C Reference slides

String Operations

- `strcpy`
- `strlen`
- `strcat`
- `strcmp`

strcpy, strlen

- `strcpy(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char
- `value = strlen(ptr);`
  - `value` is an integer
  - `ptr` is a pointer to char

```
int len;
char str[15];
strcpy (str, "Hello,
   world!");
len = strlen(str);
```

strcpy, strlen

- What's wrong with

```
char str[5];
strcpy (str, "Hello");
```

strncpy

- `strncpy(ptr1, ptr2, num);`
  - `ptr1` and `ptr2` are pointers to char
  - `num` is the number of characters to be copied

```
int len;
char str1[15],
   str2[15];
strcpy (str1,
   "Hello, world!");
strncpy (str2, str1,
   5);
```

strncpy

- `strncpy(ptr1, ptr2, num);`
  - `ptr1` and `ptr2` are pointers to char
  - `num` is the number of characters to be copied

```
int len;
char str1[15],
   str2[15];
strcpy (str1,
   "Hello, world!");
strncpy (str2, str1,
   5);
```

Caution: strncpy blindly copies the characters. It does not voluntarily append the string-terminating null character.

strcat

- `strcat(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char

- Concatenates the two null terminated strings yielding one string (pointed to by `ptr1`).

```
char S[25] = "world!";
char D[25] = "Hello, ";
strcat(D, S);
```

`strcat`

- `strcat(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char

- Concatenates the two null terminated strings yielding one string (pointed to by `ptr1`).
  - Find the end of the destination string
  - Append the source string to the end of the destination string
  - Add a NULL to new destination string

`strcat` Example

- What's wrong with

```
char S[25] = "world!";
strcat("Hello, ", S);
```

`strcat` Example

- What’s wrong with

```
char *s = malloc(11 * sizeof(char));
        /* Allocate enough memory for an
           array of 11 characters, enough
           to store a 10-char long string. */
strcat(s, "Hello");
strcat(s, "World");
```

`strcat`

- `strcat(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char

- Compare to Java and C++
  - `string s = s + " World!";`

- What would you get in C?
  - If you did `char* ptr0 = ptr1+ptr2;`
  - You would get the sum of two memory locations!

strcmp

- `diff = strcmp(ptr1, ptr2);`
  - `diff` is an integer
  - `ptr1` and `ptr2` are pointers to char
- Returns
  - zero if strings are identical
  - < 0 if `ptr1` is less than `ptr2` (earlier in a dictionary)
  - > 0 if `ptr1` is greater than `ptr2` (later in a dictionary)

```
int diff;
char s1[25] = "pat";
char s2[25] = "pet";
diff = strcmp(s1, s2);
```

Other operations

Increment & decrement

- `x++`: yield old value, add one
- `++x`: add one, yield new value

```
int x = 10;
x++;
int y = x++;      11

int z = ++x;      13
```

- `--x` and `x--` are similar (subtract one)

Math: Increment and Decrement Operators

- Example 1:

```
int x, y, z, w;
y=10; w=2;
x=++y;
z=--w;
```

- Example 2:

```
int x, y, z, w;
y=10; w=2;
x=y++;
z=w--;
```

What are **x** and **y** at the end of each example?

Math: Increment and Decrement Operators on Pointers

- Example 1:

```
int a[2];
int number1, number2, *p;
a[0]=1; a[1]=10;
p=a;
number1 = *p++;
number2 = *p;
```

- What will `number1` and `number2` be at the end?

Math: Increment and Decrement Operators on Pointers

- Example

```
int a[2];
int number1, number2, *p;
a[0]=1; a[1]=10;
p=a;
number1 = *p++;
number2 = *p;
```

Hint: ++ increments pointer p not variable *p

- What will `number1` and `number2` be at the end?

Logic: Relational (Condition) Operators

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Review

Review

- `int p1;`
  What does `&p1` mean?

Review

- How much is `y` at the end?

```
int y, x, *p;

x = 20;
*p = 10;
y = x + *p;
```

Review

- What are the differences between `x` and `y`?

```
char* f() {
   char *x;
   static char*y;
   return y;
}
```

Review: Debugging

```
if(strcmp("a","a"))
  printf("same!");
```

Review: Debugging

```
int i = 4;
int *iptr;
iptr = &i;
*iptr = 5;//now i=5
```

Review: Debugging

```
char *p;
p=(char*)malloc(99);
strcpy("Hello",p);
printf("%s World",p);
free(p);
```

Review: Debugging

```
char msg[5];
strcpy (msg,"Hello");
```

Operator	Description	Associativity
()	Parentheses (function call)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (change type)	
*	Dereference	
&	Address	
sizeof	Determine size in bytes	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

C No Evil

A practitioner's guide

Playing with fire

- Program arguments
- Output
- Stack memory

ARGCount ARGValues

```
int main(argc, char** argv)
int main(argc, char* argv[])
```

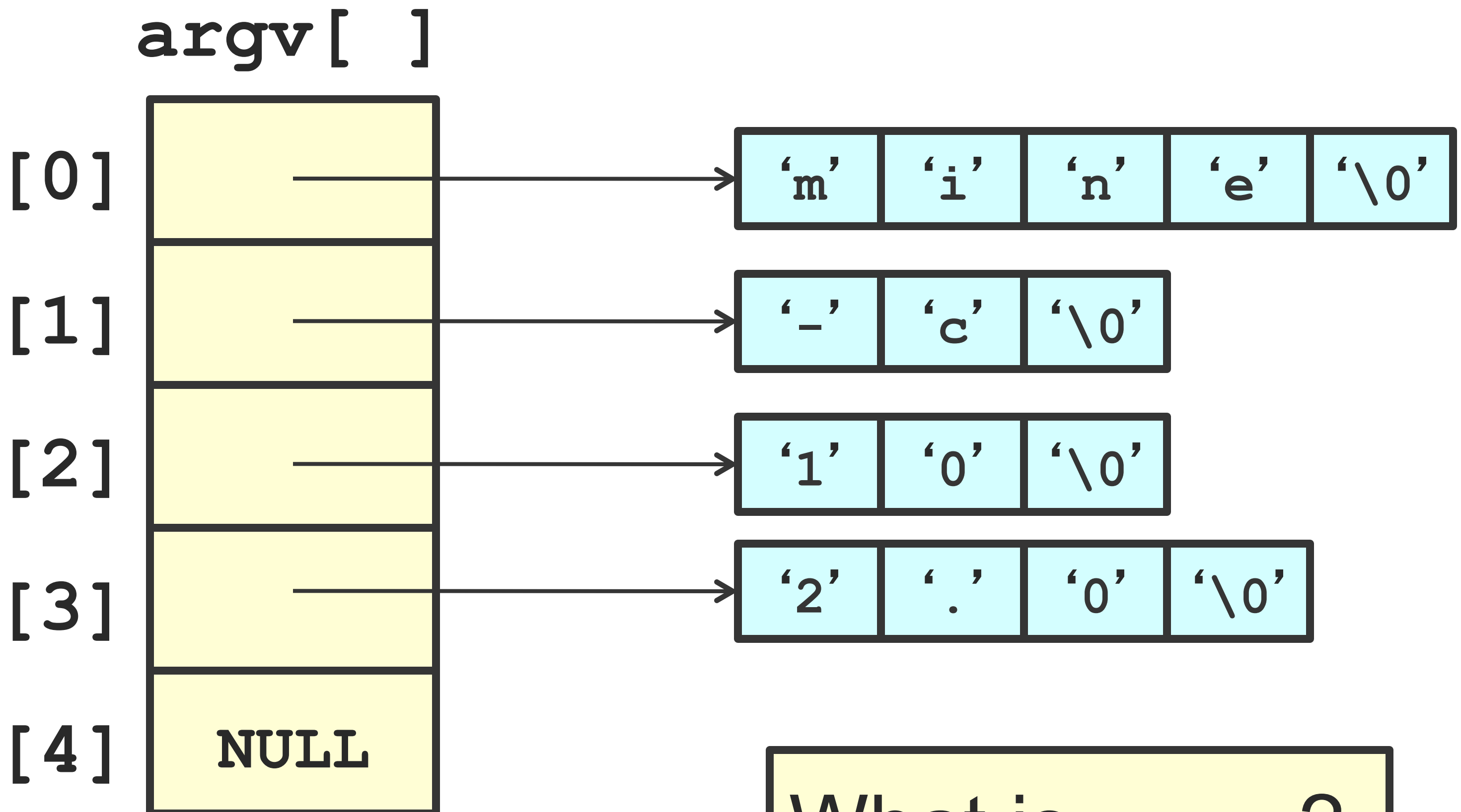
- `argc`
  - Argument count
  - The number of arguments that are passed to `main` in the argument vector `argv`.
  - the value of `argc` is always one greater than the number of command-line arguments that the user enters.

ARGCount ARGValues

```
int main(argc, char** argv)
int main(argc, char* argv[])
```

- `argv`
  - argument vector
  - An array of string pointers passed to a C program's `main` function
  - `argv[0]` is always the name of the command
  - `argv[argc]` is a null pointer

ARGCount ARGValues



What is `argc`?

Type questions

- `char **argv;`
  What type is `argv`?
  What type is `*argv`?
  What type is `**argv`?

Review

- `main(int argc, char**argv) {`
  what is `*argv`?
  what is `argv[argc]`?

`printf` Format Identifiers

`%d` or `%i`	Decimal signed integer
`%o`	Octal integer
`%x` or `%X`	Hex integer
`%u`	Unsigned integer
`%c`	Character
`%s`	String
`%f` or `%g`	Double
`%p`	Pointer

All of the parameters should be the value to be inserted EXCEPT %s, this expects a pointer to be passed

printf Basic Data Types

```
#include <stdio.h> // for printf
int main(int argc, char *argv[]) {

    // print "the date is: 01/25/2010",
    // i.e. 2- or 4-digit with leading zeros
    // using 32-bit 'long' datatype
    long day = 25;
    long month = 1;
    long year = 2010;
    printf("the date is: %02ld/%02ld/%04ld\n", month, day, year);

    // - print 8-digit hex value
    // - print a pointer value
    unsigned long ulID = 0x12345678;
    unsigned long *pID = &ulID;
    printf("hex value: 0x%02lX at address: %p\n", ulID, pID);
```

printf Basic Data Types

```
    // - print 4 bytes of a 32-bit ulong value
    // as separate hex values
    unsigned char uc1 = (unsigned char)(ulID >> 24);
    unsigned char uc2 = (unsigned char)(ulID >> 16);
    unsigned char uc3 = (unsigned char)(ulID >> 8);
    unsigned char uc4 = (unsigned char)(ulID >> 0);
    printf("hex bytes: %02X %02X %02X %02X\n",uc1,uc2,uc3,uc4);

    // - print double value like "70.35000"
    double dTemp = 70.35;
    printf("temperature: %5.5f\n", dTemp);
}
```

printf Escape Sequences

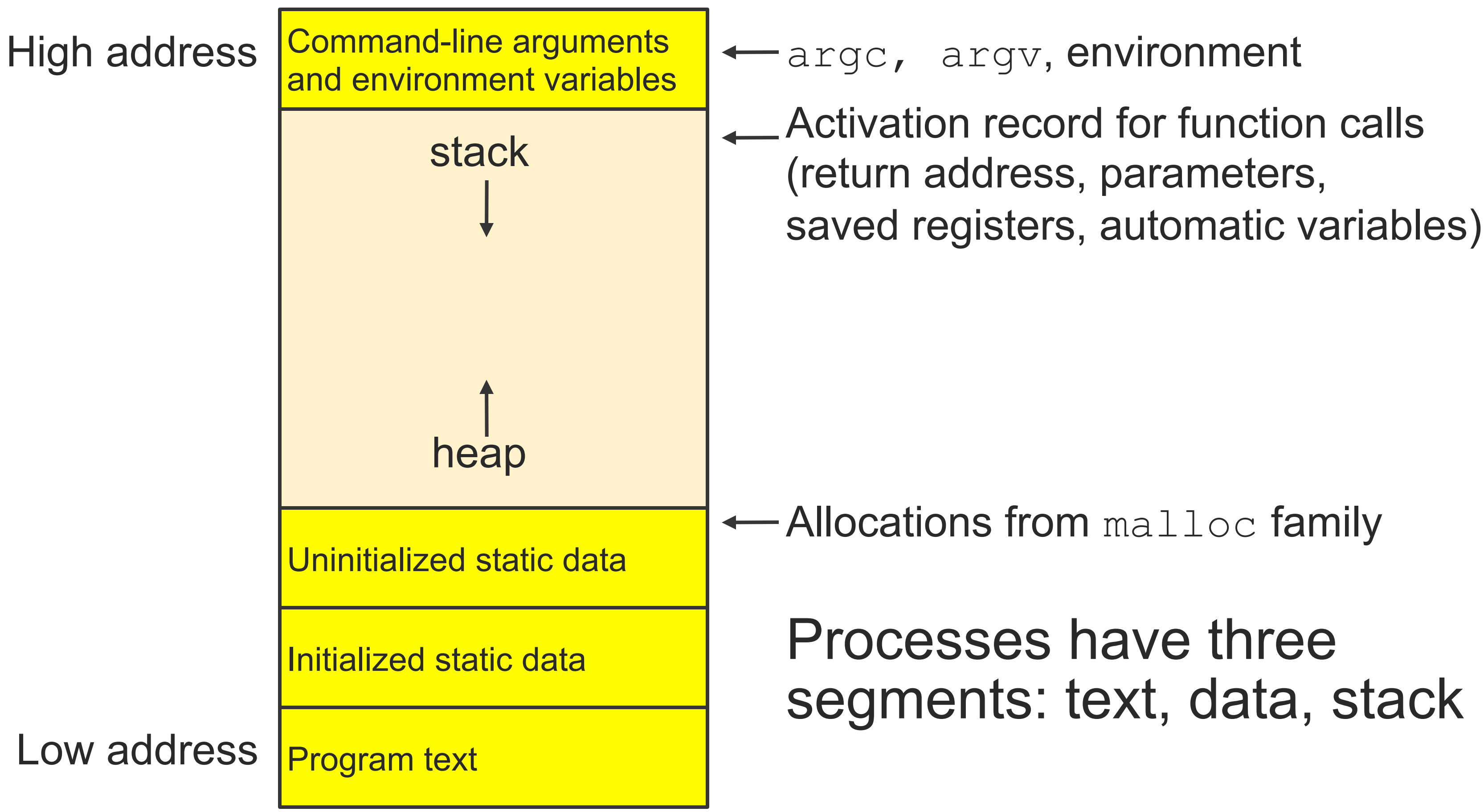
\a	<bell>
\b	<backspace>
\e	<escape>
\f	<form-feed>
\n	<new-line>
\r	<carriage return>
\t	<tab>
\v	<vertical tab>
\0	<null>

\'	<single quote>
\”	<double quote>
\?	<question mark>
\\	<backslash>
\num	an 8-bit character with ASCII value of the 1-, 2-, or 3-digit octal number num.
%%	<percent>

Common Pitfall

- Returning a variable in stack memory from a function
  - What is stack memory?

Sample layout for program image in main memory



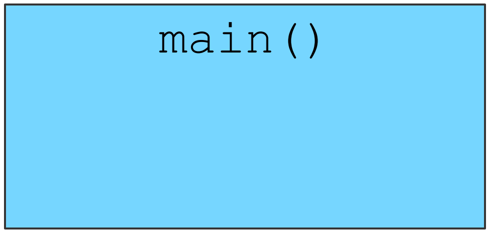
Processes have three segments: text, data, stack

Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main(int argc,
    char **argv) {
    /* … */
    a();
}
```

At the beginning of the program, the OS creates a stack frame for `main()`

Stack Memory:

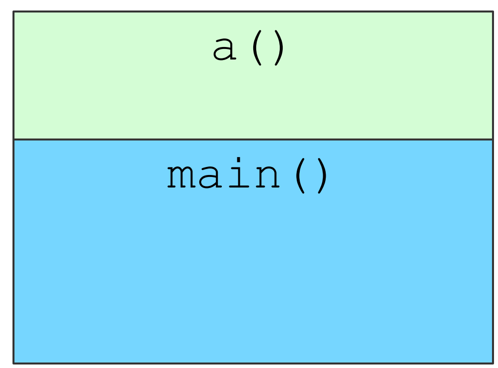


Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main(int argc,
    char **argv) {
    /* … */
    a();
}
```

When `a()` is called, the OS creates a new stack frame for `a()`

Stack Memory:

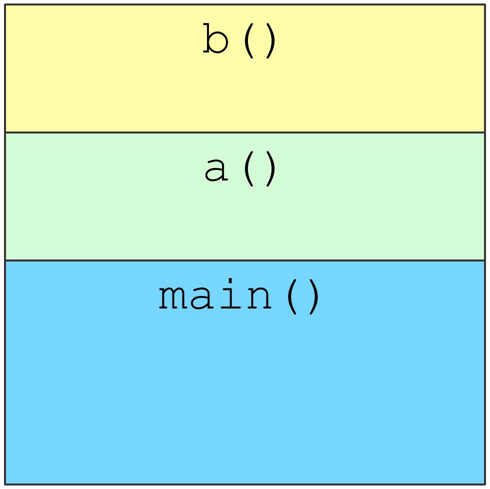


Example

Same for `b()` ...

```
int b() {
    /* … */
}

int a() {
    /* … */
    b();
}

int main(int argc,
    char **argv) {
     /* … */
     a();
}
```

Stack Memory:



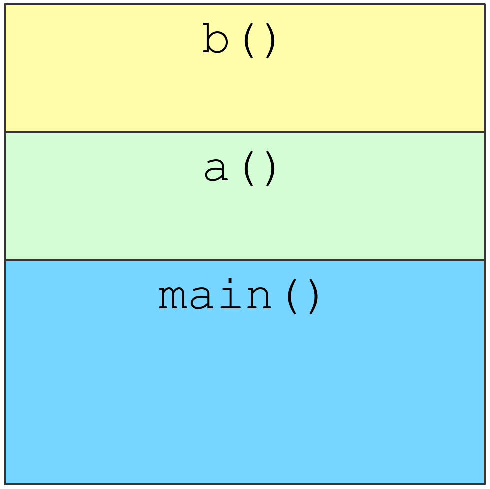
Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main(int argc,
   char **argv) {
   /* … */
   a();
}
```

When b() finishes running, its stack frame is removed!

What happens to the memory?

Stack Memory:



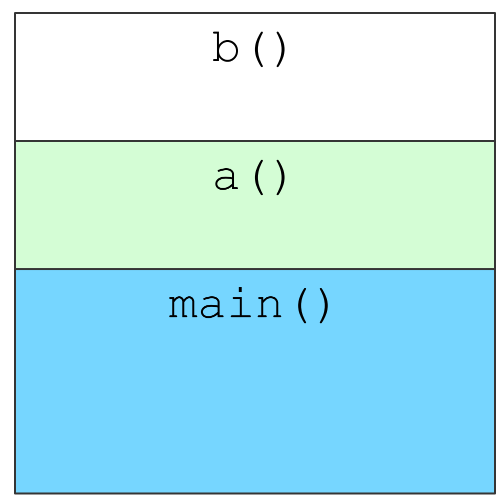
Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main(int argc,
   char **argv) {
    /* … */
    a();
}
```

When `b()` finishes running, its stack frame is removed!

What happens to the memory?

Stack Memory:

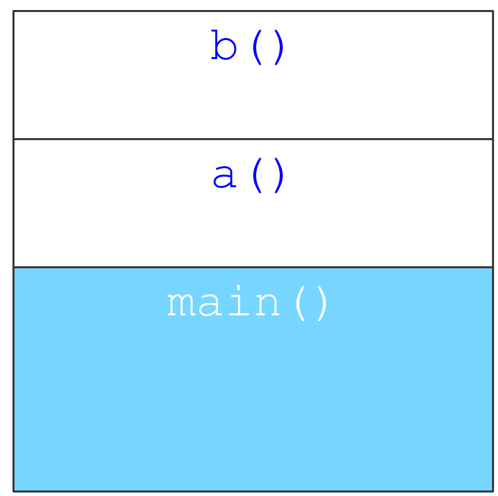


Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main(int argc,
    char **argv) {
    /* … */
    a();
}
```

And so on …

Stack Memory:



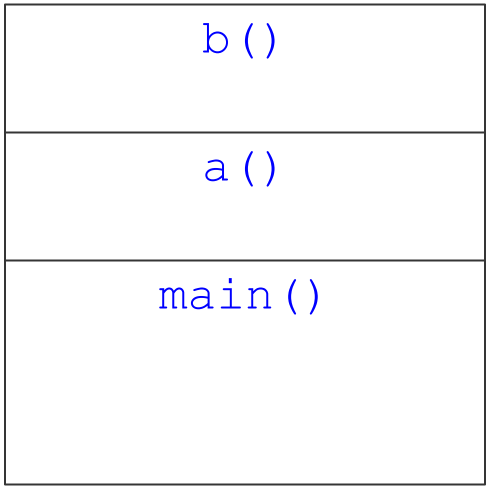
Example

And so on …

```
int b() {
    /* … */
}

int a() {
    /* … */
    b();
}

int main(int argc,
    char **argv) {
     /* … */
     a();
}
```

So What?

Stack Memory:



Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}

int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

```
int main(int argc,
     char **argv) {
     int myVal = 3;
     a(myVal);
}
```

`main()` still calls `a()`
`a()` still calls `b()`
`b()` returns a pointer to `a()`
`a()` returns an `int` to `main()`
`my_queue` is a custom `struct`

Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}

int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

```
int main(int argc,
    char **argv) {
    int myVal = 3;
    a(myVal);
}
```

**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}

int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

```
int main(int argc,
        char **argv) {
        int myVal = 3;
        a(myVal);
}
```

myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}

int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

```
int main(int argc,
        char **argv) {
        int myVal = 3;
        a(myVal);
}
```

yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}
```

```
int main(int argc,
     char **argv) {
     int myVal = 3;
     a(myVal);
}
```

```
int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

myVal (4 bytes)	???????
yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}
```

```
int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

```
int main(int argc,
    char **argv) {
    int myVal = 3;
    a(myVal);
}
```

*myQueue (8 bytes)	???????
myVal (4 bytes)	???????
yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}
```

```
int main(int argc,
    char **argv) {
    int myVal = 3;
    a(myVal);
}
```

```
int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

*myQueue (8 bytes)	???????
myVal (4 bytes)	6
yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

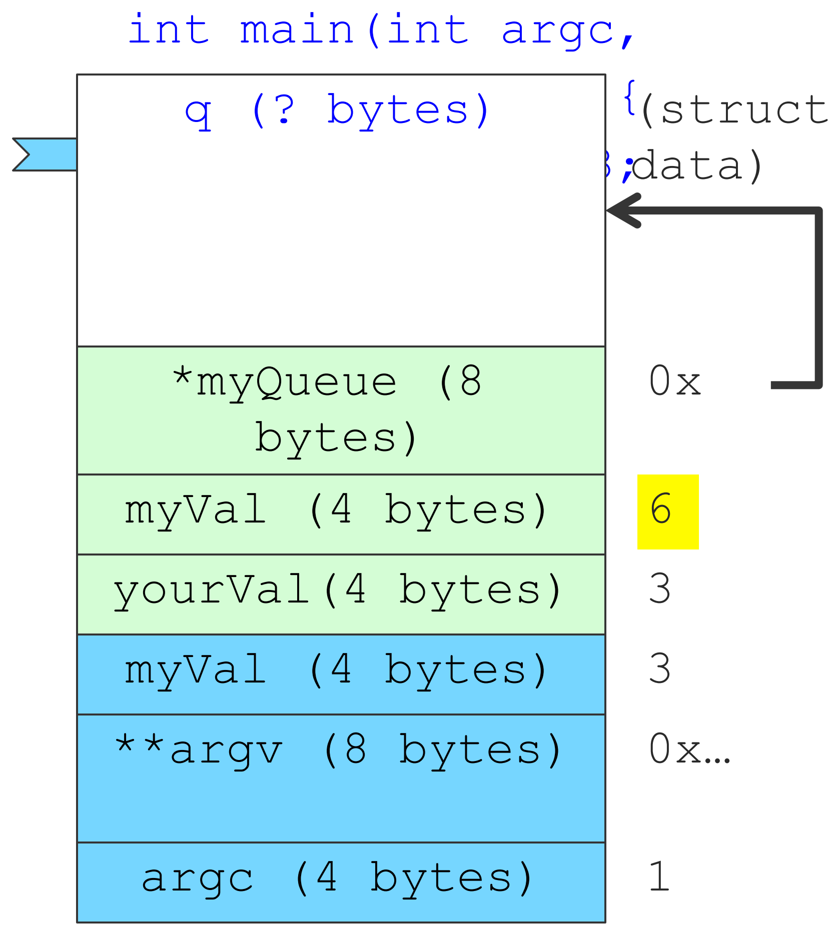
```
my_queue * b() {
    my_queue q;
    return &q;
}

int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

```
int main(int argc,
{(struct
;data)
```

q (? bytes)	
*myQueue (8 bytes)	???????
myVal (4 bytes)	6
yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

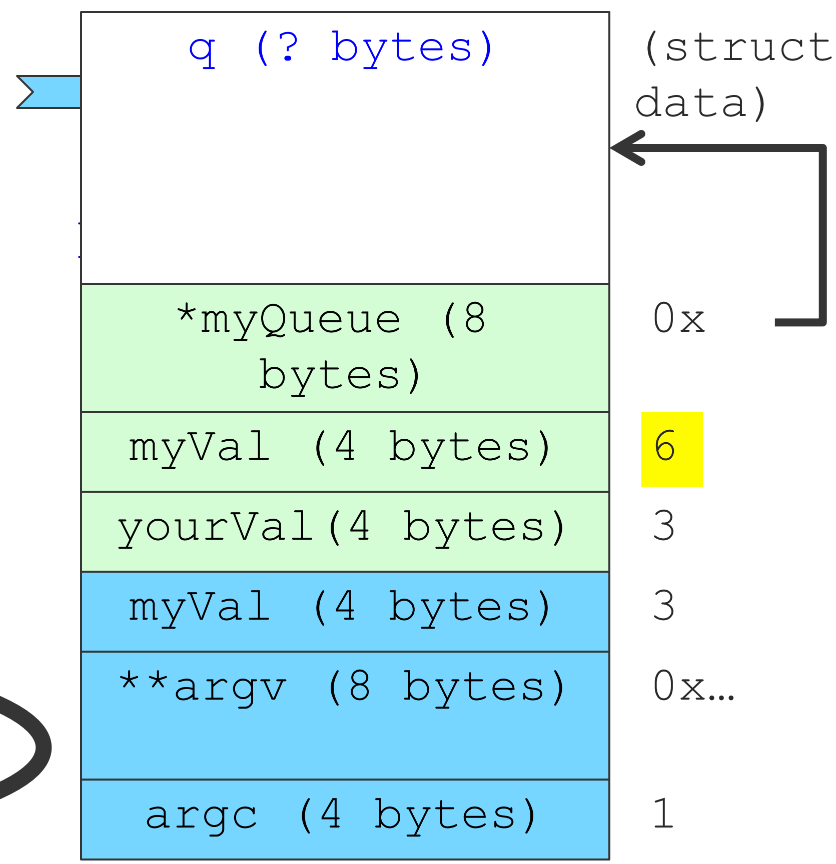
```
my_queue * b() {
    my_queue q;
    return &q;
}

int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```



Better Example

```
my_queue * b() {
    my_queue q;
    return &q;
}

int a(int yourVal) {
    int myVal;
    my_queue *myQueue;
    myVal = yourVal + 3;
    myQueue = b();
    return
      remove_int(myQueue);
}
```

int main(int argc,



Use your stack wisely

- Returning a pointer to a stack variable results in unpredictable behavior
- Three 'common' fixes
  - Good: Pass in a pointer to the variable you want to use
  - Good: Use a heap variable
  - Very Bad (usually): Use a global variable