Structures

Ritankar Mandal

Structure

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

```
struct complex
{
    float x;
    float y;
};
```

- struct introduces a structure declaration.

- The variables named in a structure are called **members**. The structure member operator "." connects the structure name and the member name.

Structure: Declaration & Initialization

Declaration: `struct complex z1, z2;`

Initialization of members: `z1.x = 1.2; z1.y = 3.2;`

Declaration & Initialization of members:

```
struct complex z2 = {2.2, 2.8};
```

Structure: Declaration & Initialization using a Function

```c
/* make a complex number from x and y components */
struct complex getcomplex(float x, float y)
{
    struct complex temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

There is no conflict between the argument name and the member with the same name e.g, x and y.

Structures & Functions

```c
struct complex complex_add(struct complex z1, struct
    complex z2)
{
    struct complex result;
    result.x = z1.x + z2.x;
    result.y = z1.y + z2.y;
    return temp;
}
```

Do complex_sub, complex_multiplication.

Array of Structures

```c
#include<stdio.h>

struct complex
{
    float x;
    float y;
} complexNumbers[10];

int main()
{
    complexNumbers[1].x = 2.3; complexNumbers[1].y = 3.3;
    printf("complexNumbers[%d] = (%f, %f) \n", 1,
        complexNumbers[1].x, complexNumbers[1].y);
    return 0;
}
```

Size of Structures

```c
#include<stdio.h>
struct collection{
           int p;
           float q;
           char r;
};
int main(){
    int a;
    float b;
    char c;
    struct collection d;

    printf("Size of of a: %u\n", sizeof(a));
    printf("Size of of b: %u\n", sizeof(b));
    printf("Size of of c: %u\n", sizeof(c));
    printf("Size of of d: %u\n", sizeof(d));
    return 0;
}
```

Size of Structures

The `sizeof` operator for a `struct` is not always equal to the sum of `sizeof` of each individual member. When applied to a structur, the result is the number of bytes in the object, including any required padding.

Pointer to Structures

Structure pointers are just like pointers to ordinary variables.

```c
struct complex z, *pz;

z.x = 1.2; z.y = 3.2;
pz = &z;

printf("The number is (%f, %f) \n", (*pz).x, (*pz).y);
printf("The number is (%f, %f) \n", pz->x, pz->y);
```

The parentheses are necessary in (*pz).x because the precedence of the structure member operator . is higher then *. The expression *pz.x means *(pz.x), which is illegal here because x is not a pointer.

Typedef

typedef is used for creating new data type names.

```
typedef int Length;
```

makes the name Length a synonym for int.

Length can be used in declarations, casts, etc., in exactly the same ways that the int type can be:

```
Length len, maxlen;
Length *lengths[];
```

Typedef

```
typedef char *String;
```

makes String a synonym for `char *` or character pointer, which may then be used in declarations and casts.

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

The type being declared in a `typedef` appears in the position of a variable name, not right after the word typedef.

Typedef of Structures

```c
#include<stdio.h>

struct complex
{
    float x;
    float y;
};

typedef struct complex Comp;

int main()
{
    Comp z1, z2;
    z1.x = 1.2; z1.y = 3.2;
    printf("z1.x = %f, z1.y = %f\n", z1.x, z1.y);

    return 0;
}
```

Typedef of Structures

```c
#include<stdio.h>

typedef struct complex
{
    float x;
    float y;
} Comp;

int main()
{
    Comp z1, z2;
    z1.x = 1.2; z1.y = 3.2;
    printf("z1.x = %f, z1.y = %f\n", z1.x, z1.y);

    return 0;
}
```

Self-referential Structures

```
struct node
{
    float val;
    struct node *next;
};
```

It is illegal for a structure to contain an instance of itself. But

```
struct node *next;
```

declares next to be a pointer to a node, not a node itself.

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

typedef struct node
{
    float val;
    struct node *next;
} Node;

int main()
{
    Node *pnode1;
    /* 1. allocate memory */
    pnode1 = (Node *)malloc(sizeof(Node));
    /* 2. assign data */
    pnode1->val = 1;
    pnode1->next = NULL;

    printf("%f \n", pnode1->val);
    return 0;
}
```

```c
int main()
{
    Node *pnode1;
    /* 1. allocate memory */
    pnode1 = (Node *)malloc(sizeof(Node));
    /* 2. assign data */
    pnode1->val = 1;
    /* 3. allocate memory */
    pnode1->next = (Node *)malloc(sizeof(Node));
    /* 4. assign data */
    pnode1->next->val = 2;
    /* 5. Make the final pointer NULL */
    pnode1->next->next = NULL;

    printf("%f, %f \n", pnode1->val, pnode1->next->val);
    return 0;
}
```

```c
int main()
{
    Node *pnode1, *pnode2;
    /* 1. allocate memory and assign data */
    pnode1 = (Node *)malloc(sizeof(Node));
    pnode1->val = 1;
    /* 2. allocate memory and assign data */
    pnode2 = (Node *)malloc(sizeof(Node));
    pnode2->val = 2;
    /* 3. make the 1st node point to the 2nd node and 2nd
        node points to NULL */
    pnode1->next = pnode2;
    pnode2->next = NULL;

    printf("%f, %f, %f \n", pnode1->val, pnode2->val,
        pnode1->next->val);

    return 0;
}
```