Computer Science I

Encapsulation

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

Part I: Introduction

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.

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*
- You can define structures that group data called *members* or *fields*
- Demonstration

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

Using Structures

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

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
- 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

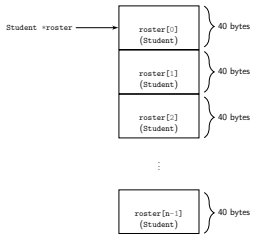
```
1  void printStudent(const Student *s) {
2
3    char *str = studentToString(s);
4    printf("%s\n", str);
5    //clean up after yourself:
6    free(str);
7    //printf("%s, %s (%08d), %.2f\n", s->lastName, s->firstName, s->nuid, s->gp
8    return;
9  }
10
11 char * studentToString(const Student *s) {
12   char buffer[1000];
13   sprintf(buffer, "%s, %s (%08d), %.2f", s->lastName, s->firstName, s->nuid,
14   char *result = (char *) malloc( (strlen(buffer)+1) * sizeof(char));
15   strcpy(result, buffer);
16   return result;
17 }
```

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

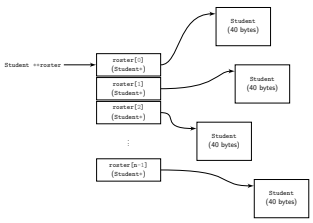
Arrays of Structures

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



Arrays of Structure Pointers

```
1  Student **roster = (Student **) malloc(sizeof(Student*) * n);
2  ...
3  roster[i] = (Student *) malloc(sizeof(Student));
```



Hybrid Solution

```
1  Student **roster = (Student **) malloc(sizeof(Student*) * n);
2  Student *rosterData = (Student *) malloc(sizeof(Student) * n);
3  ...
4  roster[i] = &rosterData[i];
```