Systems and Internet Infrastructure Security
Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

Arrays and Pointers (Part 2)

Devin J. Pohly

Assignment advice

- Use `gcc` for the linking step, not `ld`
- Pay close attention to the function parameters and descriptions in the table!
- Ask questions – the TAs and I are happy to help!
  - Office hours
  - Email

What are pointers?

- Address of some bytes in virtual memory
  - Can point to any segment: stack, heap, data, code (!)
  - Need to understand these segments to avoid pointer bugs
    - Stack memory goes away when the function returns
    - Heap memory goes away when you free() it
    - Code and data (static/global) stick around
- Has an associated type at compile time

Declaring pointers

```
int *head, *tail;
```

- Here the type of head or tail is "pointer to `int`"
- Note the * is repeated for multiple declarations
  - Sometimes you see
    `int* head;`
  - This looks like it declares two pointers, but it doesn't:
    `int* head, tail;`
  - Actually declares *one pointer and one integer*!
  - Recommended style: put the * next to the variable name

Getting an address

- The & operator gives us the virtual address of many things
  - Variables
  - Arrays
  - Array elements
  - Functions (!)
- If `var` is of type `foo`, then `&var` is of type `foo *` ("pointer to foo")

```c
#include <stdio.h>

int foo(int x) {
  return x + 1;
}

int main(int argc, char *argv[]) {
  int x, y;
  int a[2];

  printf("x    is at %p\n", &x);
  printf("y    is at %p\n", &y);
  printf("a    is at %p\n", &a);
  printf("a[0] is at %p\n", &a[0]);
  printf("a[1] is at %p\n", &a[1]);
  printf("foo  is at %p\n", &foo);
  printf("main is at %p\n", &main);
  return 0;
}
```

Setting pointers

- We can then store this address in a matching pointer!

```
int *p = &x;
```

  - We now have a pointer to x that we can use, pass to functions, etc.

```
int *q = p;
```

  - Now both p and q point to x
  - Can point to a different address later:

```
p = &y;
```

  - Now p points to y, and q still points to x

Dereferencing pointers

- To access the *value that is pointed at*, use the * operator:

```
int *p = &x;
```

  - Read: `int y = *p;`
  - Write: `*p = 5;`
  - Called "dereferencing"... which just means follow the arrow

- The * and & operators are complementary

```
*(&x) == x
```

Pointer types

- At runtime, a pointer is just a number
  - But then so is everything else!
- At compile time, it has an associated type
  - Pointer to _____
  - So the compiler can do type checking on & and *

```
int *p = &x;
```

  - What is the type of *p?

Pointer arithmetic

- Caution: addition and subtraction work slightly differently on pointers
  - One of C's more unusual features

```
int *p = &x;
p = p + 1;
```

  - This is legal and useful
  - Moves p forward, not by one byte, but by `sizeof(int)`!
- Why might this be?

Arrays are pointers!

- Suppose we have

```
int a[10];
int *p = &a[0];
```

  - What is `p + 1`?
  - What is `*(p + 5)`?

Arrays are pointers!

- Suppose we have

```
int a[10];
int *p = &a[0];
```

  - What is `p + 1`?
  - What is `*(p + 5)`?
- This is what the `[]` operator does!
  - Syntactic sugar

Returning pointers

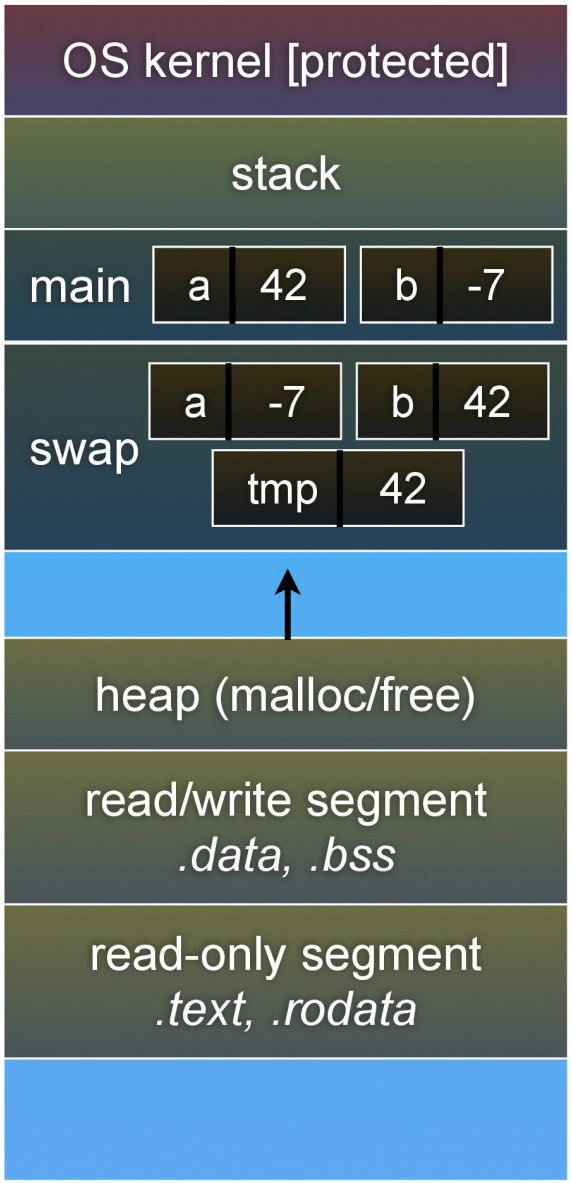
- Functions can *return a pointer* just like any other type
  - Usually returning NULL means failure
  - Not checking this can lead to dereferencing a NULL pointer
    - Segfault!
  - Example: malloc()

```c
#include <stdio.h>

int *find(int arr[], int len, int x) {
  int i;

  for (i = 0; i < len; i++)
    if (arr[i] == x)
      return &arr[i];

  // Didn't find it
  return NULL;
}

int main(int argc, char *argv[]) {
  int arr[20];

  // ...

  int *p = find(arr, 20, -3);
  if (p == NULL) {
    // print an error
    return 1;
  }
  return 0;
}
```

Passing by reference

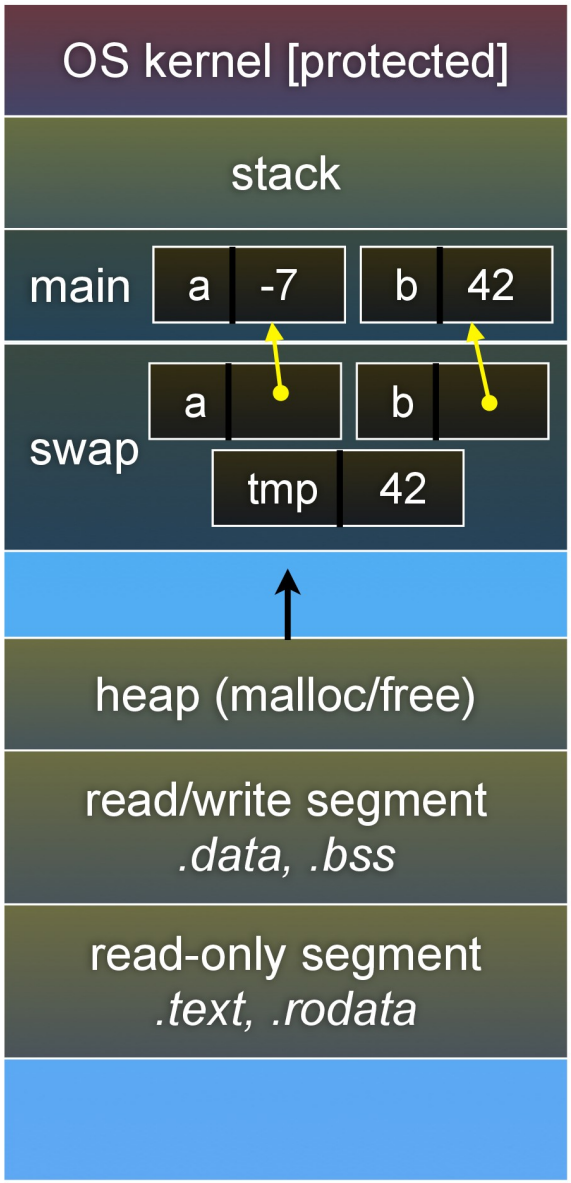
- C makes a copy of each function parameter on the stack
  - But if the parameter is an address, you get a copy of the address
  - Can still get at the original variable
- To jog your memory...

Recap: pass-by-value




```c
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char *argv[]) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Recap: pass-by-reference




```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char *argv[]) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Output parameters

- Pointer parameters to a function can be used both for input and output
- Functions have only one "real" return value
  - And this is often already being used for an error code
  - To return something else, use a pointer
  - To return multiple values, use pointers
- This is normal, idiomatic C

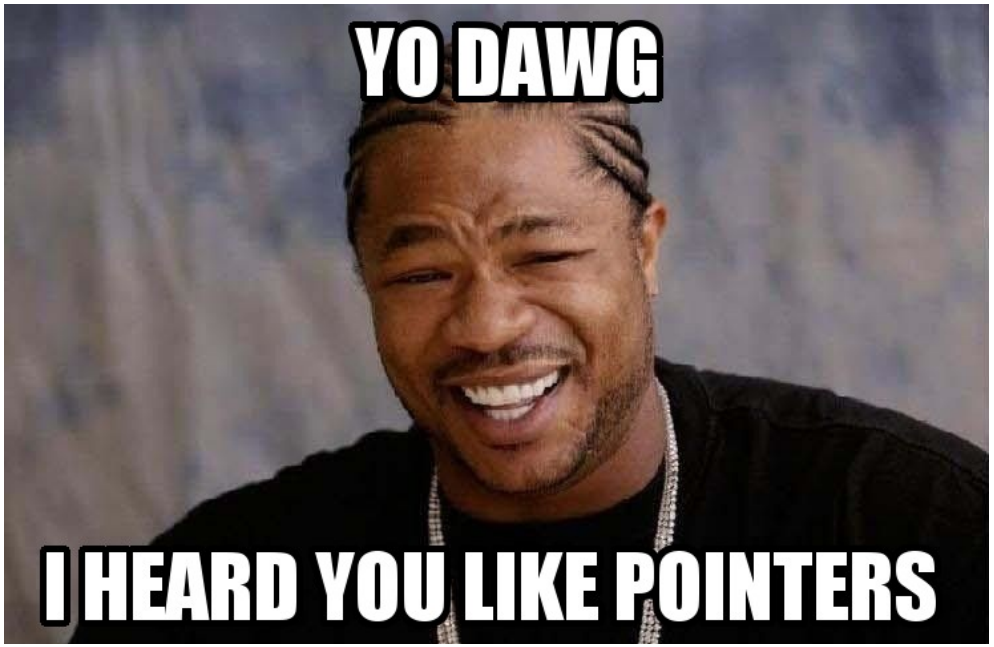
```c
#include <stdio.h>

void get_two_nums(int *a, int *b) {
  *a = 5;
  *b = 20;
}

int main(int argc, char *argv[]) {
  int x, y;

  // Two return values
  get_two_nums(&x, &y);

  printf("x = %d\n", x);
  printf("y = %d\n", y);
  return 0;
}
```

Pointers to pointers

- Pointers can point to any type... including other pointers!

```
int *p = &x;
int **q = &p;
```

- For example, what if...
  - you want to pass a *pointer* by reference?
  - you want an array of pointers?
  - you want to return multiple pointers?



Pointers to void

```
void *ray;
```

- Wait... what type does this point to??
  - No type, treated as raw bytes
  - Must *cast* to a different pointer type before it is useful:

```
int *p = (int *) ray;
```

  - Example: malloc() returns a pointer to void