UNIVERSITY of WASHINGTON

LECTURE 3 · CSE 333 · Summer 2026

# Pointers; Memory

**Discussion:** Have you been following any of the World Cup games?

# Reminders

**Assessments**:

- Exercise 1 due yesterday at 11:59pm
  - Can still use one late day before we automatically release solutions
- Exercise 2 released today at some point. Due next Thursday at 11:59pm
- Homework 1 released soon. Due on Thursday of week 3.

**General**:

- Notify us ASAP if you have conflicts with
  - Midterm (7/27) from 9:40am to 10:40am in IEB G106
  - Final exam (8/18) from 1:30pm to 3:30pm in CSE2 G10
- Participation points will be updated over the weekend

# ~~Um, Actually~~

**(CSE 333-edition)**

# Corrections

# Oops!

```c
#include <stdio.h>
#include <stdlib.h>

void Greet(void) {
  char* course = "CSE 333";
  printf("Hello, %s!\n", course);
}

int main(int argc, char* argv[]) {
  Greet();
  return EXIT_SUCCESS;
}

```

# Oops!

```c
#include <stdio.h>
#include <stdlib.h>

// Prints a friendly greeting to stdout
void Greet(void);

int main(int argc, char* argv[]) {
  Greet();
  return EXIT_SUCCESS;
}

void Greet(void) {
  char* course = "CSE 333";
  printf("Hello, %s!\n", course);
}
```

**Reminder:**

Many of my examples include functions defined above main.

In your submissions, functions should **always** be declared above main (or in a local header file) and defined below it.

Additionally, function declarations should always have a comment above them.

No deductions in ex1, but in the future…

# Compilation

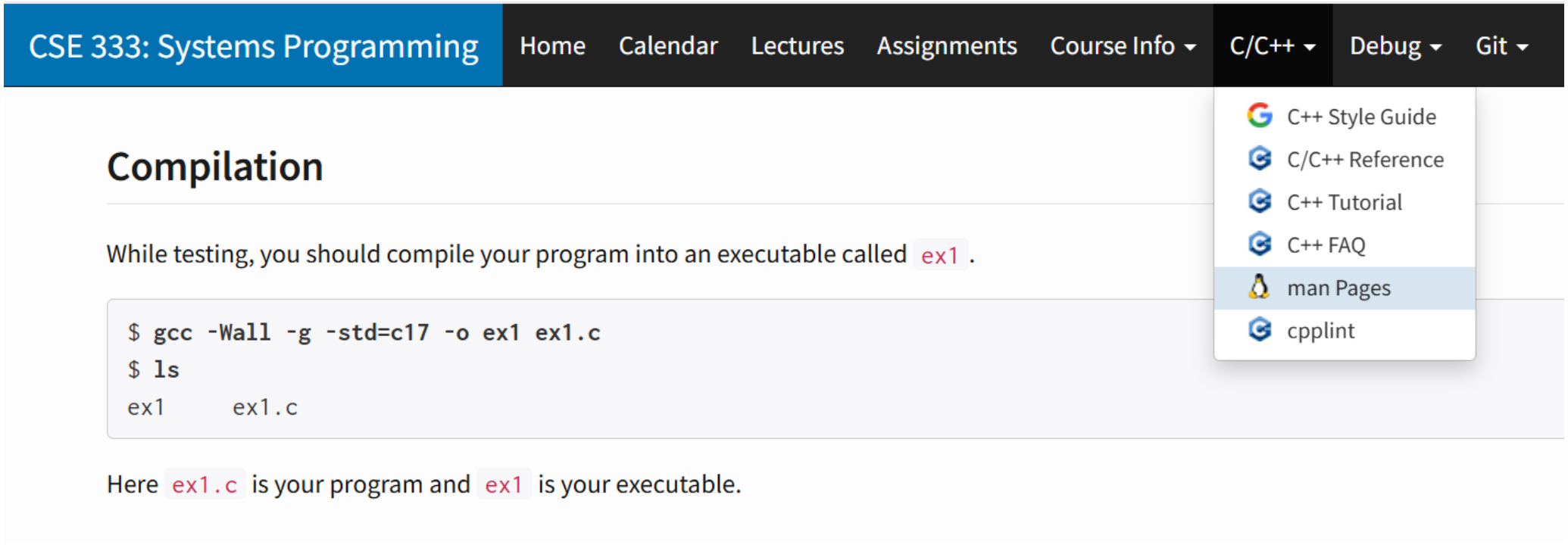
# Using gcc

In the homework, we ask you to run this:

```
$ gcc -Wall -g -std=c17 -o ex1 ex1.c
```

But what does it do? There are a few ways to find out…

# (1) man pages


CSE 333: Systems Programming
Home
Calendar
Lectures
Assignments
Course Info
C/C++
Debug
Git
C++ Style Guide
C/C++ Reference
C++ Tutorial
C++ FAQ
man Pages
cpplint
Compilation
While testing, you should compile your program into an executable called ex1 .
$ gcc -Wall -g -std=c17 -o ex1 ex1.c
$ ls
ex1 ex1.c
Here ex1.c is your program and ex1 is your executable.

# (1) man pages

**-g** Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

On most systems that use stabs format, **-g** enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but probably makes other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use **-gstabs+**, **-gstabs**, **-gxcoff+**, **-gxcoff**, or **-gvms** (see below).

**-Wall**

This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings described in **C++ Dialect Options** and **Objective-C and Objective-C++ Dialect Options**.

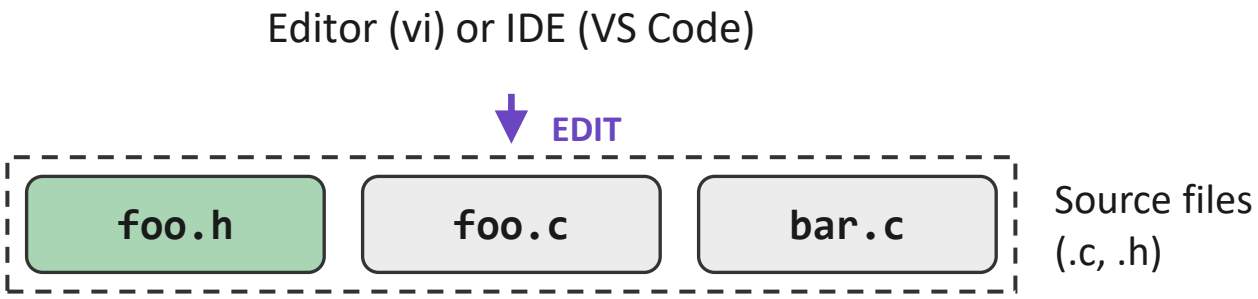
**-Wall** turns on the following warning flags:

**-Waddress -Warray-bounds=1** (only with **-O2**) **-Wbool-compare -Wbool-operation -Wc++11-compat -Wc++14-compat -Wcatch-value**

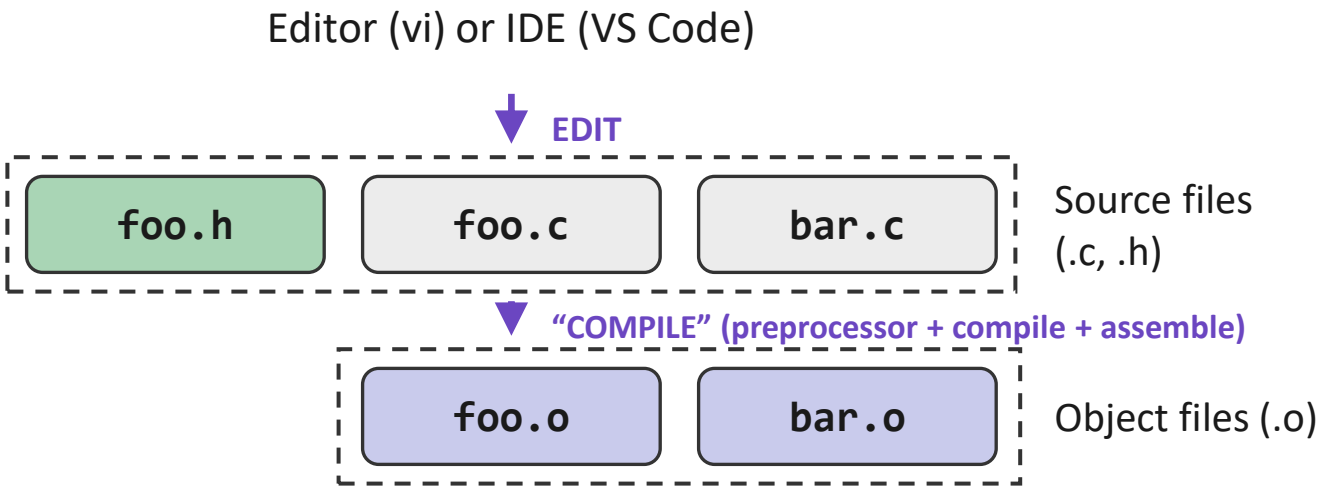
# (2) Just try it!

```
$ cd ./ex1/
$ ls
ex1.c
$ gcc -Wall -g -std=c17 -o ex1 ex1.c
$ ls
ex ex1.c
$ ./ex1
some output...
```

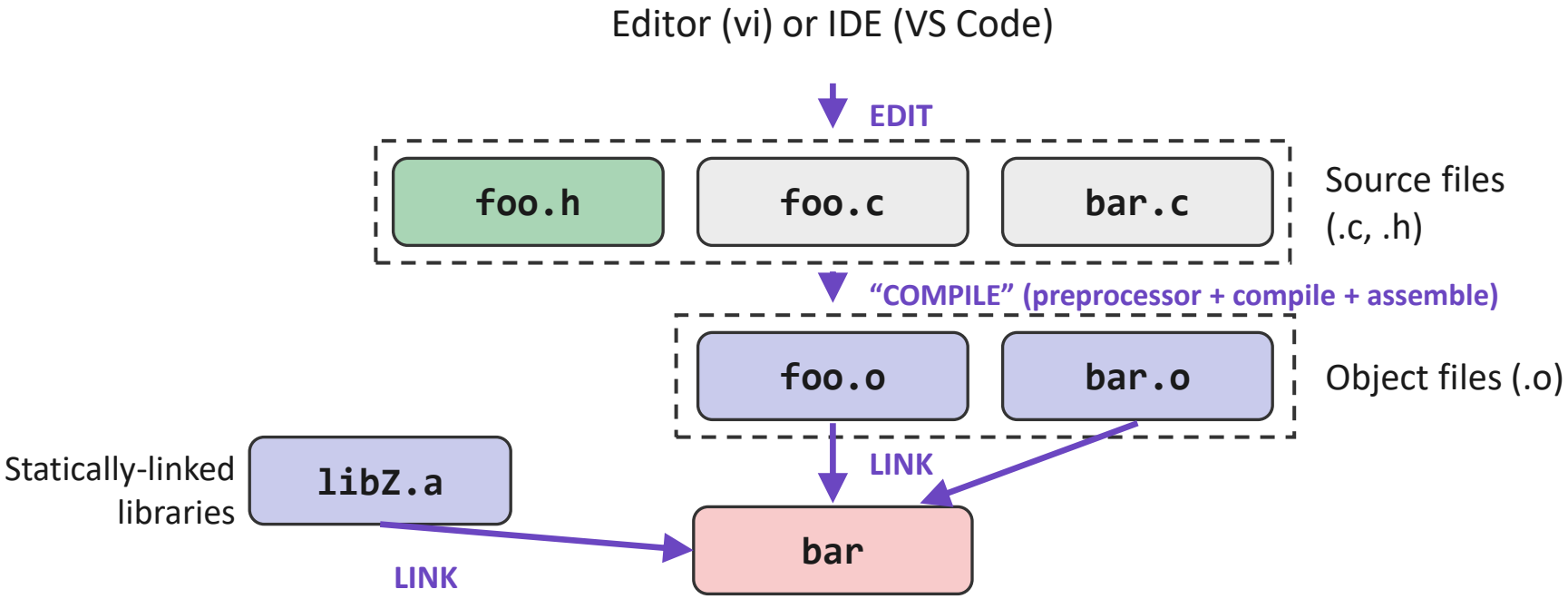
# (3) Read about it



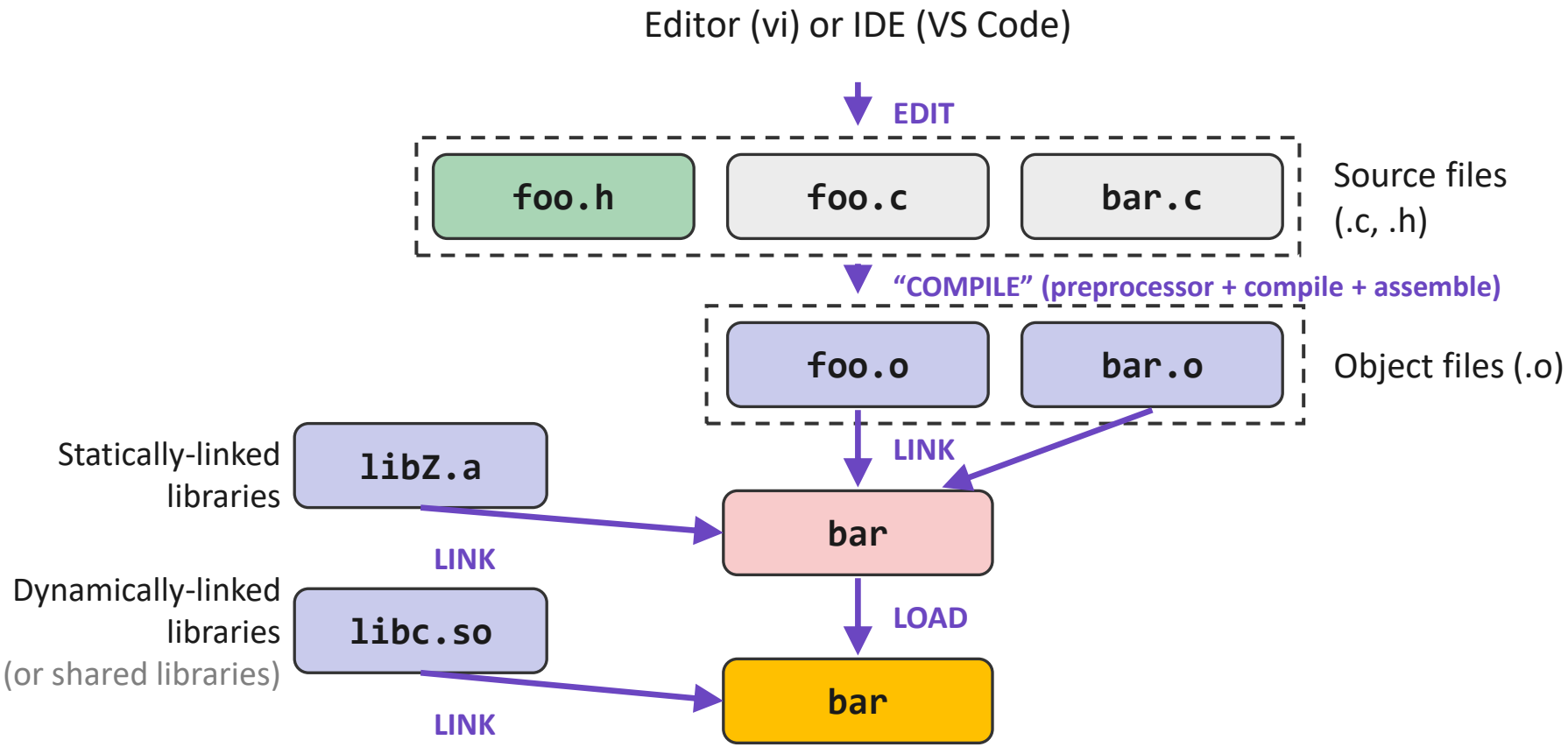
# (3) Read about it



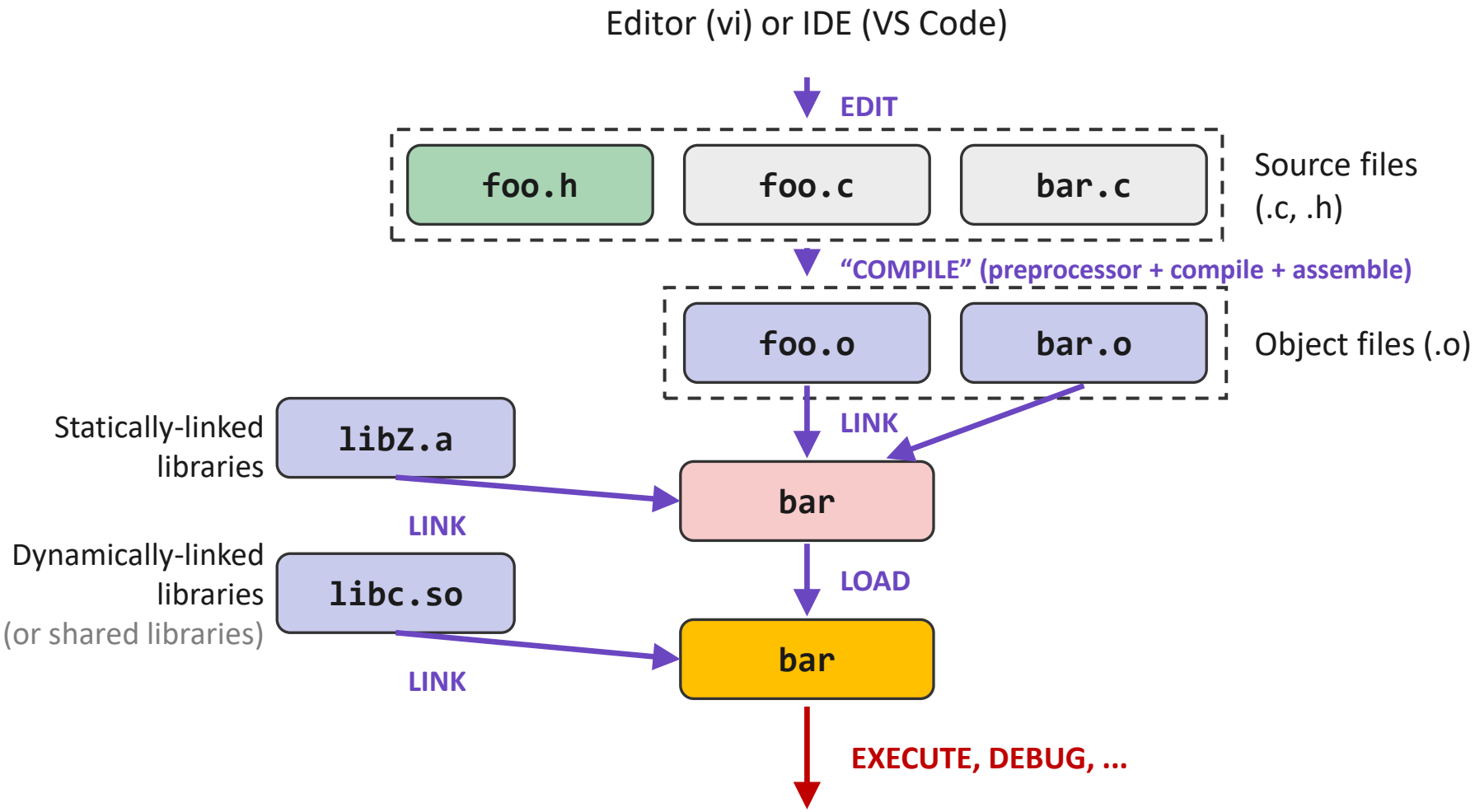
# (3) Read about it



# (3) Read about it



# (3) Read about it



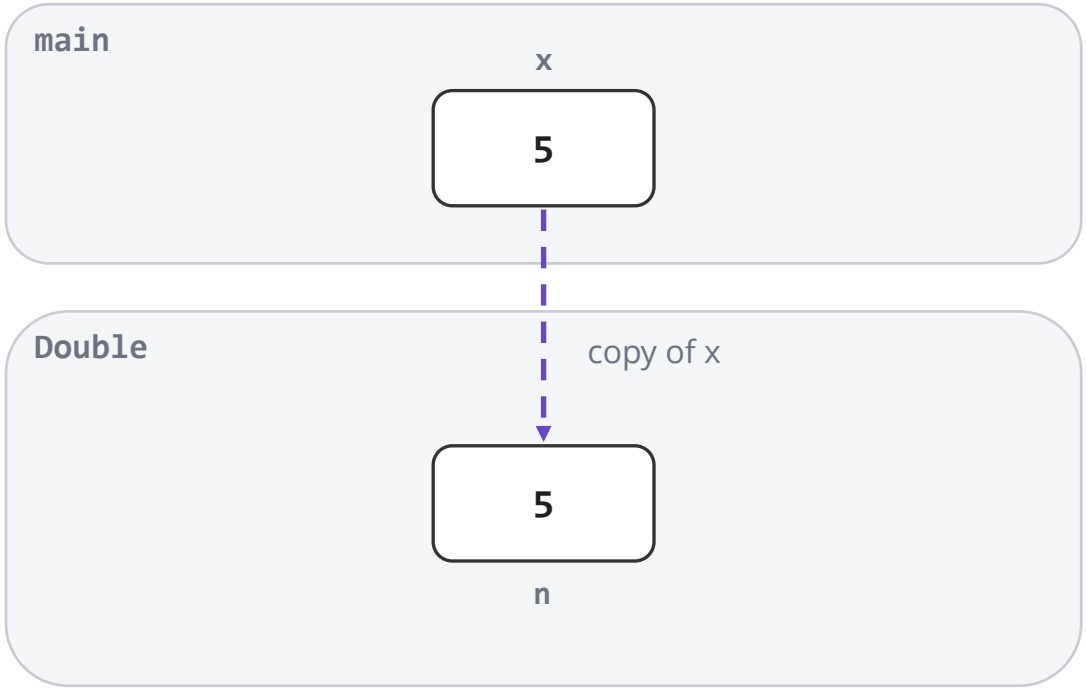
# Reference vs. Value

# Pass-by-Value

```
1  // Tries to double n.
2  void Double(int n) {
3    n = n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    Double(x);
9    printf("x = %d\n", x);  // 5, not 10!
10   return EXIT_SUCCESS;
11 }
```

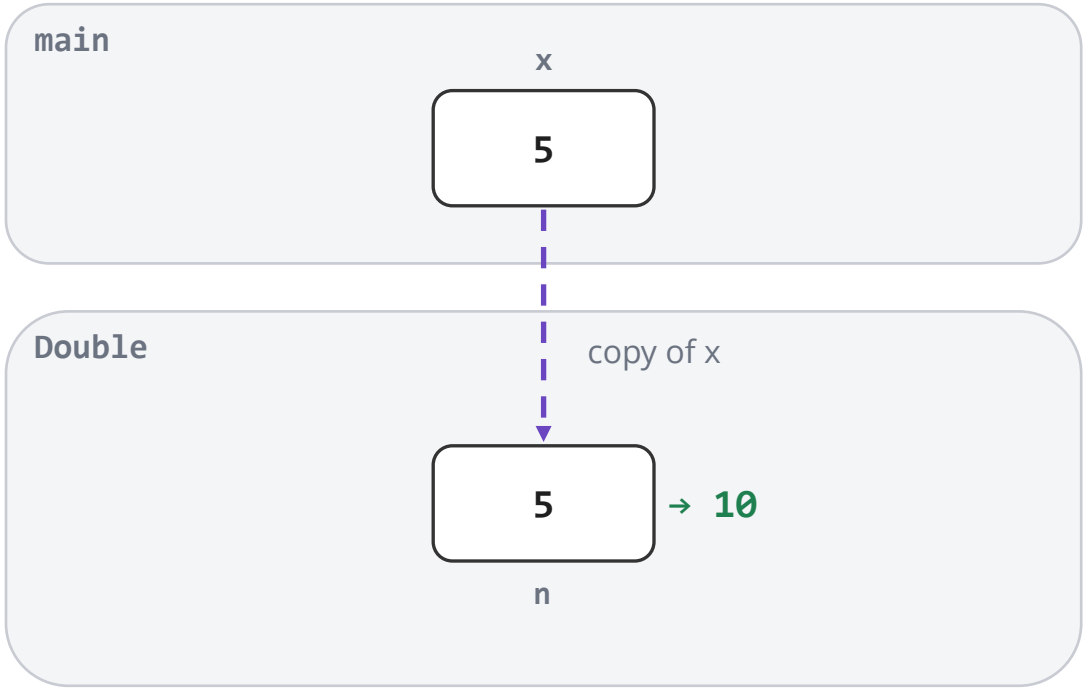
# Pass-by-Value

```
1  // Tries to double n.
2  void Double(int n) {
3    n = n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    Double(x);
9    printf("x = %d\n", x);  // 5, not 10!
10   return EXIT_SUCCESS;
11 }
```



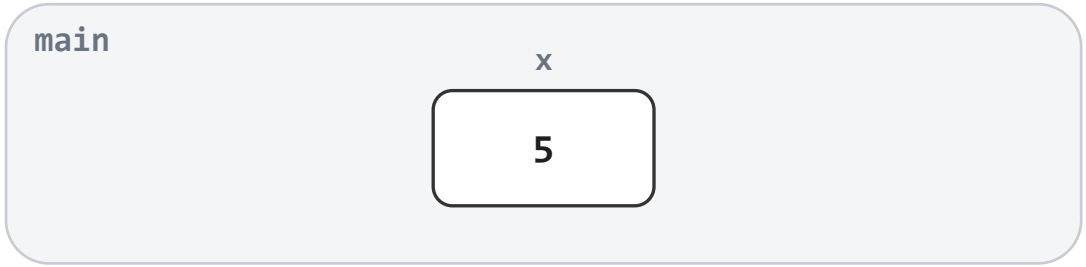
# Pass-by-Value

```
1  // Tries to double n.
2  void Double(int n) {
3    n = n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    Double(x);
9    printf("x = %d\n", x);  // 5, not 10!
10   return EXIT_SUCCESS;
11 }
```



# Pass-by-Value

```
1  // Tries to double n.
2  void Double(int n) {
3    n = n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    Double(x);
9    printf("x = %d\n", x);  // 5, not 10!
10   return EXIT_SUCCESS;
11 }
```



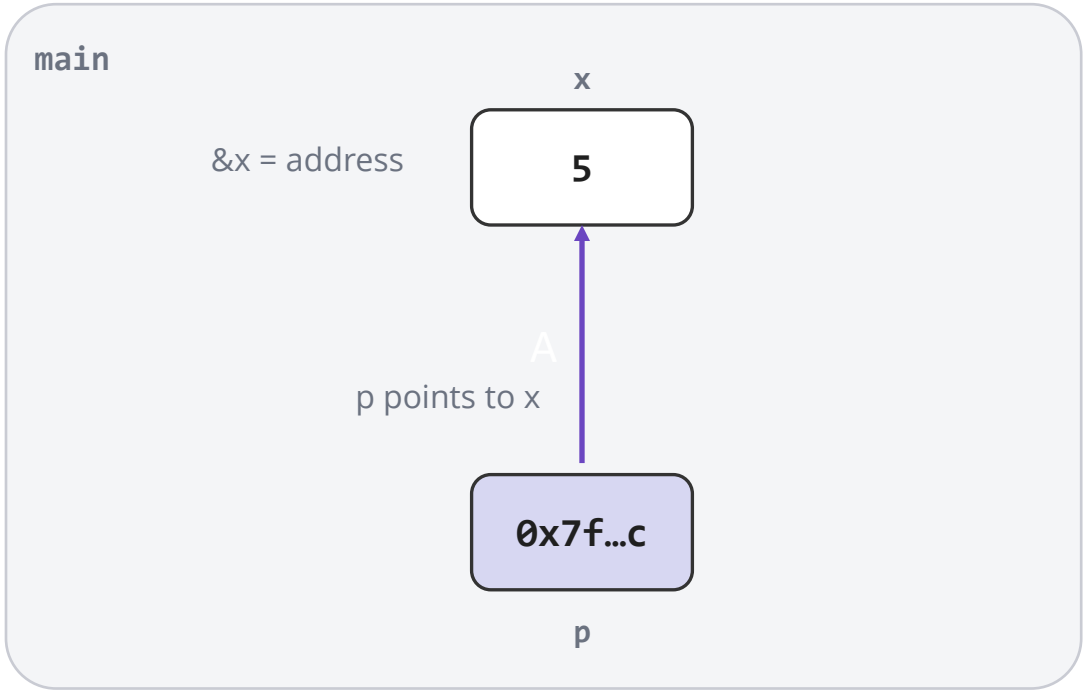

**Takeaway:** C passes arguments by value (i.e. the function gets its own copy). Assigning to a parameter never changes the caller's variable.

# Pointers

```
1  int main(int argc, char* argv[]) {
2    int x = 5;
3    int* p = &x;    // p holds x's address
4    *p = 10;        // write through p
5    printf("x = %d\n", x);  // 10
6    return EXIT_SUCCESS;
7  }
```

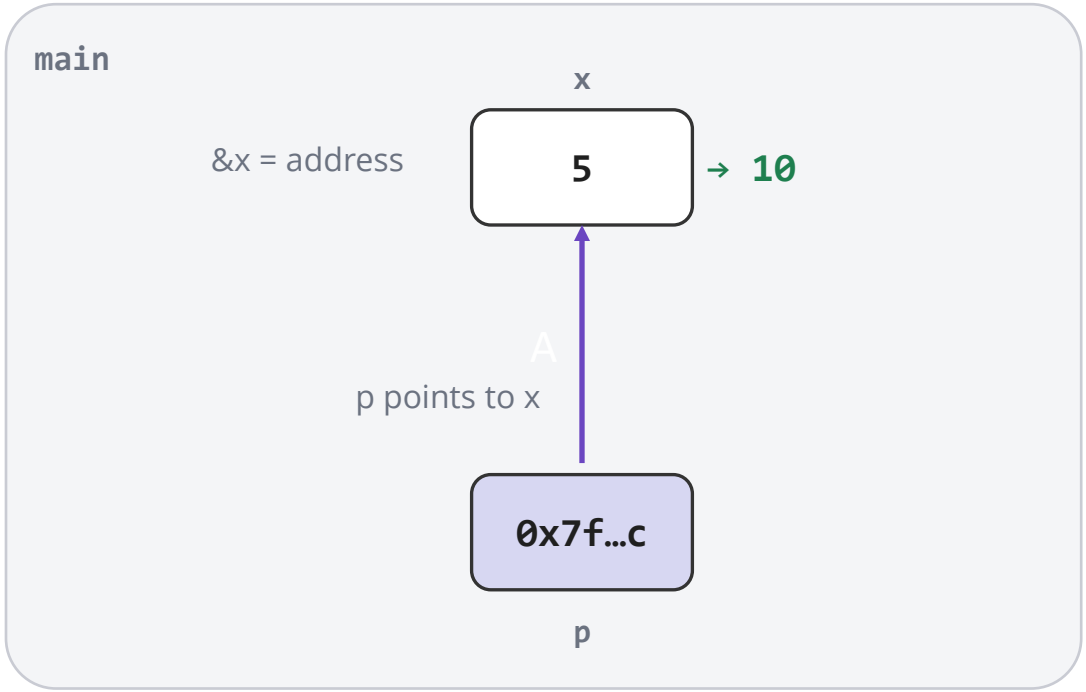
# Pointers

```
1  int main(int argc, char* argv[]) {
2    int x = 5;
3    int* p = &x;    // p holds x's address
4    *p = 10;        // write through p
5    printf("x = %d\n", x);  // 10
6    return EXIT_SUCCESS;
7  }
```



# Pointers

```
1  int main(int argc, char* argv[]) {
2    int x = 5;
3    int* p = &x;    // p holds x's address
4    *p = 10;        // write through p
5    printf("x = %d\n", x);  // 10
6    return EXIT_SUCCESS;
7  }
```



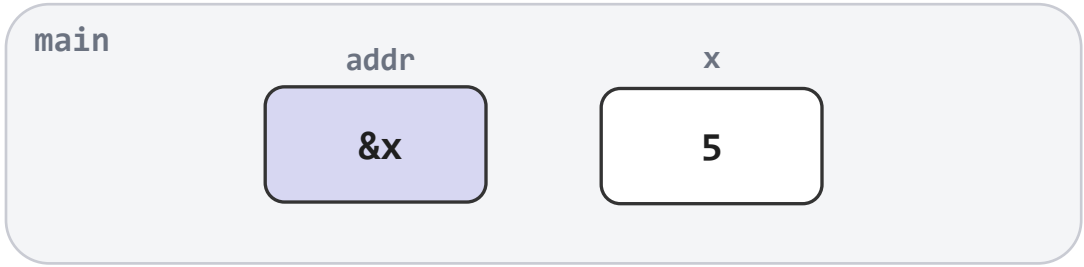

**Takeaway:** &x is the address of x. *p is the value p points to (dereference). A pointer just holds an address.

# Pass-by-Reference (sort of)

```
1  // Doubles the value n points to.
2  void Double(int* n) {
3    *n = *n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    int* addr = &x;
9    Double(addr);
10   printf("x = %d\n", x);  // 10
11   return EXIT_SUCCESS;
   }
```

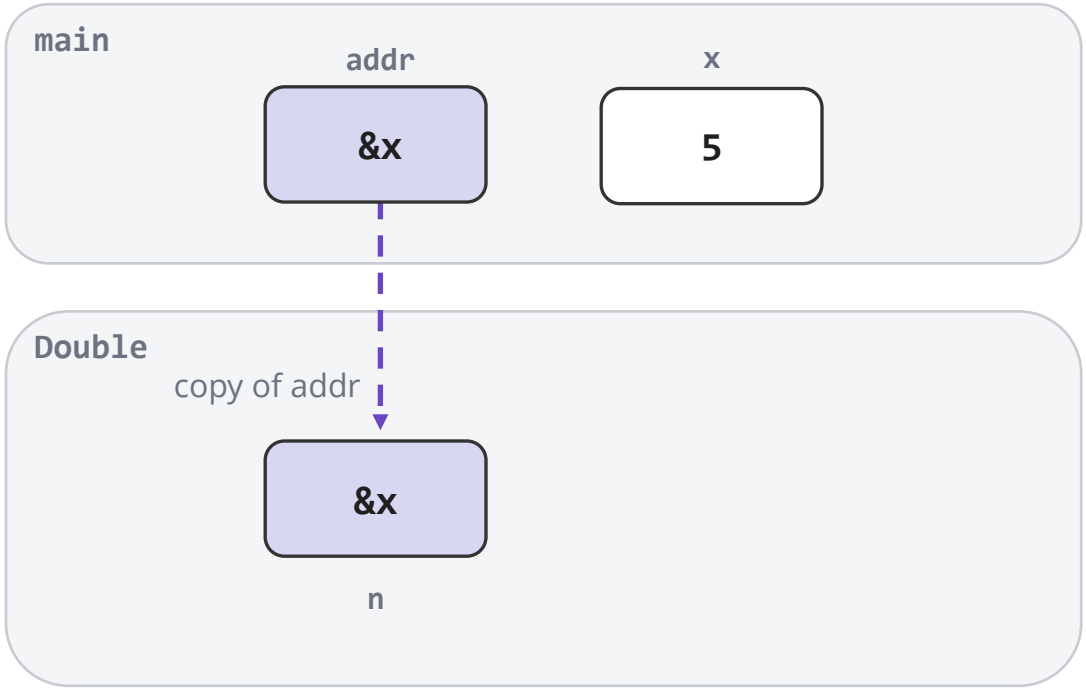
# Pass-by-Reference (sort of)

```
1  // Doubles the value n points to.
2  void Double(int* n) {
3    *n = *n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    int* addr = &x;
9    Double(addr);
10   printf("x = %d\n", x);  // 10
11   return EXIT_SUCCESS;
   }
```



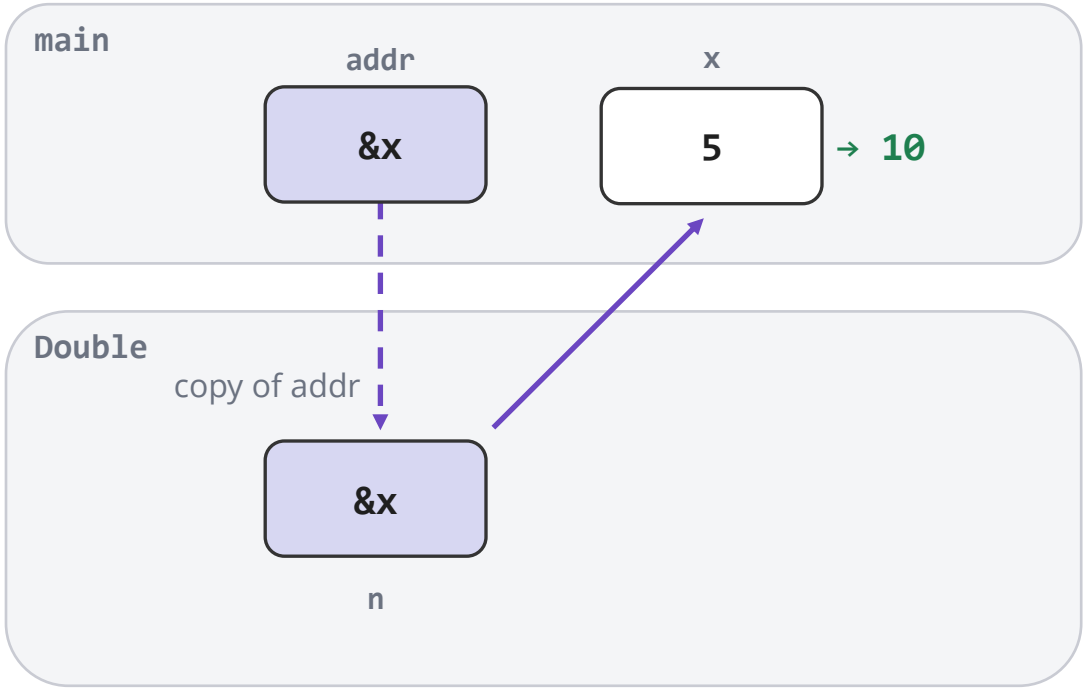
# Pass-by-Reference (sort of)

```
1  // Doubles the value n points to.
2  void Double(int* n) {
3    *n = *n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    int* addr = &x;
9    Double(addr);
10   printf("x = %d\n", x);  // 10
11   return EXIT_SUCCESS;
   }
```



# Pass-by-Reference (sort of)

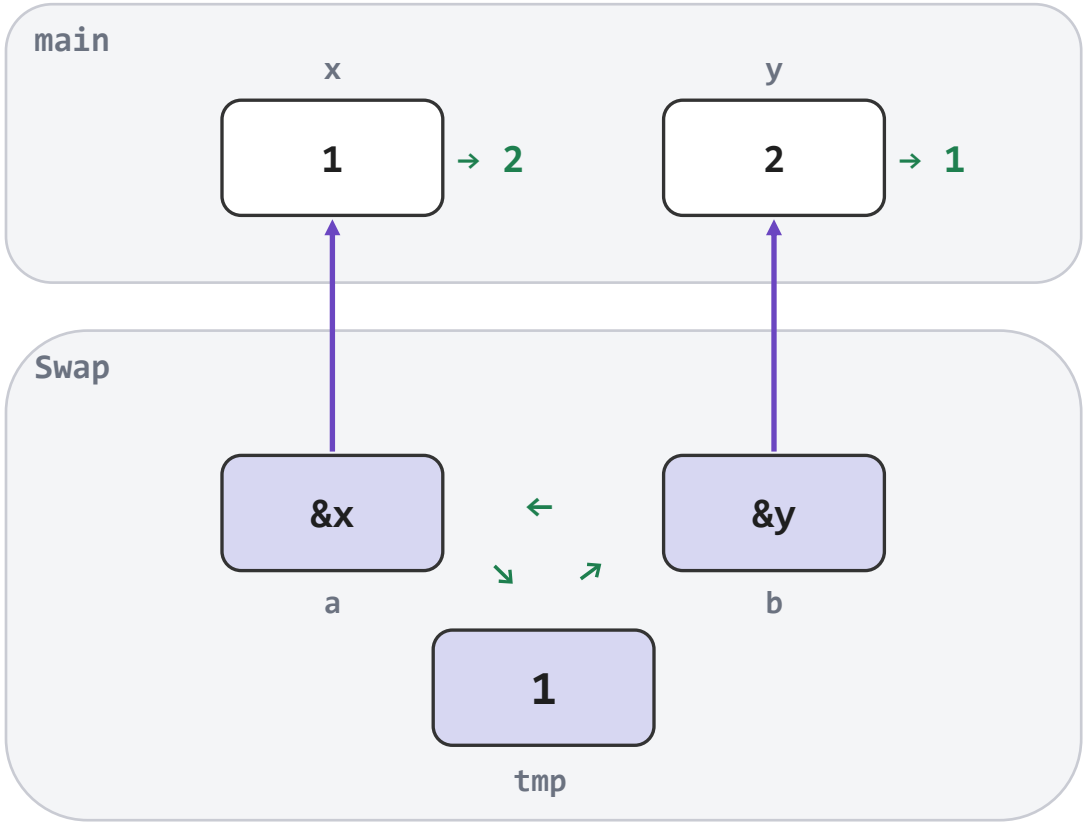
```
1  // Doubles the value n points to.
2  void Double(int* n) {
3    *n = *n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    int* addr = &x;
9    Double(addr);
10   printf("x = %d\n", x);  // 10
11   return EXIT_SUCCESS;
   }
```




**Takeaway:** Pass &x so the function can reach back and modify the caller's variable. Notably, this is still pass-by-value the value copied is now an address.

# Swap

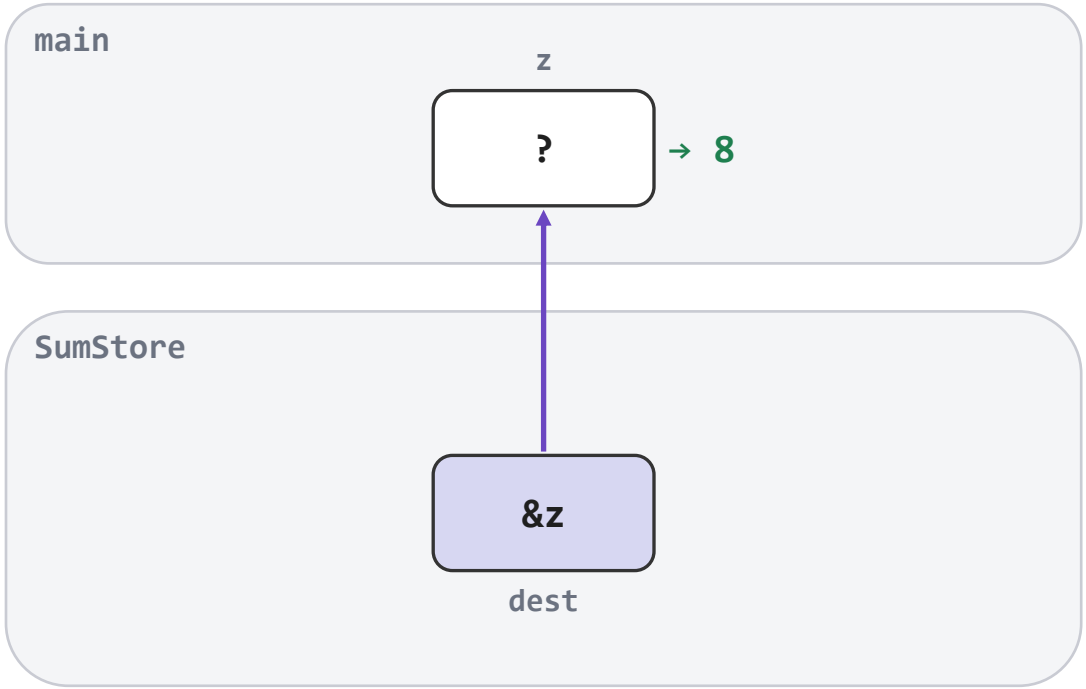
```
1  void Swap(int* a, int* b) {
2    int tmp = *a;
3    *a = *b;
4    *b = tmp;
5  }
6
7  int main(int argc, char* argv[]) {
8    int x = 1, y = 2;
9    Swap(&x, &y);
10   printf("%d %d\n", x, y);  // 2 1
11   return EXIT_SUCCESS;
12 }
```




**Takeaway:** Swap must take pointers. By value it would only swap copies, and the caller's x and y would be unchanged.

# Output Parameters

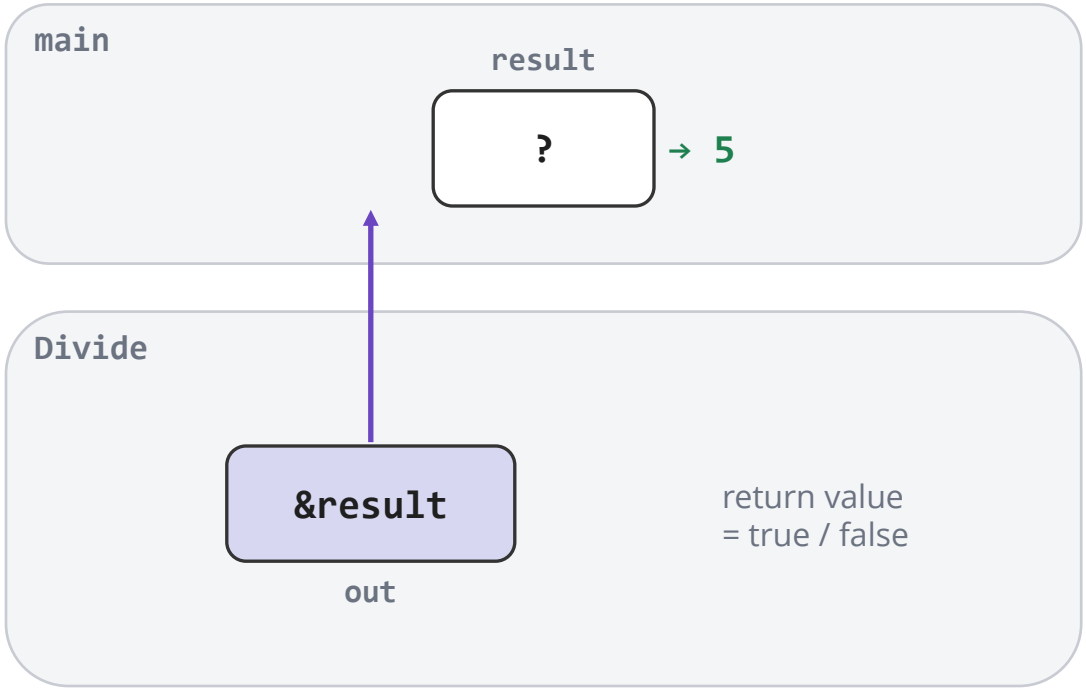
```
1  // Returns x + y through *dest.
2  void SumStore(int x, int y, int* dest) {
3    *dest = x + y;
4  }
5
6  int main(int argc, char* argv[]) {
7    int z;
8    SumStore(3, 5, &z);
9    printf("z = %d\n", z);  // 8
10   return EXIT_SUCCESS;
11 }
```




**Takeaway:** An output parameter is a pointer the function writes through to "hand back" a result. Allows us to return more than one value.

# Output Parameters & Error Codes

```
1  // false if b == 0; else writes a/b to *out.
2  bool Divide(int a, int b, int* out) {
3    if (b == 0) {
4      return false;
5    }
6    *out = a / b;
7    return true;
8  }
9
10 int main(int argc, char* argv[]) {
11   int result;
12   if (Divide(10, 2, &result)) {
13     printf("%d\n", result);  // 5
14   }
15 }
```




**Takeaway:** A function returns one value. Use the return value for status / an error code and an output parameter for the result, just like strtol and sscanf.

# Memory Model

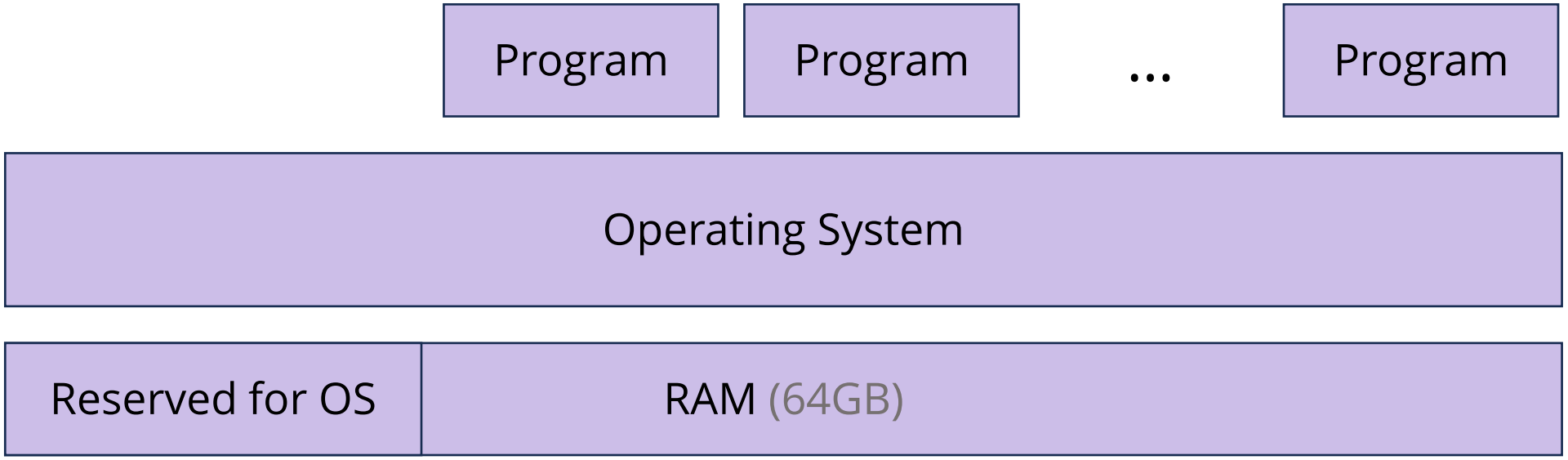
# Where do my variables actually live?

```c
#include <stdio.h>
#include <stdlib.h>

int g_total = 0;            // a global

int main(int argc, char* argv[]) {
  int local = 5;            // a local
  int* buf = malloc(40);    // 10 ints
  // ... use buf ...
  free(buf);
  return EXIT_SUCCESS;
}
```

- **g_total** is global one copy for the whole program.
- **local** is created fresh on every call to main.
- **buf** holds an address; the 40 bytes it points to come from malloc.

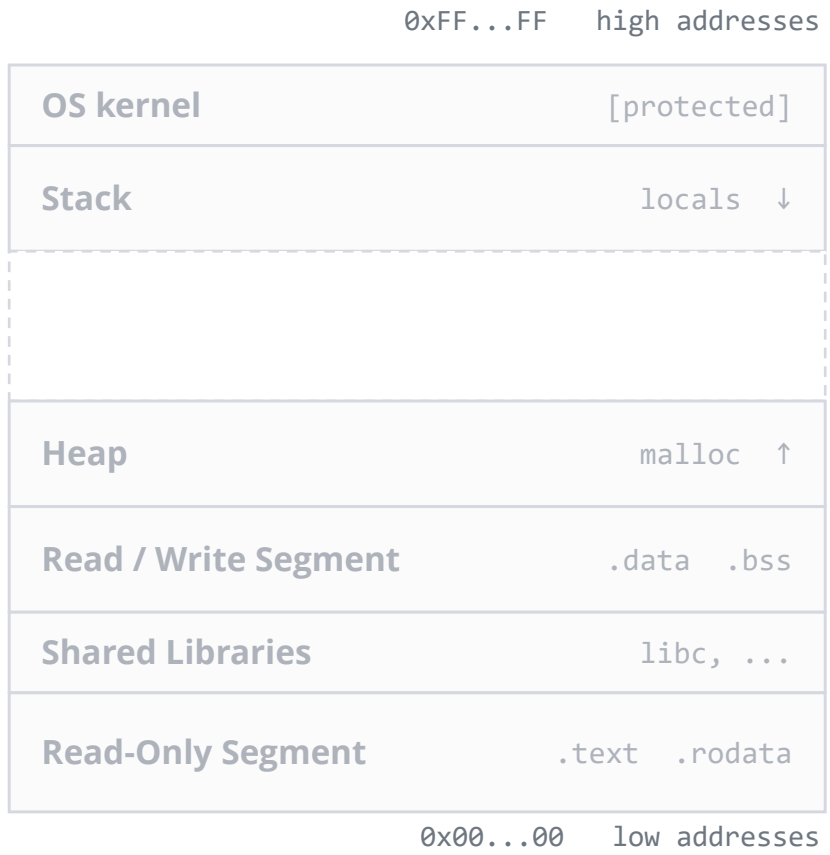
# OS manages many programs

The OS decides which programs get to run. It gives every process the illusion of its own private memory (i.e. an address space).



# Each process gets its own address space

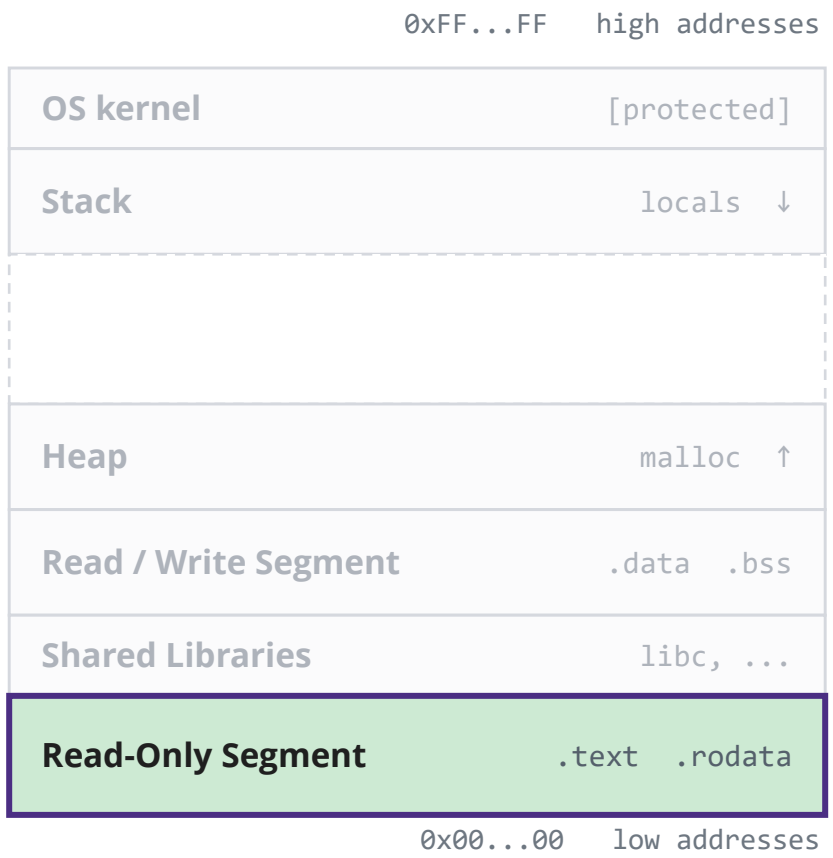
- Addresses run from 0x00…00 (low) up to 0xFF…FF (high).
- The OS carves this space into regions, each with a different job.
  - We'll fill them in one at a time.




**Takeaway:** Think of memory as one tall column of addresses that your program "owns".

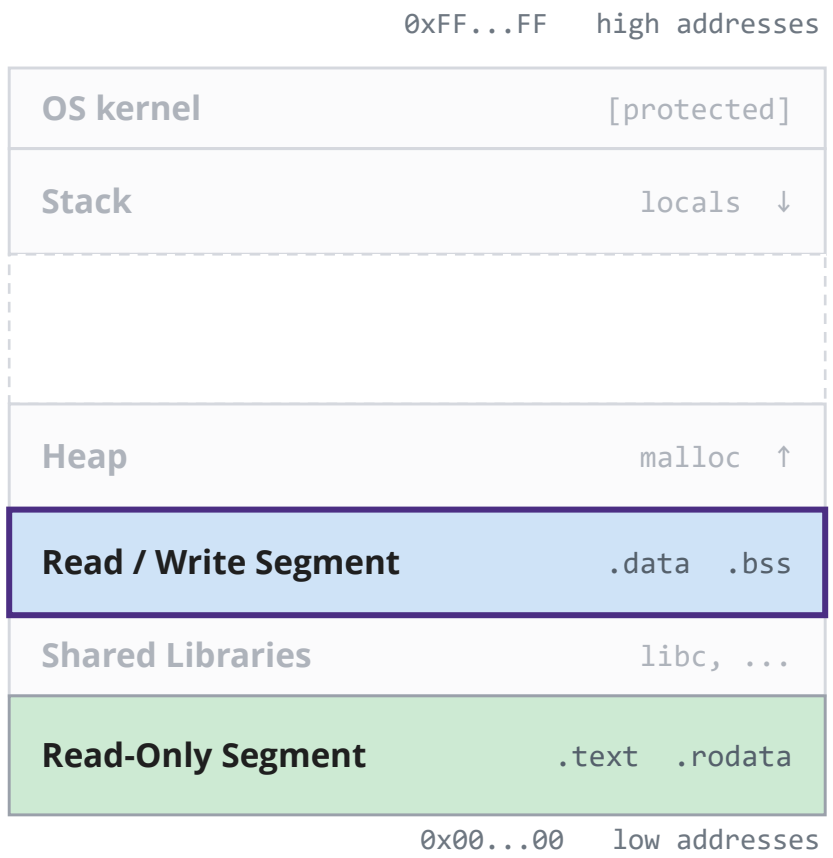
# Read-Only Segment: .text + .rodata

- **.text**: your compiled machine instructions (the code itself).
- **.rodata**: read-only data, e.g. string and other literals.
- Filled in directly from the executable file when the program loads.
- Read-only: the OS faults if you try to write here.
  - char* s = "hi"; s[0] = 'H'; → **crash (SIGSEGV)**
- Lifetime: the entire run of the program; never grows or shrinks.



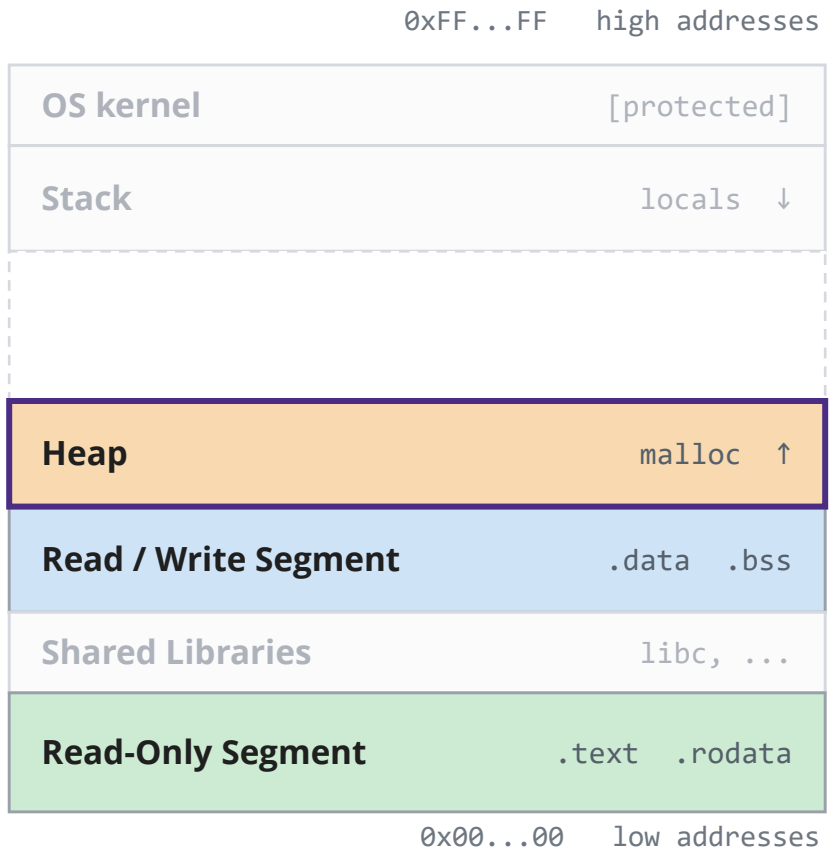
# Read/Write Segment: .data + .bss

- Home of global variables and static local variables.
- **.data**: globals/statics with a non-zero initial value.
  - int g_limit = 10;
- **.bss**: globals/statics that start at zero (stored as just a size).
  - int g_count;  // auto zero-initialized
- Statically allocated: created when the process starts, freed when it exits.
  - Exactly one copy, alive for the whole program.



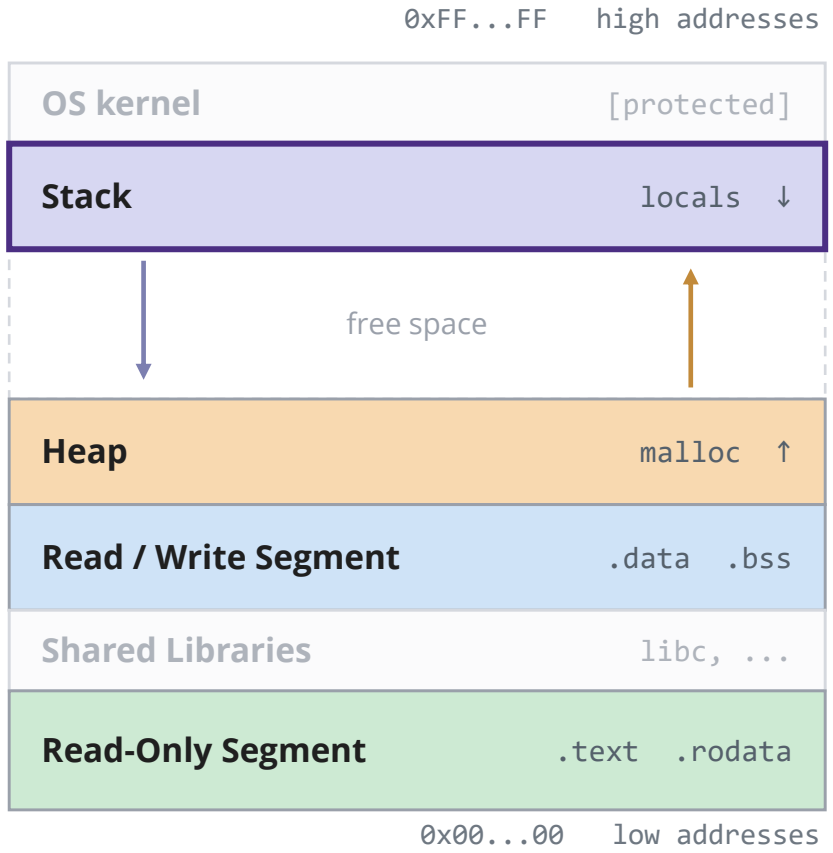
# The Heap: malloc / free

- Dynamically allocated memory you request explicitly at runtime.
  - int* a = malloc(10 * sizeof(int));
- You decide both when it is born and when it dies.
  - free(a);  to give it back when done.
- Grows UP toward higher addresses as you allocate more.
- **Forget to free** → memory leak.
- **Use after free / double free** → undefined behavior.



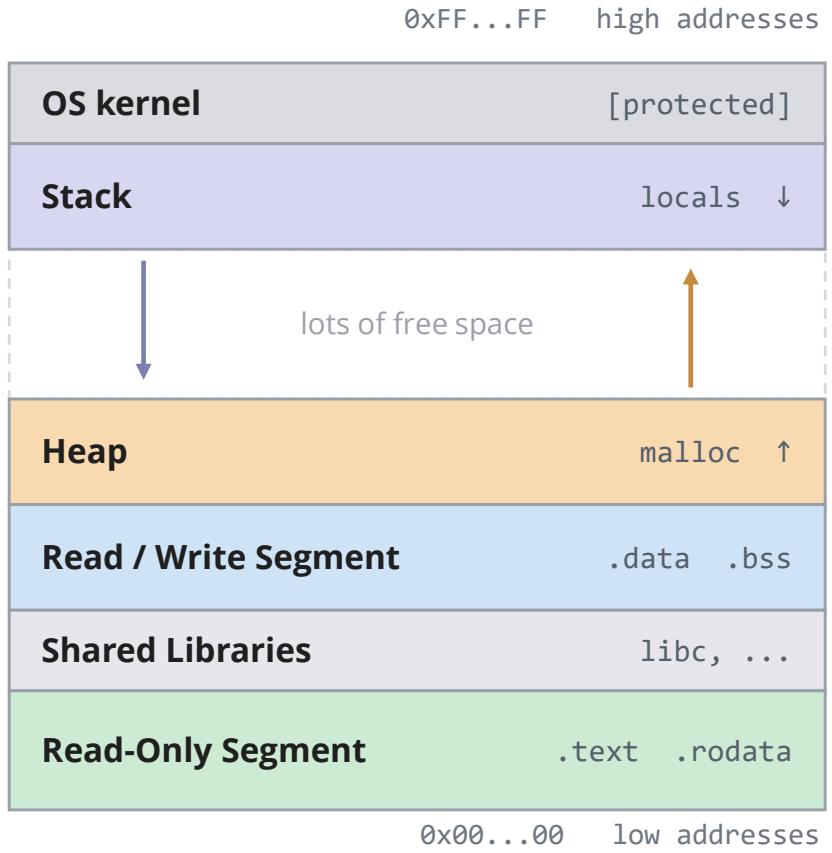
# The Stack: locals + function calls

- Holds a frame for each active function call.
  - Parameters, local variables, return address, saved registers.
- Push a frame on call; pop it on return. This is fully automatic.
  - The compiler emits the push/pop code for you.
- Grows DOWN toward lower addresses, meeting the heap in the middle.
- Locals vanish when the function returns — **never return a pointer to one!**



# The whole address space at once

```c
int g_limit = 10;       // .data  (init global)
int g_count;            // .bss   (zero global)
const char* k = "hi";   // ptr .data, "hi" .rodata

int main(int argc, char* argv[]) {
  int local = 5;        // stack
  int* buf =            // buf -> stack
      malloc(40);       // *buf -> heap
  free(buf);
  return EXIT_SUCCESS;
}
```



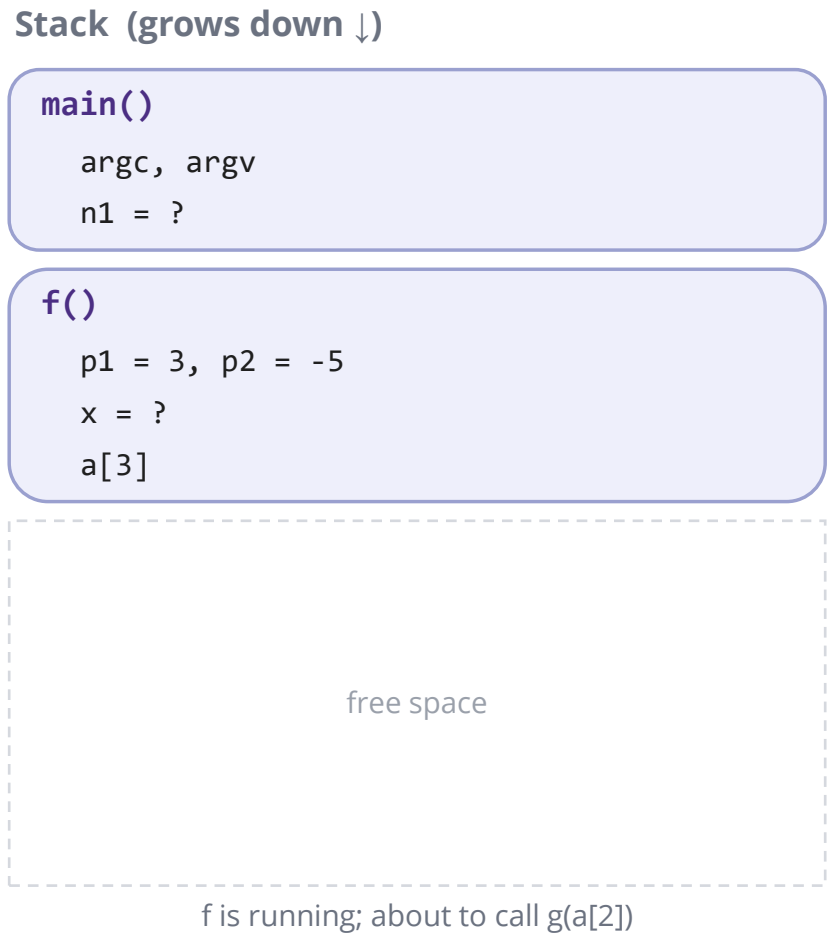
# Stack in action (1/4)

```
1  int f(int p1, int p2);
2  int g(int param);
3
4  int main(int argc, char* argv[]) {
5    int n1 = f(3, -5);
6    n1 = g(n1);
7    return EXIT_SUCCESS;
8  }
9
10 int f(int p1, int p2) {
11   int x;
12   int a[3];
13   x = g(a[2]);
14   return x;
15 }
16
17 int g(int param) {
18   return param * 2;
19 }
```


main is running; about to call f(3, -5)

# Stack in action (2/4)

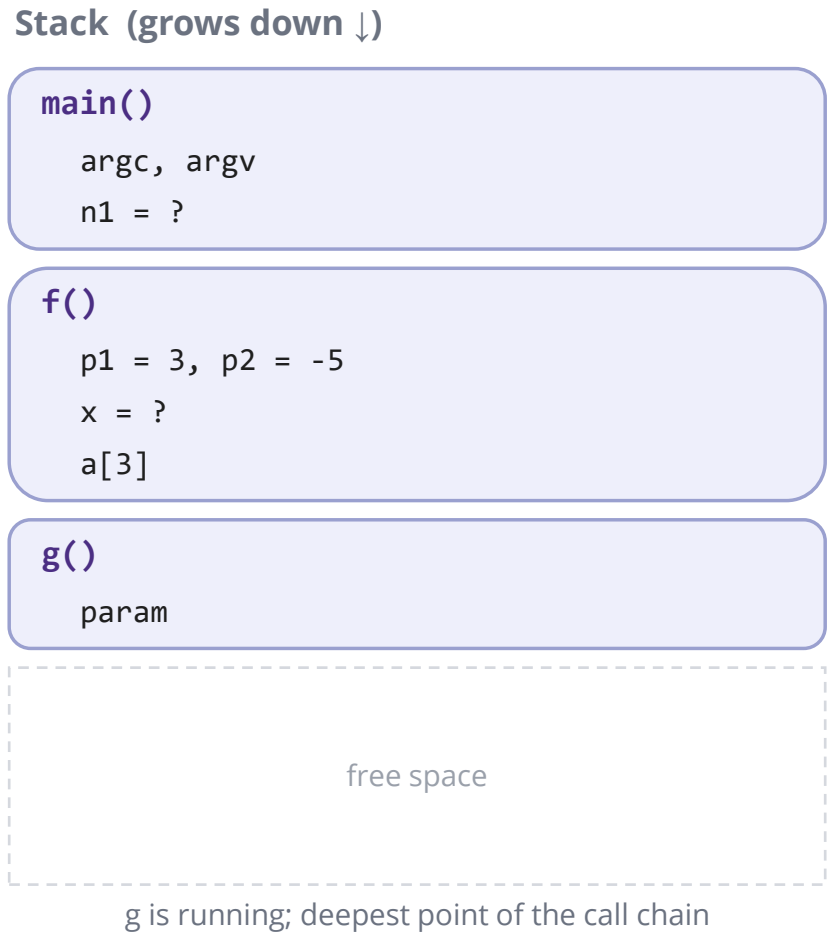
```
1  int f(int p1, int p2);
2  int g(int param);
3
4  int main(int argc, char* argv[]) {
5    int n1 = f(3, -5);
6    n1 = g(n1);
7    return EXIT_SUCCESS;
8  }
9
10 int f(int p1, int p2) {
11   int x;
12   int a[3];
13   x = g(a[2]);
14   return x;
15 }
16
17 int g(int param) {
18   return param * 2;
19 }
```




f is running; about to call g(a[2])

# Stack in action (3/4)

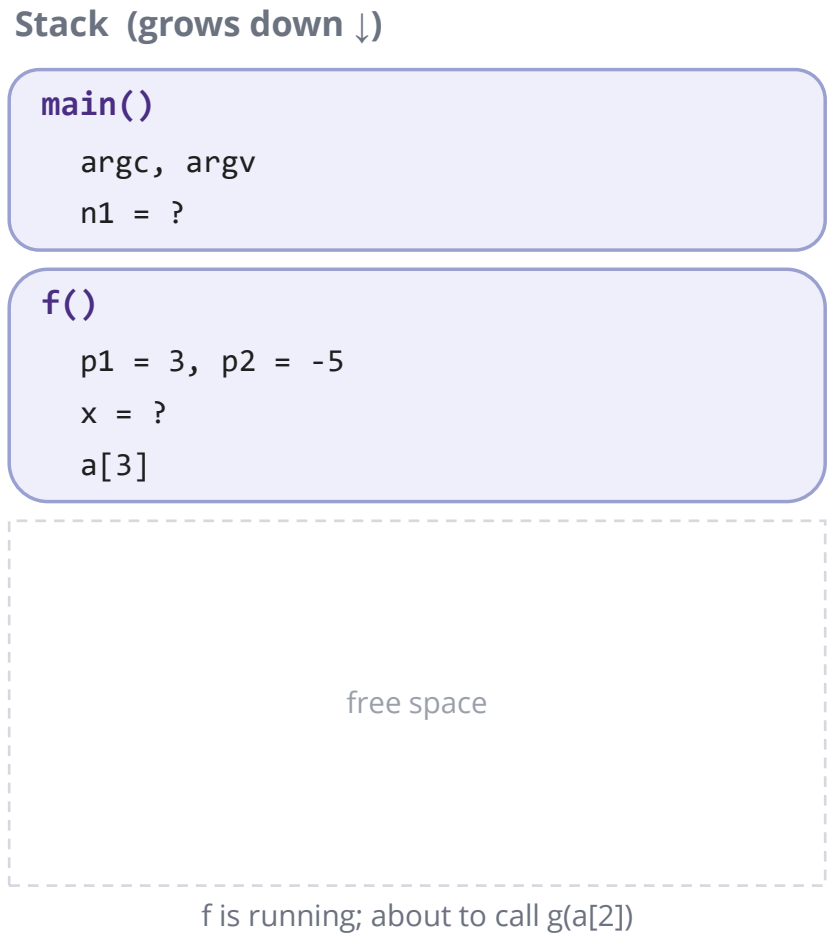
```
1  int f(int p1, int p2);
2  int g(int param);
3
4  int main(int argc, char* argv[]) {
5    int n1 = f(3, -5);
6    n1 = g(n1);
7    return EXIT_SUCCESS;
8  }
9
10 int f(int p1, int p2) {
11   int x;
12   int a[3];
13   x = g(a[2]);
14   return x;
15 }
16
17 int g(int param) {
18   return param * 2;
19 }
```




g is running; deepest point of the call chain

# Stack in action (4/4)

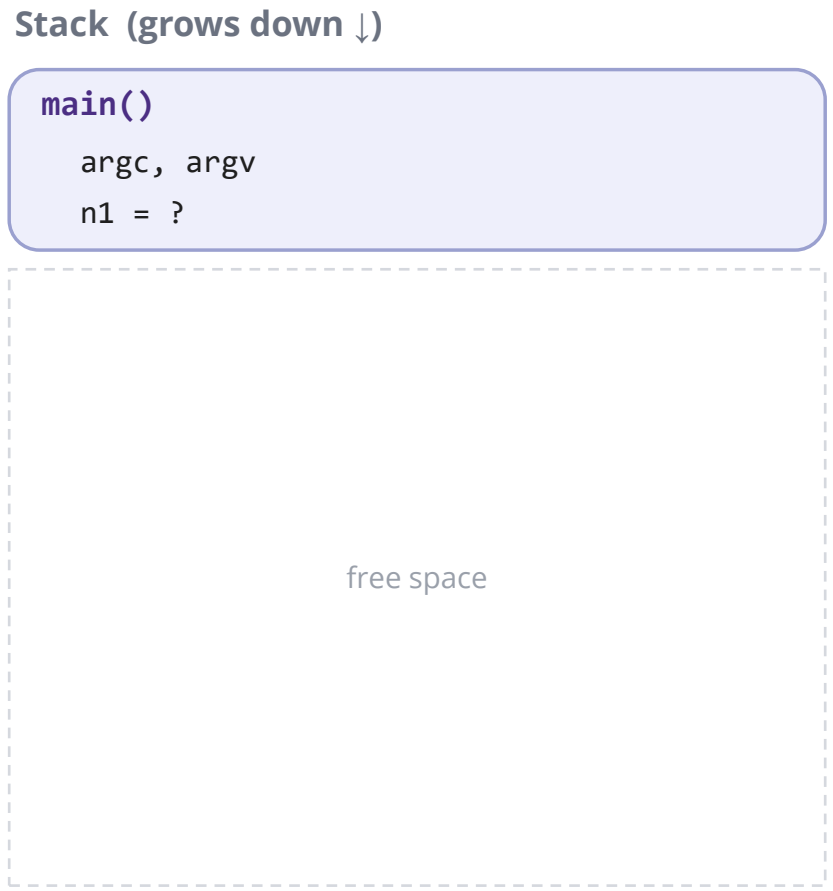
```
1  int f(int p1, int p2);
2  int g(int param);
3
4  int main(int argc, char* argv[]) {
5    int n1 = f(3, -5);
6    n1 = g(n1);
7    return EXIT_SUCCESS;
8  }
9
10 int f(int p1, int p2) {
11   int x;
12   int a[3];
13   x = g(a[2]);
14   return x;
15 }
16
17 int g(int param) {
18   return param * 2;
19 }
```




f is running; about to call g(a[2])

# Stack in action (4/4)

```
1  int f(int p1, int p2);
2  int g(int param);
3
4  int main(int argc, char* argv[]) {
5    int n1 = f(3, -5);
6    n1 = g(n1);
7    return EXIT_SUCCESS;
8  }
9
10 int f(int p1, int p2) {
11   int x;
12   int a[3];
13   x = g(a[2]);
14   return x;
15 }
16
17 int g(int param) {
18   return param * 2;
19 }
```




g and f have returned; only main remains

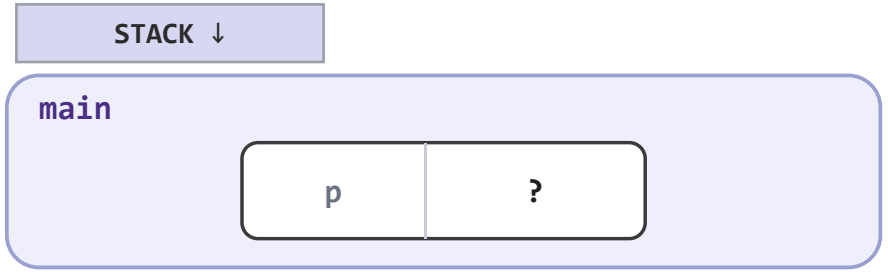
# Summary: four regions, four lifetimes

| Region | What lives there | Lifetime | Who manages it | Watch out for |
|---|---|---|---|---|
| `Stack` | locals, parameters, call bookkeeping | per function call | automatic (compiler) | dangling pointer to a local |
| `Heap` | malloc'd blocks | malloc → free | YOU, by hand | leaks, use-after-free |
| `.data / .bss` | globals & statics | whole program | automatic (loader) | shared mutable state |
| `.text / .rodata` | code + literals | whole program | read-only (loader) | writing a literal => segfault |

# Extras

# Stack: dangling pointer to a local

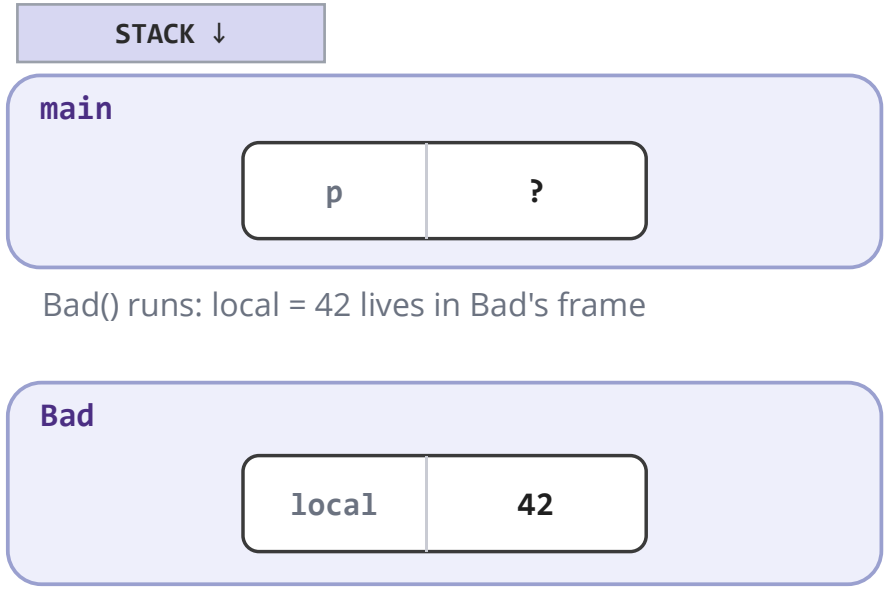
```
1  int* Bad(void) {
2    int local = 42;
3    return &local;   // address of a local!
4  }
5
6  int main(int argc, char* argv[]) {
7    int* p = Bad();
8    printf("%d\n", *p);  // reads dead memory
9    return EXIT_SUCCESS;
10 }
```




main runs; about to call Bad()

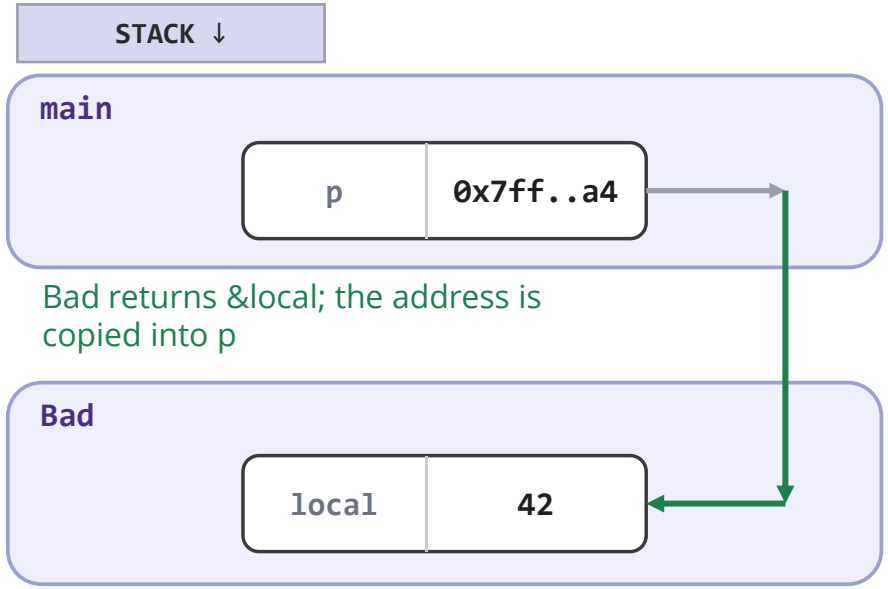
# Stack: dangling pointer to a local

```
1  int* Bad(void) {
2    int local = 42;
3    return &local;   // address of a local!
4  }
5
6  int main(int argc, char* argv[]) {
7    int* p = Bad();
8    printf("%d\n", *p);  // reads dead memory
9    return EXIT_SUCCESS;
10 }
```



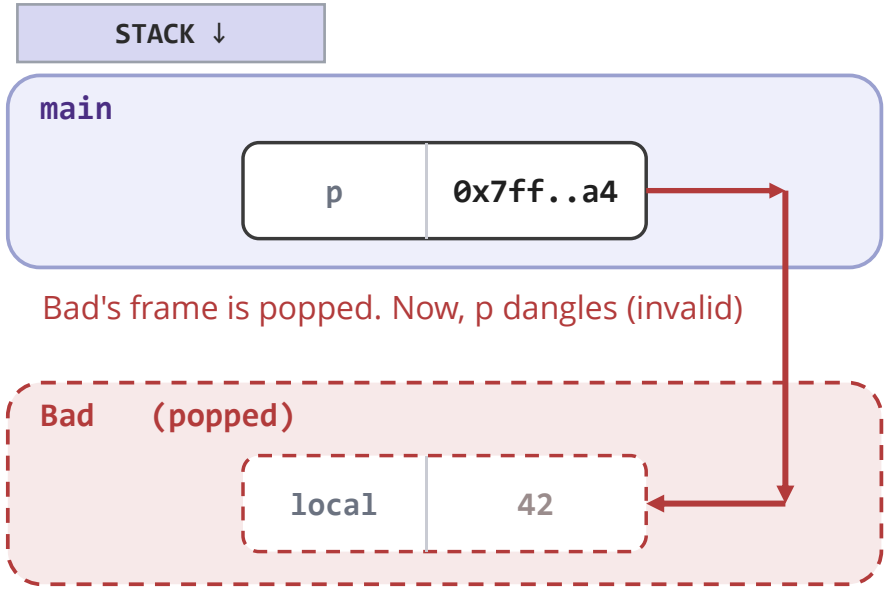
# Stack: dangling pointer to a local

```
1  int* Bad(void) {
2    int local = 42;
3    return &local;   // address of a local!
4  }
5
6  int main(int argc, char* argv[]) {
7    int* p = Bad();
8    printf("%d\n", *p);  // reads dead memory
9    return EXIT_SUCCESS;
10 }
```



# Stack: dangling pointer to a local

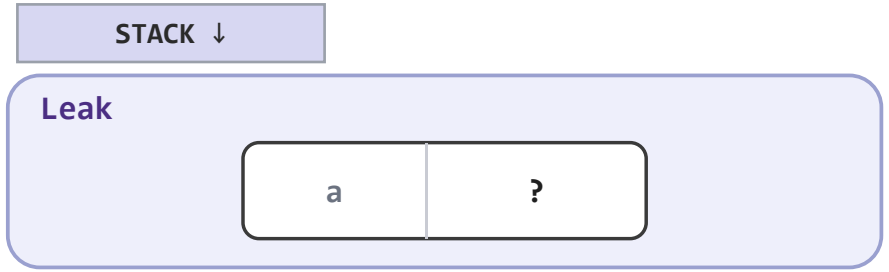
```
1  int* Bad(void) {
2    int local = 42;
3    return &local;   // address of a local!
4  }
5
6  int main(int argc, char* argv[]) {
7    int* p = Bad();
8    printf("%d\n", *p);  // reads dead memory
9    return EXIT_SUCCESS;
10 }
```




**Takeaway:** Never return the address of a local. Its frame is popped on return, so the pointer dangles. Return by value, or malloc the storage so it lives on the heap.

# Heap: memory leak

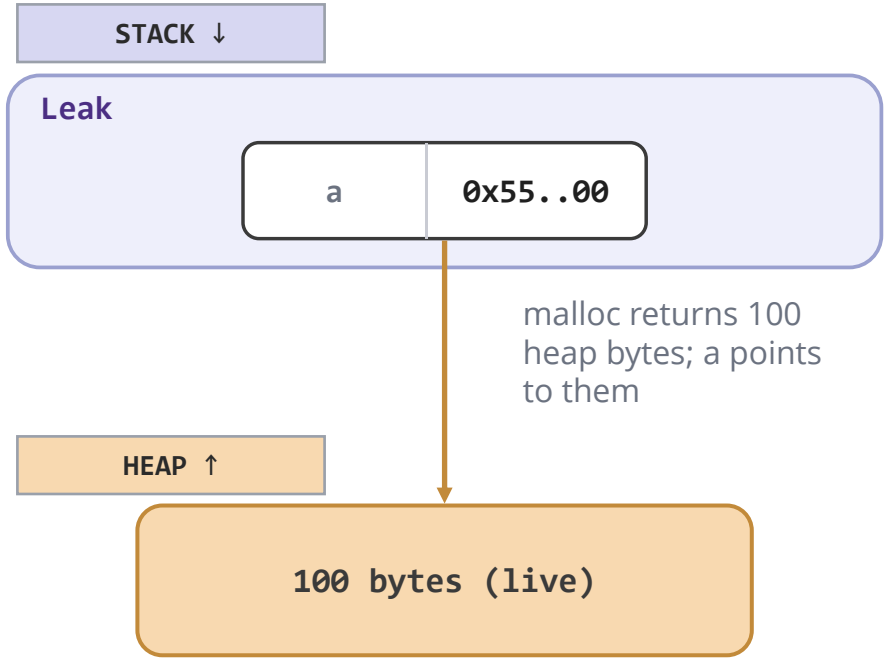
```
1  void Leak(void) {
2    int* a = malloc(100);  // 100 bytes
3    // ... use a ...
4  }  // a disappears; bytes never freed
5
6  int main(int argc, char* argv[]) {
7    Leak();
8    // 100 bytes now unreachable
9    return EXIT_SUCCESS;
10 }
```




main calls Leak(); a is not set yet

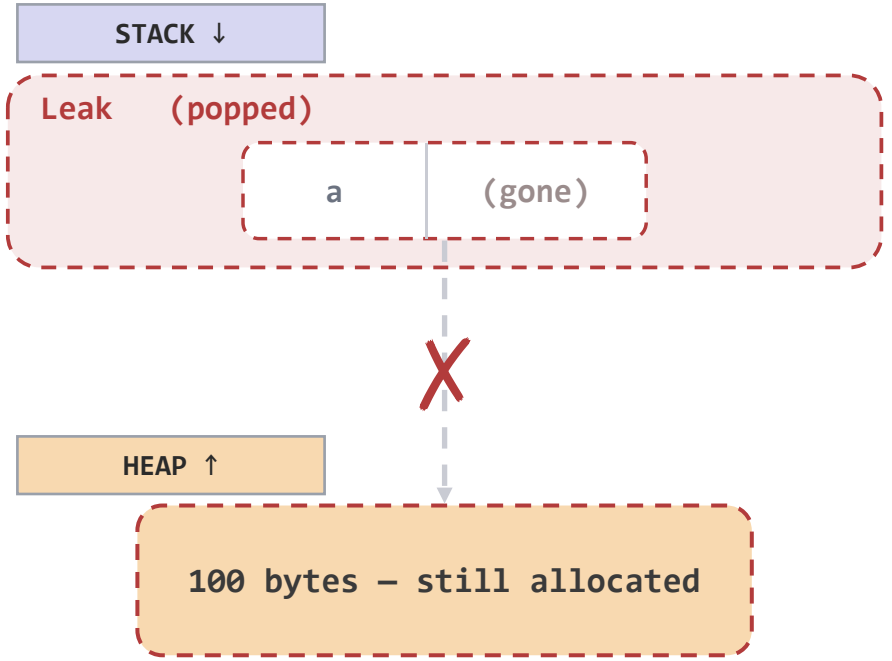
# Heap: memory leak

```
1  void Leak(void) {
2    int* a = malloc(100);  // 100 bytes
3    // ... use a ...
4  }  // a disappears; bytes never freed
5
6  int main(int argc, char* argv[]) {
7    Leak();
8    // 100 bytes now unreachable
9    return EXIT_SUCCESS;
10 }
```



# Heap: memory leak

```c
void Leak(void) {
  int* a = malloc(100);  // 100 bytes
  // ... use a ...
}  // a disappears; bytes never freed

int main(int argc, char* argv[]) {
  Leak();
  // 100 bytes now unreachable
  return EXIT_SUCCESS;
}
```

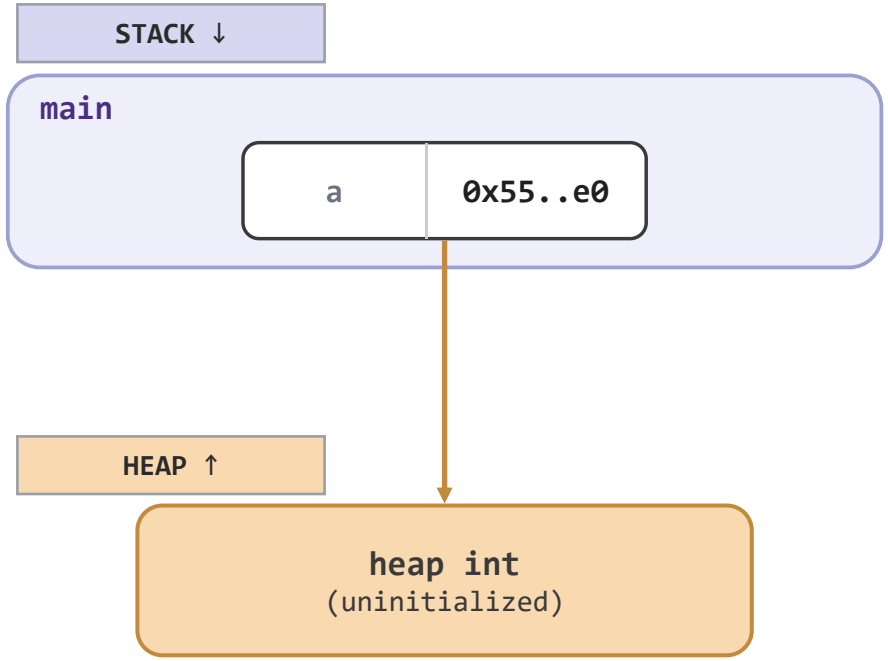



Leak returned: a is gone, block never freed → leak

**Takeaway:** A leak is heap memory you can no longer reach. When the last pointer to a block is lost without free(), those bytes are stuck until the program exits. valgrind finds these.

# Heap: use-after-free

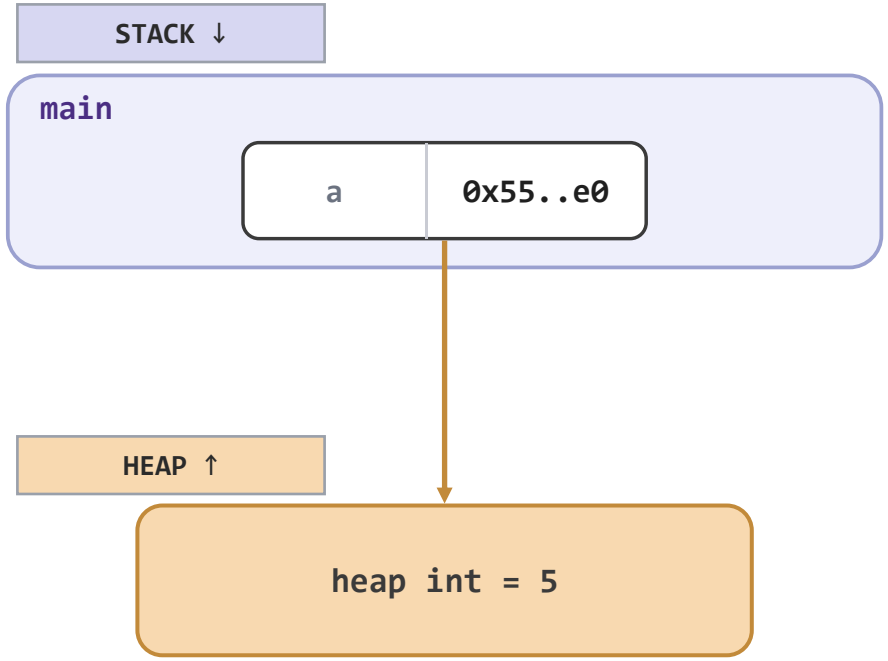
```
1  int main(int argc, char* argv[]) {
2    int* a = malloc(sizeof(int));
3    *a = 5;
4    free(a);             // block returned
5    printf("%d\n", *a);  // use after free!
6    // free(a);          // (double free!)
7    return EXIT_SUCCESS;
8  }
```




malloc gives main a heap int; a points to it

# Heap: use-after-free

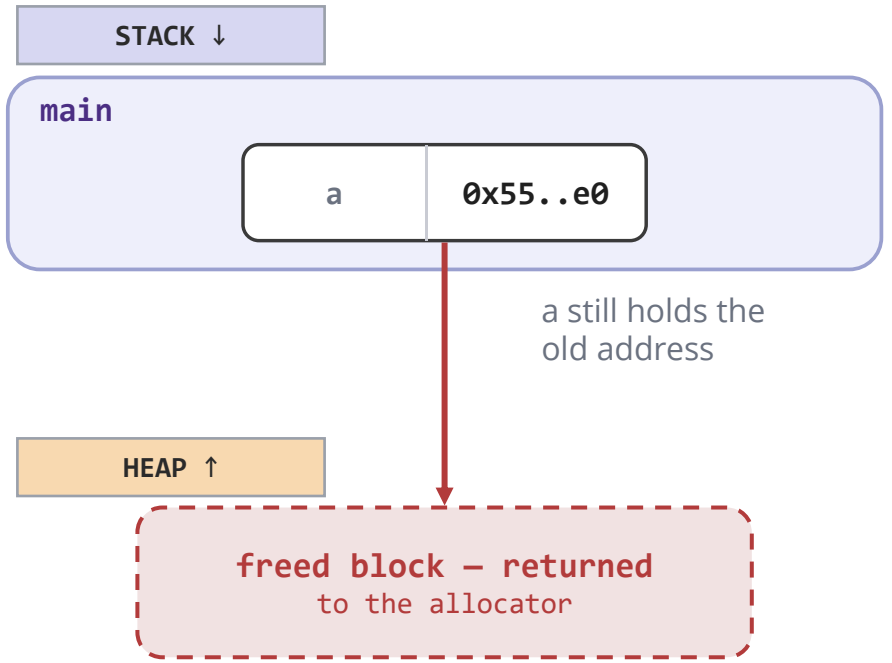
```
1  int main(int argc, char* argv[]) {
2    int* a = malloc(sizeof(int));
3    *a = 5;
4    free(a);             // block returned
5    printf("%d\n", *a);  // use after free!
6    // free(a);          // (double free!)
7    return EXIT_SUCCESS;
8  }
```




*a = 5 stores a value into the heap block

# Heap: use-after-free

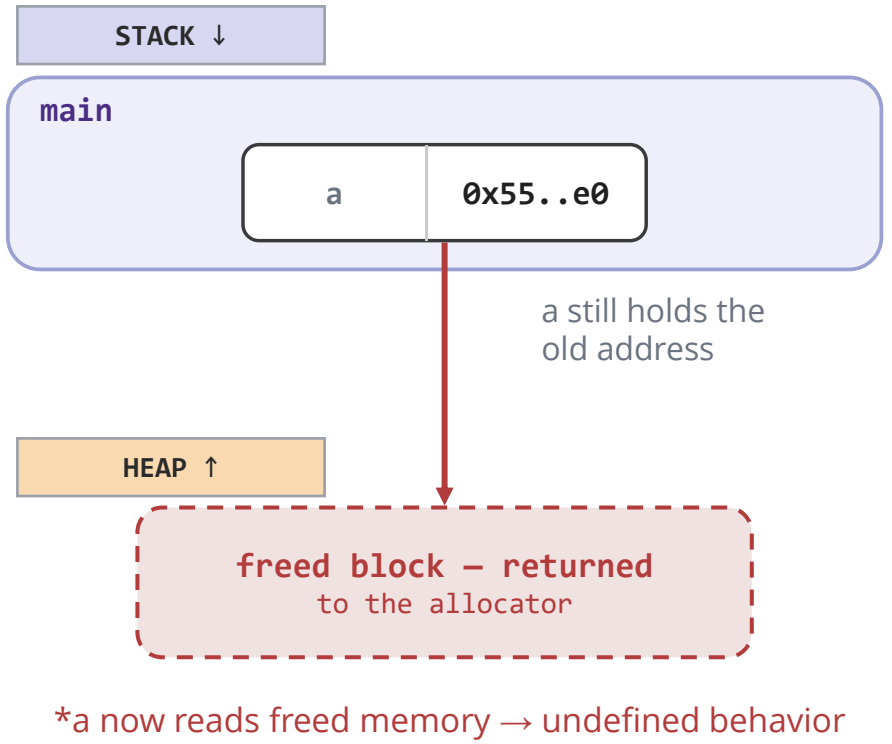
```
1  int main(int argc, char* argv[]) {
2    int* a = malloc(sizeof(int));
3    *a = 5;
4    free(a);             // block returned
5    printf("%d\n", *a);  // use after free!
6    // free(a);          // (double free!)
7    return EXIT_SUCCESS;
8  }
```




free(a): the block is returned, but a is unchanged

# Heap: use-after-free

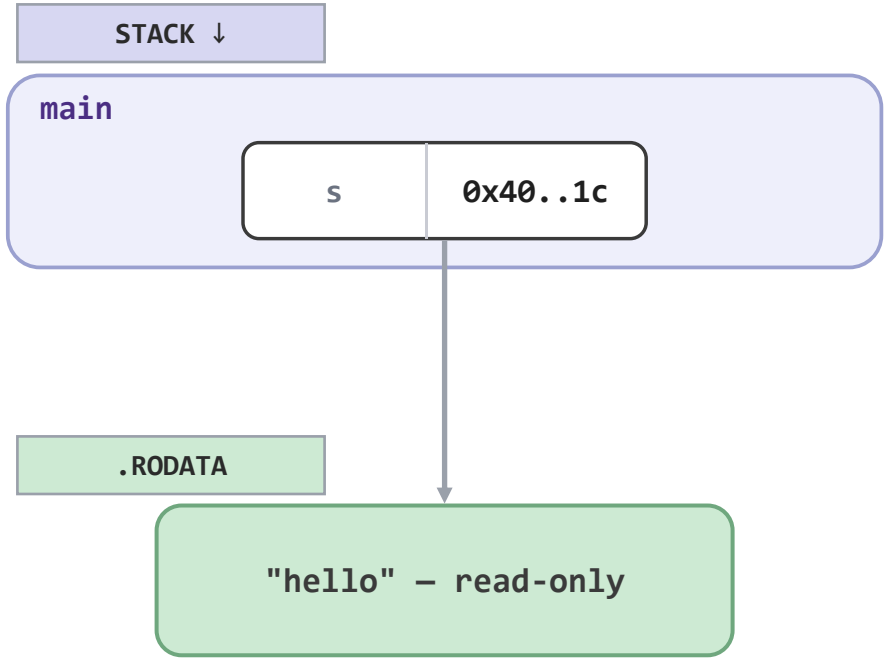
```
1  int main(int argc, char* argv[]) {
2    int* a = malloc(sizeof(int));
3    *a = 5;
4    free(a);             // block returned
5    printf("%d\n", *a);  // use after free!
6    // free(a);          // (double free!)
7    return EXIT_SUCCESS;
8  }
```




**Takeaway:** After free(a), the block is gone but a still points at it. Reading or writing *a (or freeing again) is undefined behavior. Set a = NULL right after freeing.

# Read-only: writing a string literal

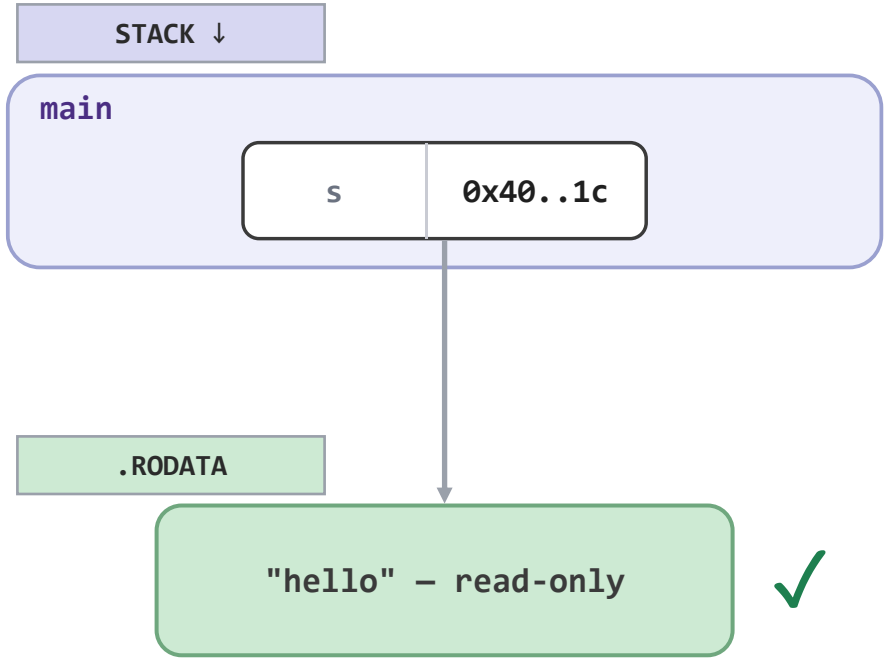
```
1  int main(int argc, char* argv[]) {
2    char* s = "hello";  // points into .rodata
3    printf("%s\n", s);  // reading is fine
4    s[0] = 'H';         // writing is NOT
5    return EXIT_SUCCESS;  // crash: SIGSEGV
6  }
7
8  // Fix: char s[] = "hello";  // writable copy
9  //                            // on the stack
```




s points at the literal in read-only memory

# Read-only: writing a string literal

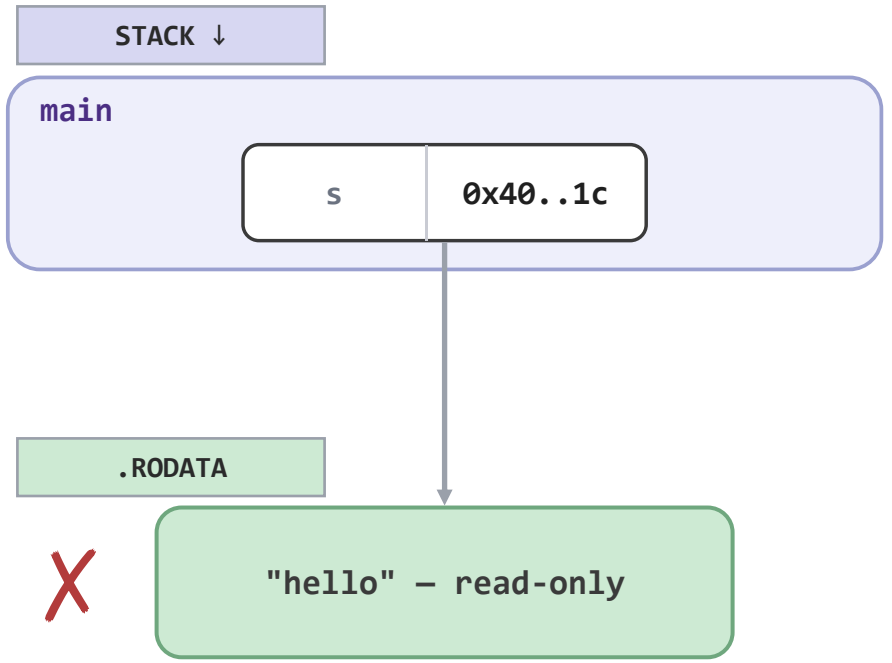
```
1  int main(int argc, char* argv[]) {
2    char* s = "hello";  // points into .rodata
3    printf("%s\n", s);  // reading is fine
4    s[0] = 'H';         // writing is NOT
5    return EXIT_SUCCESS;  // crash: SIGSEGV
6  }
7
8  // Fix: char s[] = "hello";  // writable copy
9  //                            // on the stack
```




reading through s is fine (printf just looks)

# Read-only: writing a string literal

```
1  int main(int argc, char* argv[]) {
2    char* s = "hello";  // points into .rodata
3    printf("%s\n", s);  // reading is fine
4    s[0] = 'H';         // writing is NOT
5    return EXIT_SUCCESS;  // crash: SIGSEGV
6  }
7
8  // Fix: char s[] = "hello";  // writable copy
9  //                            // on the stack
```

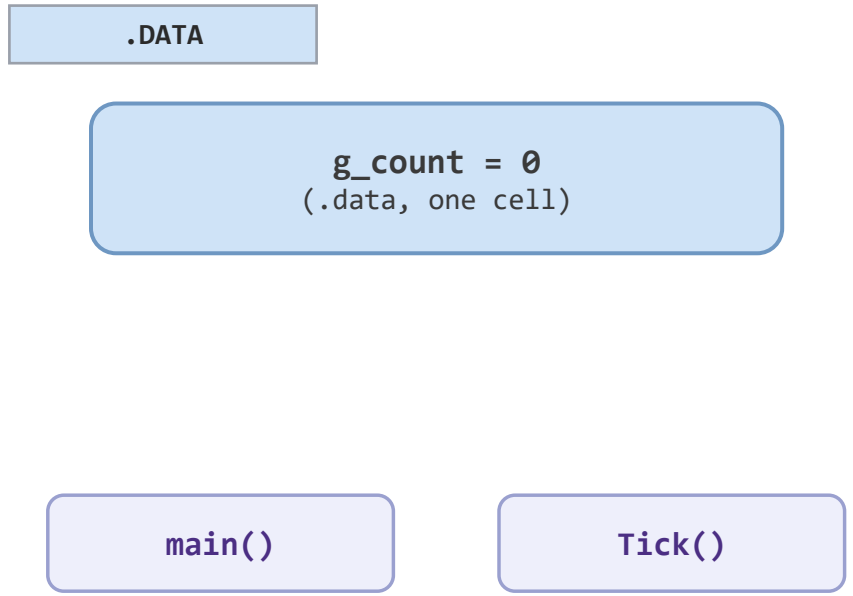



s[0] = 'H' writes read-only memory → SIGSEGV

**Takeaway:** A string literal lives in read-only memory; s points at it, so s[0] = 'H' faults. Use char s[] = "hello" to get a writable copy on the stack.

# Static data: shared mutable state

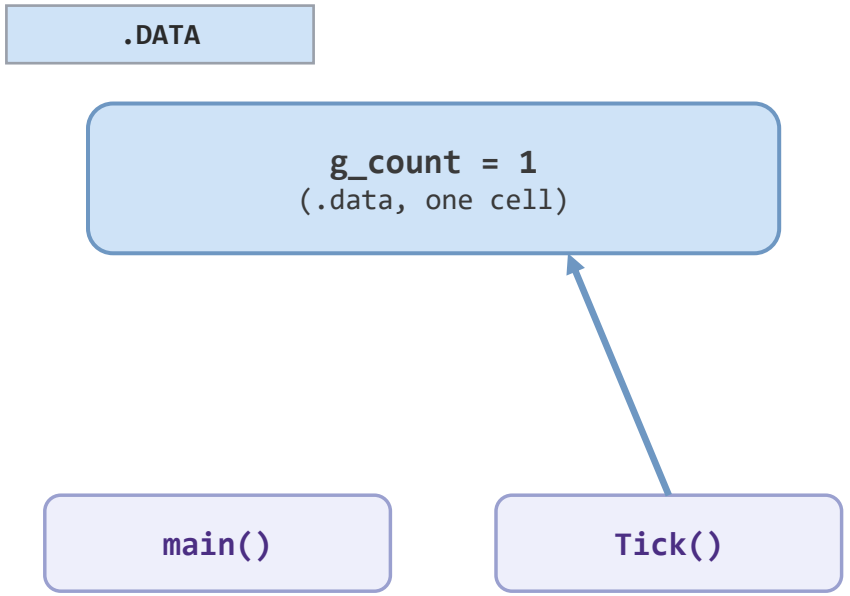
```
1  int g_count = 0;    // one global, in .data
2
3  void Tick(void) { g_count++; }
4
5  int main(int argc, char* argv[]) {
6    Tick();           // g_count -> 1
7    g_count = 100;    // changed from afar
8    Tick();           // g_count -> 101
9    // who set g_count? hard to reason
10   return EXIT_SUCCESS;
11 }
```




one global cell, initialized to 0

# Static data: shared mutable state

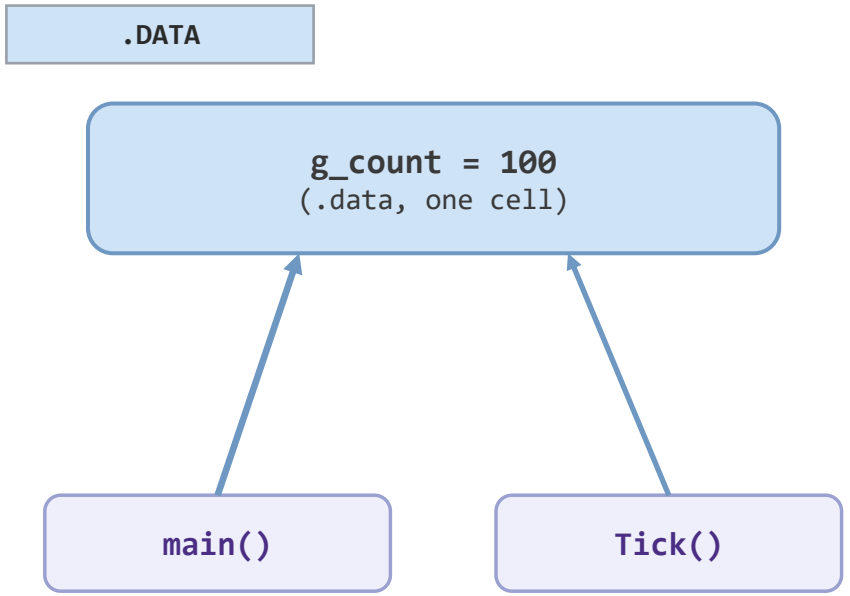
```
1  int g_count = 0;    // one global, in .data
2
3  void Tick(void) { g_count++; }
4
5  int main(int argc, char* argv[]) {
6    Tick();           // g_count -> 1
7    g_count = 100;    // changed from afar
8    Tick();           // g_count -> 101
9    // who set g_count? hard to reason
10   return EXIT_SUCCESS;
11 }
```




Tick() increments g_count to 1

# Static data: shared mutable state

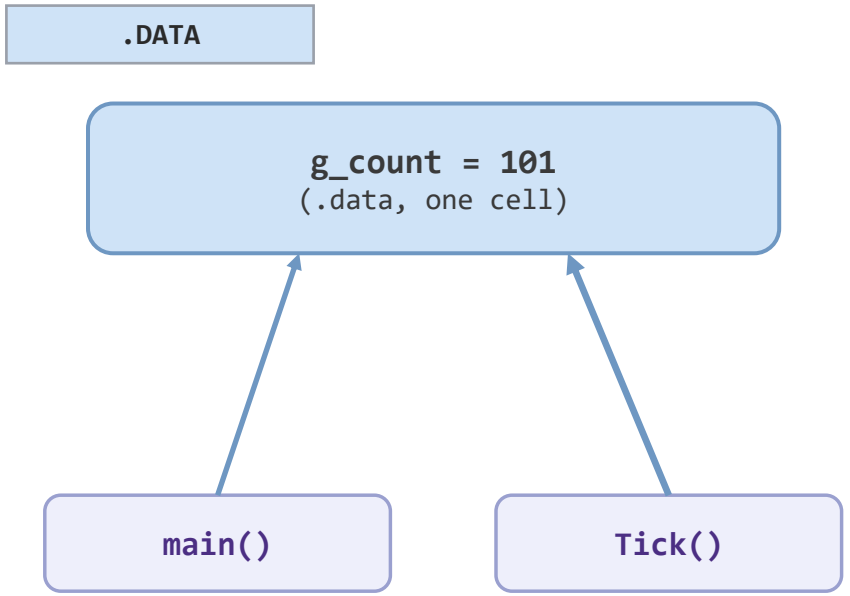
```
1  int g_count = 0;    // one global, in .data
2
3  void Tick(void) { g_count++; }
4
5  int main(int argc, char* argv[]) {
6    Tick();           // g_count -> 1
7    g_count = 100;    // changed from afar
8    Tick();           // g_count -> 101
9    // who set g_count? hard to reason
10   return EXIT_SUCCESS;
11 }
```




main writes 100 directly — from far away

# Static data: shared mutable state

```
1  int g_count = 0;    // one global, in .data
2
3  void Tick(void) { g_count++; }
4
5  int main(int argc, char* argv[]) {
6    Tick();           // g_count -> 1
7    g_count = 100;    // changed from afar
8    Tick();           // g_count -> 101
9    // who set g_count? hard to reason
10   return EXIT_SUCCESS;
11 }
```




another Tick → 101; who set it? hard to track

**Takeaway:** Not a crash, but a hazard: a global is one cell any function can change. That makes behavior hard to reason about and is a trap once threads share it. Prefer locals and parameters.

# Reminders

**Assessments**:

- Exercise 1 due yesterday at 11:59pm
  - Can still use one late day before we automatically release solutions
- Exercise 2 released today at some point. Due Thursday at 11:59pm
- Homework 1 released soon

**General**:

- Notify us ASAP if you have conflicts with
  - Midterm (7/27) from 9:40am to 10:40am in IEB G106
  - Final exam (8/18) from 1:30pm to 3:30pm in CSE2 G10
- Participation points will be updated over the weekend

# Questions?

Thanks for coming - see you next lecture!