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);
}

Reading: Reader: Ch 8, Pointers, Generic functions with void *, and Pointers to Functions, K&R Ch 1.6, 5.6-5.9

Lecturers: Chris Gregg
Today's Topics

• Logistics
  • Assign3 - out tomorrow
  • Gabbi’s office hours are going to be 10am-12pm Monday/Tuesday in the Women's Community Center
• Reading: Reader: Ch 8, Pointers, Generic functions with void *, and Pointers to Functions
• More on heap allocation: contractual guarantees, undefined behavior
• Heap allocation, the good, the bad
• How to choose: stack or heap?
• Generic pointers, void *
  • Why we use them
  • How to use them
  • Examples
I received an anonymous feedback (https://sayat.me/chrisgregg/) that asked, "May I kindly ask if there is any possibility that on Mondays, you give a 2-minute overview of what our net Assignment will be"

Good idea! :)

This week's assignment will include code readings on the `gets` ("get string") function (which is tremendously terrible) and on possible implementations of `calloc` and `realloc`.

You will have to write a function called `read_line` which improves upon `gets` and `fgets` (the better version of `gets`).

You will also write two utilities, `myuniq` and `mytail`, where you will use `read_line` (so make it good!). Let's look at `uniq` and `tail`. 
**char **p_input**

Assign2 has the following definition for `scan_token`:

```c
bool scan_token(const char **p_input, const char *delimiters,
                char buf[], size_t buflen);
```

There have been some excellent questions on Piazza about the `p_input` variable. Why is it a `char **`?

The `scan_token` function populates the buffer with the next token, and then "update[s] the pointer held by `p_input` to point to the next character following the token that was just scanned."

The type of `p_input` is important! Let's look at it in more detail.
Let's see what happens when we call `scan_token`:

```c
scan_token(&remaining, delim, buf, sizeof(buf))
```

This is a diagram of the stack when we are inside of the function that calls `scan_token`.

`remaining` is a `char *` and it points to the string "abc,def"

Assume `delim` and `buf` are elsewhere on the stack.

If we pass in the value of `remaining`, which is `0x7fffffffde97c`, how could we ever update `remaining` to point later in the string? We can't!

We must instead pass the address of `remaining`, which is `0x7fffffffde974`.
Let's see what happens when we call `scan_token`:

```c
scan_token(&remaining, delim, buf, sizeof(buf))
```

Now assume we are inside `scan_token`:

```c
bool scan_token(const char **p_input, const char *delimiters, char buf[], size_t buflen)
```

Because we have the address of `remaining`, we can update `remaining`! The type of `p_input` is a `char **` because it points to a `char *` (which is `remaining`).
Let's see what happens when we call `scan_token`:

```c
scan_token(&remaining, delim, buf, sizeof(buf))
```

Now assume we are inside `scan_token`:

```c
bool scan_token(const char **p_input, const char *delimiters, char buf[], size_t buflen)
```

Because we have the *address* of `remaining`, we can update `remaining`! The type of `p_input` is a `char **` because it points to a `char *` (which is `remaining`).

This is what `remaining` looks like at the end of the function call.
void *malloc(size_t nbytes);
void *calloc(size_t count, size_t size);
void *realloc(void *ptr, size_t nbytes);
void free(void *ptr);

malloc, calloc, and realloc guarantee:
- NULL on failure
- Memory is contiguous; the number of bytes is >= to the requested amount.
- They are not recycled unless you call free
- realloc preserves existing data
- calloc initializes bytes, malloc and realloc do not

Undefined behavior occurs when:
- If overflow (i.e., beyond bytes allocated)
- if use after free, or if free is called twice on a location.
- realloc/free non-heap address
Why do we like heap allocation?

**Plentiful** — you can request lots of heap memory if your program needs it.

**Allocation and deallocation are under the program's control** — you can precisely determine the lifetime of a block of memory.

**Can resize** — you can use `realloc` to resize a block.
Why don't we like heap allocation?

Only moderately efficient — The operating system needs to be involved, and it needs to search for available space, update its records, etc.

Low type safety — You get back a `void *` pointer, and that limits the compiler's ability to provide warnings.

Memory management is tricky — you have to remember to initialize, you have to keep track of how many bytes you've requested, you have to remember to `free` but to not `free` twice, etc.

Leaks possible — this is less critical, but causes programs to waste memory.
How do you choose between stack and heap allocation?

Use stack if possible, go to heap only when you must
Stack is safer, more efficient, more convenient

When is heap allocation required?
Very large allocation that could blow out stack
Dynamic construction, not known at compile-time what declarations will be needed
Need to control lifetime — memory must persist outside of function call
Need to resize memory after initial allocation

With heap, comes responsibility
Your responsibility for correct allocation at right time and right size
Your responsibility to manage the pointee type and size
Your responsibility for correct deallocation at right time, once and only once

valgrind is your friend!
We are now going to go into an area that I like to call "the wild west" of pointers. We are going to discuss the `void *` pointer, which is a pointer that has an unspecified pointee type. In other words, it is a pointer, but does not have a width associated with the underlying data based on some type.

You can pass `void *` pointers to and from functions, and you can assign them values with the `&` operator. E.g.,

```c
int arr[] = {2,4,6,8,10};
void *arr_p1 = arr;
int *arr_p2 = arr_p;
```
You cannot dereference a `void *` pointer, nor can you use pointer arithmetic with it. E.g.,

```c
int arr[] = {2, 4, 6, 8, 10};
void *arr_p1 = arr;
arr_p1++; // gives compiler warning about incrementing void *
printf("%d\n", arr_p1[0]); // warns, but also causes compiler error because you cannot dereference void *
```
Generic Pointers

Why would we ever want a type where we lose information?

Sometimes, a function needs to be generic so it can deal with any type. We have seen this with `realloc` and `free`:

```c
void free(void *ptr);
```

It would not be very nice if we had to have a different `free` function for every type of pointer!
What if you wanted to write a program to swap the first and last element in an `int` array? You might write something like this:

```c
void swap_ends_int(int *arr, size_t nelems)
{
    int tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

int main(int argc, char **argv)
{
    int i_array[] = {10,40,80,20,-30,50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);
    swap_ends_int(i_array,i_nelems);
    return 0;
}
```

Great! But what if you also wanted to swap the first and last element in a `long` array?
Great! But what if you also wanted to swap the first and last element in a `long` array?

```c
void swap_ends_int(int *arr, size_t nelems)
{
    int tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

void swap_ends_long(long *arr, size_t nelems)
{
    long tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

int main(int argc, char **argv)
{
    int i_array[]   = {10,40,80,20,-30,50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);
    swap_ends_int(i_array,i_nelems);

    long l_array[]  = {100,400,800,200,-300,500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);
    swap_ends_long(l_array,l_nelems);
    return 0;
}
```

Bummer. We have to write a function that is virtually identical, with the only difference being that we handle the type of the array elements differently.

In other words, the type system is getting in the way! We would like to write a single `swap_ends` function that handles any array, but the type system foils us.
**Generic Pointers**

void * to the rescue! In this case, the pointer type gives us information about the size of the elements being pointed to (either 4-bytes for int, or 8-bytes for long, in the previous example).

By using void * and explicitly including the width of the type, we can write a function that can take any type as the elements to swap:

```c
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}
```

Remember last time we showed that we can copy bytes using `memcpy`?

We must pass the width of the elements in the array because the void * pointer doesn't carry that information.
void swap_ends(void *arr, size_t nelems, int width) {
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

Let's look at this function in more detail. First, we have a `void *` pointer passed in as the array.
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

Let's look at this function in more detail. First, we have a `void *` pointer passed in as the array.

Next, we create a `char` array to hold the bytes. Remember: `char` is the only 1-byte type we have, and using a `char` array is how we can create an array that is exactly the number of bytes we want. We will use this almost every time we use `void *` pointers, so get used to it!
Let's look at this function in more detail. First, we have a `void *` pointer passed in as the array.

Next, we create a `char` array to hold the bytes. Remember: `char` is the only 1-byte type we have, and using a `char` array is how we can create an array that is exactly the number of bytes we want. We will use this almost every time we use `void *` pointers, so get used to it!

We copy the bytes with `memmove`. 
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

This part takes some time to get used to!

Notice that we need a pointer to the element that we are trying to copy into. We already said that we cannot do pointer arithmetic on a `void *` pointer, so we first cast the pointer to `char *`, and then manually calculate the pointer arithmetic to get us to the correct location. In this case, because we want the last element in the array, the calculation is:

```
(char *)arr + (nelems - 1) * width
```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

In other words, what is the location of the 42?

(char *)arr + (nelems - 1) * width

0x7ffeea3c9484 + (5 * 4) == 0x7ffeea3c9498

A key point to understand is that the pointer arithmetic increases by exactly 20
because of the char * cast, which means that +1 equals 1 byte.
Very often, we will need to find the \( i \)th element in an array. You should be **extremely** familiar with the following idiom:

```c
for (size_t i=0; i < nelems; i++) {
    // get ith element
    void *ith = (char *)arr + i * width;
}
```
Very often, we will need to find the \( i \)th element in an array. You should be **extremely** familiar with the following idiom:

```c
for (size_t i=0; i < nelems; i++) {
    // get ith element
    void *ith = (char *)arr + i * width;
}
```

### Table

<table>
<thead>
<tr>
<th>( i )</th>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(char *)0x7ffeea3c9484 + 0 * 4</td>
<td>0x7ffeea3c9484</td>
</tr>
<tr>
<td>1</td>
<td>(char *)0x7ffeea3c9484 + 1 * 4</td>
<td>0x7ffeea3c9488</td>
</tr>
<tr>
<td>2</td>
<td>(char *)0x7ffeea3c9484 + 2 * 4</td>
<td>0x7ffeea3c948c</td>
</tr>
<tr>
<td>3</td>
<td>(char *)0x7ffeea3c9484 + 3 * 4</td>
<td>0x7ffeea3c9490</td>
</tr>
<tr>
<td>4</td>
<td>(char *)0x7ffeea3c9484 + 4 * 4</td>
<td>0x7ffeea3c9494</td>
</tr>
<tr>
<td>5</td>
<td>(char *)0x7ffeea3c9484 + 5 * 4</td>
<td>0x7ffeea3c9498</td>
</tr>
</tbody>
</table>

Important! These numbers are pointers to the type held in the array!!!!!
Let's walk through this example.

First, we create an `int` array, then we find its size.
Let's walk through this example.

First, we create an `int` array, then we find its size.

Next, we create a long array, then we find its size.
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp,arr,width);

    // copy the last element to the first
    memmove(arr,(char *)arr + (nelems - 1) * width,width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width,tmp,width);
}

int main(int argc, char **argv)
{
    int i_array[] = {10,40,80,20,-30,50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100,400,800,200,-300,500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends(i_array,i_nelems,sizeof(i_array[0]));
    swap_ends(l_array,l_nelems,sizeof(l_array[0]));
...
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

int main(int argc, char **argv)
{
    int i_array[] = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100, 400, 800, 200, -300, 500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(l_array, l_nelems, sizeof(l_array[0]));
    ...
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

int main(int argc, char **argv)
{
    int i_array[] = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100, 400, 800, 200, -300, 500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(l_array, l_nelems, sizeof(l_array[0]));
    ...

Create a char array to hold the width of the element we want to swap.

At this point, all information about the int array is gone, so we just have to rely on the \texttt{width} argument.

Move 4 bytes from the first element in the array to \texttt{tmp}. 
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp,arr,width);

    // copy the last element to the first
    memmove(arr,(char *)arr + (nelems - 1) * width,width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width,tmp,width);
}

int main(int argc, char **argv)
{
    int i_array[] = {10,40,80,20,-30,50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100,400,800,200,-300,500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends(i_array,i_nelems,sizeof(i_array[0]));
    swap_ends(l_array,l_nelems,sizeof(l_array[0]));
...
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

int main(int argc, char **argv)
{
    int i_array[] = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100, 400, 800, 200, -300, 500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(l_array, l_nelems, sizeof(l_array[0]));
    ...

Create a char array to hold the width of the element we want to swap.

At this point, all information about the int array is gone, so we just have to rely on the width argument.

Move 4 bytes from the first element in the array to tmp.

Move 4 bytes from the last element in the array to the first element.

Move 4 bytes from tmp to the last position in the array.
void swap_ends(void *arr, size_t nelems, int width) {
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, ((char *)arr + (nelems - 1) * width), width);

    // copy tmp to the last element
    memmove(((char *)arr + (nelems - 1) * width), tmp, width);
}

int main(int argc, char **argv) {
    int i_array[] = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100, 400, 800, 200, -300, 500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(l_array, l_nelems, sizeof(l_array[0]));
...

Repeat the process for the long array, which will pass in a width of 8:

sizeof(l_array[0]);
We've seen that this function works on elements of an integer type.

The beauty is that it will work on any array. What about char ** arrays, like argv? Sure — argv is a pointer to an array, and we'll be swapping pointers, not moving string chars.

```
$ gcc -g -O0 -std=gnu99 -Wall prog_name_to_end.c -o prog_name_to_end
$ ./prog_name_to_end abc def ghi
ghi
abc
def
$ ./prog_name_to_end abc def ghi
```
void swap_ends(void *arr, size_t nelems, int width)
{
    char tmp[width];

    memmove(tmp, arr, width);

    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

int main(int argc, char **argv)
{
    swap_ends(argv, argc, sizeof(argv[0]));
    for (int i=0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
# include<stdio.h>
# include<stdlib.h>
# include<string.h>

void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[sizeof(argv[0])];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

int main(int argc, char **argv)
{
    swap_ends(argv, argc, sizeof(argv[0]));
    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```c
// file: prog_name_to_end.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swap_ends(void *arr, size_t nelems, int width) {
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

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

What is the type of `argv[0]`?

- char *

What is `sizeof(argv[0])`?
// file: prog_name_to_end.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp,arr,width);

    // copy the last element to the first
    memmove(arr,(char *)arr + (nelems - 1) * width,width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width,tmp,width);
}

int main(int argc, char **argv)
{
    swap_ends(argv,argc,sizeof(argv[0]));
    for (int i=0; i < argc; i++) {
        printf("%s\n",argv[i]);
    }
    return 0;
}
It's really generic

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

void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

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

What is the type of `argv[0]`?

char *

What is `sizeof(argv[0])`?

8

What is the type of `argv`?
It's really generic

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

void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

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

What is the type of `argv[0]`? `char *`

What is `sizeof(argv[0])`? `8`

What is the type of `argv`? `char **`
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

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

What is the type of `argv[0]`?
```
char *
```

What is `sizeof(argv[0])`?
```
8
```

What is the type of `argv`?
```
char **
```

What is the underlying type (unknown to the function) of `arr` in `swap_ends`?
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

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

What is the type of `argv[0]`?

```
cchar *
```

What is `sizeof(argv[0])`?

```
8
```

What is the type of `argv`?

```
cchar **
```

What is the underlying type (unknown to the function) of `arr` in `swap_ends`?

```
cchar **
```
void swap_ends(void *arr, size_t nelems, int width) {
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memmove(tmp, arr, width);

    // copy the last element to the first
    memmove(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memmove((char *)arr + (nelems - 1) * width, tmp, width);
}

int main(int argc, char **argv) {
    swap_ends(argv, argc, sizeof(argv[0]));
    for (int i=0; i < argc; i++) {
        printf("%s
", argv[i]);
    }
    return 0;
}
Now it is time to really ramp up the generic nature of \texttt{void *} pointers.

So far, we've seen how we can manipulate array elements by knowing their width. But we can't do much else with them — sometimes you really do need to know the underlying type to be able to work with a piece of data!

For example, what if we wanted to print arrays generically. In other words, we want a function like this:

\begin{verbatim}
void print_array(void *arr, size_t nelems, int width)
\end{verbatim}

We want the function to print the elements. What challenges are there in this case?
The challenges are that we have no idea how to print something pointed to by a `void *`!

Let's make a first attempt at our function:

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

Houston, we have a problem.
The challenges are that we have no idea how to print something pointed to by a `void *`!

Let's make a first attempt at our function:

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

Houston, we have a problem. What goes in the format string for the `printf` call?
The challenges are that we have no idea how to print something pointed to by a
void *!

Let's make a first attempt at our function:

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

Houston, we have a problem. What goes in the format string for the `printf` call?

We have no idea! Because we don't know what the elements in `arr` are, we have no hope to print them correctly. They could be floats, chars, ints, char *s, etc…
Wouldn't it be nice if we could have the calling function tell us how to print the elements in the array?

Well, we can!

In C, we are allowed to pass function pointers as parameters to other functions. A function pointer tells the other function, "Hey, run this function on the element when you get to it!" The other function has no idea what type it will work on, but it just runs the function with the element and gets back the result, which it can use to perform more work.

In this way, we can write a generic function to do something, but when it works on each element, it uses the other function to do the work.
Function pointers

The function pointer syntax is a bit strange. Here is an example:

```c
void (*myfunc)(int);
```

This says, there is a function called `myfunc` that takes one `int` parameter and returns `void` (no return value).

Let's look at a more detailed example:

```c
void *(*myfunc)(int *);
```

You read this "inside out" — this is a function called `myfunc` that takes an `int *` parameter, and returns a `void *` pointer.
You can use the website cdecl.org (or the program, cdecl on Myth) to get details about the type if you have trouble figuring it out:

$ cdecl explain "void (*myfunc)(int)"
declare myfunc as pointer to function (int) returning void

$ cdecl explain "void *(*myfunc)(int *)"
declare myfunc as pointer to function (pointer to int) returning pointer to void

$ cdecl explain "long *(*myfunc)(void *,char,char *)"
declare myfunc as pointer to function (pointer to void, char, pointer to char) returning pointer to long
Let's go back to our print array elements example:

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

Instead of `printf`, we want to call a function that knows how to print the data. We want a function that takes a `void *` element pointer, and then prints it.
Function pointers

Something like this:

```c
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("", ");
    }
}
```

In other words, we need to pass in a function (called `pr_func` in this case), that will do the printing for us.
Function pointers

Something like this:

```c
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(",
");
    }
}
```

In other words, we need to pass in a function (called `pr_func` in this case), that will do the printing for us.
Function pointers

Something like this:

```c
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(",");
    }
}
```

In other words, we need to pass in a function (called `pr_func` in this case), that will do the printing for us.

The calling function provides the function pointer, and the function pointer is specific to the type of data stored in the array.
Let's write a function pointer that will print `int` elements.

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

When you have a function like this, it does know the type of the data stored in the `void *` pointer! We created this function specifically to print `ints`, so it knows that it has an `int` pointer, and we can cast it to that pointer.
Let's write a function pointer that will print int elements.

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

When you have a function like this, it does know the type of the data stored in the `void *` pointer! We created this function specifically to print ints, so it knows that it has an int pointer, and we can cast it to that pointer.

If we know it is an int pointer, why can't we just have the following function definition?

```c
void print_int(int *arr)
```

We can't, because the print_arr function required a generic function.
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"\n") : printf(\,"\n\n");
    }
}

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;
}
```
### 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("%lu", 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_coordinates.c` file from `/afs/ir/class/cs107/samples/lect7`

Also available in the course reader: 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
#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.
// 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

```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`?
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: \texttt{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

Therefore, the type that gets passed to \texttt{compar}_\texttt{str} must be \texttt{char **} pointers. (e.g., \texttt{0x108}, \texttt{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;
}
```

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"
• 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
Extra Slides