void swap_generic(void *arr, int index_x, int index_y, int width)
{
    char tmp[width];
    void *x_loc = (char *)arr + index_x * width;
    void *y_loc = (char *)arr + index_y * width;

    memmove(tmp, x_loc, width);
    memmove(x_loc, y_loc, width);
    memmove(y_loc, tmp, width);
}
Today's Topics

- Logistics
  - Midterm next Thursday, 6-8pm, Hewlett 200! If you have OAE accommodations and I haven't emailed you, I don't know about them, so please email me!
  - If you need to take an alternate exam, please email me with the subject line, "alternate exam cs107" and tell me why you need a different exam time. Options for the alternate exam are earlier on Thursday, or Friday morning.
  - Go to next week's lab! The material will be on the midterm exam, and you will get good practice! Thursday labs: if you want to do the lab at a different time, that is fine (or on your own).
  - Office hours: yes, we do want to move some to Thur/Fri. I will have them today from 5-6:30pm in Gates 191.
- Reading: Reader: Ch 8, Pointers, Generic functions with void *, and Pointers to Functions
- Lab: \texttt{winky}, \texttt{change\_char}, and \texttt{change\_ptr}
- More on Generic pointers, void *
  - The standard library's \texttt{qsort}
  - A generic stack
There is a malloc typo in the mytail.c starter code (thanks to Trey C. for finding the bug!). The following line:

```c
char **lines = use_stack ? stack_arr : malloc(n));
```

should be:

```c
char **lines = use_stack ? stack_arr : malloc(n * sizeof(char *));
```
I've seen a few students who have been frustrated with stepping through functions in gdb. Sometimes, they will accidentally step into a function like `strlen` or `printf` and get stuck.

There are three important gdb commands about stepping through a program:

**step** (abbreviation: s) : executes the next line and goes into function calls.

**next** (abbreviation: n) : executes the next line, and does not go into function calls. I.e., if you want to run a line with `strlen` or `printf` but don't want to attempt to go into that function, use **next**.

**finish** (abbreviation: fin) : completes a function and returns to the calling function. This is the command you want if you accidentally go into a function like `strlen` or `printf`! This continues the program until the end of the function, putting you back into the calling function.
$ gdb print_arr
The target architecture is assumed to be i386:x86-64
Reading symbols from print_arr...done.
(gdb) b 35
Breakpoint 1 at 0x400700: file print_arr.c, line 35.
(gdb) run
Starting program: /afs/ir/class/cs107/samples/lect8/print_arr
Breakpoint 1, main (argc=1, argv=0x7fffffffea38) at print_arr.c:35
35 print_array(i_array,i_nelems,sizeof(i_array[0]),print_int);
(gdb) n
0, 1, 2, 3, 4, 5
36 print_array(l_array,l_nelems,sizeof(l_array[0]),print_long);
(gdb) s
print_long (arr=0x7fffffffffe910) at print_arr.c:23
23 long l = *(long *)arr;
(gdb) s
24 printf("%ld",l);
(gdb) s
__printf (format=0x4007dc "%ld") at printf.c:28
28 printf.c: No such file or directory.
(gdb) s
32 in printf.c
(gdb) n
33 in printf.c
(gdb) n
32 in printf.c
(gdb) finish
Run till exit from #0 __printf (format=0x4007dc "%ld") at printf.c:32
print_long (arr=0x7fffffffffe910) at print_arr.c:25
Value returned is $1 = 1
(gdb) where
#0 print_long (arr=0x7fffffffffe910) at print_arr.c:25
#1 0x000000000004005d8 in print_array (arr=0x7fffffffffe910, nelems=6, width=8, pr_func=0x400648 <print_long>) at print_arr.c:10
#2 0x00000000000400734 in main (argc=1, argv=0x7fffffffffe910) at print_arr.c:36

(gdb) s
print_long (arr=0x7fffffffffe910) at print_arr.c:23
23 long l = *(long *)arr;
(gdb) s
24 printf("%ld",l);
(gdb) s
__printf (format=0x4007dc "%ld") at printf.c:28
28 printf.c: No such file or directory.
(gdb) s
32 in printf.c
(gdb) n
33 in printf.c
(gdb) n
32 in printf.c
(gdb) finish
Run till exit from #0 __printf (format=0x4007dc "%ld") at printf.c:32
print_long (arr=0x7fffffffffe910) at print_arr.c:25
Value returned is $1 = 1
(gdb) where
#0 print_long (arr=0x7fffffffffe910) at print_arr.c:25
#1 0x000000000004005d8 in print_array (arr=0x7fffffffffe910, nelems=6, width=8, pr_func=0x400648 <print_long>) at print_arr.c:10
#2 0x00000000000400734 in main (argc=1, argv=0x7fffffffffe910) at print_arr.c:36

(gdb) s
print_long (arr=0x7fffffffffe910) at print_arr.c:23
23 long l = *(long *)arr;
(gdb) s
24 printf("%ld",l);
(gdb) s
__printf (format=0x4007dc "%ld") at printf.c:28
28 printf.c: No such file or directory.
(gdb) s
32 in printf.c
(gdb) n
33 in printf.c
(gdb) n
32 in printf.c
(gdb) finish
Run till exit from #0 __printf (format=0x4007dc "%ld") at printf.c:32
print_long (arr=0x7fffffffffe910) at print_arr.c:25
Value returned is $1 = 1
(gdb) where
#0 print_long (arr=0x7fffffffffe910) at print_arr.c:25
#1 0x000000000004005d8 in print_array (arr=0x7fffffffffe910, nelems=6, width=8, pr_func=0x400648 <print_long>) at print_arr.c:10
#2 0x00000000000400734 in main (argc=1, argv=0x7fffffffffe910) at print_arr.c:36
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

This is critical to understand what is happening here!

I strongly suggest drawing a picture of the situation. Make up addresses, but write everything down.

Let's go through it together.
In lab this week, I noticed a lot of students had trouble really understanding the *winky* example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");
    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

*word* is an array at address:

At this point, *pw* is a *pointer* at address:

The value of *pw* is:

\[ \text{null} \]
In lab this week, I noticed a lot of students had trouble really understanding the winky example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- `word` is an array at address: 0x128
- `pw` is a pointer at address: 0x110

The value of `pw` is: ollleh
In lab this week, I noticed a lot of students had trouble really understanding the winky example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- `word` is an array at address: **0x128**
- `pw` is a pointer at address: **0x110**

The value of `pw` is: **hello**
In lab this week, I noticed a lot of students had trouble really understanding the winky example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- `word` is an array at address: **0x128**
- `pw` is a pointer at address: **0x110**
- The value of `pw` is: **0x128**
In lab this week, I noticed a lot of students had trouble really understanding the winky example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");
    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:
- `word` is an array at address: 0x128
- `pw` is a pointer at address: 0x110
- The value of `pw` is: 0x128

What value will be passed into `change_char`?
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");
    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:
- `word` is an array at address: **0x128**
- `pw` is a pointer at address: **0x110**
- The value of `pw` is: **0x128**

What value will be passed into `change_char`? **0x128**
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland"
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford"
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- `s` is a `pointer` at address:
  - `s` holds the `value` of:
    - `hello`
In lab this week, I noticed a lot of students had trouble really understanding the **winky** example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- **s** is a *pointer* at address: **0x100**
- **s** holds the value of: `"hello"`
In lab this week, I noticed a lot of students had trouble really understanding the \texttt{winky} example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- \texttt{s} is a \texttt{pointer} at address: \texttt{0x100}
- \texttt{s} holds the \texttt{value} of: \texttt{0x128}

In this function, \texttt{s} points to the 'b' at address \texttt{0x128}.
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");
    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- `s` is a pointer at address: **0x100**
- `s` holds the value of: **0x128**

In this function, `s` points to the 'b' at address `0x128`.

- `*s = 'j'` dereferences `s`, and modifies the 'h' to become 'j'.

---

**Address**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x128</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0xf898</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0xf891</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x128</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0xf881</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");
    change_char(pw); // can you substitute word for pw?
    change_ptr(&pw); // can you substitute &word for &pw?
}
```

At this point:

- `s` is a pointer at address: **0x100**
- `s` holds the value of: **0x128**

In this function, `s` points to the 'b' at address 0x128.

`s = "Leyland"` will change what value above?
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");
    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

At this point:

- `s` is a pointer at address: **0x100**
- `s` holds the value of: **0x128**

In this function, `s` points to the 'b' at address 0x128.

`s = "Leyland"` will change what value above? **the value of s will change!**
In lab this week, I noticed a lot of students had trouble really understanding the \texttt{winky} example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

When the function ends, \texttt{s} goes away!
In lab this week, I noticed a lot of students had trouble really understanding the winky example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

What value gets passed into change_ptr?
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s) {
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str) {
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void) {
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

What value gets passed into `change_ptr`?

The address of `pw`!
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland"
};

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford"
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

**p_str = 'm';** double-dereferences all the way to 0x128, and updates 'j' to be 'm'.

What value gets passed into `change_ptr`? **0x110**

The address of `pw`!
In lab this week, I noticed a lot of students had trouble really understanding the winky example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

*p_str = "Stanford"; dereferences to 0x110 and changes pw's value to point to "Stanford"

What value gets passed into change_ptr? 0x110

The address of pw!
In lab this week, I noticed a lot of students had trouble really understanding the `winky` example:

```c
void change_char(char *s)
{
    *s = 'j';
    s = "Leland";
}

void change_ptr(char **p_str)
{
    **p_str = 'm';
    *p_str = "Stanford";
    p_str = NULL;
}

void winky(void)
{
    char word[6];
    char *pw = word;
    strcpy(word, "hello");

    change_char(pw);    // can you substitute word for pw?
    change_ptr(&pw);    // can you substitute &word for &pw?
}
```

What value gets passed into `change_ptr`? **0x110**

The address of `pw`!

`p_str = NULL` sets the value of `p_str` to 0.
Function pointers

From Monday, here is a function pointer example that accepts a function that knows how to print an element from an array.

```c
#include<stdio.h>
#include<stdlib.h>

void print_array(void *arr,size_t nelems,int width,void(*pr_func)(void *))
{
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        pr_func(element);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
}

void print_int(void *arr)
{
    int i = *(int *)arr;
    printf("%d",i);
}

void print_long(void *arr)
{
    long l = *(long *)arr;
    printf("%ld",l);
}

int main(int argc, char **argv)
{
    int i_array[] = {0,1,2,3,4,5};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {0,10,20,30,40,50};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    print_array(i_array,i_nelems,sizeof(i_array[0]),print_int);
    print_array(l_array,l_nelems,sizeof(l_array[0]),print_long);
    return 0;
}
```

We read function pointers from the inside-out, so this means "a function called `pr_func` that takes a `void *` argument and returns `void`."
Function pointers

From Monday, here is a function pointer example that accepts a function that knows how to print an element from an array.

```c
#include<stdio.h>
#include<stdlib.h>

void print_array(void *arr,size_t nelems,int width,void(*pr_func)(void *)) {
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        pr_func(element);
        i == nelems - 1 ? printf("
") : printf(", ");
    }
}

void print_int(void *arr) {
    int i = *(int *)arr;
    printf("%d",i);
}

void print_long(void *arr) {
    long l = *(long *)arr;
    printf("%ld",l);
}

int main(int argc, char **argv) {
    int i_array[] = {0,1,2,3,4,5};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {0,10,20,30,40,50};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    print_array(i_array,i_nelems,sizeof(i_array[0]),print_int);
    print_array(l_array,l_nelems,sizeof(l_array[0]),print_long);
    return 0;
}
```

When we need to process the element, we call the helper function with the argument.
Function pointers

From Monday, here is a function pointer example that accepts a function that knows how to print an element from an array.

```c
#include<stdio.h>
#include<stdlib.h>

void print_array(void *arr,size_t nelems,int width,void(*pr_func)(void *)) {
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        pr_func(element);
        i == nelems - 1 ? printf("\n") : printf(" , ");
    }
}

void print_int(void *arr) {
    int i = *(int *)arr;
    printf("%d",i);
}

void print_long(void *arr) {
    long l = *(long *)arr;
    printf("%ld",l);
}

int main(int argc, char **argv) {
    int i_array[] = {0,1,2,3,4,5};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {0,10,20,30,40,50};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    print_array(i_array,i_nelems,sizeof(i_array[0]),print_int);
    print_array(l_array,l_nelems,sizeof(l_array[0]),print_long);
    return 0;
}
```

We create a function that meets the requirement of the generic function. In this case, a function that has a `void *` pointer and returns `void`.
Function pointers

From Monday, here is a function pointer example that accepts a function that knows how to print an element from an array.

```c
#include<stdio.h>
#include<stdlib.h>

void print_array(void *arr,size_t nelems,int width,void(*pr_func)(void *))
{
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        pr_func(element);
        i == nelems - 1 ? printf("\n") : printf(" , ");
    }
}

void print_int(void *arr)
{
    int i = *(int *)arr;
    printf("%d",i);
}

void print_long(void *arr)
{
    long l = *(long *)arr;
    printf("%ld",l);
}

int main(int argc, char **argv)
{
    int i_array[] = {0,1,2,3,4,5};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {0,10,20,30,40,50};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    print_array(i_array,i_nelems,sizeof(i_array[0]),print_int);
    print_array(l_array,l_nelems,sizeof(l_array[0]),print_long);

    return 0;
}
```

This function is built specifically for `int *`, so it knows that it can cast `arr` to `int *`. 
C standard library example: `qsort`

The C standard library has a number of functions that expect function pointers. The `qsort` function is one of them:

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

The `base`, `nmemb`, and `size` variables are just standard pointer-to-array details. The `compar` function is a comparison function that expects two elements from the array, and will perform a comparison on them. This is a standard comparison with the following return `int` value possibilities:

- **negative**: the first element is less than the second element
- **zero**: the elements are equal
- **positive**: the first element is greater than the second element
The C standard library has a number of functions that expect function pointers. The `qsort` function is one of them:

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

If you want to use the `qsort` function, you need to write a `compar` function yourself. Sometimes, we just need to build a function that utilizes another built-in function, like `strcmp`, to do the work:

```c
int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}
```
The C standard library has a number of functions that expect function pointers. The `qsort` function is one of them:

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

If you want to use the `qsort` function, you need to write a `compar` function yourself. Sometimes, we just need to build a function that utilizes another built-in function, like `strcmp`, to do the work:

```c
int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}
```

Important! Look at the type of `s1` and `s2` in the comparison function! This is a case where we must draw the situation!
C standard library example: \texttt{qsort} full example

```c
// file: qsort_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}

int main(int argc, char **argv)
{
    // ignore program name
    argc--;  
    argv++;  

    qsort(argv,argc,sizeof(argv[0]),compar_str);
    for (int i=0; i < argc; i++) {
        printf("%s",argv[i]);
        i == argc - 1 ? printf("\n") : printf("", ");
    }

    return 0;
}
```

$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck

At this point in the program, this is what the situation looks like.
C standard library example: \texttt{qsort} full example

\begin{verbatim}
// file: qsort_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}

int main(int argc, char **argv) {
    // ignore program name
    argc--;
    argv++;
    qsort(argv,argc,sizeof(argv[0]),compar_str);
    for (int i=0; i < argc; i++) {
        printf("%s",argv[i]);
        i == argc - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
\end{verbatim}

We have updated \texttt{argc} and \texttt{argv} to ignore the program name.

$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck
C standard library example: qsort full example

// file: qsort_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}

int main(int argc, char **argv) {
    // ignore program name
    argc--;
    argv++;
    qsort(argv, argc, sizeof(argv[0]), compar_str);
    for (int i = 0; i < argc; i++) {
        printf("%s", argv[i]);
        i == argc - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}

$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck

Based on the diagram above, what number gets passed as the first argument of qsort?
C standard library example: qsort full example

// file: qsort_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}

int main(int argc, char **argv)
{
    // ignore program name
    argc--;
    argv++;
    qsort(argv, argc, sizeof(argv[0]), compar_str);
    for (int i=0; i < argc; i++) {
        printf("%s", argv[i]);
        i == argc - 1 ? printf("\n") : printf(" , ");
    }
    return 0;
}

$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck

Based on the diagram above, what number gets passed as the first argument of qsort? **0x108**
C standard library example: `qsort` full example

```c
// file: qsort_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}

int main(int argc, char **argv)
{
    // ignore program name
    argc--;               
    argv++;               
    qsort(argv,argc,sizeof(argv[0]),compar_str);
    for (int i=0; i < argc; i++) {
        printf("%s",argv[i]);
        i == argc - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

```
$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck
```

`qsort` has no way to dereference `argv`, so it can only pass char ** pointers to sort (e.g., 0x108, 0x110)
C standard library example: `qsort` full example

```c
// file: qsort_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}

int main(int argc, char **argv) {
    // ignore program name
    argc--;
    argv++;

    qsort(argv, argc, sizeof(argv[0]), compar_str);
    for (int i=0; i < argc; i++) {
        printf("%s", argv[i]);
        i == argc - 1 ? printf("\n") : printf("", "");
    }

    return 0;
}
```

Therefore, the type that gets passed to `compar_str` must be char ** pointers. (e.g., 0x108, 0x110)

$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck
C standard library example: qsort full example

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int compar_str(const void *s1, const void *s2) {
    return strcmp(*(char **)s1, *(char **)s2);
}

int main(int argc, char **argv) {
    // ignore program name
    argc--; 
    argv++;

    qsort(argv,argc,sizeof(argv[0]),compar_str);
    for (int i=0; i < argc; i++) {
        printf("%s",argv[i]);
        i == argc - 1 ? printf("\n") : printf(", ");
    }

    return 0;
}
```

$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck

So, we are correct to cast s1 and s2 to char **, and then dereference to get char * to pass to strcmp.
Function pointers

Function pointer takeaways:

1. Function pointers allow us to add generic features to our functions, so that even if the function doesn't know what the underlying type of a `void *` is, it can still do something useful with the data.
2. Function pointers have some strange syntax, and you read from "inside out"
3. The calling function passes in a function that knows how to deal with the correct type for the elements in the array.
4. We have a different function for each type we want to pass as a function pointer, and each function can cast its inputs correctly, because they are specific to that type.
Let's build a generic stack. We are going to be using structs extensively for this example, and they are fair game for the midterm exam. So, make sure you understand this example!

First, let's remind ourselves what the stack data structure does (back to CS 106B!):
1. A stack is a last-in-first-out data structure that can store elements. The first element in the stack is the last element out of the stack.
2. The push operation adds an element onto the stack
3. The pop operation removes an element from the stack.

Note, we are not talking about the program stack, but a generic version of the stack abstract data type!
Example: Building a generic stack

Let's build a generic stack. We are going to be using structs extensively for this example, and they are fair game for the midterm exam. So, make sure you understand this example!

We'll start by defining a node that will hold a pointer to a "next" node, and some data:

```c
typedef struct node {
    struct node *next;
    void *data;
} node;
```

A note on syntax: We are defining a type here (thus, `typedef`), and we are defining a node to be a "struct node". This is different from C++, where we can just define a struct and use its name. In C, without the `typedef`, we would constantly have to be referring to "struct node" every time we wanted to use it. Not fun!
Let's build a generic stack. We are going to be using structs extensively for this example, and they are fair game for the midterm exam. So, make sure you understand this example!

We'll start by defining a node that will hold a pointer to a "next" node, and some data:

```c
typedef struct node {
    struct node *next;
    void *data;
} node;
```

We don't know anything about the type of thing that `data` will point to, although the stack itself will know its width.
Next, let's build the **stack** type. It will have a defined width for each node, and it will also keep track of how many elements it holds. It will also keep track of the top of the stack. Again, we want to typedef it so we don't have to continually say "struct stack" when we want to use it.

```c
typedef struct stack {
    int width;
    int nelems;
    node *top;
} stack;
```

Remember, a node is generic, so this stack can hold any time, although once it has a width defined, all elements you push must have that width.
Example: Building a generic stack

How do we create a default stack? We could do it manually:

```c
stack s1;
s1.width = sizeof(int); // store ints
s1.nelems = 0;
s1.top = NULL;
```

But let's create a function for it, in which case we should use a pointer:

```c
stack *s = stack_create(...);
```
Example: Building a generic stack

Our stack creation function:

```c
stack *stack_create(int width) {
    stack *s = malloc(sizeof(stack));
    s->width = width;
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

Let's investigate...
Example: Building a generic stack

Our stack creation function:

```c
stack *stack_create(int width) {
    stack *s = malloc(sizeof(stack));
    s->width = width;
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

A particular stack must have a set width (otherwise, we would have to pass in the width each time, and this doesn't make sense for `pop` -- we wouldn't know what type we were popping off!)
Example: Building a generic stack

Our stack creation function:

```c
stack *stack_create(int width)
{
    stack *s = malloc(sizeof(stack));
    s->width = width;
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

Get enough memory from the heap to create the stack.
Example: Building a generic stack

Our stack creation function:

```c
stack *stack_create(int width) {
    stack *s = malloc(sizeof(stack));
    s->width = width;
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

Set the initial conditions.
Example: Building a generic stack

Our stack creation function:

```c
stack *stack_create(int width)
{
    stack *s = malloc(sizeof(stack));
    s->width = width;
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

Return the pointer to the memory we just requested and initialized.
Example: Building a generic stack

Let's look at our `push` function:

```c
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->width);
    memcpy(new_node->data, data, s->width);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```
Let's look at our `push` function:

```c
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->width);
    memcpy(new_node->data, data, s->width);
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

The stack function takes a stack as a parameter! The stack isn't an object, and it doesn't have functions built in. If we really wanted to, we could create a stack struct that has function pointers, but that is more advanced. A pointer to the data is also required.
Let's look at our `push` function:

```c
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->width);
    memcpy(new_node->data, data, s->width);
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Each time we add an element to the stack, we need to create a `node`, and we get that off the heap, too.
Let's look at our `push` function:

```c
void stack_push(stack *s, const void *data)
{
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->width);
    memcpy(new_node->data, data, s->width);
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Guess what? We also have to use heap memory to store the data! We are making a copy of the data, not just pointing to it!
Example: Building a generic stack

Let's look at our `push` function:

```c
void stack_push(stack *s, const void *data)
{
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->width);
    memcpy(new_node->data, data, s->width);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

We copy the data pointed to into our node. This could be anything, but we know the width. If it is a pointer, we'll copy the pointer, but it could be integer data, or any other kind of data.
Example: Building a generic stack

Let's look at our `push` function:

```c
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->width);
    memcpy(new_node->data, data, s->width);
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

We have to do some wiring here (kind of like linked lists). We are inserting this node before the top of the stack.
Example: Building a generic stack

Let's look at our `push` function:

```c
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->width);
    memcpy(new_node->data, data, s->width);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Don't forget to update the number of elements.
Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr,n->data,s->width);
    // rewire
    s->top = n->next;
    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```
Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr,n->data,s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

Let's return a boolean value to say whether or not we had an element to return. In other words, if the stack is empty, return `false`; otherwise, return `true`. 
Example: Building a generic stack

Let's look at our \texttt{pop} function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

\begin{verbatim}
bool stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
\end{verbatim}

Again, \texttt{pop} has a stack argument, and a pointer to a memory location to hold the data we are going to copy.
Example: Building a generic stack

Let's look at our pop function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

Check to see if the stack is empty.
Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }

    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

Might as well create a temporary pointer so we don't have to do a bunch of double "->" references.
Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr,n->data,s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

We'll copy the data back to the memory location we were provided.
Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr,n->data,s->width);
    // rewire
    s->top = n->next;
    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

Re-wiring is pretty easy -- the top is now just the next element in the stack.
Let’s look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr,n->data,s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

We have to clean up. First, we free the data (remember, we malloc’d it originally!)
Example: Building a generic stack

Let's look at our **pop** function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n); // free the node itself (because we malloc'd it!)
    s->nelems--;
    return true;
}
```
Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;
    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

Don't forget to decrement the number of elements!
Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```c
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
    return false;
    }
    node *n = s->top;
    memcpy(addr,n->data,s->width);
    // rewire
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
    return true;
}
```

We did have an element to return, so we return `true`. 

Example: Building a generic stack

Now we can try it. Let's push on an array of ints, and then pop them all off:

```c
int main(int argc, char **argv)
{
    // start with an int array
    int iarr[] = {0,2,4,6,8,12345678,24680};
    int nelems = sizeof(iarr) / sizeof(iarr[0]);

    stack *intstack = stack_create(sizeof(iarr[0]));
    for (int i=0; i < nelems; i++) {
        stack_push(intstack, iarr + i);
    }

    int popped_int;
    while (stack_pop(intstack,&popped_int)) {
        printf("%d\n",popped_int);
    }
    free(s); // clean up!
    return 0;
}
```

What is the size of each element?
Now we can try it. Let's push on an array of `int`s, and then pop them all off:

```c
int main(int argc, char **argv)
{
    // start with an int array
    int iarr[] = {0,2,4,6,8,12345678,24680};
    int nelems = sizeof(iarr) / sizeof(iarr[0]);

    stack *intstack = stack_create(sizeof(iarr[0]));
    for (int i=0; i < nelems; i++) {
        stack_push(intstack, iarr + i);
    }

    int popped_int;
    while (stack_pop(intstack,&popped_int)) {
        printf("%d\n",popped_int);
    }
    free(s); // clean up!
    return 0;
}
```

What is the size of each element?

4

(because we will be storing `int`s in the stack)
Now we can try it. Let's push on an array of ints, and then pop them all off:

```c
int main(int argc, char **argv)
{
    // start with an int array
    int iarr[] = {0,2,4,6,8,12345678,24680};
    int nelems = sizeof(iarr) / sizeof(iarr[0]);

    stack *intstack = stack_create(sizeof(iarr[0]));
    for (int i=0; i < nelems; i++) {
        stack_push(intstack, iarr + i);
    }

    int popped_int;
    while (stack_pop(intstack,&popped_int)) {
        printf("%d\n",popped_int);
    }
    free(s); // clean up!
    return 0;
}
```

$ ./stack
24680
12345678
8
6
4
2
0
7
Example: Building a generic stack

Let's try and push one more int onto the stack (assume we do this before the call to free:

```c
int main(int argc, char **argv)
{
    ...
    int x = 42;
    stack_push(intstack, x);
}
```

Does this work? Recall:

```c
void stack_push(stack *s, const void *data)
```
Example: Building a generic stack

Let's try and push one more int onto the stack (assume we do this before the call to free:

```c
int main(int argc, char **argv) {
    ... 
    int x = 42;
    stack_push(intstack, x);
}
```

Does this work? Recall:

```c
void stack_push(stack *s, const void *data)
```

This does not work -- we need a pointer to x. So, we should do:

```c
stack_push(intstack, &x);
```
Example: Building a generic stack

Let's go ahead and use an array of `char *` pointers -- remember, our stack is generic, and will work for any pointer! Let's push all the command line args onto the stack:

```c
stack *s = stack_create(sizeof(argv[0]));
for (int i=1; i < argc; i++) {
    stack_push(s, argv+i);
}
char *next_arg;
while (stack_pop(s, &next_arg)) {
    printf("%s\n", next_arg);
}
```

What is the size of each element?

We're pushing on all but the program name.
Example: Building a generic stack

Let's go ahead and use an array of `char *` pointers -- remember, our stack is generic, and will work for any pointer! Let's push all the command line args onto the stack:

```c
stack *s = stack_create(sizeof(argv[0]));
for (int i=1; i < argc; i++) {
    stack_push(s,argv+i);
}

char *next_arg;
while (stack_pop(s,&next_arg)) {
    printf("%s
",next_arg);
}
```

We're pushing on all but the program name.

What is the size of each element?

8

because the size of a `char *` pointer is 8.
Let's go ahead and use an array of char * pointers -- remember, our stack is generic, and will work for any pointer! Let's push all the command line args onto the stack:

```c
stack *s = stack_create(sizeof(argv[0]));
for (int i=1; i < argc; i++) {
    stack_push(s,argv+i);
}
char *next_arg;
while (stack_pop(s,&next_arg)) {
    printf("%s\n",next_arg);
}
```

We're pushing on all but the program name.
Example: Building a generic stack

Can we push on one more string?

```c
...  
string *h = "hello";
stack_push(s,h);
```

This should work, right? `h` is a pointer! Recall:

```c
void stack_push(stack *s, const void *data)
```
Can we push on one more string?

```c
... string *h = "hello";
stack_push(s,h);
```

This should work, right? h is a pointer! Recall:

```c
void stack_push(stack *s, const void *data)
```

**This doesn't work!** We need a pointer to the memory we are pushing onto the stack. We aren't pushing string characters, we are pushing a string pointer! So, we need:

```c
stack_push(s,&h); // &h is a char **
```
References and Advanced Reading

• **References:**
  • K&R C Programming (from our course)
  • Course Reader, C Primer
  • Awesome C book: http://books.goalkicker.com/CBook
  • Function Pointer tutorial: https://www.cprogramming.com/tutorial/function-pointers.html

• **Advanced Reading:**
  • virtual memory: https://en.wikipedia.org/wiki/Virtual_memory