->) used with pointers to structure variables

```
struct student *p = &s1;
printf("%s", p->name);
printf("%s", (*p).name);
```

Structures and Functions

- Passing structure to functions
  - Pass entire structure OR pass individual members: Both are Call by Value
  - Pass pointers to structures
- To pass arrays as Call by Value
  - Create a structure with the array as a member
  - Pass the structure

typedef

- Creates synonyms (aliases) for previously defined data types
- Use `typedef` to create shorter type names
- Example:

```
typedef struct{
    char name[40];
    int roll;
    int age;
} student;

typedef student *studentPtr;

student s1, s2;
studentPtr p1, p2;
```

- `typedef` does not create new data types

Structures as Arguments and Return Values I

- Functions may have structures as arguments

```
void printStudent(student s){
    printf("Roll : %d\n", s.roll);
    printf("Age : %s\n", s.age);
    printf("Name : %d", s.name);
}
...
student s1 = {"abc", 1234, 20};
printStudent(s1);
```

Structures as Arguments and Return Values II

- Functions may have structures as return values

```
student buildStudent(char *name, int roll, int age){
    student s;
    strcpy(s.name, name);
    s.roll=roll;
    s.age=age;
    return(s);
}
...
student s1 = buildStudent("abc", 1234, 20);
```

Nested Structures I

- Example

```
typedef struct{
    char first[20];
    char middleInitial;
    char last[20];
} personName;

typedef struct{
    personName name;
    int roll;
    int age;
} student;

student s1, s2;
strcpy(s1.name.first, "ABC");
s1.name.middleInitial='C';
strcpy(s1.name.last, "XYZ");
```

Arrays of Structures I

- Example

```
student BCSE[70];
...
for(i=0; i<70;i++)
    printStudent(BCSE[i]);
...
BCSE[10].age=20;
strcpy(BCSE[10].name.first, "ABC");
```

Unions I

- Compiler allocates space for the largest of the members
- Assigning a new value to one member alters the values of the other members as well
- Union declarations $\Rightarrow$ same as structures

```
typedef union{
  int x;
  unsigned char c[4];
} Number;

Number n;
int i;

n.x=10;
for(i=0;i<4;i++)
    printf("%d ", n.c[i]); /*Output 10 0 0 0*/
```

- Only the first member of a union can be given an initial value

Unions II

- Designated initializer can be used to initialize only one member which need not be the first one
- Size of a union is the size of it's largest member

Endianness

- Endianness refers to the sequential order in which bytes are arranged into larger numerical values when stored in memory or when transmitted over network
  - **Big-endian**: Low order byte is at the highest address
  - **Little-endian**: Low order byte is at the lowest address
- To find out endianness of a machine

```
unsigned int i = 1;
char *c = (char*)&i;
if (*c)
    printf("Little endian");
else
    printf("Big endian");
```