ECE 220 Computer Systems & Programming

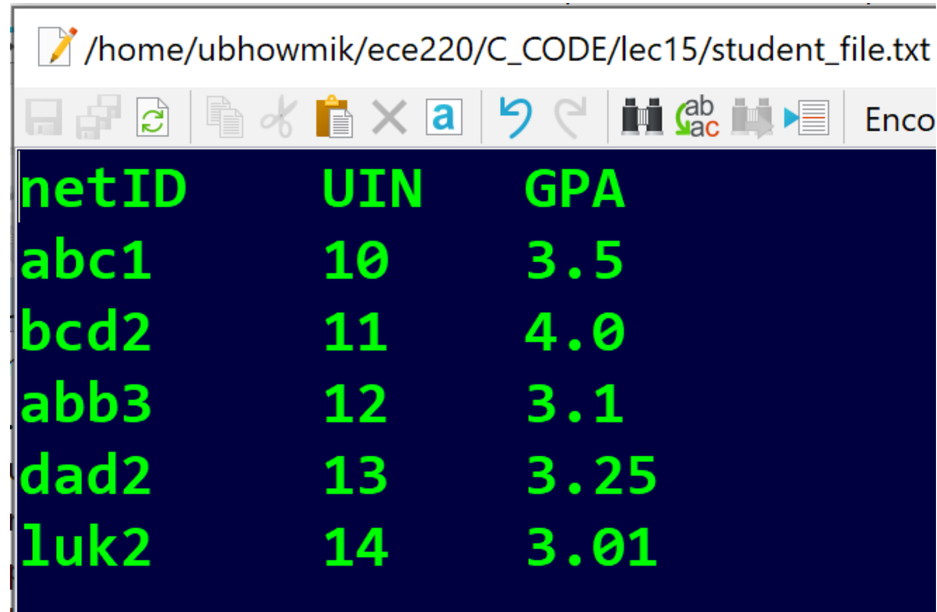
Lecture 15 – Data Structures

ECE ILLINOIS

ILLINOIS

Processing Student Records:

- Given a data file:


/home/ubhowmik/ece220/C_CODE/lec15/student_file.txt

netID	UIN	GPA
abc1	10	3.5
bcd2	11	4.0
abb3	12	3.1
dad2	13	3.25
luk2	14	3.01

- **We want to sort the data according to the GPA ??**
  - The file could have 100's of students' records?

Data Type

Three fundamental data types:

- integer
- float/double
- char

We also discussed:

- Array
- Pointer

The type journey

Objects

...

`struct *`

`struct []`

`struct, typedef, enum`

`int *, char *, float *`

`int[], char[], float[]`

`int, char, float`

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
    - netID, can be captured as an array of chars (string):
      char name[100];
    - Student UIN, can be stored as an int;
    - GPA of the student, 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 netID[100];
  - Int UIN;
  - float GPA;
- If we have many (N) students, we can allocate arrays:
  - **char netID[N][100]; or char *netID[N]; ??**
  - Int UIN[N];
  - float GPA[N];
- to access information about a particular student, we would need to access data in all three arrays: netID[i], UIN[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 netID[10];
    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.netID, "abc1", sizeof(student.netID));
student.UIN = 123456789;
student.GPA = 3.89;
//student.netID ="abc1"; //Compiler Error
//or we can just use one line
struct StudentStruct student = {"abc1", 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 netID[10];
    int UIN;
    float GPA;
};
struct studentStruct student;

student.UIN = 12;
student.GPA = 3.89;
strncpy(student.netID, "abc1", sizeof(student.netID));
```

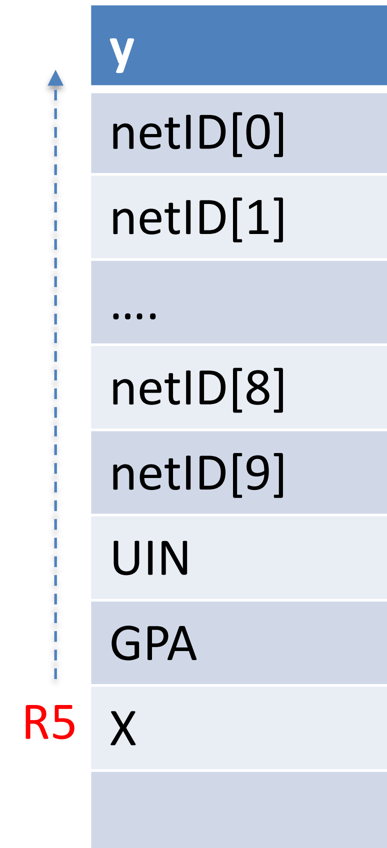
Structures (run-time stack)

```
struct studentStruct
{
    char netID[10];
    int UIN;
    float GPA;
};

int main()
{
int x;
struct studentStruct student;
int y;

student.UIN=0;

}
```



LC3 code of `student.UIN=0;`

```
AND R1, R1, #0;   zero out R1
ADD R0,R5, #-12;  R0 contains the base
                  address of student
STR R1, R0, #10   ; student.UIN=0
```

Using typedef

- C allows to give names to user-defined data types using **typedef** keyword.

- Example:

```
typedef int color;
color  image[10][20];
```

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 netID[100];
    int UIN;
    float GPA;
};

typedef struct studentStruct student;
student s1, s2;
```

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 netID[100];
    int UIN;
    float GPA;
}student;

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;
```

Arrays of structs

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

student s[100];
```

```
//create an array of student struct
student s[100];

//access each element of the array
s[0]
s[1]

//access individual fields in each element
s[0].netID[0] = 'a';
s[0].netID[1] = 'b';
s[0].netID[2] = 'c';
S[0].netID[3] = '1';
s[0].UIN[3] = '11';
s[0].GPA = 3.89;
```

Read the student_file.txt and create an array of structs of student records:

```
int main()
{
student s[BUF];
char filename[20];
int no_of_student;
printf("Enter the Student_record filename: ");
scanf("%s",filename);
no_of_student=load_data(filename, s);
print_data(s, no_of_student);
}
```

Read the student_file.txt and create an array of structs of student records:

/home/ubhowmik/ece220/C_CODE/lec15/student_file.txt

netID	UIN	GPA
abc1	10	3.5
bcd2	11	4.0
abb3	12	3.1
dad2	13	3.25
luk2	14	3.01

```
int main()
{
student s[BUF];
char filename[20];
int no_of_student;
printf("Enter the Student_record filename: ");
scanf("%s",filename);
no_of_student=load_data(filename, s);
print_data(s, no_of_student);
}
```

```
int load_data(char filename[], student s[]){

    FILE *in;
    in=fopen(filename,"r");
    char temp[BUF];
    fgets(temp, BUF, in);
    int n=0;
while((fscanf(in,"%s %d %f",s[n].netID, &s[n].UIN,&s[n].GPA))!=EOF)
        n++;

    return n++;
}
```

Printing the student records:

```
void print_data(student s[],int n){
    int i;
    printf("netID    UIN   GPA\n");
    for (i=0; i<n;i++)
printf("%s     %d    %f\n", s[i].netID, s[i].UIN, s[i].GPA);

}
```

```
netID       UIN     GPA
abc1        10      3.500000
bcd2        11      4.000000
abb3        12      3.100000
dad2        13      3.250000
luk2        14      3.010000
```

Sort student's records based on GPA

```
void sort_GPA(student s[], int n){

    int i;
    int flag=1;

    while(flag){
        flag=0;
    for (i=0; i<(n-1);i++)
    {
        if (s[i].GPA>s[i+1].GPA){
            swap_student(&s[i],&s[i+1]);
            flag=1;
        }
    }
  }
}
```

Swap student's record:

```
void swap_student(student *s1, student *s2){
student temp;
temp=*s1;
*s1=*s2;
*s2=temp;
}
```

```
netID     UIN    GPA
abc1      10     3.500000
bcd2      11     4.000000
abb3      12     3.100000
dad2      13     3.250000
luk2      14     3.010000

 Sorted GPA record
netID     UIN    GPA
luk2      14     3.010000
abb3      12     3.100000
dad2      13     3.250000
abc1      10     3.500000
bcd2      11     4.000000
```

Pointer to Struct

```
typedef struct studentStruct
{
    char Name[100];
    int UIN;
    float GPA;
}student;
```

```
student ece220[200];

student *ptr;
ptr = ece220; //pointer to a struct array
//ptr = &ece220[5];
ptr++; //where is ptr pointing to now?

strncpy(ptr->Name, "John Doe", sizeof(ptr->Name));
ptr->UIN = 123456789;   //(*ptr).UIN
ptr->GPA = 3.89;        //(*ptr).GPA

//which student record has been changed?
```

Struct within a Struct

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

```
typedef struct StudentStruct
{
    name Name;
    int UIN;
    float GPA;
}student;
```

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

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

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};
```

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 = 5.25;  /* now holds float */
    /* but not both! */
    ```
    ```