F.Y.B.C.A. (SCIENCE) SEM-II
ADVANCED C PROGRAMMING

Unit 5

Advanced Features

MRS. MADHURI ABHIJIT DAREKAR
ASSISTANT PROFESSOR
MIT ARTS, COMMERCE AND SCIENCE COLLEGE, ALANDI (D)

- **Union in C**
- Like Structures, union is a **user defined data type**.
- C Union is a collection of different data types which are grouped together. **Each element in a union is called member**.
- But in union, **all members share the same memory location. Hence only one of the members is active at a time.**

- **Definition:**

  A union is a variable that contains multiple members of possibly different data types grouped together under a single name. However all of them share the same memory area. Hence only one of the members is active at a time.

- **Defining a union**
- Union can be defined in same manner as structures, for defining union use **union** keyword

```
union  <union-name>
{       <data_type>  <data_member1>;
        <data_type>  <data_member2>;
        ...
} ;
```

- **Declaration of a union:**

```
union StudentInfo
{           int Rno;
        char Sname[20];
}u1;
union StudentInfo *ptr;
```

  Here either Rno or Sname can be accessed or used at a time. **Both members do not exist simultaneously.**

- Union **allocates one common memory space for its all members**. Union find which member need more memory than other member, then it allocate that much memory space.
- Here **‘u1’ union variable will allocate 20 bytes** of memory space.
- **Accessing members of an union: (using dot operator)**

  The member of unions can be accessed in similar manner as Structure.

  Here to access ‘Sname’ member use u1.Sname.

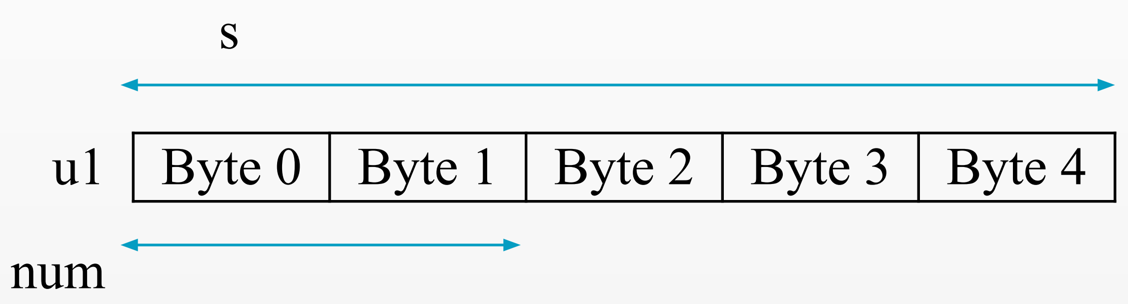
  Accessing members of an union: (using pointer)

  ptr=&u1;

  Here to access ‘Sname’ member use ptr->Sname.

Example:

```
union u
{  char s[5];
   int num;
}u1;
```



In example, the **variable u1 will be allocated memory to accommodate the largest member of union,** which is 5 bytes. Only the member, i.e., either the string or the integer num will be stored and can be accessed at a time. Both do not exist simultaneously. **Only one member of a union can be used at a time.**

Let us consider an example to store smart-phone information.

i. Name of company
ii. IMEI number
iii. Type of phone (Android or IPhone A/I)
iv. If Android, store Android OS version name.
v. If iPhone, store iPhone version number.

```
struct smartphone
{     char company[20];
      char IMEI[16];
      char type;
      union u
      {     char android_os_name[20];
            int iphoneversion;
      }info;
}s1;
```

Here union allows only one member to be stored i.e. either android_os_name or iphoneversion.

Example:

```
s1.company="Samsung";
s1.IMEI="152076249800002";
s1.type='A';
s1.info.android_os_name="Marshmallow";
```

Initializing a Union:

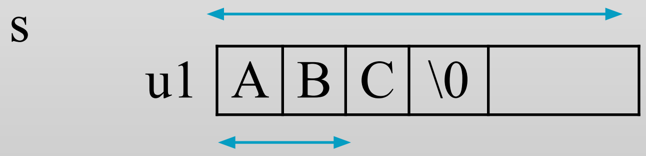
The initialization operation assigns a value to a variable when it is declared. In the case of a union, **since only one member of the union can be used at a time, only one member can be initialized**. The initializer for the union is a value for the first member of the union, i.e., **only the first member can be initialized.**

Example:

```
union u
{
    char s[5];
    int num;
};
```

Here, the first member is string. Hence only the string can be initialized.

union u u1={“ABC”};



Difference between Structures and Unions

Structures	Unions
To define structure use struct keyword.	To define union use union keyword.
Structure occupies more memory space than union.	Union occupies less memory space than Structure.
In Structure we can access all members of structure at a time.	In union we can access only one member of union at a time.
Structure allocates separate memory space for its every members.	Union allocates one common memory space for its all members.
The size of a structure variable is the sum of sizes of all members of the structure.	The size of an union is the size of the largest member of the union.
All members of the structure can be initialized.	Only the first member of the union can be initialized.
Structure members start at different memory locations.	Union members start at the same location in memory.

- **C Program to store and access "name, subject and percentage“ of 2 students (using union)**

```
#include<stdio.h>
#include<string.h>
union student
{
    char name[20];
    char subject[20];
    float per;
};
void main()
{
    union student record2;
    union student record1;
    strcpy(record1.name,"Ram");
    strcpy(record1.subject,"Maths");
    record1.per=86.50;

    printf("\nUnion record1");
    printf("\n Name: %s",record1.name);
    printf("\n Subject: %s",record1.subject);
    printf("\n Percentage: %.2f",record1.per);
    printf("\n\nUnion record2");
    strcpy(record2.name,"Malini");
    printf("\n Name: %s", record2.name);
    strcpy(record2.subject,"Physics");
    printf("\n Subject: %s", record2.subject);
    record2.per=99.50;
    printf("\n Percentage: %.2f", record2.per);
}
```

Here this program shows that in union we can access only one member of union at a time.

Output:

```
Union record1
 Name:
 Subject:
 Percentage: 86.50

Union record2
 Name: Malini
 Subject: Physics
 Percentage: 99.50
```

C Program using structures within union:

```
#include<stdio.h>
struct StudentInfo
{   int Rno;
    char Sname[20];
    float percentage;
};
union StudUnion
{  struct StudentInfo s1;
}u1;
void main()
{   printf("Enter  student details:");
    printf("\nEnter Student Roll no : ");
    scanf("%d", &u1.s1.Rno);
    printf("\nEnter Student Name  :  ");
    scanf("%s", u1.s1.Sname);
    printf("\nEnter  Student percentage :");
    scanf("%f", &u1.s1.percentage);
    printf("\nThe student details are : \n");
    printf("\n Student Rollno : %d", u1.s1.Rno);
    printf("\n Student name : %s", u1.s1.Sname);
    printf("\n Student Percentage : %.2f", u1.s1.percentage);
}
```

Output:

```
Enter  student details:
Enter Student Roll no : 1
Enter Student Name  :  aaaa

Enter  Student percentage :89

The student details are :

 Student Rollno : 1
 Student name : aaaa
 Student Percentage : 89.00
```

- **Union within structures:**
  A union can be nested within a structure.
  Suppose we wish to store employee information, name, id and designation. If the designation is ‘M’ (for manager) we want to store the number of departments he manages and if his designation is ‘W’ (for worker), the department name should be stored.

  Example:

```
struct employee
{
  char name[20];
  int id;
  char desig;
  union info
  {
       int no_of_depts;
       char deptname[20];
  }details;
}emp;
```

- **C program to demonstrate example of pointer to union**

```
#include <stdio.h>
int main()
{           union number
    {           int a;
            int b;
    };
    union number n = { 10 };
    union number* ptr = &n;
    printf("a = %d\n", ptr->a);
    ptr->a = 20;
    printf("a = %d\n", ptr->a);
    ptr->b = 30;
    printf("b = %d\n", ptr->b);
    return 0;
}
```

Output:

```
a = 10
a = 20
b = 30
```

- **Nested Union** is Union which has another Union as a member in that Union. A member of Union can be Union itself , this what we call as Nested Union.
- **Example:**

```
union student
{       int rollno;
        char div;
        union name
        {
                char first_name[20];
                char last_name[20];
        }n1;
}s1;
#include<stdio.h>
#include<string.h>
void main()
{
    s1.rollno = 10;
    s1.div = 'A';
    strcpy(s1.n1.first_name, "Ram");
    strcpy(s1.n1.last_name, "Patil");
}
```

Enumeration (enum)

Enum is user defined data type like structure, union. It is used to define a set of constants of type int. To declare enum, we use keyword **enum**.

Syntax :

enum tag{enumeration list};

Example:

enum day { MON, TUE, WED, THU, FRI, SAT, SUN };

Here name of enumeration is **day.**

And MON, TUE, WED,.... are the values of type **day. By default value of, enum element MON is 0**, TUE is 1 and so on. This is done by compiler and it starts from 0, **You can change default values of enum elements during declaration (if necessary).**

Example: enum type {batsman=1, bowler=2, wkeeper=3};

Since **enumerated data types** are integers, they can be **used** anywhere integers are allowed. One of the best places is **in**

Example:

```
#include <stdio.h>
enum week{Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun};
enum day{Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund};
int main()
{
  printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",Mon ,
Tue, Wed, Thur, Fri, Sat, Sun);
  printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",Mond ,
     Tues, Wedn, Thurs, Frid, Satu, Sund);
  return 0;
}
```

Output:

```
The value of enum week: 10 11 12 13 10 16 17
The default value of enum day: 0 1 2 3 18 11 12
```

Bit Fields:

- In C, we can **specify size (in bits) of structure and union members**. The idea is to **use memory efficiently** when we know that the value of a field or group of fields will never exceed a limit or is within a small range.
- For example, consider the following declaration of date without the use of bit fields.

```
 struct date
 {
    unsigned int d;
    unsigned int m;
    unsigned int y;
 };
```

The above representation of ‘date’ takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, the value of m is from 1 to 12, we can optimize the space using bit fields.

C program using Bit Field to optimize space.

```
#include <stdio.h>
struct date
{    // d has value between 1 and 31, so 5 bits are sufficient
     unsigned int d : 5;

     // m has value between 1 and 12, so 4 bits are sufficient
     unsigned int m : 4;
     unsigned int y;
};
int main()
{
     printf("Size of date is %lu bytes\n", sizeof(struct date));
     struct date dt = { 31, 12, 2020 };
     printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
     return 0;
}
```

Output:

```
Size of date is 8 bytes
Date is 31/12/2020
```

Here the above representation of 'date' takes only 8 bytes since we are using concept of Bit field

- **Multi-file programs:**
- If a **program contains many functions** then it is difficult to read and maintain because when the number of functions increases the size of the program/file will also increase. To avoid this some functions can be written in separate file.
- The **individual files will be compiled separately and then linked together** to form one executable object program.
- This will **help to editing and debugging** because each file can be maintained at a manageable size.
- Multi-file programs are those **programs where code is distributed into multiple files communicating with each other** this is **more practical and realistic approach towards advanced programming** where we want loosely coupled code module that can communicate with each other.

C Program using Multiple files:

//create first file of name addsub_header.h

```
#include <stdio.h>
int add (int, int);
int sub (int, int);
```

//create second file of name addsub_impl.c

```
#include "addsub_header.h"
int add(int a, int b)
{
   return a+b;
}
int sub(int a, int b)
{
   return a-b;
}
```

```
//create third file of name addsub_driver.c
#include “addsub_impl.c”
int main( )
{
printf(“%d”, add(2,3));
printf(“%d”, sub(9,4));
return 0;
}
```

To execute addsub_driver.c file, compile first and second file individually and then compile and execute third file.