Windows Programming

Lecture 04

User-defined
or
Custom Data Types

- Structures
- Unions
- typedefs
- Enumerations

structure

A structure is a collection of variables under a single name. A structure is a convenient way of grouping several pieces of related information together. These variables can be of different data types, and each has a name which is used to select it from the structure.

Pointer to Structures

```
struct Person
{
  char name[30];      //30 bytes
  int age;            //4 bytes
  float length;       //4 bytes
};

struct Person abc, *ptr;
ptr = &abc;
ptr = ptr + 1;        // skips 38 bytes
```


char name[30]
int age
float height
30 bytes
4 bytes
4 bytes

How to Access Members of a Structures

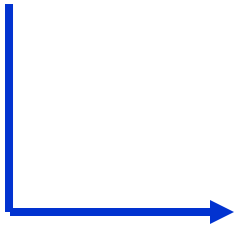
Member of a structure is accessed using member access operator (.) which selects a member from a structure.

How to Access Members of a Structure…

```
abc.name;        // accesses first member
                    of struct Person
abc.age;         // accesses second member
                    of struct Person
abc.length;      // accesses third member
                    of struct Person
```

Member Access Operator

```
abc.age;
```



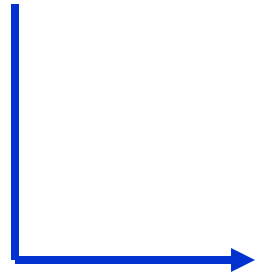
member access operator

Indirect Member Access (Arrow) Operator

Members of a structure can also be accessed using pointer to structures with (->) operator.

Indirect Member Access (Arrow) Operator

```
struct Person *ptr;
ptr -> length;
```



Indirect Member Access Operator

Member Access Operators

When we have a pointer to a structure, we can dereference the pointer and then use dot as a member access operator.

```
ptr -> age
```

is equivalent to

```
(* ptr).age
```

Nested Structures

```
Struct Person
{
  char name[30];          //30 bytes
  int age;                //4 bytes
  float length;           //4 bytes
};
struct Student
{
  struct Person p;        //38 bytes
  int rollno;             //4 bytes
};
```

Nested Structures

```
struct Student st, *ptr;
st . p . height = 5.42;

ptr = &st;
ptr -> p .height;
```

Must use '.' operator

Un-named Nested Structures

```
struct Person
{
   char name[30];          //30 bytes
   int age;                //4 bytes
   float length;           //4 bytes
};

struct Student
{
    struct
    {
      char name[30];
      int age;
      float length;
    }p;
    int rollno;            //4 bytes
};
```

Union

Unions allows a same portion of memory to be accessed as different data types. Its declaration and use is similar to that of structures but its functionality is totally different.

```
union model_name
{
    type1 element1;
    type2 element2;
    type3 element3;
    ...
} object_name;
```

All the elements of the *union* declaration occupy the same space of memory. Its size is equal to the greatest element of the declaration.

Union

Example:

```
union mytypes_t
{
        char c;
        int i;
        float f;
}mytypes;
```

This union defines three elements:

```
mytypes.c
mytypes.i
mytypes.f
```

Each one is of a different data type. All of them are referring to the same location in memory. The modification of one of the elements will affect the value of all of them.

Union

```
union Person
{
  char name[30];          //30 bytes
  int age;
  float length;
};

union Person abc, * ptr;
ptr = &abc;
ptr = ptr + 1;          //skips 30 bytes
```

Little Endian and Big Endian

In Little Endian format, least significant byte is stored on lower memory location

In Big Endian format, most significant byte is stored on lower memory location

Nested Unions

```
union  MyUnion
{
  long a;       // contains 4 bytes
  struct
  {
     short low;
     short high;
  }s;
  char c[4];
};
union MyUnion abc;
abc.a = 1;

abc.s.low = 2;
abc.s.high = 3;
abc.c[0] = 'f';
```

typedef

C allows us to define our own types based on other existing data types. In order to do that keyword **typedef** is used.

```
typedef    existing_type    new_type_name;
```

where *existing_type* is a C fundamental or any other defined type and *new_type_name* is the name that the new type we are going to define will receive.

typedef

Example:

```
typedef char C;
C a[5];          // a is a pointer to
                 // character
typedef unsigned int WORD;
WORD i;          // i is an unsigned int
```

Advantages of `typedef`

- Long chain of keyword in declarations can be shortened.
- Actual definition of the datatype can be changed.

Examples of **`typedef`**

```
typedef  unsigned char byte;
typedef  struct Person  HUMAN;
typedef  char * pCHAR;
```

Enumerations

Enumerations serve to create data types to contain something different that is not limited neither to numerical or character constants.

```
enum model_name { value1, value2, value3,
                  ...};
```

For example, we could create a new type of variable called **color** to store colors with the following declaration:

```
enum colors {black, green, blue, red};
```

Enumerations

```
enum colors {black, green, blue, red};
enum colors myColor;
myColor = black;

if(mycolor == green)
{
      …       …       …
}
```

Enumerations

- **Internally treated as integers**

```
enum months { JANUARY, FEBRUARY, ….};
```

JANUARY stands for 0
FEBRUARY stands for 1 and so on

- **An initial value can be specified for the enumeration constant**

```
enum months { JANUARY=1, FEBRUARY, ….};
```

JANUARY stands for 1
FEBRUARY stands for 2 and so on

Questions

- Nested unions

```
struct in_addr
{
  union
  {
      struct
      {
        u_char s_b1,s_b2,s_b3,s_b4;
      } S_un_b;
      struct
      {
        u_short s_w1,s_w2;
      } S_un_w;
      u_long S_addr;
  } S_un;
};
```

Questions

- `Typedef char *(*FUNC_TYPE)(void)`
- `enum SQUARE {ONE=1, TWO=4, THREE=9};`