Difference between assembly and machine code?
- Why do we need the assembler?

```
top:  add .... address
      move ....
      jmp top 
```
Assembler:
- Scan each line and convert to machine code
- Assign each symbol (it can) an address
- These symbol-to-address "translations" are put in a symbol table

Assembly language

Assembler

Machine code ➔ Symbol table

objdump -t

After assembling into machine code
"Linking" - collects and combines machine code and data into a file that is
"Loaded" - copied into memory to be ready to execute
Addressing in machine code
- Absolute - the actual (virtual) address
- Relative - save space use PC + offset
(Indirect)

\[
\text{movl } 0x5000000, \%eax
\]

\[
\text{while (...) \{ }
\]

\[
\text{jmp 0x80415ab - 10}
\]

---

need a linker to compute these addresses

\[
\text{mod1.c}
\]

\[
\text{int a = 12;}
\]

\[
\text{main () \{ }
\]

\[
\text{f1();}
\]

\[
3 \Rightarrow \text{Assembly doesn't know address}
\]

\[
\text{mod2.c}
\]

\[
\text{f1();}
\]

\[
\text{a++;}
\]

\[
3 \Rightarrow \text{Assembler can't know what address this is}
\]

later the linker fills in the missing addresses
int a = 12;

int main(int argc, char *argv[]) {
    for (int i=0; i<10; i++) {
        a += 1;
    }
    return 0;
}
addressing: file format elf32-i386

Disassembly of section .text:

08048394 <main>:
  08048394: 55            push %ebp
  08048395: 89 e5         mov %esp,%ebp
  08048397: 83 ec 10       sub $0x10,%esp
  0804839a: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%ebp)
  080483a1: eb 11         jmp 080483b4 <main+0x20>
  080483a3: a1 2e 06 04 08  mov 0x804962c,%eax
  080483a8: 83 c0 01       add $0x1,%eax
  080483ab: a3 2e 06 04 08  mov %eax,0x804962c
  080483b0: 83 45 fc 01     addl $0x1,-0x4(%ebp)
  080483b4: 83 7d fc 09     cmpl $0x9,-0x4(%ebp)
  080483b8: 7e e9          jle 080483a3 <main+0xf>
  080483ba: b8 00 00 00 00  mov $0x0,%eax
  080483bf: c9             leave

Disassembly of section .data:

08049628 <__data_start>:
  08049628: 00 00
  ...
  0804962c <a>:
  0804962c: 0c 00
  ...
  0804962c <ca>:
  0804962c: 0c 00
  or $0x0,%al
  ...
gcc -c → only do preprocessor + compiler + assembler

L → .o file

ld → linker → links .o files together

CPP (pre processor)  GCC → all these together

CC (compiler)

AS (assembler)

ld (linker)

---

Linker fills in other missing info too
Creates the ELF file format for executables

→ 15+ segments
  - data, rodata
  - text, bss

mod 1.0 symbols

data 1

code1

mod 2.0

code2

data

code2

code1

code2

app
fills in blanks, assigns addresses

- relocate object files
  - update any addresses of moved objects

---

**Loading**

- done (mostly) by the OS
- copy executable (created by linking) into memory
- OS allocates some space for stack
  - sets `%esp`
- OS allocates some space for heap
  - sets `dptr` pointer
- sets the PC to first instruction
  - starts executing code
    - logically jumps to main
      - (libraries actually goes to `init`)
Previously, we assumed static linking

Dynamic linking is common.

In parts of the linking step, saved until execution/loading.

Why dynamic?
- two programs both using same func.
  - save memory by having 1 copy
- lots of code never needed
- reduces executable size
- eliminates duplication in executables
- to dynamically choose which function