Announcements

- **Midterm:**
  - Tuesday, June 23 from 7-8pm in UC 266
Announcements

• Extra Office Hours:
  • Friday, June 19, 1-3pm in BA3289
  • Monday, June 22, 1-3pm in BA3201
Announcements

- **Assignment 2:**
  - Has been released
  - Due on July 1, 2015 (*less than 2 weeks*)
  - No lecture after this one, before due date
Announcements

• Labs:
  • Consolidating down to BA2210 and BA2220 only
  • Everyone attending should muster there (there will not be TA’s in the other two rooms)
Agenda

• Standard library string functions
• Using the C Preprocessor for Sharing Definitions
• Makefiles
• Midterm
void *

... even though **void** is an *incomplete* type, has no values and you cannot instantiate any variable to be of type **void** ...
void *p = (void *) 0x1000;  

// p + 1 == (void *) 0x1001
// p + 2 == (void *) 0x1002
// p + 3 == (void *) 0x1003
// p + 4 == (void *) 0x1004
// ...
Strings in C
char
char

• The char datatype represents an 8-bit integer (is signed by default, alternatively can be declared as unsigned)

• Small integers can fit in a char sized value holder:
  • char x = 100;

• Oversized values will be truncated and wrap around:
  • 1 == (unsigned char) 257

• One ASCII character enclosed in single quotes is a char literal value:
  • ‘A’ == 65 since the ASCII code for A is decimal 65
  • ‘0’ == 48 since the ASCII code for 0 is decimal 48
  • ‘\0’ == 0 uses an escape code
  • ‘\n’ == 10 (linefeed aka newline)
Strings in C

- A **NULL** (zero) terminated array/sequence of `char` (*byte*) values

  - Typically passed around as a pointer (`char *`) to the first character

  - No extra information about length or maximum size

  - Modified *in place* (not necessarily copied)

- **String literals** will include an implicit NULL byte

  - `char s[] = "CSC"; s[3] == 0`
strrepr.c:

```c
char course1[] = { 67, 83, 67, 50, 48, 57, 0 };
char course2[] = { 'C', 'S', 'C', '2', '0', '9', '\0' };
char course3[] = "CSC209";
```

\[ \forall 0 \leq i \leq 6: \]

\[ \text{course1}[i] == \text{course2}[i] == \text{course3}[i] \]
C Standard Library String Functions

- A small but useful set of functions that help you to manipulate C-style strings
  - They require care and attention to detail when using
  - Many traps for young players; source of many bugs
- Don’t forget to `#include <string.h>` to get the function prototypes
strlen
strlen - calculate the length of a string

size_t strlen(const char *s)

From the manpage: “The strlen() function calculates the length of the string s, excluding the terminating null byte (\0).”
String Length vs Storage Size
strlen.c:

```c
char hello[] = { 'C', 'S', 'C', '2', '0', '9', '\0' };

printf("hello          = "\%s\n", hello);
printf("sizeof (hello) = %zu\n", sizeof (hello));
printf("strlen(hello)  = %zu\n", strlen(hello));
```

```
hello          = "CSC209"
sizeof (hello) = 7
strlen(hello)  = 6
```

length + 1 \(\leq\) size
strcmp
strcmp("CSC209", "CSC209") == 0

strcmp("CSCB09", "CSC209") > 0
since 'B' > '2' (ASCII 66 vs 50)

strcmp("CSC209", "CSC309") < 0
since '2' < '3' (ASCII 50 vs 51)
strchr and strlen
strchr & strrchr — locate first/last occurrence of a character within a string

char *strchr (const char *s, int c)
char *strrchr(const char *s, int c)

From the manpage: “The strchr() / strrchr() function returns a pointer to the first/last occurrence of the character c in the string s.”
strchr.c
strcat
strcat - append (concatenate) one string onto another

char *strcat(char *dest, const char *src)

From the manpage: “The strcat() function appends the src string to the dest string, overwriting the terminating null byte (‘\0’) at the end of dest, and then adds a terminating null byte. ... returns dest.”
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char s[16];
    int i;

    s[0] = '\0';

    // Concatenate each `argv[i]` onto `s`
    for (i = 1; i < argc; i++) {
        strcat(s, argv[i]);
        strcat(s, '!');
    }

    printf("%s\n", s);
    return 0;
}
```
argv[1] = "hello"
argv[2] = "csc209"
argv[3] = "fun"

strcat(s, argv[1]);
strcat(s, "!");
strcat(s, argv[2]);
strcat(s, "!");
strcat(s, argv[3]);
strcat(s, "!");
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```c
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
hello! '\0' - - - - - - - - - -

strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
hello! CSC209

```c
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```
hello!

```c
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
Hello! CSC209!
```c
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```
hello! CSC209! fun

```c
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```
Hello! CSC209! fun

```c
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```
hello! CSC209! fun

```c
strcat(s, "hello");
strcat(s, "!");
strcat(s, "CSC209");
strcat(s, "!");
strcat(s, "fun");
strcat(s, "!");
```

`s[0]` through `s[15]` are set to:

```c
s[0] = 'h';
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = ' ';
s[6] = 'C';
s[7] = 'S';
s[8] = 'C';
s[9] = '2';
s[10] = '0';
s[11] = '9';
s[12] = ' ';
s[13] = 'f';
s[14] = 'u';
s[15] = 'n';
```

`s[16]` and `s[17]` are set to:

```c
s[16] = '!
';
s[17] = '\0';
```
BOOM!!!
STACK SMASH
Caveat: This was on a Mac OS X laptop…

Try “fun12345678” on CDF for similar effect…
What was in memory at $s[16]$ and $s[17]$, and what other purpose was it serving?
Regardless, \texttt{strcat} went further in memory than \textit{we} should have allowed it.
Solution: keep track of how big `dest` will need to be, and correctly allocate sufficient space using `malloc`.
This is error prone and very easy to get wrong.
`strcat2.c`:

```c
size_t len = 0;

// Add up all string lengths
for (i = 1; i < argc; i++) {
    len += strlen(argv[i]);
    // Add +1 for each '!
    len++;
}

// Another +1 for the NUL terminator
char *s = (char *) malloc(len + 1);

s[0] = '\0';
```

strcat

- Only use it once you’ve determined the size of `src` and know that `dest` has sufficient `free` space to accommodate

- No way of telling `strcat` how large `dest` is (so that it won’t accidentally go beyond the end)
strcpy
strcpy - copy a string

char *strcpy(char *dest, const char *src)

From the manpage: “The strcpy() function copies the string pointed to by src, including the terminating null byte (\0), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. … returns dest.”
Whereas `strcat` looks for the `NUL` terminator of `dest` and then appends the contents of `src` starting there, `strcpy` overwrites from the beginning of `dest`. 
Hello CSC209!

CSC209!
Another trap: what happens if \texttt{src} is larger than \texttt{dest}?
char s[16];
s[0] = '\0';

strcpy(s, "The quick brown fox jumps over the lazy dog");
Introducing strncpy
`strncpy` - copy a string

```c
char *strncpy(char *dest, const char *src, size_t n)
```

From the manpage: “The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied.

**Warning:** If there is no zero byte among the first `n` bytes of `src`, the string placed in `dest` will not be zero terminated.”
It is your responsibility to keep track of how many bytes are free/unused in dest.
strncpy.c
You should *always* use `strncpy`, and *never* use `strcpy` because it is so outrageously unsafe.
See also `strncpy` and `strncat` (but they still only deal with `src` size, not `dest` size.)
strdup
strdup - duplicate a string

char *strdup(const char *s)

From the manpage: “The strdup() function returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc(3), and can be freed with free(3)…”
It’s not unusual for a C standard library function to modify a string argument *in-place.*
Functions that take a `const char *` argument are guaranteeing that they will not modify that argument.
```c
char *strdup209(const char *s) {
    size_t len = strlen(s);

    char *dup = (char *) malloc(len + 1);
    if (!dup) {
        return NULL;
    }

    int i;
    for (i = 0; s[i] != '\0'; i++) {
        dup[i] = s[i];
    }
    dup[i] = '\0';

    return dup;
}
```
String Function Summary

• Remember that we are just changing bytes in memory

• *Always* keep in mind who is taking responsibility for the zero/*\texttt{NUL}* terminating byte

  • … and ultimately, it is still up to you to ensure it is where it is suppose to be!

• Prefer the \texttt{strn—} variations whenever possible
• Find the length of a string:
  • `size_t strlen(const char *s)`

• Compare two strings:
  • `int strcmp(const char *s1, const char *s2)`
  • `int strncmp(const char *s1, const char *s2, size_t n)`

• Find the first/last occurrence of a character within a string
  • `char *strchr(const char *s, int c)`
  • `char *strrchr(const char *s, int c)`

• Append one string onto another:
  • `char *strcat(char *dest, const char *src)`

• Copy one string to another:
  • `char *strncpy(char *dest, const char *src)`
  • Avoid using `char *strcpy(char *dest, const char *src)`

• Duplicate a string:
  • `char *strdup(const char *s)`
Compilation Process
Compilation Process

* `.c` source code

C Compiler:

1. Pre-processor
2. Lexical Analysis
3. Parsing
4. Typecheck
5. Optimizations…
6. Codegen
7. Linking

Executable binary
Using the C Preprocessor for Sharing Definitions
example.c:

```
#include <stdlib.h>

...
void *ptr = malloc(...);
...
```

```
$ gcc example.c
example.c:15:5: warning: implicit declaration of function 'malloc'
```
Pre-processor Inclusion Mechanism

Original include.c:

```c
/* Comments before */
#include "header.h"
/* Comments after */
```

header.h:

```c
// From `header.h`
```

After pre-processing include.c:

```c
/* Comments before */
// From `header.h`
/* Comments after */
```
example.c:
```c
#include <stdlib.h>
...
void *ptr = malloc(...);
...
```

stdlib.h:
```c
... 
void *malloc(size_t len);
...
```

After pre-processing example.c:
```c
void *malloc(size_t len);
...
void *ptr = malloc(...);
...
```
prog3a.c
#include <stdio.h>

extern int flag;

void do_hello();

int main(int argc, char *argv[]) {
    printf("main flag=%d\n", flag);
    flag = 2;
    
    do_hello();
    
    printf("main flag=%d\n", flag);
    return 0;
}

prog3b.c
#include <stdio.h>

int flag = 1;

void do_hello() {
    printf("hello flag=%d\n", flag);
    flag = 3;
}

From Assignment 2...
heap209.h:

```c
typedef struct _Chunk Chunk;

extern void *heap_region;
extern Chunk *free_list;
extern Chunk *alloc_list;

void *malloc209(size_t nbytes);
int free209(void *addr);
void heap209_init(size_t heap_size);
void heap209_cleanup(void);
```

Forward declarations of function prototypes and extern global variables
heap209.c:

```c
#include "heap209.h"

void *heap_region = NULL;
Chunk *free_list = NULL;
Chunk *alloc_list = NULL;

void *malloc209(size_t nbytes)
{
    ...
}
```

heap209.h:

```c
typedef struct _Chunk Chunk;

extern void *heap_region;
extern Chunk *free_list;
extern Chunk *alloc_list;

void *malloc209(size_t nbytes);
int free209(void *addr);
void heap209_init(size_t heap_size);
void heap209_cleanup(void);
```
• **heap209.h**
  
  - Forward declarations of function prototypes, `struct` definition and `extern` declarations
  
  - No actual function implementations or global variables defined

• **heap209.c**
  
  - *Implementations* of all the forward declarations in **heap209.h**

• **diagnostics.h**
  
  - Forward declarations of function prototypes for heap debugging functions

• **diagnostics.c**
  
  - *Implementations* of heap debugging functions defined in **diagnostics.h**
  
  - Uses the `struct` definition and `extern` global variables defined in **heap209.h**

• **test-basic1.c**
  
  - Uses the functions defined in **heap209.h** and **diagnostics.h**
What *should* go into header files?

- Function prototypes
- Type definitions: `struct`, `union` and `typedef`
- `extern` global variables
- *Don’t declare actual global variables*
```c
#include "badheader.h"

int main()
{
    x = 1;
    return 0;
}

#include "badheader.h"

void file2_utility()
{
    x = 2;
}
```

After pre-processing:

```c
int x = 209;

int main()
{
    x = 1;
    return 0;
}

void file2_utility()
{
    x = 2;
}
```
wolf:~$ gcc -Wall -g file1.c file2.c -o prog
/tmp/ccdpcL9T.o:(.data+0x0): **multiple definition of `x'
/tmp/ccrZRB1f.o:(.data+0x0): first defined here
collect2: ld returned 1 exit status
Protecting Headers from Multiple Inclusion

- Sometimes the same header file can be included multiple times inadvertently:
  
  - `prog.c` includes `foo.h`
  
  - `prog.c` also includes `bar.h`
  
  - `bar.h` includes `foo.h`
  
  - `prog.c` will see the contents of `foo.h` twice!
```
#include "redefine.h"
#include "redefine.h"

int main()
{
    return 0;
}
```

```
struct S {
    int x;
};
```

In file included from redefine.c:2:
./redefine.h:1:8: **error: redefinition of 'S'
    struct S {
        ^
./redefine.h:1:8: note: previous definition is here
    struct S {
        ^
1 error generated.
```
redefine.h:

```c
#ifndef REDEFINE_H
#define REDEFINE_H

struct S {
    int x;
};
#endif
```

- On first inclusion:
  - Is REDEFINE_H a currently defined preprocessor symbol? **No** (ifndef directive is true)
  - Define a preprocessor symbol REDEFINE_H
  - Emit body of header file (the `struct` definition)
- On second, third, fourth, etc... inclusion:
  - Is REDEFINE_H a currently defined preprocessor symbol? **Yes** (ifndef directive is false)
  - Skip to `endif` directive
Summary

• Put common definitions into *.h (header) files

• Protect your header files from multiple inclusion

• For each function prototype or external global variable, there should be some *.c source code file that provides the actual definition/declaration
Summary

• This is one of the core tools for modularity and code reuse when programming in C
Makefiles
Makefiles

- Originally designed to support *separate compilation* of C files
Compiling test-basic1 by hand

```
wolf:~$ gcc -Wall -g -c heap209.c -o heap209.o
wolf:~$ gcc -Wall -g -c diagnostics.c -o diagnostics.o
wolf:~$ gcc -Wall -g -c test-basic1.c -o test-basic1.o
wolf:~$ gcc test-basic1.o heap209.o diagnostics.o \ 
    -o test-basic1
```
Anatomy of a Makefile

A Makefile contains 1 or more rules.
Each rule has one target, and 1 or more prerequisites.
Each rule may have 0 or more actions (one per line).
Running `make(1)`

- `$ make`
  - With no options looks for a file called `Makefile`, and evaluates the `first` rule
- `$ make test-basic2`
  - Looks for a file called `Makefile` and looks for a rule with the target `test-basic2` and evaluates it
- `$ make -f foo ...`
  - Looks for a Makefile with the name `foo`
How it works

• Make looks at the when the target and its prerequisites were last modified
  – It assumes targets are files and checks the dates of the files
• Make does nothing…
  – If the target exists, and
  – Is more recent than all its prerequisites
• Make executes the actions…
  – If the target doesn’t exist, or
  – If any prerequisite is more recent than the target
Variables — User defined

Define common parts of action commands that you are likely to repeat multiple places:

CFLAGS= -Wall -g

prog : prog.c
   gcc $(CFLAGS) prog.c -o prog

prog2 : prog2.c
   gcc ${CFLAGS} prog.c -o prog
Variables — Built-ins

Make defines variables to represent parts of rules:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$@$</td>
<td>Target</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>First prerequisite</td>
</tr>
<tr>
<td>$?</td>
<td>All out of date prerequisites</td>
</tr>
<tr>
<td>$^$</td>
<td>All prerequisites</td>
</tr>
</tbody>
</table>

```
CFLAGS= -Wall -g
prog : main.c util.c
    gcc ${CFLAGS} $^ -o $@
```
CFLAGS= -Wall -g

prog : main.o util.o
  gcc $^ -o $@

main.o : main.c
  gcc ${CFLAGS} -c $^ -o $@

util.o : util.c
  gcc ${CFLAGS} -c $^ -o $@
CFLAGS= -Wall -g
prog : main.o util.o
  gcc $^ -o $@

%.o : %.c
  gcc ${CFLAGS} -c $< -o $@
A2 Makefile

CC=gcc  
CFLAGS=-Wall -g  
LDFLAGS=  
OBJS=heap209.o diagnostics.o

dependencia: test-basic1

test-basic1: test-basic1.o $(OBJBS)  
 $(CC) $(LDFLAGS) $^ -o $@

%.o: %.c heap209.h  
 $(CC) $(CFLAGS) -c $< -o $@

clean:  
 rm -f test-basic1 *.o
Makefile Summary

• They provide a higher level of abstraction than writing out shell commands in a script file

• They simplify the process of building larger projects
Midterm

- Shell usage
- C language:
  - Syntax
  - Data types (including structures and unions)
    - Pointers and Memory
- File I/O using streams
- C-style strings
- Makefiles
Midterm

• Be aware of the differences in this course and the courses that previous midterms reflect

• Our midterm date falls in a different week

• We have not yet covered shell *programming*, just emphasized *usage*
• **Labs:** Everyone attending show go to either **BA2210** or **BA2220** (they are conjoined)

• **Extra Office Hours:**
  
  • Friday, June 19, 1-3pm in BA3289
  
  • Monday, June 22, 1-3pm in BA3201

• **Midterm:** Tuesday, June 23 from 7-8pm in UC 266

• **Assignment 2:** Due on July 1, 2015