Objectives – Fri 2/14/2020

- Build steps
  - preprocessor
  - compiler
  - linker
- Preprocessor
  - `#define`
    - `#define` ▒▒▒ ▒▒▒
    - `#define` ▒▒▒(▒) ▒▒▒
    - `#define` ▒▒▒
  - `#include`
  - `#ifdef`

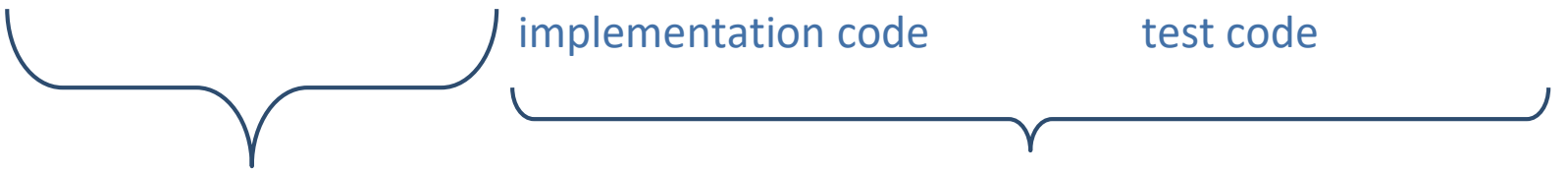
```
#ifdef ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
#else
#ifdef ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
#endif
```

Note: I very rarely use PowerPoint slides in lecture. Most of these were handwritten in class. I digitized them after the lecture.

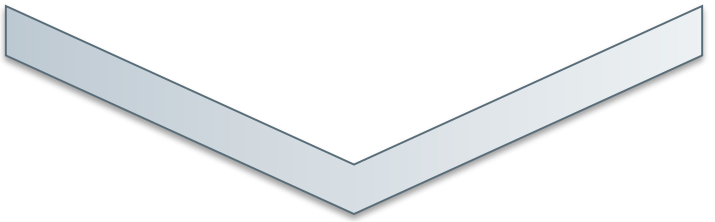
Snippets from today's lecture can be found on the Schedule page of the course web site.

Building a program entails three stages.

```
gcc -o test_mintf mintf.c test_mintf.c
```



executable source files



1. Preprocessor
2. Compiler
3. Linker

Each stage converts your code to another form.

1. Preprocessor
   - Remove comments
   - Process preprocessor directives (#▒▒▒▒▒▒▒▒)
2. Compiler
   - Convert each C function to machine instructions
   - ▒▒▒▒▒.c ⇒ ▒▒▒▒▒.o ⋯ one file at a time
   - `int main(…) { … }` ➔ 01001010111001101011101
3. Linker
   - Create a single executable from any number of .o files
   - Connect function calls to function code

Compiled functions are in separate object files.

test_mintf.c

```
int main() {
  ▒▒▒▒▒▒▒▒▒▒▒▒
  ▒▒▒▒▒▒▒▒▒▒▒▒
}
```

mintf.c

```
void mintf(…){
  ▒▒▒▒▒▒▒▒▒▒▒▒
  ▒▒▒▒▒▒▒▒▒▒▒▒
}

void print_integer(…) {
  ▒▒▒▒▒▒▒▒▒▒▒▒
  ▒▒▒▒▒▒▒▒▒▒▒▒
}
```

test_mintf_.o

```
<main>
  1110110101010110
  0011101101001101
  0010110110011000
  1000101111000110
```

mintf.o

```
<mintf>
  0011111011101000
  0111111001001011
  1000101111000110
  0001001011101001

<print_integer>
  0110000110111011
  1110110110110110
  0001001011101001
  0000100110000011
```

libc.so

```
…
<fputc>
  0011111001101000
  0111111001001011
  0001001011101001

<printf>
  0110000110111011
  0001001011101001
  0000100110000011
…
```

Linker joins compiled functions into an executable.

test_mintf_.o

```
<main>
  1110110101010110
  0011101101001101
  0010110110011000
  1000101111000110
```

mintf.o

```
<mintf>
  0011111011101000
  0111111001001011
  1000101111000110
  0001001011101001

<print_integer>
  0110000110111011
  1110110110110110
  0001001011101001
  0000100110000011
```

libc.so

```
…
<fputc>
  0011111001101000
  0111111001001011
  0001001011101001

<printf>
  0110000110111011
  0001001011101001
  0000100110000011
…
```

test_mintf

```
01100001101110111011011110110110 1
10000100101110100100001001100000 1101
10000110111011101101111011011011 0110
00010010111010010010111010010000 1001
10000011011000011011101110110111 1011
01101101100001001011101001000010 0110
00001101100001101110111011011110 1101
10110110000100101110100100001000 0001
00110000011011000011011101110110 1111
```

Build stages can be performed separately.

1. Preprocessor

   `/usr/bin/cpp ▒.c ⇒ ▒.c`

2. Compiler

   `gcc -c ▒.c ⇒ ▒.o`

3. Linker

   `ld -o ▒▒▒▒ ▒.o ▒.o ▒.o`

Understand preprocessor effects with /usr/bin/**cpp**.

```
/usr/bin/cpp test_count_words.c
```

 # … to see preprocessor will transform your code.

```
/usr/bin/cpp test_count_words.c -DDEBUG
```

 # … to define a symbol, just as you would with GCC.

```
/usr/bin/cpp test_count_words.c | indent
```

 # … to indent the result for better legibility.

#define ▒▒▒▒▒▒▒▒(…) macros are used like functions.

```
#define squared(x) ((x) * (x))
```

```
squared(4)  ⇒   16
```

⚠ #define does not reduce argument to a value.

`#define` macros use simple text substitution.

```
#define triple(x)  x * 3          BAD!!!

triple(2 + 5)    ⇒    2 + 5 * 3
                 ⇒    2 + 15
                 ⇒    17          ✗


#define add_five(x)  x + 5        BAD!!!

add_five(7) * 4  ⇒    7 + 5 * 4
                 ⇒    7 + 20
                 ⇒    27          ✗
```

Rule: Parenthesize (almost) everything for safety.

```
#define triple(x)  ((x) * 3)      GOOD!!!

triple(2 + 5)       ⇒   ((2 + 5) * 3)
                    ⇒   ((7) * 3
                    ⇒   21          ✔
```

```
#define add_five(x)  ((x) + 5)          GOOD!!!

add_five(7) * 4     ⇒   ((7 + 5) * 4)
                    ⇒   ((12) * 4)
                    ⇒   48            ✔
```

Don't use printf(…) to debug

```
printf("%d\n", squared(1));
printf("%d\n", squared(2));
printf("%d\n", squared(3));
printf("%d\n", squared(4));
printf("%d\n", squared(5));
```

OUTPUT

```
1
4
9
16
25
```

Problem: Output is hard to interpret.

Don't use `printf(…)` to debug

```
printf("squared(1) == %d\n", squared(1));
printf("squared(2) == %d\n", squared(2));
printf("squared(3) == %d\n", squared(3));
printf("squared(4) == %d\n", squared(1));
printf("squared(5) == %d\n", squared(5));
```

Problem: Separating your debugging printf's from the printf's that support the core functionality may be error-prone.

Partial solution: Use a variadic macro

```
#define my_printf(...)  printf(__VA_ARGS__)


my_printf("squared(1) == %d\n", squared(1));
my_printf("squared(2) == %d\n", squared(2));
my_printf("squared(3) == %d\n", squared(3));
my_printf("squared(4) == %d\n", squared(1));
my_printf("squared(5) == %d\n", squared(5));
```

Problem: Copying expression in two places per line may be error-prone.

Better solution: Use #ifdef ⋯ #endif to toggle.

```
#ifdef ENABLE
#   define my_printf(...)  printf(__VA_ARGS__)
#else
#   define my_printf(...)
#endif

#define ENABLE
my_printf("squared(5) == %d\n", squared(5));

printf("squared(5) == %d\n", squared(5));

// #define ENABLE
my_printf("squared(5) == %d\n", squared(5));

;
```

clog.h exists to solve this in a versatile way.

- log_int(…), log_str(…), and log_addr(…) allow you to print an expression and its value.

- `#ifdef … #endif` prevents interference with the main functionality of any project.

- log_*«color»*(…) make the output easier to read and interpret, even when intermingled with the program's normal output.

gcc [redacted] -DDEBUG ⇔ #define DEBUG

```
#ifdef ENABLE
#   define my_printf(...)  printf(__VA_ARGS__)
#else
#   define my_printf(...)
#endif
```

```
gcc -o test_squared test_squared.c squared.c -DDEBUG
my_printf("squared(5) == %d\n", squared(5));
↓
printf("squared(5) == %d\n", squared(5));
```

```
gcc -o test_squared test_squared.c squared.c -DDEBUG
my_printf("squared(5) == %d\n", squared(5));
↓
;
```