C No Evil

A practitioner's guide

Playing with fire

- Program arguments
- Pointer arithmetic
- Output
- Stack memory

ARGCount ARGValues

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

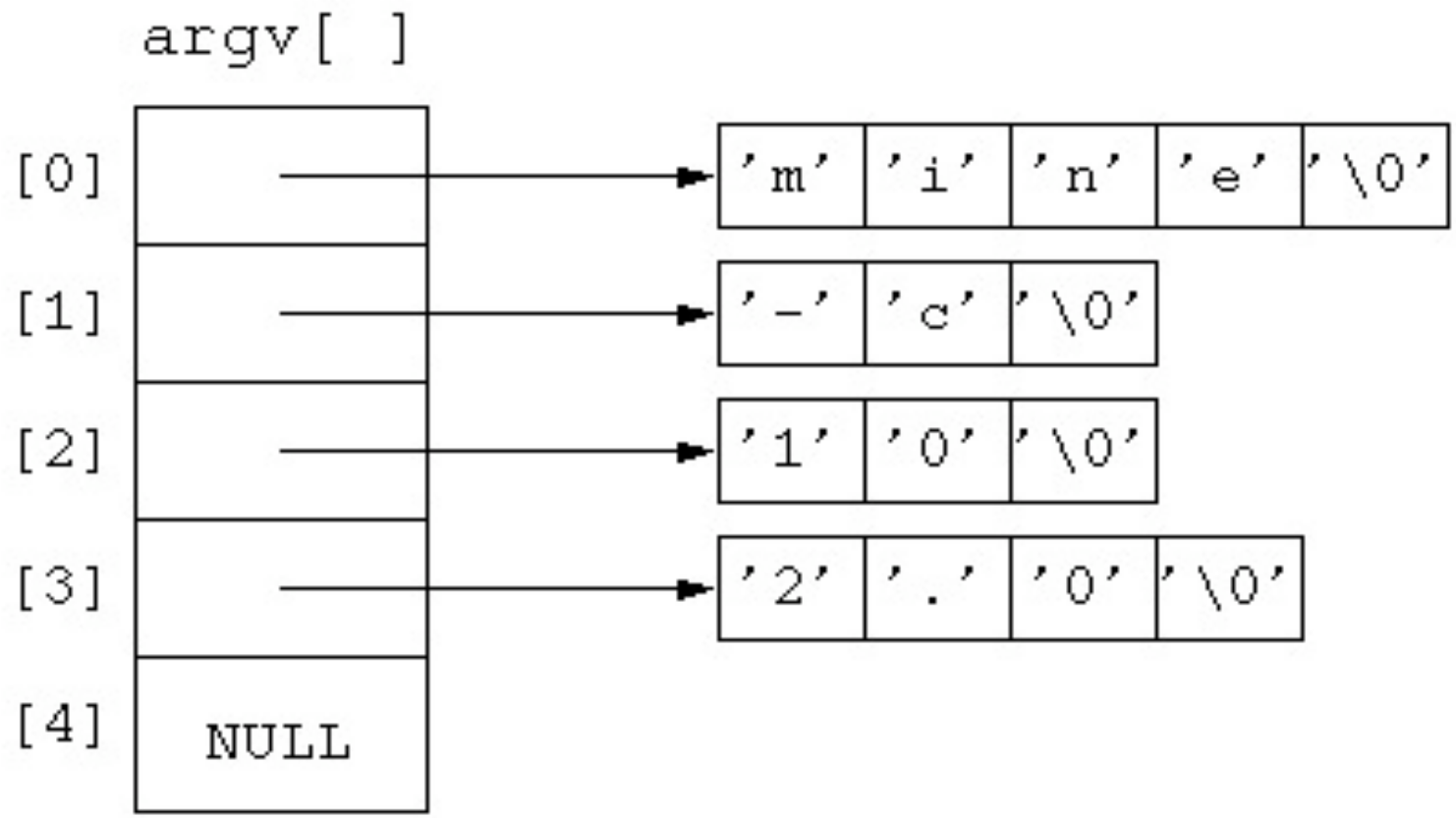
- **`argc`**
  - Argument count
  - The number of arguments that are passed to **`main`** in the argument vector **`argv`**.
  - the value of **`argc`** is always one greater than the number of command-line arguments that the user enters.

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



ARGCount ARGValues

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

- `*(argv + argc)` is `NULL`
- `argv[argc]` is `NULL`

Type questions

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

Followup: Can add integers to pointers

- Compiler uses the type information
  - `long *p;`
  - `p ▸ [long][long][long]`
- What address is `p + 2`?
  - … `p + sizeof(long) * 2`

Followup: output

- C stdio library functions

```
printf("Hello %x %s %d", arguments…)
fprintf(STDERR, "%x%s%d",…)
```

- Later… system call
  `write(int,void*,size_t)`

`printf` Format Identifiers

`%d %i`	Decimal signed integer.
`%o`	Octal integer.
`%x %X`	Hex integer.
`%u`	Unsigned integer.
`%c`	Character.
`%s`	String.
`%f`	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>

Typecasting

- C allows programmers to perform typecasting by
  - Place the type name in parentheses and place this in front of the value

```
main() {
    float a;
    a = (float)5 / 3;
}
```

- Result is a = 1.666666
  - Integer 5 is converted to floating point value before division and the operation between float and integer results in float
- What would `a` be without the `(float)`?

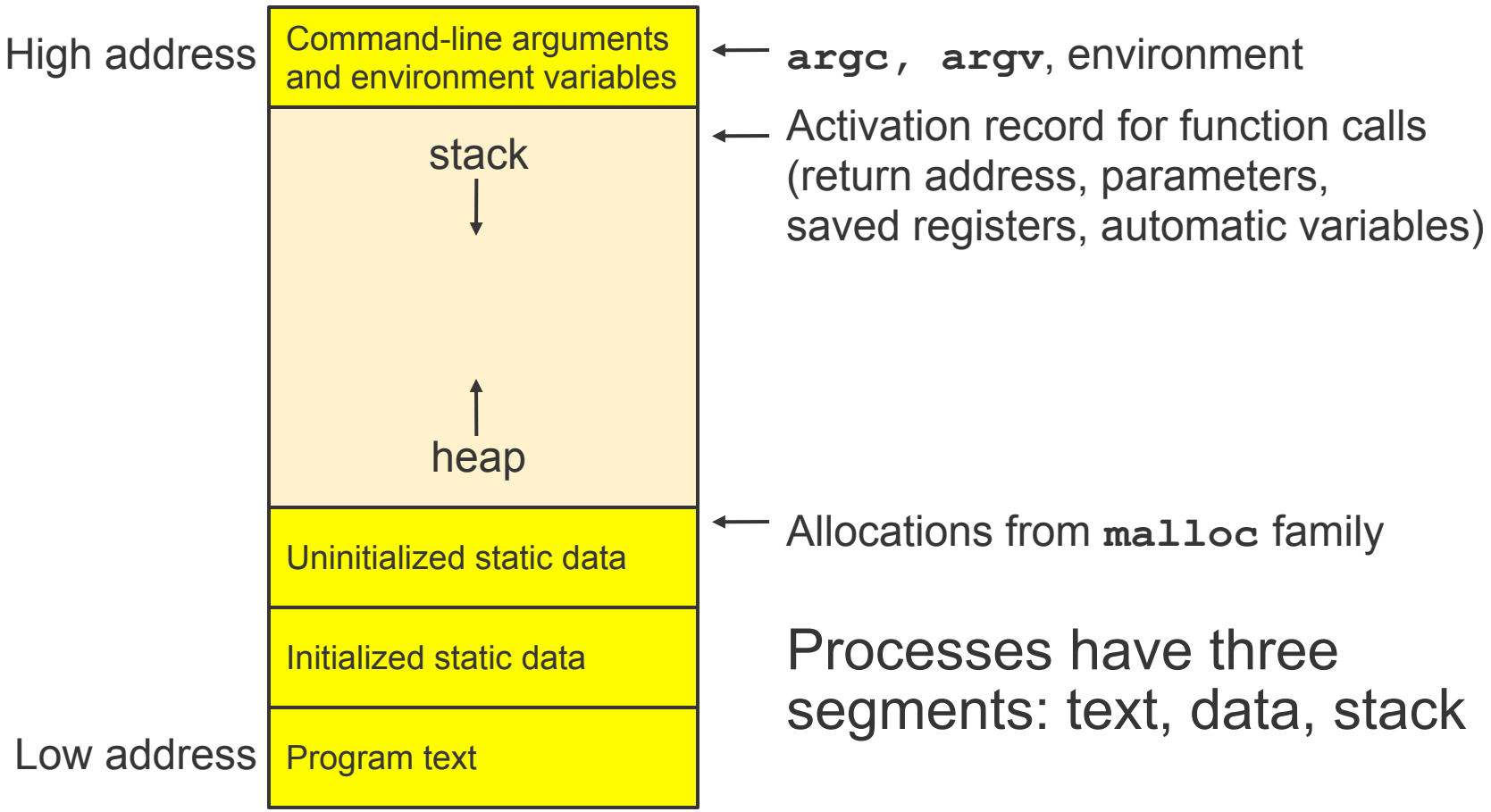
Typecasting

- Take care about using typecast
- If used incorrectly, may result in loss of data
  - e.g., truncating a `float` when casting to an `int`

Common Pitfall

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

Sample Layout for program image in main memory



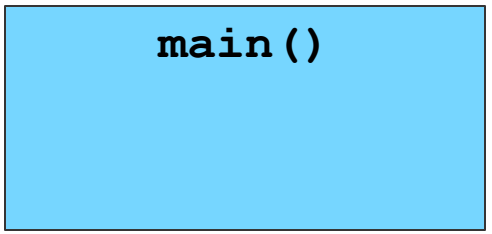
Processes have three segments: text, data, stack

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:

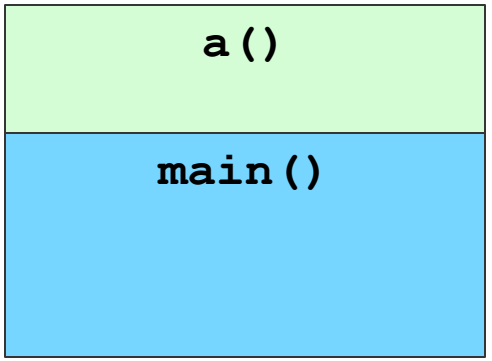


Example

```
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:

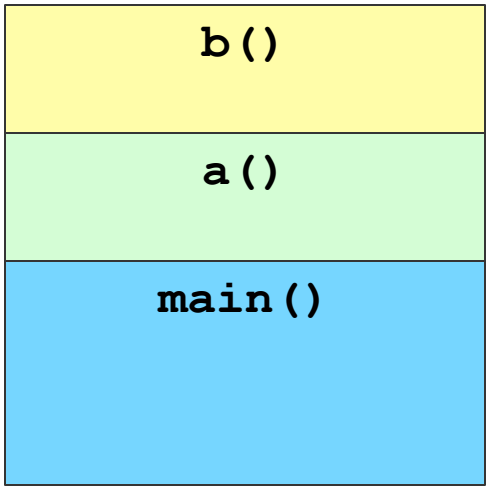


Example

```
int b() {
    /* … */
}

int a() {
    /* … */
    b();
}

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

Same for `b()` ...

Stack Memory:



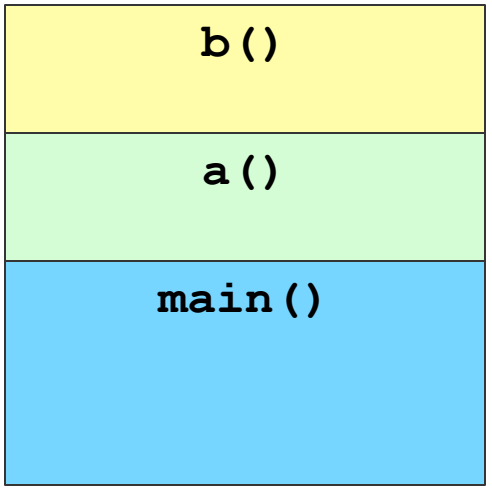
Example

```
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:

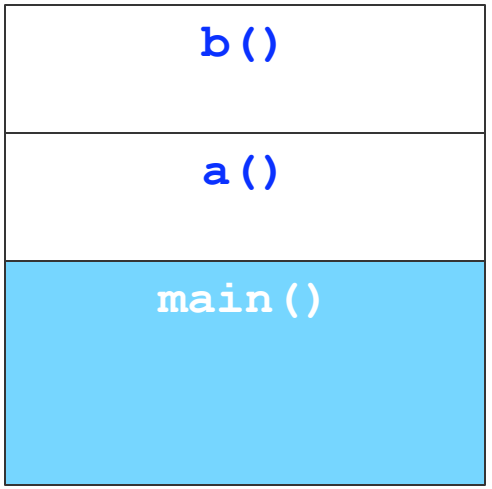


Example

```
int b() {
    /* … */
}

int a() {
    /* … */
    b();
}

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

And so on …

Stack Memory:

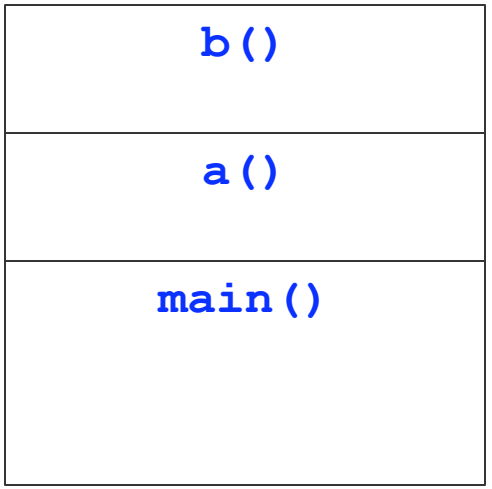


Example

```
int b() {
    /* … */
}

int a() {
    /* … */
    b();
}

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

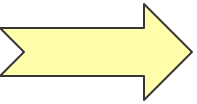
And so on …

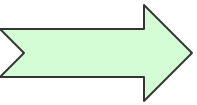
Stack Memory:

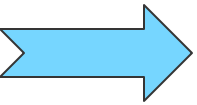


Example

```
int b() {
    /* … */
}

int a() {
    /* … */
    b();
}

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



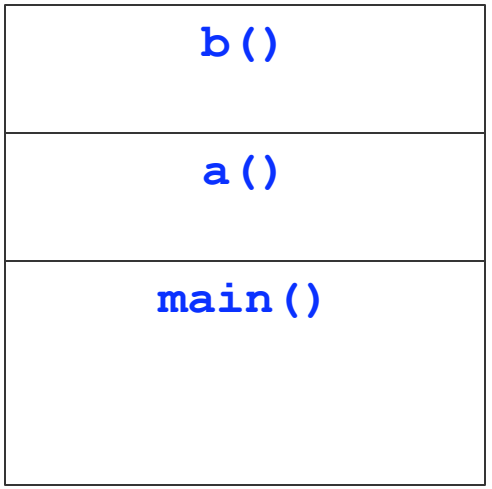




And so on …

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

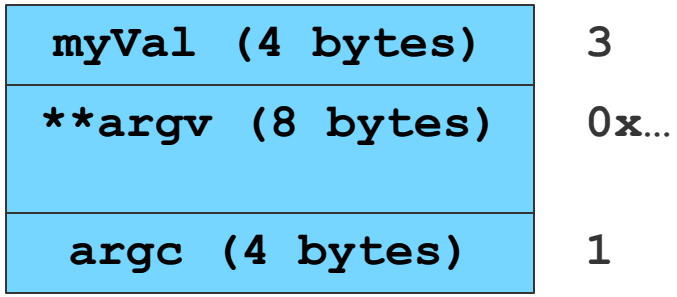
```
my_queue * b() {
    my_queue q;
    return &q;
}

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);
}
```

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);
}
```

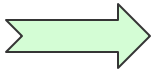


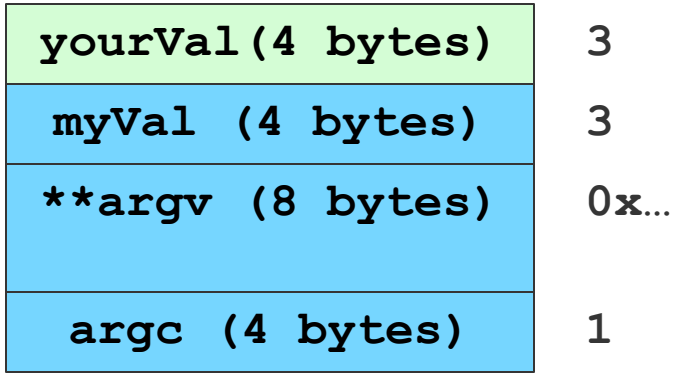
Better Example

```
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);
}
```

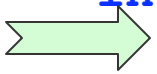




Better Example

```
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);
}
```



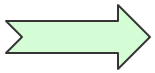
myVal (4 bytes)	???????
yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

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);
}
```

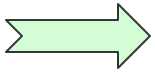


*myQueue (8 bytes)	???????
myVal (4 bytes)	???????
yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

Better Example

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

int 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);
}
```

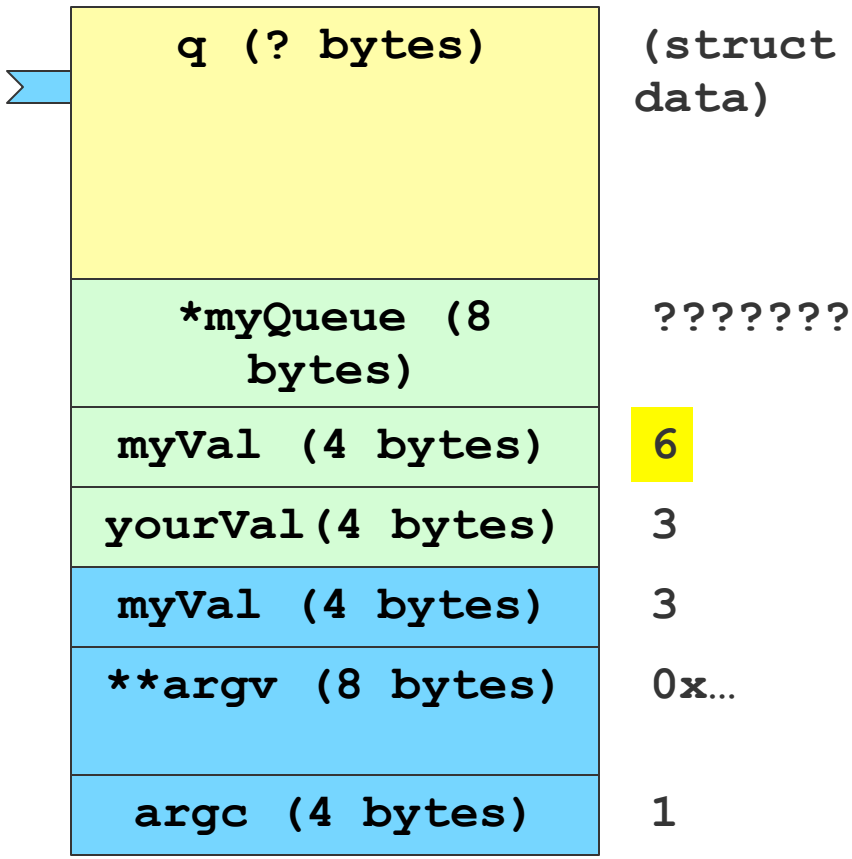


*myQueue (8 bytes)	???????
myVal (4 bytes)	6
yourVal(4 bytes)	3
myVal (4 bytes)	3
**argv (8 bytes)	0x…
argc (4 bytes)	1

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);
}
```

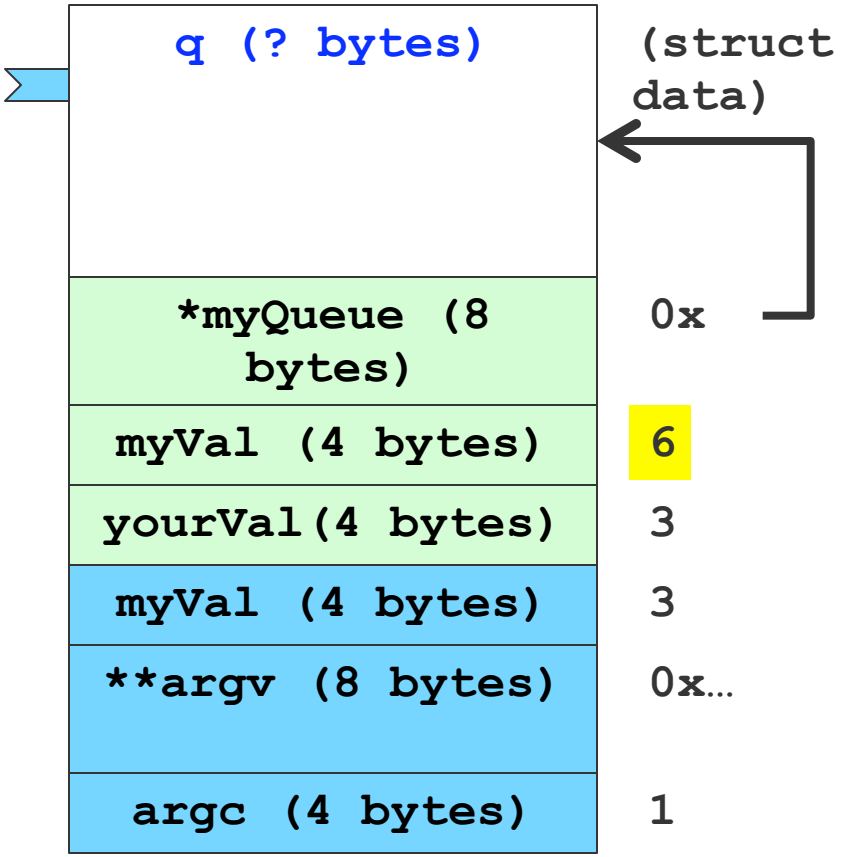
```
int main(int argc,
```



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);
}
```

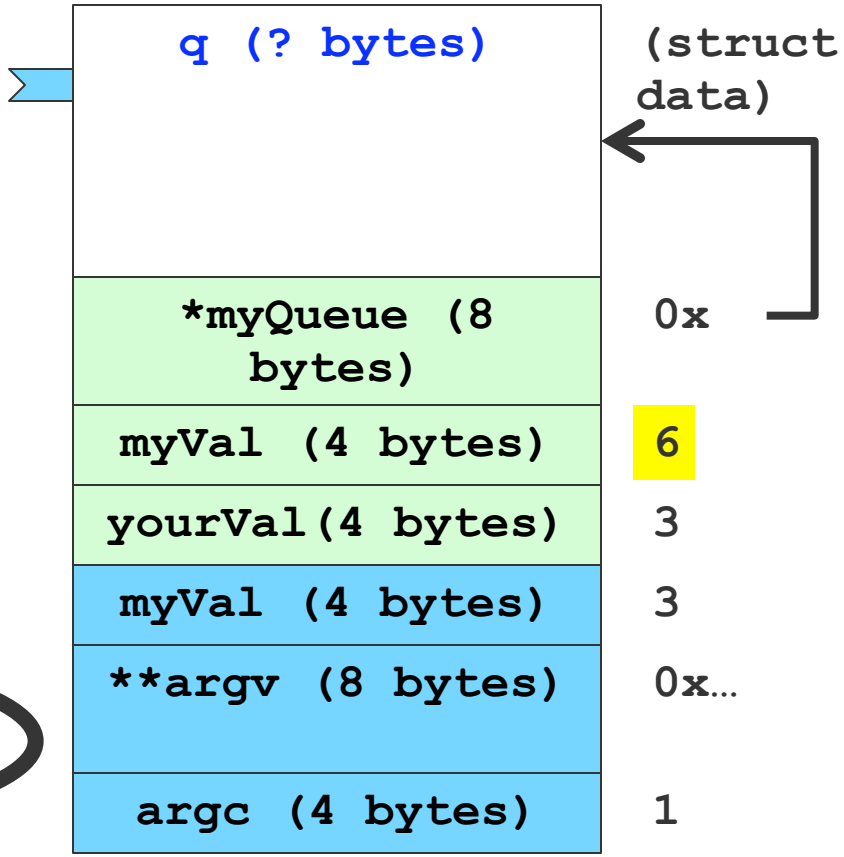
```
int main(int argc,
```



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: Use a global variable