CSCB09: Software Tools and Systems Programming

Bianca Schroeder
bianca@cs.toronto.edu
IC 460

Announcements …

- Midterm on June 21
  - 2 hour time slot
  - All material up to reading week (PCRS, lectures, assignments)
  - One page double-sided handwritten cheat sheet
- A2 is out!
  - Gets you practice with structs and pointers
- No office hours today
  - Make-up office hours on Monday 11-12.

Finishing up last week's worksheet …

- Write a program that declares 3 strings
  - "Monday" on the stack,
  - "Tuesday" string literal,
  - "Wednesday" on the heap.
- Shorten the strings to the abbreviations of the weekday names, e.g. "Monday" becomes "Mon".
- Add an array, where each element points to one of the strings above.
- Draw the memory model.

Structs

- Build your own data type!
- A struct is a collection of related data items

- E.g. an application managing a database of a department's students could define the following struct with 3 members:

```
struct student {
   char firstName[20];
   char secondName[20];
   int year;
}
```

Working with structs

```
struct student {
    char firstName[20];
    char secondName[20];
    int year;
 };
```

Now we can use `struct student` as a datatype, just like an `int, float`, ….

To make some students and modify their members:

```
struct student my_first_student;
my_first_student.year = 3;
strcpy (my_first_student.firstName, “Dan”);
strcpy (my_first_student.lastName, “Jones”);
```

Pointers to structs

```
struct student student1;
…… // code for initializing student1 here
struct student *p;
p = &student1;
```

How can we access `student1`'s members?

```
student1.year = 3;
(*p).year = 3;
p->year = 3;
```

Structs and malloc

- Struct can be used with malloc like any other datatype:

```
struct student *s;
s = malloc (sizeof (struct student));
s->year = 3;
```

Structs and functions

- Functions can take structs as parameters and access their elements:

```
void PrintStudent (struct student s) {
            printf (“First name: %s\n”, s.firstName);
            printf (“Last name: %s\n”, s.lastName);
            printf (“Year: %s\n”, s.year);
}
```

One more thing to think about …

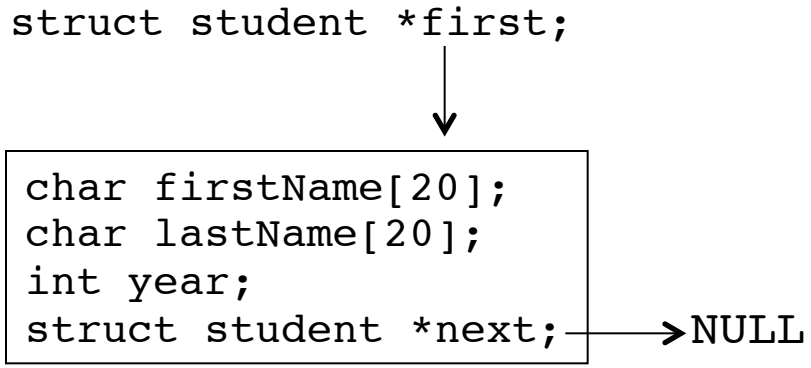
- Going back to the example of the student database:
- We need to manage many students
- The number of students will grow over time
- How do we manage all the different student variables?

- One large array, e.g. `struct student s_arr[1000]` ?
- Disadvantages?
  - Might be too big: wastes space not needed
  - Might be too small: requires frequent realloc to adjust (expensive!)

Idea: Create a linked list

- We only allocate new space when a new student is added to the system, i.e. malloc for one student struct.
- How do we manage all the pointers to the different structs we created. In array? Same disadvantages as before ..

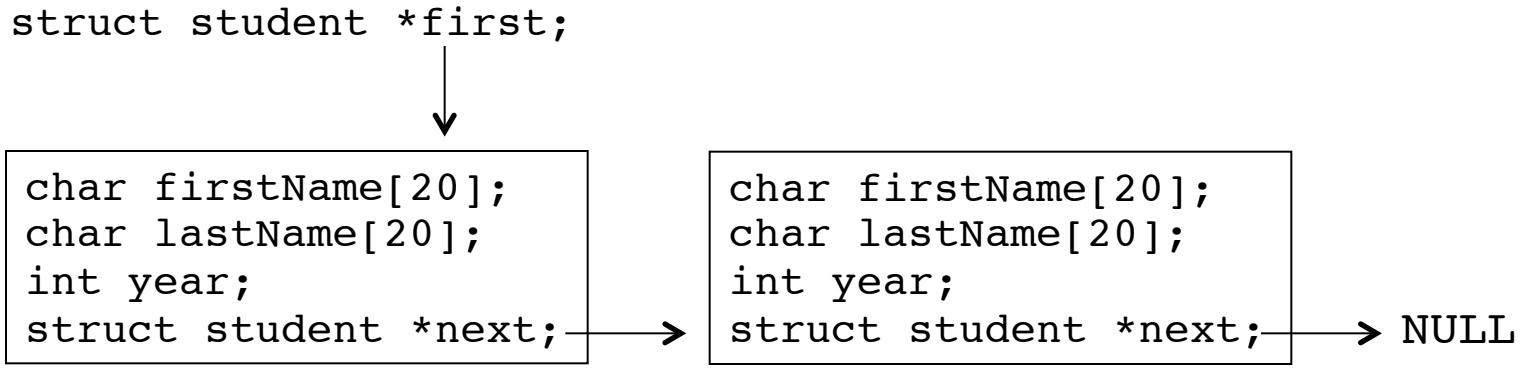
- Instead: in each struct keep a pointer to the following struct!



Idea: Create a linked list

- We only allocate new space when a new student is added to the system, i.e. malloc for one student struct.
- How do we manage all the pointers to the different structs we created. In array? Same disadvantages as before ..

- Instead: in each struct keep a pointer to the following struct!



Think about this and read King, chapter 17.5 (Linked lists)!

Let’s get some practice with structs…