CS 107
Lecture 11: More Heap, and Void *

Friday, February 2, 2024

```
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);
}
```

Computer Systems
Winter 2024
Stanford University
Computer Science Department

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

Lecturer: Chris Gregg

Today's Topics

- Logistics
  - Assign3 - Due next Wednesday
  - Midterm next Thursday
- 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

Assign3:

You will have to write a function called **`read_line`** which improves upon the terrible **`gets`** function, and the better **`fgets`** function.

You will also write two utilities, **`myuniq`** and **`mytail`**, where you will use **`read_line`** (so make it good!). Let's look at how **`fgets`** works, and how structs work, and then let's look specifically at **`uniq`** and **`tail`**.

fgets

Here are the important parts of **`man fgets`**:

```
char *fgets(char *s, int size, FILE *stream);

 fgets() reads in at most one less than size characters from stream and
stores them into the buffer pointed to by s.  Reading stops after an EOF or
a newline.  If a newline is read, it is stored into the buffer. A
terminating null byte ('\0') is stored after the last character in the
buffer.

fgets() returns s on success, and NULL on error or when end of file occurs
while no characters have been read.
```

Let's write a quick little program to investigate what all of this means.

fgets

```
// file: fgets-test.c
#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE 32
int main(int argc, char **argv)
{
    FILE *f;
    f = fopen(argv[1], "r");

    char *buffer = malloc(32);
    char *retptr;
    fgets(buffer, BUFSIZE, f);
    while (retptr != NULL) {
        printf("buffer:\n\"%s\"\n", buffer);
        printf("about to call fgets again: \n");
        retptr = fgets(buffer, BUFSIZE, f);
    }
    return 0;
}
```

```
$ cat colors
red
green
yellow
blue
$
```

```
$ ./fgets-test colors
buffer:
"red
"
about to call fgets again:
buffer:
"green
"
about to call fgets again:
buffer:
"yellow
"
about to call fgets again:
buffer:
"blue
"
about to call fgets again:
$
```


fgets

```
// file: fgets-test.c
#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE 32
int main(int argc, char **argv)
{
    FILE *f;
    f = fopen(argv[1], "r");

    char *buffer = malloc(32);
    char *retptr;
    fgets(buffer, BUFSIZE, f);
    while (retptr != NULL) {
        printf("buffer:\n\"%s\"\n", buffer);
        printf("about to call fgets again: \n");
        retptr = fgets(buffer, BUFSIZE, f);
    }
    return 0;
}
```

```
$ cat column-numbers.txt
12345678911234567
123456789112345678921234567893123456789
123456789112345678921234567893123456789
$
```

```
$ ./fgets-test column-numbers.txt
buffer:
"12345678911234567
"
about to call fgets again:
buffer:
"123456789112345678921234567893"
about to call fgets again:
buffer:
"23456789
"
about to call fgets again:
buffer:
"123456789112345678921234567893"
about to call fgets again:
buffer:
"23456789
"
about to call fgets again:$
```


fgets

```
// file: fgets-test.c
#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE 32
int main(int argc, char **argv)
{
    FILE *f;
    f = fopen(argv[1], "r");

    char *buffer = malloc(32);
    char *retptr;
    fgets(buffer, BUFSIZE, f);
    while (retptr != NULL) {
        printf("buffer:\n\"%s\"\n", buffer);
        printf("about to call fgets again: \n");
        retptr = fgets(buffer, BUFSIZE, f);
    }
    return 0;
}
```

```
$ cat colors_no_end_newline
red
green
yellow
blue$
```

```
$ ./fgets-test colors_no_end_newline
buffer:
"red
"
about to call fgets again:
buffer:
"green
"
about to call fgets again:
buffer:
"yellow
"
about to call fgets again:
buffer:
"blue"
about to call fgets again:
$
```


structs

In C, much like in C++, we often want to group different data types together, and we do this in C with a *struct*. Example:

```
struct rectangle {
    int width;
    int height;
    char *name;
};
```

This defines a struct called **`rectangle`** with three fields: **`width`**, **`height`**, and **`name`**.

The type is: **`struct rectangle`**, and you must use the full type when referring to the type. You can use sizeof(struct rectangle) to get the number of bytes of the struct. E.g.,

```
struct rectangle *array_of_10_rects = malloc(10 * sizeof(struct rectangle));
```

At this point, we now have enough memory to store 10 rectangles.

structs

Example of how we can populate / print a rectangle:

```
// file: rect-test.c
#include <stdio.h>
#include <stdlib.h>

struct rectangle {
    int width;
    int height;
    char *name;
};

int main(int argc, char **argv)
{
    printf("sizeof(struct rectangle) = %zu\n", sizeof(struct rectangle)); // prints 16
    struct rectangle *array_of_10_rects = malloc(10 * sizeof(struct rectangle));
    array_of_10_rects[0].width = 5;
    array_of_10_rects[0].height = 10;
    array_of_10_rects[0].name = "tall"; // the string does NOT live in the struct! Only the pointer does!

    printf("Rect 0: w=%d, h=%d, name=%s\n", array_of_10_rects[0].width,
            array_of_10_rects[0].height,
            array_of_10_rects[0].name);
    return 0;
}
```


More on Heap Allocation

```
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!

Generic Pointers

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.,

```
int arr[] = {2, 4, 6, 8, 10};
void *arr_p1 = arr;
int *arr_p2 = arr_p1;
```


Generic Pointers

You **cannot** dereference a `void *` pointer, nor can you use pointer arithmetic with it. E.g.,

```
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`**:

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

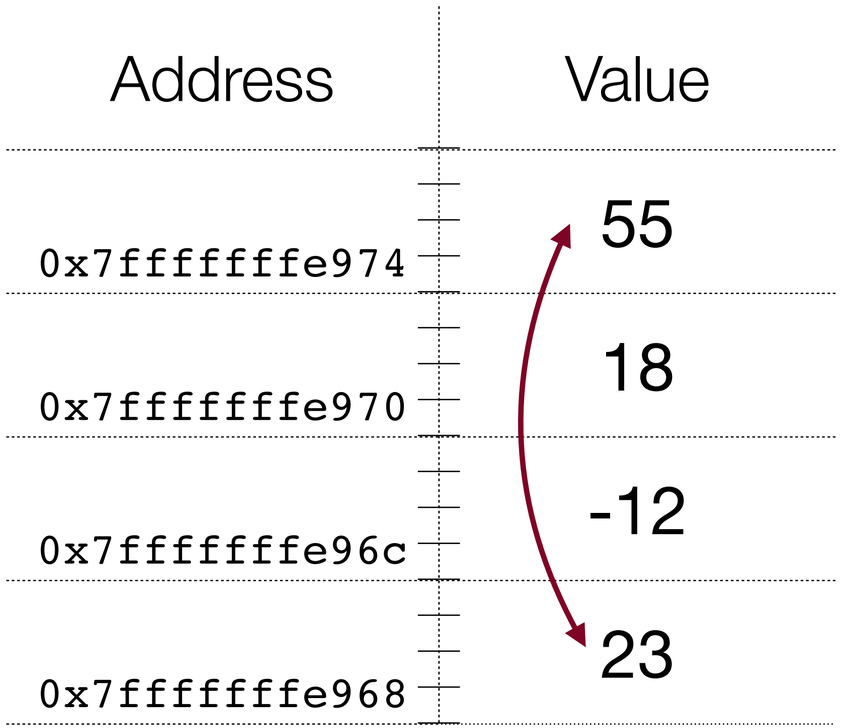
It would not be very nice if we had to have a different **`free`** function for every type of pointer!

Generic Pointers

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:

```
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?

Generic Pointers

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

```
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:

```
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 **`memmove`**?

We must pass the width of the elements in the array because the **`void *`** pointer doesn't carry that information.

Generic Pointers

```
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.

Generic Pointers

```
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!

(we could also use **`malloc`** if we wanted to, but it isn't really necessary here, as the array works just fine*. Regardless, we would still use a **`char *`** pointer)

*not true for C++!

Generic Pointers

```
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!

We copy the bytes with **`memmove`**.

Generic Pointers

```
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
```


Generic Pointers

```
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
```

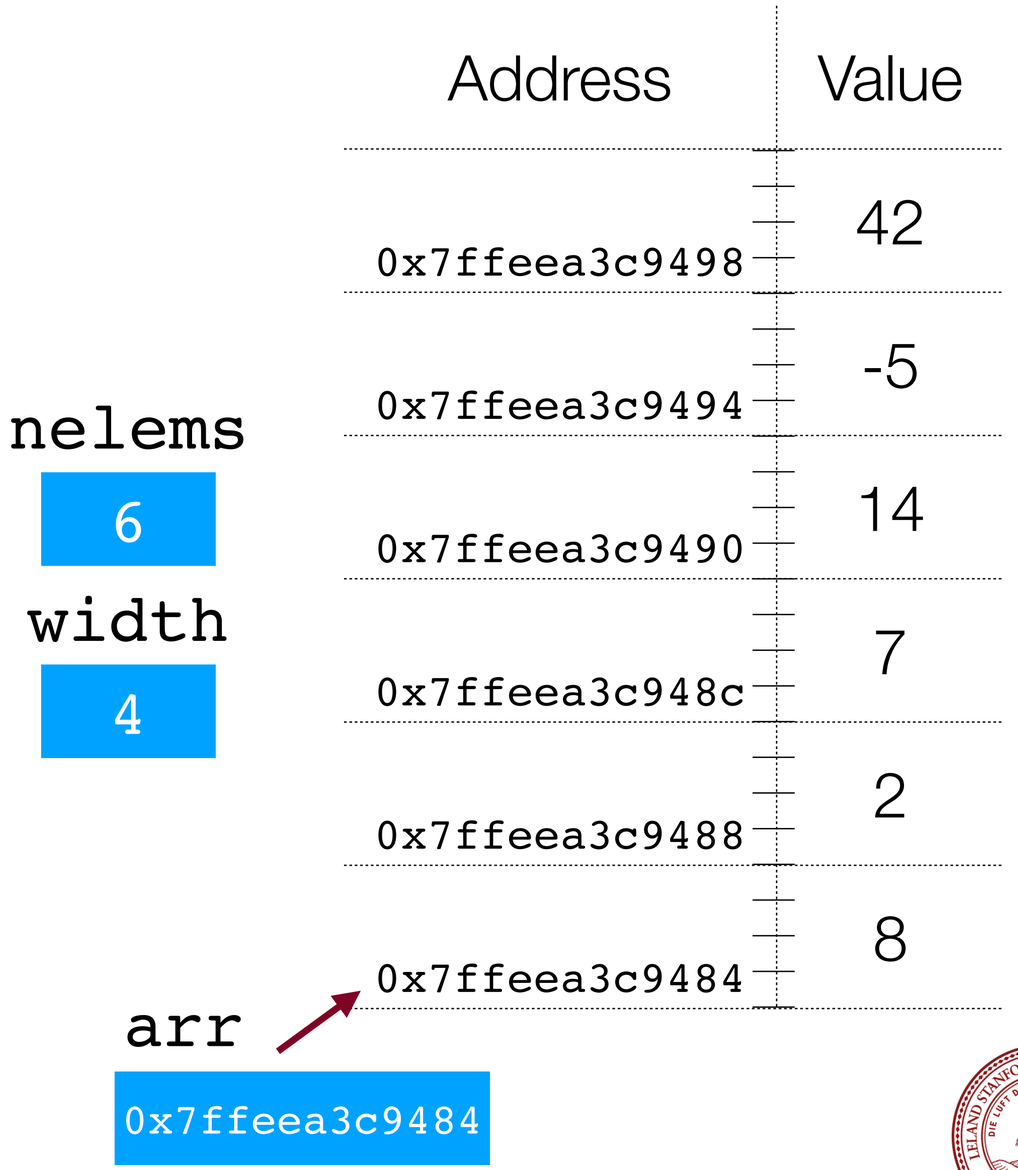
```
0x7ffeea3c9484 + (5 * 4) == 0x7ffeea3c9498
```

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.

Generic Pointers

Very often, we will need to find the i^{th} element in an array. You should be **extremely** familiar with the following idiom:

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

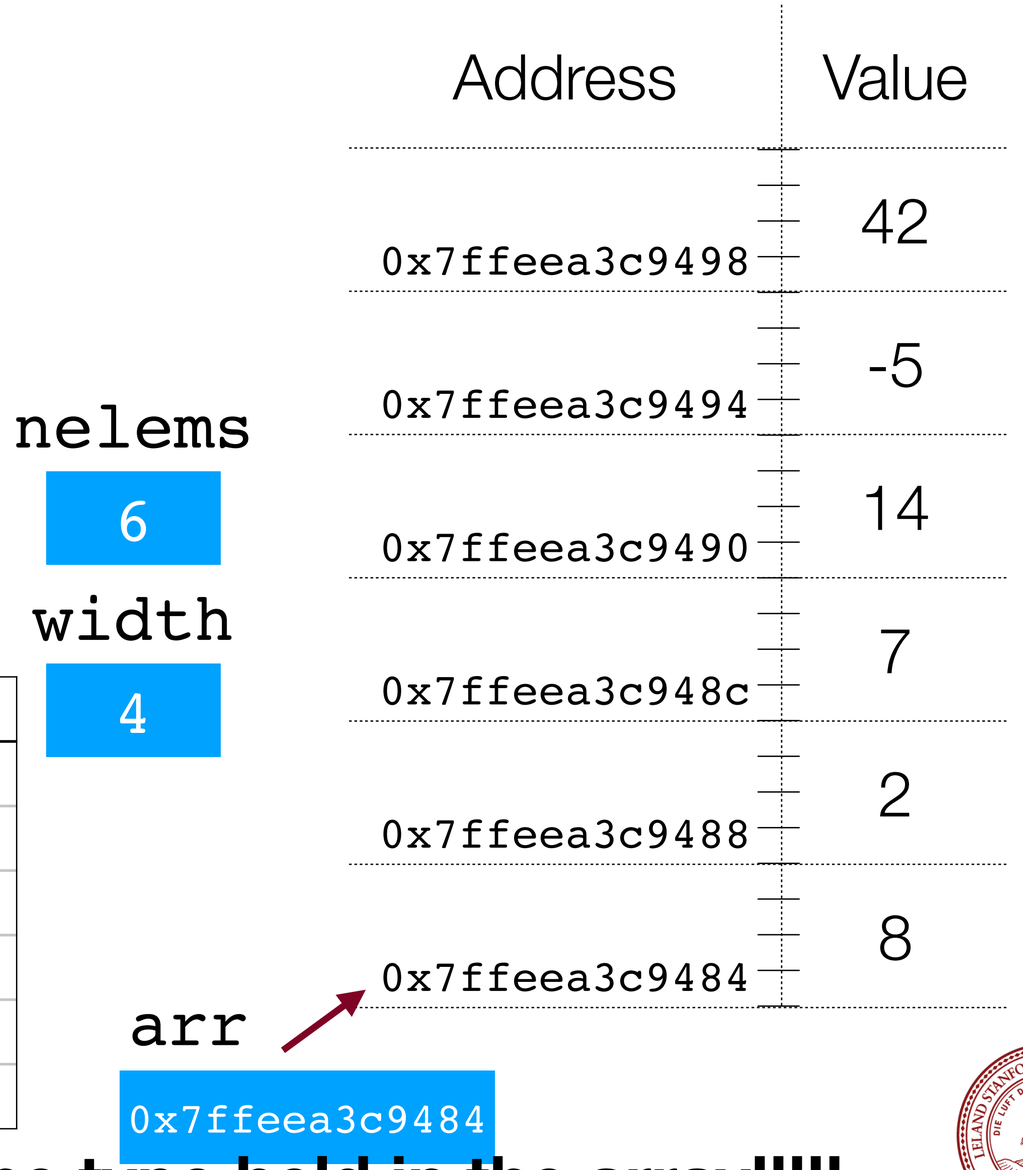


Generic Pointers

Very often, we will need to find the i^{th} element in an array. You should be **extremely** familiar with the following idiom:

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

i	expression	result
0	(char *)0x7ffeea3c9484 + 0 * 4	0x7ffeea3c9484
1	(char *)0x7ffeea3c9484 + 1 * 4	0x7ffeea3c9488
2	(char *)0x7ffeea3c9484 + 2 * 4	0x7ffeea3c948c
3	(char *)0x7ffeea3c9484 + 3 * 4	0x7ffeea3c9490
4	(char *)0x7ffeea3c9484 + 4 * 4	0x7ffeea3c9494
5	(char *)0x7ffeea3c9484 + 5 * 4	0x7ffeea3c9498



Important! These numbers are pointers to the type held in the array!!!!!

Generic example with two different arrays

```
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]));
  ...
```

Let's walk through this example.

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

Generic example with two different arrays

```
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]));
  ...
```

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.

Generic example with two different arrays

```
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]));
...
```

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.

Then, we call swap_ends on the **`int`** array.

Note that we pass in the width, which is 4:

`sizeof(i_array[0])`

Generic example with two different arrays

```
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.

Generic example with two different arrays

```
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`**.

Generic example with two different arrays

```
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.

Generic example with two different arrays

```
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.

Generic example with two different arrays

```
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]);
```

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