Abstract Data Types (ADTs)

Gaddis 11.1 - 11.10

CS 2308 :: Spring 2016
Molly O'Neil
Abstract Data Types

- Abstraction: a general model for something, defining only general characteristics without the details

- **Abstract data type (ADT)**: a model of a data type from the point of view of its user, defined in terms of its
  - possible values that can be stored
  - operations that can be done on the values

- Not the same as a data structure
  - Data structure is a concrete representation of data
  - From point of view of implementer, not the user
Abstract Data Types

• ADTs are a theoretical computer science concept
• Programming languages often provide a means of creating (something like) ADTs
  • User-defined data types
  • That have types of values that can be stored
  • And operations supported on them
• You know one of them in C++ already...
Structures

• C++ construct that allows multiple variables to be grouped together

```cpp
struct Student {
    int id;
    string name;
    double gpa;
};
```

• Struct names generally start with capital letter
  • Signifies a user-defined type, as opposed to a variable
Creating and Manipulating Structs

• Struct declaration does not allocate memory

• To define a variable:

```
Student s1;
```

• Use the dot operator to access members of a struct variable

• Member variables can be used in any manner appropriate to their data type

```
cin >> s1.id;
s1.gpa = 4.0;
```
Arrays of Structures

- You can have an array of structs
- Need [ ] operator to access individual struct (one element of the array)
- Then need dot operator to access member of struct

```cpp
const int NUM_STUDENTS = 10;
Student studentList[NUM_STUDENTS];

for(int i = 0; i < NUM_STUDENTS; i++) {
    cout << studentList[i].name << endl;
}
```

Why not `studentList.name[i]`?
Pointers to Structures

- Just like every other variable, a struct variable has an address
- Pointers to structs are variables that hold the address of a struct variable

```c
Student s1 = { 12345, "John Adams", 3.75 };    
Student *stuPtr;                               
stuPtr = &s1;                                  
```
Accessing Struct Members via Pointers

• Must de-reference pointer to get access to the struct, before applying dot operator:

```cpp
    cout << (*stuPtr).gpa;
```

• The -> operator (structure pointer operator) combines the * (de-reference) and dot (member access) into one operation for clearer notation:

```cpp
    cout << stuPtr->gpa;
```
Dynamic Allocation of Structures

• Just like other variable types, we can dynamically allocate structs on the heap

```cpp
Student *stuPtr = new Student;
stuPtr->id = 98765;
stuPtr->name = "Abigail Adams";
stuPtr->gpa = 4.0;
delete stuPtr;
```

• Arrays of structs can also be dynamically allocated

```cpp
Student *stuList = new Student[100];
stuList[0]->id = 98765;
stuList[0]->name = "Abigail Adams";
stuList[0]->gpa = 4.0;
delete [] stuPtr;
```
Operations on Structures

- Can assign one struct to another
- Can pass entire structs to functions
- Can return entire structs to functions

- Cannot compare structs with comparison operator
- Cannot use arithmetic operators on structs
- Cannot print structs using $\ll$ operator
  - Each of these behaviors must be done member-wise