

ECE 220 Computer Systems & Programming

Lecture 17 – Data Structures



The type journey

Objects

struct *

struct []

struct, typedef, enum

int *, char *, float *

int[], char[], float[]

int, char, float

Data Type

Three fundamental data types:

- **integer**
- **float/double**
- **char**

We also discussed:

- **Array**
- **Pointer**

Enumeration Constants:

Enumerated data type:

- An enumeration, introduced by the keyword `enum`, is a set of integer constants represented by identifiers.
- Values in an enum start with 0, unless specified otherwise, and are incremented by 1.

Syntax: `enum [tag] { enumerator-list }`

Example:

```
enum Months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

```
enum Months cur_month;
```

```
cur_month = MAR; //Here JAN equals 0, FEB equals 1, and so on..
```

```
//what is the value of cur_month?
```

```
//what if we define it this way?
```

```
enum Months {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

Another Example (enum):

```
1  #include <stdio.h>
2
3  enum months { JAN = 7, FEB, MAR, APR, MAY, JUN,
4  |             JUL, AUG, SEP, OCT, NOV, DEC };
5
6  int main()
7  {
8      enum months month;
9      const char *monthName[] = { "", "January", "February",
10 |                                "March", "April", "May",
11 |                                "June", "July", "August",
12 |                                "September", "October",
13 |                                "November", "December" };
14
15      for ( month = JAN; month <= DEC; month++ )
16          printf( "%2d%11s\n", month, monthName[ month ] );
17
18      return 0;
19  }
```

Structures

- allow user to define a new type consists of a combination of fundamental data types (**aggregate data type**)
- Example: a repository of students and their grades in this class
 - Name, can be captured as an array of chars (string):
`char name[100];`
 - Student ID, can be stored as an int: `int ID;`
 - Grade for the class, can be stored as a float: `float GPA;`
 - There may be many other characteristics that we would want to capture..

How do we capture them?

Structure – why we need it?

- If we only have one student, we can declare one variable per property:
 - `char name[100];`
 - `int ID;`
 - `float GPA;`
- If we have many (N) students, we can allocate arrays:
 - `char name[N][100];`
 - `int ID[N];`
 - `float GPA[N];`
- to access information about a particular student, we would need to access data in all three arrays: `name[i]`, `ID[i]`, `GPA[i]`
 - if there are only a few properties that we care about, this solution (using separate arrays) may be acceptable
- but if we have many properties, the solution with arrays becomes cumbersome
 - think about passing a large number of arguments to a function
- **a better solution is to aggregate all the properties into a single object**

Structures

- struct construct allows to create a new data type consisting of several member elements (**aggregate data type**)

Example: student record

```
struct StudentStruct  
{
```

```
    char Name[20];
```

```
    int UIN;
```

```
    float GPA;
```

```
}; //In this example, we have created a new data type and gave it the tag StudentStruct;
```

To declare a variable of this type, we can use the new data type's name:

```
struct StudentStruct student;
```

```
strncpy(student.Name, "John Doe", sizeof(student.Name));
```

```
student.UIN = 123456789;
```

```
student.GPA = 3.89;
```

```
//student.name ="John Doe"; //Compiler Error
```

```
//or we can just use one line
```

```
struct StudentStruct student = {"John Doe", 123456789, 3.89};
```


Structures (run-time stack)

- struct construct allows to create a new data type consisting of several member elements (**aggregate data type**)

Example: student record

```
struct StudentStruct
```

```
{
```

```
    char Name[20];
```

```
    int UIN;
```

```
    float GPA;
```

```
};
```

```
struct StudentStruct student;
```

```
student.UIN = 123456789;
```

```
student.GPA = 3.89;
```

```
strncpy(student.Name, "John Doe", sizeof(student.Name));
```

student.name [0]
student.name [1]
student.name [2]
student.name [3]
.....
.....
student.name [18]
student.name [19]
student.UIN
student.GPA

Using typedef

- C allows to give names to user-defined data types using typedef keyword. Thus, we can give an alternative (shorter) name to “struct tag”:
 - `typedef struct tag myType;`
`myType <varName>;`
 - here old name “struct tag” will be given a new name myType.

```
struct StudentStruct
{
    char Name[100];
    int UIN;
    float GPA;
};
typedef struct StudentStruct student;
student s1, s2;
```

Using typedef (both approaches are same)

```
struct StudentStruct
{
    char Name[100];
    int UIN;
    float GPA;
};
typedef struct StudentStruct student;
student s1, s2;

/*****/

typedef struct StudentStruct
{
    char Name[100];
    int UIN;
    float GPA;
}student;
student s1, s2;
```

```
1  #include <stdio.h>
2  #include <string.h>
3
4  typedef struct studentStruct
5  {
6      char Name[100];
7      int UIN;
8      float GPA;
9  } student;
10
11 void higher_GPA(student s1, student s2)
12 {
13     if(s1.GPA > s2.GPA)
14         printf("Student with higher GPA is: %s\n", s1.Name);
15     else
16         printf("Student with higher GPA is: %s\n", s2.Name);
17 }
18
19
20 int main()
21 {
22     student s1 = {"John Doe", 123456789, 3.89};
23     student s2 = {"Jane Doe", 130000000, 3.98};
24
25     higher_GPA(s1, s2);
26
27     return 0;
28 }
```

Unions

- a **union** data type is similar to a struct, however, it defines a single location in memory that can be given many different names
- Example:

- ```
union valueUnion
{
 long int i_value;
 float f_value;
}
```

```
union valueUnion v;
```

```
v.i_value = 5; /* holds integer */
v.f_value = 5.25f; /* now holds float */
/* but not both! */
```

# Arrays of Structs

```
//create an array of student struct
student ece220[200];
```

```
//access each element of the array
ece220[0]
ece220[1]
```

```
//access individual fields in each element
ece220[0].Name[0] = "J";
ece220[0].Name[1] = "o";
ece220[0].Name[2] = "h";
ece220[0].Name[3] = "n";
ece220[0].UIN = 123456789;
ece220[0].GPA = 3.89;
```

# Pointer to Struct

```
student ece220[200];
student s1;
student *ptr, *ptr2;
ptr = ece220; //pointer to a struct array
ptr2 = &s1; //pointer to a struct
```

```
strncpy(ptr->Name, "John Doe", sizeof(s1.Name));
ptr->UIN = 123456789;
ptr->GPA = 3.89;
//which student record has been changed?
```

```
ptr++; //where is ptr pointing to now?
```

```
//What is the difference between the following function calls?
PrintName(s1);
PrintName(&s1);
```

# Struct within a Struct

```
typedef struct StudentName
{
 char First[30];
 char Middle[30];
 char Last[40];
}name;
```

```
student ece220[200];
student *ptr;
ptr = ece220;
```

```
//How can we set the 'First' name in the first student record?
strncpy(, "John",);
```



name

|            |
|------------|
| First[0]   |
| ...        |
| First[29]  |
| Middle[0]  |
| ...        |
| Middle[29] |
| Last[0]    |
| ...        |
| Last[39]   |
| UIN        |
| GPA        |
| First[0]   |
| ...        |
| First[29]  |
| Middle[0]  |
| ...        |
| Middle[29] |
| ...        |

ece220[0]

ece220[1]