B.C.A. Semester – 2

UNIT - 2

Structures & Unions

BCA -201

Advance Programming Language ‘C’

Structures & Unions

Introduction:

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures ca; simulate the use of classes and templates as it can store various information.

The **struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

```
struct structure_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

Declaring structure variable:

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

- By struct keyword within main() function
- By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{
  int id;
  char name[50];
  float salary;
};
```

Now write given code inside the main() function.

struct employee e1, e2;

2nd way:

Let's see another way to declare variables at the time of defining the structure.

struct employee

{ int id;

char name[50];

float salary;

}e1,e2;

Accessing members of the structure

There are two ways to access structure members:

- By . (member or dot operator)
- By -> (structure pointer operator)

Let's see the code to access the id member of e1 variable by. (member) operator.

e1.id

Let's see a simple example of structure in C language.

```c
#include <stdio.h>
#include <string.h>
struct employee
{
    int id;
    char name[50];
} e1; // declaring e1 variable for structure
int main()
{
    // store first employee information
    e1.id = 101;
    strcpy(e1.name, "Tony Stark"); // copying string into char array
    // printing first employee information
    printf("employee 1 id : %d\n", e1.id);
    printf("employee 1 name : %s\n", e1.name);
    return 0;
}
```

Output:
employee 1 id : 101
employee 1 name : Tony Stark

Copying and Comparison of structures variables

Copying and comparison of structure variables in C can be done using the assignment operator (=) and the equality operator (==) respectively.

Here's an example:

```c
#include <stdio.h>
#include <string.h>
struct Person // Create a struct called Person
{
  char name[20];
  int age;
}; // Don't forget the semicolon

void main()
{
  struct Person p1;
  struct Person p2;
  strcpy(p1.name, "Naruto"); // Store "Naruto" in p1.name
  strcpy(p2.name, "Itachi"); // Store "Itachi" in p2.name
  p1.age = 20;
  printf("Name: %s\n", p1.name);
  printf("Age: %d\n", p1.age);

  if (strcmp(p1.name, p2.name) == 0) // Compare p1.name and p2.name
  {
    printf("Naruto is equal to Itachi\n"); // If they are equal,
print this
  }
  else
  {
    printf("Naruto is not equal to Itachi\n");
  }
}
```

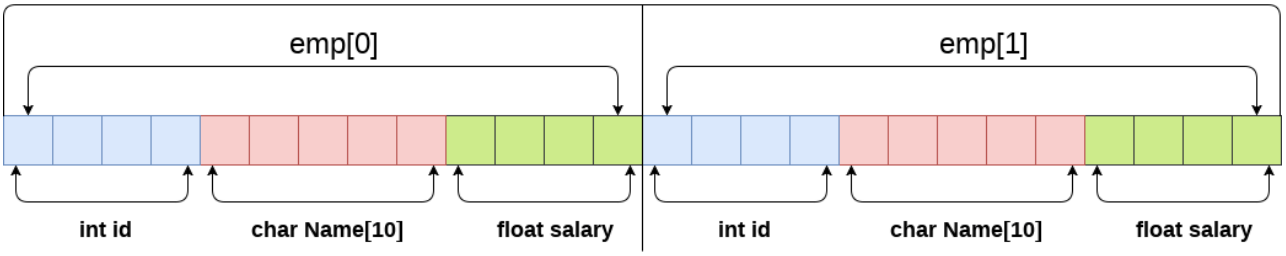
Output

```
Name: Naruto
Age: 20
Naruto is not equal to Itachi
```

Arrays of structures

An array of structures in C can be defined as the collection of multiple structure variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Array of structures




```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Here is an example program that demonstrates simple arrays of structures in C:

```c
#include <stdio.h>
#include <string.h>

struct Person {
    char name[20];
    int age;
};
void main() {
    struct Person people[3]; // declare an array of 3 Person structures
    int i;

    // initialize the elements of the array
    strcpy(people[0].name, "Itachi");
    people[0].age = 25;

    strcpy(people[1].name, "Naruto");
    people[1].age = 30;
```

```
    strcpy(people[2].name, "Sasuke");
    people[2].age = 35;

    // print the contents of the array
    for (i = 0; i < 3; i++) {
        printf("Person %d: ", i );
        printf("Name: %s, " ,people[i].name);
        printf("Age: %d ", people[i].age);
        printf("\n");
    }
getch();
}
```

Output:

Person 0: Name: Itachi, Age: 25
Person 1: Name: Naruto, Age: 30
Person 2: Name: Sasuke, Age: 35

Arrays within structures

we can have structure members of type arrays. Any structure having an array as a structure member is known as an array within the structure. We can have as many members of the type array as we want in C structure.

Example of using an array within a structure in C:

```
#include <stdio.h>
struct Person
{
  char name[20];
  int age;
  int marks[3];
};
void main()
{
  struct Person p1 = {"John", 25, {80, 90, 85}};
  printf("Name: %s\n", p1.name);
  printf("Age: %d\n", p1.age);
  printf("Marks: %d %d %d\n", p1.marks[0], p1.marks[1], p1.marks[2]);
  getch();
}
```

Output:

Name: John

Age: 25

Marks: 80 90 85

Structures within Structures

Structures within structures, also known as nested structures, refer to the concept of defining a structure within another structure. This means that one structure can have another structure as its member.

Example:

```c
#include <stdio.h>
#include <string.h>
struct Date
{
  int day;
  int month;
  int year;
};
struct Person
{
  char name[10];
  int age;
  struct Date dob;
};
void main()
{
  struct Person p1;
  strcpy(p1.name, "John");
  p1.age = 22;
  p1.dob.day = 18;
  p1.dob.month = 2;
  p1.dob.year = 2000;
  printf("Name: %s\n", p1.name);
  printf("Age: %d\n", p1.age);
  printf("DOB: %d-%d-%d\n", p1.dob.day, p1.dob.month, p1.dob.year);
}
```

Output:

```
Name: John
Age: 22
DOB: 18-2-2000
```

Structures and functions

When structures and functions are combined, it allows for the creation of more complex and modular programs. Functions can be used to perform operations on structures, such as initializing them or modifying their values. Structures can also be used as parameters in function calls or as return values.

```c
#include <stdio.h>
struct Person
{
    char name[50];
    int age;
};
void printPerson(struct Person p) // pass by value (copy)
{
    printf("Name: %s\n", p.name);
    printf("Age: %d\n", p.age);
}
void main()
{
    struct Person john = {"John", 22};
    printPerson(john);
}
```

Output

```
Name: John
Age: 22
```

Difference between Structure and Array

	Array	Structure
Definition	**Collection of elements of homogeneous data type**	**Collection of elements of heterogeneous data type**
Access	**Uses subscripts [ ] for element access**	**Uses the dot . operator for element access**
Pointer	**Array is a pointer as it points to the first element of the collection**	**Structure is not a pointer**

Data type	Array is a non-primitive data type	Structure is a user-defined data type
Memory	Array elements are stored in contiguous memory locations	Structure elements may or may not be stored in a contiguous memory location

Union

- A union is a data structure in C that allows multiple variables to be stored in the same memory location.
- However, unions can also be difficult to use correctly and can lead to unexpected behavior if not handled properly.
- Unions are typically defined using the union keyword, followed by the name of the union and a list of member variables.
- Each member variable can have a different type, but they all share the same memory location.
- When a value is assigned to one member variable, the values of the other member variables may be changed as well.

Syntax of Union

union [union tag] {

member definition;

member definition;

...

} [one or more union variables];

Example

```
#include <stdio.h>
#include <string.h>

union Data
```

```
{
    int i;
    float f;
    char str[20];
};

void main()
{

    union Data data;

    printf("Memory size occupied by data : %d \n", sizeof(data));
}
```

Output:

Memory size occupied by data : 20

This program defines a union named Data with three member variables: an integer i, a float f, and a character array str. The program then creates a union variable named data and prints the size of the union using the sizeof operator. Since all three member variables share the same memory location, the size of the union is equal to the size of its largest member variable, which in this case is the character array str.

Advantages of unions

- Unions can be used to conserve memory by allowing multiple variables to share the same memory location.
- They can also simplify code by providing a way to interpret the same data in different ways.

Disadvantages of unions

- Unions can be difficult to use correctly and can lead to unexpected behavior if not handled properly.
- They can also make code less readable and more confusing, especially if the union contains many different member variables.

Size of structures

The sizeof operator in C is used to determine the size of a data type or variable in bytes. It can be used with both structures and unions.

Here is an example of using sizeof with a structure:

```
#include <stdio.h>
struct person
{
    char name[20];
    int age;
    float height;
};

void main()
{
    struct person p;
    printf("Size of struct person: %lu bytes\n", sizeof(struct person));
    printf("Size of variable p: %lu bytes", sizeof(p));
}
Output:
```

In this example, we define a structure person with three member variables: a character array name, an integer age, and a float height. In the main function, we create a variable p of type person and print the sizes of the structure and the variable using the sizeof operator. The output will depend on the size of the data types on your machine, but might look something like this:

Size of struct person: 28 bytes
Size of variable p: 28 bytes

This shows that the size of the person structure is 28 bytes (on this machine), and that the size of the p variable is also 28 bytes, since it contains all three member variables.

Here is an example of using sizeof with a union:

```
#include <stdio.h>
union person
{
    char name[20];
    int age;
    float height;
};
void main()
{
    union person p;
    printf("Size of struct person: %lu bytes\n", sizeof(union person));
    printf("Size of variable p: %lu bytes\n", sizeof(p));
}
```

Output:
Size of struct person: 20 bytes
Size of variable p: 20 bytes

This shows that the size of the data union is 4 bytes (on this machine), since all three member variables share the same memory location, and that the size of the d variable is also 4 bytes, since it contains all three member variables.

Bit fields

Definition

A bit field is a data structure in C that allows you to specify how many bits a variable should occupy in memory. This can be useful for conserving memory and for packing data into a smaller space.

How they work

Bit fields are typically defined using the struct keyword, followed by the name of the structure and a list of member variables. Each member variable is assigned a number of bits to occupy in memory. When a value is assigned to a bit field, it is automatically packed into the appropriate number of bits.

Example

Here is an example of a struct that uses bit fields to pack data into a smaller space:

```
#include <stdio.h>
struct date
{
    unsigned int day:5; // 5 bits for day because 31 is 11111 in binary, unsigned int because we don't want negative numbers, if we don't specify unsigned int,
    unsigned int month:4;  // 4 bits for month because 12 is 1100 in binary.
    unsigned int year; // 32 bits for year
    // Total saved space is 41 bits, 41 bits / 8 = 5.125, 5.125 rounded up to 8 bytes.
};

void main()
{
    struct date today = { 28, 02, 2023 };
    printf("Size of structure today is %d bytes \n\n" , sizeof(today));
    // 8 bytes because 5 + 4 + 32 = 41 bits, 41 bits / 8 = 5.125, 5.125 rounded up to 8 bytes, rounded up to 8 bytes.

    printf("Today is %d/%d/%d ", today.day, today.month, today.year);
}
Output:
Size of structure today is 8 bytes
```

Advantages of bit fields

- Bit fields can be used to conserve memory by packing data into a smaller space.
- They can also simplify code by providing a way to work with data at the bit level.

Disadvantages of bit fields

- Bit fields can be difficult to use correctly and can lead to unexpected behavior if not handled properly.
- They can also make code less readable and more confusing, especially if the structure contains many different bit fields.