CS107, Lecture 8
Introduction to Pointers

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

Ed Discussion

Pointers

- A **pointer** is a variable that stores the address of some figure in memory.
  - You dealt with pointers in C++ during your time in CS106B or its equivalent. The idea is entirely new, but C is much more reliant on pointers than C++ is.
  - There is no true pass-by-reference in C like there is in C++, so we rely on pointers to share the addresses of variables with other functions.
- Pointers are essential to dynamic memory allocation (coming soon).
- Pointers allow us to generically identify memory (coming less soon, but still soon).

Looking Back at C++

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

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

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

Looking Ahead to C

- **All parameters** in C are passed "by value". For efficiency reasons, arrays (and strings, by extension) passed in as parameters are really caught as pointers.
- If an address is passed as a parameter, the **address itself is copied as all parameters are**. 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, we can share the **location** of the value—that is, share its **address**—instead of sharing the value itself. This way we copy the *address* instead of the *value*.

Pointers

```
int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xptr = &x;

// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xptr);    // prints 2
```

`*`

If **declaration**: "pointer"
ex: `int *` is "pointer to an int"
If **operation**: "dereference/the value at address"
ex: `*num` is "the value at address num"

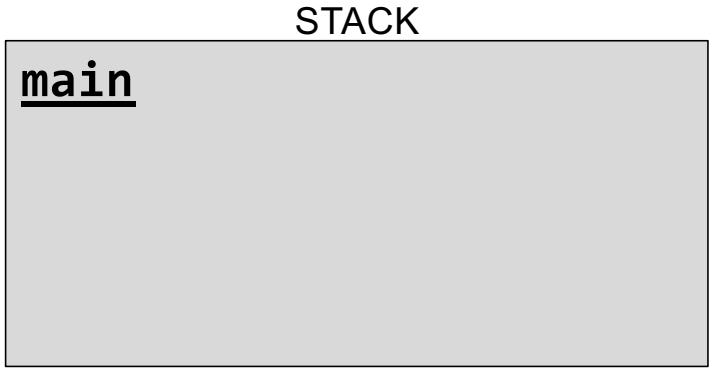
Pointers

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", x);     // 3!
     ...
}
```

Pointers

A pointer is a variable that stores a memory address.

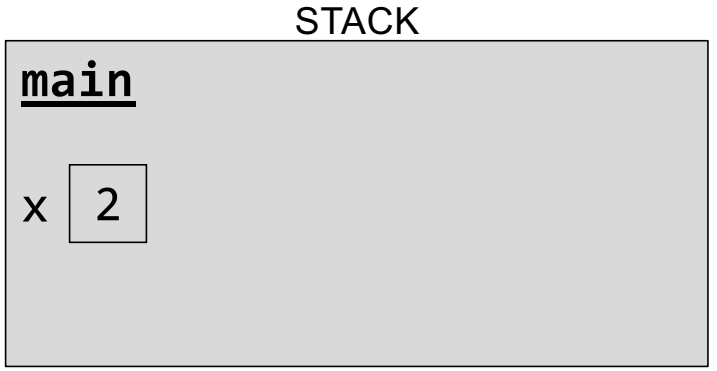



```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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

Pointers

A pointer is a variable that stores a memory address.

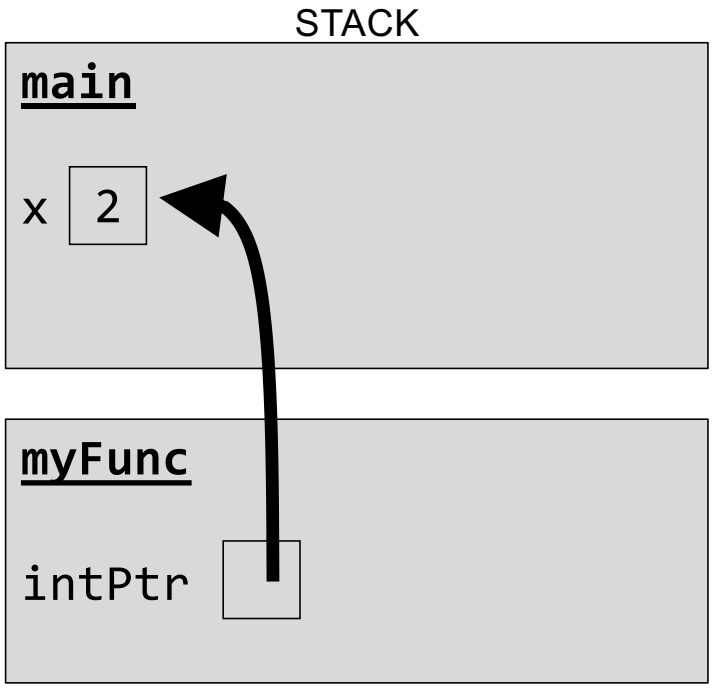
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

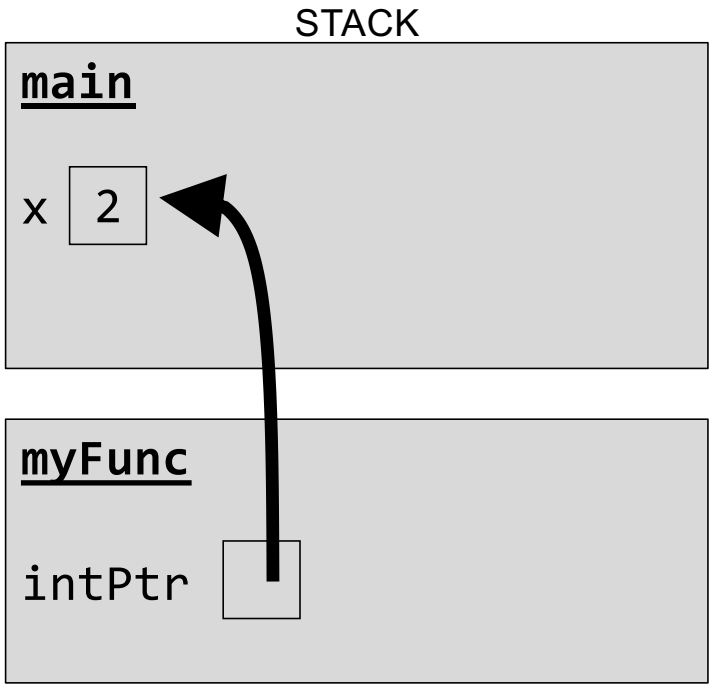
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

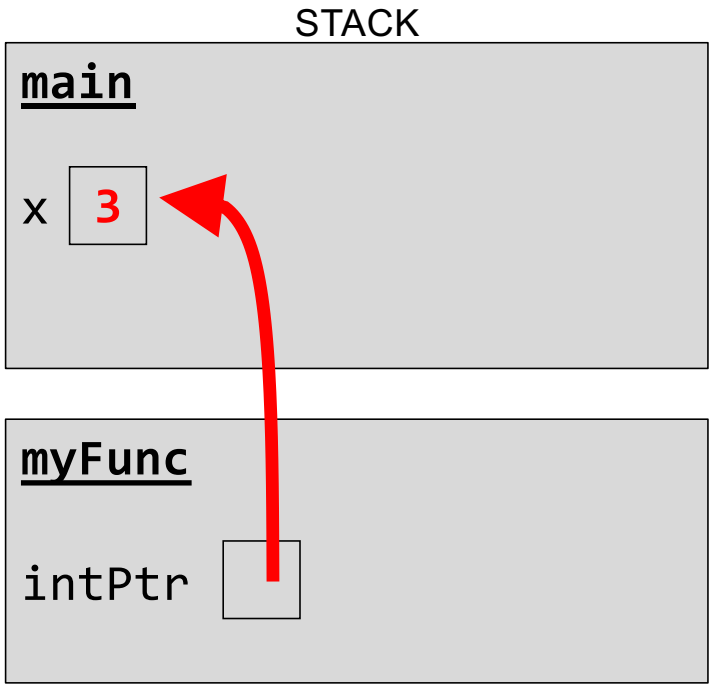
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

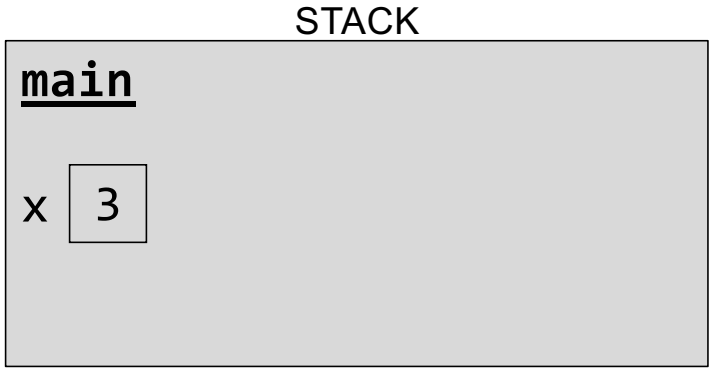
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

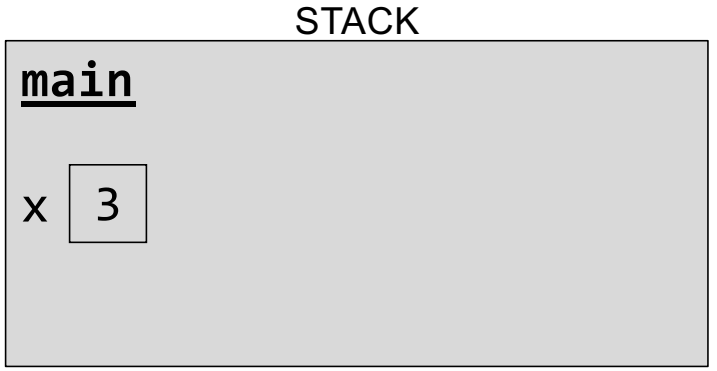
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

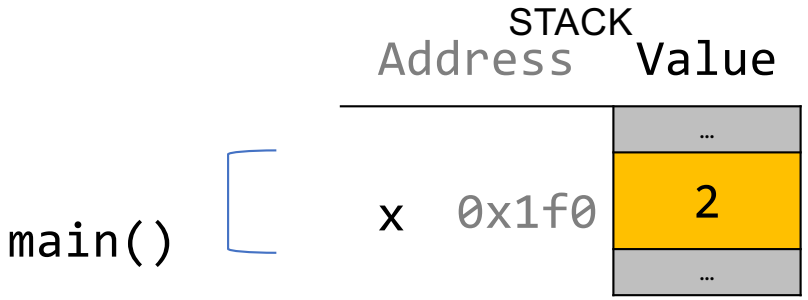
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

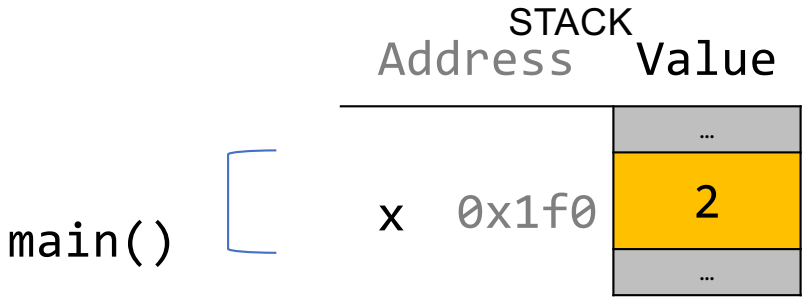
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

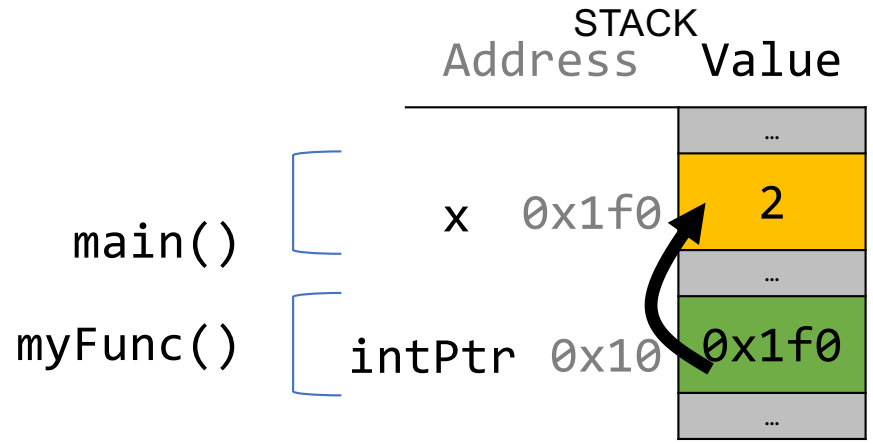
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

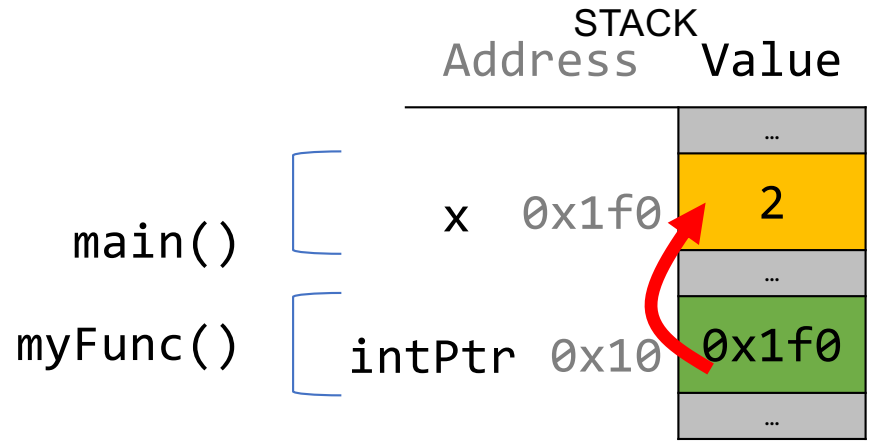
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

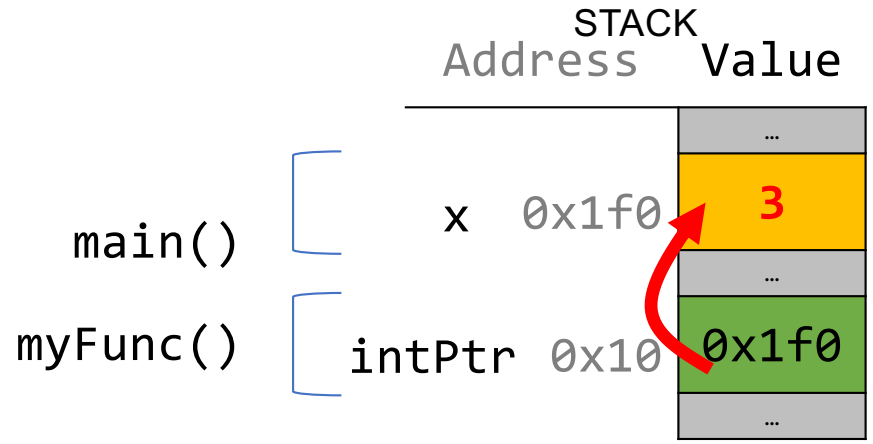
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

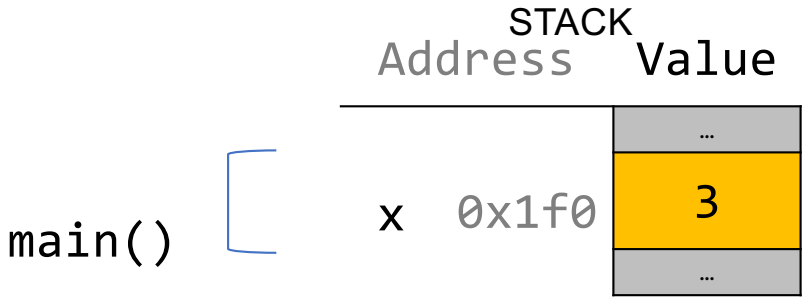
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

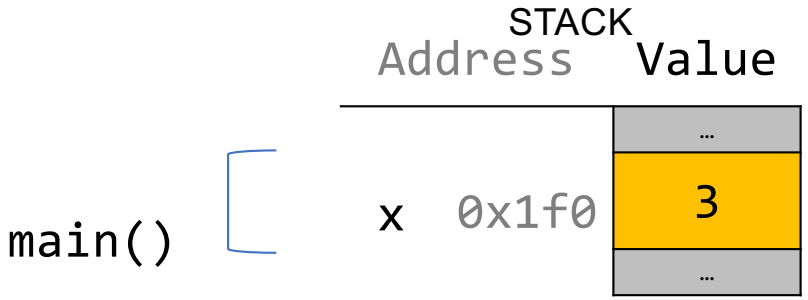
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

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", x);     // 3!
     ...
}
```



Exercise 1

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

```
void flipCase(__?__) {
      if (isupper(__?__)) {
            __?__ = __?__;
      } else if (islower(__?__)) {
            __?__ = __?__;
      }
}

int main(int argc, char *argv[]) {
      char ch = 'g';
      flipCase(__?__);
      printf("%c", ch);      // want this to print 'G'
}
```

Exercise 1

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

```
void flipCase(char *letter) {
      if (isupper(*letter)) {
            *letter = tolower(*letter);
      } else if (islower(*letter)) {
            *letter = toupper(*letter);
      }
}

int main(int argc, char *argv[]) {
      char ch = 'g';
      flipCase(&ch);
      printf("%c", ch);      // want this to print 'G'
```

We are modifying a specific instance of the letter, so we pass the **location** of the letter we would like to modify.

Exercise 2

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to, e.g., we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(__?__) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(__?__);
    printf("%s", str);      // should print "hello"
}
```

Exercise 2

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to, e.g., we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(char **strPtr) {
     // code that advances *strPtr
}

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

We are modifying a specific instance of the string pointer, so we pass the **location** of the string pointer we would like to modify.

Exercise 2

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to, e.g., we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(char *strPtr) {
     // code incapable of modifying str of main
}

int main(int argc, char *argv[]) {
     char *str = "    hello";
     skipSpaces(str);
     printf("%s", str); // should print "hello", but won't
}
```

This can only advance **skipSpace**'s own copy of the string pointer, not the instance in main.

Pointers Summary

- If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

- 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.

Demo: Swap

`fun_with_swap.c`