# Computer Science I

## Encapsulation

Dr. Chris Bourke

cbourke@cse.unl.edu

# Outline

1. Introduction
2. Using Structures
3. Structures with Functions & Arrays

# Part I: Introduction

# Structures

- Built-in primitive types ( `int` , `double` , `char` ) are limiting

# Structures

- Built-in primitive types ( `int` , `double` , `char` ) are limiting
- Not everything is *simply* a number or character

# Structures

- Built-in primitive types (`int`, `double`, `char`) are limiting
- Not everything is *simply* a number or character
- Real-world entities are made up of multiple aspects (data)

# Structures

- Built-in primitive types (`int`, `double`, `char`) are limiting
- Not everything is *simply* a number or character
- Real-world entities are made up of multiple aspects (data)
- Examples: Person, Team, Bank Account, etc.

# Structures

- Built-in primitive types (`int`, `double`, `char`) are limiting
- Not everything is *simply* a number or character
- Real-world entities are made up of multiple aspects (data)
- Examples: Person, Team, Bank Account, etc.
- In code we can define *objects* that *encapsulate* multiple pieces of data

# Encapsulation

## Definition

*Encapsulation* is a mechanism by which multiple pieces of data can be grouped together.

# Encapsulation

## Definition

*Encapsulation* is a mechanism by which multiple pieces of data can be grouped together.

More generally, encapsulation includes:

# Encapsulation

## Definition

*Encapsulation* is a mechanism by which multiple pieces of data can be grouped together.

More generally, encapsulation includes:

- Grouping of data

# Encapsulation

## Definition

*Encapsulation* is a mechanism by which multiple pieces of data can be grouped together.

More generally, encapsulation includes:

- Grouping of data
- Protection of data

# Encapsulation

## Definition

*Encapsulation* is a mechanism by which multiple pieces of data can be grouped together.

More generally, encapsulation includes:

- Grouping of data
- Protection of data
- Grouping of methods that act on an object's data

# Encapsulation

## Definition

*Encapsulation* is a mechanism by which multiple pieces of data can be grouped together.

More generally, encapsulation includes:

- Grouping of data
- Protection of data
- Grouping of methods that act on an object's data

C only provides *weak* encapsulation (only grouping of data).

# Structures

- C provides encapsulation through *structures*

# Structures

- C provides encapsulation through *structures*
- You can define structures that group data called *members* or *fields*

# Structures

- C provides encapsulation through *structures*
- You can define structures that group data called *members* or *fields*
- Demonstration

# Review

- Syntax:

# Review

- Syntax:
  - `typedef struct`

# Review

- Syntax:
  - `typedef struct`
  - Opening/closing curly brackets

# Review

- Syntax:
  - `typedef struct`
  - Opening/closing curly brackets
  - Fields: type, name, semicolon

# Review

- Syntax:
  - `typedef struct`
  - Opening/closing curly brackets
  - Fields: type, name, semicolon
  - Ends with name and semicolon

# Review

- Syntax:
  - `typedef struct`
  - Opening/closing curly brackets
  - Fields: type, name, semicolon
  - Ends with name and semicolon
- Structures may contain other structures; called *composition*

# Review

- Syntax:
  - `typedef struct`
  - Opening/closing curly brackets
  - Fields: type, name, semicolon
  - Ends with name and semicolon
- Structures may contain other structures; called *composition*
- Order of declaration matters

# Review

- Syntax:
  - `typedef struct`
  - Opening/closing curly brackets
  - Fields: type, name, semicolon
  - Ends with name and semicolon
- Structures may contain other structures; called *composition*
- Order of declaration matters
- Structures are declared in header files

# Review

- Syntax:
  - `typedef struct`
  - Opening/closing curly brackets
  - Fields: type, name, semicolon
  - Ends with name and semicolon
- Structures may contain other structures; called *composition*
- Order of declaration matters
- Structures are declared in header files
- Modern convention: `UpperCamelCasing` for structure names, `lowerCamelCasing` for fields

# Part II: Using Structures

# Using Structures

- Once declared, structures can be used like normal variables

# Using Structures

- Once declared, structures can be used like normal variables
- Declaration:

```
Student s;
```

# Using Structures

- Once declared, structures can be used like normal variables
- Declaration:

```
Student s;
```

- To access members, you can use the *dot operator*

# Using Structures

- Once declared, structures can be used like normal variables
- Declaration:

```
Student s;
```

- To access members, you can use the *dot operator*
- `s.nuid = 1234;`

# Using Structures

- Once declared, structures can be used like normal variables
- Declaration:

  ```
  Student s;
  ```

- To access members, you can use the *dot operator*

-
  ```
  s.nuid = 1234;
  ```
- Demonstration

# Using Structures

```c
Student s;
s.nuid = 12345678;
s.firstName = (char *) malloc(sizeof(char) * 10);
strcpy(s.firstName, "Katherine");
s.lastName = (char *) malloc(sizeof(char) * 8);
strcpy(s.lastName, "Johnson");
s.gpa = 3.9;
s.dateOfBirth.year = 1918;
s.dateOfBirth.month = 9;
s.dateOfBirth.day = 26;
```

# Factory Functions

- Creating instances of structures is a common task

# Factory Functions

- Creating instances of structures is a common task
- Best to create a function to facilitate the details

# Factory Functions

- Creating instances of structures is a common task
- Best to create a function to facilitate the details
- Sometimes called *factory* functions (or *constructors* in Object-Oriented Programming languages)

# Factory Functions

- Creating instances of structures is a common task
- Best to create a function to facilitate the details
- Sometimes called *factory* functions (or *constructors* in Object-Oriented Programming languages)
- Dynamically construct an instance using `malloc()`

# Factory Functions

- Creating instances of structures is a common task
- Best to create a function to facilitate the details
- Sometimes called *factory* functions (or *constructors* in Object-Oriented Programming languages)
- Dynamically construct an instance using `malloc()`
- When using pointers to structures, you can use the *arrow operator*: `s->nuid`

# Factory Functions

- Creating instances of structures is a common task
- Best to create a function to facilitate the details
- Sometimes called *factory* functions (or *constructors* in Object-Oriented Programming languages)
- Dynamically construct an instance using `malloc()`
- When using pointers to structures, you can use the *arrow operator*: `s->nuid`
- Demonstration

# Part III: Structures with Functions & Arrays

# Passing Structures to Functions

- Already covered how to return (pointers) to dynamically allocated structures *from* functions

# Passing Structures to Functions

- Already covered how to return (pointers) to dynamically allocated structures *from* functions
- Straightforward to pass structures *to* functions

# Passing Structures to Functions

Introduction
Using Structures
Functions & Arrays
Functions
Arrays

- Already covered how to return (pointers) to dynamically allocated structures *from* functions
- Straightforward to pass structures *to* functions
- Generally want to always want to pass-by-reference

# Passing Structures to Functions

- Already covered how to return (pointers) to dynamically allocated structures *from* functions
- Straightforward to pass structures *to* functions
- Generally want to always want to pass-by-reference
- Passing by value results in a (potentially) large memory copy

# Passing Structures to Functions

- Already covered how to return (pointers) to dynamically allocated structures *from* functions
- Straightforward to pass structures *to* functions
- Generally want to always want to pass-by-reference
- Passing by value results in a (potentially) large memory copy
- Entire structure is copied to the call stack

# Passing Structures to Functions

- Already covered how to return (pointers) to dynamically allocated structures *from* functions
- Straightforward to pass structures *to* functions
- Generally want to always want to pass-by-reference
- Passing by value results in a (potentially) large memory copy
- Entire structure is copied to the call stack
- Pass by reference: only a pointer is copied

# Passing Structures to Functions

- Already covered how to return (pointers) to dynamically allocated structures *from* functions
- Straightforward to pass structures *to* functions
- Generally want to always want to pass-by-reference
- Passing by value results in a (potentially) large memory copy
- Entire structure is copied to the call stack
- Pass by reference: only a pointer is copied
- Demonstration

# Demo

```
void printStudent(const Student *s) {

  char *str = studentToString(s);
  printf("%s\n", str);
  //clean up after yourself:
  free(str);
  //printf("%s, %s (%08d), %.2f\n", s->lastName, s->firstName, s->nuid,
  return;
}

char * studentToString(const Student *s) {
  char buffer[1000];
  sprintf(buffer, "%s, %s (%08d), %.2f", s->lastName, s->firstName, s->
  char *result = (char *) malloc( (strlen(buffer)+1) * sizeof(char));
  strcpy(result, buffer);
  return result;
}
```

# Arrays of Structures

- Multiple structures can be stored in arrays

# Arrays of Structures

- Multiple structures can be stored in arrays
- Several ways to achieve this:

# Arrays of Structures

- Multiple structures can be stored in arrays
- Several ways to achieve this:
  - Array of contiguous structures

# Arrays of Structures

- Multiple structures can be stored in arrays
- Several ways to achieve this:
  - Array of contiguous structures
  - Array of pointers to dynamic structures

# Arrays of Structures

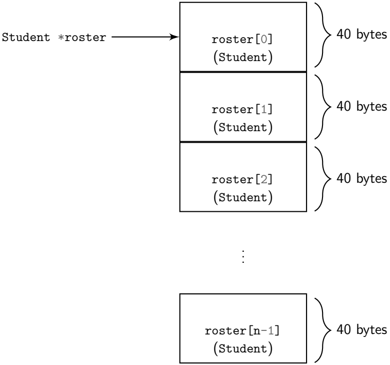
- Multiple structures can be stored in arrays
- Several ways to achieve this:
  - Array of contiguous structures
  - Array of pointers to dynamic structures
  - Array of pointers to contiguous structures

# Arrays of Structures

- Multiple structures can be stored in arrays
- Several ways to achieve this:
    - Array of contiguous structures
    - Array of pointers to dynamic structures
    - Array of pointers to contiguous structures
- Demonstration

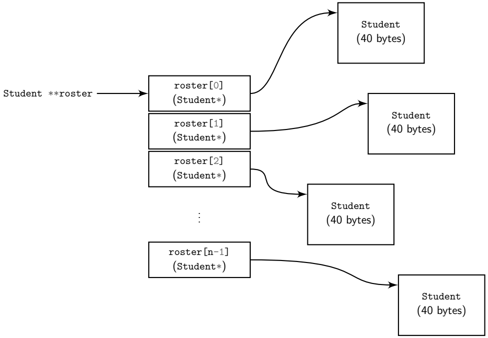
# Arrays of Structures

```
Student *roster = (Student *) malloc(sizeof(Student) * n);
```



# Arrays of Structure Pointers

```c
Student **roster = (Student **) malloc(sizeof(Student*) * n);
...
roster[i] = (Student *) malloc(sizeof(Student));
```



Student **roster

roster[0]
(Student*)

roster[1]
(Student*)

roster[2]
(Student*)

⋮

roster[n-1]
(Student*)

Student
(40 bytes)

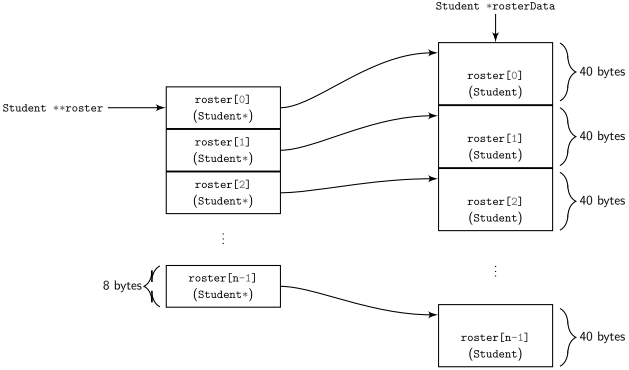
Student
(40 bytes)

Student
(40 bytes)

Student
(40 bytes)

# Hybrid Solution

```
Student **roster = (Student **) malloc(sizeof(Student*) * n);
Student *rosterData = (Student *) malloc(sizeof(Student) * n);
...
roster[i] = &rosterData[i];
```


Student *rosterData
Student **roster
roster[0]
(Student*)
roster[1]
(Student*)
roster[2]
(Student*)
⋮
8 bytes
roster[n-1]
(Student*)
roster[0]
(Student)
40 bytes
roster[1]
(Student)
40 bytes
roster[2]
(Student)
40 bytes
⋮
roster[n-1]
(Student)
40 bytes