CS107, Lecture 17

Final Exam Review Session

Reading: None!

Based on slides created by Joel Ramirez, Nick Troccoli, Chris Gregg

Lecture Plan

- End of quarter logistics
- Bits and Bytes 
- Chars and C-Strings 
- Pointers, Stack, and Heap 
- Generics 
- Assembly 
- Heap Allocators 

End of Quarter Logistics

- Final Exam: **Friday, August 14th 3:30-6:30PM in CoDa B90**
  - Folks with alternate exam accommodations will receive more info soon
  - Open book, closed electronic
- Assign6 checkpoint due **Sunday, August 9th at 11:59PM** and entire assignment due **Thursday, August 13th at 11:59PM**
  - Start early if you haven't already! This tends to be a heftier assignment.
- Board game hangout during lecture on Friday (completely optional)!
- Extra Credit quiz today – end of lecture since content in this lecture will likely help!
- No lecture next week; OH continue M/W regularly

Bits and Bytes 

Bits and Bytes Topics

- Hexadecimal
- Binary
- Two's complement
- Overflow
- `sizeof` operator
- Little/big endian
- Expanding/truncating bit representation
- Bitwise operators: |, &, ~, ^, <<, >>
- Arithmetic vs. logical right shifts
- Bitmasks (creating them, using them)

Chars and C-Strings 

Chars and C-Strings Topics

- ASCII encoding of characters
- C-strings
- Null terminator
- `strlen()` and other `string.h` functions
- `strcpy()` to copy strings NOT =!!!!
- Buffer overflows
- `char * ` vs `char[]`
- Strings in memory behavior (Lecture 5, slide 48)
  - Characters living in modifiable memory (stack, heap) vs read only memory (data segment)

Chars and C-Strings

1. If we create a string as a **`char[ ]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[ ]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[ ]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`** .
4. If we create a new string with new characters as a **`char *`** , we cannot modify its characters because its memory lives in the data segment.
5. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
6. Adding an offset to a C string gives us a substring that many places past the first character.
7. If we change characters in a string parameter, these changes will persist outside of the function.

Chars and C-Strings

At the end of the day: where do the characters pointed at/in the array live?

- In the **read-only data segment**? **Can't modify.**
- On the **stack**? **Can modify.**
- On the **heap**? **Can modify.**

Pointers, Stack, and Heap 

Pointers, Stack, and Heap Topics

- Pointers
- “Passing by reference” in C: passing the location (Add an extra *!)
- Behavior of strings in memory
  - Read-only data segment (where string literals get stored)
- Pointer arithmetic
- `sizeof` on arrays
- `const` keyword – read right to left!
- Stack
- Stack frames
- Heap
- `malloc/free/realloc`
- Memory leaks
- Use after free

Passing by Reference in C

- Any parameter passed will be COPIED → need to pass data's address, not the data itself!
- Examples:
  - Quiz 4, Question 2
  - Quiz 5, Question 2
    - Answer key has some potentially useful diagrams to illustrate this
  - Midterm, Question 4(c)

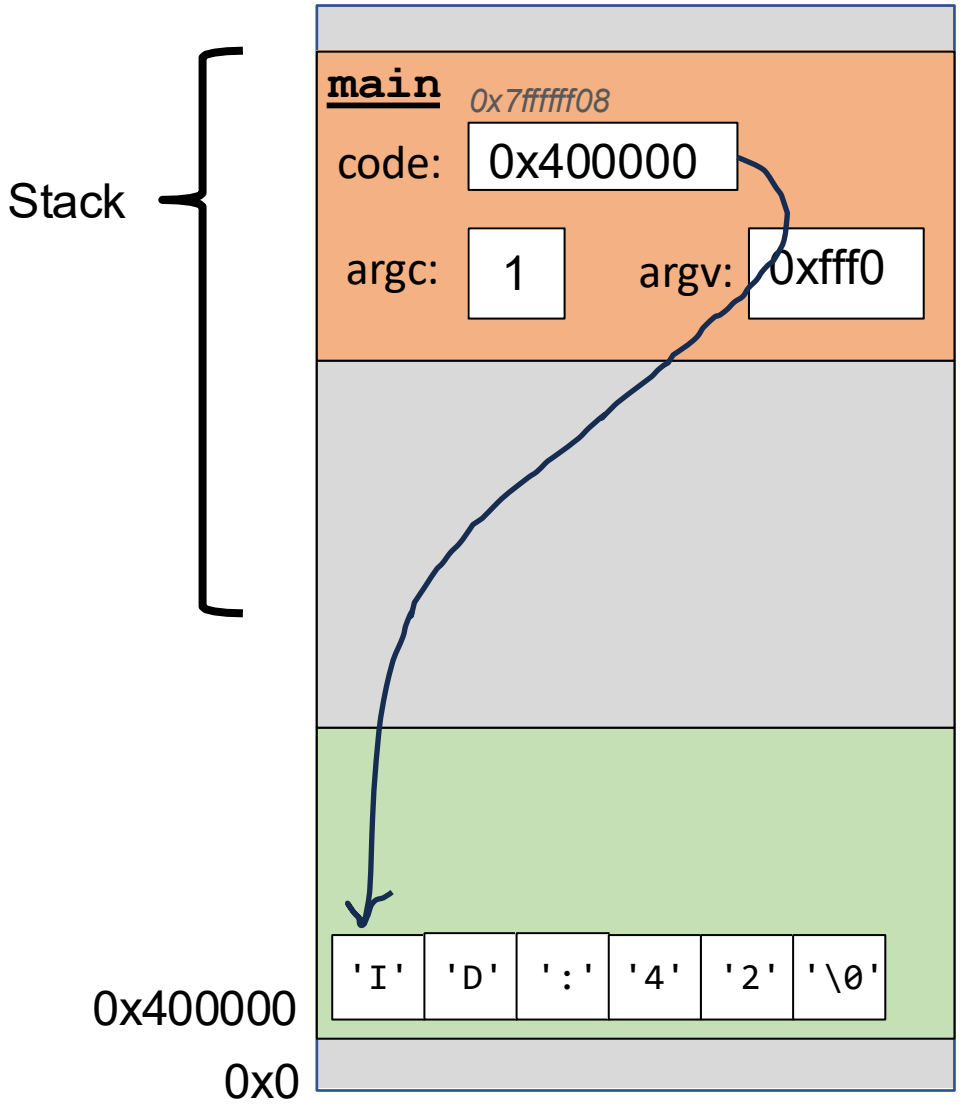
Passing by Reference in C

- Three steps to change to pass by reference:
  - Add a level of indirection to the parameters in the callee function (add a *)
  - Thus, we need to pass in the ADDRESS (&) of the data as args to the callee, not the data itself
  - Thus, we need to DEREFERENCE the arguments passed from the caller to the callee when making updates in the callee (use *)
- Diagrams + fake addresses are your friend!

Data Type	*New Pass-by-Reference Parameter Type*
`int`	`int *`
`char *`	`char **`
`long`	`long *`
`char`	`char *`
[ANY DATA TYPE]	[ANY DATA TYPE]*

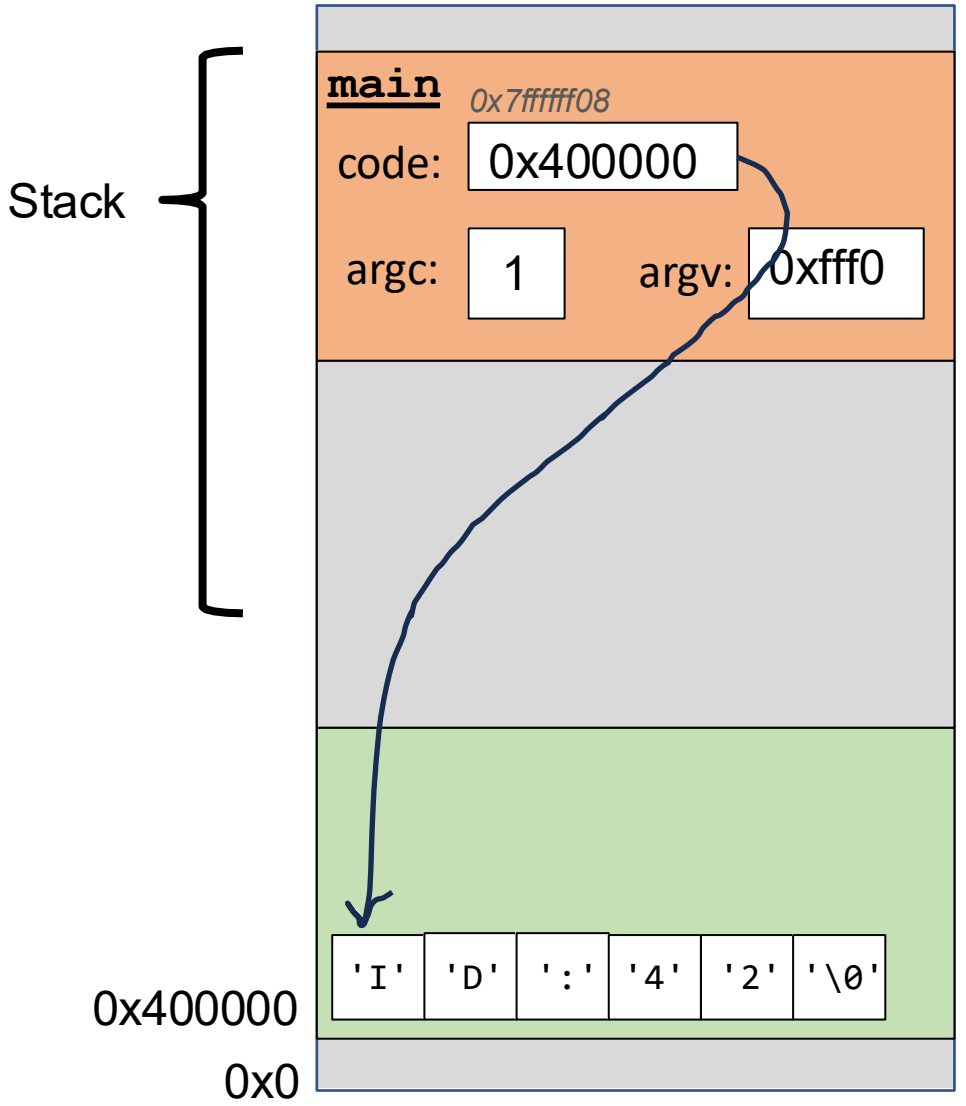
Passing by Reference in C

```
void skip_prefix(char *str) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(code);
    printf("%s\n", code);    // still prints "ID:42"
    return 0;
}
```



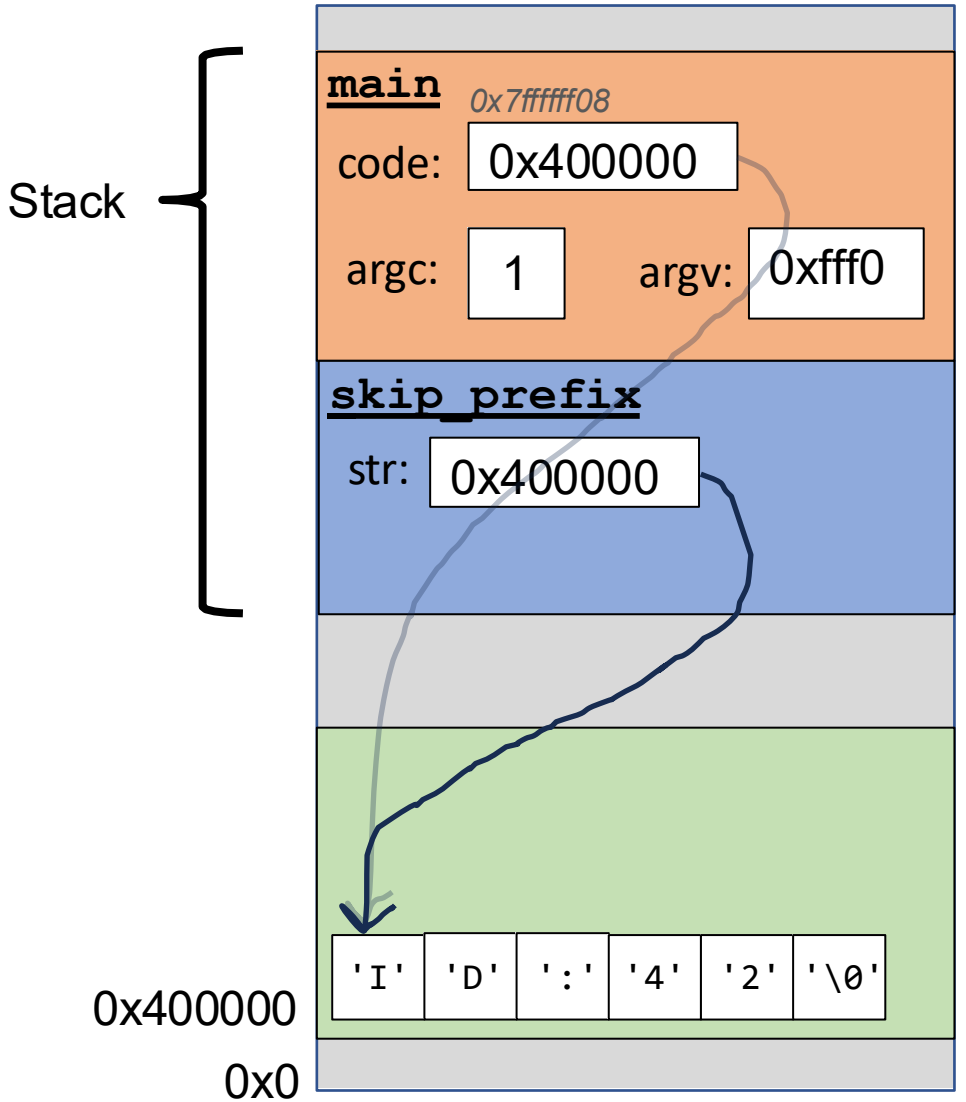
Passing by Reference in C

```
void skip_prefix(char *str) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(code);
    printf("%s\n", code);   // still prints "ID:42"
    return 0;
}
```



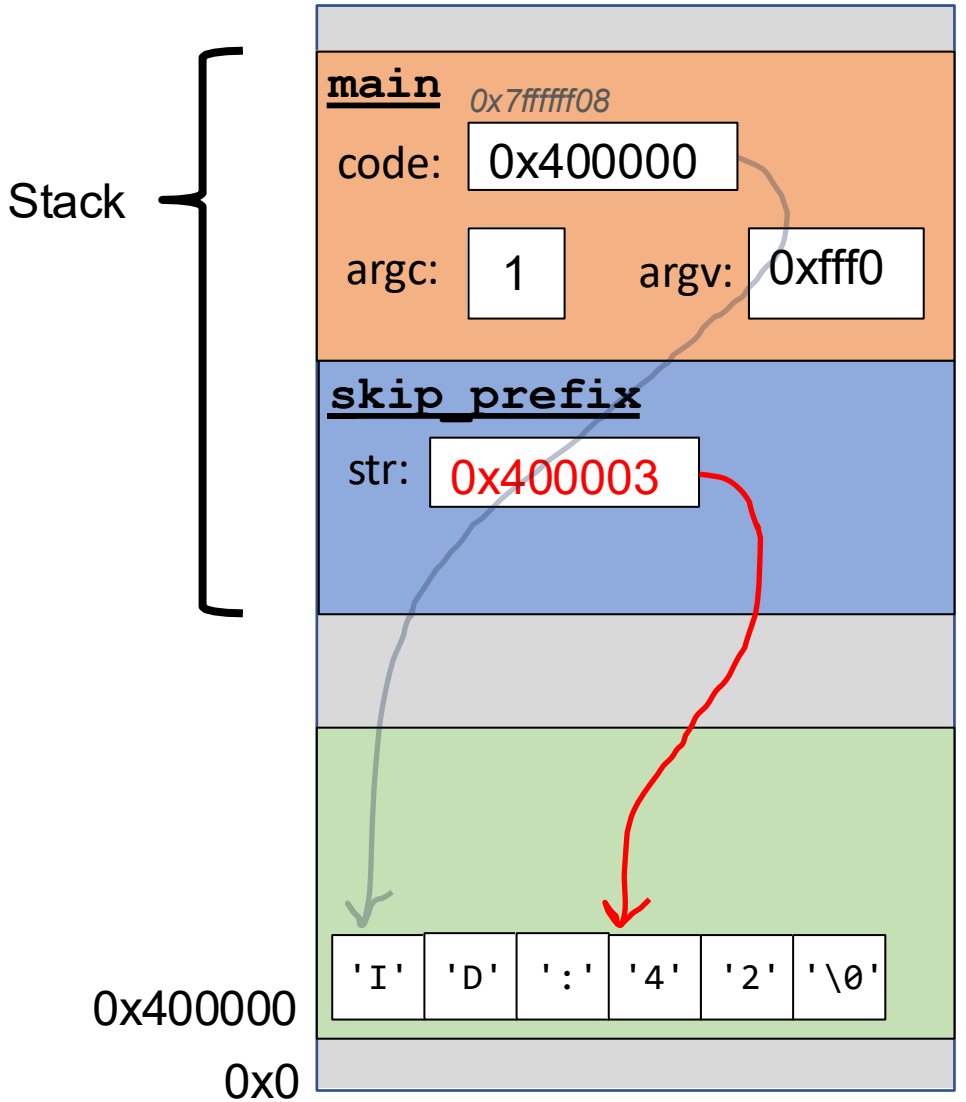
Passing by Reference in C

```
void skip_prefix(char *str) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(code);
    printf("%s\n", code);    // still prints "ID:42"
    return 0;
}
```



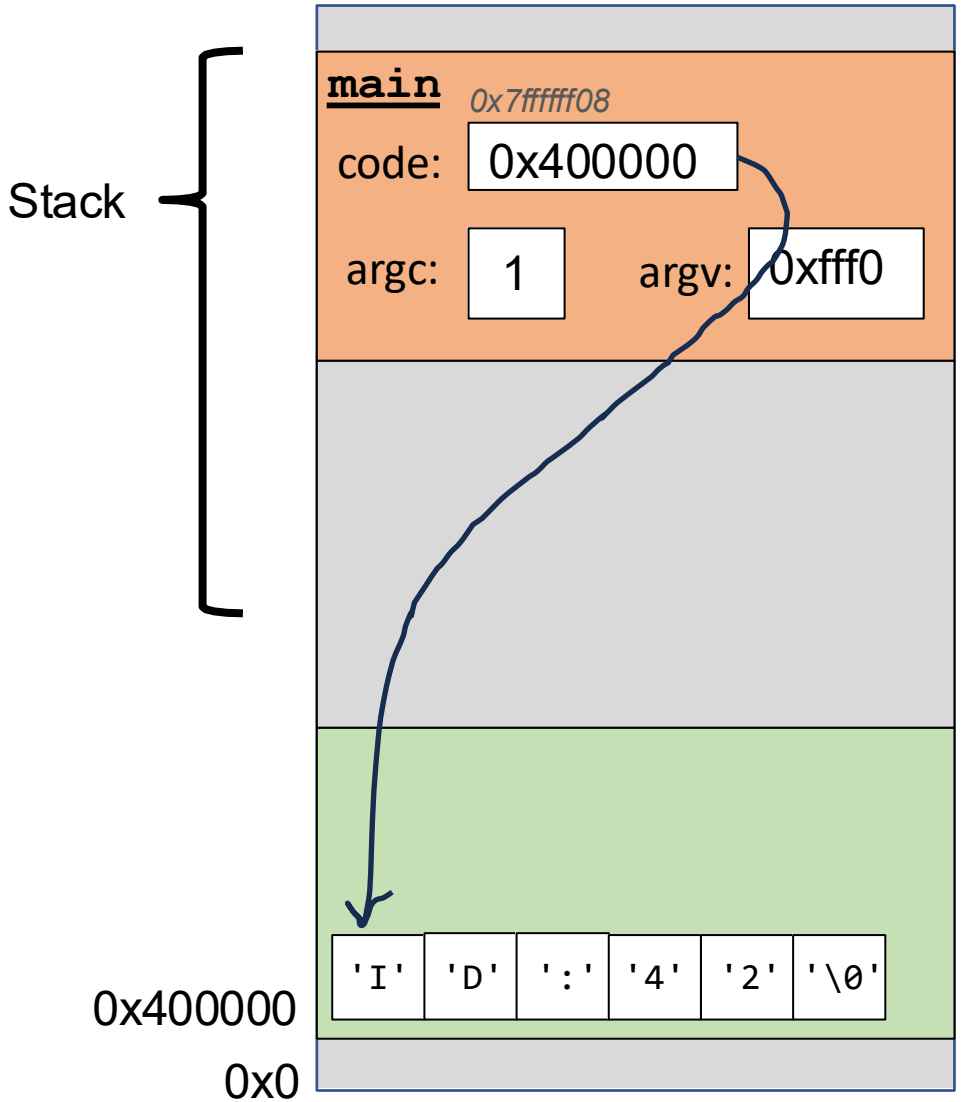
Passing by Reference in C

```
void skip_prefix(char *str) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(code);
    printf("%s\n", code);    // still prints "ID:42"
    return 0;
}
```



Passing by Reference in C

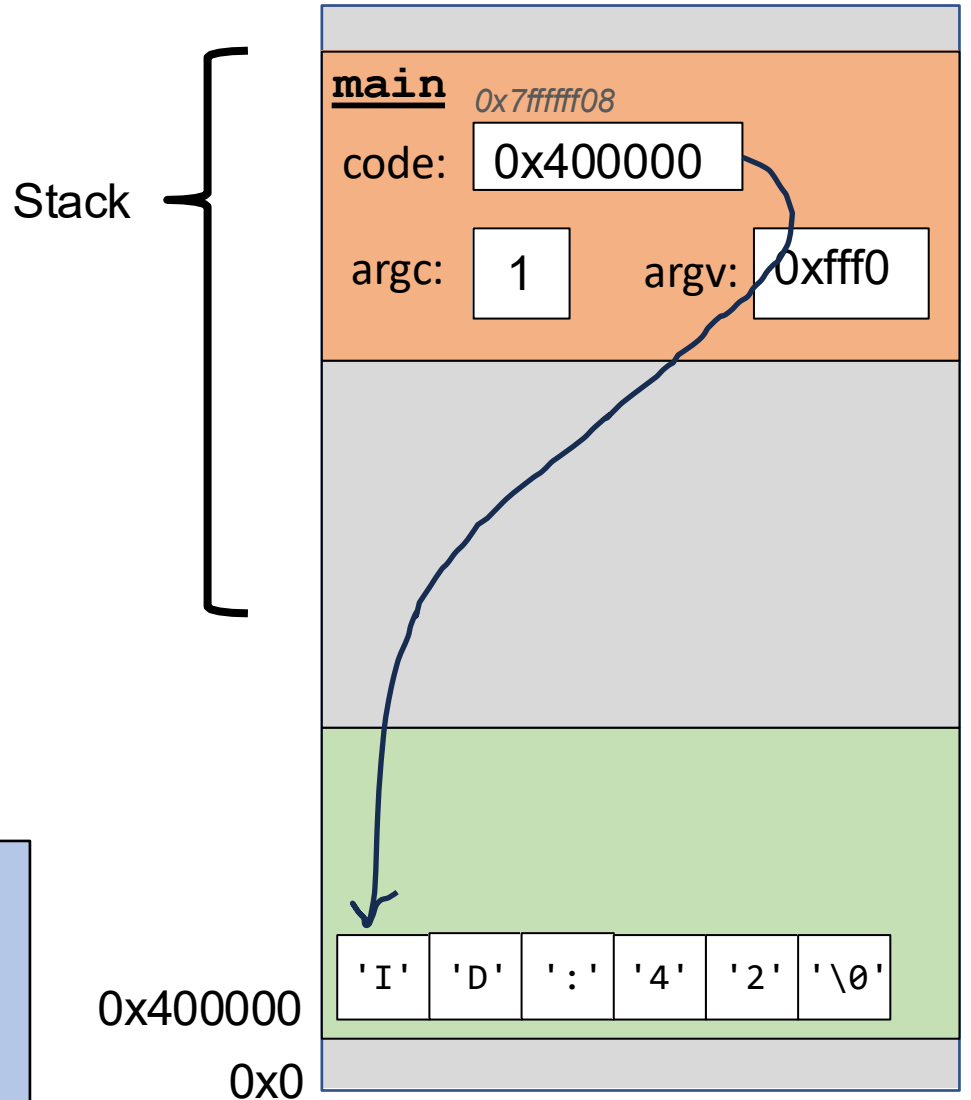
```
void skip_prefix(char *str) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(code);
    printf("%s\n", code);    // still prints "ID:42"
    return 0;
}
```



Passing by Reference in C

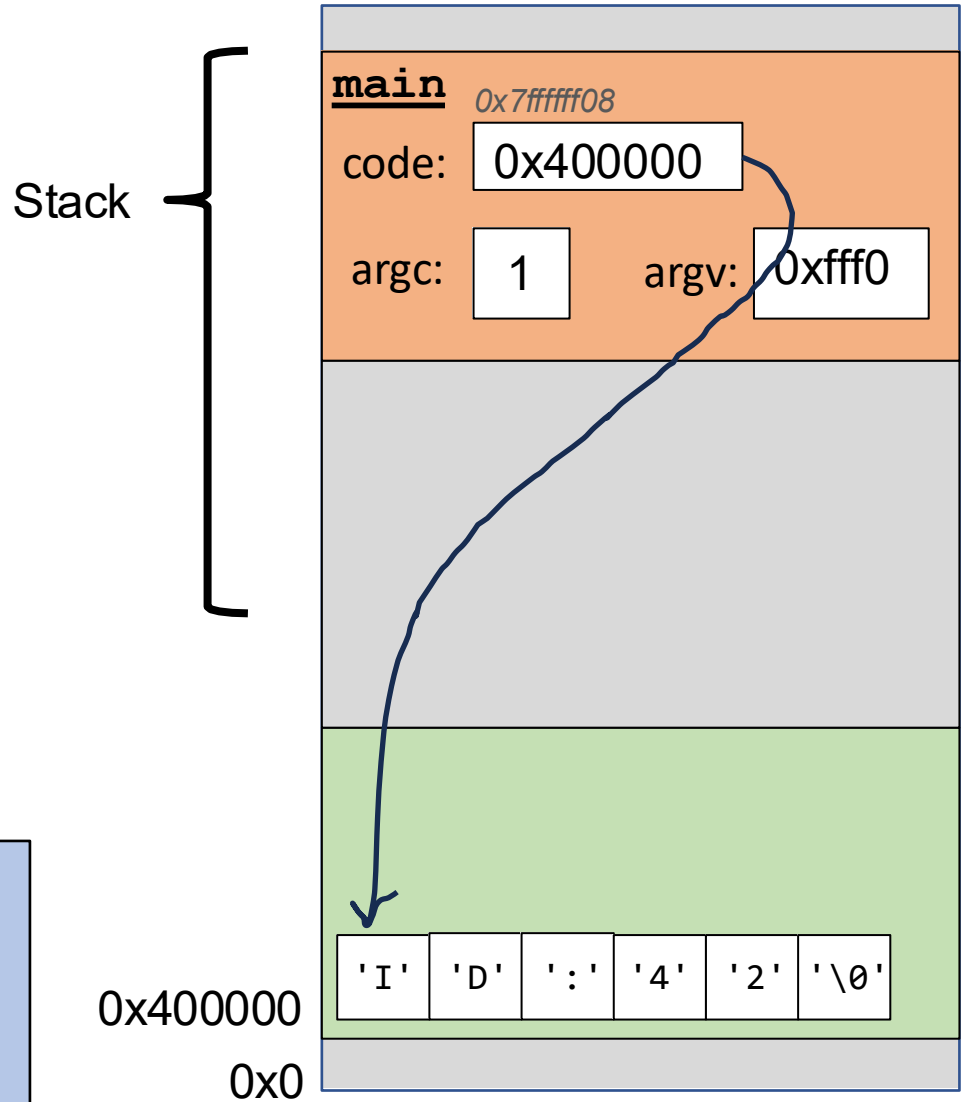
```
void skip_prefix(char *str) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(code);
    printf("%s\n", code);   // still prints "ID:42"
    return 0;
}
```

Let's fix this code!



Passing by Reference in C

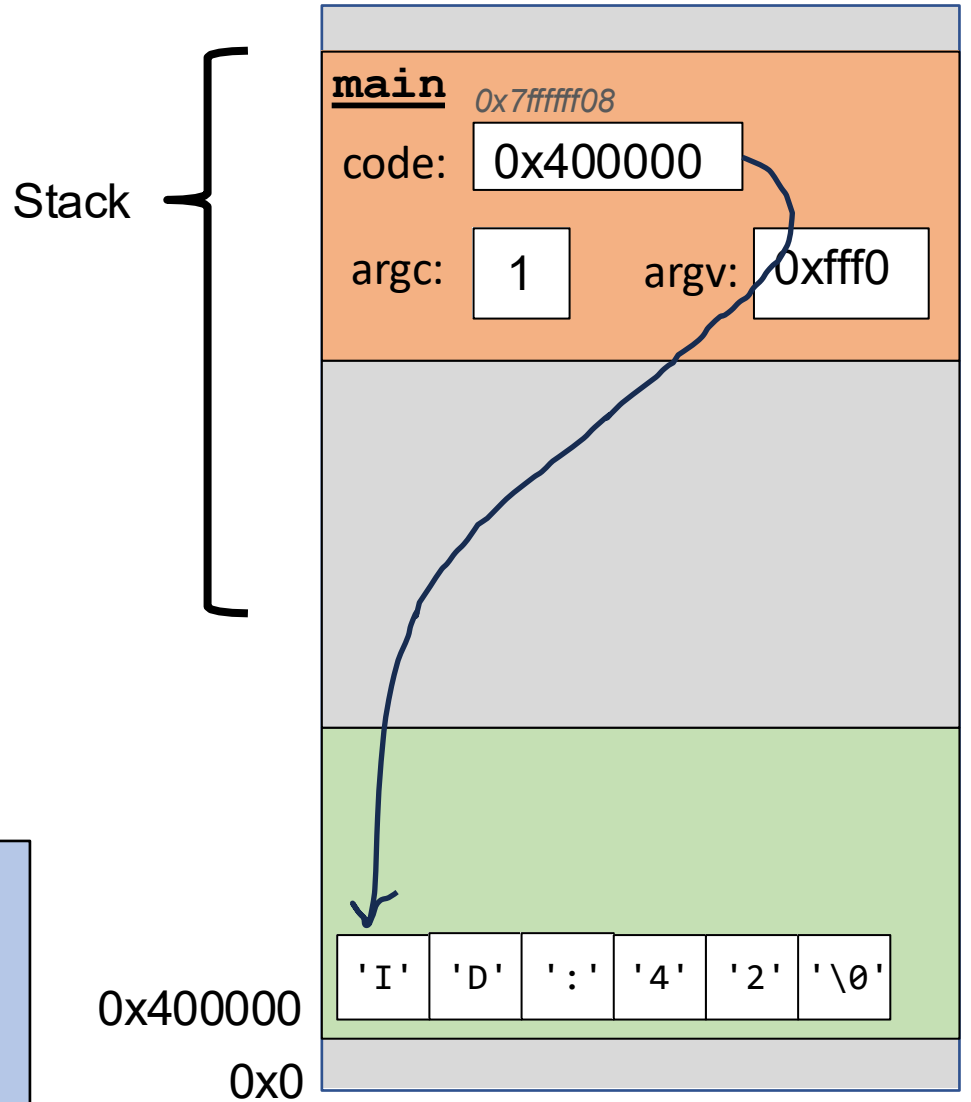
```
void skip_prefix(char **str_ptr) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(code);
    printf("%s\n", code);   // still prints "ID:42"
    return 0;
}
```



1. Add a level of indirection to the parameters in the callee function (`skip_prefix`)

Passing by Reference in C

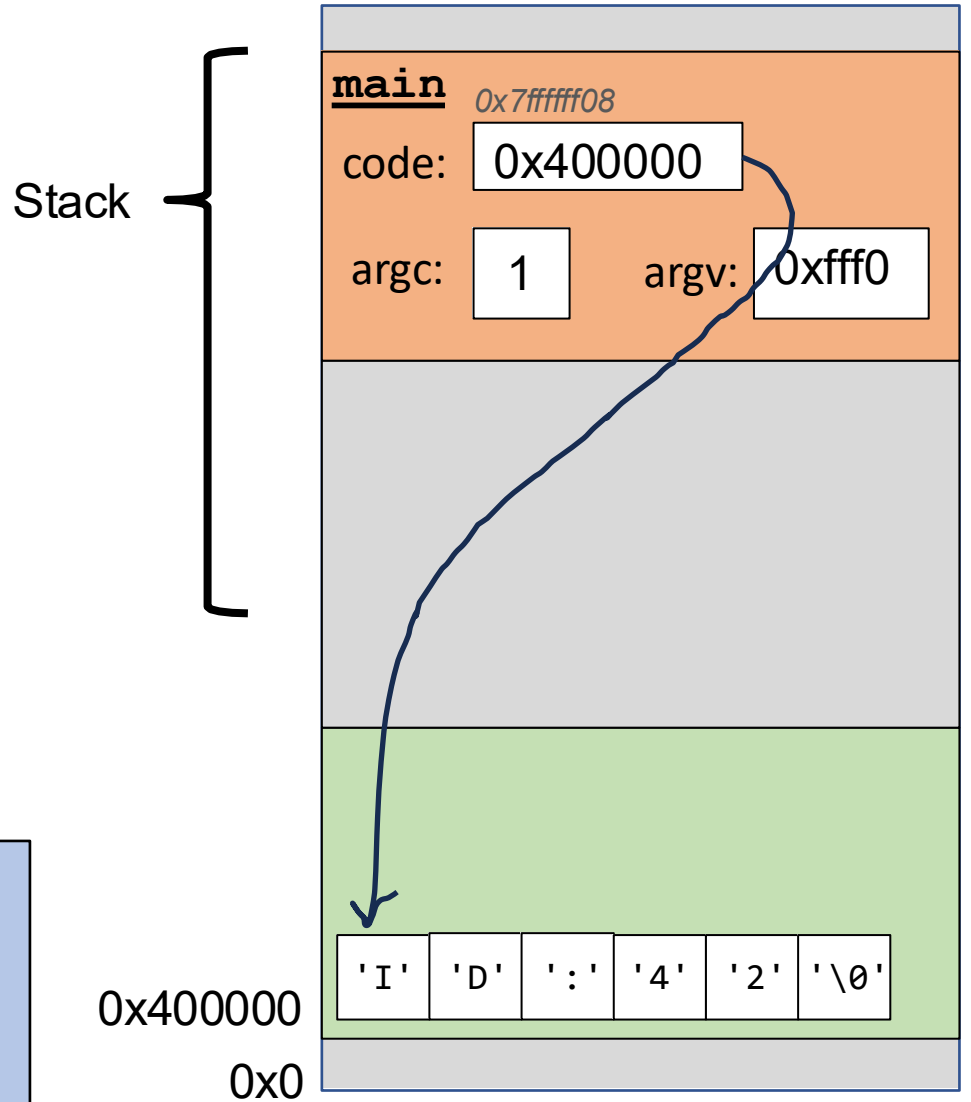
```
void skip_prefix(char **str_ptr) {
    str = str + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);    // still prints "ID:42"
    return 0;
}
```



2. Pass the ADDRESS of the data as args to the callee (`skip_prefix`), not the data itself

Passing by Reference in C

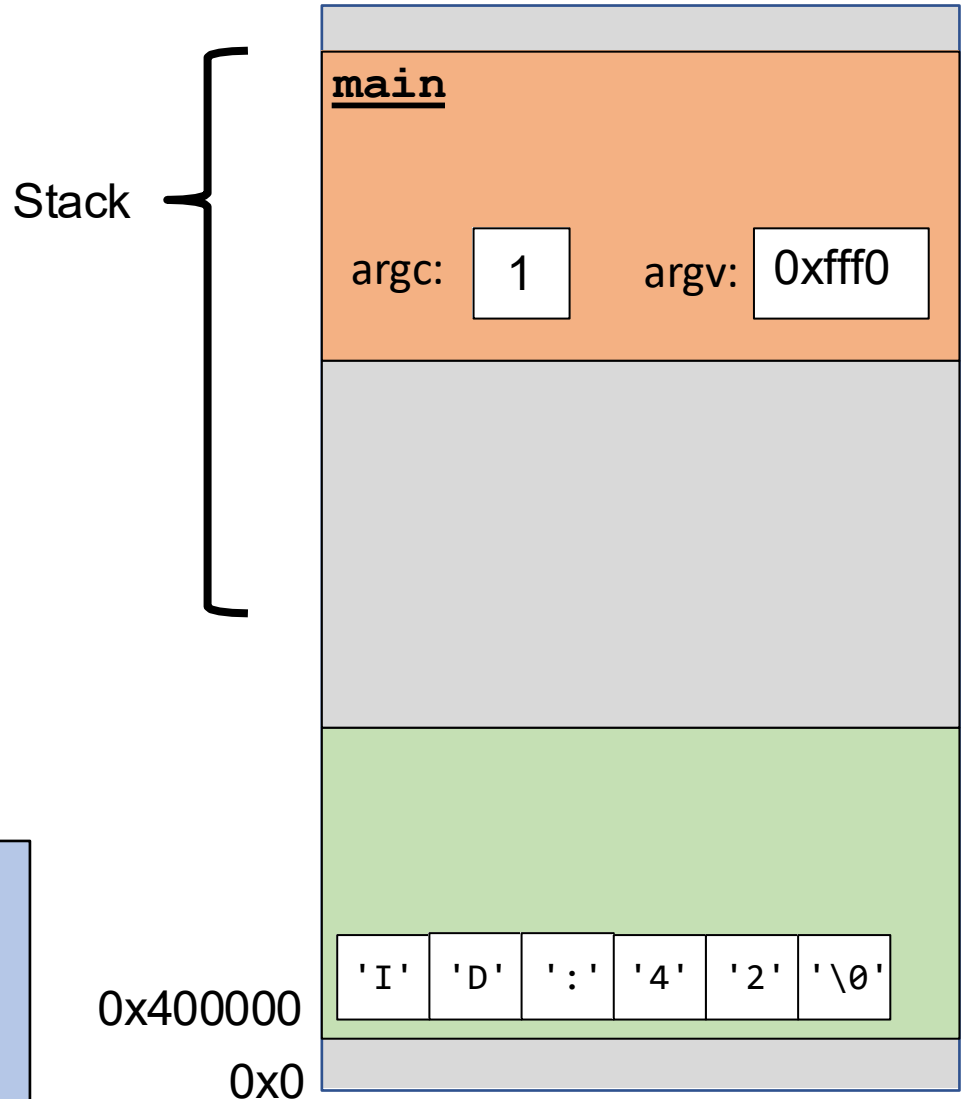
```
void skip_prefix(char **str_ptr) {
    *str_ptr = *str_ptr + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);
    return 0;
}
```



3. DEREFERENCE the arguments passed from the caller (`main`) to the callee (`skip_prefix`) when making updates in the callee (`skip_prefix`)

Passing by Reference in C

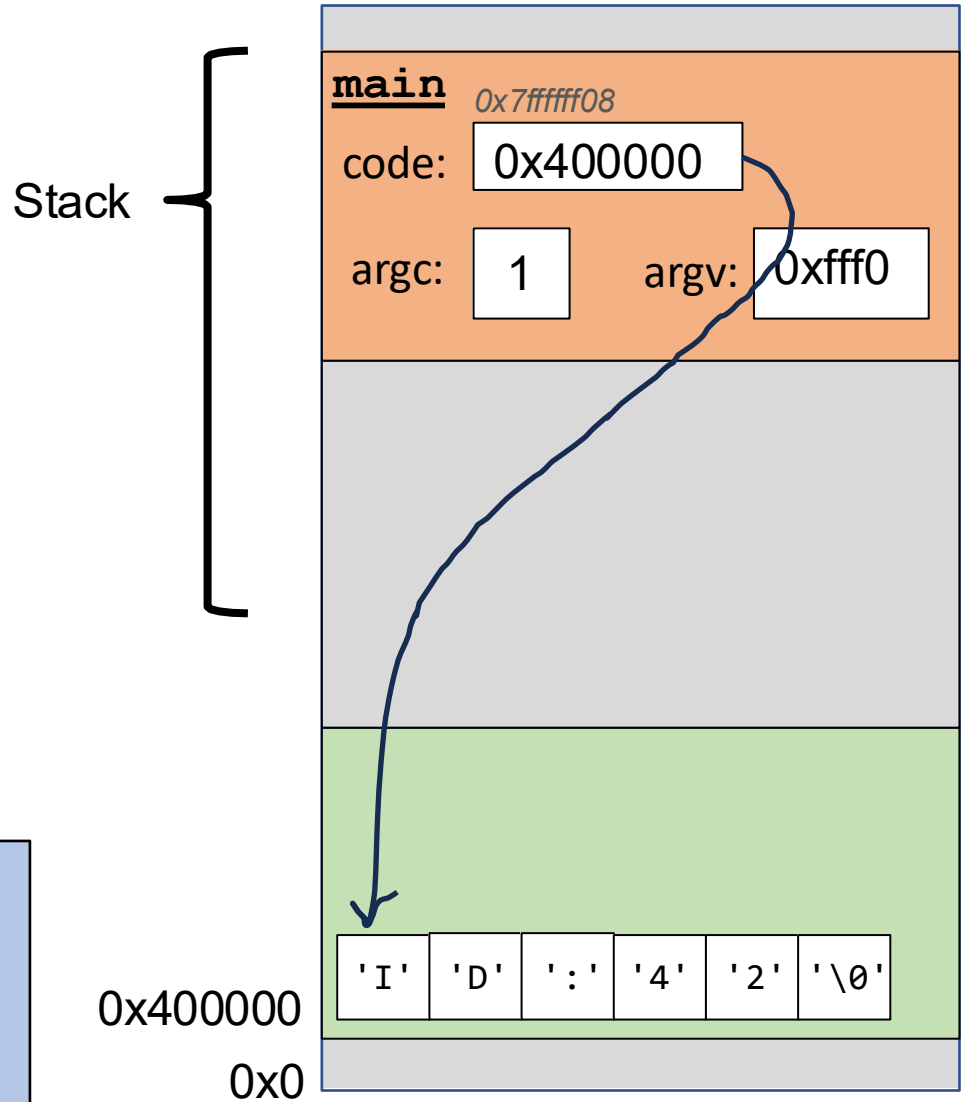
```
void skip_prefix(char **str_ptr) {
    *str_ptr = *str_ptr + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);
    return 0;
}
```



Let's trace through the updated, pass-by-reference version!

Passing by Reference in C

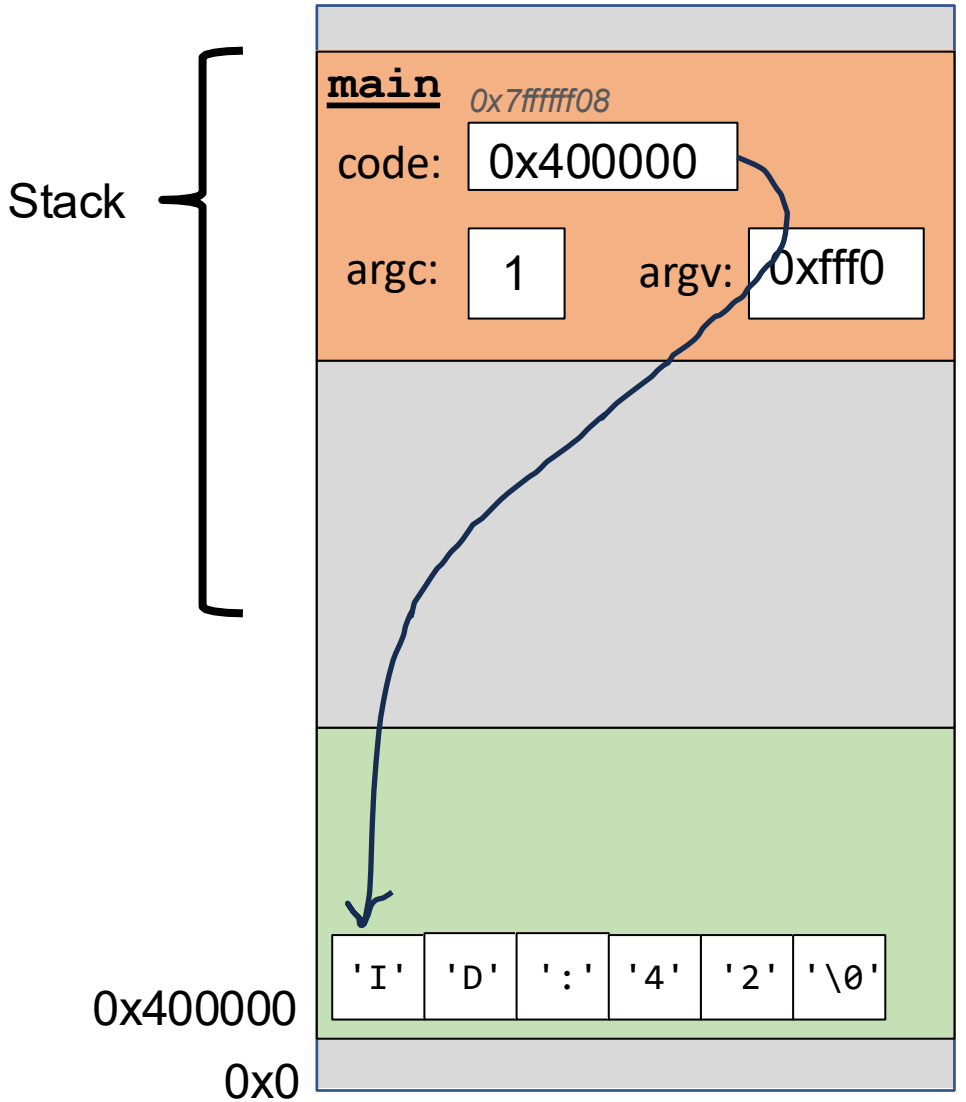
```
void skip_prefix(char **str_ptr) {
    *str_ptr = *str_ptr + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);
    return 0;
}
```



Let's trace through the updated, pass-by-reference version!

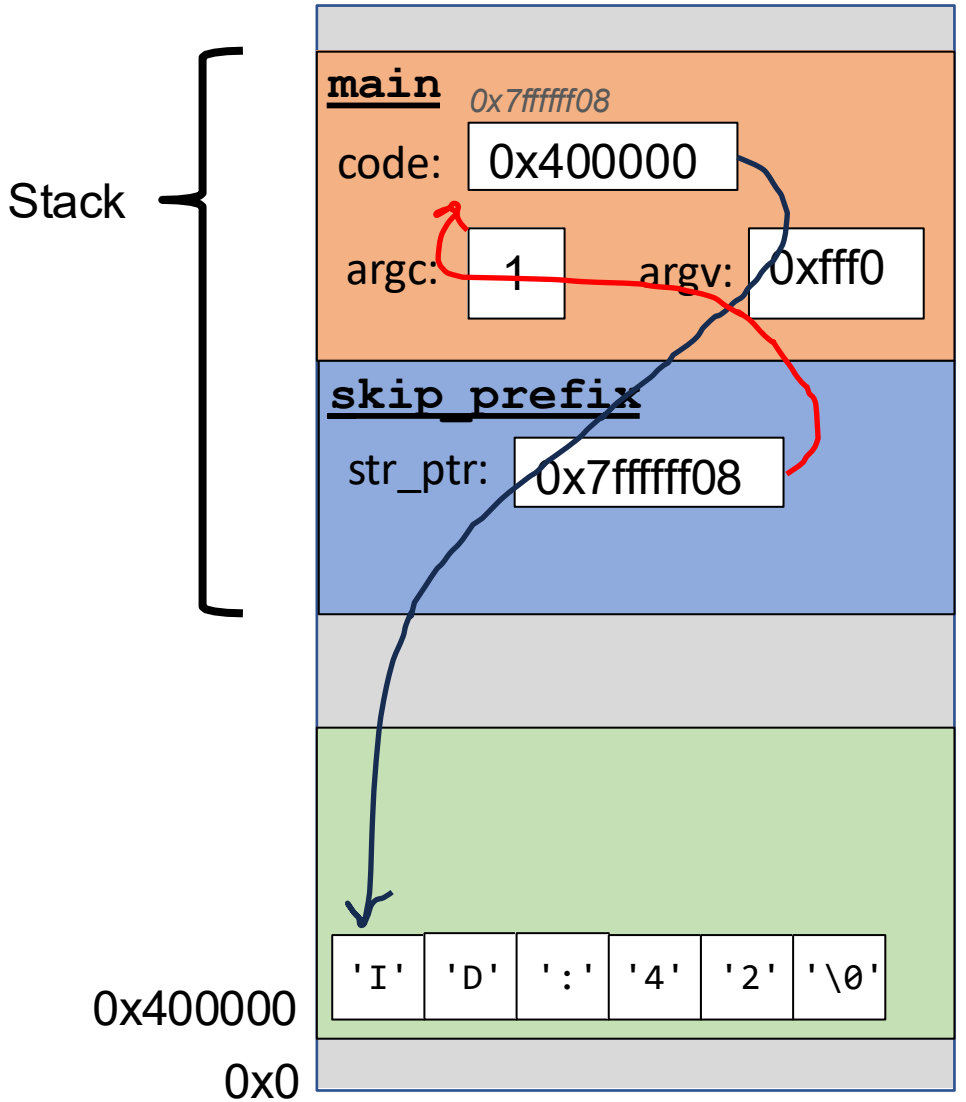
Passing by Reference in C

```
void skip_prefix(char **str_ptr) {
    *str_ptr = *str_ptr + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);
    return 0;
}
```



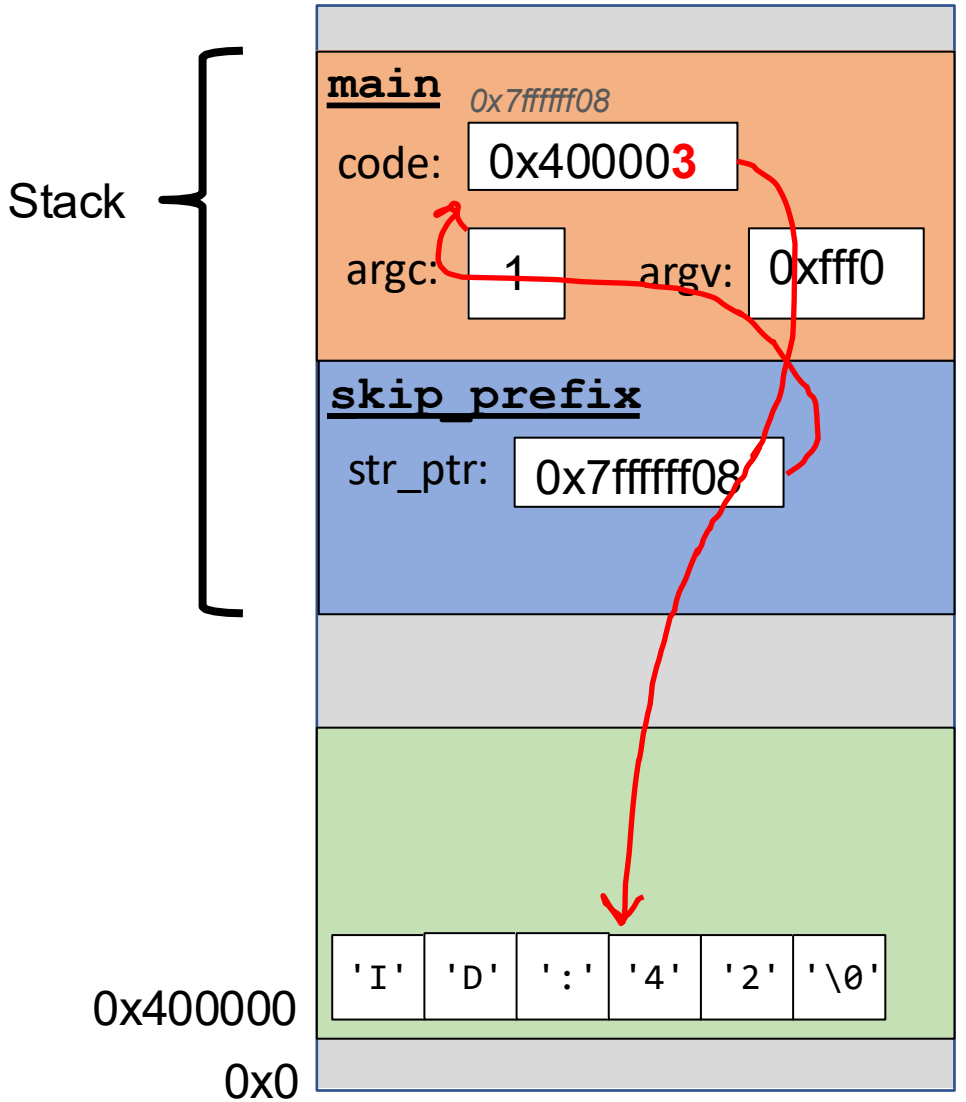
Passing by Reference in C

```
void skip_prefix(char **str_ptr) {
    *str_ptr = *str_ptr + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);
    return 0;
}
```



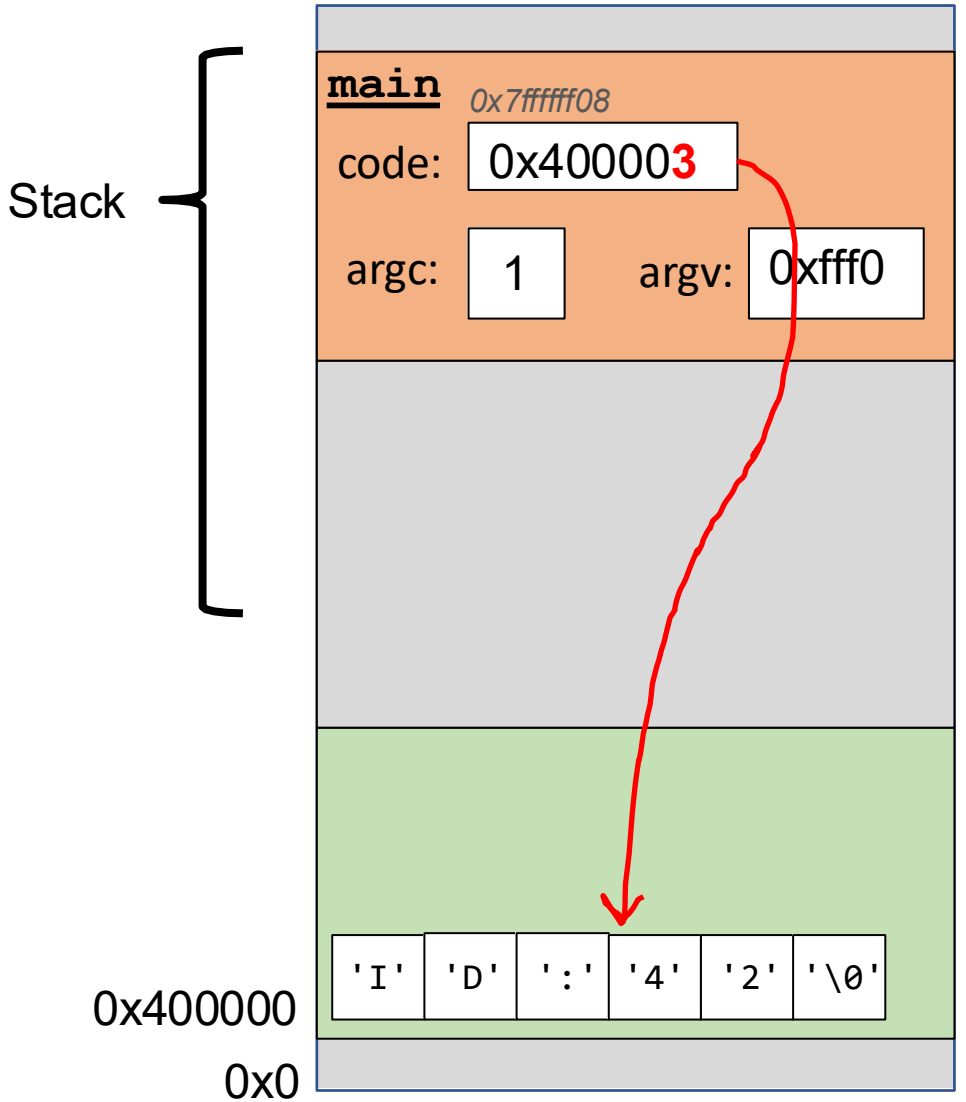
Passing by Reference in C

```
void skip_prefix(char **str_ptr) {
    *str_ptr = *str_ptr + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);
    return 0;
}
```



Passing by Reference in C

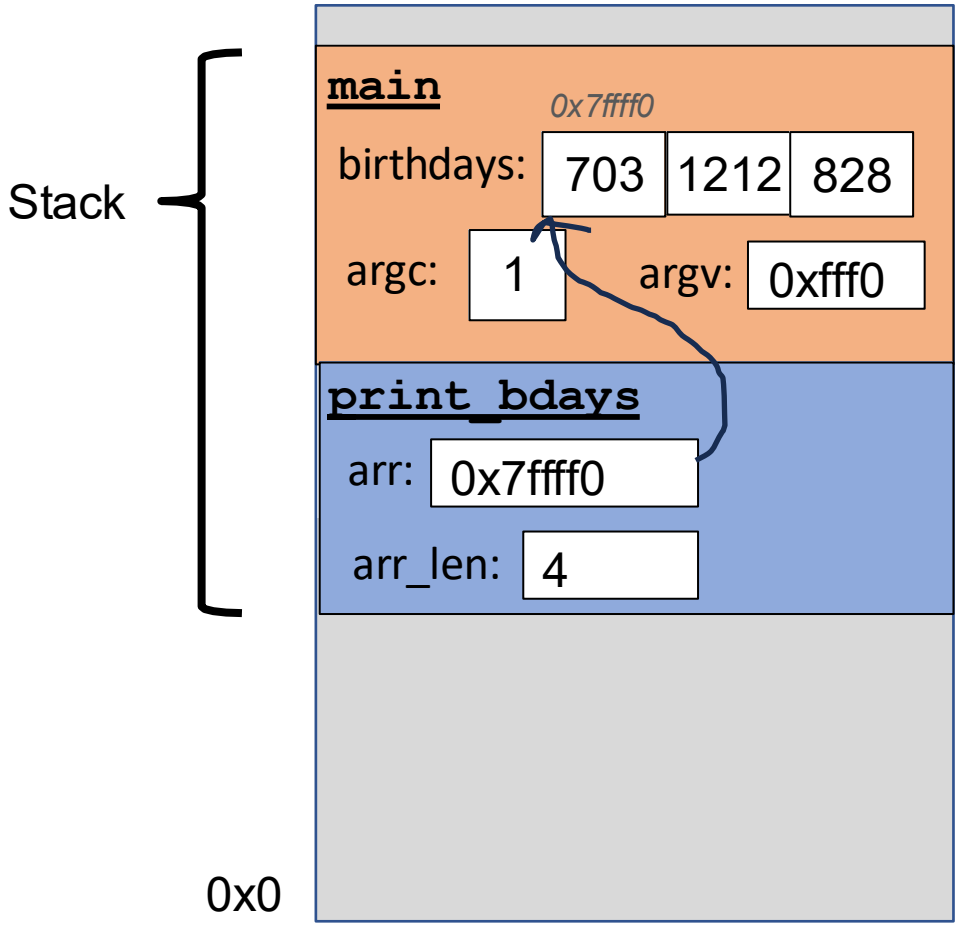
```
void skip_prefix(char **str_ptr) {
    *str_ptr = *str_ptr + 3;
}


int main(int argc, char *argv[]) {
    char *code = "ID:42";
    skip_prefix(&code);
    printf("%s\n", code);    // now prints “42”! Yay!
    return 0;
}
```



Array Decay to Pointer

- When we pass an array as a parameter to a function, it will be **implicitly converted to a pointer type within the callee function.**

```
void print_bdays(int arr[], int arr_len) {
    for (int i = 0; i < arr_len; i++) {
        printf("Birthday is: %d\n", arr[i]);
    }
}


int main(int argc, char *argv[]) {
    int birthdays[] = {703, 1212, 828};
    print_bdays(birthdays, 3);
    return 0;
}
```



`sizeof` on Arrays

```
void print_bdays(int arr[], int arr_len) {
    printf("sizeof(arr) in print_bdays is: %d\n", sizeof(arr));
    for (int i = 0; i < arr_len; i++) {
       printf("Birthday is: %d\n", arr[i]);
    }
}


int main(int argc, char *argv[]) {
    int birthdays[] = {703, 1212, 828};
    printf("sizeof(birthdays) in main is: %d\n", sizeof(birthdays));
    print_bdays(birthdays, 3);
    return 0;
}
```

What do each of these highlighted **`printf`** calls print? 🤔

Generics 

Generics Topics

- `void *`
- Generic swap example
- `memcpy()`
- `memmove()`
- Casting `void *`s
  - Can't dereference/do pointer arithmetic with `void *`s!
  - Casting a `void *` to a `char *` **simply because a character is 1 byte**
- Generic stack example
- Function pointers
- Comparison functions
  - Cast args, dereference, compare

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Return type
(bool)

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Function pointer name
(compare_fn)

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Function parameters
(two void *s)

Function Pointers

Here's the general variable type syntax:

```
[return type] (*[name])([parameters])
```

Comparison Functions

- Three step process!

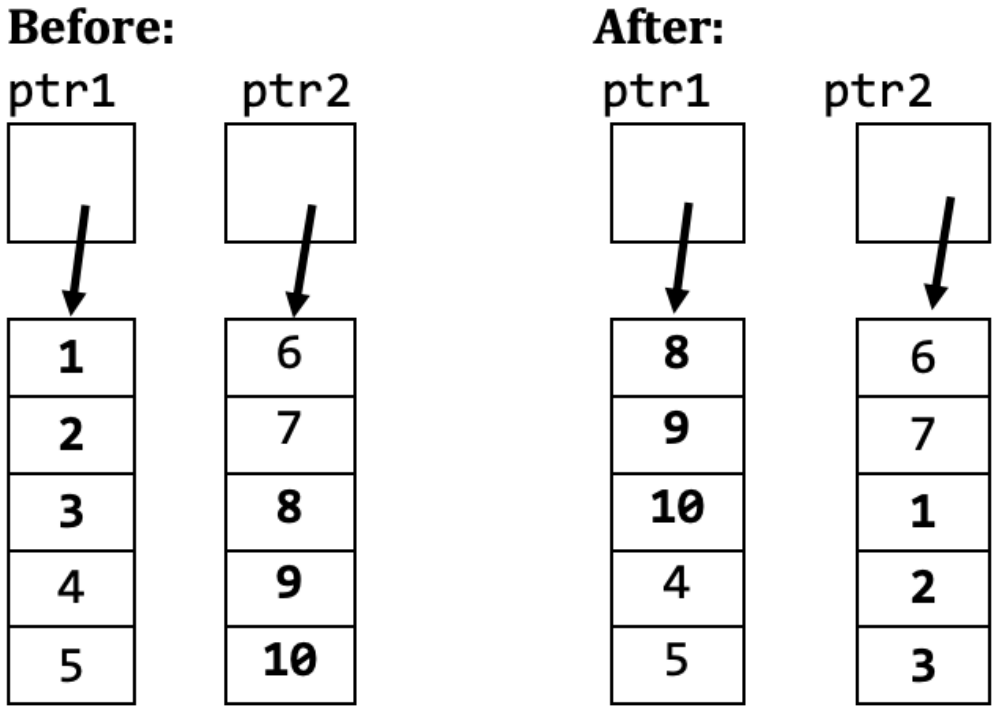
```
int integer_compare_ascending(void *a, void *b) {
    // Cast arguments to int*s
    int *a_int_ptr = (int *)a;
    int *b_int_ptr = (int *)b;

    // Dereference typed pointers
    int a_data = *a_int_ptr;
    int b_data = *b_int_ptr;

   // Compare data and return!
   return a_data - b_data;
}
```

Practice: Generics Topics

Recall our generic swap:

```
void swap(void *a, void *b, size_t size) {
    char temp[size];
    memcpy(temp, a, size);
    memcpy(a, b, size);
    memcpy(b, temp, size);
}
```

Make a call to `swap()` using `ptr1` and `ptr2` to produce the resultant image from the before image.

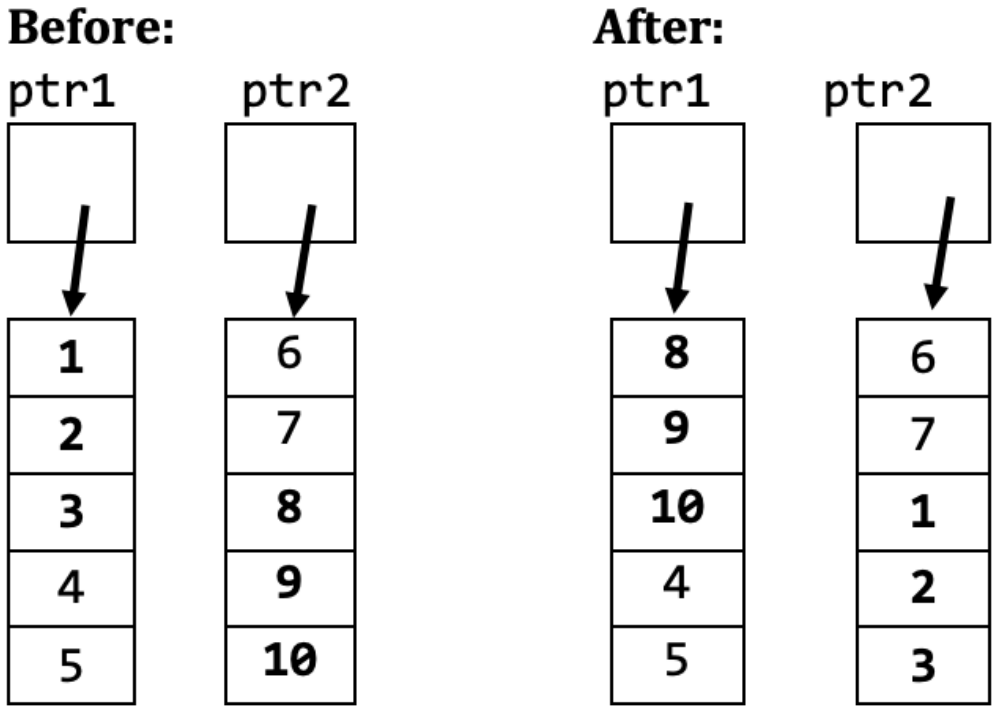


Practice: Generics Topics

Recall our generic swap:

```
void swap(void *a, void *b, size_t size) {
    char temp[size];
    memcpy(temp, a, size);
    memcpy(a, b, size);
    memcpy(b, temp, size);
}
```

Make a call to `swap()` using `ptr1` and `ptr2` to produce the resultant image from the before image.




```
swap(ptr1, ptr2 + 2, sizeof(int) * 3);
```

Assembly 

Assembly Topics

- Assembly code
- Machine code
- Literals in assembly 
- Instruction Set Architecture (ISA)
- Registers – not located in memory!
  - 64 bit, but 32, 16, and 8-bit sub registers
- `mov`
- Memory location syntax `0x10(%rbp, %rax, 2)`
  - displacement(base, index, scale) is displacement + base + (index * scale)
- `lea` (and differences from `mov`)
  - Odd one out – can use same memory location syntax, but no dereference is performed
- Content on the x86 reference sheet
  - Doesn't need to be memorized, but you should understand everything on there
  - Special purpose registers
- Caller vs. callee saved registers

Assembly Topics, continued

- Loops in assembly (two common gcc constructions)
- Condition codes
- Jumps
- Conditional instructions
- `test` instruction (bitwise and)
- `push`, `pop` instructions (and how they work with `%rsp`)
- `call`, `ret` instructions (and how they work with `%rip`)
- `%rax`
- `%rsp`
- `%rip`
- Caller vs. callee functions
- Calling functions in assembly
- Compiler optimizations (see lecture 14)

Condition Codes

- **Condition codes** get automatically set by arithmetic/compare instructions to describe the result of the operation (Zero? Negative? Overflowed? Carried?)
- **Conditional instructions** (jumps, sets, cmovs) read specific combos of these condition codes to determine whether a condition is met

- Condition codes we care about in 107:
  - ZF – Zero Flag
  - SF – Sign Flag
  - OF – Overflow Flag
  - CF – Carry Flag

Zero Flag

- The **Zero Flag** is set to 1 if the result of an operation is exactly 0, otherwise, it is set to 1.

```
cmp %rdi, %rsi          // ZF is 1 if %rsi - %rdi == 0,
                        // 0 otherwise

add %rbp, %rbx          // ZF is 1 if sum is 0, 0 otherwise


test %rax, %rax         // ZF is 1 if %rax holds 0, 0 otherwise
```

Sign Flag

- The **Sign Flag** is set to 1 if the result of an operation has an MSB equal to 1, otherwise, it is set to 0.

```
cmp %rdi, %rsi          // SF is 1 if %rsi - %rdi has MSB == 1
                        // 0 otherwise

add %rbp, %rbx          // SF is 1 if sum has MSB == 1,
                        // 0 otherwise

test %rax, %rax         // SF is 1 if %rax holds value with
                        // MSB == 1, 0 otherwise
```

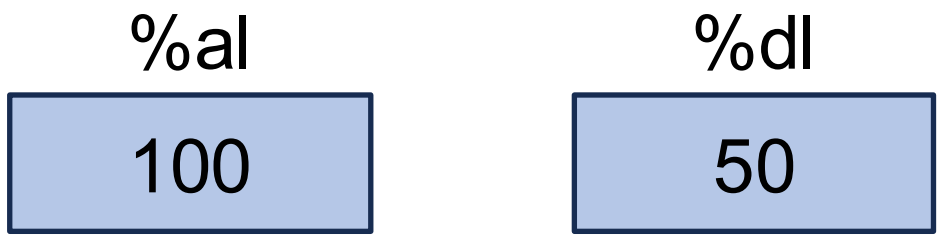
Overflow Flag

- The **Overflow Flag** is set to 1 if a **signed** arithmetic operation produces a result that can't fit in the destination (meaning the sign bit is wrong/meaningless), otherwise, it is set to 0.

```
cmp %rdi, %rsi          // OF is 1 if %rsi - %rdi (as a signed
                        // operation) overflows, 0 otherwise

add %rbp, %rbx          // OF is 1 if signed sum %rbp + %rbx
                        // overflows, 0 otherwise

test %rax, %rax         // OF is always 0 (test/and/or/xor will
                        // clear the OF)
```

Overflow Flag Example

- Let's work through an 8-bit example for simplicity!
- Reminder: 8-bit signed integer can hold values -128, …, 127

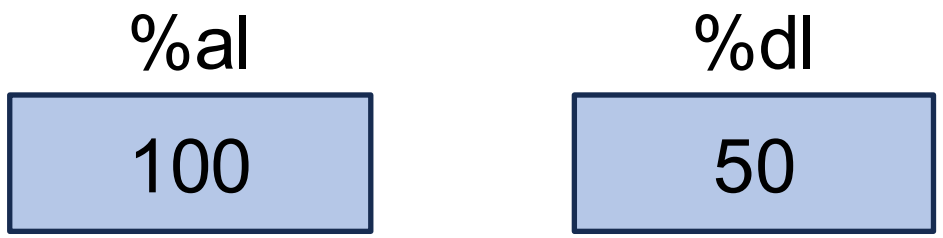



```
add %al, %dl
```

The sum of 100 + 50 = 150 – this is outside the range of valid 8-bit integers!

Overflow Flag Example

- Let's work through an 8-bit example for simplicity!
- Reminder: 8-bit signed integer can hold values -128, …, 127

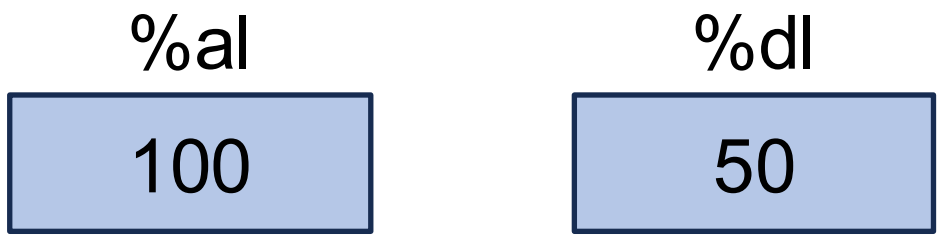



```
add %al, %dl
```

The bit pattern of the sum would be: `10010110` which is decimal `-106`…

Overflow Flag Example

- Let's work through an 8-bit example for simplicity!
- Reminder: 8-bit signed integer can hold values -128, …, 127




```
add %al, %dl
```

So we will see the OF getting set to 1 here.

Carry Flag

- The **Carry Flag** is set to 1 if an **unsigned** arithmetic operation generates a carry out of the top bit (addition) or a borrow out of the top bit (subtraction). Like the overflow flag, but for unsigned numbers

```
cmp %rdi, %rsi          // CF is 1 if %rsi < %rdi as unsigned,
                        // 0 otherwise

add %rbp, %rbx          // CF is 1 if unsigned sum of %rbp and
                        // %rbx overflows 64 bits, 0 otherwise

test %rax, %rax         // CF is always 0 (test/and/or/xor will
                        // clear the OF)
```

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = 5`** and **`%bl = 3`**

```
add %al, %bl
```

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = 5`** and **`%bl = 3`**

```
add %al, %bl
```

Zero Flag = 0	Result is 0b00001000, nonzero
Sign Flag = 0	Result is 0b**0**0001000, MSB != 0
Overflow Flag = 0	Result fits in 8 bits as a signed addition
Carry Flag = 0	Result fits in 8 bits as an unsigned addition

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = 1`** (`0b00000001`) and **`%bl = -1`** (`0b11111111`)

```
cmp %al, %bl
```

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = 1`** (`0b00000001`) and **`%bl = -1`** (`0b11111111`)

```
cmp %al, %bl
```

Zero Flag = 0	%bl - %al result is nonzero
Sign Flag = 1	Result is 0b**1**1111110, MSB == 1
Overflow Flag = 0	Subtraction result (-2) fits in 8 bits
Carry Flag = 0	As unsigned, %bl (255) > %al (1), so no borrow was needed

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = 5`** and **`%bl = 3`**

```
cmp %al, %bl
```

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = 5`** and **`%bl = 3`**

```
cmp %al, %bl
```

Zero Flag = 0	%bl - %al result is nonzero
Sign Flag = 1	Result is 0b**1**1111110, MSB == 1
Overflow Flag = 0	Subtraction result (-2) fits in 8 bits
Carry Flag = 1	As unsigned, %bl < %al , so a borrow was needed

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = -128`** (`0b10000000`) and **`%bl = -128`** (`0b10000000`)

```
add %al, %bl
```

Practice: Condition Codes

- Which of the flags will get set (ZF, SF, OF, CF) in this operation?
- Say that **`%al = -128`** (`0b10000000`) and **`%bl = -128`** (`0b10000000`)

```
add %al, %bl
```

Zero Flag = 1	%bl - %al result is zero
Sign Flag = 0	Result is 0b**0**0000000, MSB != 1
Overflow Flag = 1	Signed addition result (-256) doesn't fit in 8 bits
Carry Flag = 1	Unsigned addition result (128 + 128) doesn't fit in 8 bits (max 255)

Condition Codes

- Different instructions will then examine the state of these flags (ZF, SF, OF, CF) to determine whether to execute the jump (or other conditional instruction)
  - E.g. `je` will take the jump if ZF is set.

- But… it's probably easier and less time consuming to just look at the condition on the jump/move/set instruction.
  - E.g. `jle %rbp, %rax` – rather than thinking about which flags are set, I'll just translate this as: "Jump if `%rax` is less `than or equal` to `%rbp`". Much faster!
  - Just remember to put the second operand first in the comparison!

Practice: Quiz 9, Question 4

Question 4: Does `times_five` earn its name? *(4 points)*

The SDK exports a routine named **`times_five`**, documented as "returns its argument multiplied by 5." Before billing relies on it, you check the whole function:

```
0000000000000000 <times_five>:
   0x0:   f3 0f 1e fa             endbr64
   0x4:   48 8d 04 bf             lea    (%rdi,%rdi,4),%rax
   0x8:   c3                      ret
```

With the argument **`x`** in **`%rdi`**, what value does this routine put in **`%rax`**?

Practice: Quiz 9, Question 4

Question 4: Does `times_five` earn its name? *(4 points)*

The SDK exports a routine named **`times_five`**, documented as "returns its argument multiplied by 5." Before billing relies on it, you check the whole function:

```
0000000000000000 <times_five>:
   0x0:   f3 0f 1e fa             endbr64
   0x4:   48 8d 04 bf             lea    (%rdi,%rdi,4),%rax
   0x8:   c3                      ret
```

With the argument **`x`** in **`%rdi`**, what value does this routine put in **`%rax`**?

((what's in **`%rdi`**) * 4) + (what's in **`%rdi`**)

= 5 * (what's in **`%rdi`**)

Practice: Quiz 11, Question 1b

b. *(2 points)* The instruction at address `0x1a`, `mov 0x0(%rbp,%rbx,8),%rdi`, loads the argument for the `audit` call. Which C expression does it load? **(recall `%rbp` = `fees`, `%rbx` = `i`)**

Heap Allocators 

Heap Allocators Topics

- Utilization vs. Throughput
- Implicit allocator
- Explicit free list allocator
  - Last in, first out approach
  - Address order approach
- Bump allocator
- Headers
- Payload
- `malloc/free/realloc`
- 8-byte alignment requirement
- Fragmentation (internal/external)
- Choosing a free block: first fit, next fit, best fit
- Coalescing

Thanks for a great quarter!