

# COMP201

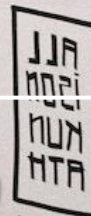
## Computer Systems & Programming

Lecture #08 – More Strings, Pointers



KOÇ  
UNIVERSITY

Aykut Erdem // Koç University // Fall 2020



# Recap

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings

# Plan for Today

- String Diamond
- Searching in Strings
- Pointers

**Disclaimer:** Slides for this lecture were borrowed from  
—Nick Troccoli's Stanford CS107 class

# Lecture Plan

- String Diamond
- Searching in Strings
- Pointers

# String Diamond

- Write a function **diamond** that accepts a string parameter and prints its letters in a "diamond" format as shown below.
  - For example, `diamond("DAISY")` should print:

```
D
DA
DAI
DAIS
DAISY
 AISY
  ISY
   SY
    Y
```



# Practice: Diamond



```
cp -r /afs/ir/class/cs107/lecture-code/lect4 .
```

# Lecture Plan

- String Diamond
- Searching in Strings
- Pointers

# C Strings

C strings are arrays of characters ending with a **null-terminating character** `'\0'`.

<i>index</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>value</i>	'H'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

String operations such as `strlen` use the null-terminating character to find the end of the string.

**Side note:** use `strlen` to get the length of a string. Don't use `sizeof`!



# Common string.h Functions

Function	Description
strlen( <i>str</i> )	returns the # of chars in a C string (before null-terminating character).
strcmp( <i>str1</i> , <i>str2</i> ), strncmp( <i>str1</i> , <i>str2</i> , <i>n</i> )	compares two strings; returns 0 if identical, <0 if <b><i>str1</i></b> comes before <b><i>str2</i></b> in alphabet, >0 if <b><i>str1</i></b> comes after <b><i>str2</i></b> in alphabet. <b>strncmp</b> stops comparing after at most <i>n</i> characters.
strchr( <i>str</i> , <i>ch</i> ) strrchr( <i>str</i> , <i>ch</i> )	character search: returns a pointer to the first occurrence of <b><i>ch</i></b> in <b><i>str</i></b> , or <b>NULL</b> if <b><i>ch</i></b> was not found in <b><i>str</i></b> . <b>strrchr</b> find the last occurrence.
strstr( <i>haystack</i> , <i>needle</i> )	string search: returns a pointer to the start of the first occurrence of <b><i>needle</i></b> in <b><i>haystack</i></b> , or <b>NULL</b> if <b><i>needle</i></b> was not found in <b><i>haystack</i></b> .
strcpy( <i>dst</i> , <i>src</i> ), strncpy( <i>dst</i> , <i>src</i> , <i>n</i> )	copies characters in <b><i>src</i></b> to <b><i>dst</i></b> , including null-terminating character. Assumes enough space in <b><i>dst</i></b> . Strings must not overlap. <b>strncpy</b> stops after at most <i>n</i> chars, and <u>does not</u> add null-terminating char.
strcat( <i>dst</i> , <i>src</i> ), strncat( <i>dst</i> , <i>src</i> , <i>n</i> )	concatenate <b><i>src</i></b> onto the end of <b><i>dst</i></b> . <b>strncat</b> stops concatenating after at most <i>n</i> characters. <u>Always</u> adds a null-terminating character.
strspn( <i>str</i> , <i>accept</i> ), strcspn( <i>str</i> , <i>reject</i> )	<b>strspn</b> returns the length of the initial part of <b><i>str</i></b> which contains <u>only</u> characters in <b><i>accept</i></b> . <b>strcspn</b> returns the length of the initial part of <b><i>str</i></b> which does <u>not</u> contain any characters in <b><i>reject</i></b> .

# Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or `NULL` if the character is not in the string.

```
char daisy[6];  
strcpy(daisy, "Daisy");  
char *letterA = strchr(daisy, 'a');  
printf("%s\n", daisy);           // Daisy  
printf("%s\n", letterA);        // aisy
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the *first* one. Use `strrchr` to obtain a pointer to the *last* occurrence.

# Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or `NULL` if it cannot be found.

```
char daisy[10];  
strcpy(daisy, "Daisy Dog");  
char *substr = strstr(daisy, "Dog");  
printf("%s\n", daisy);           // Daisy Dog  
printf("%s\n", substr);         // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

# String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char daisy[10];  
strcpy(daisy, "Daisy Dog");  
int spanLength = strspn(daisy, "aDeoi");           // 3
```

**“How many places can we go in the first string before I encounter a character not in the second string?”**

# String Spans

`strcspn` (`c = "complement"`) returns the *length* of the initial part of the first string which contains only characters not in the second string.

```
char daisy[10];  
strcpy(daisy, "Daisy Dog");  
int spanLength = strcspn(daisy, "driso");           // 2
```

**“How many places can we go in the first string before I encounter a character in the second string?”**

# C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char \***. We can still operate on the string the same way as with a **char[]**. (*We'll see why today!*).

```
int doSomething(char *str) {  
    char secondChar = str[1];  
    ...  
}
```

*// can also write this, but it is really a pointer*

```
int doSomething(char str[]) { ...
```

# Arrays of Strings

We can make an array of strings to group multiple strings together:

```
char *stringArray[5];    // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```
char *stringArray[] = {  
    "Hello",  
    "Hi",  
    "Hey there"  
};
```



# Arrays of Strings

We can access each string using bracket syntax:

```
printf("%s\n", stringArray[0]); // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the first element of the array*. This is what **argv** is in **main**! This means we write the parameter type as:

```
void myFunction(char **stringArray) {
```

```
// equivalent to this, but it is really a double pointer
```

```
void myFunction(char *stringArray[]) {
```

# Practice: Password Verification

Write a function **verifyPassword** that accepts a candidate password and certain password criteria and returns whether the password is valid.

```
bool verifyPassword(char *password, char *validChars,  
char *badSubstrings[], int numBadSubstrings);
```

**password** is valid if it contains only letters in **validChars**, and does not contain any substrings in **badSubstrings**.

# Practice: Password Verification

```
bool verifyPassword(char *password, char *validChars, char  
*badSubstrings[], int numBadSubstrings);
```

## Example:

```
char *invalidSubstrings[] = { "1234" };
```

```
bool valid1 = verifyPassword("1572", "0123456789",  
    invalidSubstrings, 1);           // true
```

```
bool valid2 = verifyPassword("141234", "0123456789",  
    invalidSubstrings, 1);           // false
```

# Practice: Password Verification



```
verify_password.c
```

# Lecture Plan

- String Diamond
- Searching in Strings
- Pointers

# Pointers

- A *pointer* is a variable that stores a memory address.
- 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.
- One (8 byte) pointer can refer to any size memory location!
- Pointers are also essential for allocating memory on the heap, which we will cover later.
- Pointers also let us refer to memory generically, which we will cover later.

# Memory

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

Address	Value
	...
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	...



# Memory

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

Address	Value
	...
261	'\0'
260	'e'
259	'l'
258	'p'
257	'p'
256	'a'
	...

# 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 “pass by value.” For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.
- This means whenever we pass something as a parameter, we pass a copy.
- If we want to modify a parameter value in the function we call and have the changes persist afterwards, we can pass the location of the value instead of the value itself. This way we make a copy of the *address* instead of a copy 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
```

# Pointers

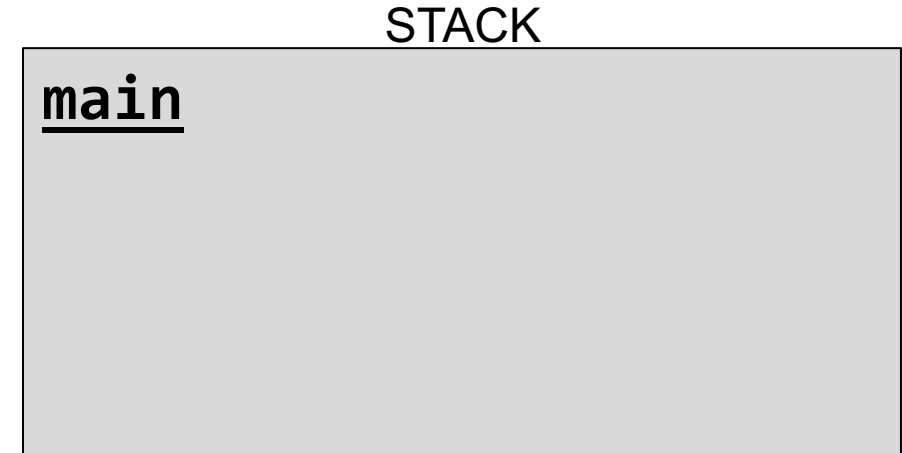
A pointer is a variable that stores a memory address.

```
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!  
    ...  
}
```

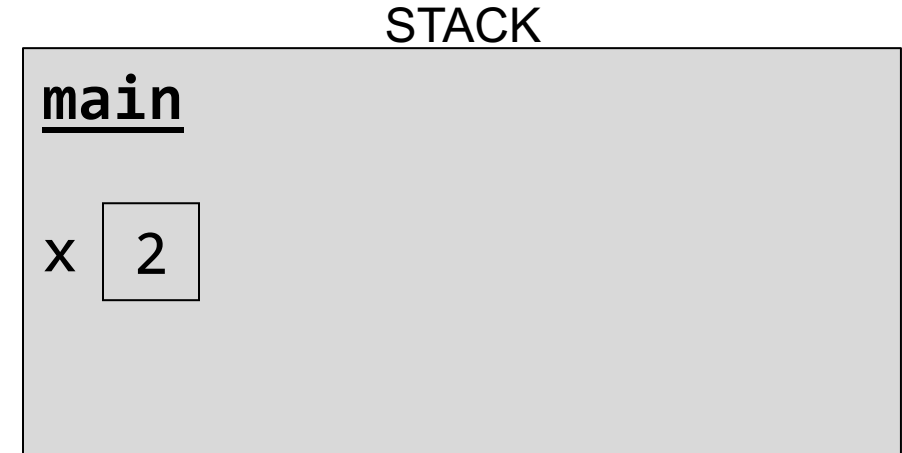


# Pointers

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!  
    ...  
}
```



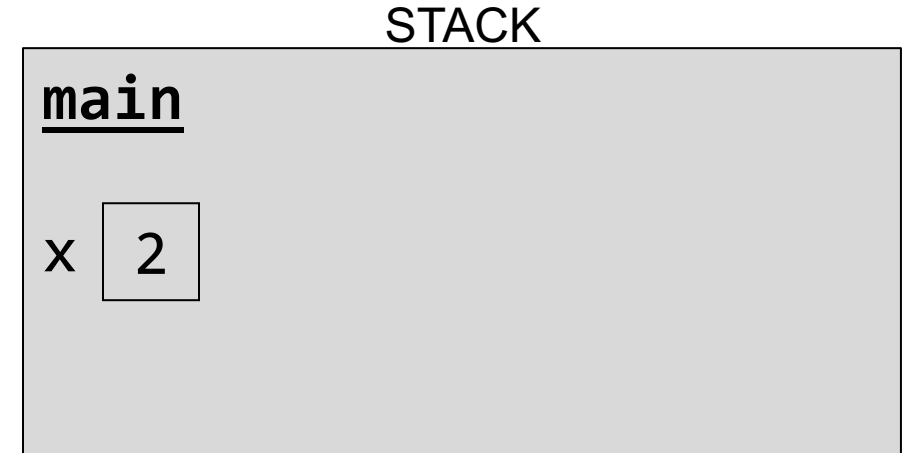


# Pointers

A pointer is a variable that stores a memory address.

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

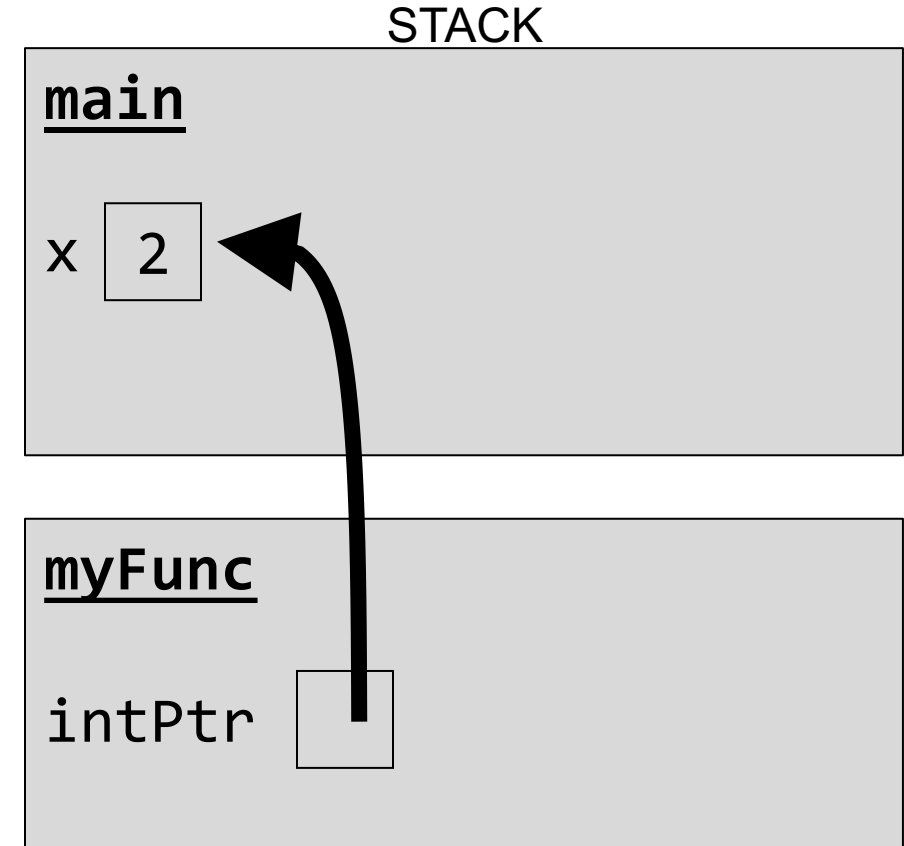
```
int main(int argc, char *argv[]) {  
    int x = 2;  
    myFunc(&x);  
    printf("%d", x);    // 3!  
    ...  
}
```



# Pointers

A pointer is a variable that stores a memory address.

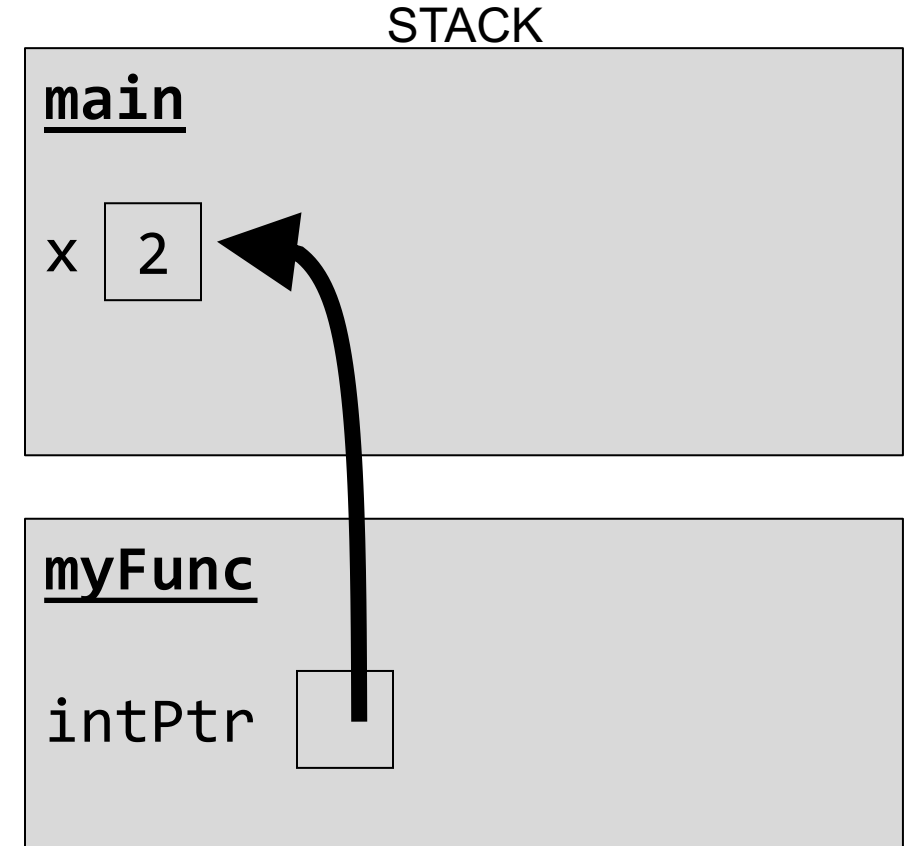
```
void myFunc(int *intPtr) {  
    *intPtr = 3;  
}  
  
int main(int argc, char *argv[]) {  
    int x = 2;  
    myFunc(&x);  
    printf("%d", x);    // 3!  
    ...  
}
```



# Pointers

A pointer is a variable that stores a memory address.

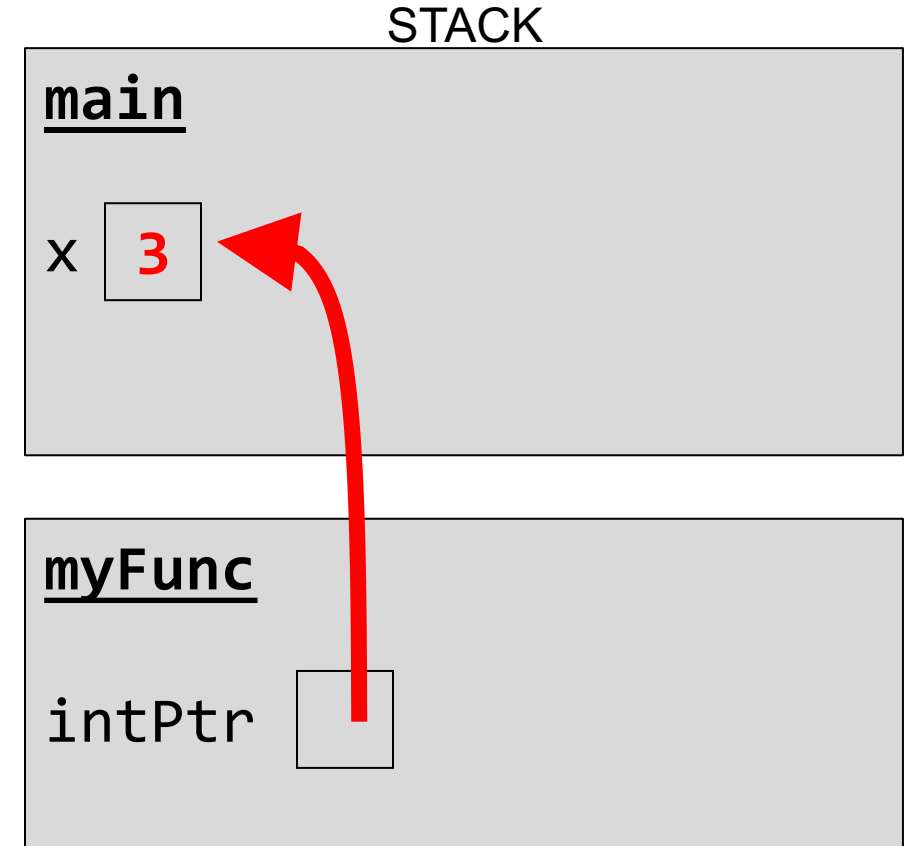
```
void myFunc(int *intPtr) {  
    *intPtr = 3;  
}  
  
int main(int argc, char *argv[]) {  
    int x = 2;  
    myFunc(&x);  
    printf("%d", x);    // 3!  
    ...  
}
```



# Pointers

A pointer is a variable that stores a memory address.

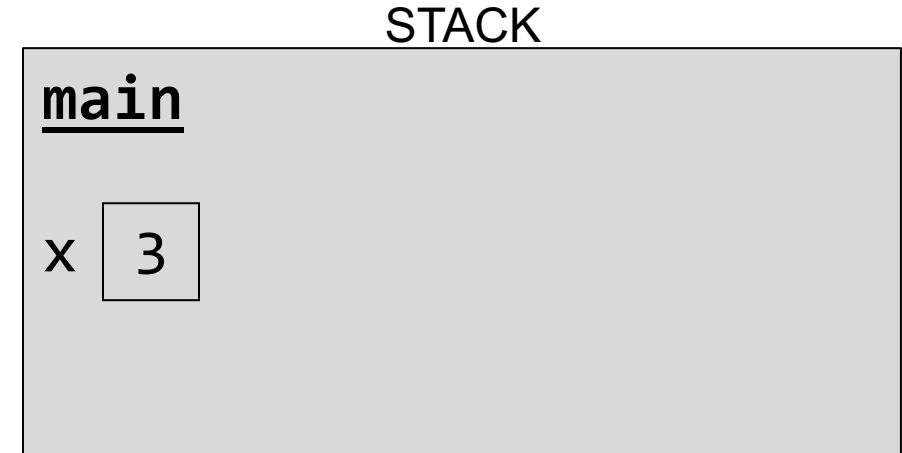
```
void myFunc(int *intPtr) {  
    *intPtr = 3;  
}  
  
int main(int argc, char *argv[]) {  
    int x = 2;  
    myFunc(&x);  
    printf("%d", x);    // 3!  
    ...  
}
```



# 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!  
    ...  
}
```

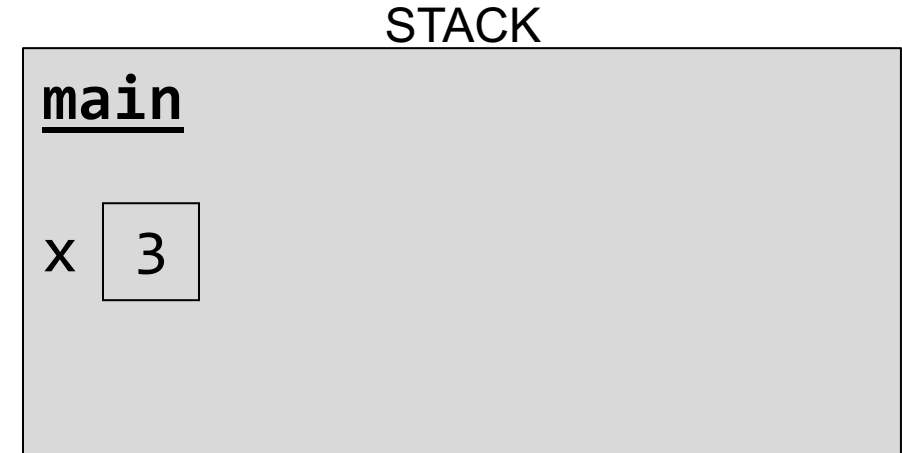


# Pointers

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!  
    ...  
}
```

main()



STACK	
Address	Value
x    0x1f0	...
	2
	...



# 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!  
    ...  
}
```

main()



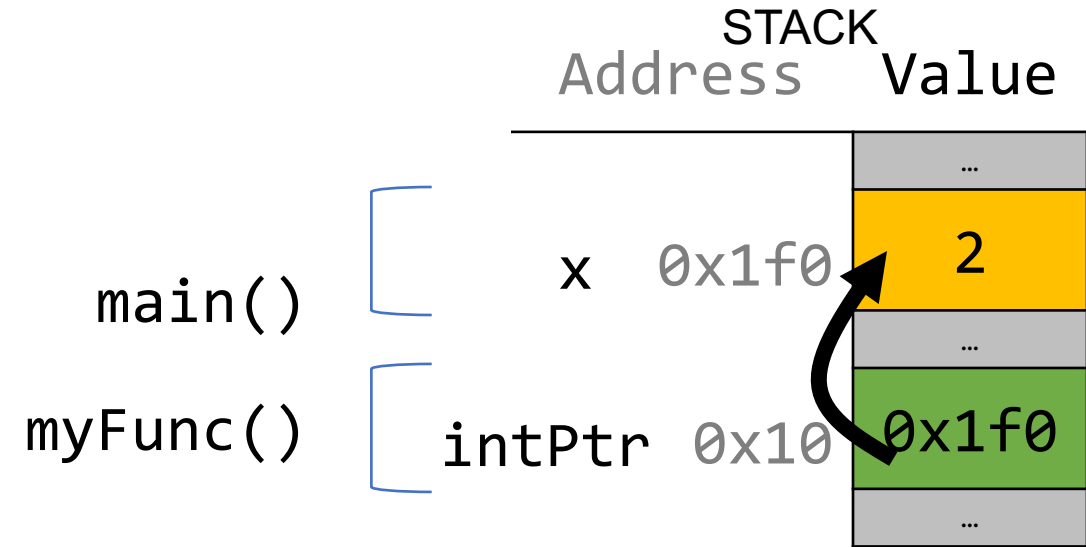
STACK	
Address	Value
x    0x1f0	...
	2
	...

# 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!  
    ...  
}
```

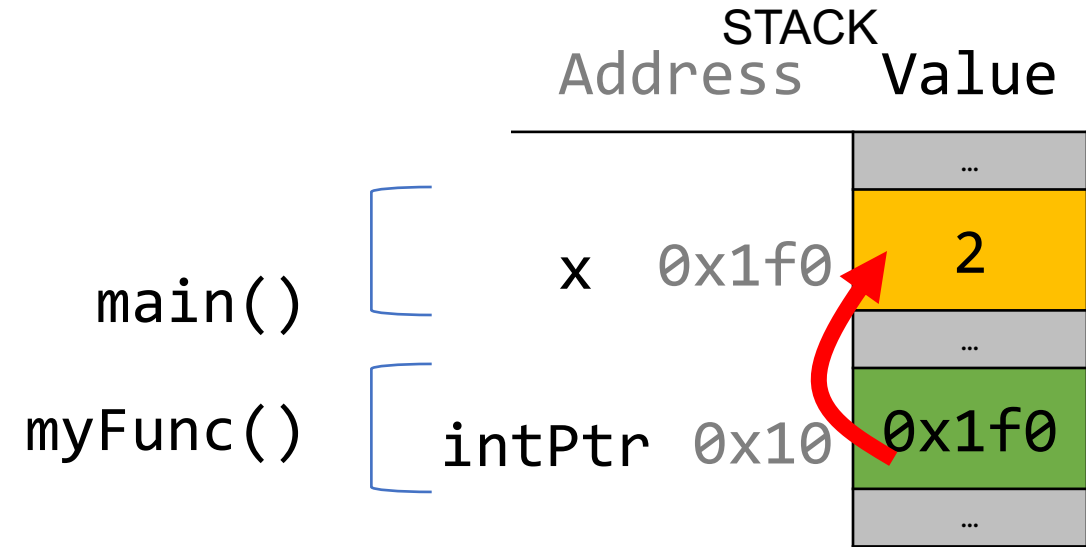


# Pointers

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!  
    ...  
}
```

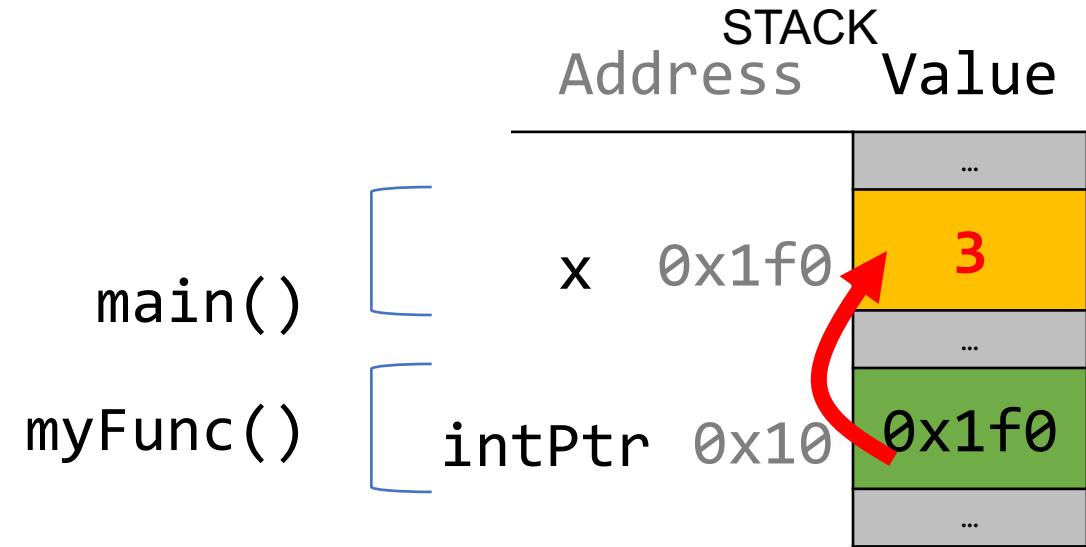


# Pointers

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!  
    ...  
}
```

main()



STACK	
Address	Value
x    0x1f0	...
	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!  
    ...  
}
```

main()



STACK	
Address	Value
x	...
	3
	...

# 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**. This makes a copy of the data.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify. This makes a copy of the data's location.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

# 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!  
    ...  
}
```

main()



STACK		
Address		Value
		...
x	0x1f0	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!  
    ...  
}
```

main()



STACK		
Address		Value
<hr/>		
		...
x	0x1f0	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!  
    ...  
}
```

main()  
myFunc()

STACK		Address	Value
		x	0x1f0
		val	0x10

# 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!  
    ...  
}
```

main()  
myFunc()

STACK		Address	Value
		x	0x1f0
		val	0x10

# Pointers

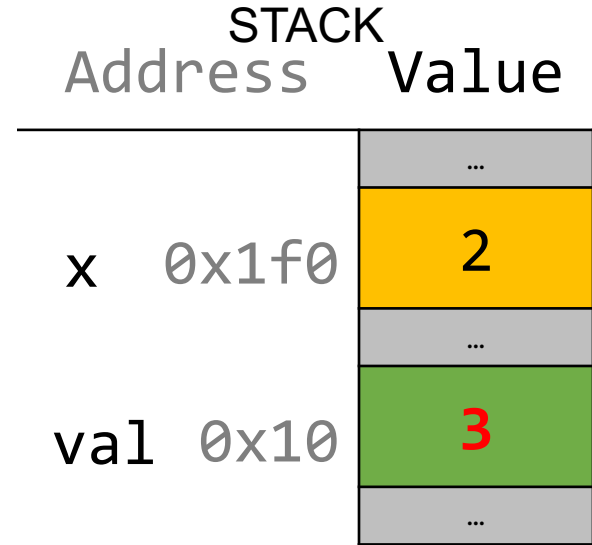
Without pointers, we would make copies.

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

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

main()  
myFunc()

STACK		Address	Value
		x	0x1f0
		val	0x10



# 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!  
    ...  
}
```

main()



STACK		
Address		Value
<hr/>		
		...
x	0x1f0	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!  
    ...  
}
```

main()



STACK		
Address		Value
<hr/>		
		...
x	0x1f0	2
		...

# Recap

- String Diamond
- Searching in Strings
- Pointers

**Next time:** Strings in Memory