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
  • Assign3 - Due on **Sunday**
  • Midterm on Thursday
• Reading: Reader: Ch 8, Pointers, Generic functions with void *, and Pointers to Functions

• Generic pointers, void *
  • Why we use them
  • How to use them
  • Examples
Function pointers

From last time: here is our full example:

```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]);
    print_array(i_array, i_nelems,sizeof(i_array[0]), print_int);

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

    return 0;
}
```
Function pointers

Here is our full example:

```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;
}
```

Note that when you pass a function pointer, you don't need to use "&" because it is implied (though you can if you want).
Function pointers

For our print_array function, we can have the printing do anything we want!

Let's look at the `printf_coord.c` file from `/afs/ir/class/cs107/lecture-code/lect13`

Also available in the course reader: [http://stanford.edu/~cgregg/107-Reader/107-Reader-code.zip](http://stanford.edu/~cgregg/107-Reader/107-Reader-code.zip)

Look in `code/Ch8_C_Low_Level`
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);
}
```
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 *));
```

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: 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;
}
```

At this point in the program, this is what the situation looks like.

```
$ ./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

We have updated argc and argv to ignore the program name.
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;
}
```

Based on the diagram above, what number gets passed as the first argument of `qsort`?
$ ./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;
}
```

$q$sort 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
#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`)
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;
}
```

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

```
$ ./qsort_ex dog cat ant duck bear
ant bear cat dog duck
```
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. The calling function passes in a function that knows how to deal with the correct type for the elements in the array.

3. Function pointers have some strange syntax, and you read from "inside out"
Midterm Thoughts

• Before you get to the exam, make sure you take a good look at the provided reference sheet.
  • It has been expanded from what students in prior CS107 classes have been given. To wit:
    • It describes the functions that you may be expected to use on the exam. You will not need any functions not on the reference sheet to successfully answer the questions.
    • It has a list of integer constants
    • It lists the decimal, hex, and binary representations for the values 0-15.
• There are four problems on the exam. Two questions involve bitwise operations, and two questions involve strings, arrays, and casting.
• You will have to write a total of four functions, and you will have to be able to read and understand a fifth function.
Midterm Thoughts (continued)

- There is a lot of text on the exam for you to read. Most of the text is geared towards examples of how the functions we ask you to write work, and details about your implementation.
  - Take the time to understand what we want you to do. Some functions are only a few lines of code, but they are packed lines.
  - Once you understand the examples, that's a good time to start working on your code. We will have scrap paper for you to jot down notes or to plan out your code.
- I strongly suggest that you review assignments 1 and 2. Some of the midterm problems are based closely on code you had to write for those assignments.
- Good luck!
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