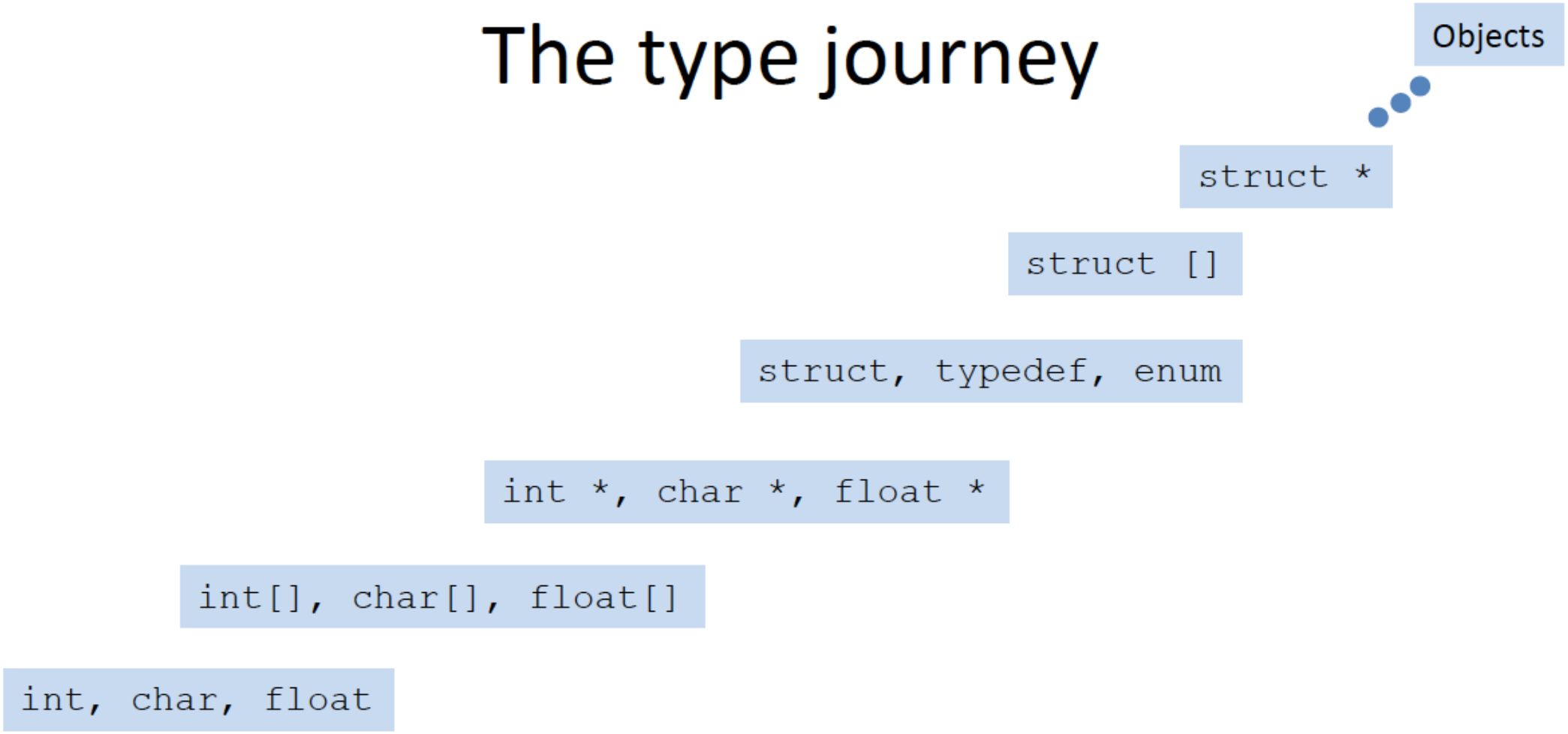
ECE 220 Computer Systems & Programming

Lecture 17 – Data Structures

ECE ILLINOIS

ILLINOIS

The type journey



Data Type

Three fundamental data types:

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

We also discussed:

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

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)

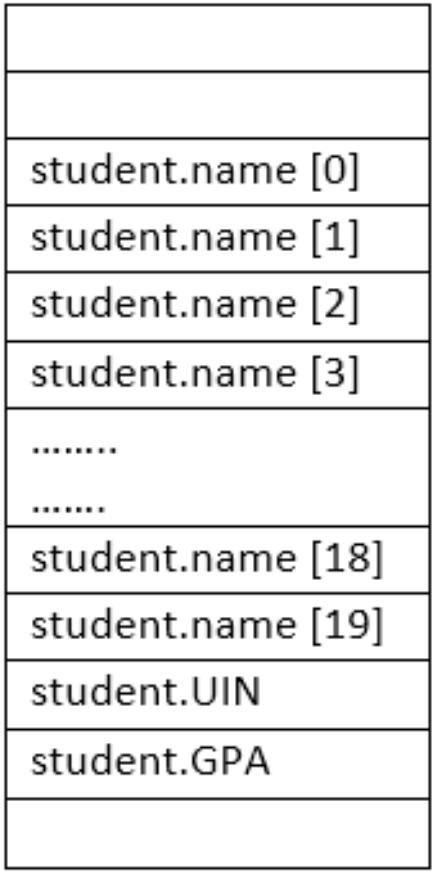
- struct construct allows to create a new data type consisting of several member elements (aggregate data type)

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));
```



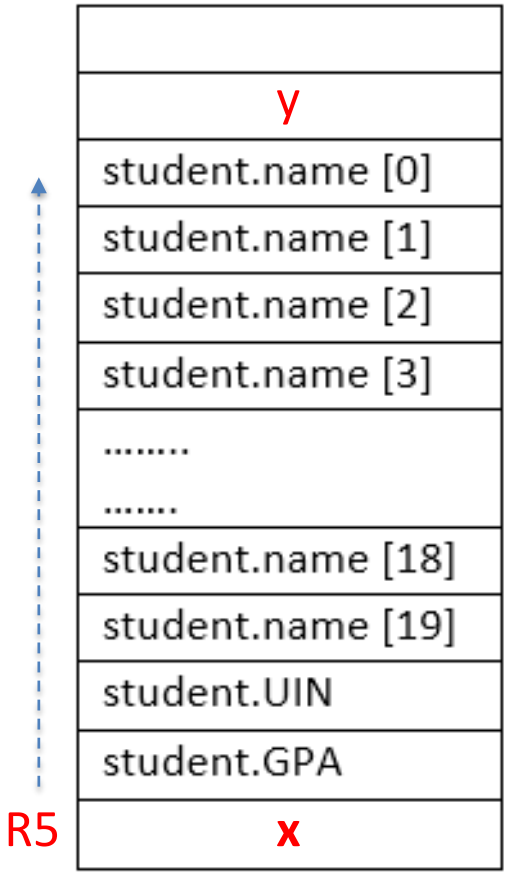
Structures (run-time stack)

```
struct StudentStruct
{
    char Name[20];
    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
AND R0,R5, #-22;  R0 contains the base
                  address of student
STR R1, R0, #20       ; student.UIN=0
```

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;
}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;
```

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

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

void higher_GPA(student s1, student s2)
{
    if(s1.GPA > s2.GPA)
        printf("Student with higher GPA is: %s\n", s1.Name);
    else
        printf("Student with higher GPA is: %s\n", s2.Name);

}

int main()
{
    student s1 = {"John Doe", 123456789, 3.89};
    student s2 = {"Jane Doe", 130000000, 3.98};

    higher_GPA(s1, s2);

    return 0;
}
```

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;
```

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

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(s1.Name));
ptr->UIN = 123456789;   //(*ptr).UIN
ptr->GPA = 3.89;        //(*ptr).GPA

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

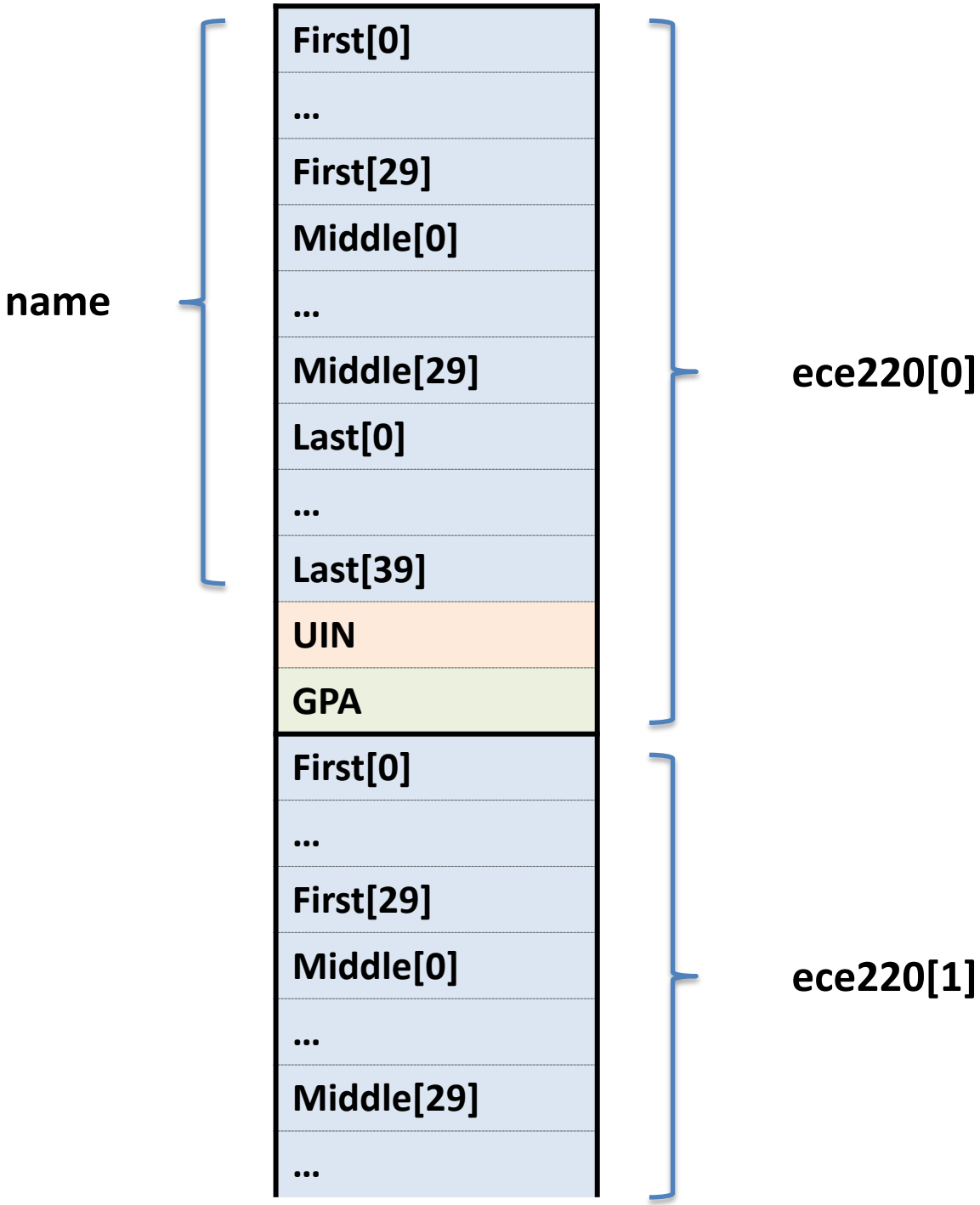
Struct within a Struct

```
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",                          );
```


name
First[0]
…
First[29]
Middle[0]
…
Middle[29]
Last[0]
…
Last[39]
UIN
GPA
ece220[0]
First[0]
…
First[29]
Middle[0]
…
Middle[29]
…
ece220[1]

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):

```c
#include <stdio.h>

enum months { JAN = 7, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };

int main()
{
    enum months month;
    const char *monthName[] = { "", "January", "February",
                                "March", "April", "May",
                                "June", "July", "August",
                                "September", "October",
                                "November", "December" };

    for ( month = JAN; month <= DEC; month++ )
       printf( "%2d%11s\n", month, monthName[ month ] );

    return 0;
}
```

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! */