CS107, Lecture 8
C Strings, Valgrind and Pointers

Reading: K&R (5.2-5.5) or Essential C section 6

CS107 Topic 2

How can a computer represent and manipulate more complex data like text?

Why is answering this question important?

- Shows us how strings are represented in C and other languages (previously)
- Helps us better understand buffer overflows, a common bug (previously)
- Introduces us to pointers, because strings can be pointers (this time)

assign2: implement 2 functions a 1 program using those functions to find the location of different built-in commands in the filesystem. You'll write functions to extract a list of possible locations and tokenize that list of locations.

Learning Goals

- Learn more about the risks of buffer overflows and how to mitigate them
- Understand how strings are represented as pointers and how that helps us better understand their behavior
- Learn about pointers and how they help us access data without making copies
- Become familiar with using memory diagrams to understand code behavior

Lecture Plan

- Buffer Overflows, Security and Valgrind
- Debugging and Testing
- Review: Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

Lecture Plan

- **Buffer Overflows, Security and Valgrind**
- Debugging and Testing
- Review: Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

Buffer Overflow Impacts

Buffer overflows are not merely functionality bugs; they can cause a range of unintended behavior:

- Let the user access memory they shouldn't be able to access
- Let the user modify memory they shouldn't be able to access
  - Change a value that is used later in the program
  - User changes the program to execute their own custom instructions instead
  - And more…

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.

Association for Computing Machinery (ACM) Code of Ethics

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 applying the principle.

On This Pa

Preamble

1. GENERA

1.1 Contrib well-being, are stakeh

1.2 Avoid h

1.3 Be hon

1.4 Be fair discriminat

1.5 Respec new ideas,

https://www.acm.org/code-of-ethics

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.

https://www.acm.org/code-of-ethics

How can we fix buffer overflows?

There's no single solution to fix all buffer overflows; instead, it's a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How can we fix buffer overflows?

There's no single solution to fix all buffer overflows; instead, it's a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- **Thorough testing to uncover issues before release**
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How Can We Fix Overflows?

- **Valgrind**: Your Greatest Ally
- Write your own tests
- Consider writing tests *before* writing the main program

✨ `cs107.stanford.edu/testing.html` ✨

Buffer Overflows

- We must always ensure that memory operations we perform don't improperly read or write memory.
  - E.g. don't copy a string into a space that is too small!
  - E.g. don't ask for the string length of an uninitialized string!
- The **Valgrind** tool may be able to help track down memory-related issues.
  - See cs107.stanford.edu/resources/valgrind
  - We'll talk about Valgrind more when we talk about dynamically-allocated memory.
  - Valgrind can detect some, but not all, stack-memory-related issues

Demo: Memory Errors

`memory_errors.c`

Debugging Tools Summary

GDB

```
gdb myprogram
…
(gdb) run [args]
```

- Pause, step through, and print out values during program execution
- Helpful to track down bugs like crashes, infinite loops, functionality discrepancies, etc.

Valgrind

```
valgrind myprogram [args]
```

- Observes program execution without interrupting it, prints findings.
- Helpful (to a certain extent) to track down memory-related issues like buffer overflows, reading uninitialized memory, etc.

Lecture Plan

- Buffer Overflows, Security and Valgrind
- **Debugging and Testing**
- **Review:** Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

Debugging

GDB and Valgrind will be essential tools throughout the quarter to find and squash bugs.

- Emphasis on both tools and the debugging process: http://cs107.stanford.edu/resources/debugging.html
  - We'll be relying on this checklist going forward for debugging – use the tips here as you work and come ask questions!
- Debugging can certainly be frustrating sometimes! Checklist emphasizes a methodical and systematic process that will save time in the long run.
- Debugging also where we learn a lot about how our code works

How do you find bugs? **Testing!**

Testing

Testing helps surface bugs and track progress in implementing a program.

Opaque-box testing: writing tests only considering the specification of what the program should do, without considering the implementation details/actual code

Clear-box testing: writing tests relying on knowledge of the design internals and code paths such as internal special cases, code structure, etc.

http://cs107.stanford.edu/testing.html

- *Test case size*: small tests help early on to catch bugs; large tests help later to stress test the system.

Aim to write tests throughout the development of a program.

- **Before writing the program:** documents intended behavior, avoids code assumptions
- **During writing the program:** add additional test cases as needed

Goal: incremental development, testing at each step.

Lecture Plan

- Buffer Overflows, Security and Valgrind
- Debugging and Testing
- **Review: Pointers**

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

Strings and Pointers

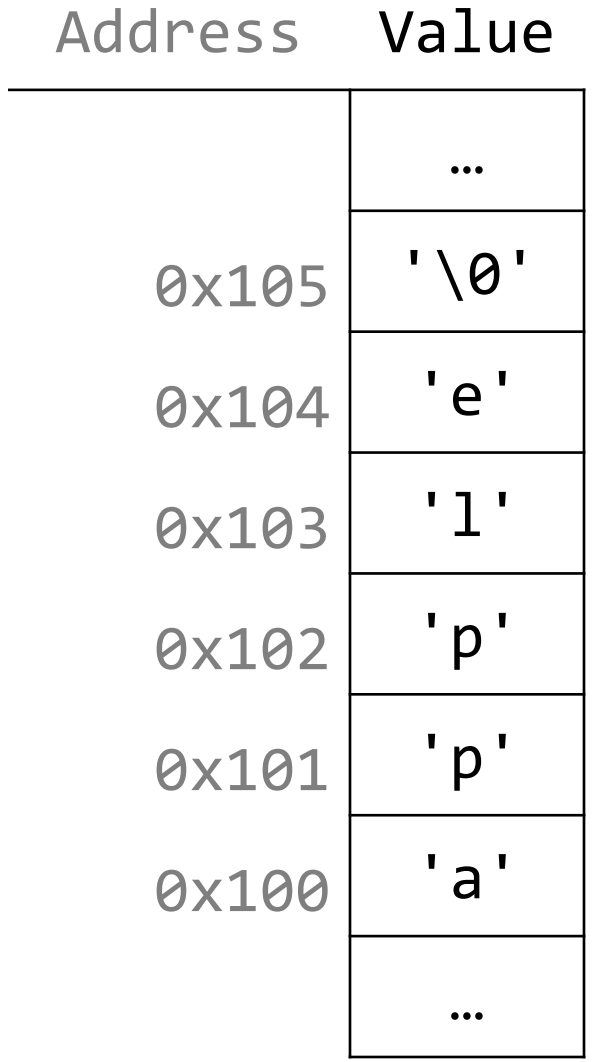
C strings can be represented as **char[]** or **char ***.

- When we create **char[]**, we are creating space for characters
- When we create **char ***, we are creating space for an address of character(s)
- Strings are implicitly converted to char * when passed as parameters
  - E.g. all string functions take char * parameters, but accept char[]
- A char * is technically a pointer to a **single character**. But we commonly use char * as string by having the character it points to be followed by more characters and ultimately a null terminator. But a char * could also just point to a single character (not a string).

Pointers and Memory

A *pointer* is a variable that stores a memory address.

- Memory is a big array of bytes, and each byte has a unique numeric index that is commonly written in hexadecimal. A pointer stores one of these "indexes".
- Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
- Pointers are also essential for allocating memory on the heap, and to refer to memory generically, both of which we will cover later.

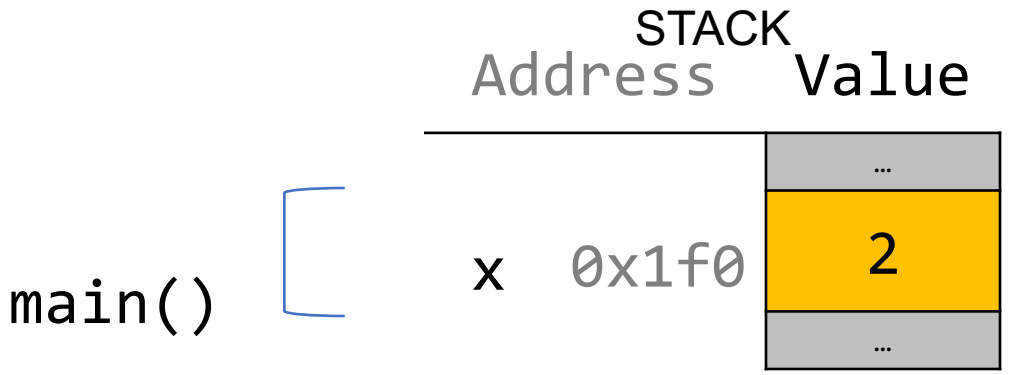


Pass By Value

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

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

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

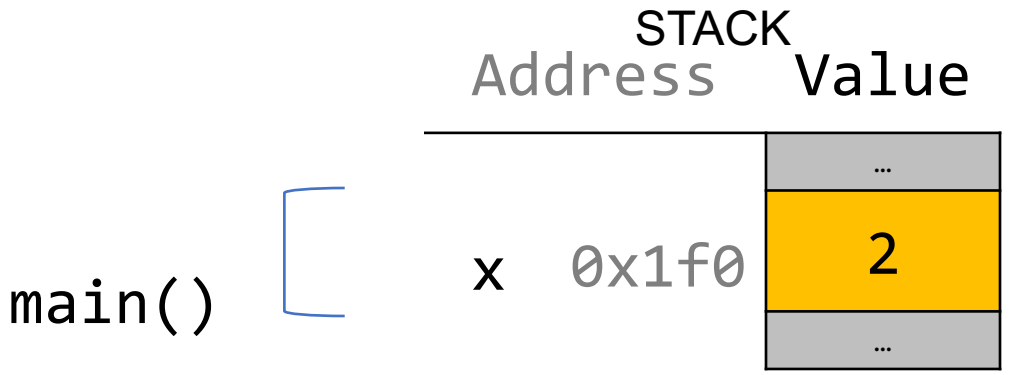


Pass By Value

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

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

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

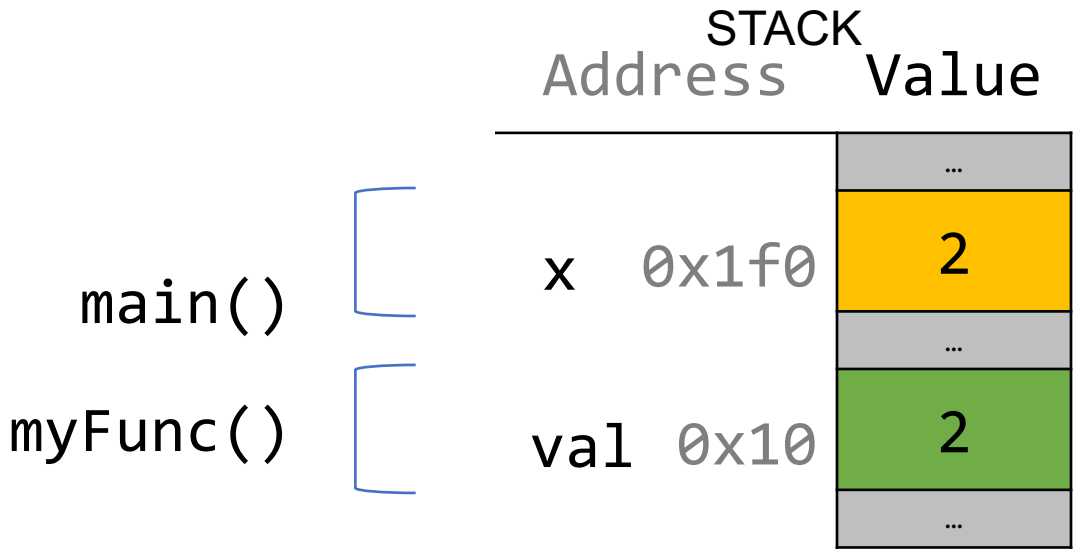


Pass By Value

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

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

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

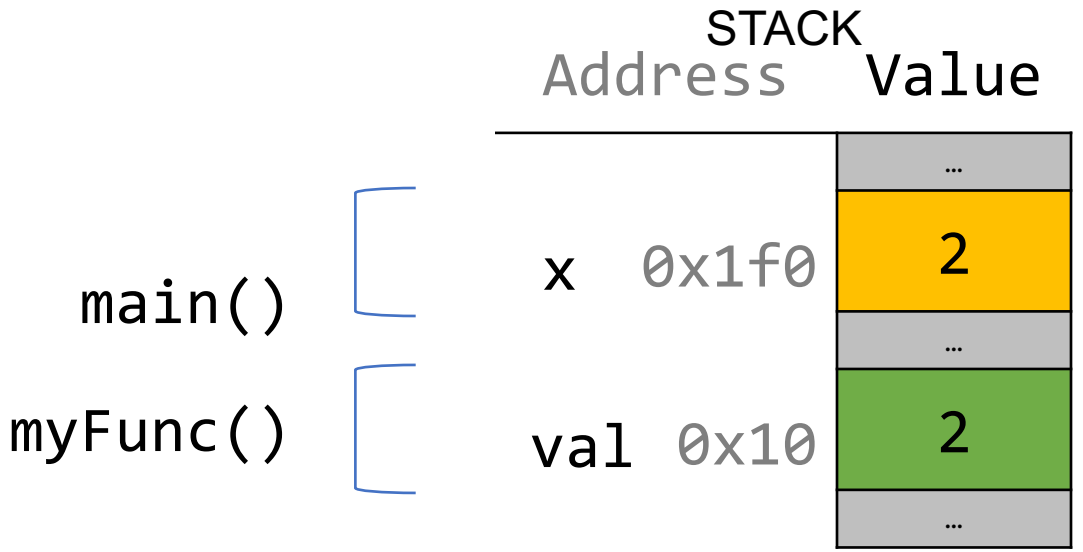


Pass By Value

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

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

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

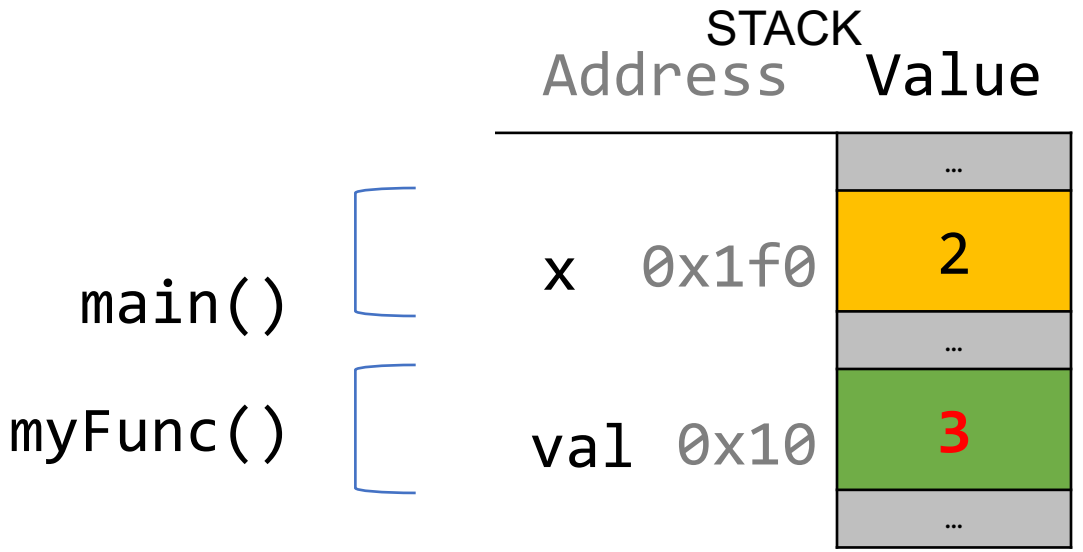


Pass By Value

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

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

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

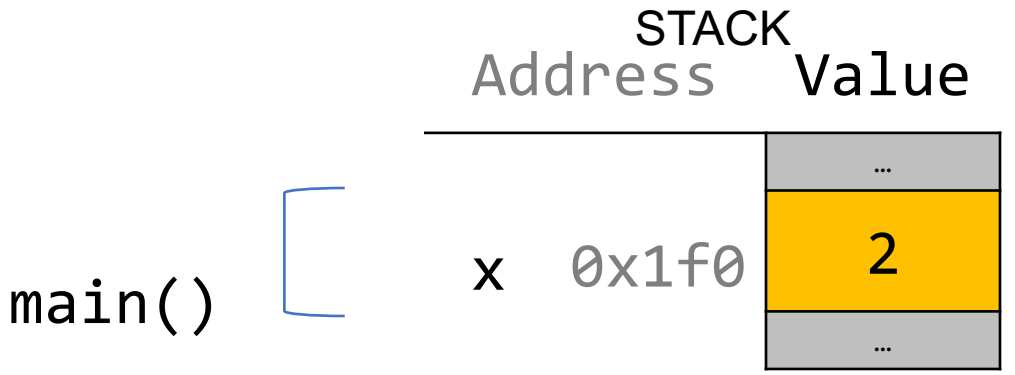


Pass By Value

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

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

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



Pointers

Pointers allow us to pass around the *location* of data so that the original data can be modified in other functions.

Example: I want to write a function *myFunc* that can change the value of an existing integer to be 3.

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

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
```

`*`

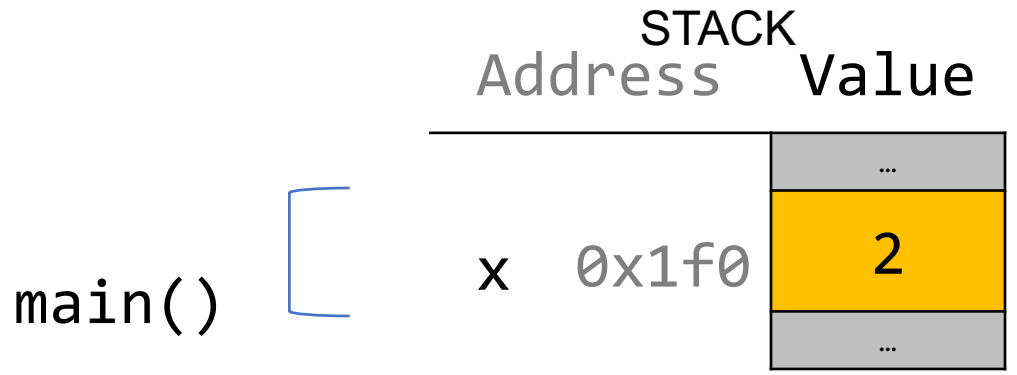
If **declaration**: “pointer”
ex: `int *` is "pointer to an int”
If **operation**: "dereference/the value at address”
ex: `*num` is "the value at address num"

Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

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

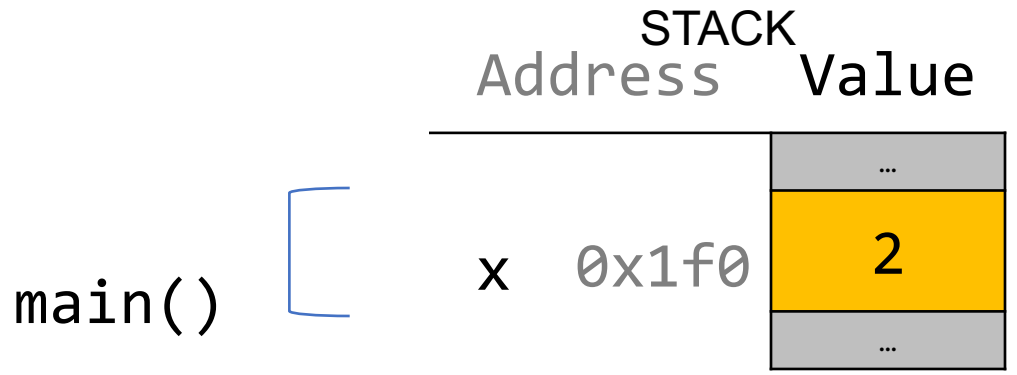


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

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

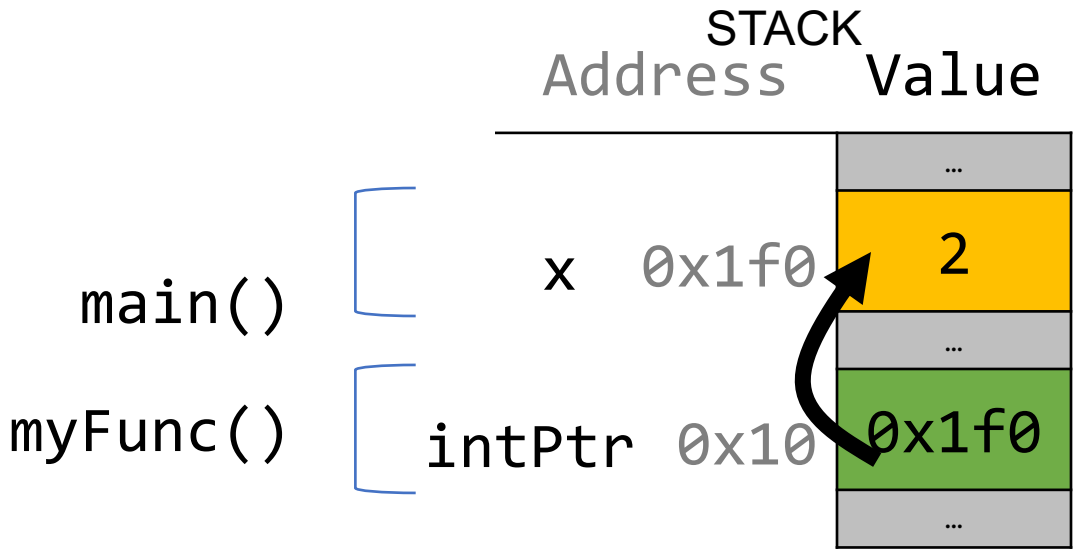


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

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

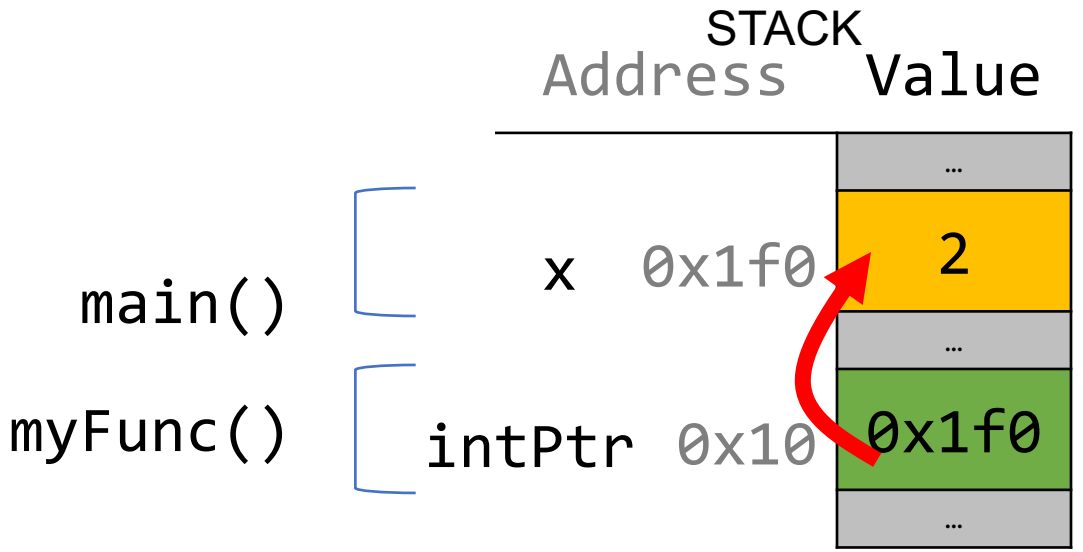


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

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

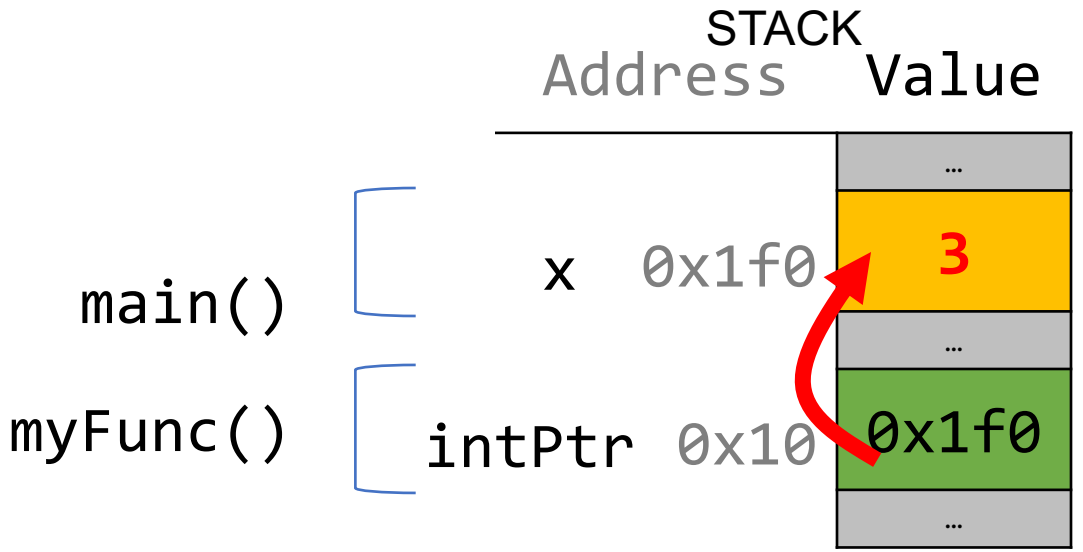


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

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

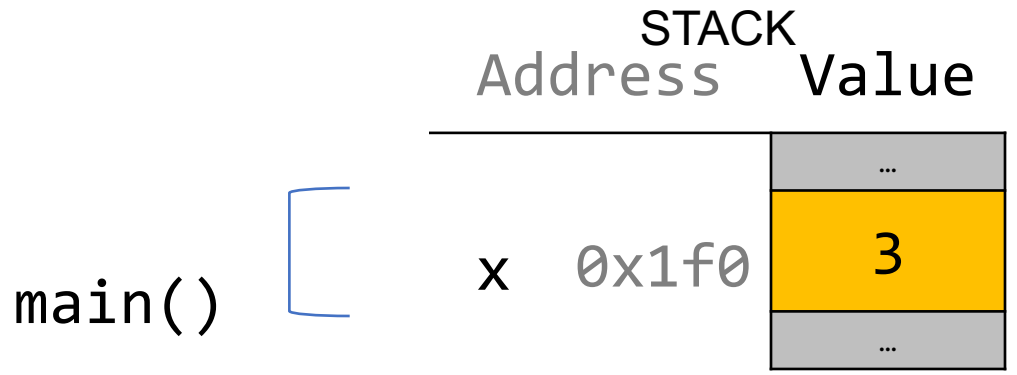


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

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



C Parameters

- 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 and dereference that location to access what's there.

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 Practice

```
void makeUpper(char *ptr) {
        __1__ = toupper(__2__);
}

int main(int argc, char *argv[]) {
        char ch = 'h';

        // want to modify ch to be capital
        makeUpper(__3__);
        printf("%c\n", ch); // should print 'H'
        return 0;
}
```

What should go in each of the blanks so that this code correctly modifies **ch** to be capitalized?

Respond on PollEv: pollev.com/cs107 or text CS107 to 22333 once to join.

What should go in each of the blanks so that this code correctly modifies ch to be capitalized?

Pointers Practice

```
void makeUpper(char *ptr) {
      *ptr = toupper(*ptr);
}

int main(int argc, char *argv[]) {
      char ch = 'h';

      // want to modify ch to be capital
      makeUpper(&ch);
      printf("%c\n", ch); // should print 'H'
      return 0;
}
```

What should go in each of the blanks so that this code correctly modifies **ch** to be capitalized?

Recap

- Buffer Overflows, Security and Valgrind
- Debugging and Testing
- Review: Pointers

Lecture 8 takeaway: C strings are pointers and arrays. C strings are error-prone, and issues like buffer overflows can arise! Valgrind is a tool that can help detect memory errors.

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```