Computer Systems

CS107

Cynthia Lee

Today's Topics

LECTURE:

- The build process
  - Taking a look at each step of the process
  - Preprocessor, compiler, assembler, linker, loader

Build components

Understanding the role of (and possible errors in) each step of "compiling"/building an executable.

Build components

PREPROCESSOR:

- Takes #include and #define and other preprocessor directives and replaces them with appropriate text (code.c + .h files → code.i)
  - `gcc -E code.c`

COMPILER:

- Takes processed C code and outputs appropriate AMD64 code (code.i → code.s)
  - `gcc -S code.c  # uppercase S`

ASSEMBLER:

- Takes assembly output and makes machine output (code.s → code.o)
  - `gcc -c code.c  # lowercase c`

LINKER:

- Takes .o in question, plus other module .o files, joins them together to make the executable (code.o +.o files → code)
  - `gcc code.c -o code # lowercase o`

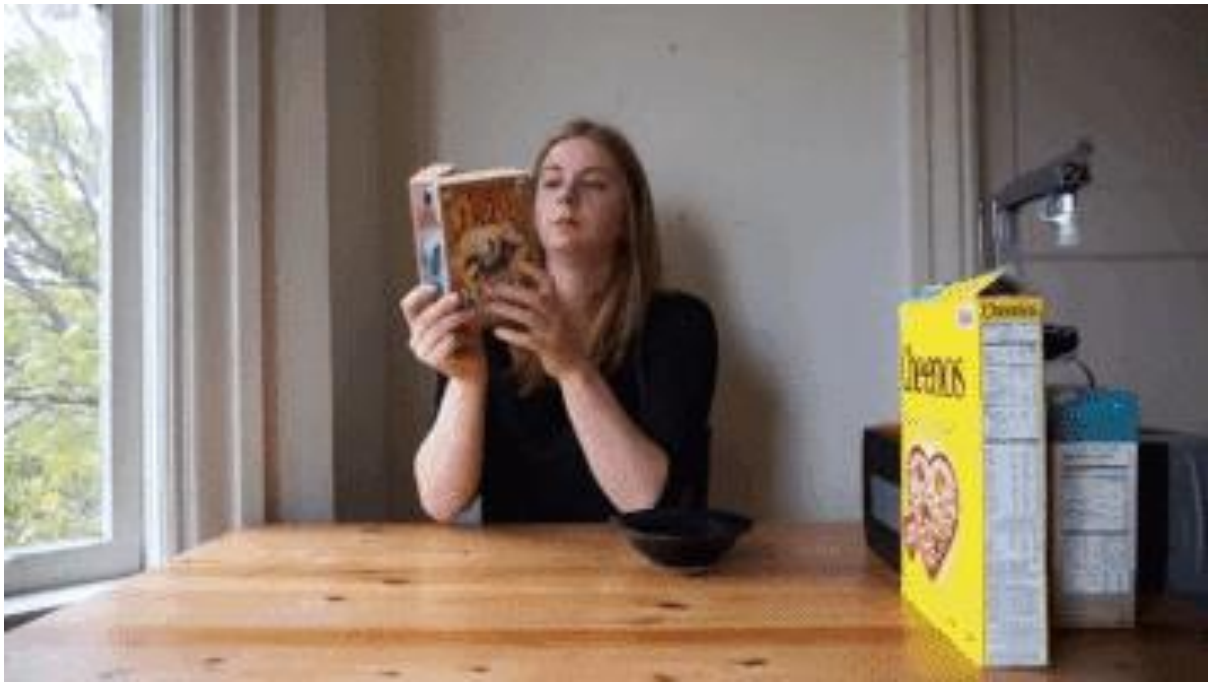
Preprocessor

```
#ifndef _CVEC_H_
#define MAX_FRAG_LENGTH 1024
#include <stdio.h>
```

The Preprocessor

- Takes #include and #define and other preprocessor directives and replaces them with appropriate text (code.c + .h files → code.i)
  - `gcc -E code.c`

The Preprocessor



Dead simple:

just pastes text

- `#include <foo.h>`
  - Pastes full text of foo.h in place of this line
- `#define FOO BAR`
  - Pastes “BAR” in place of every “FOO”

- Doesn’t really understand much about C/C++ specifically
- Doesn’t do much of any error checking
  - Will warn about unmatched/unterminated string literal quotes " "
  - **Can you think of one error it will give?**

The Preprocessor: macros

- #define is not just for constant integers—replaces **any** keyword with arbitrary other text
  - `#define MAX_FRAG_LENGTH 1000`
  - `#define DO_FOREVER while(1)`
  - `#define AVERAGE`
- Macros with arguments:
  - `#define ABS(x) ((x) < 0 ? -(x) : (x))`
- Macros tend to create non-obvious errors:
  - Variables names accidentally overlap and are redefined or changed
    - `#define while if`
  - Non-obvious cause & effect

- **Alternative: `inline`**

```
static inline int Absolute(int x) {
    return (x < 0 ? -x : x);
}
```

The Preprocessor: Your turn

- Here are a few source and header files:

basic.c:

```
int one = 1;
#include "myinc1.h"
int two = 2;
```

myinc1.h:

```
int three = 3;
#include "myinc2.h"
int four = 4;
```

myinc2.h:

```
int five = 5;
```

- **In what order are the global variables listed in basic.i?**

A. five, three, four, one, two
B. one, three, five, four, two
C. one, three, five, two, four
D. one, two, three, four, five
E. Other

Include interactions

- Here are a few source and header files:

basic.c:

```
#include "vector.h"
#include "pq.h"

int main(int argc, char *argvp[]) {
    v_global = pq_global;
    ...
```

vector.h:

```
int v_global = 5;
...
```

pq.h:

```
#include "vector.h"
int pq_global = 7;

void update() {
   v_global++;
}
...
```

- **Which of the following errors does the above code generate?**

A. undeclared variable(s) in basic.c
B. undeclared variable(s) in pq.h
C. Variable(s) are declared outside functions
D. re-declaration of variable(s)
E. Other/none/or more than one of the above

The Preprocessor: multiple include guards

- You've seen this in 106B and this quarter:

```
#ifndef _MY_INCLUDE_H_
#define _MY_INCLUDE_H_

[body of myinclude.h file]

#endif
```

- Preprocessor isn't smart enough to handle problem of a file being included multiple times
- This could lead to things like global variables being declared more than once—compiler error!
- It's on us to guard against multiple include by having the .h file set up its own defense

Compiler

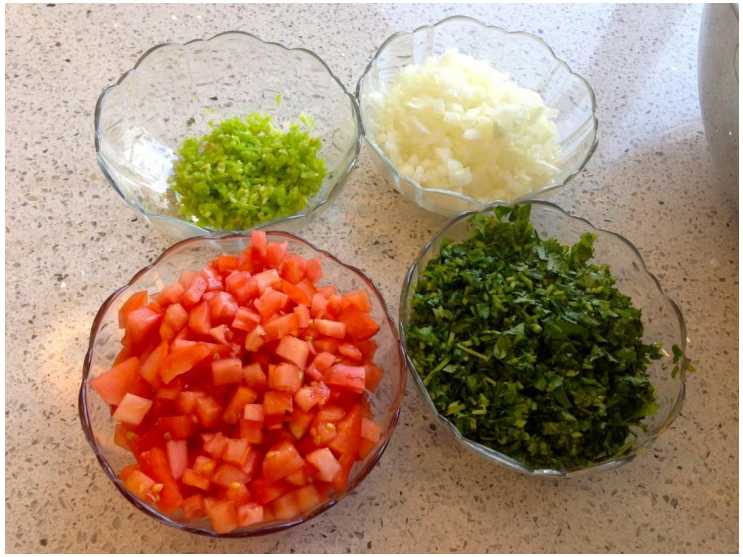
UNLIKE PREPROCESSOR, *REALLY* SMART!

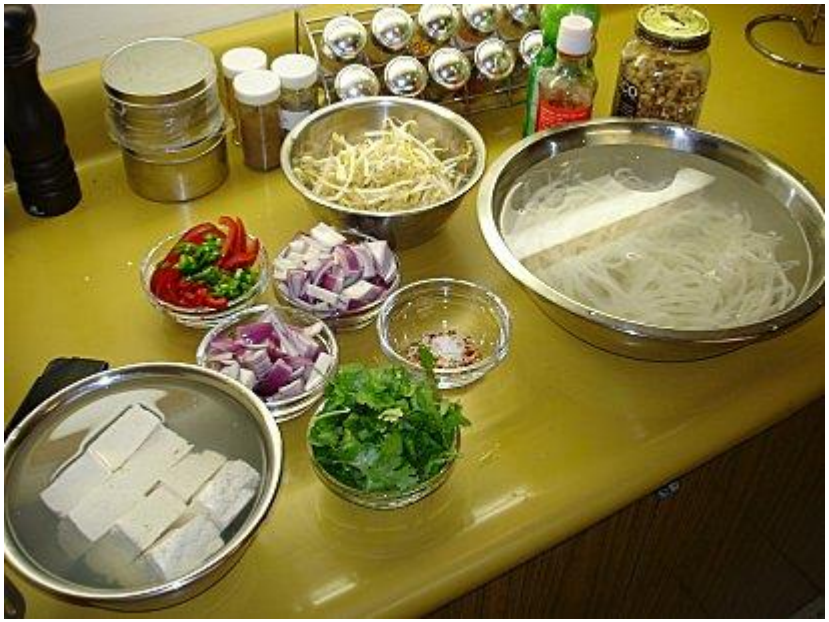
Compilers are super smart

- (good ones anyway)
- Responsible for detecting syntax errors and semantic errors (e.g., type errors)
- Responsible for determining best sequence of assembly instructions to accomplish what C code asked for—optimizations!

- **Wow, it does all that! What does it not do?**
  - Doesn't fully resolve all symbols
  - Some are given as prototypes, but not defined yet
  - It will just leave a placeholder for these

- Look at "`nm code.o`" output to see symbols





Linker

All the ingredients are chopped, prepared, ready to go. Now we combine them for the final dish!

Linker combines .o output of many modules

- **Where did we leave off from the previous steps? *(What does the compiler not do?)***
  - Doesn't fully resolve all functions and data
  - Some are given as prototypes, but not defined yet
  - Compiler will just leave a placeholder for these

- **Linker:**
  - Wants to have one and only one definition for each symbol
  - Combines text section for each module and lays them out in the address space
  - Now goes back to placeholders and fills them in (base address of the text section for that module + offset within the module)

Static linking vs. dynamic linking

- **Static linking**
  - All modules' text section is actually built into the executable
  - Completely stand-alone
  - Very large file size

- **Dynamic linking**
  - Standard libraries are held in common between executables on the system
    - Both on the file system, and also while running
  - A more modern way of handling linking

Let’s Play: When does that error show up?

Which component **detects and alerts you** to the problem?

- Here are a few source and header files:

basic.c:

```
#define TWO 2;
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc >= TWO) {
        printf("%s\n", argv[1]);
    }
    return 0;
}
```

A. **Preprocessor**
B. **Compiler**
C. **Assembler**
D. **Linker**
E. **Other or Runtime or Never**

Which component **detects and alerts you to** the problem?

- Here are a few source and header files:

basic.c:

```
#define ONE 1
#define TWO 2
#include <stdioooo.h>

int main(int argc, char *argv[]) {
    if (argc >= TWO) {
        printf("%s\n", argv[1]);
    }
    return 0;
}
```

A. Preprocessor

B. Compiler

C. Assembler

D. Linker

E. Other or Runtime or Never

Which component **detects and alerts you to** the problem?

- Here are a few source and header files:

basic.c:

```
#define ONE 1
#define TWO 2
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc >= TWO) {
        printfff("%s\n", argv[1]);
    }
    return 0;
}
```

A. **Preprocessor**
B. **Compiler**
C. **Assembler**
D. **Linker**
E. **Other or Runtime or Never**

Which component **detects and alerts you to** the problem?

- Here are a few source and header files:

basic.c:

```
#define ONE 1
#define TWO 2
#include <stdio.h>
void printfff(char* str, char* str2);
int main(int argc, char *argv[]) {
    if (argc >= TWO) {
        printfff("%s\n", argv[1]);
    }
    return 0;
}
```

A. Preprocessor

B. Compiler

C. Assembler

D. Linker

E. Other or Runtime or Never

Which component **detects and alerts you to** the problem?

- Here are a few source and header files:

basic.c:

```
#define ONE 1
#define TWO 2
//#include <stdio.h>
int printf( const char* format, ... );
int main(int argc, char *argv[]) {
    if (argc >= TWO) {
        printf("%s\n", argv[1]);
    }
    return 0;
}
```

A. Preprocessor

B. Compiler

C. Assembler

D. Linker

E. Other or Runtime or Never

Midterm #2 Information

Midterm #2

- Not intended to be “comprehensive” but at the same time can’t not be (pointers continue to be important, obviously)

- **Assign3**
  - Midterm 1 covered client use of cvector, Midterm 2 will cover implementation side. Can you write a new function for cvector or cmap? Can you write a malloc line of code to allocate a different kind of blob? Can you join two blobs in a linked list?
- **Assign4**
  - Minimal floating point question
  - We already did integer representation and bitwise operators on Midterm1, but perhaps minimal additional question on it
- **Assign5**
  - Given a piece of assembly code, could you figure out the corresponding C code?
  - In-class exercises are good practice, as well as working on bomb

Binary Bomb Tips

Binary Bomb tips

1. **Securely defuse your bomb**
   - What if, when running it in gdb, execution always stopped RIGHT before things got dicey, wouldn't that be nice? Hmm….how do we cause gdb execution to stop in a specified location?
   - Once you figure that out, you may want to set that up in your .gdbinit file so you won't accidentally forget to manually set it every time you run gdb
2. **Review the assembly code for your bomb**
   - Do objdump, save it to a file, and begin annotating the file with C code and/or pseudocode translations of what is happening
   - Suggestion: go through and normalize the register names so you can make connections between places in the same function that are talking about the same register but with different names (e.g. %rax = %eax). It can be harder to quickly see the threads connecting parts when the names are changing (keep in mind that very occasionally the difference is meaningful for some reason—e.g. only one byte is wanted—but still useful to connect the parts)
3. **Level 5**
   - Tricky—lets' talk about that….