C No Evil

A practitioner's guide

Playing with fire

- Typecasting
- Program arguments
- Pointer arithmetic
- Output
- Stack memory

Typecasting

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`

Typecasting pointers

```
int* p = 160;

printf("%p %p %p\n",
       p, p+1,
        ((char*) p) + 1
      );
```

- Does not change pointer value
- Does affect pointer arithmetic

Typecasting pointers

```
int* p = 160;

printf("%p %p %p\n",
       p, p+1,
       ((char*) p) + 1
      );
```

- It works, but compiler complains….

➔ warning: incompatible integer to pointer conversion initializing 'int *' with an expression of type 'int'

Typecasting pointers

```
int* p = (int*) 160;

printf("%p %p %p\n",
        p, p+1,
         ((char*) p) + 1
       );
```

- Does not change pointer value
- Does affect pointer arithmetic
- Avoids compiler warnings

What type are we using?

Are we dealing with strings or numbers?

```
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';
((char*)&x)[3] = '\0';


printf("This class is %s.\n", &x);
printf(”Hexadecimal value of x is %x.\n", x);
```

What type are we using?

```
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';
((char*)&x)[3] = '\0';
```

Are we dealing with strings or numbers?

They are just a sequence of bytes "interpreted" as different types

```
printf("This class is %s.\n", &x);
printf("Hexadecimal value of x is %x.\n", x);
```

```
Output:
This class is fun.                              |\0|n|u|f|
The dec and hex value of x are 7239014     6e7566.
```

What is endianness?

```
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';
((char*)&x)[3] = '\0';


printf("This class is %s.\n", &x);
printf(”Hexadecimal value of x is %x.\n", x);
```

Endianness: are integers represented with the most significant byte stored at the lowest address (big endian) or at the highest address (little endian)?

What is endianness?

```
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';
((char*)&x)[3] = '\0';
```

Quiz: based on the output, can you tell what is the endianness of this machine?

```
printf("This class is %s.\n", &x);
printf("Hexadecimal value of x is %x.\n", x);
```

```
Output:
This class is fun.                              |\0|n|u|f|
The dec and hex value of x are 7239014    6e7566.
```

ARGCount ARGValues

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

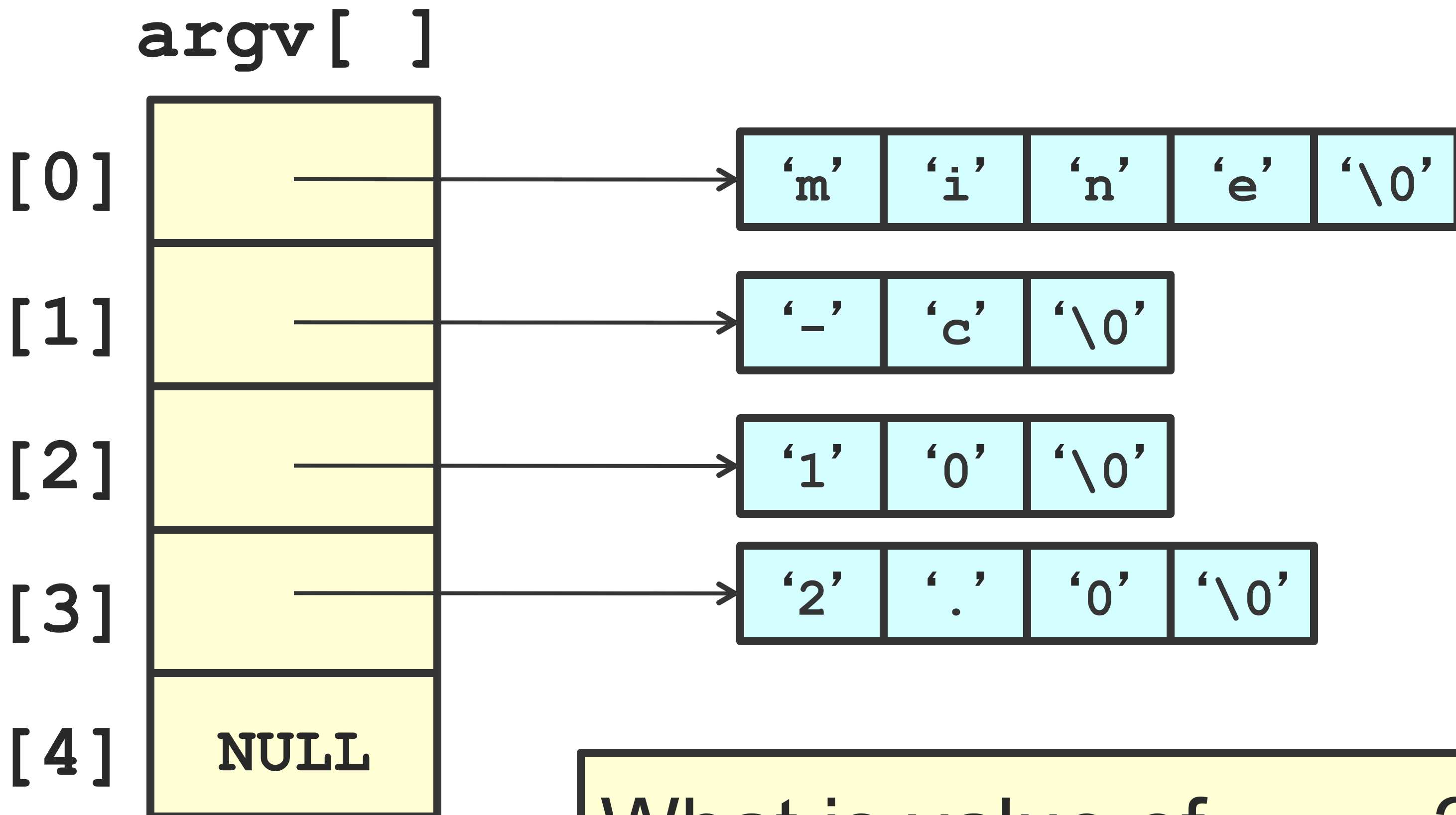
- `argc`
  - Argument count
  - The number of arguments that are passed to `main` in the argument vector `argv`.
  - the value of `argc` is always one greater than the number of command-line arguments that the user enters.

ARGCount ARGValues

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

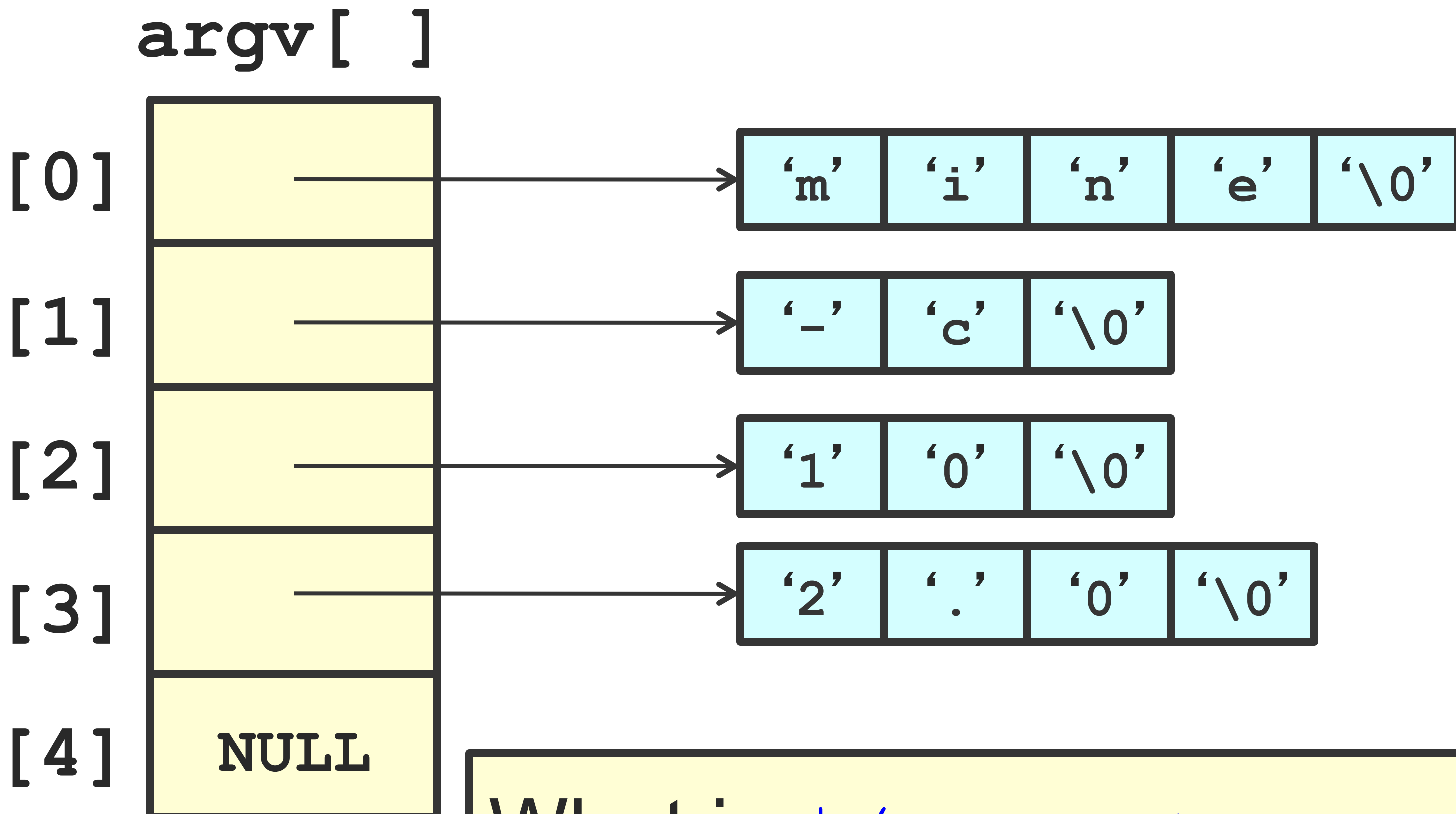
- `argv`
  - argument vector
  - An array of string pointers passed to a C program's `main` function
  - `argv[0]` is always the name of the command
  - `argv[argc]` is a null pointer

ARGCount ARGValues



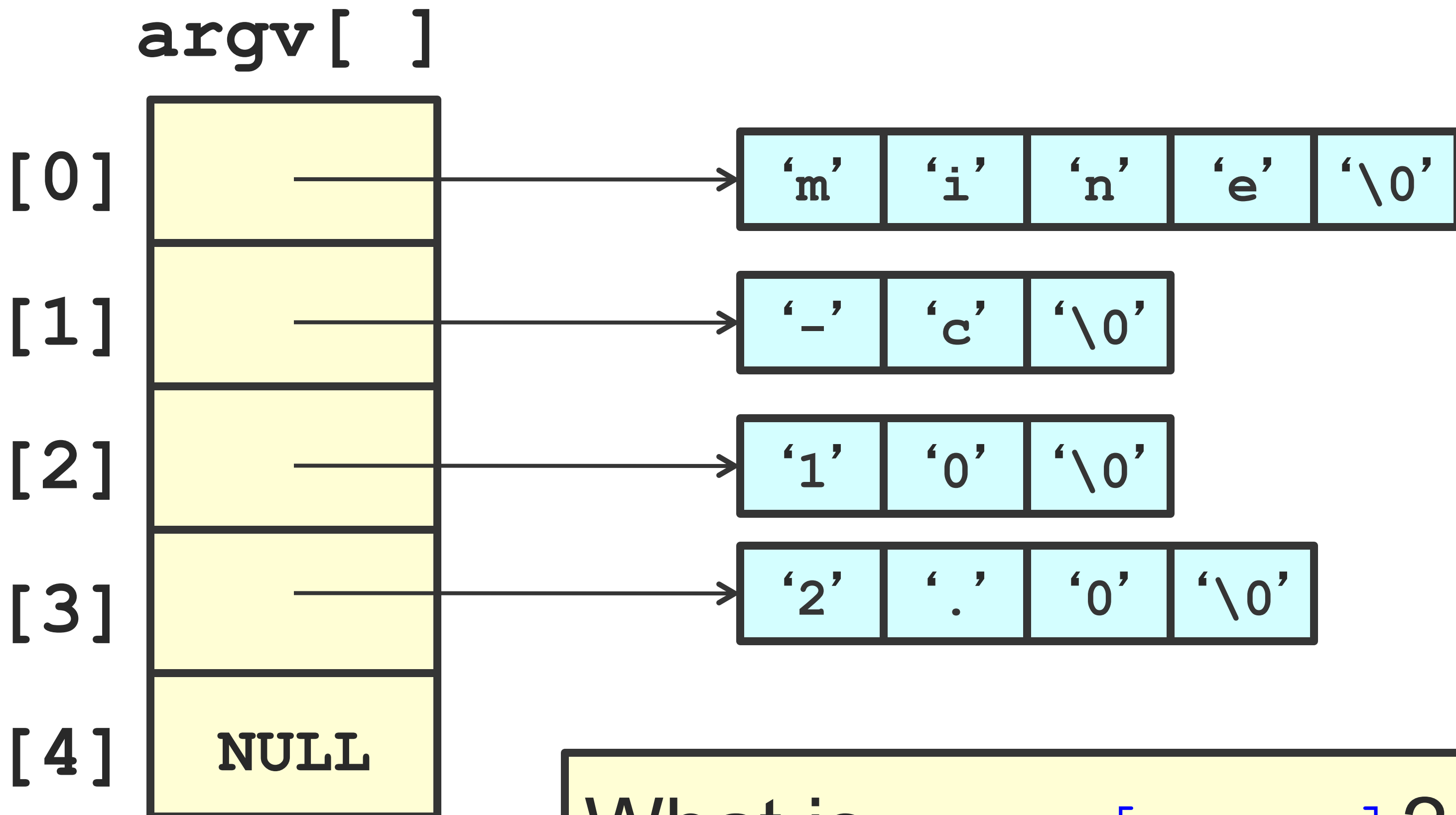
What is value of `argc`?

ARGCount ARGValues



What is `*(argv + argc)`?

ARGCount ARGValues



What is `argv[argc]`?

Type questions

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

Adding integers to pointers

- Compiler uses the type information
  - `long *p;`
  - `p` ▶ `[long][long][long]`

- What address is `p + 2`?
  - … `p + sizeof(long) * 2`

Output

- C stdio library functions

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

`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

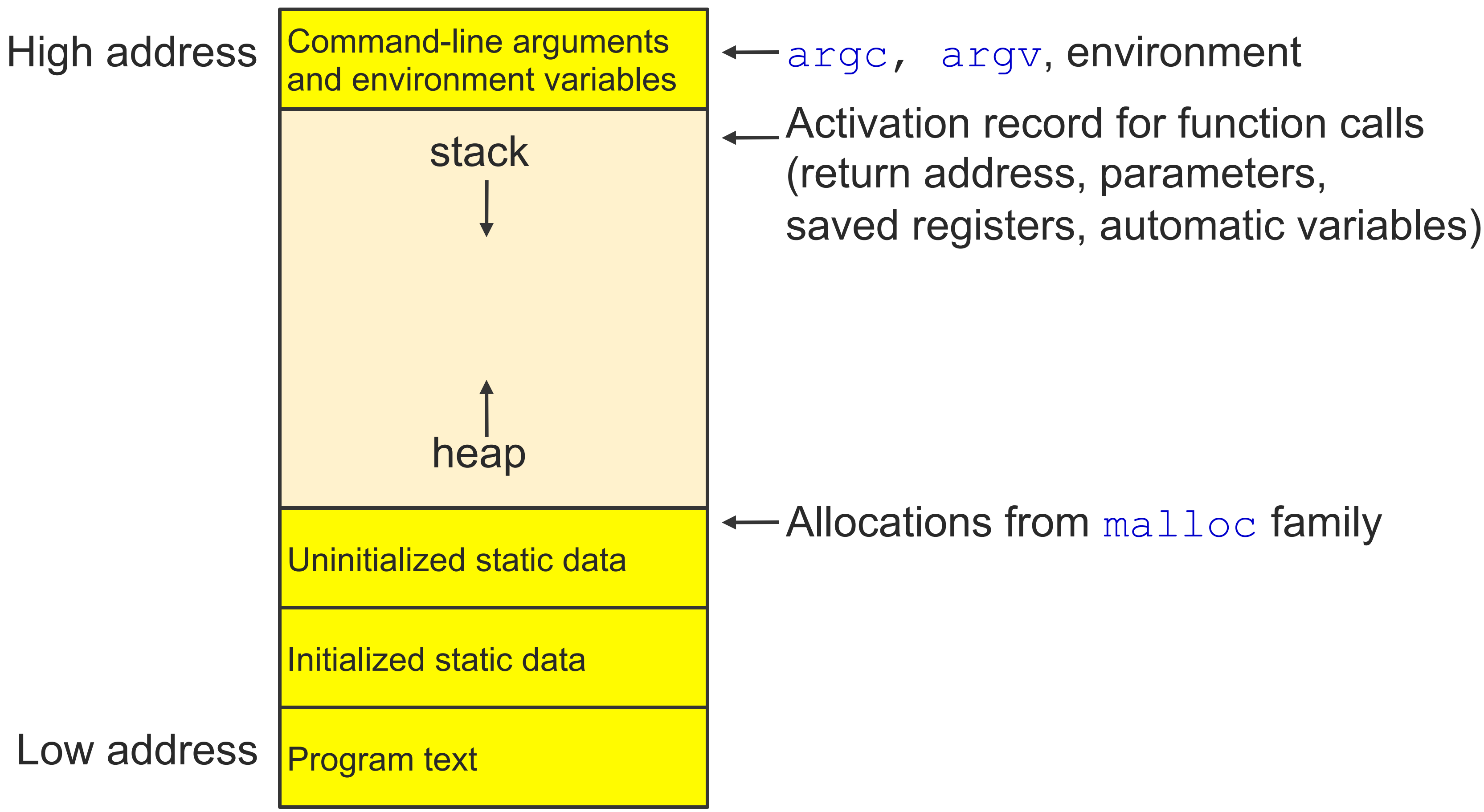
printf Basic Data Types

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

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

    // - 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: %X %X %X %X\n",uc1,uc2,uc3,uc4);
}
```

Common Pitfall

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

Sample layout for program image in main memory



Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

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

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:

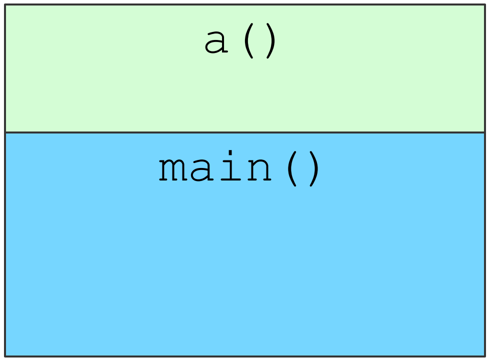
main()

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:

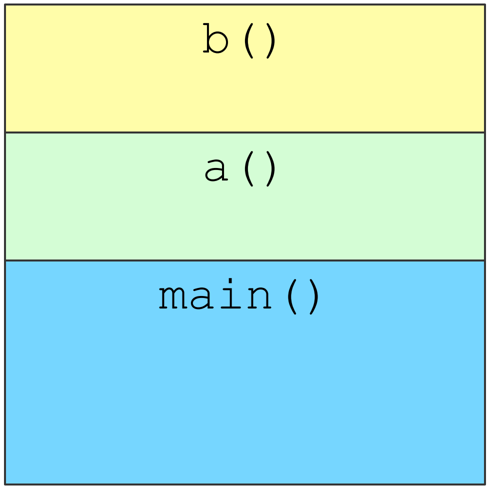


Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

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

Same for `b()` …

Stack Memory:



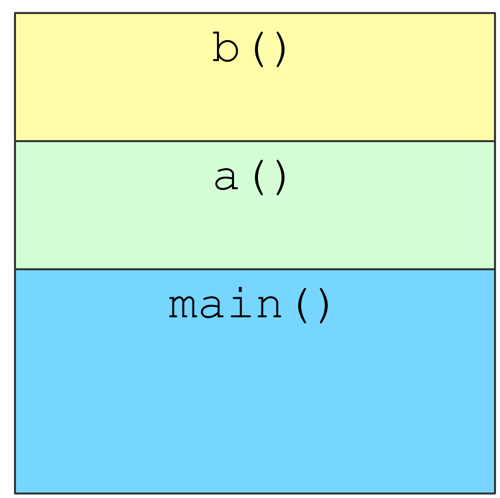
Example

```
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:



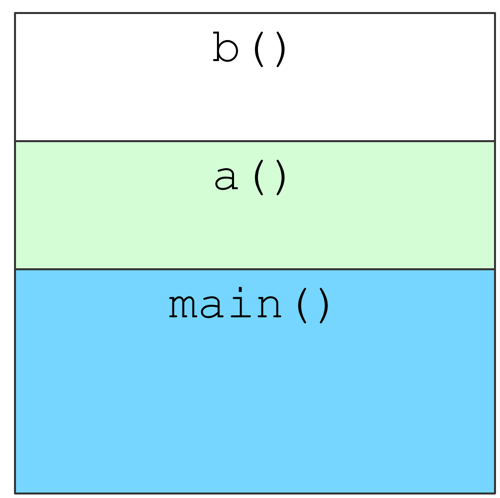
Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

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

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

What happens to the memory?

Stack Memory:

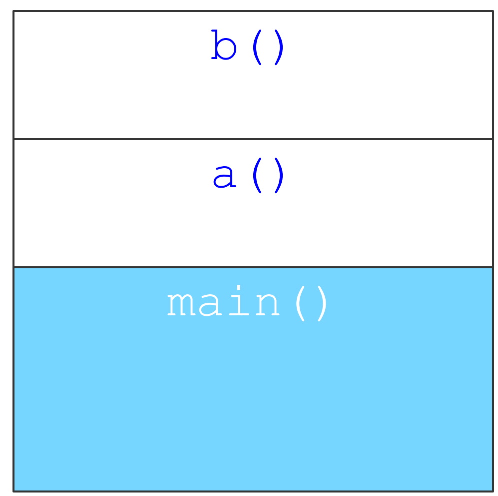


Example

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

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

And so on …

Stack Memory:



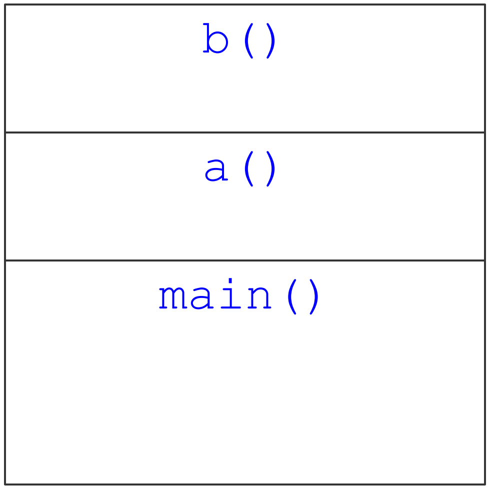
Example

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);
}
```

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);
}
```

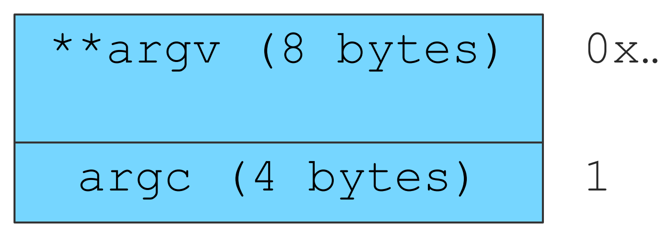
```
int main(int argc,
    char **argv) {
    int myVal = 3;
    a(myVal);
}
```

`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);
}
```

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;
}
```

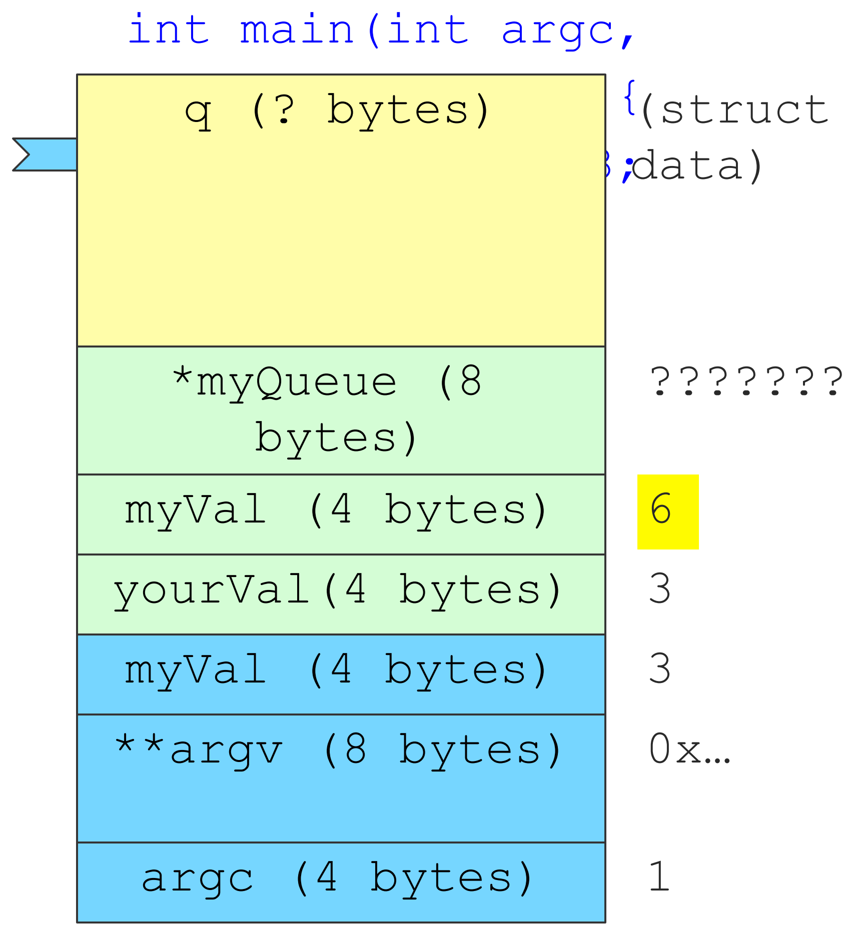
```
int main(int argc,
    char **argv) {
    int myVal = 3;
    a(myVal);
}
```

```
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

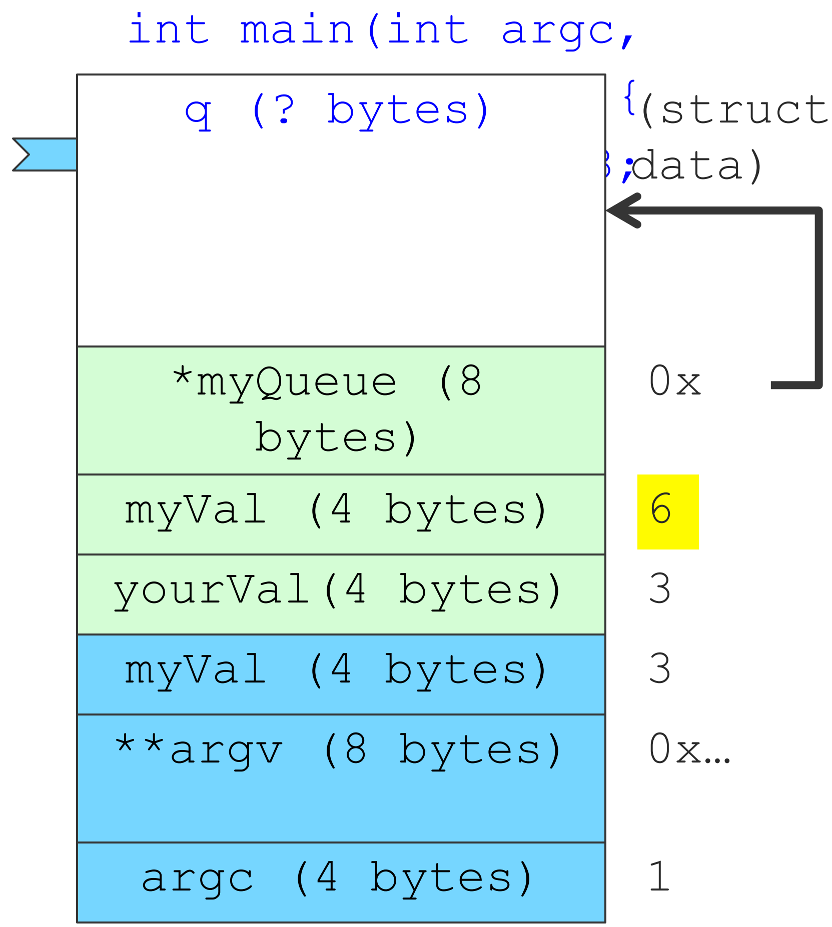
Better Example

```
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);
}
```



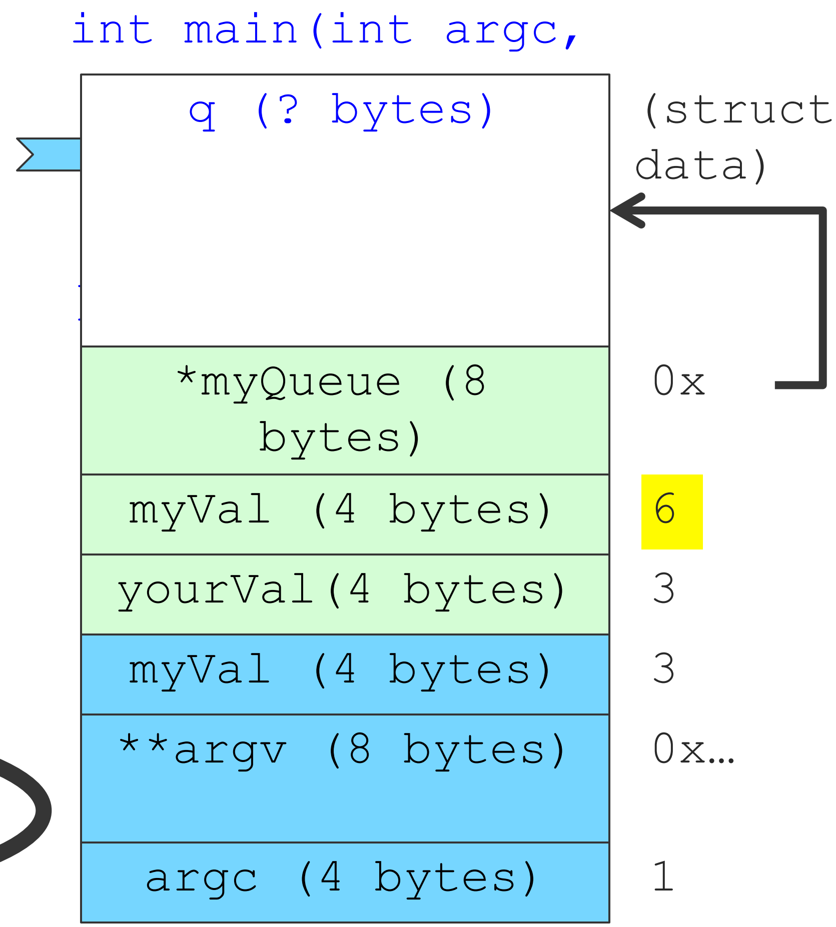
Better Example

```
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);
}
```



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
  - Bad (usually): Use a global variable