MPCS 51040 – C Programming

Lecture 5 – Quiz & Recursion, Linked List

Dries Kimpe

October 24, 2016

Overview

- Before break: quiz
- After break:
  - Recursion
  - Linked Lists
  - Homework HW3 discussion & HW4-2 assignment

Quiz

Some things to keep in mind:

- 90 minutes
- Read the question carefully...
- Please make it easy for me to grade: write legibly
- You can use the back of the sheets for draft;
  I will ignore the back of the test sheets unless clearly requested not to do so in the answer section of a question
- Check that you have all pages (8) before beginning.
- No phones, backpacks, laptops, ...

Announcements/Reminders

- Extra TA hours this wednesday (see Piazza announcement for exact hours)
- There will be class and/or exercise session on 11/14
- Reminder: feedback regarding homework, lectures, ... always possible&appreciated
- Final project: discussion

Recursion

```
1  void repeat(unsigned int i)
2  {
3      char somelocalvar;
4      if (!i)
5          return;
6      // do something
7      puts("X");
8      repeat(i-1);
9  }
```

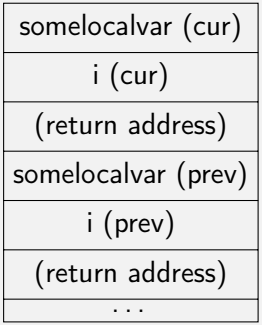
Recursion:

- Recursion happens when a function calls itself.
- Each invocation of the function receives its own copy of local (automatic) variables (variables go on the stack).
- Local variables exists until the function returns.
  - Watch out for memory (stack) usage!
- Recursion is a form of looping.
- Code on left: **tail** recursion

Recursion needs to end: *base case*.

Recursion

Typical implementation (compiler detail)

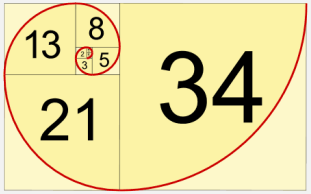


stack

- A stack structure is perfect to keep track of function call data
  - We only need access to the data (local variables, parameters, ...) of the *current* function.
  - When returning, we want to restore the previous situation
  - Multiple concurrent invocations of the function should be possible
- Most processors have built-in support for keeping track of strack structures (`push` and `pop` operation)

Recursion

Example



Calculate fibonacci numbers: $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0, F_1 = 1$

Solution

The base case and recursion step are explicit in the mathematical definition.

Calculate fibonacci numbers (`fibonacci.c`)

Recursion

Example

Recursion is very useful for divide-and-conquer type algorithms: split the problem in smaller problems and try to solve the smaller problem.

Example

Count the number of occurrences of a number in an array.

Solution

- *Base case (conquer): array of size 1*
- *Divide: split in two arrays, add counts.*

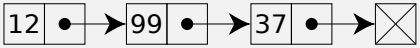
```
unsigned int count (int * array, unsigned int size);
```

Implement count().

What are linked lists

Linked List

A linked list is a data structure (i.e. it stores data). There are multiple kinds (single, double, circular, skiplist, ...) but the principle is the same: a list *node* holds a pointer to another node belonging to the same list.



Typical operations:

- Query the size of the list
- Insert an item at a specific position
- Remove an item
- Search for an item
- ...

The type used to represent the list might or might not be a pointer.

Why linked lists?

Why not use arrays?

Linked lists and arrays serve different purposes and have different strengths and weaknesses. Some examples:

- Arrays are less flexible
  - In order to add elements, you might have to create a new array and copy all existing elements.
    - This would invalidate any pointers to existing array elements!
  - Adding elements in the middle creates similar issues
- Arrays require contiguous memory blocks
- Linked lists are flexible, but generally are slower to access and have higher overhead.
  - Might need to traverse the list to get to the n^{th} element.
  - We need to store link information

Forward Declarations and Incomplete types

Linked Lists in C

```
1  // Struct without tag
2  typedef struct {
3      int a;
4  } MyStruct;
5
6  MyStruct a;
7
8  // Need tag to refer to self
9  struct Link
10 {
11     int data;
12     struct Link * next;
13 };
```

- The struct on line 2 does not have a tag (which is OK)
- In order to link to other structs of the same type, a tag is needed (line 9)

Generic Data Structures

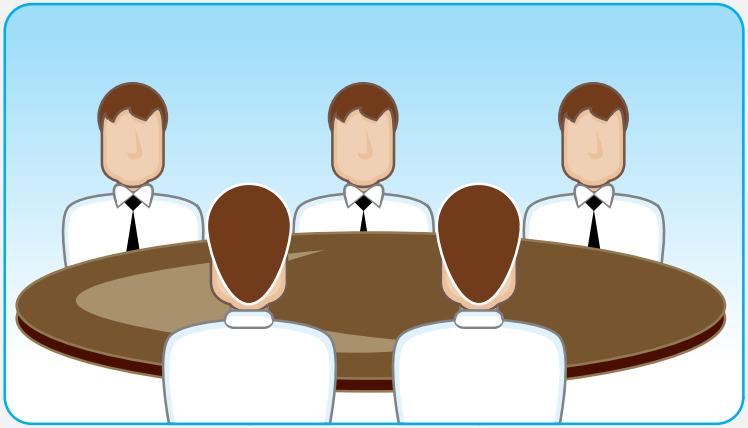
```
1 // Single-linked list storing
2 // void * pointers
3 struct ListItem
4 {
5     void * data;
6     struct ListItem * next;
7 };
```

- Other than `void *`, there is no good way to provide 'generic' data structures.
- Decide (and document) if you are storing a value type or not. Consequences if not storing value types!
  - Destruction, copy, initialization
  - Operations such as testing for equality or partial order.
- By using `void *`, we no longer can rely on the compiler to catch type errors.

Linked list implementation demo

Homework 4

Discussion



What data structures did you use?

Homework 3 - Remarks

- Grades added...
- Do not declare your own prototype for (other people's) library functions!
- Please pay detailed attention to the instructions
  (checking for special characters, return codes, ...)
- Think about how your code looks to other people
  (debug statements, ...)
- Use of `.c` and `.h` files
  - **Make sure you protect your header against multiple inclusion (Questions?)**
  - As little as possible goes into the header; speeds up compilation and helps with encapsulation.
- Strings need space for terminating 0 character!
- Makefile: don't write to `a.out`

Homework 4

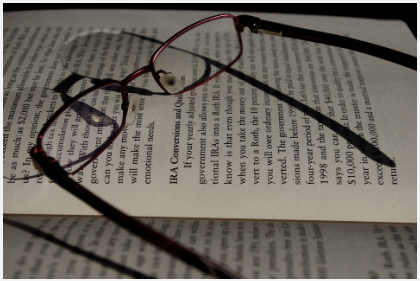
Part 1 - First impressions

HW4

- What I should not see:
  - Broken makefiles
    (for example, spaces instead of tabs)
  - Compiler errors: files which do not compile
    (You can comment out code – still get credit)
  - Warnings
- Very promising:
  - Header guards ☺
- Beware:
  - Will you need to *reallocate*? (`man realloc`)
  - Pick a storage method which 'matches' your algorithms...

Test (and/or develop) on linux.cs.uchicago.edu!!!

Reading Assignment



Reading Assignment

O'Reilly Mastering Algorithms in C:

Required Chapter 6, 7