4 Lecture: Pointers and Arrays in C

Outline:

Announcements
Array types
A programming example: cat
    The stdio library
    A program example: `cat`
The C Memory Model
Pointers and Addresses
    Basics
Pointers: Simple
    Declaration and use
Pointers and Arithmetic
Pointers and Arrays
Arrays as parameters
Strings
Add a discussion of scope and lifetime here
Dynamic Memory Management: malloc() and free()
    Strings and dynamic memory: `strdup()`
Error handling
    Example: safe_malloc()
Tool of the week: Make(1)

4.1 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.

- Lab02 will be out shortly.

  1. A few written exercises
  2. Picture
  3. GDB
  4. Learn Make(1)
  5. write `uniq(1)`

- Asgn2 out shortly
- how to determine which headers you need
- `getopt(3)` is not ANSI. It's still in the library, though.
- Web authentication will be through the portal. (for solutions)
- Reminder about how to use late days

4.2 Array types

4.3 A programming example: cat

Last time we did a major overview of C, let's do some.

4.3.1 The stdio library

This is Chapter 7 of K&R.

Based on `FILE *` streams

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int feof(FILE *stream);
```

4.3.2 A program example: `cat`

```
#include <stdio.h>

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

  for(c=getchar(); c != EOF ; c = getchar() ) {
    putchar(c);
  }

  return 0;
}
```

Figure 7: A simple implementation of `cat()`

```
#include <stdio.h>

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

  while ( (c=getchar()) != EOF )
    putchar(c);

  return 0;
}
```

Figure 8: A more clever version `cat()`

```
#include <stdio.h>

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

  while ( EOF != putchar(c=getchar()))
    /* This program is broken */;

  return 0;
}
```

Figure 9: A version that's so clever it's broken

4.4 The C Memory Model

C exposes memory completely to the programmer:

- quick review of what memory is and layout
- quick review of layout of process memory.
- quick review of memory access from the assembly level.

4.5 Pointers and Addresses

K&R Chapter 5: Pointers

4.5.1 Basics

Pointers expose the underlying memory system:

- Necessary: Allows functions to modify parameters, e.g.
- Dangerous: Allows functions to modify parameters

4.6 Pointers: Simple

4.6.1 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.

4.7 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.

```
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 10: Exchanging two values: `swap.c`

4.8 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)`.

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

4.9 Arrays as parameters

Array parameters can be declared two ways:

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

4.10 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.

4.11 Add a discussion of scope and lifetime here

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

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

Figure 11: 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 12: A pointer-based version of `strlen()`.

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

Figure 13: An array-based version of `puts()`.

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

Figure 14: A pointer-based version of `puts()`.

4.12 Dynamic Memory Management: malloc() and free()

(*remember C doesn't itself provide memory management. This is part of the library. Contrast to:*

```
int brk(void *end_data_segment);
void *sbrk(ptrdiff_t increment);
```

provided by the OS)

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();

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.

Using malloc() and free():

- Always use `sizeof()` for portability. This is particularly true for structures. (There may be gaps due to alignment issues.)
- Always cast your return value appropriately.
- **Always** check your return value for success and handle it appropriately. I usually write something like `safe_malloc()` (below)
- Always free() anything you allocated once you're done with it. (Memory leaks are a bad thing.) This can be trickier than it looks for complex programs. Reference counters can be helpful.
- Never free something that was not allocated by malloc().

4.12.1 Strings and dynamic memory: `strdup()`

Function	Header	Implementation
`char *strdup(const char *s);`	`#include<string.h>`	Fig. 15

```
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 15: A string duplication function: `strdup()`.

4.13 Error handling

Always check the return values of library functions and system calls. (except `exit()`)

```
#include <stdio.h>

void perror(const char *s);

#include <errno.h>

const char *sys_errlist[];
int sys_nerr;
int errno;
```

4.13.1 Example: safe_malloc()

This demonstrates error-checking for malloc() and perror() for error handling.
Errors set `errno` to indicate the error. Calls to `perror()` use this value, too.
From the Solaris Manual:

```
ENOMEM    The physical limits of the system are exceeded  by
          size  bytes of memory which cannot be allocated.

EAGAIN    There is not enough memory available at this point
          in time to allocate size  bytes of memory; but the
          application could try again later.
```

The code can be found in Figure 16

```
void *safe_malloc ( int size ) {
  void *new;
  new = malloc(size);
  if ( new == NULL ) {
    perror(_FUNCTION_);
    exit(−1);
  }
  return new;
}
```

Figure 16: A malloc() that always succeeds (or never comes back)

4.14 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.