Computer Science & Engineering 155E
Computer Science I: Systems Engineering Focus

Lecture – Structures

Christopher M. Bourke
cbourke@cse.unl.edu

Introduction

- We've seen built-in simple data types: `int`, `double`, `char`, etc.
- We've also seen user defined simple data types (enumerated types)
- *Simple* data types hold one value at any one time
- *Complex* data types can hold a *collection* of values

Structures

- C allows user-defined complex types through the use of *structures*
- Structures are data types that have *components*
- Components can either be simple data types or other structures

Structure I

Syntax

- Declare a structure using the `typedef struct` syntax:

```
typedef struct {
  int anInt;
  double aDbl;
  int anotherInt;
  char aString[20];
} aStructure;
```

- `aStructure` is the structure's name
- It has 4 simple data type components

Structures

Motivation

- Certain data need to be logically grouped together
- *Records* in a database: data associated with a single person should be kept together
- A *person* may consist of first name, last name, NUID, birth date, etc.
- In turn, a birth date has a month, day, year
- Structures allow us to define such records

Structures

Encapsulation

C Structures provide (partial) support for *encapsulation*.

Definition

Encapsulation is a mechanism by data can be *grouped* together along with the functionality for acting on that data. Encapsulation also provides a means to *protect* data.

- Unfortunately, C structs cannot (easily) encapsulate functions
- Unfrotuantely, C structs cannot protect data
- C structs only provide weak encapsulation (grouping of data)

Structures
Usage

- Declare an instance of a structure like you would an atomic type:
  `aStructure anExampleOfAStructure;`
  `aStructure anArrayOfStructures[10];`
- To set or access the individual member variables we use the following syntax:
  `myStructure.memberVariable`
- Known as *component selection operator*

Structures
Example

```c
typedef struct {
  char month[15];
  int day;
  int year;
} date_t;

typedef struct {
  char firstName[30];
  char lastName[30];
  int  NUID;
  date_t birthDate;
} student;

int main(int argc, char *argv[])
{
  ...

  student aStudent;
  strcpy(aStudent.firstName,"Tom");
  strcpy(aStudent.lastName,"Waits");
  aStudent.NUID = 12345678;
  strcpy(aStudent.birthDate.month, "December");
  aStudent.birthDate.day = 7;
  aStudent.birthDate.year = 1949;

  ...
}
```

Structures & Pointers I

- Pointers to structures can also be used just as with simple data types
- Syntax: `student *myStudent = NULL;`
- You can reference ordinary structures as well:
  `myStudent = &aStudent;`

Structures & Pointers II

- Dynamically allocate memory for structures using `malloc`
- Structure components are fixed at compile time, so C knows "how big" they are; use the `sizeof` function
- Pitfall: remember to cast your pointer!

```c
student *newStudent = NULL;
newStudent=(student *)malloc(sizeof(student));
```

Structures & Pointers I

- If a structure is referenced via a pointer, there is different syntax for accessing its members
- Instead of a period, we use a *pointer*: `->`
- Known as *indirect component selection operator*
- "Equivalent" to `(*newStudent).NUID` (dereference, then component selection)

```c
student *newStudent = NULL;
newStudent=(student *)malloc(sizeof(student));

strcpy(newStudent->lastName, "Jones");
newStudent->NUID = 24681012;
newStudent->birthDate.year = 1940;
```

Structures
As Function Arguments

- It is possible to pass and return structures as function arguments
- Same syntax as simple data types
- You can pass by value or by reference

```c
void printStudentRecordByValue(student s);
void printStudentRecordByRef(student *s);
```

Returning Structures

- As with arrays, we cannot return structures that are *local* in scope
- We must return a *pointer* to a dynamically allocated structure

```
student * readInStudent()
{
  student *newStudent = NULL;
  newStudent=(student *)malloc(sizeof(student));
  printf("Enter First Name>");
  scanf("%s", newStudent->firstName);
  printf("Enter NUID>");
  scanf("%d", &newStudent->NUID);
  return newStudent;

}
```

Common Errors I

- Careful: structures must be declared before they are referenced or used
- Usually declared between preprocessor directives and function prototypes
- The size of a structure is not bounded: you can include as many components as you want

Common Errors II

- Passing structures *by value* is generally a bad idea
- The entire structure is copied to the system stack on a by-value function call: very inefficient
- Pass structures by reference whenever possible
- Know when to use the `.memberVariable` syntax and when to use the `->memberVariable` syntax

Exercise 1

Album Structure

Design a structure for album data. Include components for album title, artist, track titles, number of tracks, year, and any other relevant data.

Exercise 2

Complex Numbers

- A complex number consists of two real numbers: a real component and an imaginary component
- Complex numbers are able to handle roots of negative numbers: Examples:

$$\sqrt{-4} = 0 + 2i$$

$$\sqrt[4]{-4} = 1 + i$$

- Define a structure to handle complex numbers. Write a function to print them in a nice format. Also write a function to compute the multiplication of two complex numbers, define as:

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i$$

- Rewrite the `quadraticRoots` function to compute the roots of a quadratic equation as `complex` types

Exercise 3

Do Programming Project 1 in Chapter 11 (p601)