Warmup #1

Bill Cheng

http://merlot.usc.edu/cs402-f18

Discussion Sections

- ***IMPORTANT:***
  - please understand that discussion section material are ***NOT*** substitute for reading the specs and the grading guidelines
    - you are expect to read the ***specs***
    - you are expect to read the ***requirements*** the specs refer to
    - you are expect to read the ***grading guidelines***
    - it’s your responsibility

Programming & Good Habbits

- *Always* check return code!
  - `open(), write()`
  - `malloc()`
  - `switch (errno) { ... }`
- Initialize *all* variables!
  - `int i=0;`
  - `struct timeval timeout;`
    `memset(&timeout, 0, sizeof(struct timeval));`
- *Never* leak any resources!
  - `malloc() and free()`
  - `open() and close()`
  - Delete temporary files

Programming & Good Habbits

- *Don't* assume external input will be short
  - use `strncpy()` and not `strcpy()`
  - use `snprintf()` and not `sprintf()`
  - use `sizeof()` and not a constant, for example,

```
unsigned char buf[80];

buf[0] = '\0'; /* initialization */
strncpy(buf, argv[1], sizeof(buf));
buf[sizeof(buf)-1] = '\0'; /* in case argv[1] is long */
```

- Fix your code so that you have *zero* compiler warnings!
  - use `-Wall` when you compile to get all compiler warnings

Notes on gdb

- The debugger is your friend! Get to know it ***NOW!***

compile program with:	`-g`
start debugging:	`gdb [-tui] warmup1`
set breakpoint:	`(gdb) break foo.c:123`
run program (w/ arguments):	`(gdb) run [arg1 arg2 ...]`
clear breakpoint:	`(gdb) clear`
stack trace:	`(gdb) where`
print field:	`(gdb) print f.BlockType`
print in hex:	`(gdb) print/x f.BlockType`
single-step at same level:	`(gdb) next`
single-step into a function:	`(gdb) step`
print field after every cmd:	`(gdb) display f.BlockType`
assignment:	`(gdb) set f.BlockType=0`
continue:	`(gdb) cont`
quit:	`(gdb) quit`

- Start using the debugger with warmup 1!
  - get help from TA, course producer, and me

Some General Requirements

- Some major requirements for all programming assignments
  - severe pentalty for failing `make`
    - we will attempt to fix your Makefile you `make` fails
    - if we cannot get it to work, you need to figure out how to fix it by regrade time
  - severe pentalty for using large memory buffers
  - severe pentalty for any segmentation fault -- you must test your code well
  - if input file is large, you must not read the whole file into into a large memory buffer
    - must learn how to read a large file properly
  - severe pentalty for not using separate compilation or for having all your source code in header files -- you must learn to plan how to write your program

Grading Requirements

- For *warmup assignments*, it's important that every byte of your data is read and written correctly

- For *warmup assignments*, you should run your code against the *grading guidelines* on `nunki.usc.edu`
  - must not change the commands there
    - we will change the data for actual grading, but we will stick to the commands (as much as we can)
  - to be fair to all, running scripts in the grading guidelines is *the only way we will grade*

Separate Compilation

- Break up your code into *modules*
  - *compile the modules separately*, at least one rule per module per rule in the `Makefile`
  - a separate rule to *link* all the modules together
    - if your program requites additional libraries, add them to the link stage

- To receive full credit for separate compilation
  - to create an executable, at a minimum, you must run the compiler at least *twice* and the linker *once*

README

- Start with the README templates from the spec
  - *BUILD & RUN* (required)
    - replace "Comments: ?" with how to create your executable (e.g., "make", "gmake", etc.)
  - *SELF-GRADING* (required)
    - grade yourself
      - replace each "(Comments?)" with a *numerical score*
  - *BUGS / TESTS TO SKIP* (required)
    - replace "Comments: ?" with a list of known bugs
      - you can still lose points, but we need to see that you are aware of your bugs or we will deduct additional points
    - you won't get plus points; but you may lose less points
  - *OTHERS* (optional)
    - you can delete this section or write "this section intentionally left blank"
- There should be no "?" in a required section

Code Design - Functional vs. Procedural

- Don’t design your program "procedurally"
- You need to learn how to write functions!
  - a function has a well-defined interface
    - what are the meaning of the parameters
    - what does it suppose to return
  - pre-conditions
    - what must be true when the function is entered
    - you assume that these are true
      - you can verify it if you want
  - post-conditions
    - what must be true when the function returns
  - you design your program by making designing a sequence of function calls

Warmup #1

- 2 parts
  - develop a *doubly-linked circular list* called *My402List*
    - this corresponds to part (A) of the grading guidelines
    - to implement a traditional *linked-list abstraction*
      - internally, the implementation is a *circular list*
      - internally, it behaves like a *traditional list*
      - why? circular list implementation may be a little "cleaner"
  - use your doubly-linked circular list to implement a command:
    - *sort* - sort a list of bank transactions
    - this corresponds to part (B) of the grading guidelines

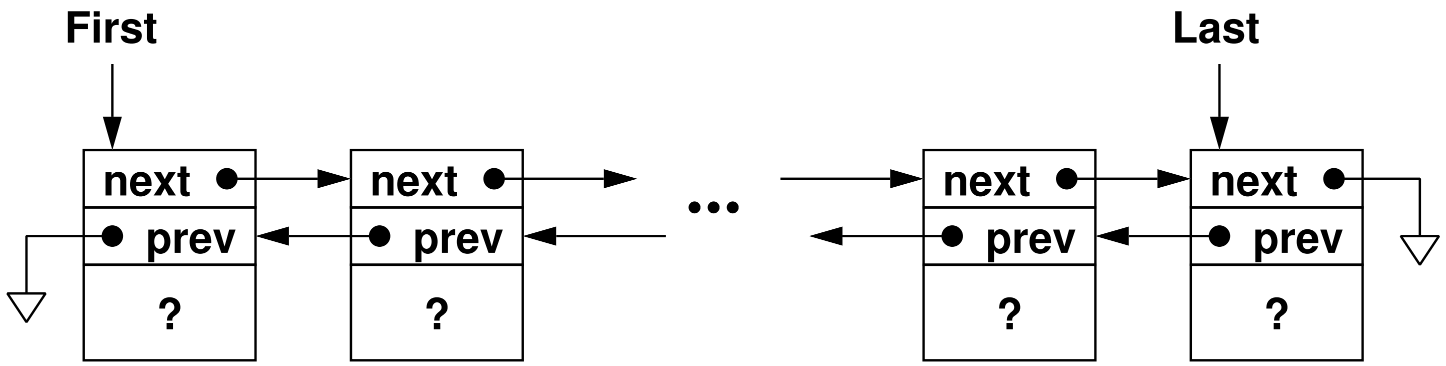
A Linked-List Abstraction

- A list of elements, linked so that you can move from one to the next (and/or previoius)
  - each element holds an object of some sort
- ***Functionally:***
  - First()
  - Next()
  - Last()
  - Prev()
  - Insert()
  - Remove()
  - Count()
- Need to have a well-defined interface
  - once you have a good interface, if the implementation is broken, fix the implementation!
    - don’t fix the "application"

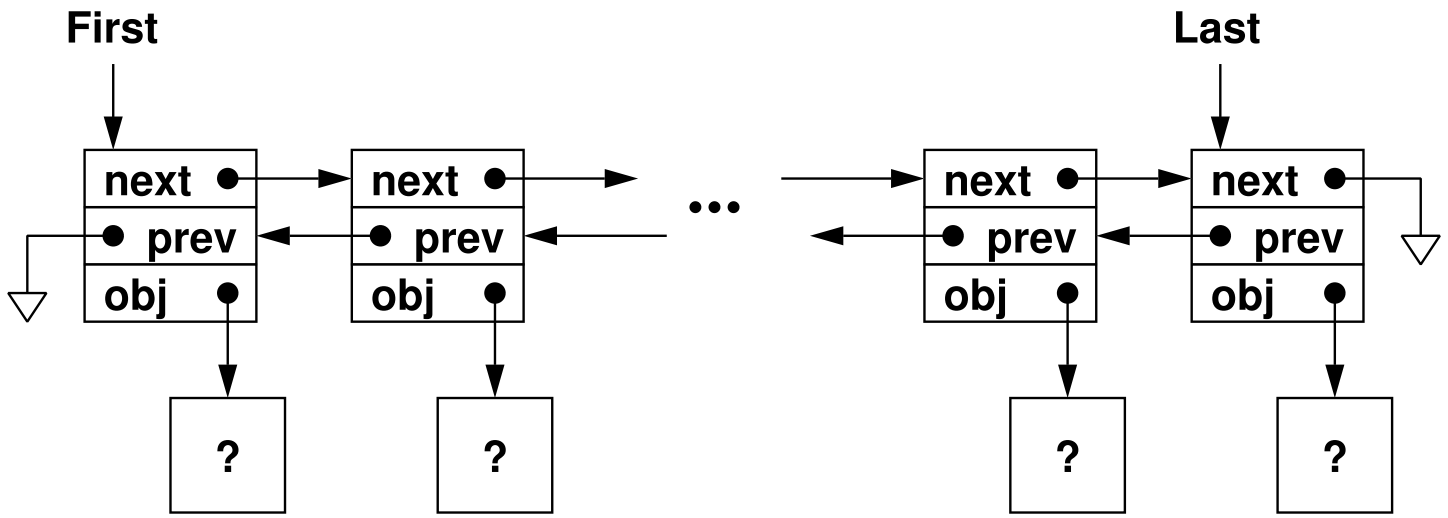
A Linked-List Abstraction

- There are basically two types of lists
  1) next/prev pointers in list items
  2) next/prev pointers outside of list items

- (1) has a major drawback that a list item cannot be inserted into multiple lists

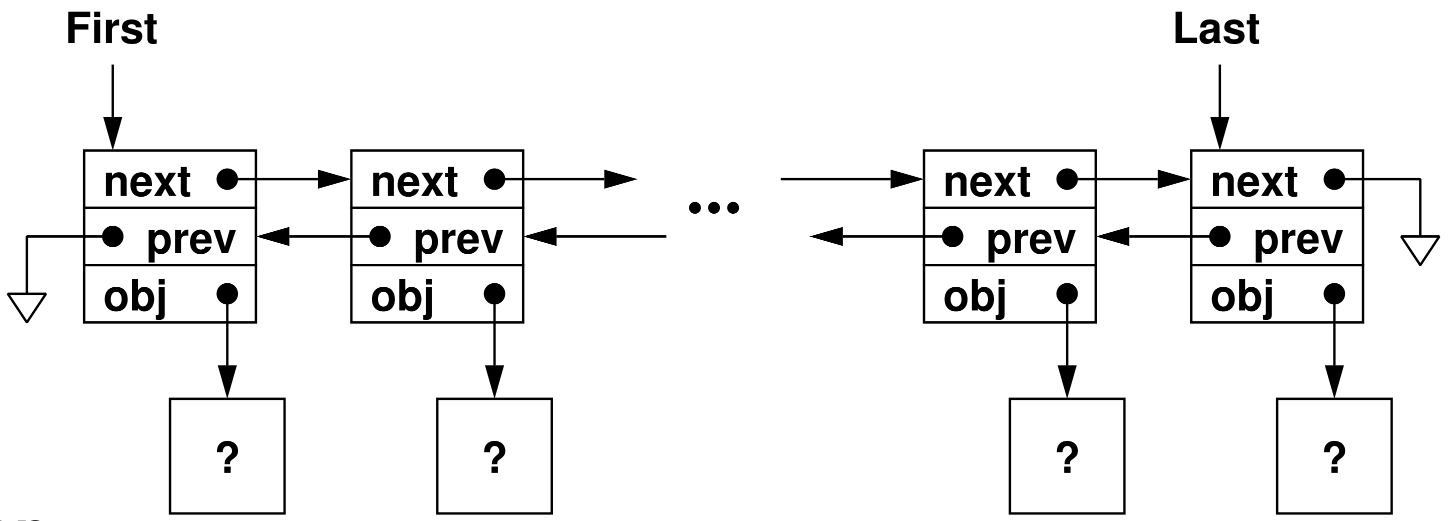


  - We will implement (2)

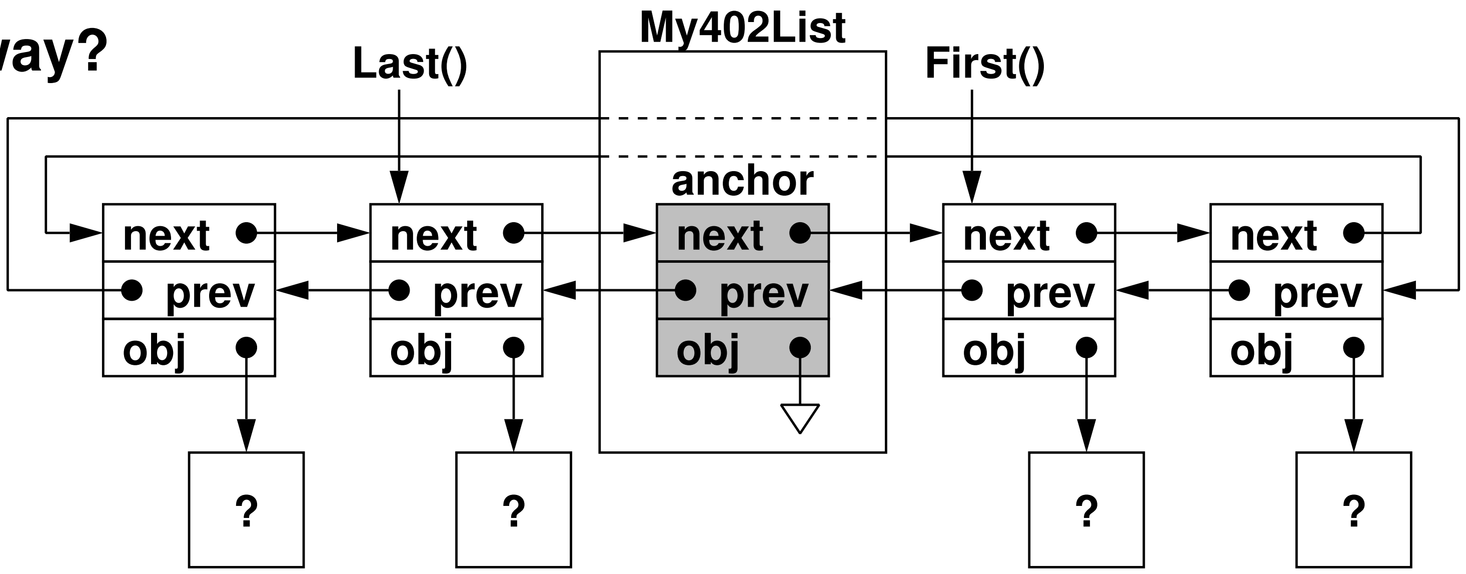


Doubly-linked Circular List

- **Abstraction**



- **Implementation**
  - **why this way?**



  - **your job is to implement the traditional list abstraction using a circular list**

my402list.h

```
#ifndef _MY402LIST_H_
#define _MY402LIST_H_

#include "cs402.h"

typedef struct tagMy402ListElem {
    void *obj;
    struct tagMy402ListElem *next;
    struct tagMy402ListElem *prev;
} My402ListElem;

typedef struct tagMy402List {
    int num_members;
    My402ListElem anchor;

    /* You do not have to set these function pointers */
    int  (*Length)(struct tagMy402List *);
    int  (*Empty)(struct tagMy402List *);

    int  (*Append)(struct tagMy402List *, void*);
    int  (*Prepend)(struct tagMy402List *, void*);
    void (*Unlink)(struct tagMy402List *, My402ListElem*);
    void (*UnlinkAll)(struct tagMy402List *);
```

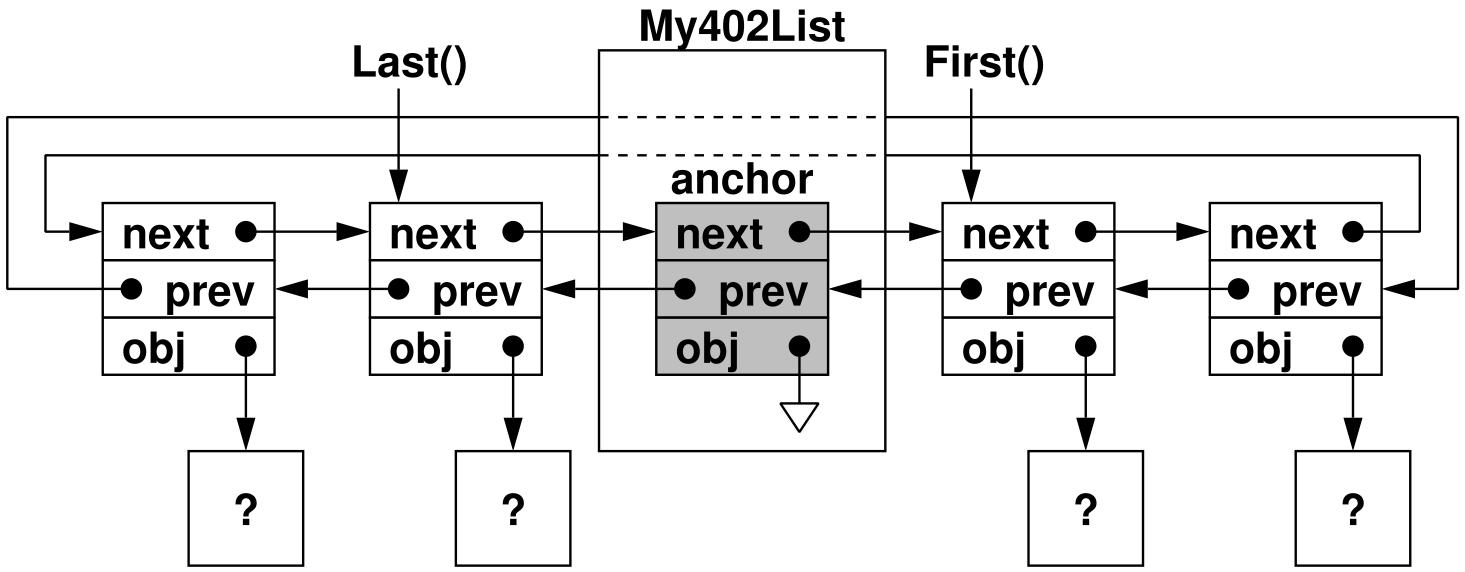
- **You need to learn to ignore things you don’t understand**
  - **assume that they are perfect**

my402list.h

```
    int   (*InsertBefore)(struct tagMy402List *, void*, My402ListElem*);
    int   (*InsertAfter)(struct tagMy402List *, void*, My402ListElem*);

    My402ListElem *(*First)(struct tagMy402List *);
    My402ListElem *(*Last)(struct tagMy402List *);
    My402ListElem *(*Next)(struct tagMy402List *, My402ListElem *cur);
    My402ListElem *(*Prev)(struct tagMy402List *, My402ListElem *cur);

    My402ListElem *(*Find)(struct tagMy402List *, void *obj);
} My402List;

extern int  My402ListLength(My402List*);
extern int  My402ListEmpty(My402List*);

extern int  My402ListAppend(My402List*, void*);
extern int  My402ListPrepend(My402List*, void*);
extern void My402ListUnlink(My402List*, My402ListElem*);
extern void My402ListUnlinkAll(My402List*);
extern int  My402ListInsertAfter(My402List*, void*, My402ListElem*);
extern int  My402ListInsertBefore(My402List*, void*, My402ListElem*);

extern My402ListElem *My402ListFirst(My402List*);
extern My402ListElem *My402ListLast(My402List*);
extern My402ListElem *My402ListNext(My402List*, My402ListElem*);
extern My402ListElem *My402ListPrev(My402List*, My402ListElem*);

extern My402ListElem *My402ListFind(My402List*, void*);

extern int My402ListInit(My402List*);
#endif /*_MY402LIST_H_*/
```

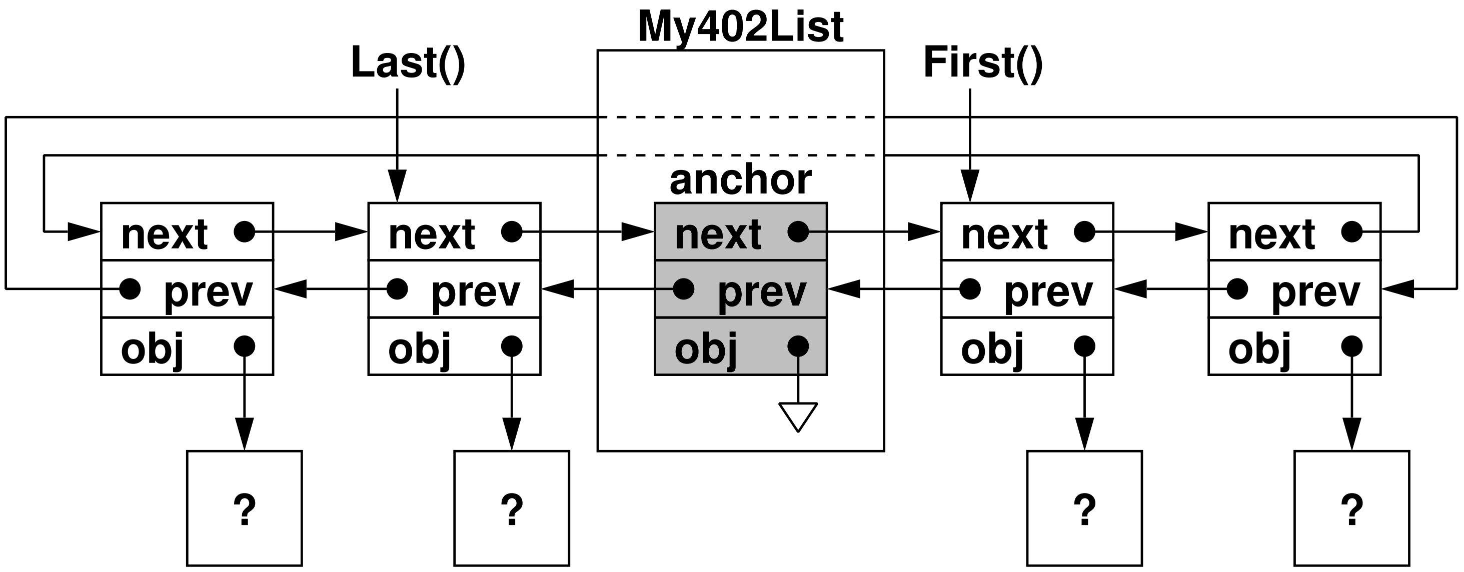
⇨ **You need to implement all the mentioned functions**

Implementation



```
int Length() { return num_members; }
int Empty() { return num_members<=0; }

int Append(void *obj);
int Prepend(void *obj);
void Unlink(My402ListElem*);
void UnlinkAll();
int InsertBefore(void *obj, My402ListElem *elem);
int InsertAfter(void *obj, My402ListElem *elem);

My402ListElem *First();
My402ListElem *Last();
My402ListElem *Next(My402ListElem *cur);
My402ListElem *Prev(My402ListElem *cur);

My402ListElem *Find(void *obj);
```

Usage - Traversing the List



```
void Traverse(My402List *list)
{
  My402ListElem *elem=NULL;

  for (elem=My402ListFirst(list);
       elem != NULL;
       elem=My402ListNext(list, elem)) {
    Foo *foo=(Foo*)(elem->obj);

    /* access foo here */
  }
}
```

- This is how an *application* will use `My402List`
  - you must support this "*contract*" with you application

`listtest`

- Use provided `listtest.c` and `Makefile` to create `listtest`
  - `listtest` must run without error and you must not change `listtest.c` and `Makefile`
  - They specifies how your code is expected to be used
- You should learn how to run `listtest` under `gdb`

Sort Command

- `warmup1 sort [tfile]`
  - Produce a sorted transaction history for the transaction records in `tfile` (or `stdin`) and compute balances
- Input is an ASCII text file
  - Each line in a `tfile` contains 4 fields delimited by `<TAB>`
    - transcation type (single character)
      - `"+"` for deposit
      - `"-"` for withdrawal
    - transcation time (UNIX time)
      - `man -s 2 time`
    - amount (a number, a period, two digits)
    - transcation description (textual description)
      - cannot be empty
- Output must be in the specified format exactly
  - Use the grading guidelines to check if you miss something
    - formatting bugs should be very easy to fix

Sort Command

Output

```
00000000011111111112222222222333333333344444444445555555555666666666677777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890

+-----------------+--------------------------+----------------+----------------+
|       Date      | Description              |         Amount |        Balance |
+-----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                      |      1,723.00  |      1,723.00  |
| Wed Dec 31 2008 | ...                      | (       45.33) |      1,677.67  |
| Mon Jul 13 2009 | ...                      |     10,388.07  |     12,065.74  |
| Sun Jan 10 2010 | ...                      | (      654.32) |     11,411.42  |
+-----------------+--------------------------+----------------+----------------+
```

How to keep track of balance

- First thing that comes to mind is to use `double`
- The weird thing is that if you are not very careful with double, your output will be wrong (by 1 penny) once in a while
- Recommendation: keep the balance in cents, not dollars
  - No precision problem with integers!

Sort Command

```
00000000011111111112222222222333333333344444444445555555555666666666677777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890

+-----------------+--------------------------+----------------+----------------+
|       Date      | Description              |         Amount |        Balance |
+-----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                      |       1,723.00 |       1,723.00 |
| Wed Dec 31 2008 | ...                      | (       45.33) |       1,677.67 |
| Mon Jul 13 2009 | ...                      |      10,388.07 |      12,065.74 |
| Sun Jan 10 2010 | ...                      | (      654.32) |      11,411.42 |
+-----------------+--------------------------+----------------+----------------+
```

- **The spec requires you to call `ctime()` to convert a Unix timestamp to string**
  - **then pick the right characters to display as date**
  - **e.g., `ctime()` returns `"Thu Aug 30 08:17:32 2012\n"`**
    - **becareful, `ctime()` returns a pointer that points to a *global variable*, so you must *make a copy***

```
char date[16];
char buf[26];
strncpy(buf, ctime(...), sizeof(buf));
date[0] = buf[0];
date[1] = buf[1];
...
date[15] = '\0';
```

Sort Command

```
000000000111111111122222222223333333333444444444455555555556666666666777777777778
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

+-----------------+--------------------------+----------------+----------------+
|       Date      | Description              |         Amount |        Balance |
+-----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                      |      1,723.00  |      1,723.00  |
| Wed Dec 31 2008 | ...                      | (       45.33) |      1,677.67  |
| Mon Jul 13 2009 | ...                      |     10,388.07  |     12,065.74  |
| Sun Jan 10 2010 | ...                      | (      654.32) |     11,411.42  |
+-----------------+--------------------------+----------------+----------------+
```

- Format your data in your own buffer
  - write a function to "format" numeric fields into null-terminated strings
    - it's a little more work, but you really should have this code isolated
      - in case you have bugs, just fix this function
  - you can even do the formatting when you append or insert your data structure to your list
    - need more fields in your data structure
  - this way, you can just print things out easily
  - use `printf("%s", ...)` to print a field to `stdout`

Warmup #1

- I'm giving you a lot of details on how to do things in C
  - this is the first and last assignment that I will do this!
  - you must learn C (and Unix) on your own
- Read man pages
- Ask questions in class Google Group
  - or send e-mail to me
- Come to office hours, especially if you are stuck

Warmup #1 - Miscellaneous Requirements

- Run your code against the *grading guidelines*
  - must not change the test program
- You must not use any *external code fragments*
- You must not use *array* to implement any list functions
  - must use pointers
- If input file is large, you must not read the whole file into into a large memory buffer
- It's important that every byte of your data is read and written correctly.
  - `diff` commands in the grading guidelines must *not* produce *any* output or you will not get credit
- Please see Warmup #1 spec for additional details
  - please read the *entire* spec *yourself*

ITS Solaris Machine Access

- You need to log into aludra/nunki.usc.edu
  - if your USC e-mail address is YOURLOGIN@usc.edu
    - then your login name is YOURLOGIN (same password)
  - for warmup assignments and to run `bsubmit`
  - SSH from a console (make sure to use "`ssh -X -Y ...`")
  - On Windows, use VirtualBox, Xwin, Cygwin or PuTTY
    - Ubuntu (Linux)
- Transferring Files
  - "`scp`" from a console
  - SFTP/SCP programs
    - Cyberduck, Fugu, etc. (Mac)
    - FileZilla, WinSCP, etc. (Windows)
- Text Editors
  - `emacs`, `pico`, `vi`
- Compiler
  - "`gcc --version`" should say it's version 4.something

ITS Solaris Machine Access

- On Windows, try to avoid `FileZilla` if you can and just use `putty`
  - actually, if you have Ubuntu 14.04 installed or if you are using a Mac, you should use `scp`
  - if not, use `putty` as the `ssh` client and use `sftp` to transfer files between your laptop and `nunki`

- Get familiar with "*Warmup #1 FAQ*" and "*Programming FAQ*"