Data Structures in C

Prof. Georg Feil

C Structures

Summer 2018

Acknowledgement

- These lecture slides are partly based on slides by Professor Simon Hood
- Additional sources are cited separately

Reading Assignment (required)

- ❑ Data Structures (recommended textbook)
  - ▪ Chapter 2, sections 2.1 to 2.5, 2.7
  - ▪ Chapter 3, section 3.6

C Structures

- A structure is a collection of one or more variables, often of different types, grouped together under a single name
  - Structures let you organize related data items
  - Each part (field) of the structure has its own name and data type
- Java comparison: A C structure is similar to a Java class without any methods (just fields)
- Example: Imagine you want to store information about a student – the student's name, ID number, and GPA

  → You can make a structure named 'student' with 3 fields …

Structure declaration

```
struct student {
    char name[50];
    int id;
    double GPA;
};
```

- Shown above is how you might declare the student structure in C using the struct keyword
  - student is the structure name or “tag”
  - It has three fields
  - Don’t forget the semicolon after the }

Structures as a data type

- A structure is used as a data type for variables (like a Java class)
  - Until you use a structure to declare a variable, no runtime memory has been “used”
- A variable whose data type is ‘student’ represents one student

```
struct student var;    // One student
```

- Many students: To store information about a whole class you could create an array of students

```
struct student var[30];    // 30 students
```

Accessing structure fields

- If we declare a variable of the student structure

```
struct student var;
```

then individual fields are accessed using the dot operator

```
printf("%s", var.name);
printf("%d", var.id);
printf("%f", var.GPA);
```

- We can use the fields just like regular variables, e.g.

```
scanf("%s", var.name);
var.id = 16094126;
var.GPA = var.GPA + 0.5;
```

Exercise 1: A simple structure

- Create a C program that declares the student structure
  - Put the declaration before the main function
- In your main function
  - Create a variable of the student structure type
  - Input values for all three fields from the user
  - Print the values of all three fields

Typedef

- The typedef keyword lets you create a data type that's an alias (or synonym) for an existing data type
  - You can choose almost any name for the "new" data type
- Examples:

```
typedef int integer;
typedef unsigned int uint;
typedef unsigned long int size_t;
```

- This is useful to define shorter, more meaningful names for data types
  - After the above typedef declarations you can do: `uint num;`
  - Typedef can cause confusion in C programs, so be careful (use only when it makes your program easier to understand, more portable, etc.)

Exercise 2: Simple typedef

- Create a C program that declares typedefs for these unsigned integer types:

  unsigned int
  unsigned short int
  unsigned char

- Put the typedef declarations before the main function
- In the main function
  - Create a variable of each new data type
  - Assign values to the variables and print them out
  - Experiment to find the biggest and smallest numbers the variables can hold

Typedef and structures

- Notice that to use the ‘student’ structure we had to put the keyword ‘struct’ in front of the structure name

```
struct student var;
```

- We can use typedef when declaring structs to make this shorter

```
typedef struct student {
    char name[50];
    int id;
    double GPA;
} Student;

Student var;   // One student
```

Typedef and structures… variations

- We can leave out the structure tag (name) when using typedef to declare structures

```
typedef struct {
    char name[50];
    int id;
    double GPA;
} Student;
```

- This is the typical way to declare a structure in C
- We can also have structs that contain other structs (nested)
  - For example we could create a Date struct with year/month/day fields, then use Date to store the date of birth for each student

Exercise 3: Typedef with structures

- Convert your program from Exercise 1 to use typedef when declaring the structure

- Change the student variable declaration(s) so they don’t use the struct keyword

Exercise 4: Nested Structures

- Update your program from Exercise 3 to include the following structure definitions (save a copy of the original ex. 3 program)

```
typedef struct {
    char first[21];
    char middle;
    char last[21];
} Name;
```

```
typedef struct {
    int day;
    int month;
    int year;
} Date;
```

- Now change the Student struct so it contains a Name field (replacing the string) and a Date field for date of birth
  - You'll have to use two dots to access fields of a nested structure

Passing structure variables to functions

- A variable whose data type is a structure can be passed to a function just like any other variable
- C structure variables are passed to functions by value
  - In other words, a copy of the structure is made
  - Changes made by the function don't affect the original structure
- For large structures this can be inefficient because a lot of data needs to be copied
  - Imagine a structure containing a large array
- For this reason you may want to pass a pointer to a large structure (call by reference), not the actual structure

Exercise 5: Passing structures

- Update your program from Exercise 3 (no struct within a struct) so that it has a separate function to print a Student structure, declared as follows

```
void printStudent(Student st)
```

- Call the new function from the main function when printing a student

- Done too soon?Do it using both *call by value* and *call by reference* (read ahead on pointers to structures and the -> operator in the following slides)

Arrays of structures

- To work with many students, the simplest data structure we can use is an array
- Each element of the array is a Student structure

```
Student studentList[30];   // Many students
```

- To access the fields of an element the dot operator goes after the square brackets, e.g. to set the 3rd student’s id

```
studentList[2].id = 12345678;
```

- To set the 1st student’s name

```
strcpy(studentList[0].name, "Stuart Studious");
```

Exercise 6: Array of structures

- Update your program from Exercise 5 so that it creates an array of three students (save a copy of the original ex. 5 program)
- Initialize the first element from user input, the rest from hard-coded values
- Use a loop to call printStudent 3 times

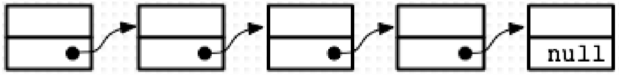
Homework Exercise

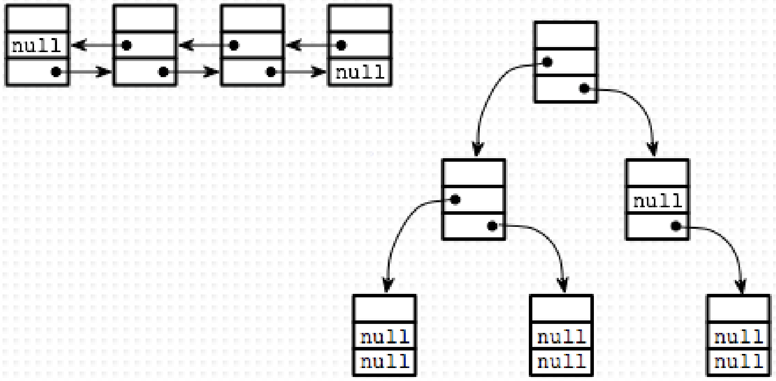
- Do the “Structures Homework Exercise” posted on SLATE in the same folder as these slides

Pointers to Structures

Pointers to structures

- Using pointers with structures facilitates more complex data structures
  - Linked lists, queues, stacks, trees …





Pointers to structures

- We can create pointers to entire structures just as easily as creating a pointer to an int or double
  - Use the usual operators: & to create a pointer to a variable, * to declare or dereference a pointer

- Let’s imagine a structure to represent a player in a game

```
typedef struct {
    char name[31];
    int score;
    int level;
} Player;
```

Pointers to structures

- We can create a Player variable

```
Player player1;
```

- This allocates a player structure in memory (either static data or stack). Then we can create a pointer that points to this variable

```
Player* pPlayer = &player1;
```

- Here the `pPlayer` pointer points to the (start of) the `player1` variable in memory
  - Its data type is “pointer to Player”

Dereferencing pointers to structures

- This is one way to dereference the player pointer

```
(*pPlayer).score = 100;   // Same as player1.score
```

- We need to dereference first, then apply the dot operator
  - The brackets are needed to get the order of operations right

- Here’s how you might print all three player fields

```
printf("Name: %s, score %d, level %d\n",
    (*pPlayer).name, (*pPlayer).score, (*pPlayer).level);
```

Better way: Dereferencing with ->

- Dereferencing pointers to structures is so common in C that a special operator has been defined, the arrow ->
- Here's what the same code looks like with the arrow operator

```
pPlayer->score = 100;
```

- Printing all three player fields

```
printf("Name: %s, score %d, level %d\n",
    pPlayer->name, pPlayer->score, pPlayer->level);
```

Dereferencing with ->

- The arrow operator (->) will be very handy when working with more advanced data structures like linked lists, trees etc.

- Remember, this operator lets you access a particular field of the structure a pointer points to

- It’s easy to remember because it actually looks like a pointer!

Exercise 7: Pointers to structures

- Update your program from Exercise 5 so that it has a separate function to input a student structure, declared as follows

  ```
  void inputStudent(Student* pSt)
  ```

- Call the new function from the main function to input values for all fields of the student structure
- Also update the printStudent function to use a pointer parameter, to ensure the structure doesn’t get copied
  - Use const with this parameter since it should never be changed
- Hint: Use the -> operator to access fields using pointers