Introduction to Scientific and Engineering Computation

(BIL 102E)

LECTURE 11

Special Data Types

The enum Data type

The enumerated (enum) data type can be use to declare named integer constants. It makes the C program more readable and easier to maintain. Syntax:

enum tag_name{item1, item2, …} var1, var2, ...;

Here tag_name is the name of the enumeration, item1, item2, ... are the names to represent integer constants, and var1, var2 ... are the variables of this newly defined type. For example,

enum gender{male, female} gen_student;

Declares an enumarated type called gender, which can have values male (0) or female (1), and a variable named gen_student. Then we can do the following:

gen_student=male; /* The same as gen_student=0; */

printf("%d\n",gen_student); /* Prints 0 on the screen */


```
#include <stdio.h>
enum gender{male,female};
void PrintGender(enum gender g) {
  if(g == male)
          printf("male");
  else if(g == female)
          printf("female");
  else printf("unknown");
}

main() {
  enum gender guser;
  printf("Are you male or female?\n");
  printf("0 for male, 1 for female\n");
  scanf("%d",&guser);
  printf("Your gender : ");
  PrintGender(guser);
  printf("\n");
}
```

We can define the new type in beginning of the file and refer to it later in the program.

Getting an enumerated argument of type gender. The name of the argument is g.

It is possible to use male and female instead of 0 and 1, respectively.

Define a variable called guser to represent the gender of the user. We can use guser wherever we can use an integer.

```
Are you male or female?
0 for male, 1 for female
0
Your gender : male
```


Type definitions

It is possible to define an alias for a given data type using the typedef statement.

For example,

```
typedef  int  INTEGER;
```

defines an alias for the data type int called INTEGER.

Later in the program, INTEGER can be used wherever int is used. For example,

```
INTEGER k, *pk;
```

declares two variables one of which is an integer and the other is a pointer to integers.

There are two usages of typedef:

1. It can provide a shorthand alias for data types that are long and difficult to comprehend.
2. It brings flexibility to the program in the sense that the types of all variables in a certain data type can be changed easily. For example, if we want all variables of type INTEGER to become long integers in the program all we have to do is to change the typedef line in the program.

Example:

```
#include <stdio.h>
#define MAX_ELEMENTS 2
typedef char *string;
typedef string string_array[MAX_ELEMENTS];

main() {
  string str="Hello World!";
  string_array str_arr={"Hello","World"};
  printf("%s\n",str);
  printf("%s\n",str_arr[0]);
  printf("%s\n",str_arr[1]);
}
```

This defines a new type called string, which.is equivalent to a pointer tp characters.

This defines a new type called string_array, which.is equivalent to an array of pointers to characters.

```
Hello World!
Hello
World
```


Structures

The data types we have seen so far were simple in the sense that they represent only a single data unit such as an integer or real number. Sometimes the data is of a more complex type. For example, the data to hold the information about a person may consist of the name, surname, gender and date of birth of that person. In order to give reference to such data by using a single variable, it is possible to define new data types called structures.

A structure can be defined using the following syntax:

struct *struct_tag* **{**
  [component_definition_1]
  [component_definition_ 2]
  …
} ;

Here, definitions for components are in the same form with the definitions for variables:

data_type comp1, comp2, …;

Example

```
struct Complex{
   double Re, Im;
}
```

shows a data type that consists of two components with names *Re* and *Im* both of which are of type double.

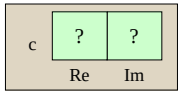
Stucture variables can be defined by using the following syntax:

struct struct_tag var1, var2, …;

For example,

struct Complex c;

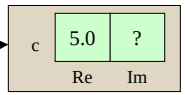
defines a variable named *c* of Complex type.



It is possible to give reference to components of a structure variable by using the dot (.) operator.

For example,

```
c.Re = 5.0;
```



causes the Re component of the variable c to become 5.0.

Components referenced in this way can be used as if they were variables. (In **printf** and **scanf** statements, as arguments to functions etc.)

For example,

```
printf("%d",c.Re);
```

Prints 5.0 on the screen.

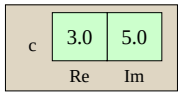
Initializing Structures

A structure can be initialized by a list of data called initializers. The types of corresponding items in component list and initializers should match.

For example,

```
struct Complex c={3.0, 5.0};
```

declares a complex variable c and assigns 3.0 and 5.0 to *Re* and *Im* components, respectively.



Remarks

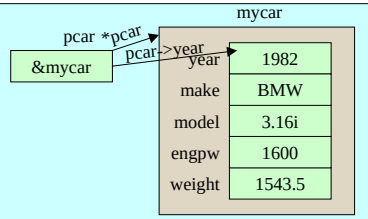
- It is possible to have arrays and other structures as components of structures.
- It is also possible to have array variables of structures.
- It is possible to pass structures as arguments to functions and functions can return structures as their output value.

Pointers to Structures

It is possible to declare pointers that can point to structure variables. *The arrow (->) operator* can be used with the pointer to directly reach the members of the structure. For example,

```
#include <stdio.h>
  struct automobile{
    int year;
    char make[8], model [8];
    int engpw;
    float weight;
  };

main() {
  struct automobile mycar = {1982, "BMW", "3.16i", 1600, 1543.5};
  struct automobile *pcar;
  pcar = &mycar;
  printf("mycar.year \t= %d\n", mycar.year);
  printf("(*pcar).year \t= %d\n",(*pcar).year);
  printf("pcar->year \t= %d\n",pcar->year);
}
```



Reaching the same memory location with 3 different ways.

```
mycar.year      = 1982
(*pcar).year    = 1982
pcar->year      = 1982
```

Note how a member of the structure pointed by the pointer pcar is referenced by -> operator.

Structures as arguments of functions

It is possible to pass an entire structure to a function. In addition, a function can return a structure back to its caller. E.g.

```
typedef struct automobile sauto;
 sauto ChangeMake(sauto acar) {
   sauto temp;
   temp=acar;
   printf("Current make is %s\n", acar.make);
   printf("What is the make of the car? \n");
   gets(temp.make);
   return temp;
 }
main() {
  struct automobile mycar = {1982, "BMW", "3.16i", 1600, 1543.5};
  struct automobile *pcar;
  pcar = &mycar;
  mycar=ChangeMake(mycar);
  printf("mycar.make \t= %s\n", mycar.make);
}
```

Instead of writing *struct automobile* each time we can now shortly use *sauto* type.

This function changes the make of a given car and returns the result as a new car.

```
Current make is BMW
What is the make of the car?
Anadol
mycar.make      = Anadol
```

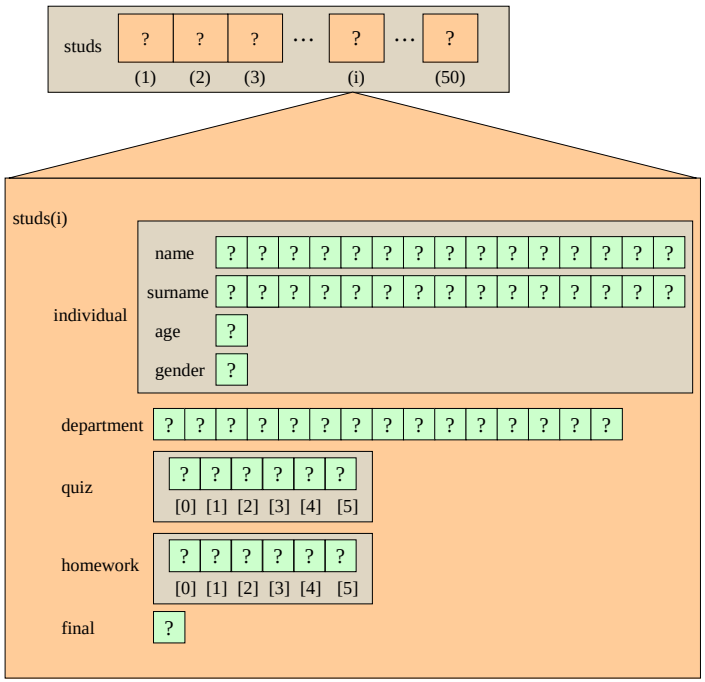

Example

```
#define MAX_CHARS 15
typedef char name_string[MAX_CHARS];

enum gen{male,female};
typedef enum gen type_gender;

struct structperson{
  name_string name, surname;
  int age;
  type_gender gender;
};
typedef struct structperson person;

struct structstudent{
  person individual;
  name_string department;
  int quiz[6];
  int homework[6];
  int final;
};
typedef struct structstudent student;
student studs[50];
```

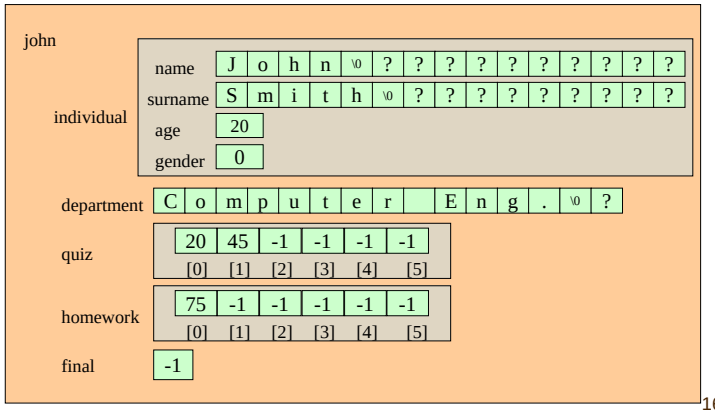




```
main(){
  student john={ {"John","Smith",20,male}, "Computer Eng.",
                {20,45,-1,-1,-1,-1},{75,-1,-1,-1,-1,-1},-1};

  printf("%s\n",john.individual.name);
  printf("%d\n",john.quiz[1]);
}
```

→ John

→ 45



Example

Write a function named AvgQM(studs, num_studs, n) that finds the average mark of n^{th} Quiz for male students in a class. The function should return –1, if there are no male students in the class.

```
float AvgQM(student *studs, int num_studs, int qn)
{
  int i, num_male=0;
  float sum=0.0;
  student *pstud=studs;

  for(i=0;i<num_studs;i++, pstud++)
    if(pstud->individual.gender == male){
       sum+=pstud->quiz[qn];
       num_male++;
    }

  if(num_male == 0)
    return -1;
  else
    return sum/num_male;
}
```