UNIVERSITY of WASHINGTON

LECTURE 6 · CSE 333 · Summer 2026

Decompilation and File I/O

Discussion: What's the most recent compliment you've received?

Reminders

Assessments:

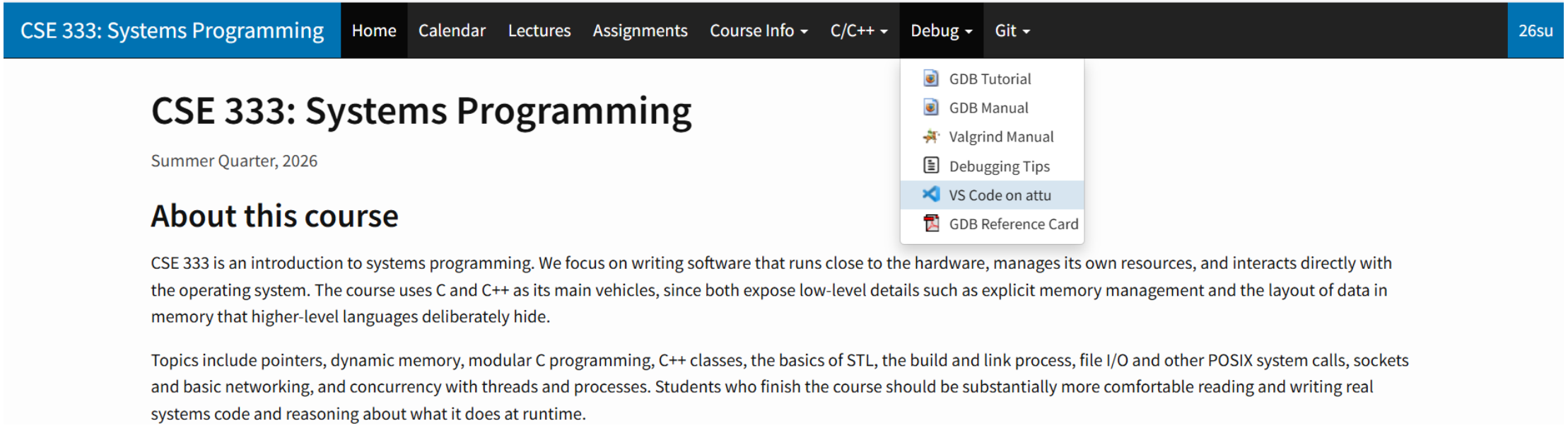
- Exercise 3 due Wednesday at 11:59pm
- Homework 1 due Thursday at 11:59pm

General:

- Sign up for a coffee chat!
- Exams are scheduled
  - 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
- Today is a Frankenstein lecture 🧟 (lots of disjoint topics)

VS Code Debugging

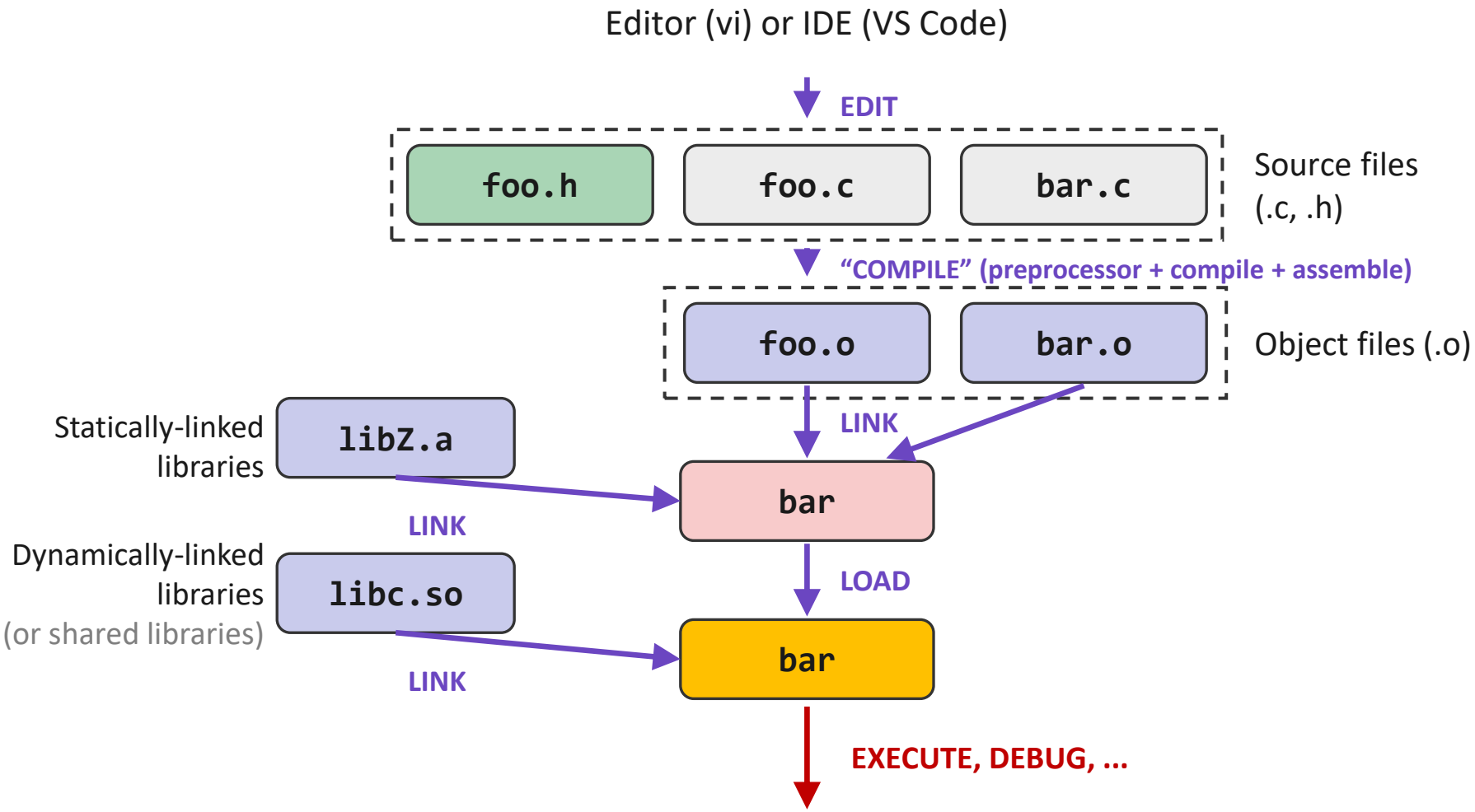
Check out the new guide!*



*Attu was down again over the weekend so the changes to the website haven't propagated yet...
*You will need to `git pull` and either resolve a merge conflict of run `git pull --rebase`

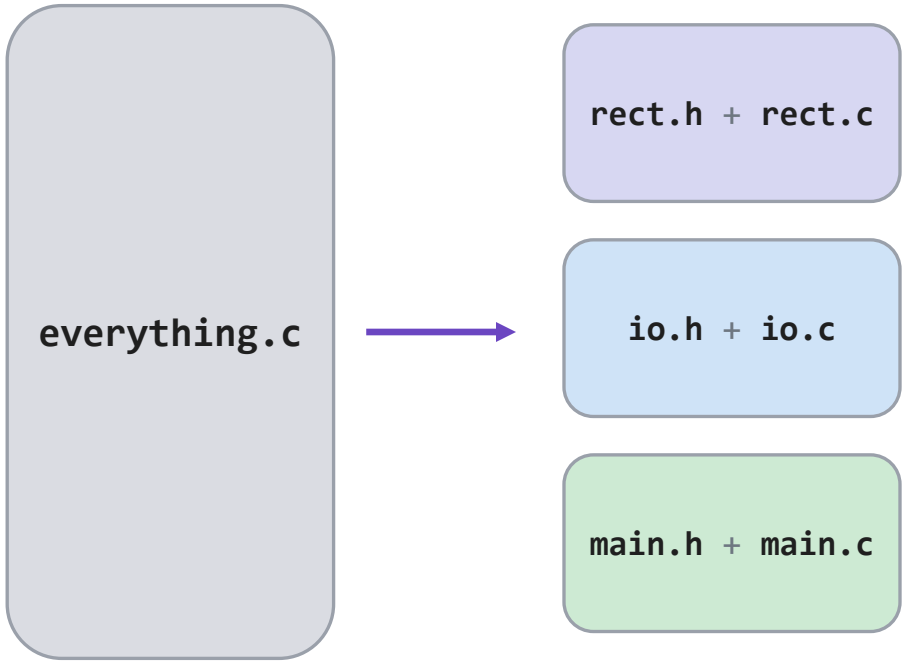
Review

Compilation



The benefits of modules

- **One giant file does not scale.** Thousands of lines are hard to read, search, and reason about.
- **Rebuild only what changed.** Touch one file and recompile just that piece, not the whole program.
- **Reuse.** A tested module can be dropped into another program without copy-paste.
- **Divide the work.** Different people can own different modules at the same time.
- **Hide the details.** Callers see the header only; the messy internals stay in the .c.



Every module has interface and implementation(s)

rect.h the interface

```
1  int RectArea(int w, int h);
2  int RectPerimeter(int w, int h);
```

❖ **Declarations only:** function prototypes, struct definitions, constants.

❖ Says **what** exists, not how it is done.

❖ This is what callers read and #include.

❖ Public documentation (no implementation details)

rect.c the implementation

```
1  #include "rect.h"
2
3  int RectArea(int w, int h) {
4    return w * h;
5  }
6  ...
```

❖ **Definitions:** the actual function bodies, the real code.

❖ Says **how** each thing is done.

❖ Callers never need to read this.

REVIEW

#define names a piece of text

You write

```
1  #define BAD 1 + 2
2  #define GOOD(x) (x + 2)
3
4  int x = BAD * 10;
5  int y = GOOD(3) * 10;
```

After the preprocessor runs

```
1  int x = 1 + 2 * 10;
2  int y = (3 + 2) * 10;
```

Rule: **#define NAME text** replaces every **NAME** below it with **text**, exactly as written.

Can use a parameter, e.g. **#define NAME(x) (x)** to do more complex replacements.

REVIEW

#include pastes a file in place

nums.h

```
1  int answer = 42;
```

main.c

```
1  #include "nums.h"
2  int x = answer;
```

After the preprocessor runs (main.c)

```
1  int answer = 42;
2  int x = answer;
```

Rule: **#include "nums.h"** copies the entire text of **nums.h** into main.c, right where the directive was.

Trickier example from prev. lecture

config.h

```
1  #define MAX 100
```

util.h

```
1  #include "config.h"
2  #define LIMIT (MAX - 1)
```

main.c

```
1  #include "util.h"
2  #define SCALE 2
3  int n = SCALE * LIMIT;
```

Preprocessor state

Macro	Replacement text
MAX	100
LIMIT	(MAX - 1)
SCALE	2

What the compiler sees

```
1  int n = 2 * (100 - 1);
```

Expand and rescan: **SCALE * LIMIT → 2 * (MAX - 1) → 2 * (100 - 1) = 198.**

Takeaway: An #include can pull in more #includes. Macros defined in any of them all land in the same shared state.

extern: shared variable across files

foo.c

```c
#include <stdio.h>
int counter = 1;       // external by default
void Bar(void);        // defined in bar.c

int main(void) {
  printf("%d\n", counter);
  Bar();
  printf("%d\n", counter);
  return EXIT_SUCCESS;
}
```

bar.c

```c
#include <stdio.h>
extern int counter;    // defined in foo.c
void Bar(void) {
  counter++;
  printf("Bar: counter = %d\n", counter);
}
```

foo.c defines **counter** (external by default).
bar.c only declares it with **extern**, so the linker points both files at the **one copy**. Bar's **++** is visible back in main.

output

```
1
Bar: counter = 2
2
```

- **bar.c** has **no counter of its own**.
- **extern** is a promise it lives in **foo.c**

static: different variable across files

foo.c

```c
#include <stdio.h>
static int counter = 1; // private to foo.c
void Bar(void);

int main(void) {
  printf("%d\n", counter);
  Bar();
  printf("%d\n", counter);
  return EXIT_SUCCESS;
}
```

bar.c

```c
#include <stdio.h>
static int counter;   // private to bar.c
void Bar(void) {
  counter++;
  printf("Bar: counter = %d\n", counter);
}
```

static gives each **counter** internal linkage
foo.c and **bar.c** get **separate variables** that share a name.
Nothing bar.c does can reach foo's copy, which is why we print **1**.

output

```
1
Bar: counter = 101
1
```

- Two **counter**s: 1 and 100.
- Bar bumps **its own** to 101.
- foo's stays **1**.

REVIEW

static functions

util.c

```c
#include <stdio.h>

// private: only util.c calls it
static int Triple(int x) {
  return 3 * x;
}

// external: anyone may call it
int Scale(int x) {
  return Triple(x) + 1;
}
```

main.c

```c
#include <stdio.h>

int Scale(int x);     // extern by default
// int Triple(int x); // static: link error

int main(void) {
  printf("%d\n", Scale(5));  // gives 16
  return 0;
}
```

- **static int Triple** is internal: only **util.c** can call it.
- **int Scale** is external (default), so **main.c** can declare and call it.
- Calling Triple from main.c is a **linker error**: undefined reference to Triple.

REVIEW

static for local variables (different!)

```c
#include <stdio.h>

void Foo(void) {
  static int count = 0;   // persists across calls
  count++;
  printf("Foo called %d times\n", count);
}

void Bar(void) {
  int count = 0;          // fresh each call
  count++;
  printf("Bar called %d times\n", count);
}

int main(void) {
  Foo(); Foo(); Foo();
  Bar(); Bar(); Bar();
  return EXIT_SUCCESS;
}
```

output

```
Foo called 1 times
Foo called 2 times
Foo called 3 times
Bar called 1 times
Bar called 1 times
Bar called 1 times
```

- Inside a function, **static** is not about linkage.
- Foo's **count** gets **one storage** that lives for the whole program and **keeps its value** between calls.
- Bar's plain **count** is **re-created** at 0 on every call.
- **Same keyword, different job.** Context tells you which.

More Preprocessor

Example #1

date.h

```
1  struct Date { int y, m, d; };
```

event.h

```
1  #include "date.h"
2  struct Event { struct Date start, end; };
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

- date.h defines a Date struct.
- event.h defines an Event struct (which consists of two Date structs). It must #includes date.h
- main.c needs both so it #includes date.h and event.h.

- **The question:** we know #include is literal copy-paste. So when we build main.c, what text does the compiler actually end up seeing?

Example #1

date.h

```
1  struct Date { int y, m, d; };
```

event.h

```
1  #include "date.h"
2  struct Event { struct Date start, end; };
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees (main.c expanded)

```
1
2
3
4
```

The three files, before we start. We compile **main.c**, so the preprocessor walks it top to bottom and pastes each #include in place.

Example #1

date.h

```
1  struct Date { int y, m, d; };
```

event.h

```
1  #include "date.h"
2  struct Event { struct Date start, end; };
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees (main.c expanded)

```
1  struct Date { int y, m, d; };
2
3
4
```

main.c line 1: **#include "date.h"** pastes date.h here, so **struct Date** lands in the output.

Example #1

date.h

```
1  struct Date { int y, m, d; };
```

event.h

```
1  #include "date.h"
2  struct Event { struct Date start, end; };
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees (main.c expanded)

```
1  struct Date { int y, m, d; };
2  struct Date { int y, m, d; };
3
4
```

main.c line 2: **#include "event.h"**. But event.h's own first line is **#include "date.h"**, so struct Date gets pasted a **second time.**

Example #1

date.h

```
1  struct Date { int y, m, d; };
```

event.h

```
1  #include "date.h"
2  struct Event { struct Date start, end; };
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees (main.c expanded)

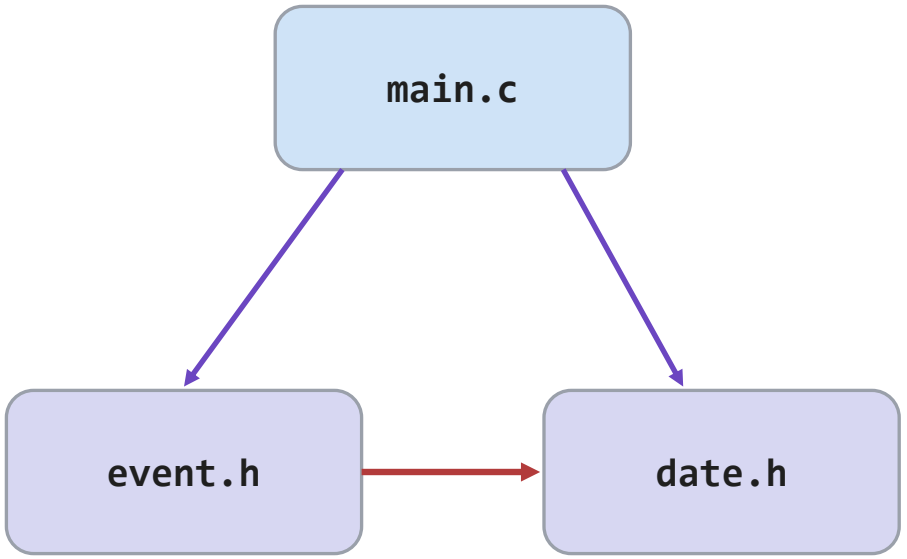
```
1  struct Date { int y, m, d; };
2  struct Date { int y, m, d; };
3  struct Event { struct Date start, end; };
4  int main() { ... }
```

compiler error

```
main.c: error: redefinition of 'struct Date'
date.h: note: originally defined here
```

Then event.h's **struct Event** and main's body paste in. Result: **struct Date is defined twice.**

Multiple includes



Two paths reach **date.h**, **so it is pasted twice.**

❖ **You did not include date.h twice on purpose.** It came in once directly and once through event.h.

❖ **This is normal.** Real projects have deep include trees; the same header is reached along many paths all the time.

❖ **The goal:** a header should be safe to #include any number of times, but its contents should only take effect **once.**

❖ **The idea:** have the preprocessor remember the first time a header is pasted and skip that header every time after.

Takeaway: Double inclusion is not a mistake you made; it is a structural fact of any real codebase!

The pattern: #ifndef / #define / #endif

date.h (guarded)

```
1  #ifndef DATE_H_
2  #define DATE_H_
3  struct Date {
4    int y, m, d;
5  };
6  #endif  // DATE_H_
```

Read it as: "if DATE_H_ has not been defined, keep everything down to #endif; otherwise, delete all of it."

- **#ifndef DATE_H_**
  - "if not defined." The whole block is kept only when DATE_H_ is not yet a macro.
  - The name DATE_H_ is arbitrary,
- **#define DATE_H_**
  - leave the note. Recall a #define just names some text; here the text is empty, so DATE_H_ is only a flag.
- **...the real contents...**
  - your struct, prototypes, and #defines go here.
- **#endif**
  - closes the #ifndef block.

Example #1 (fixed)

date.h (guarded)

```h
#ifndef DATE_H_
#define DATE_H_
struct Date {
  int y, m, d;
};
#endif  // DATE_H_
```

event.h (guarded)

```h
#ifndef EVENT_H_
#define EVENT_H_
#include "date.h"
struct Event {
  struct Date start, end;
};
#endif  // EVENT_H_
```

main.c

```c
#include "date.h"
#include "event.h"
int main() { ... }
```

What the compiler sees

```
1
2
3
4
5
6
```

Example #1 (fixed)

date.h (guarded)

```
1  #ifndef DATE_H_
2  #define DATE_H_
3  struct Date {
4    int y, m, d;
5  };
6  #endif  // DATE_H_
```

event.h (guarded)

```
1  #ifndef EVENT_H_
2  #define EVENT_H_
3  #include "date.h"
4  struct Event {
5    struct Date start, end;
6  };
7  #endif  // EVENT_H_
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees

```
1  struct Date {
2    int y, m, d;
3  };
4
5
6
```

Example #1 (fixed)

date.h (guarded)

```
1  #ifndef DATE_H_
2  #define DATE_H_
3  struct Date {
4    int y, m, d;
5  };
6  #endif  // DATE_H_
```

event.h (guarded)

```
1  #ifndef EVENT_H_
2  #define EVENT_H_
3  #include "date.h"
4  struct Event {
5    struct Date start, end;
6  };
7  #endif  // EVENT_H_
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees

```
1  struct Date {
2    int y, m, d;
3  };
4
5
6
```

Example #1 (fixed)

date.h (guarded)

```
1  #ifndef DATE_H_
2  #define DATE_H_
3  struct Date {
4    int y, m, d;
5  };
6  #endif  // DATE_H_
```

event.h (guarded)

```
1  #ifndef EVENT_H_
2  #define EVENT_H_
3  #include "date.h"
4  struct Event {
5    struct Date start, end;
6  };
7  #endif  // EVENT_H_
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees

```
1  struct Date {
2    int y, m, d;
3  };
4
5
6
```

Example #1 (fixed)

date.h (guarded)

```
1  #ifndef DATE_H_
2  #define DATE_H_
3  struct Date {
4    int y, m, d;
5  };
6  #endif  // DATE_H_
```

event.h (guarded)

```
1  #ifndef EVENT_H_
2  #define EVENT_H_
3  #include "date.h"
4  struct Event {
5    struct Date start, end;
6  };
7  #endif  // EVENT_H_
```

main.c

```
1  #include "date.h"
2  #include "event.h"
3  int main() { ... }
```

What the compiler sees

```
1  struct Date {
2    int y, m, d;
3  };
4  struct Event {
5    struct Date start, end;
6  };
```

Doing it right, every time

- **Guard every header.** Wrap the entire contents of every .h in the three lines. Make it a habit, not a decision.
- **Name it after the file.** A common convention: the filename in caps with a trailing underscore.
- **Make the name unique.** If two headers share a guard name, the second one gets silently skipped. That bug is nasty to find.
- **It also stops infinite include loops** when two headers include each other.
- **#pragma once** does the same thing in one line. It is widely supported but *technically* not standard C

the standard shape of any .h

```
1  #ifndef DATE_H_
2  #define DATE_H_
3
4  // ... all declarations ...
5
6  #endif  // DATE_H_
```

or, non-standard shorthand (probably fine?)

```
1  #pragma once
2  // ... all declarations ...
```

Pokémon

20 years of Pokemon, three languages

	Games	Years	Console	Language
Gen 1	Red · Blue · Yellow	1996–98	Game Boy	**Assembly**
Gen 2	Gold · Silver · Crystal	1999–2000	Game Boy Color	**Assembly**
Gen 3	**Ruby · Sapphire · Emerald · FireRed · LeafGreen**	2002–04	Game Boy Advance	**C**
Gen 4	Diamond · Pearl · HGSS	2006–09	Nintendo DS	**C / C++**
Gen 5	Black · White · B2W2	2010–12	Nintendo DS	**C / C++**
Gen 6	X · Y · ORAS	2013–14	Nintendo 3DS	**C++**
Gen 7	Sun · Moon · USUM	2016–17	Nintendo 3DS	**C++**
Gen 8	Sword · Shield	2019	Switch	**C++**
Gen 9	Scarlet · Violet	2022	Switch	**C++ / C#**

Handhelds started in raw **assembly**, moved to **C** on the GBA, then **C++** once consoles grew. Our target is Gen 3: **pokeemerald**, the Emerald decomp, in plain C.

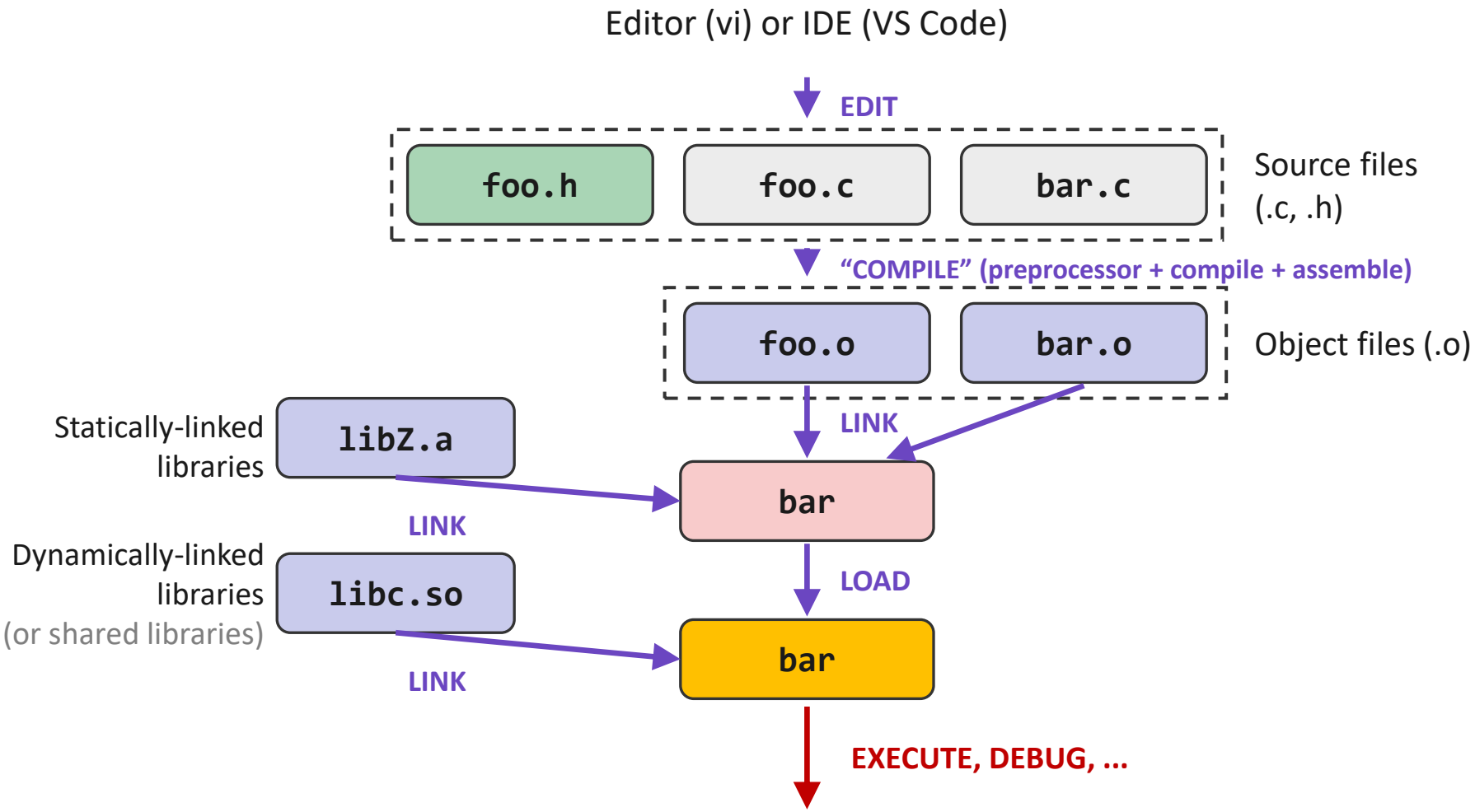
Game Boy Advance SP

What is decompilation?

- Compilation turns **source code** into a **binary** the machine runs.
- Decompilation goes the other way: start from the shipped **binary**, recover readable source.
- Nintendo never released the Pokemon source. All we have is the ROM.
- A **matching** decompilation is the gold standard: the recovered C must compile back to the **exact same bytes**.
  - If it rebuilds byte-for-byte, you have proven the source is faithful.

Takeaway: Decompilation is compilation run backwards: from a finished binary back to source you can read, edit, and rebuild.

Compilation



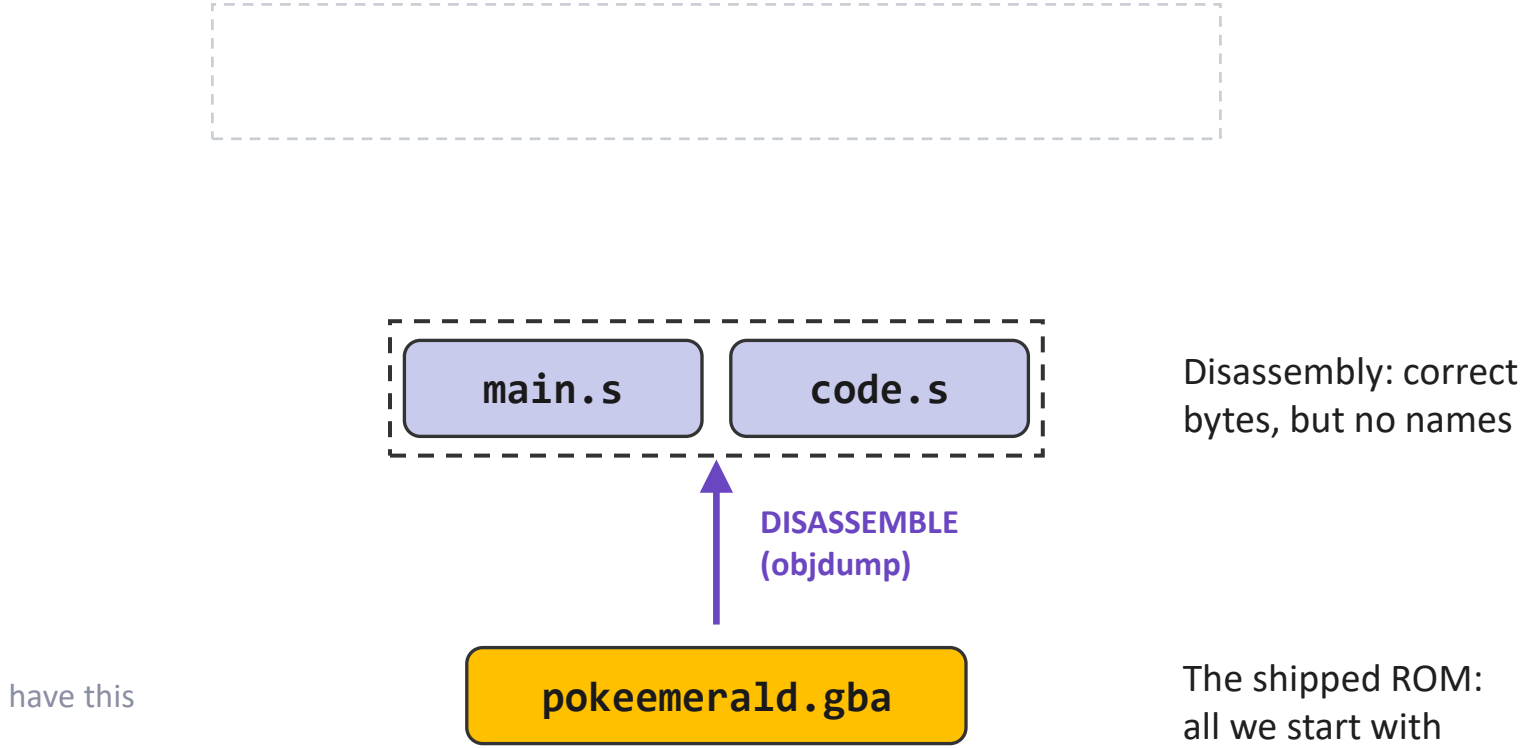
Compilation, in reverse

have this

`pokeemerald.gba`

The shipped ROM:
all we start with

Compilation, in reverse

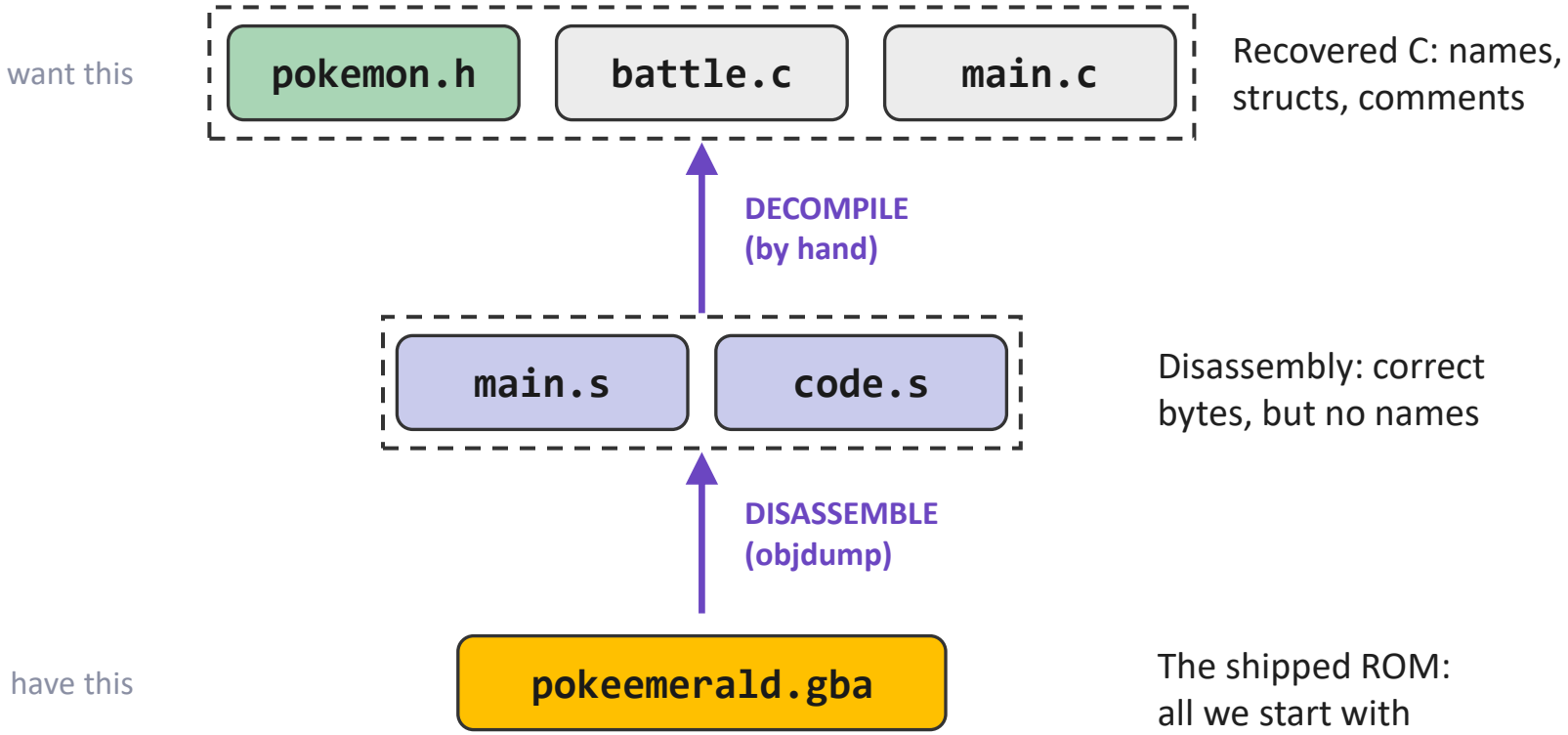


Disassembly: correct bytes, but no names

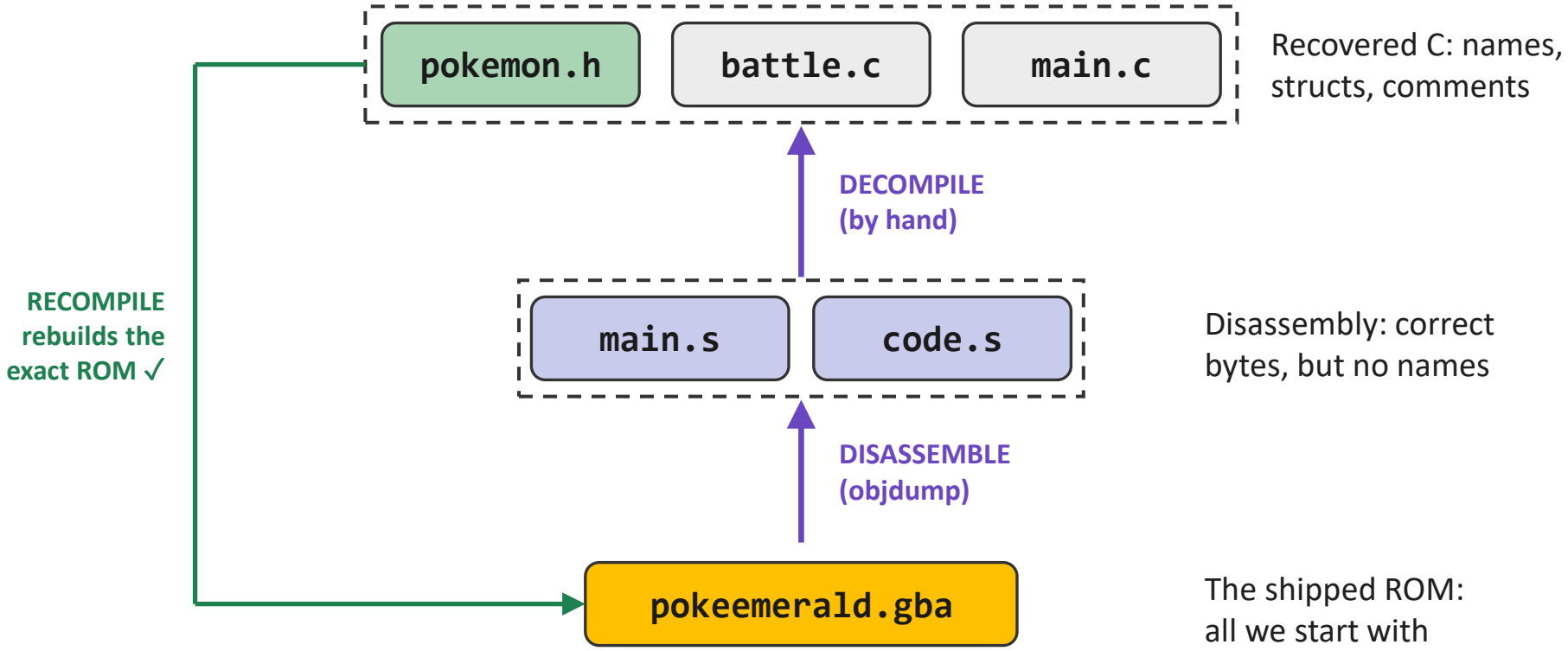
have this

The shipped ROM: all we start with

Compilation, in reverse



Compilation, in reverse!



Takeaway: Decompilation is the just the compilation pipeline in reverse.

Start here: the ROM is just bytes

```
$ file pokeemerald.gba
pokeemerald.gba: Game Boy Advance ROM image

$ xxd pokeemerald.gba          // dump the raw bytes
0008a2c0: 0018 7047 80b5 0446 ...
          └─────────── 4 bytes = one tiny function
```

- No names. No types. No functions.
- The compiler threw all of that away.
- It is **machine code**: raw numbers the CPU runs.
- Those 4 bytes **00 18 70 47** are our example.

Bytes → assembly (programmatic)

```
// the 4 bytes, as the CPU sees them
00 18   70 47

// little-endian 16-bit THUMB opcodes:
0x1800   0x4770
```

```
$ objdump -d pokeemerald.gba

0800a2c0 <sub_800A2C0>:
   adds   r0, r0, r0   // 0x1800
   bx     lr           // 0x4770
```

- THUMB is the instruction set for the GameBoy (equivalent to AMD for computers or ARM for phones)
- A disassembler maps each byte pattern to exactly one **instruction**. It is automatic, but you still get no names: the function is just **its address**.

Assembly → C (the human-hard part)

```
0800a2c0 <sub_800A2C0>:
   adds   r0, r0, r0    // r0 = r0 + r0
   bx     lr            // return r0
```

→

```
// recovered C (names are our guess)
int Double(int x) {
    return x + x;
}
```

- ARM convention: the first argument arrives in **r0**, and the return value goes back in **r0**. So r0 is **x**.
- **adds r0, r0, r0** is **x + x**; **bx lr** is **return**. Tools like Ghidra automate most of this.

pokeemerald: a "matching" decompilation

```
// src/data/battle_moves.h  (real, verbatim)
[MOVE_POUND] = {
    .effect = EFFECT_HIT,
    .power = 40,
    .type = TYPE_NORMAL,
    .accuracy = 100,
    .pp = 35,
    .priority = 0,
},
// this C compiles to the exact bytes
// found in the retail Emerald cartridge
```

- Built by **pret**, a Pokemon reverse-engineering community.
- Emerald is fully decompiled: 100% matching C.
- Rebuilds **pokeemerald.gba** byte-for-byte.
- Readable names, structs, and comments the original never shipped.
- This is the C you are learning!

Takeaway: pokeemerald is real, readable C that recompiles to the exact retail ROM. That byte match is what makes it trustworthy.

What a matching decompilation gets you

- **Understanding:** you can finally read how a beloved game actually works.
- **Preservation:** readable source outlives fragile cartridges and dead hardware.
- **Modding:** change the C, rebuild, and you have a brand-new ROM hack.
- **Proof:** the byte match guarantees the source is faithful, not a guess.
- **Learning:** it is real, shipped C at a scale you can actually explore.

Check it out at **https://github.com/pret/pokeemerald/**

- We are familiar with *most* but not all of the syntax used
- Recommend starting with smaller “concepts” like items.h and money.h rather than pokemon.h
- All sorts of projects built in C (e.g. git)

Hacking Pokémon

Who is allowed to touch memory?

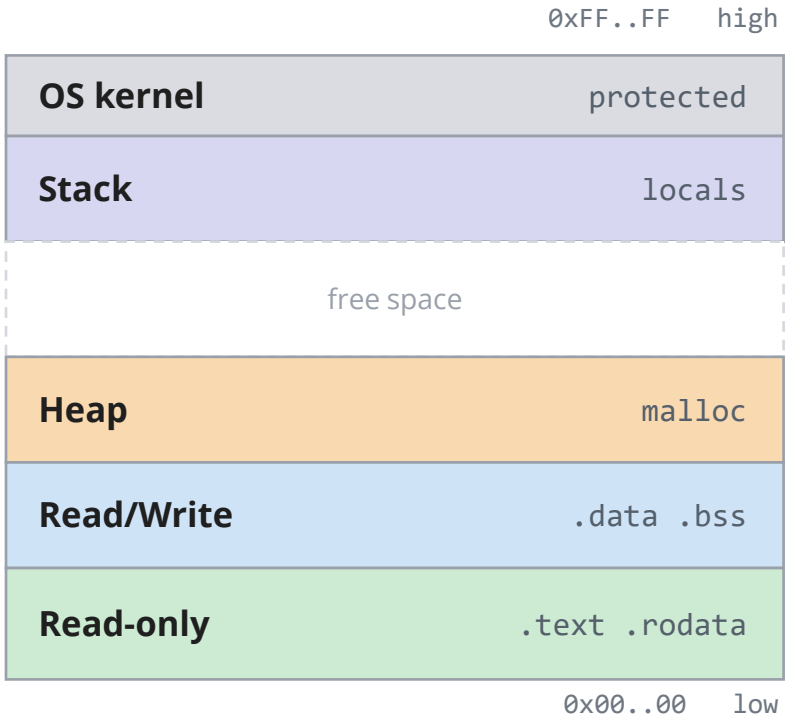
- Your C code is full of **addresses**: every pointer and variable lives somewhere in memory.
- But who decides where things go, and who guards the door?
- On a normal computer, the **operating system** sits between your program and memory.

- On the **GBA** there is no OS: the game touches memory directly.
- A cheat device like **Action Replay** only works because that door is wide open.

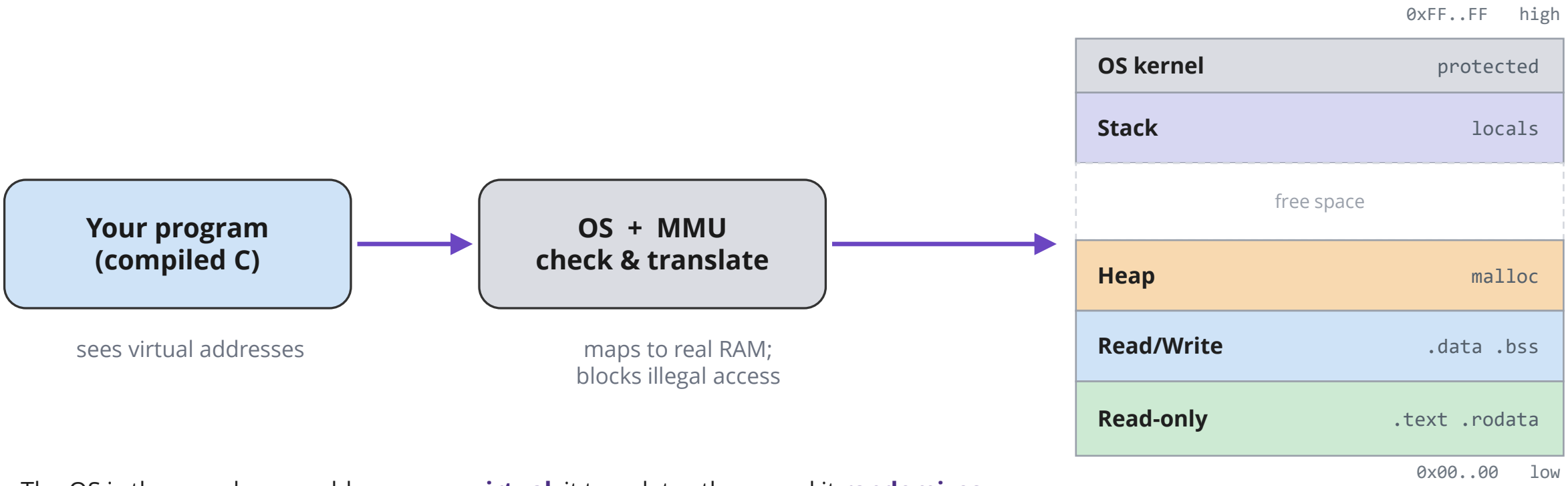
RECALL FROM LAST LECTURE

A program's memory: the address space

- Code and constants at the bottom (**.text**), globals in the **data** segment.
- The **heap** grows up (malloc), the **stack** grows down (locals).
- A pointer is just an address into this column.
- Keep this picture: it is the memory in every diagram today.

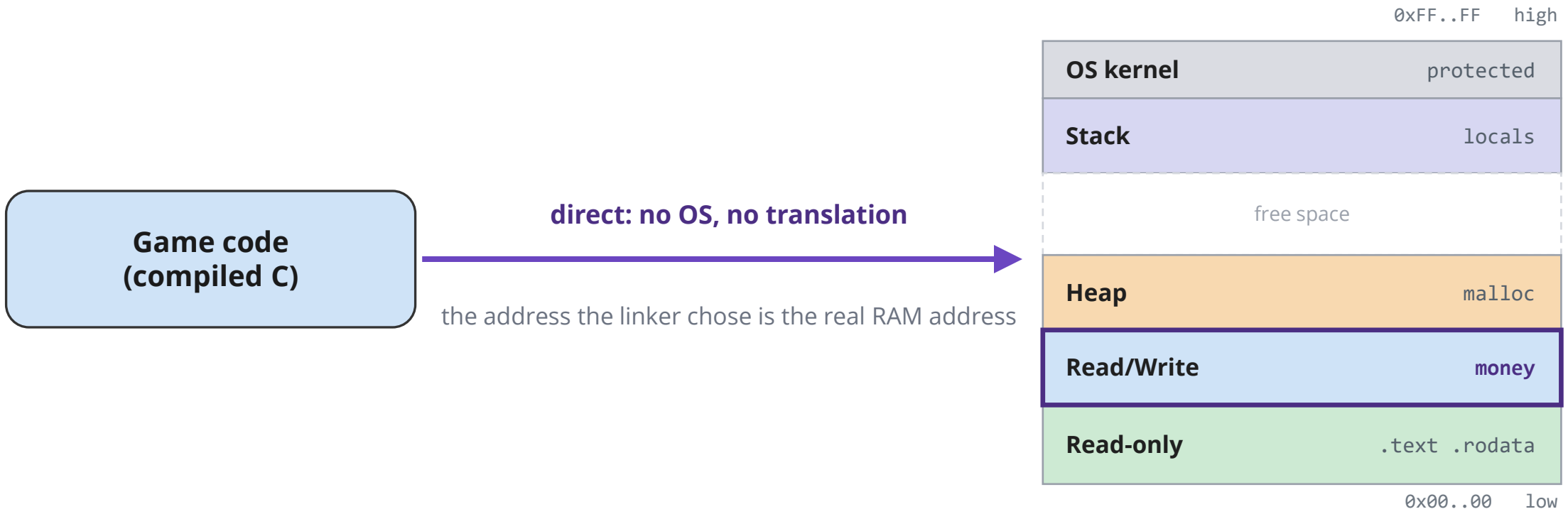


program → OS → memory



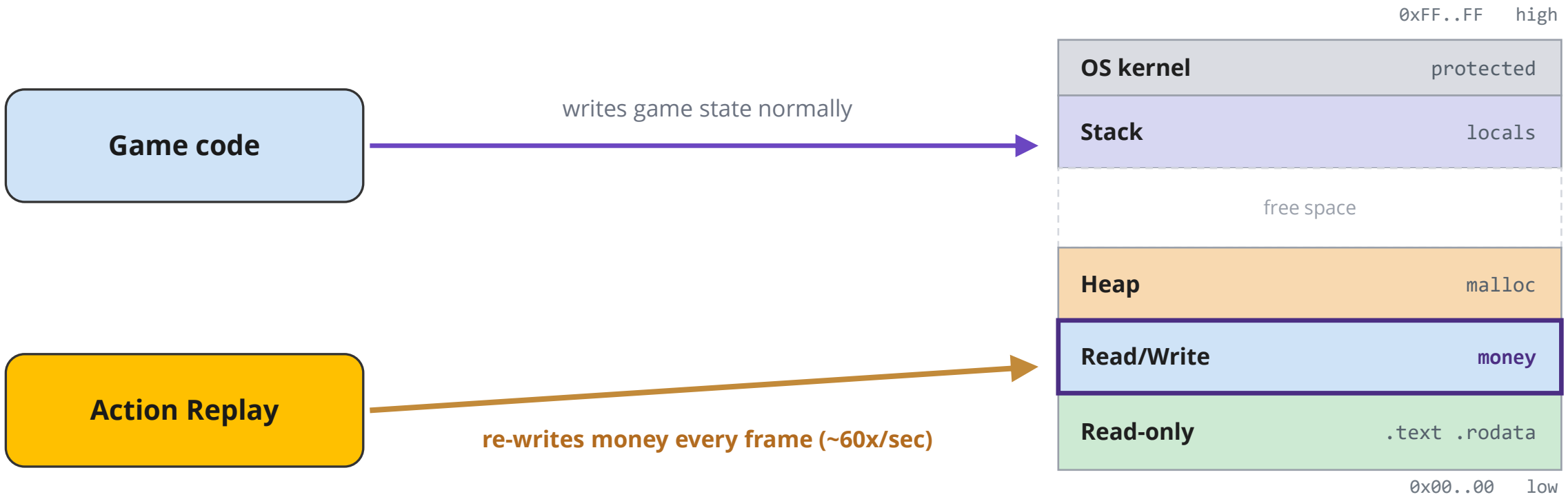
The OS is the guard: your addresses are **virtual**, it translates them, and it **randomizes them each run (ASLR)**.

program → memory (no OS)

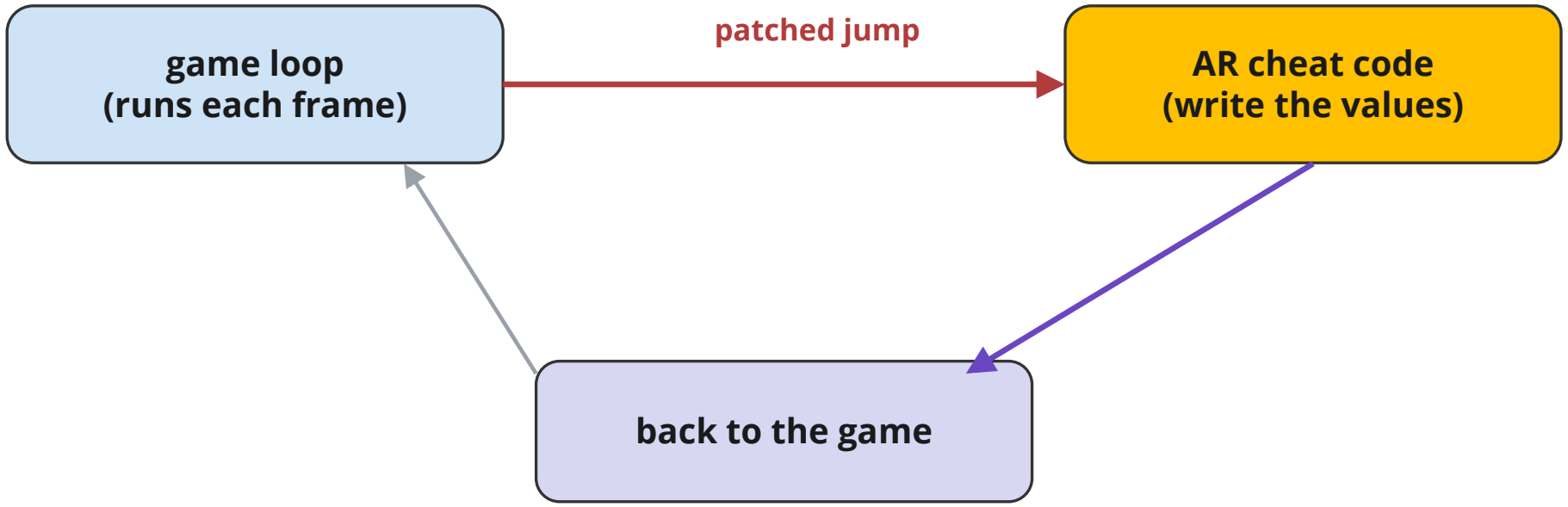


* on real hardware the save/data region is called EWRAM, at a fixed address like 0x02..... Same idea as the data segment.

A second writer into the same memory



How does the AR run every frame?

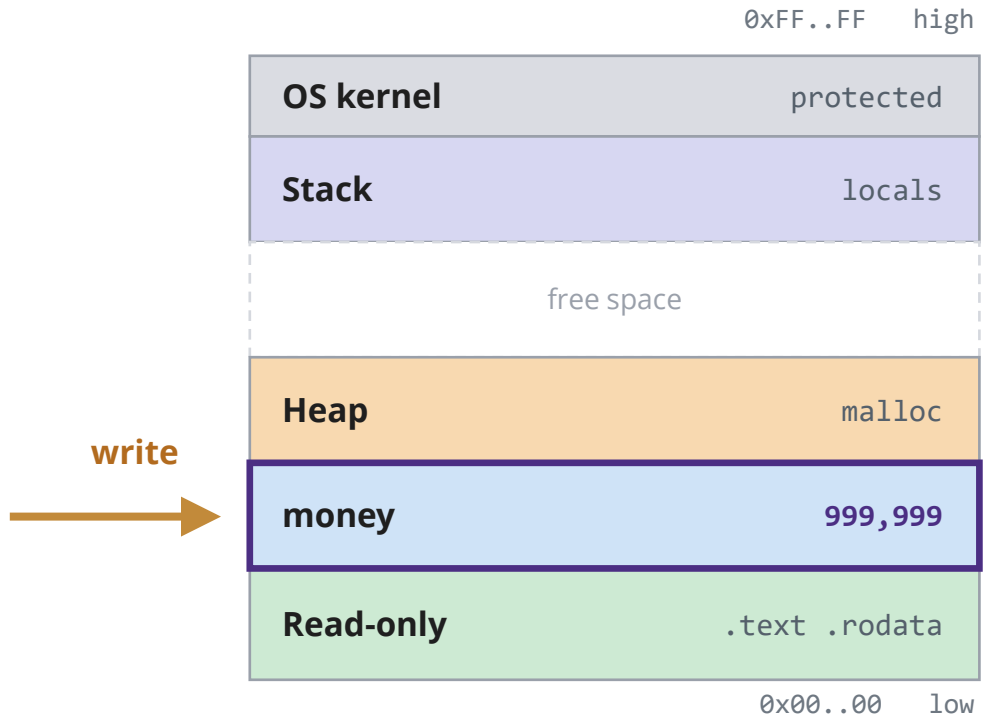


❖ The **Master Code** overwrites one jump instruction in the game so control detours into the AR, **like repointing a function pointer**.

A REAL CHEAT

Emerald Infinite Money = address + value

```
1  // Infinite Money (what it means):
2
3  address = 0x02025E90    // where money lives
4  value   = 999,999       // the money cap
5
6  // the AR writes value -> address, every frame
```



* the code you actually type is encrypted; the AR decrypts it into this address + value first. Ignore that for now.

What 0x02025E90 actually is

```
// pokeemerald: src/new_game.c
SetMoney(&gSaveBlock1Ptr->money, 3000);  // starting cash

// src/money.c  (simplified)
void SetMoney(u32 *moneyPtr, u32 newValue) {
    *moneyPtr = newValue;   // * game also scrambles it
}
```

- The decomp gives the raw address a name.
- **0x02025E90** is **gSaveBlock1Ptr->money**.
- The game reads and writes it through SetMoney / GetMoney.
- The AR knows none of that: it just pokes the bytes.

* the game also scrambles the stored value with a key, so the on-screen number can look weird, but it stays huge.

File IO

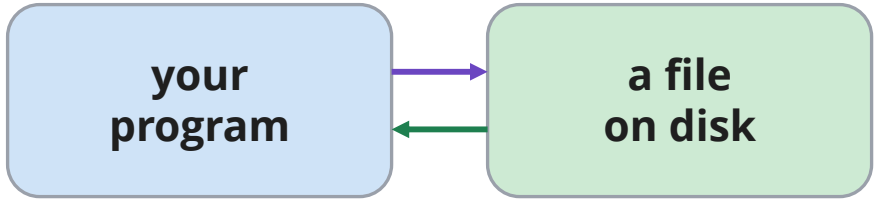
Programs need to talk to the outside world

❖ **So far, data lived in memory** and vanished when the program ended.

❖ Real programs persist data: save a file, load a config, read input, write a log.

❖ **The C standard library** gives us a simple, portable way to do this.

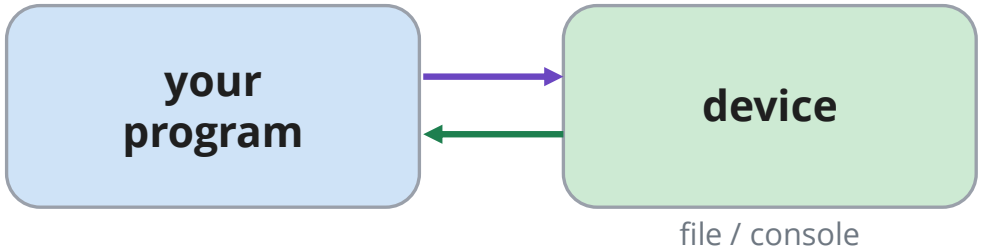
❖ **Our plan for today:** open a file, write to it, read from it, close it, one function at a time.



read and write, in both directions

First idea: a stream

- **A stream is a sequence of bytes** flowing to or from a device.
- The device can be anything: a file, the console, or a pipe. Same idea for all of them.
- Text or binary: it is all just bytes; Linux does not distinguish.
- **You do not read the whole file at once.** Bytes flow through the stream a chunk at a time.



the stream

Three streams you have already used

`stdin`	**standard input** keyboard by default	`scanf reads it`
`stdout`	**standard output** the console by default	`printf writes it`
`stderr`	**standard error** the console, for messages	`fprintf(stderr,...) writes`

You never opened these: **they are already open when your program starts.** So yeah… printf is really writing to the stdout stream!

FILE*: the handle you hold

- A **FILE*** is your handle to a stream.
- **You do not look inside it.** It bundles the details (the buffer, the position, error flags) that libc manages for you.
- You pass the FILE* to every stream function so it knows which stream you mean.
- Defined in **stdio.h**.
- stdin, stdout, stderr are FILE*s too!

```c
#include <stdio.h>
FILE* fp;   // a stream handle
```

You can think of it like a ticket. fopen hands you a FILE*, you show that ticket to fread / fwrite / fclose, and they act on the right stream. You rarely care what is printed on the ticket.

fopen: open a file, get a stream

```
1  FILE *fopen(const char *filename, const char *mode);
```

```
1  FILE *fp = fopen("notes.txt", "r");
```

- **filename** is the path to the file you want.
- **mode** says what you intend to do: read, write, or append (next slide).
- Returns a **FILE*** you use for everything after, or **NULL** if it could not open.

Access modes: what you plan to do

`"r"`	**read**	open an existing file for reading
`"w"`	**write**	create/truncate a file for writing
`"a"`	**append**	write, but add onto the end
`"r+"`	**read + write**	open existing for both

Add a **b** for binary, like **"rb"** or **"wb"**. On Linux it changes nothing (bytes are bytes), but it is good habit and matters on Windows.

fopen can fail: always check for NULL

```
1  FILE* fp = fopen("notes.txt", "r");
2  if (fp == NULL) {
3    fprintf(stderr, "cannot open file\n");
4    return EXIT_FAILURE;
5  }
```

- The file may not exist, or you may lack permission.
- On failure fopen returns **NULL**.
- **fprintf(stderr, ...)** writes your message to the error stream.
- **Check every fopen.** Using a NULL FILE* later crashes.

fclose: always close what you open

```
1  int fclose(FILE *stream);
```

- **Every fopen deserves an fclose.** You are done with the stream; hand it back.
- **Closing flushes buffered writes** to the file (more on buffering later), so the last bytes actually land.
- It frees resources. Open files are limited; leaking them is a real bug.
- Returns 0 on success, **EOF** on error.

The smallest complete program

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

int main() {
  FILE* fp = fopen("notes.txt", "w");
  if (fp == NULL) {
    fprintf(stderr, "cannot open\n");
    return EXIT_FAILURE; // should use a better exit code
  }
  fclose(fp);
  return EXIT_SUCCESS;
}
```

❖ **Open, check, close.** That is the whole skeleton.

❖ This really does something: mode "w" creates an empty notes.txt.

❖ **Everything else** goes between the open and the close.

Writing text: fprintf

```c
int fprintf(FILE* stream, const char* fmt, ...);
```

```c
fprintf(fp, "Score: %d\n", 42);
```

- **It is just printf with a stream in front.** Same format string, same %-conversions.
- **In fact: printf(...)** is exactly **fprintf(stdout, ...)**.
- Great for human-readable output: logs, reports, CSV, config.

fprintf in action

```c
FILE* fp = fopen("log.txt", "w");
fprintf(fp, "hello\n");
fprintf(fp, "x = %d\n", 7);
fclose(fp);
```

produces

log.txt

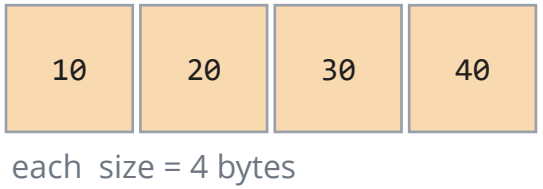
```
hello
x = 7
```

Each fprintf appends where the last one left off. **The stream remembers your position, so line two follows line one, with no need to say where.**

Writing raw bytes: fwrite

```
1  size_t fwrite(ptr, size, count, stream);
```

```
1  int a[4] = {10, 20, 30, 40};
2  fwrite(a, sizeof(int), 4, fp);
```



- Writes **count** items of **size** bytes each: here 4 ints, 16 bytes total.
- Use it for binary data: arrays, structs, whole buffers, in one call.
- Returns the number of items written (should be count; less means trouble).

Reading raw bytes: fread

```
size_t fread(ptr, size, count, stream);
```

```
int a[4];
size_t got = fread(a, sizeof(int), 4, fp);
```

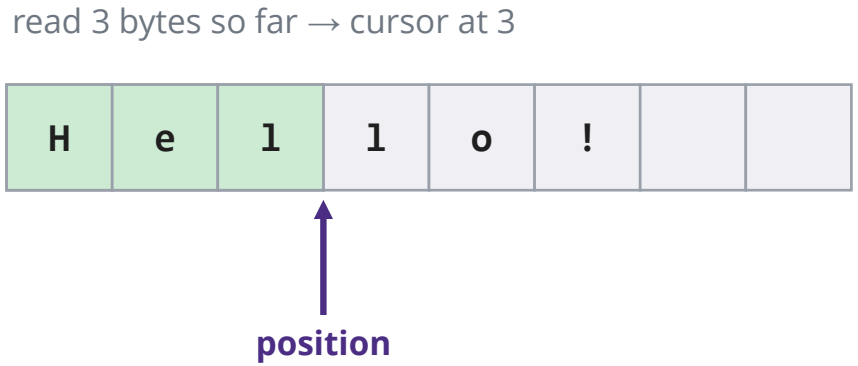
- **The exact mirror of fwrite:** same ptr, size, count, stream.
- Copies bytes from the stream into your buffer at ptr.
- **Returns how many items it actually read.** Fewer than count means end-of-file (or an error), so always check **got**.

One shared idea: the file position

❖ **Every open stream has a position,** a cursor into the file.

❖ Read or write, and it advances by that many bytes, automatically.

❖ **So repeated calls march forward:** read the first chunk, then the next, then the next.



Takeaway: Always remember that reads and writes move a the position cursor forward, so you naturally process a file in order.

Return values and errors

- **Every call reports success** through its return value: **NULL** from fopen, a short count from fread/fwrite.
- **errno** is a global the library sets to say what went wrong.
- **fprintf(stderr, ...)** sends your error message to the error stream, so it stays separate from normal output.
- **Check as you go.** Do not wait until the end to discover a write failed.

```
int ferror(stream);
```

did an error happen on this stream?

```
void clearerr(stream);
```

reset the error / EOF flags

```
fprintf(stderr, fmt, ...);
```

print an error message

error output (to stderr):

```
cannot open file
```

Step 1 of 4

Putting it together: copy a file

cp_example.c (the core loop)

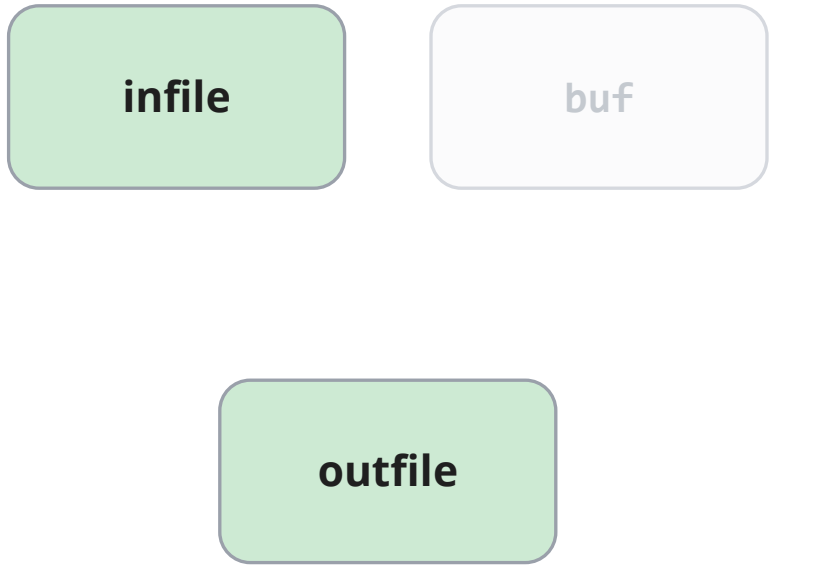
```
1  FILE *fin = fopen(argv[1], "rb");
2  if (fin == NULL) { fprintf(stderr, "no input\n"); return 1; }
3
4
5
6
7
8
9
10
11
12
13
```

Open the input for reading. If fopen returns **NULL**, report it and stop.

Putting it together: copy a file

cp_example.c (the core loop)

```
1  FILE *fin = fopen(argv[1], "rb");
2  if (fin == NULL) { fprintf(stderr, "no input\n"); return 1; }
3  FILE *fout = fopen(argv[2], "wb");
4  if (fout == NULL) {
5    fprintf(stderr, "no output\n");
6    fclose(fin);
7    return EXIT_FAILURE;
8  }
9
10
11
12
13
```

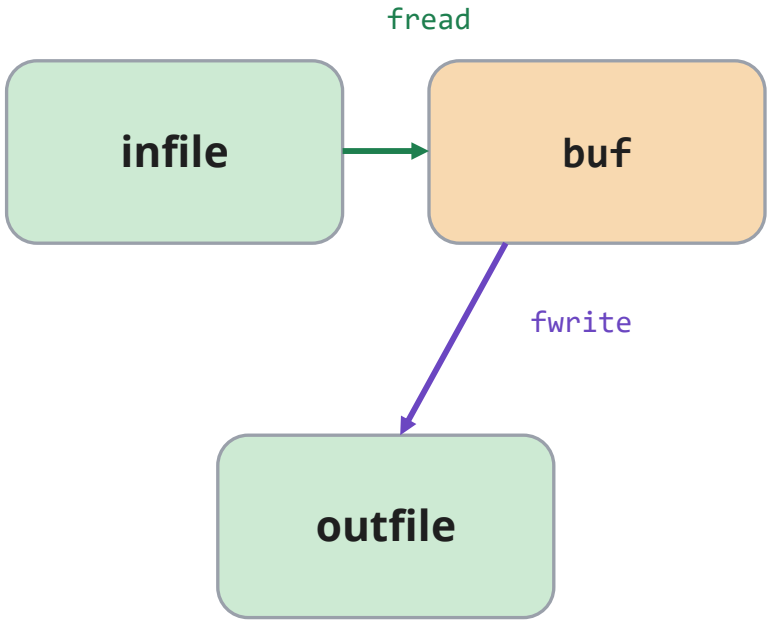


Open the output for writing. Same check, but now we also **fclose(fin)** before leaving, so we do not leak the stream we already opened.

Putting it together: copy a file

cp_example.c (the core loop)

```
1  FILE *fin = fopen(argv[1], "rb");
2  if (fin == NULL) { fprintf(stderr, "no input\n"); return 1; }
3  FILE *fout = fopen(argv[2], "wb");
4  if (fout == NULL) {
5    fprintf(stderr, "no output\n");
6    fclose(fin);
7    return 1;
8  }
9  while ((n = fread(buf, 1, SIZE, fin)) > 0) {
10   fwrite(buf, 1, n, fout);
11 }
12
13
```

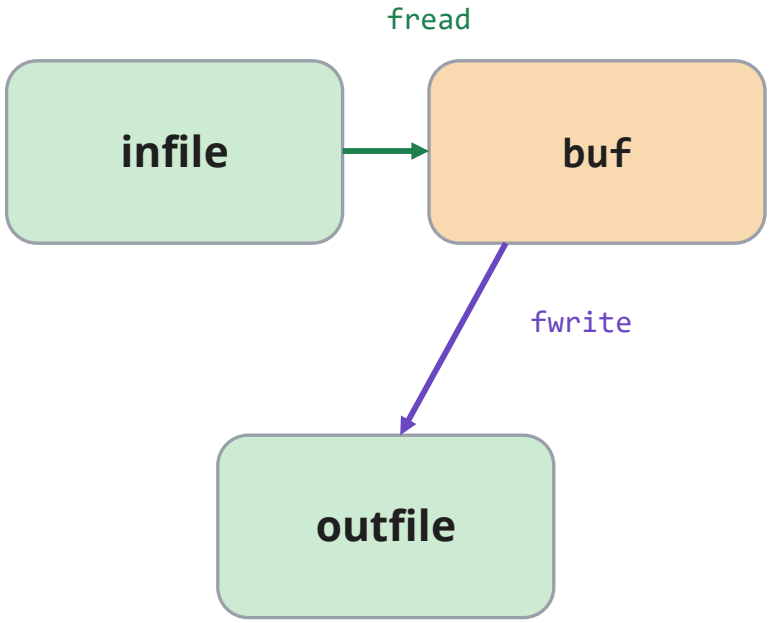


The copy loop: **fread** a chunk, then **fwrite** exactly the **n** bytes we got. Repeat until fread returns 0 (end of file).

Putting it together: copy a file

cp_example.c (the core loop)

```
1  FILE *fin = fopen(argv[1], "rb");
2  if (fin == NULL) { fprintf(stderr, "no input\n"); return 1; }
3  FILE *fout = fopen(argv[2], "wb");
4  if (fout == NULL) {
5    fprintf(stderr, "no output\n");
6    fclose(fin);
7    return 1;
8  }
9  while ((n = fread(buf, 1, SIZE, fin)) > 0) {
10   fwrite(buf, 1, n, fout);
11 }
12 fclose(fin);
13 fclose(fout);
```



Close both streams. **fclose** flushes buffered output so the final bytes truly reach the file.

File Buffering

Intuition: it is like a video buffering

❖ **Watch a video on YouTube or Netflix** and the player quietly downloads ahead of where you are, into a buffer.

❖ Playback stays smooth because it reads from that local buffer, not straight from the slow network.

❖ Network hiccup? You keep watching from the buffer. Only when it runs dry do you see “buffering…”

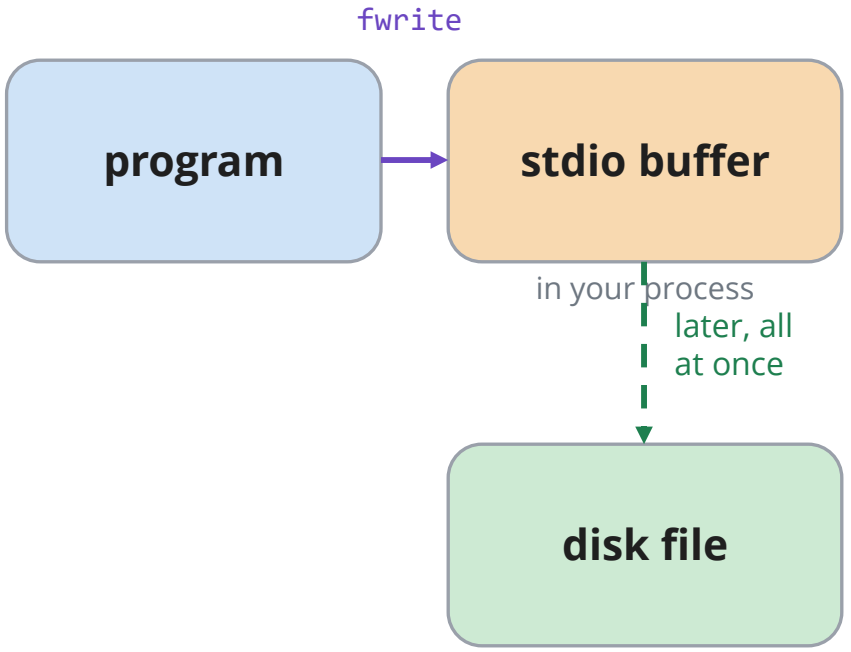
❖ **A file buffer plays the same role:** it sits between your program and the slow disk, so you are not waiting on the disk for every byte.

Same idea, two worlds:

the video player	↔	**your program**
the network (slow)	↔	**the disk (slow)**
the download buffer	↔	**the stdio buffer**

Your writes do not go straight to disk

- **fwrite copies into a buffer** that stdio keeps inside your process, not onto the disk.
- **Later, the buffer is drained** to the real file in one larger write.
- **So "the write returned"** does not mean "the bytes are on disk."



When does the buffer actually drain?

- You call **fflush(stream)** explicitly.
- The buffer fills up (often 1024 or 4096 bytes).
- Line-buffered to a console: a **'\n'** is written.
- You call **fclose(stream)**.
- The process exits gracefully (**exit()** or return from **main**).

Same program, two behaviors:

```
$ ./prog        # to a terminal
```

line buffered: you see each line as it prints.

```
$ ./prog | cat   # into a pipe
```

fully buffered: nothing appears until the buffer fills or the program exits. Same code, different device.

Buffered

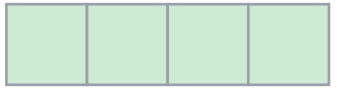
buffered (default)

```
1  FILE *fp = fopen("test.txt", "wb");
2  fwrite("h", 1, 1, fp);   // write 'h'
3  fwrite("i", 1, 1, fp);   // write 'i'
4  // fclose(fp);
```

after both fwrite calls:

stdio buffer

test.txt (disk)



still empty!

Both chars sit in the buffer. The disk file is empty until **fclose drains it. A crash here would lose both.**

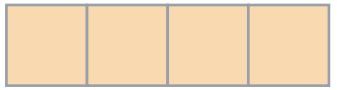
Unbuffered

unbuffered (setbuf NULL)

```
1  FILE *fp = fopen("test.txt", "wb");
2  setbuf(fp, NULL);          // buffering OFF
3  fwrite("h", 1, 1, fp);     // write 'h'
4  fwrite("i", 1, 1, fp);     // write 'i'
5  // fclose(fp);
```

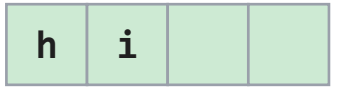
after both fwrite calls:

stdio buffer



stays empty

test.txt (disk)



Super slow!

setbuf(fp, NULL) turns buffering off, so each **fwrite** goes straight to the file. Safer, but one trip to disk per character.

Losing logs during a crash!

buggy.c

```
1  printf("hi");    // no newline!
2  int* p = NULL;
3  *p = 5;     // purposefully segfault here
```

the fix

```
1  printf("hi\n")    // newline flushes
2  fflush(stdout);   // or force it yourself
```

What you expect vs. what you get

"hi" went into stdout's buffer, not the screen. Then the segfault kills the process **before the buffer is flushed**, so the message never prints. It looks like the crash came earlier than it did.

Debugging lesson: do not trust the last printed line to mark where a program died. Add **\n** or **fflush**, or print progress to unbuffered **stderr**.

Why buffer, and why not

Why buffer

- **Performance.** Many tiny writes become one big write, so you touch the slow disk far less often.
- **Disk is slow.** A disk access is millions of times slower than a memory copy; batching hides that.
- **Convenience.** A nicer API than raw system calls (next lecture).

Why not

- **Reliability.** A crash or power loss drops whatever was still in the buffer.
- **Latency.** If you need a byte out now (a log, a prompt), buffering delays it.
- **Copy cost.** Copying into the buffer burns CPU and memory bandwidth; servers chase "zero-copy".

Reminders

Assessments:

- Exercise 3 due Wednesday at 11:59pm
- Homework 1 due Thursday at 11:59pm

General:

- Sign up for a coffee chat!
- Exams are scheduled
  - 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

Questions?

Thanks for coming - see you next lecture!