CS 241: Systems Programming Lecture 10. Structure of C Programs

Spring 2020

Prof. Stephen Checkoway

Structure of programs

Split code between header files and source files

Header files (extension: .h) contain

- Function prototypes for global functions, e.g.,
  ```
  void foo(int param);
  size_t get_size(int a, int b, float c);
  ```
- Global variable declarations, e.g.,
  ```
  extern bool some_global_variable;
  ```
- Type definitions (we'll see these later)

Structure of programs

Source files (extension: .c) contain

- Function definitions, e.g.,

```
void foo(int param) {
  printf("foo was called with %d\n", param);
}
```

- Global variable definitions (no `extern`)

```
bool some_global_variable;
```

Splitting your program up

Group related functions in the same source file, e.g., `logger.c`

Provide a corresponding header file, e.g., `logger.h` which declares the global functions (and types and global variables) defined in `logger.c`

Each source file should include the headers for every function used in the file, including the ones defined in the file itself

A source file just containing the `main` function doesn't need a header file

logger.h

```c
// A simple logging implementation.

#ifndef LOGGER_H
#define LOGGER_H

#define LOG_LEVEL_INFO 0
#define LOG_LEVEL_WARNING 1
#define LOG_LEVEL_ERROR 2

// Set the minimum log level to be displayed.
void set_minimum_log_level(int level);

// Log a message at the given level.
void log_message(int level, char const *msg);

#endif
```

logger.c 1/2

```c
#include "logger.h"

#include <stdio.h>

static int min_level = LOG_LEVEL_WARNING;

static char const *get_level_string(int level) {
  switch (level) {
  case LOG_LEVEL_INFO:
    return "INFO";
  case LOG_LEVEL_WARNING:
    return "WARNING";
  case LOG_LEVEL_ERROR:
    return "ERROR";
  default:
    return "UNKNOWN";
  }
}
```

logger.c 2/2

```c
// Set the minimum log level to be displayed.
void set_minimum_log_level(int level) {
  min_level = level;
}

// Log a message at the given level.
void log_message(int level, char const *msg) {
  if (level >= min_level)
    fprintf(stderr, "[%s]: %s\n", get_level_string(level), msg);
}
```

main.c 1/1

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

#include "logger.h"

static void set_log_level(char const *name) {
  if (strcmp(name, "info") == 0)
    set_minimum_log_level(LOG_LEVEL_INFO);
  else if (strcmp(name, "warning") == 0)
    set_minimum_log_level(LOG_LEVEL_WARNING);
  else if (strcmp(name, "error") == 0)
    set_minimum_log_level(LOG_LEVEL_ERROR);
  else
    fprintf(stderr, "Unknown log level: %s\n", name);
}

int main(int argc, char **argv) {
  if (argc == 2)
    set_log_level(argv[1]);

  log_message(LOG_LEVEL_INFO, "An info message");
  log_message(LOG_LEVEL_WARNING, "A warning message");
  log_message(LOG_LEVEL_ERROR, "An error message");
  return 0;
}
```

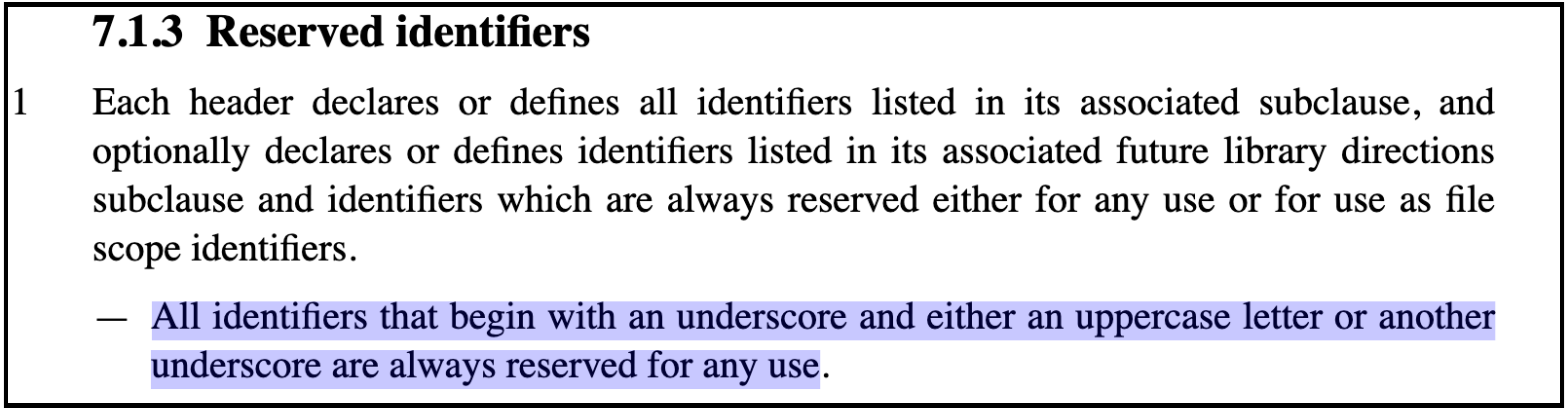
Header tips

Ensure that the order you include headers doesn't matter

- Headers should be self-contained or `#include` any needed headers

Use a header guard based on the file path

- for `foo/bar.h`, use `FOO_BAR_H`
- Do **not** use `_BLAH_HEADER_FILE_`



7.1.3 Reserved identifiers

1 Each header declares or defines all identifiers listed in its associated subclause, and optionally declares or defines identifiers listed in its associated future library directions subclause and identifiers which are always reserved either for any use or for use as file scope identifiers.

— All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.

Header include order

Group headers in the following order
(I like alphabetical in each group)

- Related header (if applicable)
- System library headers
- Other library headers
- Other headers in your code

Add a blank line between groups

Be consistent with existing code!

```c
// Inside foo.c
#include "foo.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <png.h>

#include "bar.h"
#include "qux.h"
```

Compiling

All at once

-
  ```
  $ clang -std=c11 -Wall -o program *.c
  ```

One file at a time with separate linking step

-
  ```
  $ clang -std=c11 -Wall -c -o foo.o foo.c
  $ clang -std=c11 -Wall -c -o bar.o bar.c
  $ clang -std=c11 -Wall -c -o qux.o qux.c
  $ clang -o program foo.o bar.o qux.o
  ```

printf(3)

```
int printf(char const *format, ...);
```

- Takes a format string and a variable number of parameters
- Conversion specifiers control how the additional parameters are printed

Specifier	Type	Prints	Specifier	Type	Prints
`%c`	`int`	character	`%e, %E`	`double`	[-]d.ddde±dd
`%d, %i`	`int`	decimal	`%f, %F`	`double`	[-]ddd.ddd
`%u`	`unsigned int`	decimal	`%g, %G`	`double`	like %e or %f
`%x, %X`	`unsigned int`	hexadecimal	`%a, %A`	`double`	[-]0xh.hhhhp±dd
`%o`	`unsigned int`	octal	~~`%n`~~	~~`int *`~~	~~don't use~~
`%s`	`char const *`	string	`%%`		literal %
`%p`	`void *`	0x hexadecimal			

printf(3) length modifiers

Controls the size of the integer conversion: `%d`, `%i`, `%o`, `%u`, `%x`, or `%X`

- Goes just before the `d`, `i`, `o`, `u`, `x`, or `X`
- `d` and `i` are signed, `o`, `u`, `x`, and `X` are unsigned

Modifier	Modified types (signed)	(unsigned)	Example
`hh`	`signed char`	`unsigned char`	`%hhd`
`h`	`short int`	`unsigned short int`	`%hx`
`l`	`long int`	`unsigned long int`	`%lu`
`ll`	`long long int`	`unsigned long long int`	`%lld`
`z`	`ssize_t`	`size_t`	`%zu`
`j`	`intmax_t`	`uintmax_t`	`%jd`
`t`	`ptrdiff_t`		`%td`

Type promotion for variadic functions

Variadic functions take a variable number of arguments (like `printf`)

```
int printf(char const *format, ...);
```

The variable argument portion (the `...`) doesn't (can't) specify types so promotions occur on arguments

- `bool` → `int`
- `char` → `int`
- `short` → `int`
- `float` → `double`

We can use %hhd and %hd to print `char` and `short` if we want

`%e`, `%f`, and `%g` just print `double`s

`%d` prints a `signed int` in decimal
`%u` prints an `unsigned int` in decimal

Which format string should we use to print the three variables x, y, and z?

```
short x = 10;
int y = -356;
unsigned int z = 85246;
printf(fmt, x, y, z);
```

A. `"%d %d %u"`

B. `"%u %d %u"`

C. `"%hd %d %u"`

D. Either A or C

E. Any of A, B, or C

`%d` prints a `signed int` in decimal
`%f` prints a `double` as a floating point number (with 6 digits after the .)

What does this print?

```
float x = 10.5f; // The f suffix means float
printf("%d %f\n", x, x);
```

A. 10 10.500000

B. 11 10.500000

C. -907309384 10.500000

D. Nothing, it's a run time error

E. It's undefined behavior so it could print anything

printf(3) additional stuff

A conversion specifier has the form

- Start of conversion specifier: `%`
- Zero or more flags: `#`, `0`, `-`, `' '`, and `+`
- An optional minimum field width: e.g., 3
- An optional precision: e.g., 2
- An optional length modifier: e.g., `ll`
- The specifier: e.g., `d`

Examples

- %#llx — print unsigned long long in hex with a leading 0x
- %4.3e — Floating point with a minimum width of 4 and precision of 3

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-10.html

Grab a laptop and a partner and try to get as much of that done as you can!