"Advanced C Programming"

Structures, Unions & Bit-Fields

(a) Structures

Structures

Definition:

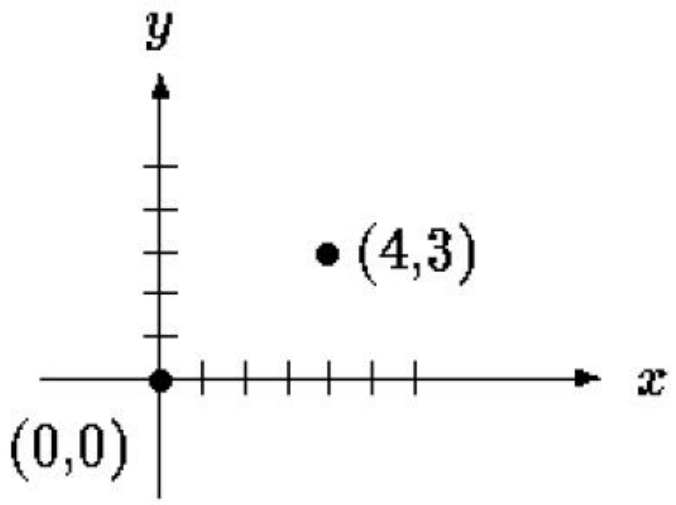
A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

```
Syntax:
    struct  structure-name {
        component-definition
        [component-definition] . . .
        };
```

- It helps to organize complex data
- The keyword “struct” introduces a structure declaration
- Structure Tag – Name of the structure
- Members – The variables of the structure

Structures - Example

Representing a point in a coordinate system:



Note:

- Same member name can occur in different structure
- A structure declaration that is not followed by a list of variables reserves no storage.

```
/* Structure declaration */
struct point {
   int  x;
   int  y;
};

/* Define the instances of the
 structure */
struct point p1, p2;

/* Structure initialization */
struct p1 = { 4, 3 };

/* Accessing members */
Printf("%d, %d", p1.x, p1.y);
```

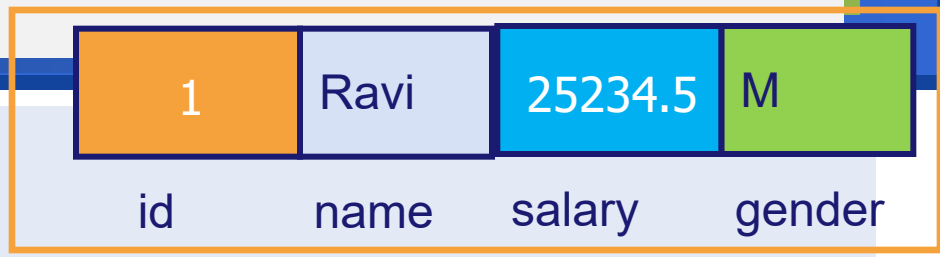
Structures - Rules

- The individual members can be ordinary variables, pointers, arrays, or other structures.
- A member name can be the same as the name of a variable that is defined outside of the structure.
- A storage class, however, cannot be assigned to an individual member
- Individual members cannot be initialized within a structure type declaration.

Structures

```
#include <stdio.h>
#include <string.h>
struct employee {
        int id;
        char name[20];
        float salary;
        char gender;
};
```

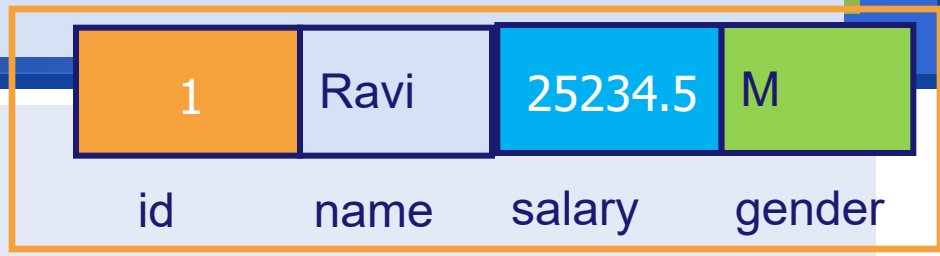
```
int main()
{
        struct employee emp1;
        emp1.id=1;
        strcpy(emp1.name, "Ravi");
        emp1.salary = 25234.5;
        emp1.gender = 'M';
        printf(" Emp id  is: %d \n", emp1.id);
        printf(" Emp Name is: %s \n", emp1.name);
        printf(" Emp salary is: %f \n", emp1.salary);
        printf(" gender is: %c \n", emp1.gender);
        return 0;
}
```



Structures

```
#include <stdio.h>
#include <string.h>
struct employee {
        int id;
        char name[20];
        float salary;
        char gender;
 }emp1;
```

```
int main()
 {
        emp1.id=1;
        strcpy(emp1.name, "Ravi");
        emp1.salary = 25234.5;
        emp1.gender = 'M';
        printf(" Emp id  is: %d \n", emp1.id);
        printf(" Emp Name is: %s \n", emp1.name);
        printf(" Emp salary is: %f \n", emp1.salary);
        printf(" gender is: %c \n", emp1.gender);
        return 0;
 }
```



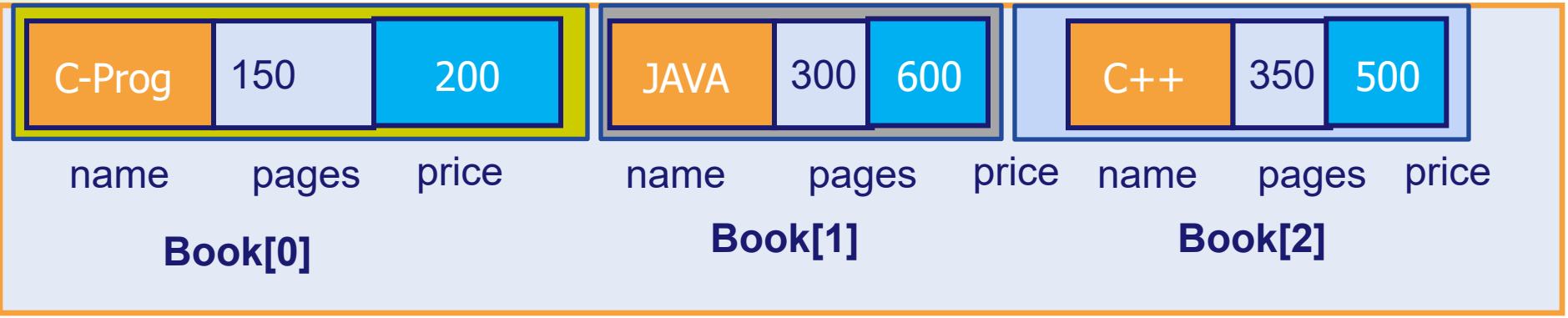
Nested Structures


```
#include <stdio.h>
#include <string.h>
struct group_dtl
{
int grp_id;
char grp_name[10];
};
struct employee {
  int id;
  char name[20];
  float salary;
  struct group_dtl
grp_dtl;
 };
int main()
 {
      struct employee emp1;
      emp1.id=1;
      strcpy(emp1.name, "Ravi");
      emp1.salary = 25234.5;
      emp1.grp_dtl.grp_id=1;
      strcpy(emp1.grp_dtl.grp_name,"embeded");;
      printf(" Emp id  is: %d \n", emp1.id);
      printf(" Emp Name is: %s \n", emp1.name);
      printf(" Emp salary is: %f \n", emp1.salary);
      printf(" Emp group id is: %d \n",
emp1.grp_dtl.grp_id);
      printf(" Emp group name is: %s \n",
emp1.grp_dtl.grp_name);
```

Array of structures

An **array of structures** is simply an array in which each element is a structure of the same type.

```
/* Array of Structures declaration */
struct bookinfo
{
        char bookname[100];
        int pages;
        int price;
}book[3];
/* Accessing the price of second book */
book[1].price;
```



Book

Array of structures…

Example:

```
/* Read and print the details of three emplyoees */

#include <stdio.h>
struct employee {
        int id;
        char name[20];
        float salary;
 }
struct employee emp[10];
```

Array of structures…

Example: Contd…

```
int main()
 {
        struct employee emp[10];
        int i;
        for (i=0; i<3;i++)
        {
                printf("Record %d",i+1);
                printf("\nenter  emplyoee id:   ");
                scanf("%d",&emp[i].id);
                printf("\nenter  emplyoee name:   ");
                scanf("%s",emp[i].name);
                printf("\nenter  emplyoee salary:  ");
                scanf("%f",&emp[i].salary);
        }
        for (i=0; i<3;i++)
        {
        printf(" \n Emp id  is: %d \n", emp[i].id);
        printf(" Emp Name is: %s \n", emp[i].name);
        printf(" Emp salary is: %f \n", emp[i].salary);
        }
        return 0;
}
```

Structures and functions

```
#include <stdio.h>
#include <string.h>
struct employee {
        int id;
        char name[20];
        float salary;
        char gender;
 };
void print_details(struct employee e)
{
        printf(" Emp id  is: %d \n", e.id);
        printf(" Emp Name is: %s \n", e.name);
        printf(" Emp salary is: %f \n", e.salary);
        printf(" gender is: %c \n", e.gender);
}
```

```
int main()
{
        struct employee emp1;
        emp1.id=1;
        strcpy(emp1.name, "Ravi");
        emp1.salary = 25234.5;
        emp1.gender = 'M';
        print_details(emp1);
        return 0;
}
```

Structures and pointers

To access members of a structure using pointers using -> operato

```
#include <stdio.h>
#include <string.h>
struct employee {
        int id;
        char name[20];
        float salary;
        char gender;
 };
```

```
int main()
 {
        struct employee emp1;
        struct employee *ptr;
        emp1.id=1;
        strcpy(emp1.name, "Ravi");
        emp1.salary = 25234.5;
        emp1.gender = 'M';
        ptr=&emp1;
        printf(" Emp id  is: %d \n", ptr->id);
        printf(" Emp Name is: %s \n", ptr->name);
        printf(" Emp salary is: %f \n", ptr->salary);
        printf(" gender is: %c \n", ptr->gender);
        return 0;
```

Pointer to the array of structures

Pointer to array of structures stores the base address of the structure array

Example:

```
struct hockey
{
    char team1[10];
    char team2[10];
    char venue[20];
    int result;
}match[4] = {
                {"IND","AUS","BANGALROE",1},
                {"IND","PAK","HYDERBAD",1},
                {"IND","NZ","CHENNAI",0},
                {"IND","SA","DELHI",1}
            };
```

Pointer to the array of structures…

Example: Contd..

```
void main()
{
struct hockey *ptr = match;// By default match[0],base address
int i;
for(i=0;i<4;i++)
    {
    printf("\nMatch : %d",i+1);
    printf("\n%s Vs %s",ptr->team1,ptr->team2);
    printf("\nPlace : %s\n",ptr->venue);
    if(match[i].result == 1)
        printf("nWinner : %s",ptr->team1);
    else
        printf("nWinner : %s",ptr->team2);
    printf("\n");

    // Move Pointer to next structure element
    ptr++;
    }
}
```

Using typedef

What is typedef?

- Typedef is a keyword used to give a new name for the existing name in a C program.
- **Advantages:** More readability, Easier to modify

Example 1: With structures

```
typedef struct student
{
   int id;
   char name[20];
   float percentage;
} status;

status record;
```

```
struct student
{
   int id;
   char name[20];
   float percentage;
};

typedef struct student
status;
status record;
```

Assignments

- Student record management System
- Write a C program to keep records and perform statistical analysis for a class of 20 students. The information of each student contains ID, Name, Sex, marks (mid-term score, final score, and total score). The program will prompt the user to choose the operation of records from a menu as shown below:
  - Menu
    1. Add student records
    2. View all student records
    3. Calculate an average of a selected student's scores.

(b) Bit-fields

Bitwise Operations

- What is Memory?
  - Collection of Bits
- In real life applications, some times it is necessary to deal with memory bit by bit
- For example,
  - Gaming and Puzzles (Ex: Sudoku)
  - Controlling attached devices (Ex: Printers)
  - Obtaining status
  - Checking buffer overflows…
- Note: The combination of bit level operators and the pointers can replace the assembly code. For example, only 10% of UNIX is written using assembly code and the rest is in C.

Bitwise Operations in Integers

There are six operators

- & – AND
  - Result is **1** if both operand bits are **1**
- | – OR
  - Result is **1** if either operand bit is **1**
- ^ – Exclusive OR
  - Result is **1** if operand bits are different
- ~ – Complement
  - Each bit is reversed
- << – Shift left
  - Multiply by 2
- >> – Shift right
  - Divide by 2

Restrictions: We can use these operators only on int and char data typed variables - Signed and unsigned char, short, int, long, long long

Bitwise Operations in Integers…

	a	0	0	1	1
	b	0	1	0	1
and	a & b	0	0	0	1
or	a | b	0	1	1	1
exclusive or	a ^ b	0	1	1	0
one's complement	~a	1	1	0	0

Examples - &, |, ^ and ~

```
unsigned short int a,b;
unsigned short int c;
```

a = 0xb786

b = 0xb420

c = a&b = 0xb400

c = a|b = 0xb765

c = a^b = 0x0365

c = ~a = 0x4879

Example - Left Shift (<<)

```
unsigned short int a = 0xb786;
A << = 3;
```

Before shift

a= 0xb786

After shift

a= 0x9a30

Last three bits are filled with zeroes

Example - Right Shift (<<)

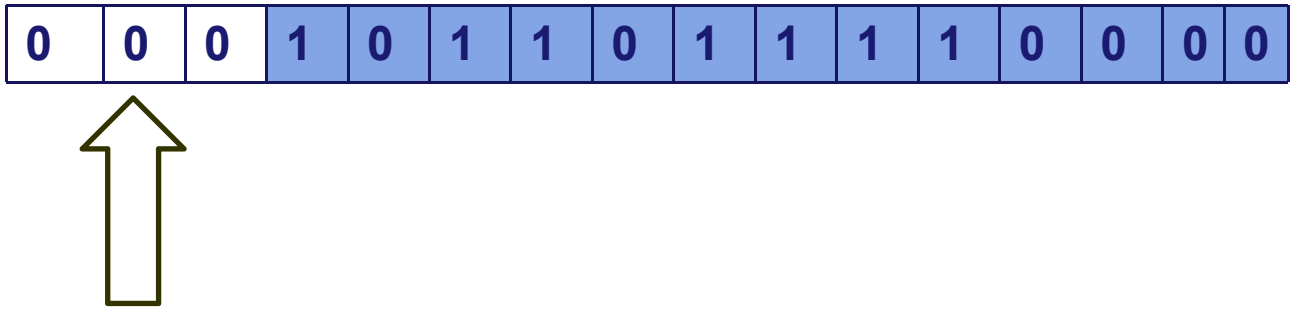
```
unsigned short int a = 0xb786;
A << = 3;
```

Before shift

a= 0xb786

After shift

a= 0x16f0



First three bits are filled with zeroes

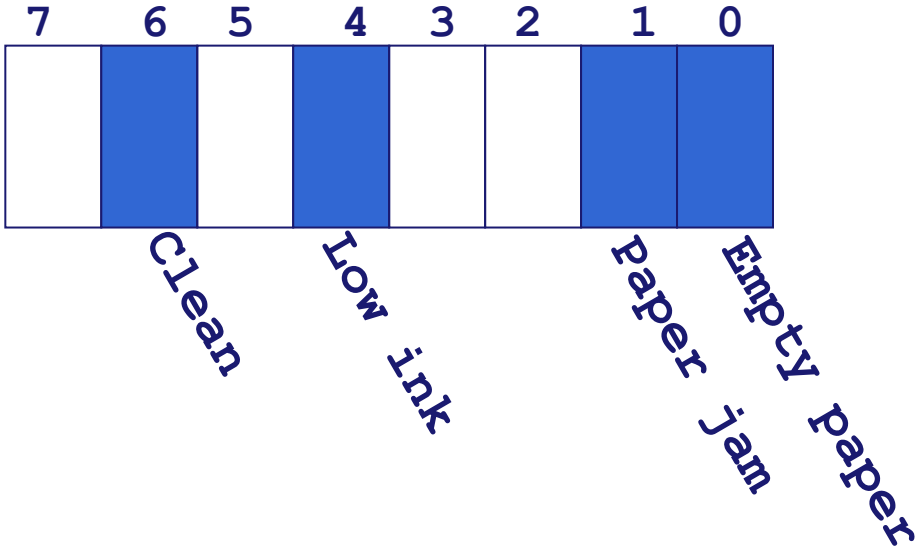
Bit manipulations - Two Approaches

There are two ways of performing bit manipulations in C.

- Traditional *C*
  - Use **#define** and a lot of bitwise operations
- Modern
  - Use ***bit fields***

Printer Status Register…

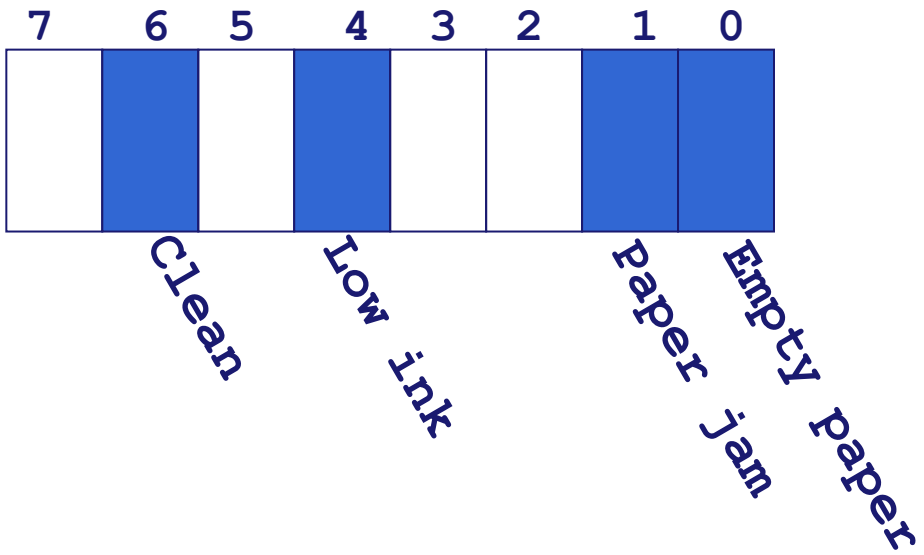
Example: Printer Status register




```
/* Write the C instructions to check the printer status */
char status;
if (status == (EMPTY | JAM)) ...;
if (status == EMPTY || status == JAM) ...;
while (! status & LOW_INK) ...;

int flags |= CLEAN  /* turns on CLEAN bit */
int flags &= ~JAM   /* turns off JAM bit */
```

Using #define – Printer Status Register

Example: Printer Status register



How to set the above bits?

#define EMPTY 01 → 0000 0001 → 01

#define JAM 02 → 0000 0010 → 02

#define LOW_INK 16 → 0001 0000 → 16

#define CLEAN 64 → 0100 0000 → 64

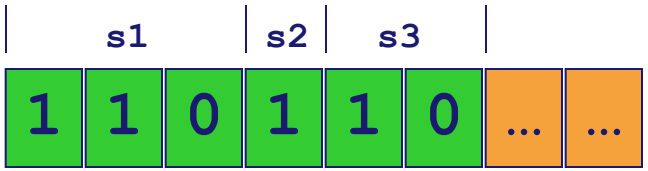
Using #define – Key points

- **Used very widely in *C***
  - Including a *lot* of existing code
- **No checking**
  - You are on your own to be sure the right bits are set
- **Machine dependent**
  - Need to know *bit order* in bytes, *byte order* in words
  - Problems of portability – Little Endian, Big Endian
- **Integer fields within a register – complex to handle**
  - Need to **AND** and shift to extract
  - Need to shift and **OR** to insert

Bit-fields

Bit-fields: A bit field is a collection of one or more bits which are declared in a structure or a union.

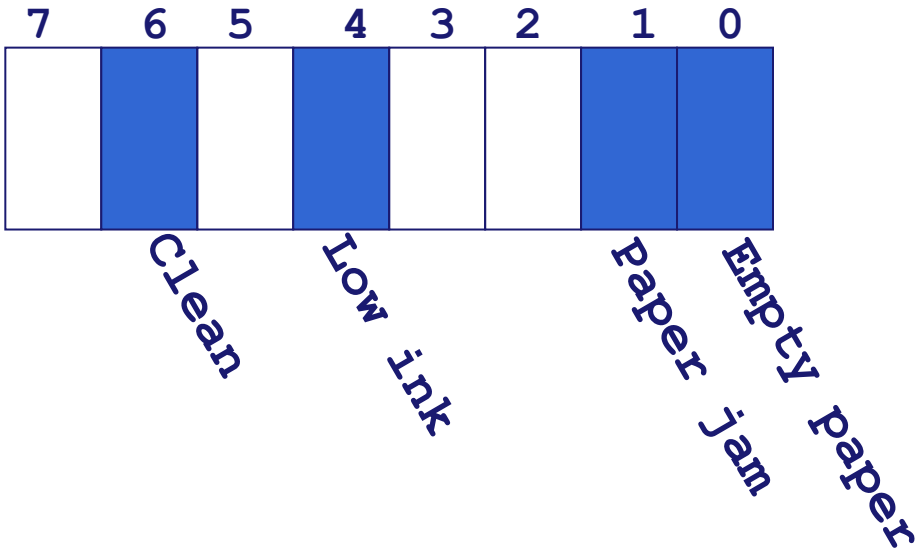
```
struct status {
    int             s1:3;
    unsigned int    s2:1;
    unsigned int    s3:2;
} my_status;

my_status.s1 = -2;
my_status.s2 = 1;
my_status.s3 = 2;
```



Bit-fields – Points to be noted

- Allows you to specify small objects with different lengths
- Data Types Allowed: int, signed int and unsigned int
- Each field can be accessed like an ordinary member of a structure
- Fields can be named or unnamed
- Both signed and unsigned are allowed
- No guarantee of the ordering of fields within machine words
- Compiler dependent & Portability issues may arise
- Packing order and storage boundary are machine dependent

Bit-fields Example – Printer Status Register




```
struct printer_status {
       unsigned int emptyPaperTray :1;
       unsigned int paperJam       :1;
                                   :2; //2 empty bits
       unsigned int lowInk         :1;
                                   :1; //1 empty bit
       unsigned int needsCleaning  :1;
                                   :1;
};
```

(c) Unions

Introduction

Definition:

A union is a special data type available in C that enables you to store different data types in the same memory location.

Syntax:

```
union   [union-tag]

{
        member-definition;
        ..................
        [member-definition];
}u;
```

- *Unions provide a way to manipulate different kinds of data in a single area of storage*
- *A single variable can legitimately hold any type of variable*
- *Members can be accessed using a dot (.) or (->)*

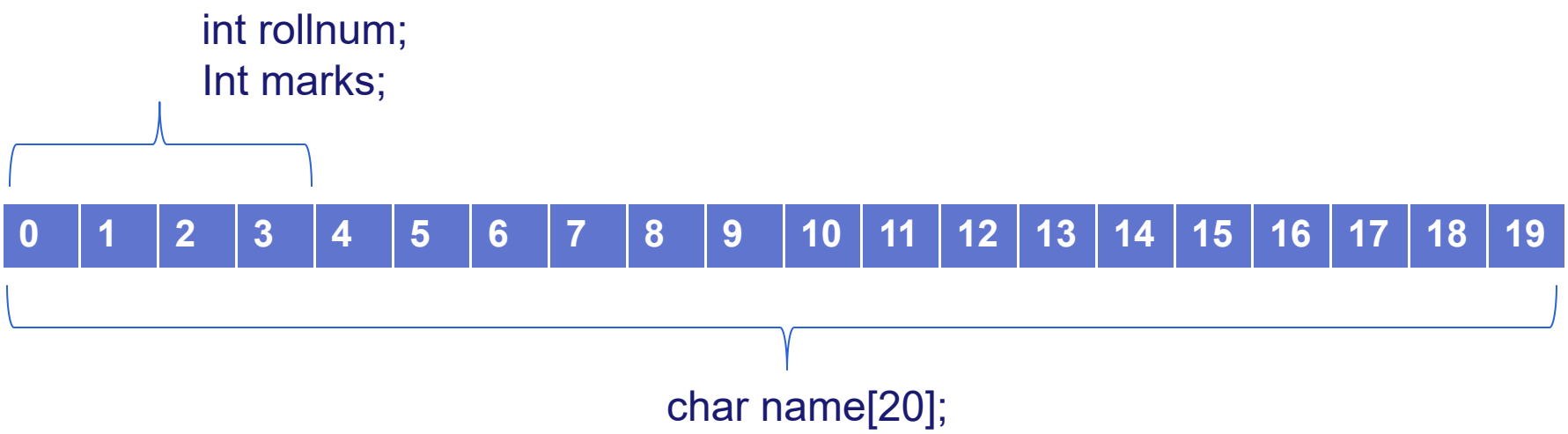
Declaration of Unions

- Syntax is similar to structures
- Only the difference is memory storage
- Unions can handle only one member at a time

```
union student
{
        int rollnum;
        int marks;
        char name[20];
}s1;
```

Memory allocation in Unions

- All the members in a union will share the same memory
- The memory occupied will be large enough to hold the largest member of the union.
- We can use any built-in or user defined data typed variables
- Union can be used in a structure, A structure can also be used in a union



Example

```
#include <stdio.h>
#include <string.h>
union Data
{
   int i;
   float f;
   char  str[20];
};
int main( )
{
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

```
struct {
char *name;
int flags;
int utype;
union {
        int ival;
        float fval;
        char *sval;
} u;
} symtab[NSYM];
```

Thank You