CS107, Lecture 8
Introduction to Pointers

Reading: K&R (1.9, 5.5, Appendix B3) or Essential C section 3

Reminiscing C++

How might we write a C++ program with a function that takes in an **`int`** and changes it? We might use pass by reference.

```
void func(int& num) {
    num = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    func(x);
    cout << x << endl; // prints 3
    return 0;
}
```

Had **num** been declared as a standard **`int`**, it would catch a **copy** of **x**'s value at the time of the call.

Because it's declared as an **`int&`**—that is, a reference to an **`int`**—**num** functions as a **second name** of a previously existing **`int`**. Accessing or updating **num** is really an operation on **x** itself.

Hot Take: **num** behaves like an **automatically dereferenced pointer**. In practice, that's **usually how references work under the hood**.

Sadly, C **does not support** C++-style pass by reference.

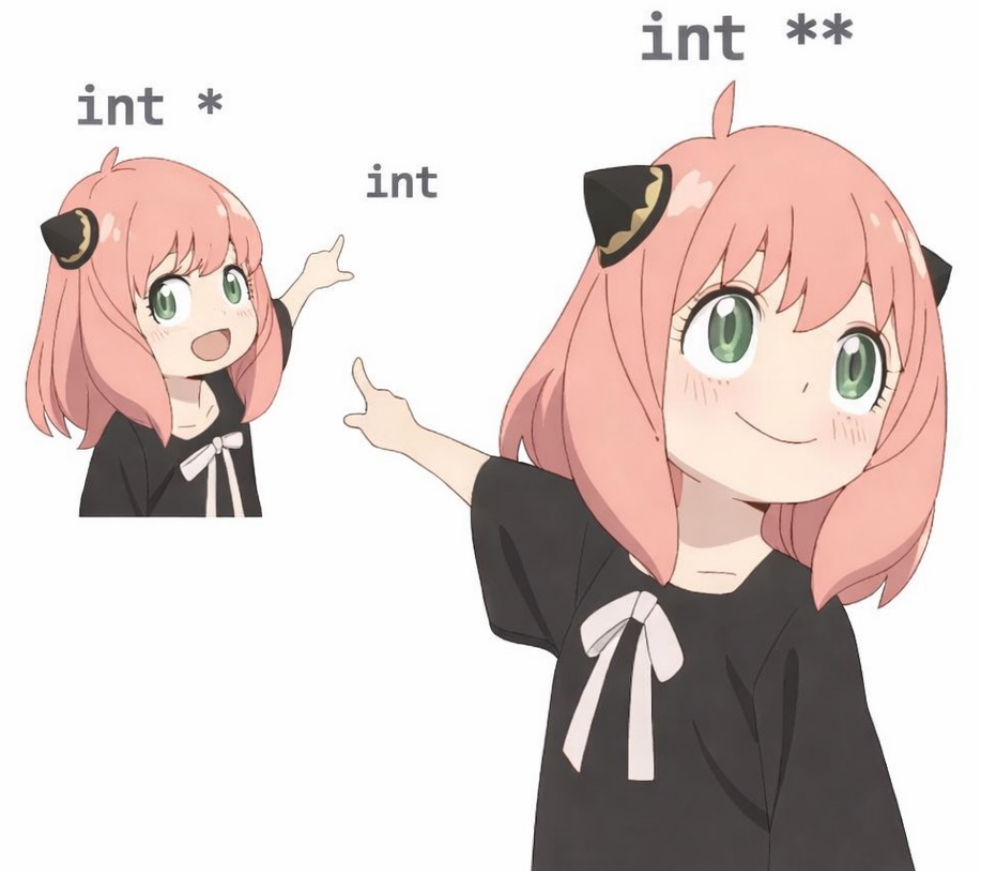
Pointers Revisited

A **pointer** is a variable that stores a memory address—typically the address of something meaningful.

- You've already used C++ pointers in CS106B. **C programmers use them even more**.
- There is **no true pass-by-reference mechanism in C** like there is in C++, so C coders rely on pointers to **expose the variable addresses** to helper functions so those functions can **access and even update their values**.

Pointers are essential to **dynamic memory allocation**—arguably more so—in pure C than they are in C++.

> C doesn't define **`vector`**s, **`map`**s, and **`hash_map`**s like C++ does via its standard libraries.
>
> C programmers often need to **wire up their own implementations of them using pointers and dynamically allocated memory**.



Looking Ahead to C

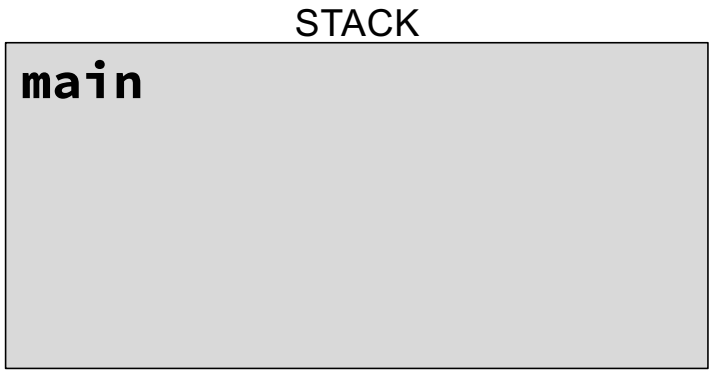
- **All parameters** in C are passed **by value**. When passing arrays as parameters, the base address of the array **decays to a pointer**.
- If an address is passed as a parameter, the **address itself is copied**, just like any other parameter.
  - But because that address is the location of some data residing elsewhere, we have **access** to and can even **modify** that data.
- More generally, if we want to modify a value in a helper function and have any changes persist after the function returns, you can pass in the **address** of the value—that is, its **location**—instead of passing the value itself. This way we copy the address instead of the value.

Pointers Revisited: CS106B Style

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
   *intPtr = 3;
}

int main(int argc, char *argv[]) {
   int x = 2;
   myFunc(&x);
   printf("%d\n", x); // prints 3
   ...
}
```

By passing **&x** to **myFunc**, **main** tells **myFunc** where **x** lives. Here, the **&** is the **address-of** operator and produces the address of **x**.



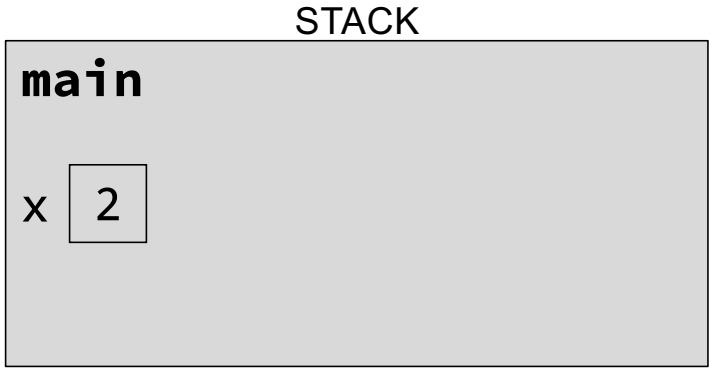
myFunc accepts that location as an address of an **int**. That's why the type here is **int *** instead of **int**.

Here, the value of **intPtr** identifies where the shared **int** lives. By dereferencing **intPtr**, the code identifies the **int** it points to—and not **intPtr** itself—as the recipient of the 3. Think of **intPtr** as a hyperlink and the ***** in front of it as a mouse click that drills through the link to the destination.

Pointers Revisited: CS106B Style

A pointer is a variable that stores a memory address.

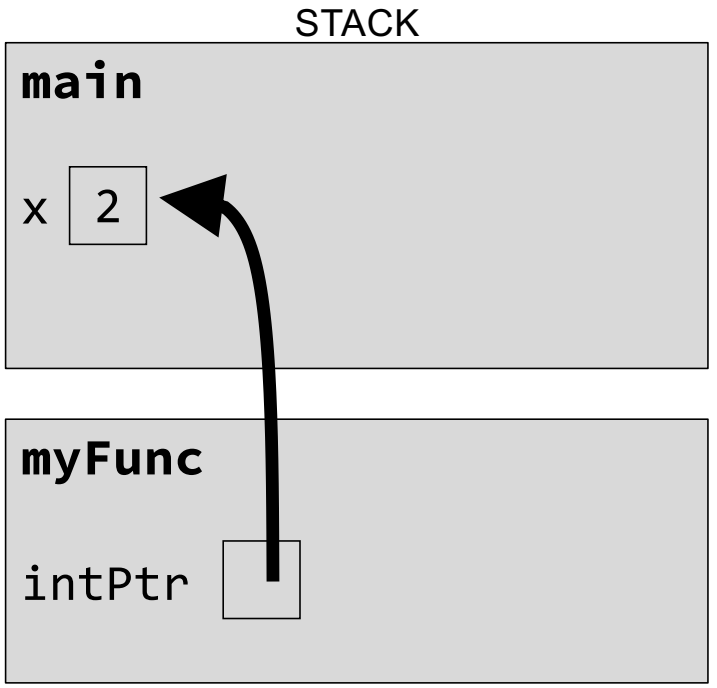
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS106B Style

A pointer is a variable that stores a memory address.

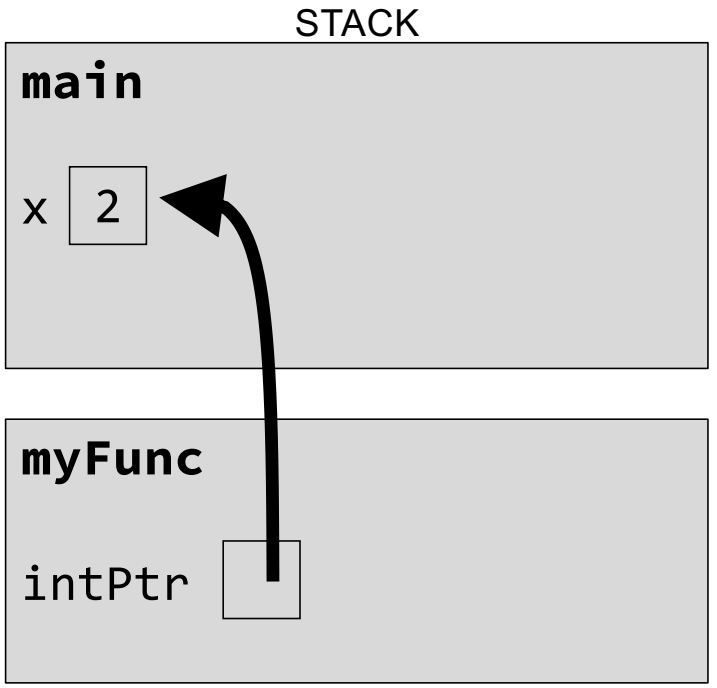
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS106B Style

A pointer is a variable that stores a memory address.

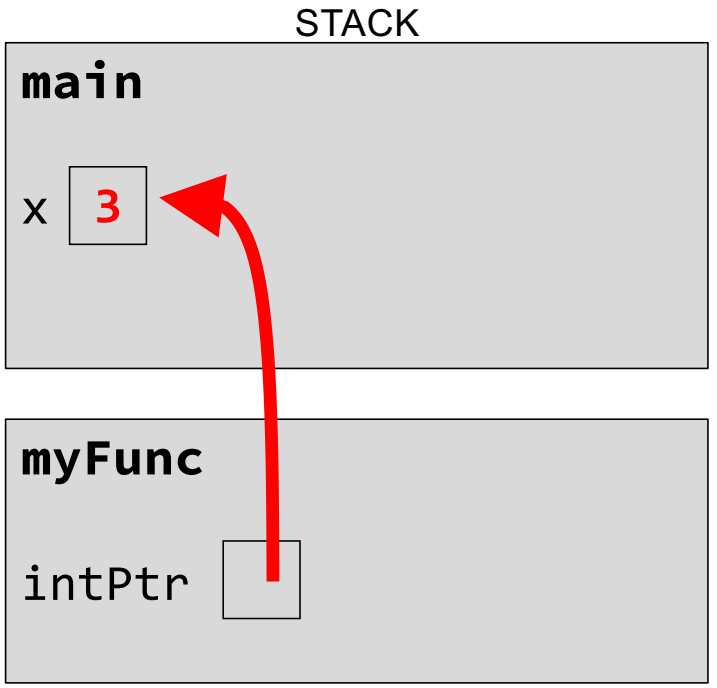
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS106B Style

A pointer is a variable that stores a memory address.

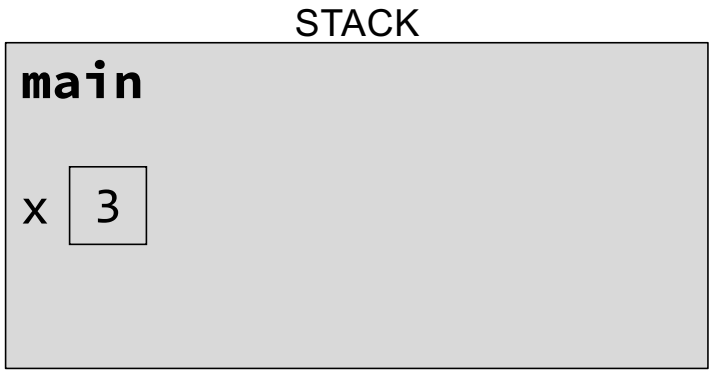
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS106B Style

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS107 Style

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
   *intPtr = 3;
}

int main(int argc, char *argv[]) {
   int x = 2;
   myFunc(&x);
   printf("%d\n", x); // prints 3
   return 0;
}
```

Pointers Revisited: CS107 Style

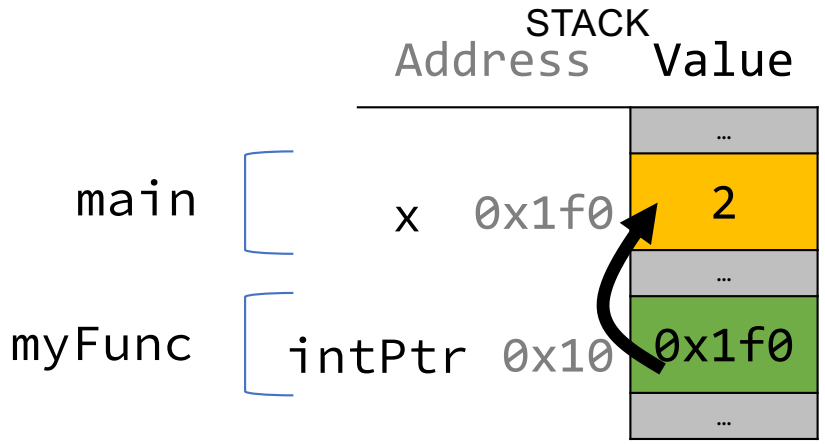
A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
   *intPtr = 3;
}

int main(int argc, char *argv[]) {
   int x = 2;
   myFunc(&x);
   printf("%d\n", x); // prints 3
   return 0;
}
```

Pointers Revisited: CS107 Style

A pointer is a variable that stores a memory address.

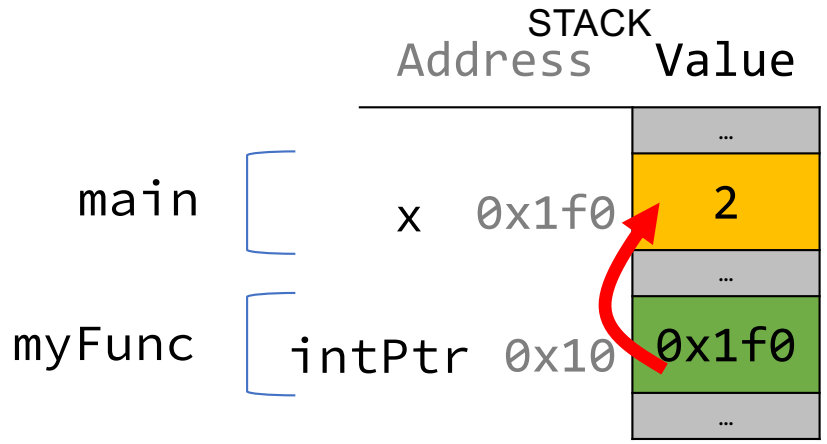
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS107 Style

A pointer is a variable that stores a memory address.

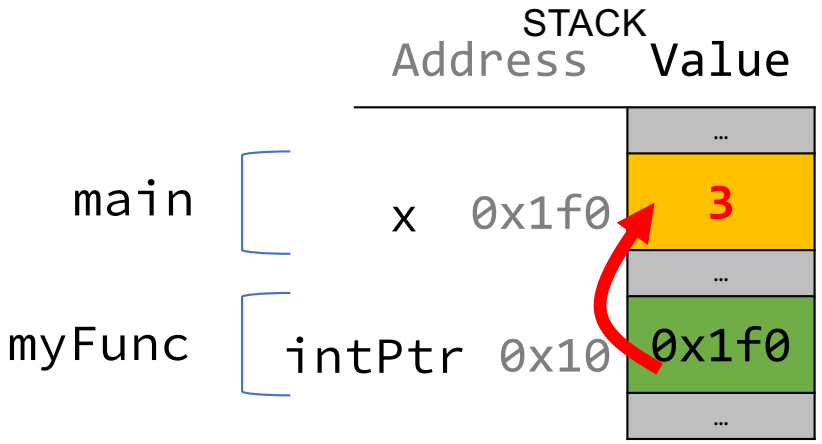
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS107 Style

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d\n", x); // prints 3
    return 0;
}
```



Pointers Revisited: CS107 Style

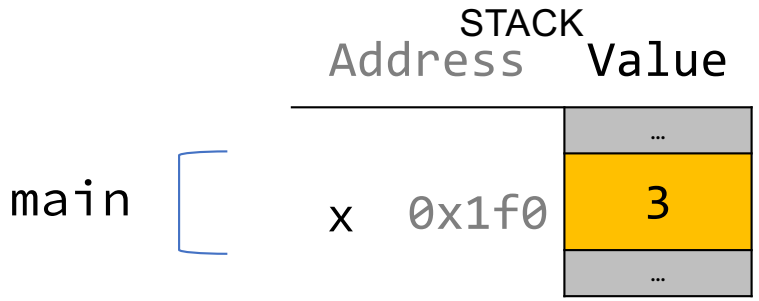
A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
   *intPtr = 3;
}

int main(int argc, char *argv[]) {
   int x = 2;
   myFunc(&x);
   printf("%d\n", x); // prints 3
   return 0;
}
```

Pointers Revisited: CS107 Style

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
   *intPtr = 3;
}

int main(int argc, char *argv[]) {
   int x = 2;
   myFunc(&x);
   printf("%d\n", x); // prints 3
   return 0;
}
```



Pointers and Parameters Etudes

We want to write a function that flips the case of a letter.
What should go in each of the blanks?

```
void flip_case(         ) {
   if (isupper(   )) {
          =              ;
   } else if (islower(   )) {
          =
   }
```

```
int main(int argc, char *argv[]) {
   char ch = 'g';
   flip_case(   );
   printf("%c\n", ch); // print 'G'
   return 0;
}
```

The **address** of a **char** must be caught by a **char *****.**

The flipped **char** must be **written back to the original** through **cp**. ***cp** identifies where the flipped **char** **should be placed**.

The case of the **char** to be flipped must be determined, and ***cp** identifies at that **char**.

We want **flip_case**, to modify **main**'s **ch**, so we need to pass **ch**'s **address**.

Pointers and Parameters Etudes

Sometimes, we want to modify a pointer—e.g., a **`char *`**—by reference by sharing the address of that pointer with a function.

Consider the following (broken) implementation of **`skip_spaces`**, which purports to update a C string to leap beyond its leading whitespace.

```
void skip_spaces(char *s) {
    s += strspn(s, " ")
}

int main(int argc, char *argv[]) {
    char *str = "             hello";
    skip_spaces(str);
    printf("%s\n", str); // should print hello
    return 0;
}
```

Here's a compact autopsy report outlining why this version is dead on arrival.

- **`skip_spaces`** receives **a copy of the address stored in `main`'s `str`** variable.
- The **`strspn`** call is correct and successfully advances the local variable **`s`**, but it **doesn't advance or otherwise change `main`'s `str`**.
- When **`skip_spaces`** returns, its updated pointer **`s`** is **discarded**. **`main`'s `str`** wasn't **touched**, much less **updated**.

Pointers and Parameters Etudes

Now consider a new version of **`skip_spaces`**, which relies on a **`char **`** parameter (😱) to capture the address of **`main`**'s **`str`**.

Because **`main`** now shares the location of **`str`** with **`skip_spaces`**, **`skip_spaces`** can access and even directly modify **`str`**'s value.

```
void skip_spaces(char **p_str) {
   *p_str += strspn(*p_str, " ");
}

int main(int argc, char *argv[]) {
   char *str = "             hello";
   skip_spaces(&str);
   printf("%s\n", str); // should print hello
   return 0;
}
```

Here's the new version's grade report:

- **`main`** now passes **`&str`**, the **address of its own `char *` variable**, not some copy of **`str`**.
- The **`char **p_str`** parameter **points directly** to **`main`**'s **`str`**, so dereferencing **`p_str`** accesses the original **`char *`**.
- The expression **`*p_str += ...`** **updates the value stored in `str`**, advancing the original pointer beyond all those spaces. What failed to happen in v1 now works.

Redux: To modify a pointer owned by the caller, the **callee must receive that pointer's address**.

Pointers: Key Takeaways

Summary:

- If you're working with some input and do not care about any changes to the input, **pass the data by value**.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify.
- If a function accept an address as a parameter, that function can travel to that address if need be.

- If a function accepts an `int *`, it can modify the `int` at the supplied address.
- If a function accepts a `char *`, it can modify the `char` at the supplied address.
- If a function accepts a `char **`, it can modify the `char *` at the supplied address.

Strongly Typed Swaps and Rotations

Here are few functions that do more meaningful work than earlier examples.

```
void swap_ints(int *one, int *two) {
   int temp = *one;
   *one = *two;
   *two = temp;
}

void swap_strings(char **one, char **two) {
   char *temp = *one;
   *one = *two;
   *two = temp;
}

void rotate(char **p, char **q, char **r) {
   swap_strings(p, q);
   swap_strings(p, r);
}
```

```
int main(int argc, char *argv[]) {
   int x = 17, y = 29;
   printf("x = %d, y = %d\n", x, y);
   swap_ints(&x, &y);
   printf("x = %d, y = %d\n", x, y);

   char *h = "Fred", *w = "Wilma";
   printf("husband: %s, wife: %s\n", h, w);
   swap_strings(&h, &w);
   printf("husband: %s, wife: %s\n", h, w);
   swap_strings(&h, &w); // restore

   char *b = "Pebbles";
   printf("husband: %s, wife: %s, baby: %s\n", h, w, b);
   rotate(&h, &w, &b);
   printf("husband: %s, wife: %s, baby: %s\n", h, w, b);

   return 0;
}
```