ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 17

Vince Weaver

`https://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

9 March 2026

Announcements

- No, the midterm isn't graded yet.
  It's unlikely it will be graded before spring break.
- Still working on HW#5
- No Lab#6 this week! Monday Lab is Lab#5. Otherwise it will be makeup lab, maybe something fun Thursday lab. Ideally everyone will be fully caught up on labs before spring break.
- Looking ahead Lab#6 will be keypad lab, it's likely there will be 9 labs total

C knowledge

- One nice thing about C is it's a relatively small language
- By this time you know most of the important C concepts
- The rest of the class will be exploring details, special cases, and more obscure features

Preprocessor

- C has a special macro processor called the "preprocessor"
- It is run on your code before the compiler gets to it, a special pass
- It can be run as a separate program, with gcc it's `cpp`
- Can run it via `gcc -E`

Preprocessor Features

- Include files
- Conditional compilation
- Macro expansion

Including Files

- We mentioned this already when we discussed `hello_world.c`
- You can include the contents of other files into your own
- The pre-processor does this before the compiler runs
- Can be system wide or local.
- You can write your own header files
- In C the extension is tt .h for header files

Header Search Path

- Files included with double quotes are in your local source directory
- Files in angle brackets are system includes, shared by everyone on system
  On Linux this defaults to `/usr/include`
- Can use `-I` compiler flag to expand directories to search
- Example: open `/usr/include/stdio.h` and look at it

```
#include <stdio.h>
#include "hw5.h"
```

Header Files – What is in them?

- Useful things to let you use code in other files
- We'll talk later about how to do this yourself
- For now we use them to interact with external "libraries" of code provided by the system
- For example, we include `stdio.h` otherwise C doesn't know how to use things like `printf()`

Function Declarations

- Remember in C you can't use a function unless you've seen it defined before
- One way is to just have it in your code before you see it
- That's not always possible, so you can pre-declare functions.

```c
#include <stdio.h>

int main(int argc, char **argv) {

    printf("sum is %d\n",sum(4,5));

    return 0;
}

int sum(int x, int y) {
    return x+y;
}
```

- Above will give error because main() uses sum() before seeing it (remember C parses file linearly starting from top of file)
- You can "predeclare" sum with something like:

```
int sum(int x, int y);
```

- Note: your pre-declaration needs to match the original declaration

Header Files – What is not in them?

- In C it is considered poor form to include code in header files
  (note, for various reasons C++ does this)

#defines

- Can use to define things like constants
- Usually use ALL CAPS to make it more obvious it's a `#define` and not a normal variable
- The pre-processor does a global search-and-replace substitution

```
#define PI 3.14159
#define MESSAGE "Hello"
#define LUGGAGE_CODE 0x12345

x=PI*2;                    // x ends up with pi*2
printf("%s\n",MESSAGE);    // prints Hello
```

Avoiding "Magic Constants"

- It can be easy, especially with low-level coding, to end up having to set complex bit patterns and put them in your code
- Many projects (like Linux kernel) prefer you use `#define` to give them more meaningful names to make the code easier to read

```
#define BARGRAPH_SEGMENT0 (1<<13)

// which is more readable?

sio_hw->gpio|=(1<<13);
```

```
sio_hw->gpio|=BARGRAPH_SEGMENT0;
```

#define – why not use constants?

- Instead of `#define PI 3.14159`
- Why not use `const double pi=3.14159`?
- A few reasons
  - Takes up space in memory (not a good reason)
  - Usually (?) you can't declare array sizes with const

```
#define ARRSIZE 1024
const int arrsize=1024;

int a[ARRSIZE]; // always works
int b[arrsize]; // this won't work
```

#define – cautions

- It's doing a global find and replace
- This might not do things you expect
- Might want to use parenthesis

```
#define EIGHT 4+4

x=EIGHT/2;

//after substitution

x=4+4/2; // you'll get 6
```

defines – advanced

- Will not find inside of a text string
- Not recognized prior to definition
- Needs to have white space around it
  so if NO defined, will not replace in SNOW
- Can have nested expansion

defines – built into compiler

- `__STDC__`
- `__TIME__`
- `__DATE__`
- `__FILE__` – current file in
- `__LINE__` – current line number
- And many others

Un-defining

- You can undefine a value

```
#define X 4

#undef X
```

- Why would you do that?

Preprocessor Parameterized Macros

```
#define macro_name(param1, ... paramN) token_sequence
```

- Question: Why the extra parenthesis?

```
#define SQR(a) ((a)*(a))
#define SUM(a,b) ((a)+(b))
```

- Note macros created this way are sort of type agnostic (can be used by both `int` and `float` for example)

Stringizing Operator

```
// extra #, puts double quotes around parameter
#define PRINT(s) printf("%s\n",#s)
```

Token Pasting Operator

```
// extra #, puts double quotes around parameter
#define PTR(s) ptr##s

if you call PTR(1) will generate ptr1
(it merged the tokens)
```

Conditional Execution

- What if you want to have some code enabled at *compile time* only if some condition is true

```
#define DEBUG 1
#if (DEBUG==1)
   printf("Extra debug message\n");
#endif
```

- You can see here you can have debug messages all over the code, and then disable them by just changing the define at the start

Conditional Execution Variants

- `#if`,`#ifdef`, `#else`,`#elif`,`#endif`

```
#if (DEBUG==1)
        printf("Debug Hi\n");
#else
#if (X==1)
        printf("No Debug Hi\n");
#endif
#endif

// can instead compact down to
#if (DEBUG==1)
        printf("Debug Hi\n");
#elif (X==1)
        printf("No Debug Hi\n");
```

```
#endif
```

- Also `#ifdef`,`#ifndef`, `#elifdef`,`#elifndef`,

Using #if to comment out code

- A quick way to comment out large blocks of code is using `#if 0`
- This is useful as `/* */` might have trouble if you try to use it on areas also containing comments

```
#if 0
    printf("Hello\n");
#endif
```

Conditional Execution Variables Advanced

- Has to be determined at compile time, can be DEFINE or CONSTANTS but not a variable
- Can use `M<<,>>,*,|,^,~,!,&&,||`

Setting Defines from Command Line

- Can set a define at command line
- `gcc -DDEBUG=1 ...`
- This way can use same code but compile into different executables (for example, debug and non-debug version)

Pragmas

- Some C compilers let you set various settings in the compiler with a special `#pragma` operator
- This is weird because it's not a pre-processor thing
- The gcc people didn't like this so unlike other compilers doesn't use it much
- Some have snuck in through various standards
- For example, OpenMP uses pragmas extensively

Other directives

- `#warning` – prints warning when compiling
- `#error` – terminates execution with error message

```
#ifndef SOMETHING_IMPORTANT
#error "You forgot to define SOMETHING_IMPORTANT"
#endif
```

Avoiding double-including header files

- Why is it bad?
- How can you avoid it?
- Could you wrap the whole header in:

```
#ifndef STDIN_H_ALREADY
#define STDIN_H_ALREADY 1
....
#endif
```

- New method: have `#pragma once` at start of file

Why use functions over macros?

- In old days macros can be evaluated at compile time, especially if mostly constants.
- These days you can use "inline functions" instead
- Macros are evaluated each time, and if long can take up more room. Functions are in one place and jumped to

assert

- discuss this next time