CS107, Lecture 8

C String Wrap, Buffer Overflow, Security, Introduction to Pointers

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

Ed Discussion: https://edstem.org/us/courses/46162/discussion/3606436

Recall: Buffer Overflows

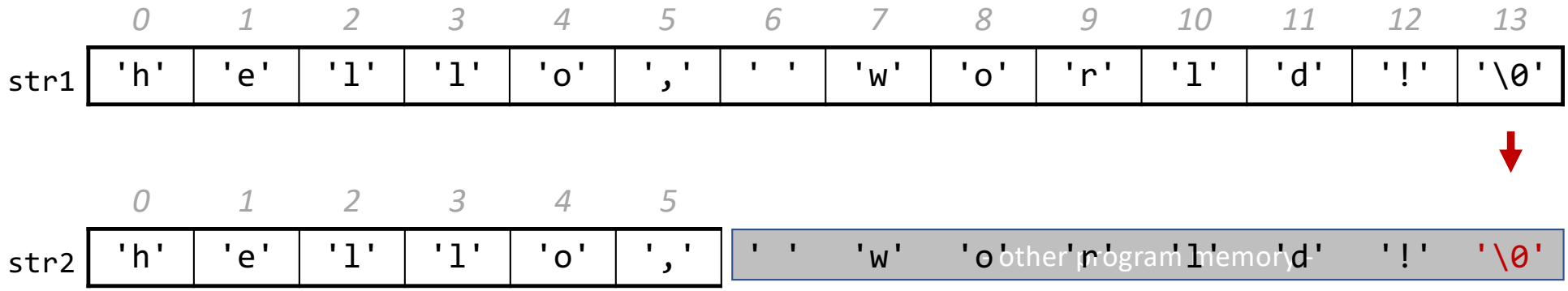
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
char str2[6];                       // not enough space!
strcpy(str2, "hello, world!");      // overwrites other memory!
```

Writing past memory bounds is called a "buffer overflow". It can allow for security vulnerabilities!

Recall: Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```



Buffer Overflow Impacts

Buffer overflows can be serious, as they can lead to runtime errors and even introduce security vulnerabilities into a program. Examples include:

- accessing memory you shouldn't be able to access
- modifying memory you shouldn't be modifying
- changing the value of a variable used later in the program
- changing the program to execute your assembly code instructions instead of its own

It's our job as programmers to find and fix buffer overflows and other bugs, not just for the functional correctness of our programs, but to protect people who use and interact with our code.

Buffer Overflow Impacts

- AOL instant messenger buffer overflow: allowed remote attackers to execute code:

https://www.cvedetails.com/cve/CVE-2002-0362/
https://www.computerworld.com/article/2586310/aol-instant-messenger-vulnerable-to-hackers.html

- Morris Worm: first internet worm to gain widespread attention; exploited buffer overflow in Unix command called "finger":

https://en.wikipedia.org/wiki/Morris_worm

How can we identify buffer overflows?

There's no single solution that works for everything. Finding and repairing overflow vulnerabilities require a combination of software development techniques:

- vigilance while programming (scrutinizing array reads and writes, pointer arithmetic)
- carefully reading documentation
- thoroughly testing to identify issues before shipping product
- thoroughly documenting assumptions in your code
- using software tools to methodically examine code for suspicious function calls

How can we identify buffer overflow?

MAN page for gets():

```
char *gets(char *s);
```

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

How can we identify buffer overflows?

- **Valgrind**: your best friend for this
- Write your own tests
- Consider writing tests *before* writing the main program

✨ `cs107.stanford.edu/testing.html` ✨

How Can We Fix Overflows?

Documentation & MAN Pages (Written by Others)

The strcpy() function copies the string pointed to by src, including the terminating null byte ('\0'), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. **Beware of buffer overruns!**

If the destination string of a strcpy() is not large enough, then **anything might happen**. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there's enough space. This may be unnecessary if you can show that overflow is impossible, but be careful: programs can get changed over time, in ways that may make the impossible possible.

Memory Safe Systems Programming

Choose your Tools & Languages Carefully

Existing code bases or requirements for a project may dictate what tools you use. Knowing C is crucial – it is and will remain widely used.

When you you are choosing tools for systems programming, consider languages that can help guard against programmer error.

- Rust (Mozilla)
- Go (Google)
- Project Verona (Microsoft)

Association for Computing Machinery (ACM) Code of Ethics

ACM Code of Ethics and Professional Conduct

ACM Code of Ethics and Professional Conduct

Preamble

Computing professionals' actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good. The ACM Code of Ethics and Professional Conduct ("the Code") expresses the conscience of the profession.

The Code is designed to inspire and guide the ethical conduct of all computing professionals, including current and aspiring practitioners, instructors, students, influencers, and anyone who uses computing technology in an impactful way. Additionally, the Code serves as a basis for remediation when violations occur. The Code includes principles formulated as statements of responsibility, based on the understanding that the public good is always the primary consideration. Each principle is supplemented by guidelines, which provide explanations to assist computing professionals in understanding and

On This Page

ACM Code of Ethics on Security

2.9 Design and implement systems that are robustly and usably secure.

Breaches of computer security cause harm. Robust security should be a primary consideration when designing and implementing systems. Computing professionals should perform due diligence to ensure the system functions as intended, and take appropriate action to secure resources against accidental and intentional misuse, modification, and denial of service. As threats can arise and change after a system is deployed, computing professionals should integrate mitigation techniques and policies, such as monitoring, patching, and vulnerability reporting. Computing professionals should also take steps to ensure parties affected by data breaches are notified in a timely and clear manner, providing appropriate guidance and remediation.

To ensure the system achieves its intended purpose, security features should be designed to be as intuitive and easy to use as possible. Computing professionals should discourage security precautions that are too confusing, are situationally inappropriate, or otherwise inhibit legitimate use.

In cases where misuse or harm are predictable or unavoidable, the best option may be to not implement the system.

Demo: Memory Errors

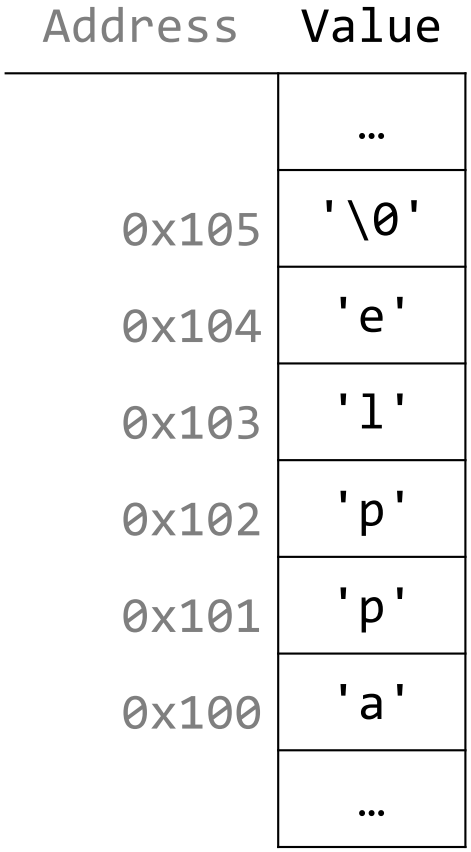
`memory_errors.c`

Pointers

- A *pointer* is a variable that stores a memory address.
- Because there is no pass-by-reference in C like in C++, we rely on pointers to share the addresses of variables with other functions.
- A single pointer can identify a single byte or an arbitrarily large data structure!
- Pointers are essential to dynamic memory allocation (coming soon).
- Pointers allow us to generically identify memory (coming less soon, but still soon).

Memory

- Memory is a big array of bytes.
- Each byte has a unique numeric index that is generally written in hexadecimal.
- A pointer stores any one of these memory addresses.



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 myFunc(int& num) {
     num = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", 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 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 data meaningful to program execution, we have access to, and can even modify, that data.
- More generally, if we want to modify a parameter value in a function and have any changes persist afterward 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

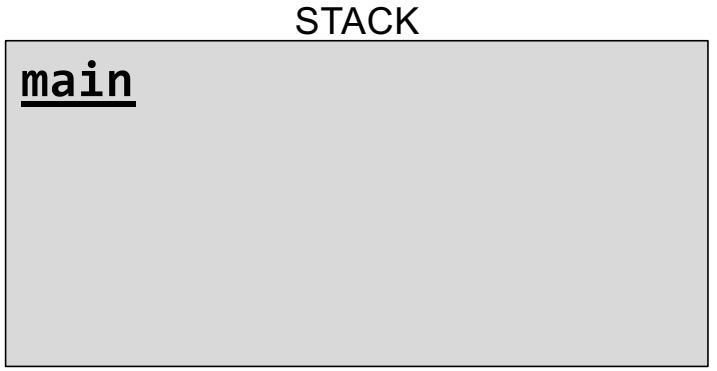
A pointer is a variable that stores a memory address.

```
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.

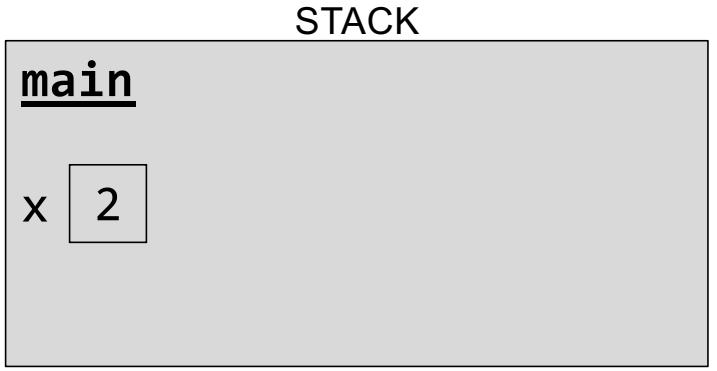
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

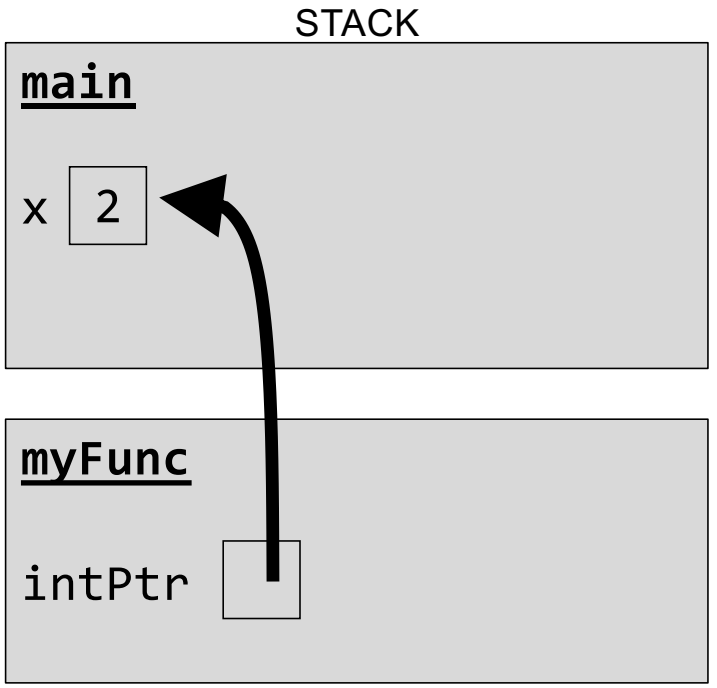
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

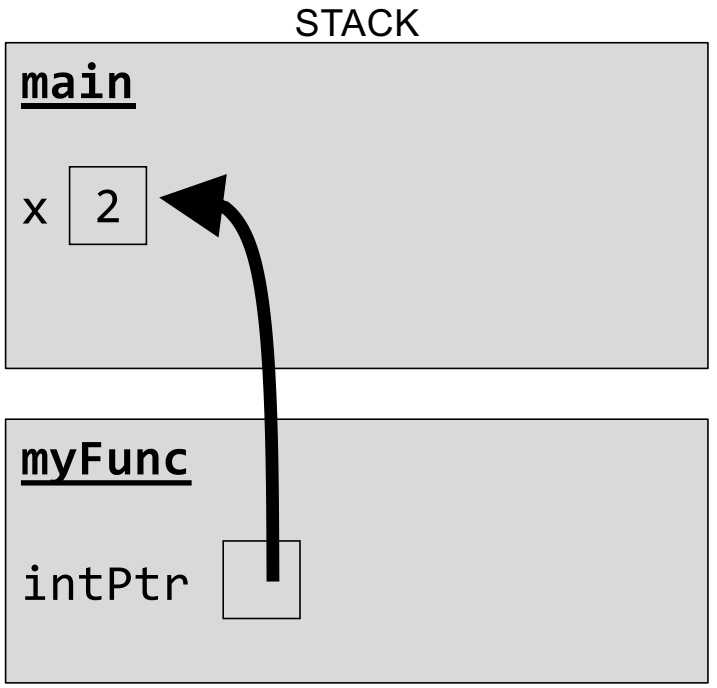
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

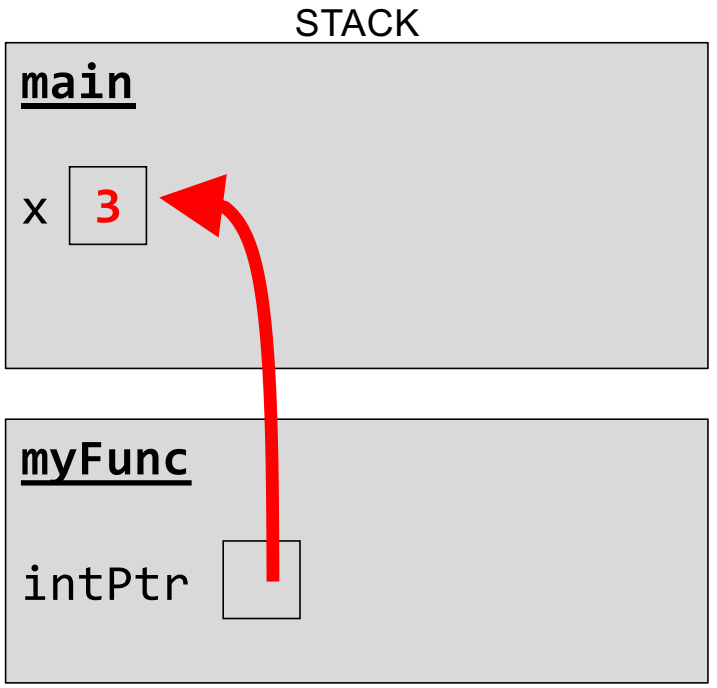
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

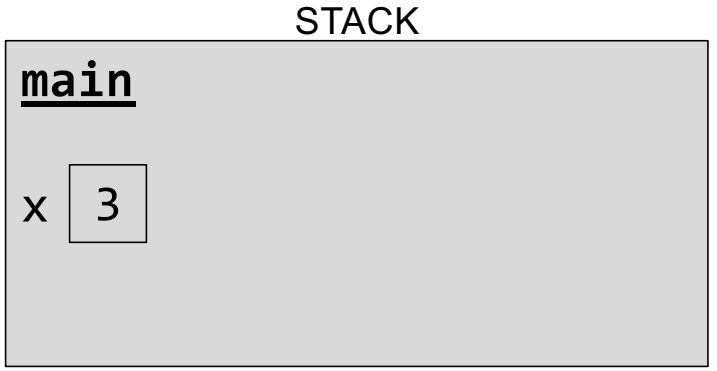
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

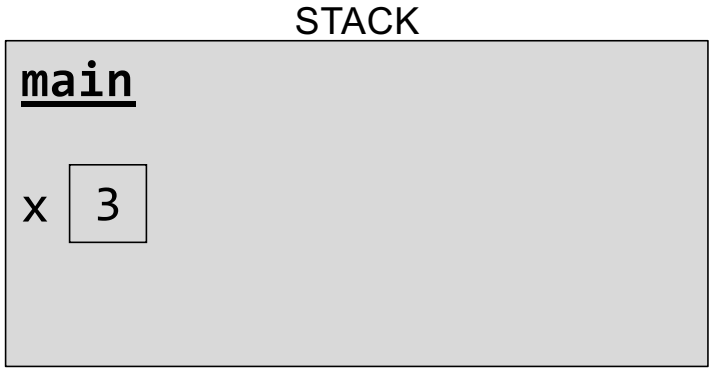
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

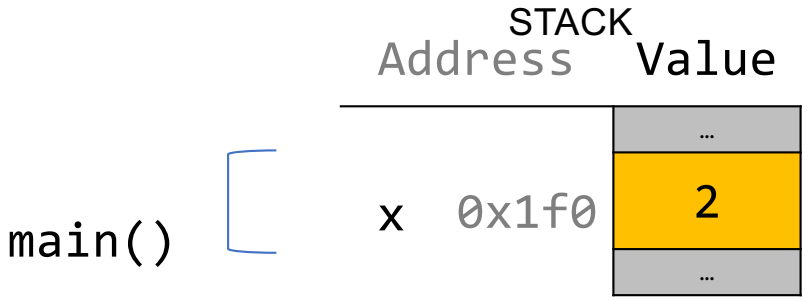
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

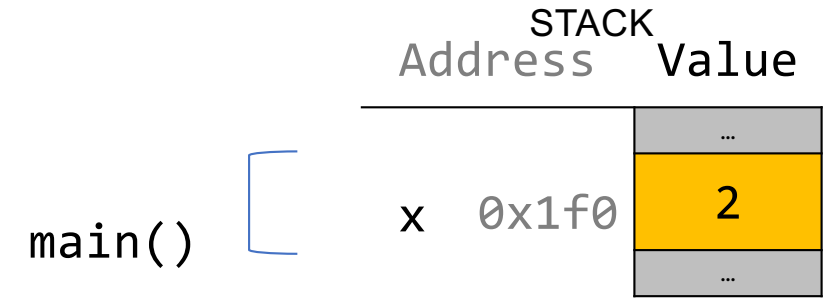
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

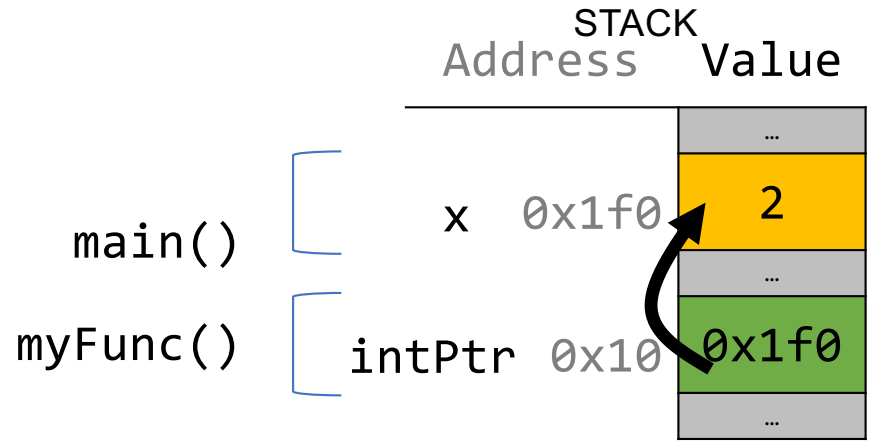
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

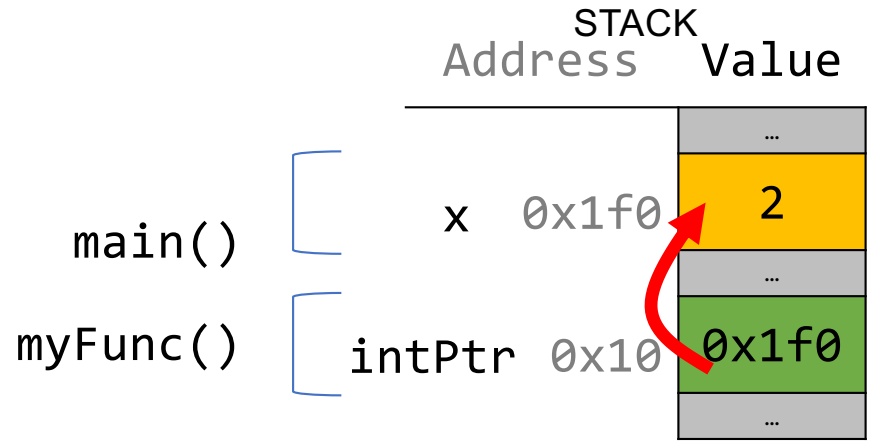
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

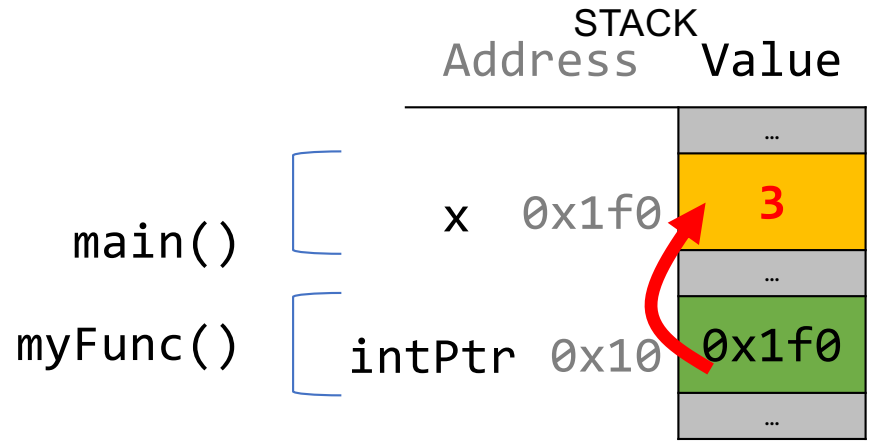
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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.

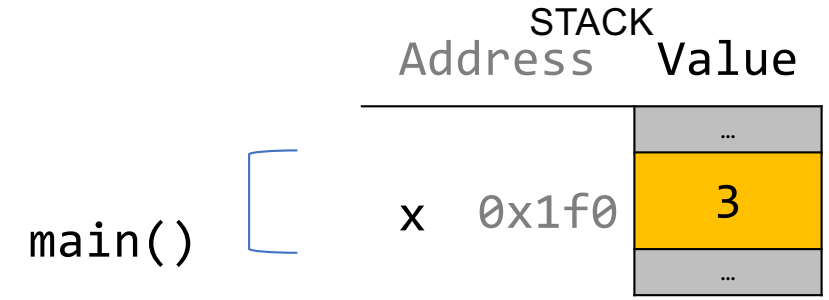
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

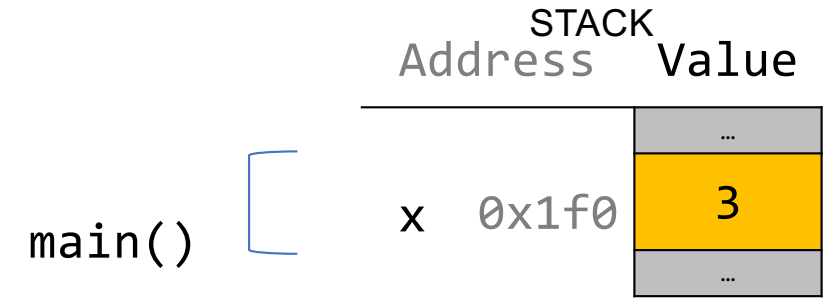
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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.

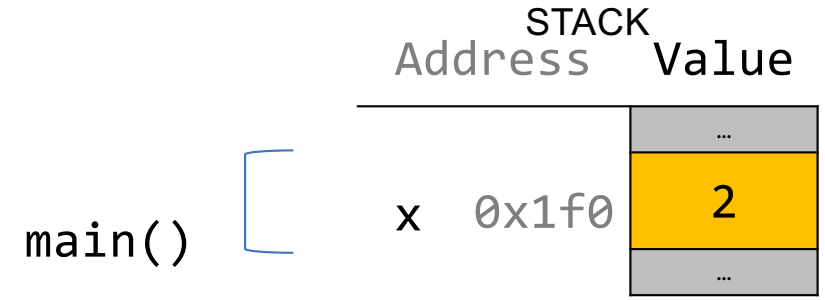
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

Without pointers, we would make copies.

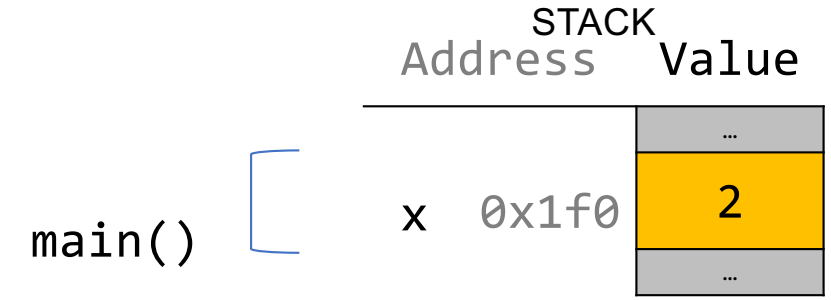
```
void myFunc(int val) {
     val = 3;
}

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



Pointers

Without pointers, we would make copies.

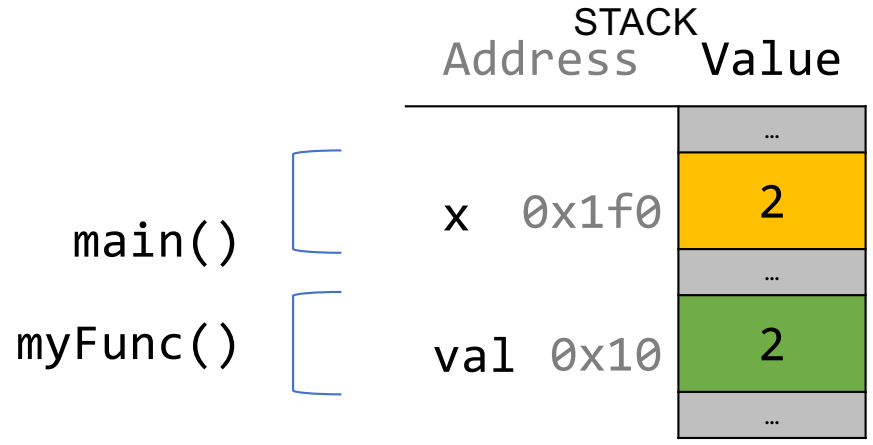
```
void myFunc(int val) {
     val = 3;
}

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



Pointers

Without pointers, we would make copies.

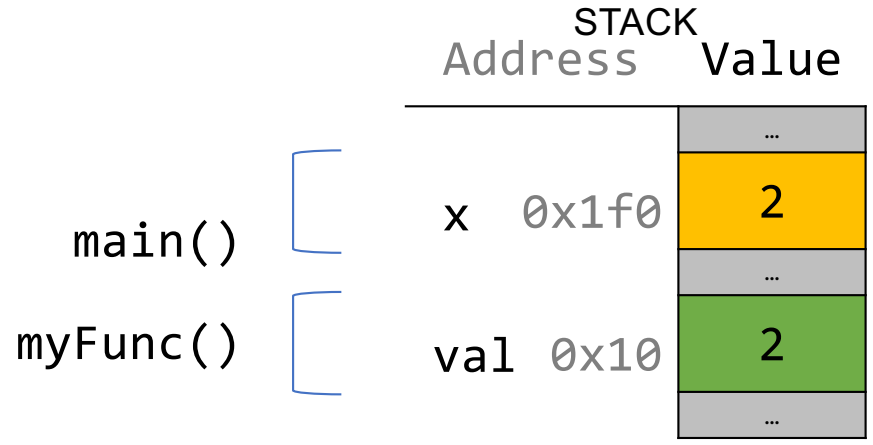
```
void myFunc(int val) {
     val = 3;
}

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



Pointers

Without pointers, we would make copies.

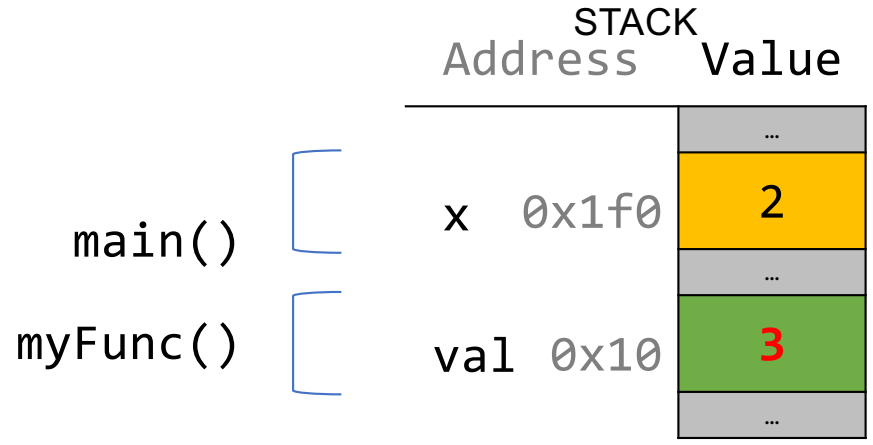
```
void myFunc(int val) {
      val = 3;
}

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



Pointers

Without pointers, we would make copies.

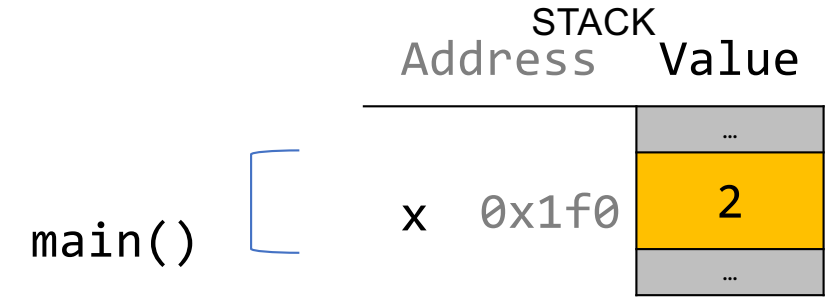
```
void myFunc(int val) {
      val = 3;
}

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



Pointers

Without pointers, we would make copies.

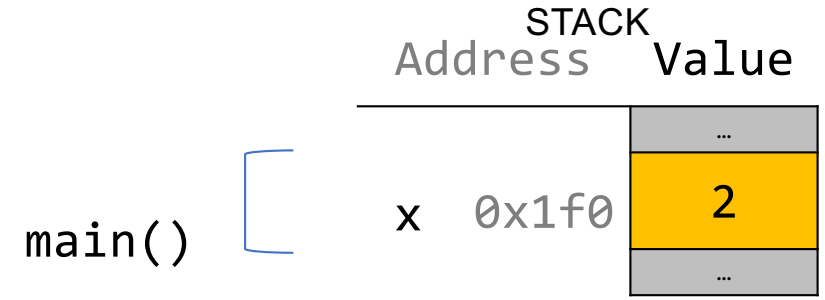
```
void myFunc(int val) {
     val = 3;
}

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



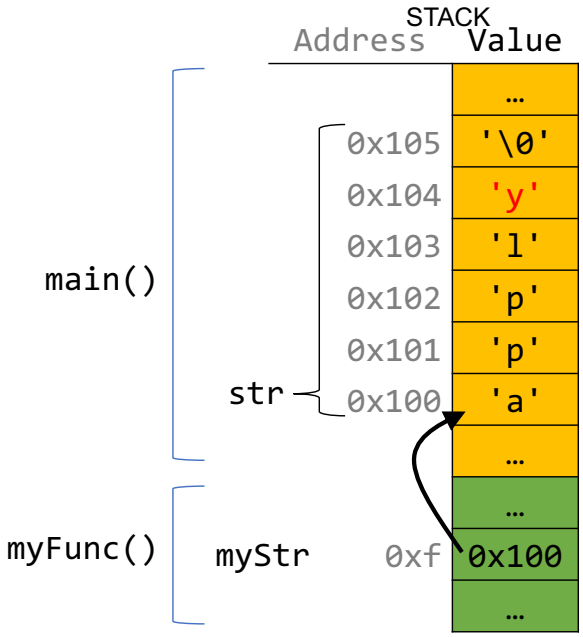
Pointers

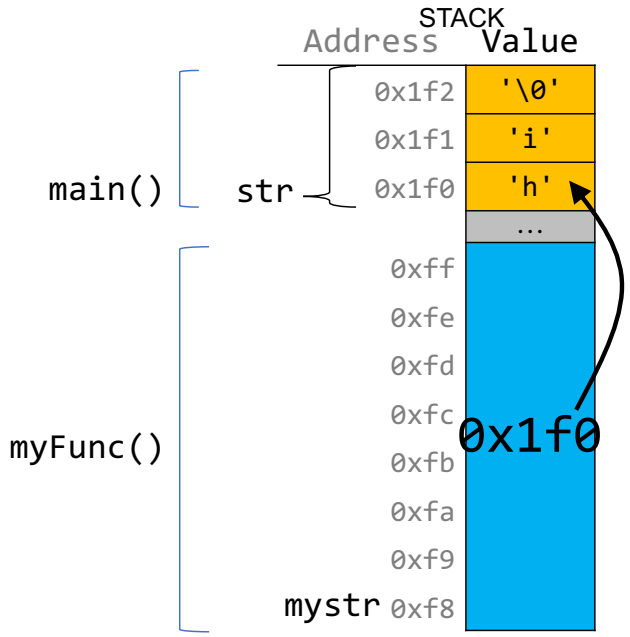
Without pointers, we would make copies.

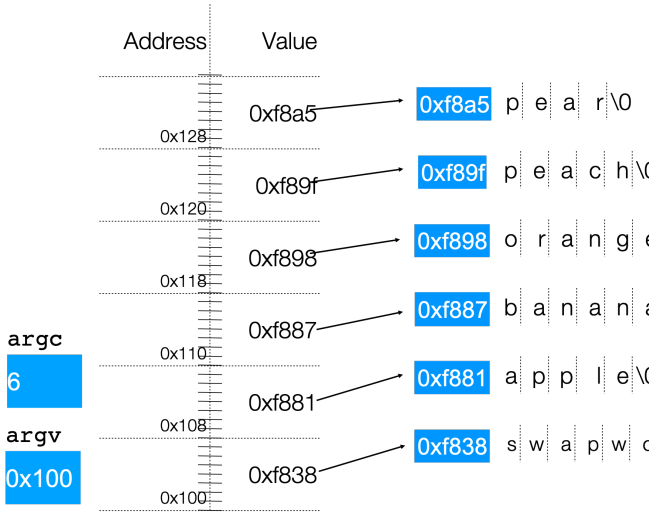
```
void myFunc(int val) {
     val = 3;
}

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



How to draw memory diagrams?







Choose whatever style is convenient for you, keeping in mind that (1) memory is contiguous, and (2) C types are different sizes.

C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```
void myFunction(int x) {
     …
}

int main(int argc, char *argv[]) {
     int num = 4;
     myFunction(num);        // passes copy of 4
}
```

C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```
void myFunction(int *x) {
     …
}

int main(int argc, char *argv[]) {
     int num = 4;
     myFunction(&num);        // passes copy of e.g. 0xffed63
}
```

C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```
void myFunction(char ch) {
     …
}

int main(int argc, char *argv[]) {
     char *myStr = "Hello!";
     myFunction(myStr[1]);        // passes copy of 'e'
}
```

C Parameters

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

Do I care about modifying *this* instance of my data? If so, I need to pass where that instance lives, as a parameter, so it can be modified.

Pointers

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

```
void capitalize(char *ch) {
     // modifies what is at the address stored in ch
}

int main(int argc, char *argv[]) {
     char letter = 'h';
     /* We don't want to capitalize any instance of 'h'.
      * We want to capitalize *this* instance of 'h'! */
     capitalize(&letter);
     printf("%c", letter);   // want to print 'H';
}
```

Pointers

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

```
void doubleNum(int *x) {
     // modifies what is at the address stored in x
}

int main(int argc, char *argv[]) {
     int num = 2;
     /* We don't want to double any instance of 2.
      * We want to double *this* instance of 2! */
     doubleNum(&num);
     printf("%d", num); // want to print 4;
}
```

Pointers

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```
void capitalize(char *ch) {
     // *ch gets the character stored at address ch.
     char newChar = toupper(*ch);

     // *ch = goes to address ch and puts newChar there.
     *ch = newChar;
}
```

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 an **char ****, it can modify the **char *** at the supplied address.