Enumeration and Structures

Enumerated Type

- Enumerated type allows us to create our own symbolic name for a list of related ideas.
- The keyword for an enumerated type is enum.
- We could create an enumerated type to represent various “account types”, by using the following statement:

```
enum act_type {saving, current, fixdeposit, minor };
enum act_type a;
a = minor;
if(a == current)
        printf(“Current account balance is: %f”, curBalance);
```

Example

What is your highest level of education?

- Middle School
- High School
- BSc.
- MSc.
- BE
- BTech
- ME
- MTech
- PhD

Actually what happens!

```
enum act_type {saving, current, fixdeposit, minor };
//saving = 0, current = 1, fixdeposit = 2, minor = 3
enum act_type a;
a = minor;          //a = 3;
if(a == current)    //if(a == 1)
        printf("Current account balance is: %f", curBalance);
```

Structures

- Consider a situation where you need to receive 100 students' roll number, student first name, student last name, branch as input. How will you manage the situation?

```
int rollNumber[100];
char firstName[100][20];
char lastName[100][20];
char branch[100][20];
```

This, however, is not realistic. You can think of a possible solution like a file containing all your details at one place, one student at a time.

What is a structure?

- It is a user defined data type and is a collection of variables under a common name.
- The variables can be of different types including arrays, pointers or structures themselves.
- Structure variables are called fields.

```
//This is called structure blueprint
struct point {
        int x;
        int y;
};
struct point pt;        //This is structure declaration
```

To access the fields of a structure variable, we use the dot operator.

```
pt.x = 0;
pt.y = 0;
```

Initializing structures

```
struct point {
        int x; int y;
};
struct point p = {0, 0};
struct point q = p;             //This is valid unlike arrays or strings
```

Array of structures

```
struct point {
        int x;
        int y;
};
struct point pt1, pt2;
struct point pts[6];            //array of structures
int i;
for(i=0; i<6; i++) {
        pts[i].x = i;
        pts[i].y = i;
}
```

Reading value into structure

```
struct point {
        int x;
        int y;
};
void main() {
        int x, y;
        struct point pt;
        scanf(“%d %d”, &(pt.x), &(pt.y));
}
```

Function with structures as parameters

```
#include<stdio.h>
#include<math.h>
struct point {
        int x; int y;
};
double norm2(struct point p) {
        return sqrt( p.x * p.x + p.y * p.y );
}
void main() {
        struct point pt;
        pt.x = 3;
        pt.y = 4;
        printf("%lf", norm2(pt));
}
```

Functions returning structures

```
struct point {
        int x; int y;          };
struct point make_pt(int x, int y) {
        struct point temp;
        temp.x = x;
        temp.y = y;
        return temp;           }
void main() {
        int x,y;
        struct point pt;
        scanf("%d %d", &x, &y);
        pt = make_pt(x, y);
}
```

Structures inside structures

```
struct point {
        int x; int y;
};
struct rect {
        struct point leftbot;
        struct point righttop;
};
void main() {
        struct rect r;
        r.leftbot.x = 0;
        r.leftbot.y = 0;
        r.righttop.x = 1;
        r. righttop.y = 1;
}
```

Passing structure address

```
struct point {
        int x; int y; };
struct rect {
        struct point leftbot;
        struct point righttop;
};
```

```
int area(struct rect *pr) {
        return ((*pr).righttop.x - (*pr).leftbot.x) *
                ((*pr).righttop.y - (*pr).leftbot.y);
}
void main() {
        struct rect r = {{0,0}, {1,1}};
        area (&r);
}
```

Things to remember

- pr is pointer to struct rect.
- To access a field of the struct pointed to by struct rect, use

  (*pr).leftbot

  (*pr).righttop
- Bracket around *pr i.e. (*pr) is essential as * has lower precedence than “.”

Structure Pointers

```
struct point {
        int x; int y;          };
struct rect {
        struct point leftbot;
        struct point righttop;          };
struct rect *pr;
```

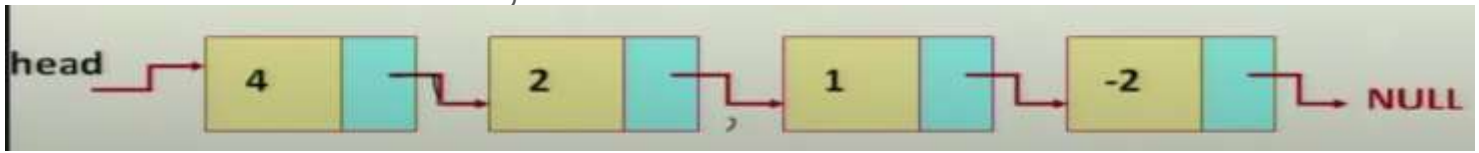
- -> notation is used to access a field of a structure pointed by a pointer.
- pr -> leftbot is equivalent to (*pr).leftbot
- -> and . are left-associative and have same precedence.

Self-referential structure

- Consider the following structure

```
struct node {
        int data;
        struct node * next;
}
```

- Defines struct node, used as a node (element) in a linked list.
- You cannot define struct node inside struct node as it will create a recursive definition of unknown or infinite size.
- This is used to create a linked list, a list of node elements.



Linked Lists

- The list is modelled by a variable (head(: points to the first node of the list.
- head == NULL implies empty list.
- The next field of the last node is NULL.
- Name head is just a convention – can give any name to the pointer to first node, but head is used most often.

Displaying a Linked List

```
void display_list(struct node *head) {
        struct node * cur = head;
        while(cur != NULL) {
                printf("%d        ", cur->data);
                cur = cur -> next;
        }
        printf("\n");
}
```

Create a new node and insert at front

```
struct node * make_node(int val) {
        struct node * nd;
        nd = (struct node *) calloc(1, sizeof(struct node));
        nd->data = val;
        nd->next = NULL;
        return nd;
}
struct node * insert_front(int val, struct node * head) {
        struct node * newnode = make_node(val);
        newnode->next = head;
        head = newnode;
        return head;
}
```

Delete a node from front

```
struct node * delete_front(struct node * head) {
        struct node * tempnode = head;
        head = head -> next;
        printf("Deleted node: %d", tempnode -> data);
        return head;
}
```

Create a new node and insert at end

```
struct node * make_node(int val) {
        struct node * nd;
        nd = (struct node *) calloc(1, sizeof(struct node));
        nd->data = val;
        nd->next = NULL;
        return nd;
}
struct node * insert_end(int val, struct node * head) {
        struct node * ptr = head;
        while(ptr -> next != NULL) {
                ptr = ptr -> next;
        }
        struct node * newnode = make_node(val);
        ptr -> next = newnode;
        return head;
}
```

Delete node at the end

```
struct node * delete_end(struct node * head) {
        struct node * ptr = head, * prev = NULL;
        while(ptr -> next != NULL) {
                prev = ptr;
                ptr = ptr -> next;
        }
        prev -> next = NULL;
        printf("Deleted node: %d", ptr -> data);
        return head;
}
```

Union

- Union is an user defined datatype in C programming language.
- It is a collection of variables of different datatypes in the same memory location.
- We can define a union with many members, but at a given point of time only one member can contain a value.

Why?

- C unions are used to save memory. To better understand an union, think of it as a chunk of memory that is used to store variables of different types. When we want to assign a new value to a field, then the existing data is replaced with new data.
- C unions allow data members which are mutually exclusive to share the same memory. This is quite important when memory is valuable, such as in embedded systems. Unions are mostly used in embedded programming where direct access to the memory is needed.

Union vs Structure

The main difference between structure and a union is that:

- Structs allocate enough space to store all of the fields in the struct. The first one is stored at the beginning of the struct, the second is stored after that, and so on.
- Unions only allocate enough space to store the largest field listed, and all fields are stored at the same space.

Syntax

```
union union_name {
        datatype field_name;
        datatype field_name;
        // more variables
};
```

To access the fields of a union, use dot(.) operator i.e., the variable name followed by dot operator followed by field name.

Example

```
#include<stdio.h>
struct test1 {
        int x, y;    };
union test {
        int x, y;    };
void main() {
        struct test1 t1={1,2};
        union test t;
        t.x = 3; // t.y also gets value 3
        printf ("after fixing x value the coordinates of t will be %d %d\n\n",t.x, t.y);
        t.y = 4; // t.x is also updated to 4
        printf ("After fixing y value the coordinates of t will be %d %d\n\n", t.x, t.y);
        printf("The coordinates of t1 are %d %d",t1.x,t1.y); return 0;
}
```

Union inside Structure

```
#include<stdio.h>
struct student {
        union {
        //anonymous union (unnamed union)
                char name[10];
                int roll;
        };
        int mark;
};
void main() {
   struct student stud;
   char choice;
   printf("\n You can enter your name or roll number
");
   printf("\n Do you want to enter the name (y or n): ");
   scanf("%c",&choice);
   if(choice=='y'||choice=='Y') {
        printf("\n Enter name: ");
        scanf("%s",stud.name);
        printf("\n Name:%s",stud.name);
}
else {
        printf("\n Enter roll number");
        scanf("%d",&stud.roll);
        printf("\n Roll:%d",stud.roll);
}
printf("\n Enter marks");
scanf("%d",&stud.mark);
printf("\n Marks:%d",stud.mark);
}
```

Structure inside Union

```
#include<stdio.h>
void main() {
        struct student {
                char name[30];
                int rollno;
                float percentage;
        };
        union details {
                struct student s1;
        };
        union details set;
        printf("Enter details:");
        printf("\nEnter name : ");
        scanf("%s", set.s1.name);
        printf("\nEnter roll no : ");
        scanf("%d", &set.s1.rollno);
        printf("\nEnter percentage :");
        scanf("%f", &set.s1.percentage);
        printf("\nThe student details are :
\n");
        printf("\name : %s", set.s1.name);
        printf("\nRollno : %d", set.s1.rollno);
        printf("\nPercentage : %f",
                set.s1.percentage);
}
```