LECTURE 5 · CSE 333 · Summer 2026

Preprocessor and Linker

Discussion: Share your favorite Pokémon with your group. Create a new Pokémon by combining all of the favorites in your group.

Reminders

Assessments:

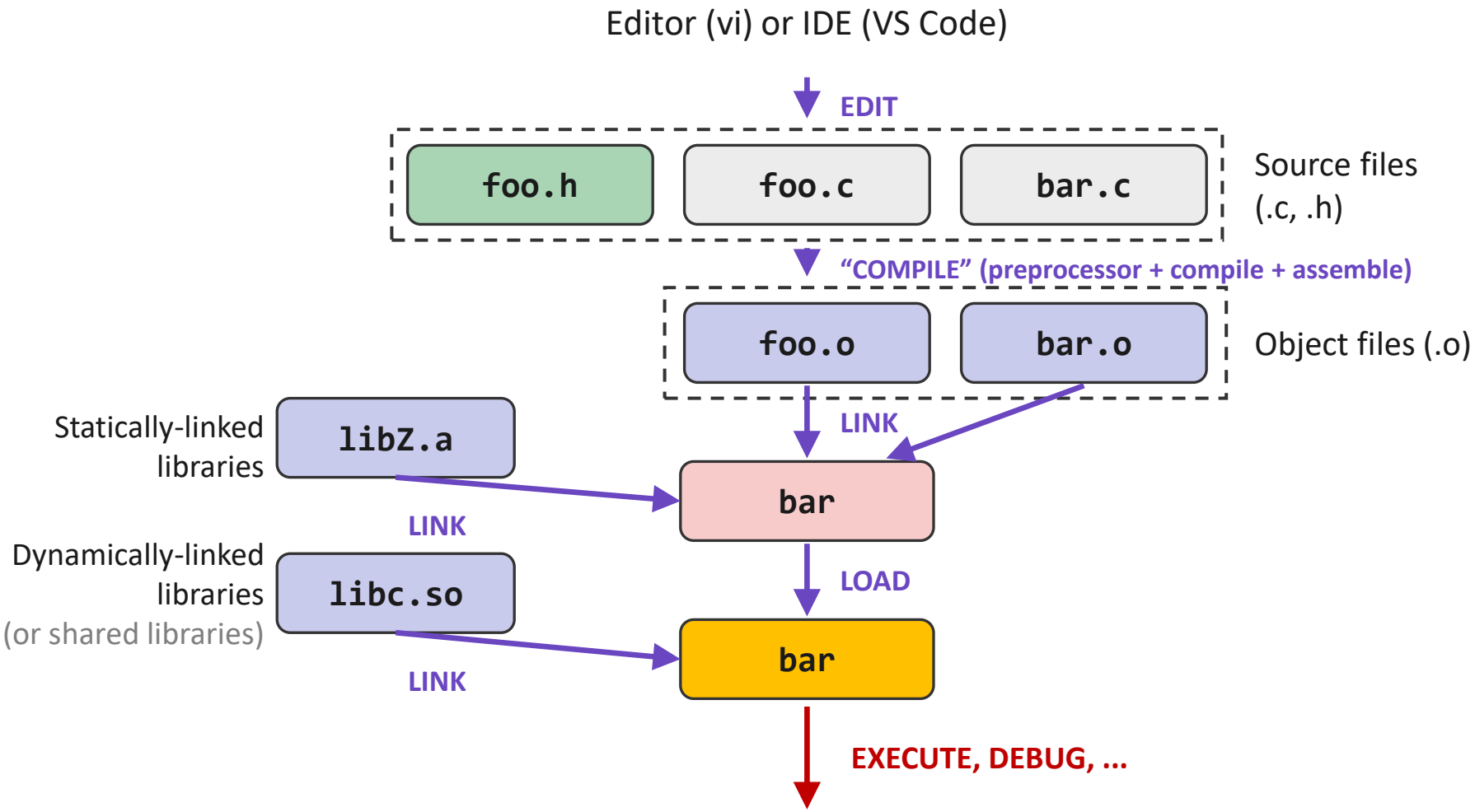
- Exercise 2 due tomorrow at 11:59pm
- Exercise 3 released on Friday. Due Wednesday at 11:59pm
- Homework 1 due next Thursday at 11:59pm
  - Much more challenging than the exercises!

General:

- Sign up for a coffee chat!
- No lecture on Friday – have a great long weekend!
- 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

Review

Compilation



REVIEW

Fixed widths: <stdint.h>

```c
#include <stdint.h>
#include <inttypes.h>  // PRI macros

uint8_t  b = 200;   // exactly 1 byte
int32_t  n = 42;    // exactly 4 bytes
uint64_t big = ...; // exactly 8 bytes

printf("%" PRId32 "\n", n);
printf("%u\n", b);  // promoted to int
```

- Names read as **[u]int<bits>_t**.
- Same size on every machine: files, networks, hardware.
- You meet **uint8_t** first in Exercise 2, not HW1.
- Print fixed widths with **PRId32 / PRIu64**.

Takeaway: stdint.h gives exact-size integers (uint8_t, int32_t, uint64_t). Use them when the byte count has to be guaranteed.

typedef

```
1  // typedef gives an existing type a new name
2  typedef uint32_t  Color;     // alias
3  typedef char*     String;    // alias
4
5  Color bg = 0xFF00FF;
6  String s = "hi";
7
8  // most useful for structs / fn pointers
9  typedef struct ll LinkedList;  // HW1
10 typedef void (*FreeFn)(void* p);
```

- Renames a type; no new bytes, same size.
- Hides ugly syntax
- HW1 uses **LinkedList** and payload typedefs.
- Read it like a declaration: last word is the new name.

REVIEW

void*

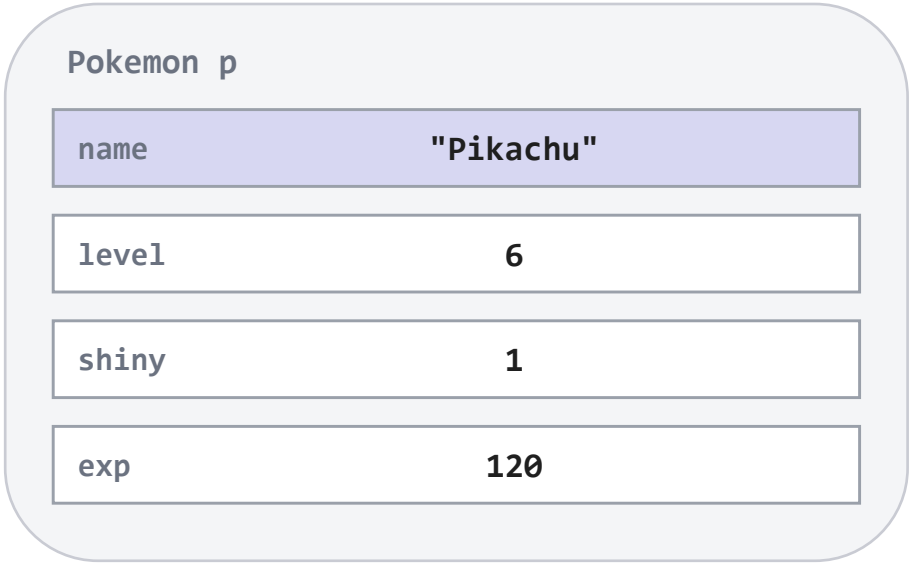
```
1  void Zero(void* p, size_t len) {
2    // *p = 0;     error: how many bytes?
3    // p[1]  = 0;  error: step size?
4  }
5
6  int  a[4];
7  char b[100];
8
9  Zero(a, sizeof(int));
   Zero(a, sizeof(char));
```

- void* can point at a or b, any type.
- But the function has no element type.
- It cannot deref or index: unknown size.
- Usually want to pass in a length or some type information

REVIEW

Structs

```
1  struct Pokemon {
2    char* name;    // -> read-only str
3    int   level;
4    char  shiny;   // 0 or 1
5    int   exp;
6  };
7
8  struct Pokemon p = {"Pikachu", 5, 1, 120};
9  p.level += 1;
```



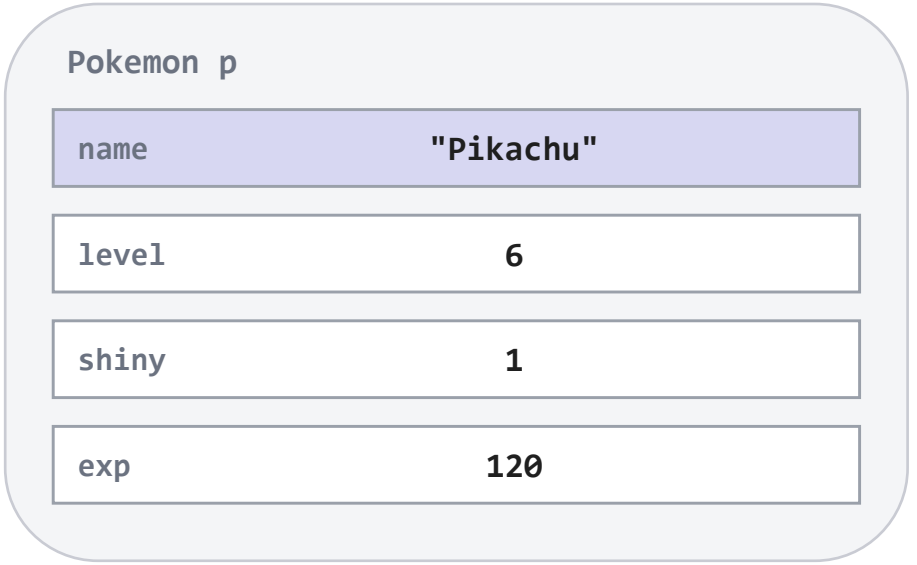
- One value bundles four fields.
- Reach each with a `.`

REVIEW

Structs + typedef

```c
typedef struct {
  char* name;    // -> read-only str
  int   level;
  char  shiny;   // 0 or 1
  int   exp;
} Pokemon;

Pokemon p = {"Pikachu", 5, 1, 120};
p.level += 1;
```



- One value bundles four fields.
- Reach each with a `.`

Use pointers to avoid copying structs

```
1  void LevelUp(Pokemon* mon) {
2    mon->level += 1;   // -> not .
3  }
4
5  Pokemon p = {"Pikachu", 5, 1, 120};
6  LevelUp(&p);
```

main: p

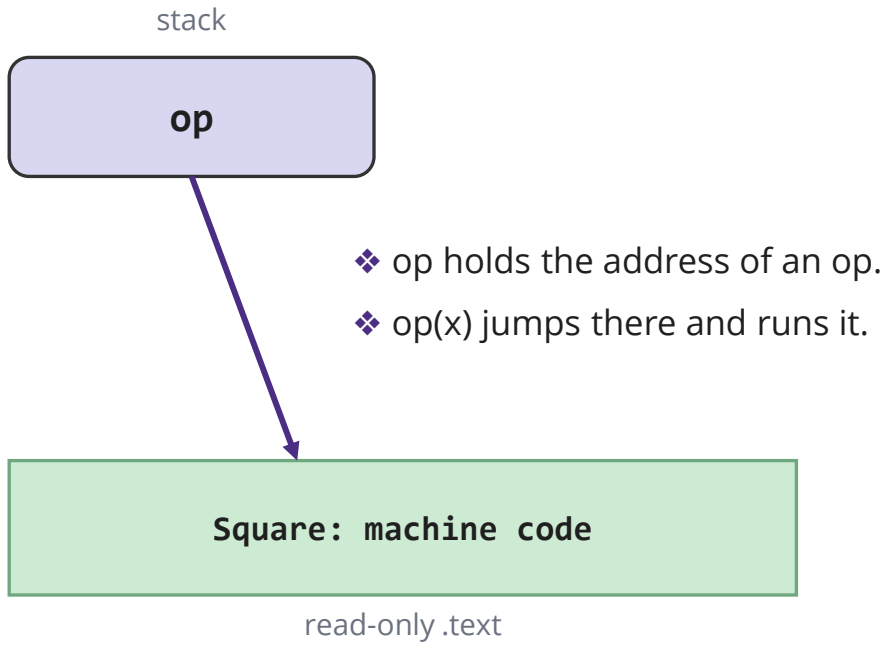
name	0x40..
level	5 -> 6
shiny	1
exp	120

LevelUp

mon	&p

Function Pointers

```
1  int Square(int n) { return n * n; }
2  int Negate(int n) { return -n; }
3
4  void Map(int a[], int len, int (*op)(int)) {
5    for (int i = 0; i < len; i++) {
6      a[i] = op(a[i]);
7    }
8  }
9
10 Map(arr, 4, Square);  // {1,2,3,4}->{1,4,9,16}
   Map(arr, 4, Negate);  // same loop, new op
```



- op holds the address of an op.
- op(x) jumps there and runs it.

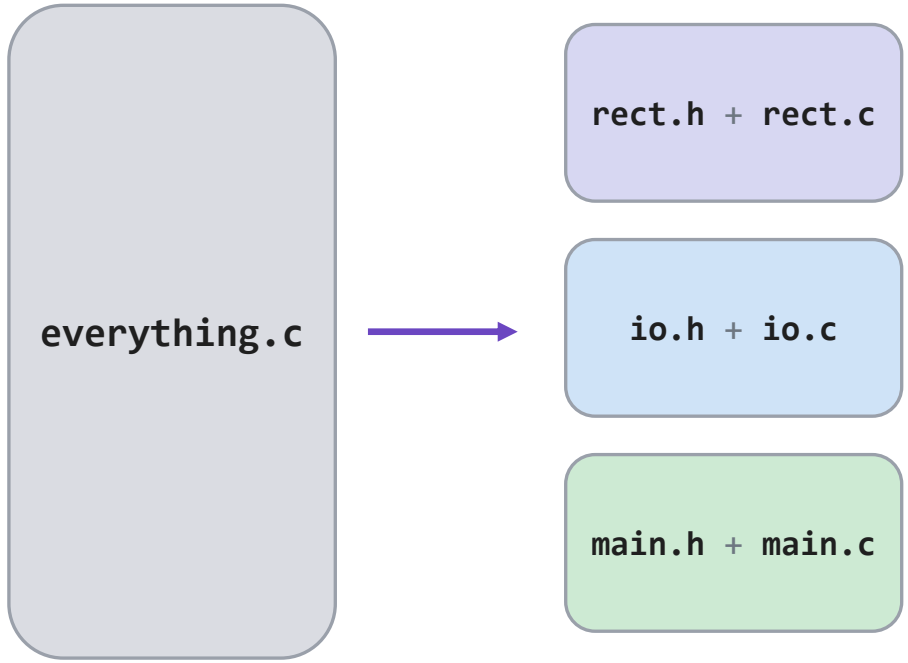
Function Pointers + typedef

```
1  int Square(int n) { return n * n; }
2  int Negate(int n) { return -n; }
3
4  typedef int (*IntFunction)(int* x);
5
6  void Map(int a[], int len, IntFunction op) {
7    for (int i = 0; i < len; i++) {
8      a[i] = op(a[i]);
9    }
10 }
11
12 Map(arr, 4, Square);  // {1,2,3,4}->{1,4,9,16}
13 Map(arr, 4, Negate);  // same loop, new op
```

- Can use typedef to create a named function pointer type
- Syntax is a little weirder than other typedef – so be careful!

Modules

Why split a program across files?

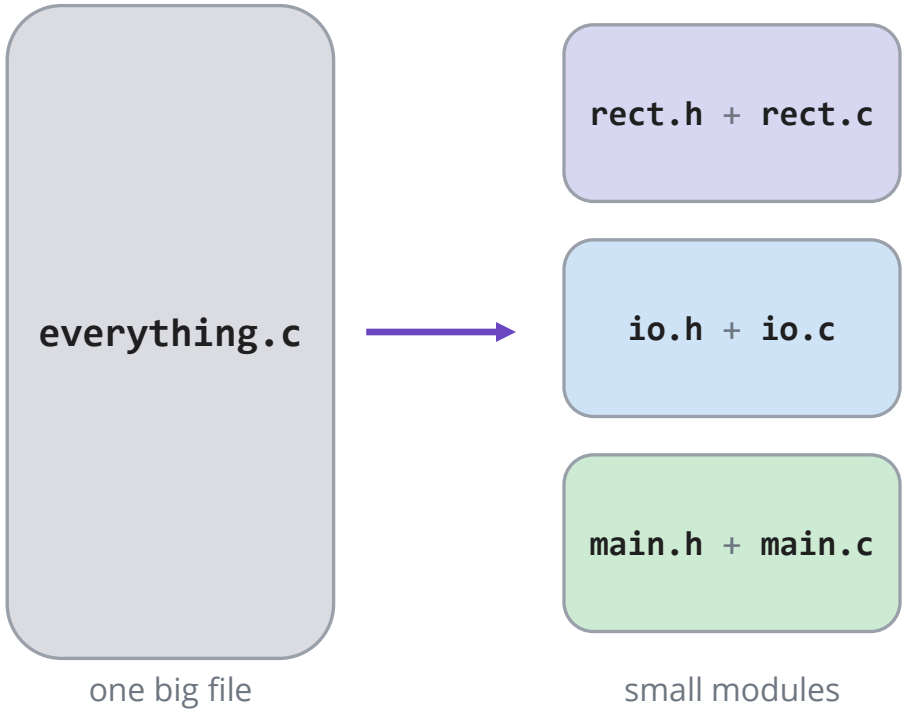


one big file

small modules

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.

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.

Example #1

rect.h (interface)

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

rect.c (implementation)

```
1  #include "rect.h"
2  int RectArea(int w, int h) {
3    return w * h;
4  }
5  int RectPerimeter(int w, int h) {
6    return 2 * (w + h);
7  }
```

main.c (a caller)

```
1  #include <stdio.h>
2  #include "rect.h"
3
4  int main(int argc, char **argv) {
5    int a = RectArea(3, 4);
6    printf("area = %d\n", a);
7    return 0;
8  }
```

main.c only #includes **rect.h**. It gets the two prototypes and can call **RectArea** without ever seeing how it is implemented.

The .c file includes its own header

rect.c

```c
#include "rect.h"

int RectArea(int w, int h) {
  return w * h;
}

int RectPerimeter(int w, int h) {
  return 2 * (w + h);
}
```

rect.h

```c
int RectArea(int w, int h);
int RectPerimeter(int w, int h);
```

- Line 1 pastes the prototypes into rect.c
- Now the compiler sees the **prototype** and the **definition** in the same file, and checks they agree.
- **Catch mismatches early.** Wrong return type or parameters becomes a compiler error (not a runtime error)

When the header and the .c disagree

rect.h (says one thing)

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

rect.c (does another)

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

compiler error

```
rect.c: error: conflicting types for 'RectArea'
rect.h: note: previous declaration was here
```

- The header promised **int RectArea(int, int)**.
- The .c defined **long RectArea(int)**: wrong return type, wrong number of parameters.
- Because rect.c included rect.h, both are visible and the compiler stops you.

Common mistake: don’t redefine a variable from a header

linker error

```
multiple definition of 'rare_candy'
main.o: first defined here
```

config.h (the trap)

```
1  int rare_candy = 100;
```

config.c (one definition)

```
1  #include "config.h"
2  int rare_candy = 100;   // the one definition
```

- Same variable defined in multiple places
- Also possible if you include the same header in multiple files that include each other.

Headers are where you document the interface

rect.h

```
1  // rect.h - rectangle math helpers
2
3  // Returns the area of a w-by-h rectangle.
4  // Requires w >= 0 and h >= 0.
5  int RectArea(int w, int h);
6
7  // Returns the perimeter of a w-by-h rectangle.
8  // Requires w >= 0 and h >= 0.
9  int RectPerimeter(int w, int h);
```

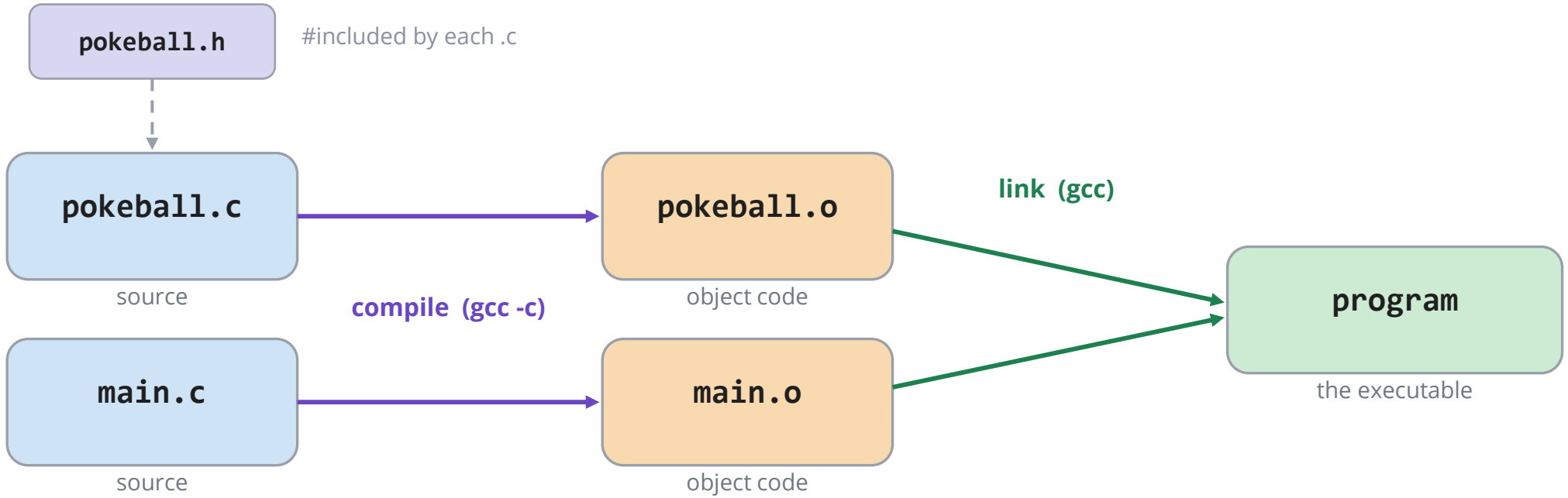
❖ **The header is the contract.** It is the one file callers read, so it is where the docs belong.

❖ For each function, say:

- what it does, in one line
- what each parameter means
- what it returns
- preconditions the caller must meet

❖ Comment the **interface** here, not the **implementation details**. Those are in rect.c

How the pieces become one program



The **linker** joins the .o files into one program, wiring each call in main.o to the real **RectArea** in rect.o.

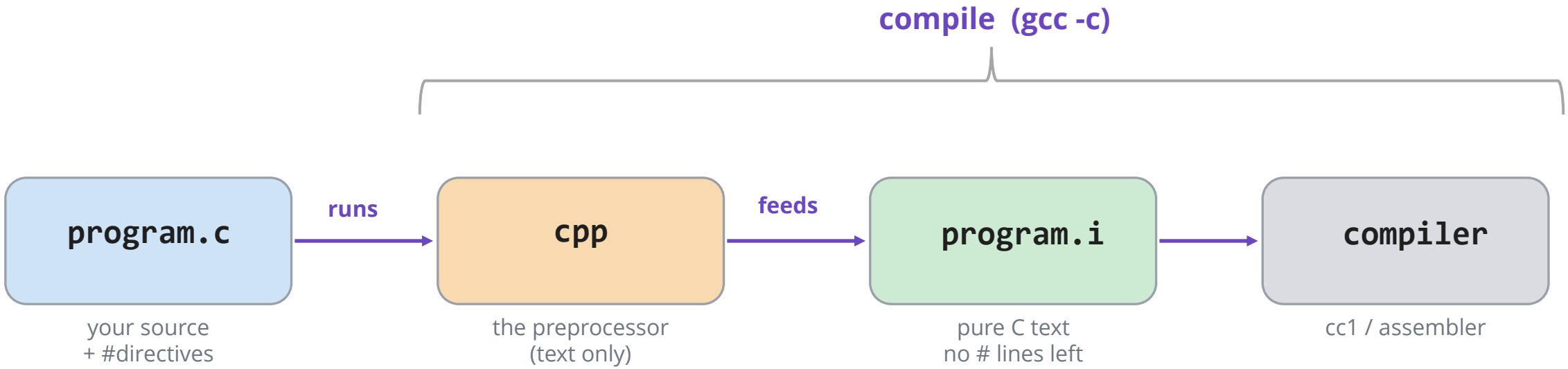
```
$ gcc -c pokeball.c main.c              # compile the source files
$ gcc pokeball.o main.o -o program      # link the object files
```

```
$ gcc pokeball.c main.c -o program      # compile + link in one step
```

Preprocessor

(first part of compile step)

Where it fits in the build



- The preprocessor only moves and edits **text**. It never checks types, scope, or C grammar.
- Try it yourself: **gcc -E program.c** prints the expanded source the compiler actually sees.

The rules

- **Pure text.** It edits characters line by line. It does not know C types, scope, or arithmetic.
- **Top to bottom.** A macro must be defined before it is used; substitution only happens after the **#define**.
- **No safety net.** The text substitution is very literal (like Ctrl + F, Ctrl + C, Ctrl + V)

#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.

#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.

Step 1 of 7

Macros are literal text substitution

demo.c

```
1  #define SIZE 4
2  #define SQUARE(x) x * x
3
4  int arr[SIZE];
5  int a = SQUARE(5);
6  int b = SQUARE(2 + 3);
```

Preprocessor state

Macro	Replacement text

What the compiler sees

```
1
2
3
```

Macros are literal text substitution

demo.c

```
1  #define SIZE 4
2  #define SQUARE(x) x * x
3
4  int arr[SIZE];
5  int a = SQUARE(5);
6  int b = SQUARE(2 + 3);
```

Preprocessor state

Macro	Replacement text
SIZE	4

What the compiler sees

```
1
2
3
```

Line 1 is a **#define**. **SIZE** → **4** enters the state; nothing is emitted yet.

Macros are literal text substitution

demo.c

```
1  #define SIZE 4
2  #define SQUARE(x) x * x
3
4  int arr[SIZE];
5  int a = SQUARE(5);
6  int b = SQUARE(2 + 3);
```

Preprocessor state

Macro	Replacement text
`SIZE`	`4`
`SQUARE(x)`	`x * x`

What the compiler sees

```
1
2
3
```

Line 2 defines a function-like macro. **SQUARE(x) → x * x** is stored as text.

Macros are literal text substitution

demo.c

```
1  #define SIZE 4
2  #define SQUARE(x) x * x
3
4  int arr[SIZE];
5  int a = SQUARE(5);
6  int b = SQUARE(2 + 3);
```

Preprocessor state

Macro	Replacement text
`SIZE`	`4`
`SQUARE(x)`	`x * x`

What the compiler sees

```
1  int arr[4];
2
3
```

int arr[SIZE]: **SIZE** is replaced by **4** → **int arr[4];**

Macros are literal text substitution

demo.c

```
1  #define SIZE 4
2  #define SQUARE(x) x * x
3
4  int arr[SIZE];
5  int a = SQUARE(5);
6  int b = SQUARE(2 + 3);
```

Preprocessor state

Macro	Replacement text
`SIZE`	`4`
`SQUARE(x)`	`x * x`

What the compiler sees

```
1  int arr[4];
2  int a = 5 * 5;
3
```

int a = SQUARE(5): copy the body with **x = 5** → **5 * 5 = 25. Looks fine.**

Macros are literal text substitution

demo.c

```
1  #define SIZE 4
2  #define SQUARE(x) x * x
3
4  int arr[SIZE];
5  int a = SQUARE(5);
6  int b = SQUARE(2 + 3);
```

Preprocessor state

Macro	Replacement text
`SIZE`	`4`
`SQUARE(x)`	`x * x`

What the compiler sees

```
1  int arr[4];
2  int a = 5 * 5;
3  int b = 2 + 3 * 2 + 3;
```

int b = SQUARE(2 + 3): the argument is pasted in as-is → **2 + 3 * 2 + 3**.

The fix: parenthesize everything

demo.c

```
1  #define SIZE 4
2  #define SQUARE(x) ((x) * (x))
3
4  int arr[SIZE];
5  int a = SQUARE(5);
6  int b = SQUARE(2 + 3);
```

Preprocessor state

Macro	Replacement text
`SIZE`	`4`
`SQUARE(x)`	`((x) * (x))`

What the compiler sees

```
1  int arr[4];
2  int a = ((5) * (5));
3  int b = ((2 + 3) * (2 + 3));
```

Redefine with parentheses: **((x) * (x))**. Now **SQUARE(2 + 3)** → **((2 + 3) * (2 + 3)) = 25.**

Unfortunately, math has rules! **2 + (3 * 2) + 3 = 11**, not 25

Since it is literal text substitution, it cannot assume any intent or help you catch mistakes.

Step 1 of 5

Tracking state: one include, two defines

dims.h

```
1  #define WIDTH 8
```

area.c

```
1  #include "dims.h"
2  #define AREA WIDTH * WIDTH
3  int a = AREA;
```

Preprocessor state

Macro	Replacement text

What the compiler sees

```
1
```

The two files, before we start. We will process **area.c** top to bottom and fill the state as we go.

Step 2 of 5

Tracking state: one include, two defines

dims.h

```
1  #define WIDTH 8
```

area.c

```
1  #include "dims.h"
2  #define AREA WIDTH * WIDTH
3  int a = AREA;
```

Preprocessor state

Macro	Replacement text

What the compiler sees

```
1
```

area.c begins with **#include "dims.h"**. First, just the directive: it will paste the header here.

Step 3 of 5

Tracking state: one include, two defines

dims.h

```
1  #define WIDTH 8
```

area.c

```
1  #include "dims.h"
2  #define AREA WIDTH * WIDTH
3  int a = AREA;
```

Preprocessor state

Macro	Replacement text
WIDTH	8

What the compiler sees

```
1
```

Paste dims.h. Its first line is a **#define** that adds **WIDTH → 8** to the state.

Tracking state: one include, two defines

dims.h

```
1  #define WIDTH 8
```

area.c

```
1  #include "dims.h"
2  #define AREA WIDTH * WIDTH
3  int a = AREA;
```

Preprocessor state

Macro	Replacement text
WIDTH	8
AREA	WIDTH * WIDTH

What the compiler sees

```
1
```

AREA is stored as the literal text **WIDTH * WIDTH**. WIDTH is not substituted yet.

Tracking state: one include, two defines

dims.h

```
1  #define WIDTH 8
```

area.c

```
1  #include "dims.h"
2  #define AREA WIDTH * WIDTH
3  int a = AREA;
```

Preprocessor state

Macro	Replacement text
WIDTH	8
AREA	WIDTH * WIDTH

What the compiler sees

```
1  int a = 8 * 8;
```

Using **AREA** expands, then rescans: **AREA → WIDTH * WIDTH → 8 * 8**.

Trickier: chained includes across files

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

What the compiler sees

```
1
```

The three files, before we start. **main.c** is what we compile; it will pull the others in.

Trickier: chained includes across files

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

What the compiler sees

```
1
```

main.c's first line is **#include "util.h"**. Note the directive.

Trickier: chained includes across files

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

What the compiler sees

```
1
```

Paste util.h. Its first line is itself an **#include** of config.h, so we follow that too.

Trickier: chained includes across files

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

What the compiler sees

```
1
```

config.h's first line: **#define MAX → 100**. It enters the one shared state.

Trickier: chained includes across files

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)

What the compiler sees

```
1
```

Back in util.h: **LIMIT** is stored as the text **(MAX - 1)**. MAX is not substituted yet.

Trickier: chained includes across files

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;
```

Back in main.c: **#define SCALE → 2**.

Preprocessor state

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

What the compiler sees

```
1
```

Trickier: chained includes across files

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.

Step 1 of 4

Super tricky (try ay home)

clamp.c

```
1  #define MAX(a, b) ((a) > (b) ? (a) : (b))
2
3  int i = 3, j = 4;
4  int m = MAX(i++, j++);
```

Preprocessor state

Macro	Replacement text

What the compiler sees

```
1
2
```

The whole file, before we start. **MAX** looks like a safe, fully parenthesized macro.

Remember ++var increments var and evaluates to the new value after incrementing.
Remember var++ increments var and evaluates to the old value before incrementing.

Super tricky (try ay home)

clamp.c

```
1  #define MAX(a, b) ((a) > (b) ? (a) : (b))
2
3  int i = 3, j = 4;
4  int m = MAX(i++, j++);
```

Preprocessor state

Macro	Replacement text
`MAX(a, b)`	`((a) > (b) ? (a) : (b))`

What the compiler sees

```
1
2
```

Define **MAX(a, b)**. Its body is stored as text, parentheses and all.

Super tricky (try ay home)

clamp.c

```
1  #define MAX(a, b) ((a) > (b) ? (a) : (b))
2
3  int i = 3, j = 4;
4  int m = MAX(i++, j++);
```

Preprocessor state

Macro	Replacement text
`MAX(a, b)`	`((a) > (b) ? (a) : (b))`

What the compiler sees

```
1  int i = 3, j = 4;
2
```

int i = 3, j = 4; has no macros, so it passes straight through to the output unchanged.

Step 4 of 4

Super tricky (try ay home)

clamp.c

```
1  #define MAX(a, b) ((a) > (b) ? (a) : (b))
2
3  int i = 3, j = 4;
4  int m = MAX(i++, j++);
```

Preprocessor state

Macro	Replacement text
`MAX(a, b)`	`((a) > (b) ? (a) : (b))`

What the compiler sees

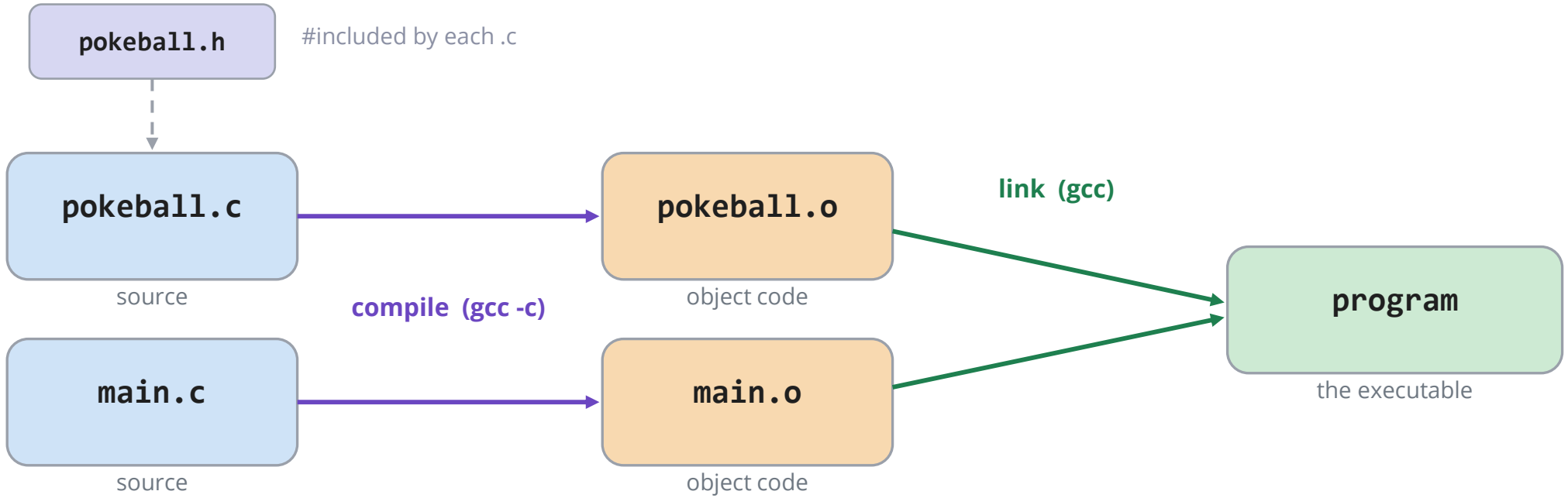
```
1  int i = 3, j = 4;
2  int m = ((i++) > (j++) ? (i++) : (j++));
```

MAX(i++, j++) expands to **((i++) > (j++) ? (i++) : (j++))**.

The winner is evaluated twice: j++ runs in the test AND in the branch, so j jumps to 6 and m gets a stale value. This is undefined behavior.

Linker

How the pieces become one program



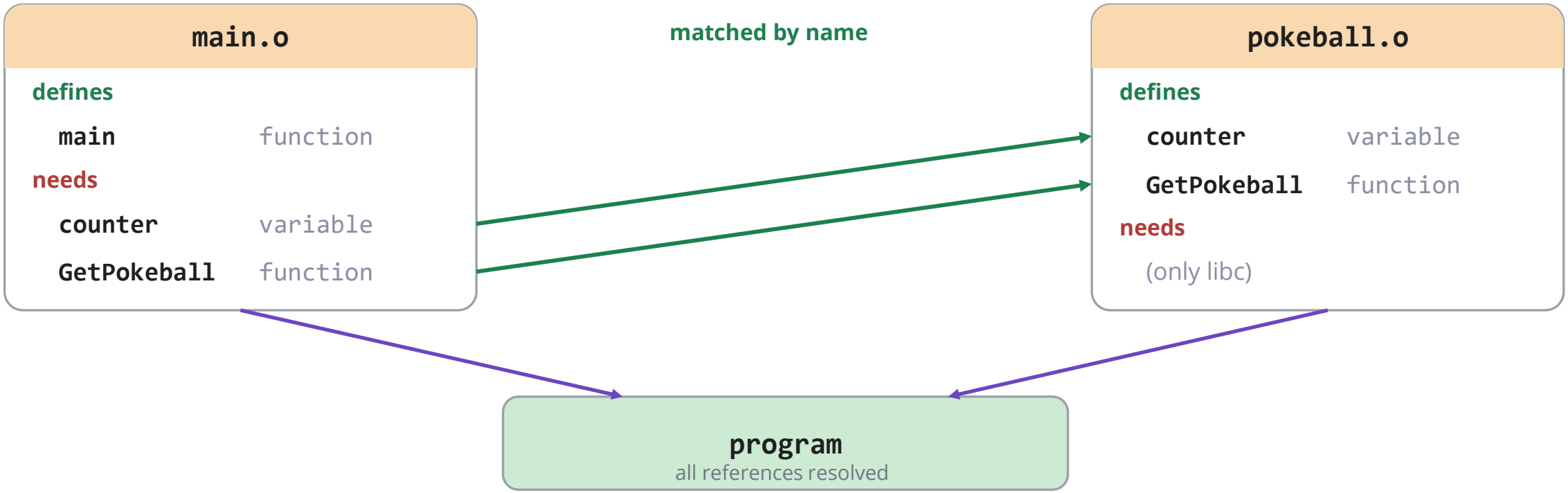
The **linker** joins the .o files into one program, wiring each call in main.o to the real **RectArea** in rect.o.

```
$ gcc -c pokeball.c main.c              # compile the source files
$ gcc pokeball.o main.o -o program      # link the object files
```

```
$ gcc pokeball.c main.c -o program      # compile + link in one step
```

What the linker does

Each **.o** defines some symbols and leaves others as **unresolved references**. The linker matches every reference to a definition **by name**.



The namespace problem

You write a global **counter** in **foo.c** and another global **counter** in **bar.c**. Are they the **same** variable, or **two** different ones?

Same variable (external linkage)

- The name is shared across files.
- Defined **once**, used everywhere else.
- Both files touch **one copy** in memory.

Two variables (internal linkage)

- The name is private to each file.
- Defined **separately** in each file.
- **Two independent copies** in memory that never interfere.

We call this **linkage**. Every global name has one, and you choose it with **extern** or **static**.

external linkage vs. internal linkage

At global scope, **external** means the linker can see the name from other files; **internal** means it cannot.

Written at global scope	Linkage	Who can reach it?
`int counter = 1;`	**external**	shared with every file
`static int counter = 1;`	**internal**	private to this file
`int Scale(int x) { ... }`	**external**	callable from any file
`static int Scale(int x) {}`	**internal**	callable in this file only
`extern int counter;`	**external**	a reference to the one definition

Globals are **external** by default. **static** flips a definition to internal; **extern** makes a declaration that points at a definition elsewhere.

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**.

extern vs. static in global variables

Access is controlled only by the **declarations** of these global variables.

extern → shared

foo.c

```
1  int counter = 1;
```

bar.c

```
1  extern int counter;
```

output

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

static → separate

foo.c

```
1  static int counter = 1;
```

bar.c

```
1  static int counter = 100;
```

output

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

Functions have linkage too

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.

Collisions: two definitions of one name

foo.c

```c
#include <stdio.h>
int pokemon_level = 1;    // external def
int main(void) { return EXIT_SUCCESS; }
```

bar.c

```c
int pokemon_level = 100;  // also external!
void Bar(void) { counter++; }
```

linker error

```
multiple definition of `pokemon_level'
bar.o: first defined here
```

- Every global is **extern** by default. Two files defining **int counter** both claim the name.
- **Best case:** the linker refuses with multiple definition.
- **Worst case:** undefined behavior!
- **Defensive programming.** Mark globals that can be private as **static** so their names cannot collide.
- **Share on purpose.** Define once in a .c, put an **extern** declaration in a header, and include it.

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.

Reminders

Assessments:

- Exercise 2 due tomorrow at 11:59pm
- Exercise 3 released on Friday. Due Wednesday at 11:59pm
- Homework 1 due next Thursday at 11:59pm
  - Much more challenging than the exercises!

General:

- Sign up for a coffee chat!
- No lecture on Friday – have a great long weekend!
- 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!