5 Lecture: Complex Data and Dynamic Data Structures

Outline:

5.1 Announcements

- Coming attractions:

Event	**Subject**	**Due Date**			**Notes**
lab01	warm up	Wed	Oct 4	23:59	
asgn1	detab	Mon	Oct 9	23:59	

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

- valgrind
- Missing office hours this afternoon
- Gradebook
- TUI (Text User Interface) mode (C-x A)
- Shell history tricks
- Shell quotes
- Redirection details
- gcc -v to get search path

```
/usr/lib/gcc/x86_64-linux-gnu/5/include
/usr/local/include}
/usr/lib/gcc/x86_64-linux-gnu/5/include-fixed
/usr/include/x86_64-linux-gnu
/usr/include
```

- Lab02 and asgn2 now have due dates
- Lab02: Reading long lines
  - Guess a length
  - Be prepared to go back for more
- -O — turns on dataflow analysis, but can break gdb
- Asgn2: `tryAsgn2` is out.

5.2 Today

1. gdb demo
2. Multi-dimensional arrays vs. arrays of arrays
3. Let's build a list

5.3 Pointers: Simple

5.3.1 Declaration and use

Pointers are *typed.* That is, a pointer must know what sort of thing it is pointing to.

Pointers to basic types: int *, char *, double *, void *

Assignment

Operators:

`&`	address of
`*`	dereference

Note:

- To do anything useful, they must be *pointed at* something.
- `NULL` is defined to be an invalid pointer. It's not necessarily zero, but guaranteed to compare as zero. (it's a standards thing.)

The standard example: `swap(int *a, int *b)` can be seen in Figure 17.

```
void swap ( int *a, int *b ) {
  /* using indirection, we can exchange the values */
  int t;
  t  = *a;
  *a = *b;
  *b = t;
}

/* A sample call: */
int x,y;
swap(&x,&y);
```

Figure 17: Exchanging two values: `swap.c`

5.4 Pointers and Arithmetic

Because pointers are memory addresses, they are subject to arithmetic and other operations, but in order for them to make sense, we have to talk about arrays again..

Onwards: Before discussing pointers with respect to compound data types, we need to talk about C's compound data types.

5.5 Pointers and Arrays

Arrays are allocated blocks of data of the given type.

An array's name can be used as a pointer to the head of the array, so, given `int foo[5]` exists, `foo[3]` is equivalent to `*(foo+3)`.

$$\begin{aligned} A[3] &= *(A+3) \\ &= *(3+A) \\ &= 3[A] \end{aligned}$$

5.6 Arrays as parameters

Array parameters can be declared two ways:

- `type *name`
- `type name[]` (Empty-braces form only in parameter lists.)

5.7 Strings

Strings are arrays of characters with a zero byte at the end. Compare the different implementations of two library calls:

Function	**Header**	**Array**	**Pointer**
`size_t strlen(const char *s);`	`#include<string.h>`	Fig. 18	Fig. 19
`int puts(const char *s);`	`#include<stdio.h>`	Fig. 20	Fig. 21

Figures 18 through 21 illustrate the difference between approaching a parameter as an array vs. as a pointer.

5.8 Add a discussion of scope and lifetime here

- Automatics
- data segment/heap objects (small=o)

5.9 From last time: Dynamic allocation

void *malloc(size_t size) Allocates a new block of memory of the requested size and returns a pointer to it (or returns NULL).

void free(void *ptr); returns the pointed-to segment of memory to the library pool to be reallocated by a subsequent call to malloc();

```
int strlen ( char s[] ) {
  int len;
  for ( len = 0 ; s[len] != '\0' ; len++ )
      /* nothing */;
  return len;
}
```

Figure 18: An array-based version of `strlen()`.

```
int strlen ( char *s ) {
  /* this might be a bit too clever for its own good. */
  int len = 0;
  while ( *s++ )
    len++;
  return len;
}
```

Figure 19: A pointer-based version of `strlen()`.

```
void puts ( char s[] ) {
  int i;
  for ( i = 0 ; s[i] != '\0' ; i++ )
    putchar(s[i]);
}
```

Figure 20: An array-based version of `puts()`.

```
void puts ( char *s ) {
  while( *s != '\0' )
    putchar( *s++ );
}
```

Figure 21: A pointer-based version of `puts()`.

void *calloc(size_t nmemb, size_t size); allocates memory for an array of `nmemb` elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

void *realloc(void *ptr, size_t size) takes the given block of memory and allocates a (possibly different) block of memory of `size` bytes with the same contents as the original block. `realloc(3)` returns a pointer to the new block and may free the old one.

5.10 Pointers Summary

Pointers and arrays (Chapter 5).
(A quick review of pointer vs. pointee. Also, see Fig. 22)

- Pointers are just that: pointers.
- To do anything useful, they must be *pointed at* something.
- `NULL` is defined to be an invalid pointer. It's not necessarily zero, but guaranteed to compare as zero. (it's a standards thing.)

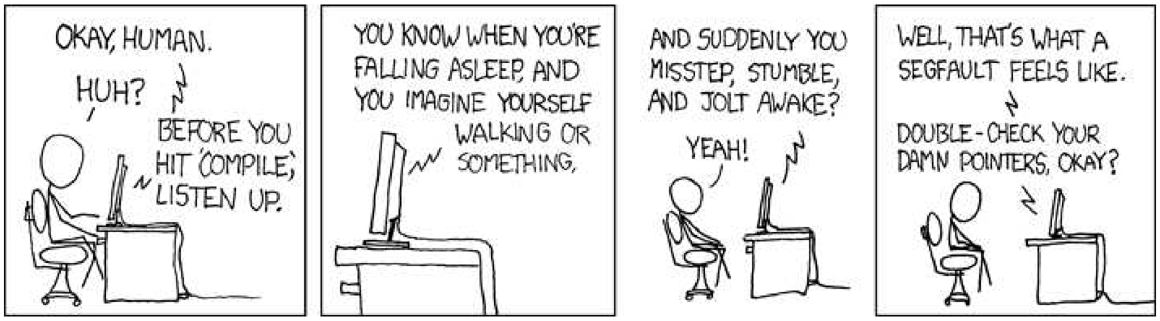

OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Figure 22: `http://xkcd.com/371/`

5.10.1 Strings and dynamic memory: `strdup()`

Function	**Header**	**Implementation**
`char *strdup(const char *s);`	`#include<string.h>`	Fig. 23

Take care as to whether it's reasonable to treat your data as a string at all. (e.g. in `xlat`, '\0' is a valid character.

```
char *strdup(char *s) {
  /* returns a copy of the given string in a newly allocated
   * region of memory
   */
  char *new=NULL;
  if ( s ) {
    new = malloc ( strlen(s)+1 );
    if ( new )
      strcpy(new,s);
  }
  return new;
}
```

Figure 23: A string duplication function: `strdup()`.

5.11 Tool of last century: rcs(1)

RCS (Revision Control System) is a version-control system.

This is very old fashioned and superceded by CVS, SVN, git, etc., but it requires no setup.

mkdir RCS Create repository

ci -u foo.c Checks in "foo.c" and removes it

ci -u foo.c Checks in "foo.c" and keeps a read-only copy (unlocked)

ci -i foo.c Checks in "foo.c" and keeps a read-write copy (unlocked)

co -u foo.c Checks out "foo.c" read-only (unlocked)

co -l foo.c Checks out "foo.c" read-write (locked)

rcsdiff foo.c shows the difference between this version and the last

rlog foo.c show the log messages.

Emacs interface:

C-x v i Install buffer into RCS

C-x C-q Toggle-read-only checks the file in or out

C-c C-c Complete checkin

5.12 Tool of the week: Make(1)

Make is a program to control program builds automagically, but it can be much, much more.

- Based on dependencies—there's no need to regenerate a file if its source hasn't changed.

  A dependency looks like: `target:  source`.

  A particular target can have multiple dependency lines

- Implicit Rules—Make knows how to do certain things (compile C source, for example: if a .o file depends on a .c file if there is one of the same name in the directory).

  You can define your own if you want.

- Explicit rules: After a tab, an series of instructions for making the thing:

```
foo.o: foo.c
	gcc -c -Wall -ansi -pedantic foo.c
```

- It supports variables. In particular CC, SHELL, CFLAGS.

  \$(VARNAME) to evaluate. \$\$ for a dollar sign

- Remember TABS

  (Quite possibly the dumbest decision since `creat()`).

- In rules:

@	do a line silently
#	comment
-	proceed even in the face of errors

- Interfaces nicely with RCS (will check out files for you.)
- To use: `make thing`

  if "thing" isn't specified, it makes the first target.
- Emacs: M-x compile

For more info, read the man page, or the info page.

Arrays as parameters Array parameters can be declared two ways:

- `type *name`
- `type name[]` (Empty-braces form only in parameter lists.)

Multi-dimensional arrays

- declaration
- as parameter (the first dimension can be unspecified)

We've been doing arrays. There's no news there.

5.13 Subtle Distinctions: Multidimensional Arrays vs. arrays of arrays

Many programmers are sloppy about vocabulary.

5.13.1 Example: Two-d Array

```
int A[2][3];
int i,j;
for(i=0;i<2;i++)
  for(j=0;j<3;j++)
    A[i][j] = 10 * i + j;
```

Logical layout:

0	1	2
10	11	12

Physical layout:

0	1	2	10	11	12

5.13.2 Example: Strings and command-line arguments

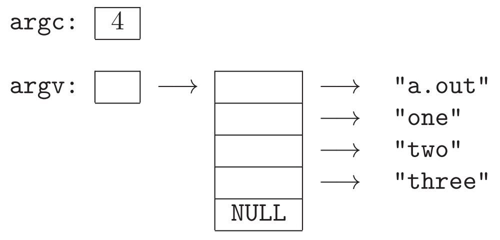
C strings are simply arrays of characters with a nul ('\0') at the end.

Pointers and strings play into the "real" definition of how a C main() is supposed to be defined. The definition takes the number of command-line arguments and an array of their values: `main(int argc, char *argv[])`

So, if your program is invoked as:

```
% a.out one two three
```

these parameters would look like:



If you wanted to print these command line arguments out, you could do something like the program shown in Figure 24. If you're an old-timer and like pointer arithmetic, you could do something like Figure 25, but don't. First of all, it's impossible to read, and second, it destroys the parameters in the process. What if you needed them?

5.13.3 Structures and Typedef

A structure allows you to group data:

```
struct list_node {
  int data;
  struct list_node *next;
};
```

Note in the example above that it is possible to use the type definition to define a pointer type before it is complete.

Select a field with ".", like:

```
struct list_node thing;
thing.data = 3;
thing.next = NULL;
```

A **typedef** allows you to define a new type. It's just like declaring a variable except the "variable" becomes the name for the new type:

```
typedef struct list_node *node;
```

or, in conjunction with the definition of the node:

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

5.14 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
```

```
#include <stdio.h>

int main(int argc, char *argv[]){
  int i;
  for ( i = 0; i < argc; i++ ) {
    printf("%s ", argv[i]);
  }
  printf("\n");

  return 0;
}
```

Figure 24: Echoing command-line arguments.

```
#include <stdio.h>

int main(int argc, char *argv[]){

  while ( argc-- )
    printf("%s ", *argv++);
  printf("\n");

  return 0;
}
```

Figure 25: Hard to read version of Figure 24.