6 Lecture: Wrapping Up C

Outline:

6.1 Announcements

- Coming attractions:

Event	Subject	Due Date	Notes

  Use your own discretion with respect to timing/due dates.

- on partner ones, only one copy
- Pre-existing stdin, stdout, stderr.
- **Array stuff we skipped last time**
- Discuss readlonglines
- C Language Quiz coming up
  - Will be similar to the questions published on the web page
  - Old exams on the web. (think 202-like material)
  - Bring questions for review Friday before the exam
- Piazza link on web page
- Headers and libraries: They're in the man page
- Skim Ch. 2.

- Assignment stats:

Assignment	Submitted	Make	Compiled	Passed all	Passed none	complaints
Lab01	60	—	59	50	0	0(!)
Asgn1	58	—	58	50	0	0(!)

- Asgn2: fw
  - Reading long strings and `realloc(3)`
  - Relationship between reading/writing, counting, finding the top k, and reading long strings: not much. Maintain abstraction.
  - If you're using a hash table, sorting after collecting them all is the approach I'd recommend. Note:
    * There's no way to know in advance which will be your k most common words: so you have to keep all the words you are given.
    * If there are n words on the input stream, of which m are unique, To check each one against your top k would require at best $O(n \lg k)$ time.
    * To do the same thing at the end with the m unique words would be $O(m \lg k)$. Since $m \leq n$, you can't lose and you're very likely to win since it's unlikely that there will be not a single repeat in your input stream of words.
    * You have to build your own data structures
- -O — turns on dataflow analysis, but can break gdb

6.2 Notes on Asgn2

- All code here must be your own.
- Design! This program decomposes fairly well.
- Reading long strings and `realloc(3)`
- My asgn2 is 399 lines total. (this is not a large program)
- `time(1)`
- `-O` turns on the optimizer
- Go ahead and look up a hash function, just don't appropriate someone's code. (cite it!)
- "Notes on Partnerships" — why

See example in Figure 36.
`/usr/man` on unixX is approx. 110MB
`/usr/share/dict/words` is 4.8MB and contains 480k words.

6.3 Thoughts on the assignment

Chapter 6; Arrays, Structures, enum, typedef, unions?

```
lagniappe% wc /usr/share/dict/words
 99171  99171 938848 /usr/share/dict/words
lagniappe%
falcon% time fw.solaris /usr/dict/words /usr/share/man/*/*
The top 10 words (out of 69900) are:
   452919 fr
   275401 the
   201833 sp
   156752 in
    88201 to
    87961 fb
    81270 pp
    78950 n
    76514 is
    75718 of
6.61u 0.53s 0:12.01 59.4%
falcon%
```

Figure 26: fw in action

6.4 Pointers to Structures: More Complex

More or less everything works in the same way with one shortcut.

```
struct list_node n;
node np;

np = &n;
n.data = 3;
n.next = NULL;
```

Dereferencing deep structures:

```
(*((*((*n).next)).next)).data
```

vs.

```
n->next->next->data
```

6.5 Example: Dynamic Data Structures (linked list)

In which we look at various aspects of dynamic data structures and trees that might be of use in the homework.

6.5.1 Dynamic Data Structures

When dealing with dynamic data:

- Remember to allocate memory for it

- Be sure it's enough
- Be sure it succeeded.
- When you blow it: Use the debugger to figure out where.

6.5.2 Writing and recovering data: A linked list example

These code snippets (Figures 27 through 31) show one way of writing and recovering a linked list of integers.

Fundamentally, the process consists of choosing a format that will faithfully represent your data structure and implementing it.

6.5.3 Or it could be done recursively

As in Figures 32 and 33.

```
typedef struct node_st *node;
struct node_st {
  int data;
  node next;
};
```

Figure 27: A struct for a linked list of integers

```
node new_node(int data){
  /* Allocate and initialize a new node with
   * the given data value.
   * Returns the node on success, NULL on
   * failure
   */
  node n;

  n = (node) malloc(sizeof(struct node_st));
  if ( ! n ) {
    perror("malloc");
    exit(1);
  }
  n->data = data;
  n->next = NULL;
  return n;
}
```

Figure 28: Allocating a new node

```
void write_list(FILE *outfile, node list) {
  /* Write the current node to the given stream
   */
  while( list ) {
    fprintf(outfile, "%d\n", list->data);
    list = list->next;
  }
}
```

Figure 29: A function to write a list of integers (iterative version)

```
node append_list(node list, node rest){
  /* concatenate two well–formed lists */
  node t;
  if ( list ) {
    for(t=list; t->next != NULL; t=t->next )
      /* nothing */;
    t->next = rest;
  } else {
    list = rest;
  }
  return list;
}

node read_list(FILE *infile) {
  /* Read the current node, then read the rest
   * Reads numbers from infile until either EOF or a
   * non–number is found.
   */
  int num;
  node n, list;

  list = NULL;

  while ( 1 == fscanf(infile," %d",&num) ) {
    /* a return value of 1 indicates a successful match */
    n = new_node(num);
    list = append_list(list,n);
  }

  return list;
}
```

Figure 30: A function to read a list of integers (iterative version)

```
node read_list(FILE *infile) {
  /* Read the current node, then read the rest
   * Reads numbers from infile until either EOF or a
   * non-number is found.
   */
  int num;
  node n, list;
  node *tail;       /* note: this is a double pointer! */

  list = NULL;
  tail = &list;

  while ( 1 == fscanf(infile," %d",&num) ) {
    /* a return value of 1 indicates a successful match */
    n = new_node(num);
    *tail = n;                  /* tie the new node in */
    tail = &n->next;            /* move the tail pointer along */
  }

  return list;
}
```

Figure 31: A more clever function to read a list of integers (iterative version)

```
void write_list(FILE *outfile, node list) {
  /* writes the current node, then the rest
   */
  if ( list ) {
    fprintf(outfile, "%d\n", list->data);
    write_list(outfile,list->next);
  }
}
```

Figure 32: A function to write a list of integers (recursive version)

```
node read_list(FILE *infile) {
  /* read the current node, then read the rest
   * Reads numbers from infile until either EOF or a
   * non–number is found.
   */
  int num;
  node n;

  if ( 1 == fscanf(infile," %d",&num) ) {
    /* a return value of 1 indicates a successful match */
    n = new_node(num);
    n->next = read_list(infile);
  } else {
    n = NULL;
  }

  return n;
}
```

Figure 33: A function to read a list of integers (recursive version)

6.6 Compound Data Wrapup

6.6.1 So Far

- Arrays
- Structs

6.6.2 Unions and enumerated types

There are two more types we haven't discussed: unions and enums

An enumerated (**enum**) type allows you to use symbolic names for values:

```
typedef enum {mon,tue,wed,thu,fri,sat,sun} day;
day tomorrow = fri;
```

or

```
typedef enum {false, true} boolean;
boolean b = false;
```

A **union** is a like a structure except all of the fields occupy the same space.

```
union kludge{
  int num;
  char bytes[sizeof(int)];
};
int i;
union kludge k;
k.num = 0x0A0B0C0D;
for(i=0; i<sizeof(k.num); i++)
   printf("Byte[%d]: 0x%02x\n", i, k.byte[i]);
putchar('\n');
```

6.7 Pointer review: It's all the same

Pointers are memory addresses.

- Pointers to simple data
- Pointers to compound data
- Pointers to functions? See below

6.7.1 Pointers to Functions: Ridiculous

Don't even worry about knowing about these at this point. It's bad enough to know they exist:

Declaration: int (*fname)(int);

Asignment: (assiming foo() exists) fname = foo;

Use: x = (*fname)(2)

6.7.2 Summing up

Things you know:

- Dynamic data structures are no more complicated than allocating space and remembering where you put things.
- A pointer is a variable that contains a memory address.
- Pointers are uninitialized; they do not necessarily point at anything in particular.
- Operators:

&	address of
`*`	dereference

- NULL is a standard invalid pointer.
- Think *types*. Pointer vs. pointee.

  Given an integer pointer, p stored at address 0x12345678 and an integer i, stored at 0xABCD-ABCD, containing the value 3:

```
int i=3;
int *p=&i
```

Expression:	&p	p	*p
Type:	int **	int *	int
Value:	0x1234	0xABCD	3

	Expression					
Variable	**&x**	**x**	***x**	****x**	*****x**	******x**
int x	int *	int	—	—	—	—
int *x	int **	int *	int	—	—	—
int **x	int ***	int **	int *	int	—	—
int ***x	int ****	int ***	int **	int *	int	—

- `sizeof()` is a macro that evaluates to the size of a given data structure.
- The name of an array is equivalent to the address of that array. Even though it is casually called a pointer, it is not a variable.
- The same is true of the name of a function.
- Free not that which thou didst not malloc().

6.8 C odds and ends

6.8.1 Short-circuit evaluation

C conditionals are evaluated left to right and evaluation stops as soon as the overall sense of the expression is known.

Thus, the following is safe:

```
if ( p && *p ) ...
```

6.8.2 The Ternary Operator

Useful in some situations, but never at the cost of clarity. Good usage:

```
printf("Found %d word%s.\n", num, (num==1)?"":"s");
```

6.8.3 The Comma Operator

The operands of *comma* are evaluated left to right, and the value of the overall expression is the value of the rightmost operand. (i.e., the value of x, y is y)

For use, see Figure 35.

```
void Pof2(int n) {
    /* print the first n powers of two */
    unsigned long i, num;
    for(i=0, num=1; i < n; i++, num*=2)
        printf("2^%-2d = %u\n", i, num);
}
```

Figure 34: Pof2(): Printing the first n powers of 2.

6.8.4 Variable modifiers

unsigned
const
extern
static
volatile
register
restrict

6.8.5 Promotion rules

C data types will automatically be promoted to "wider" types. Note that while magnitude is preserved, precision may not be. (e.g., `int` to `float`)

Promotion is only done as needed. Consider

```
f = 100.0 * 3/4;
g = 3/4 * 100.0;
printf("f = %f\ng = %f\n", f, g);

f = 75.000000
g = 0.000000
```