C Introduction

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

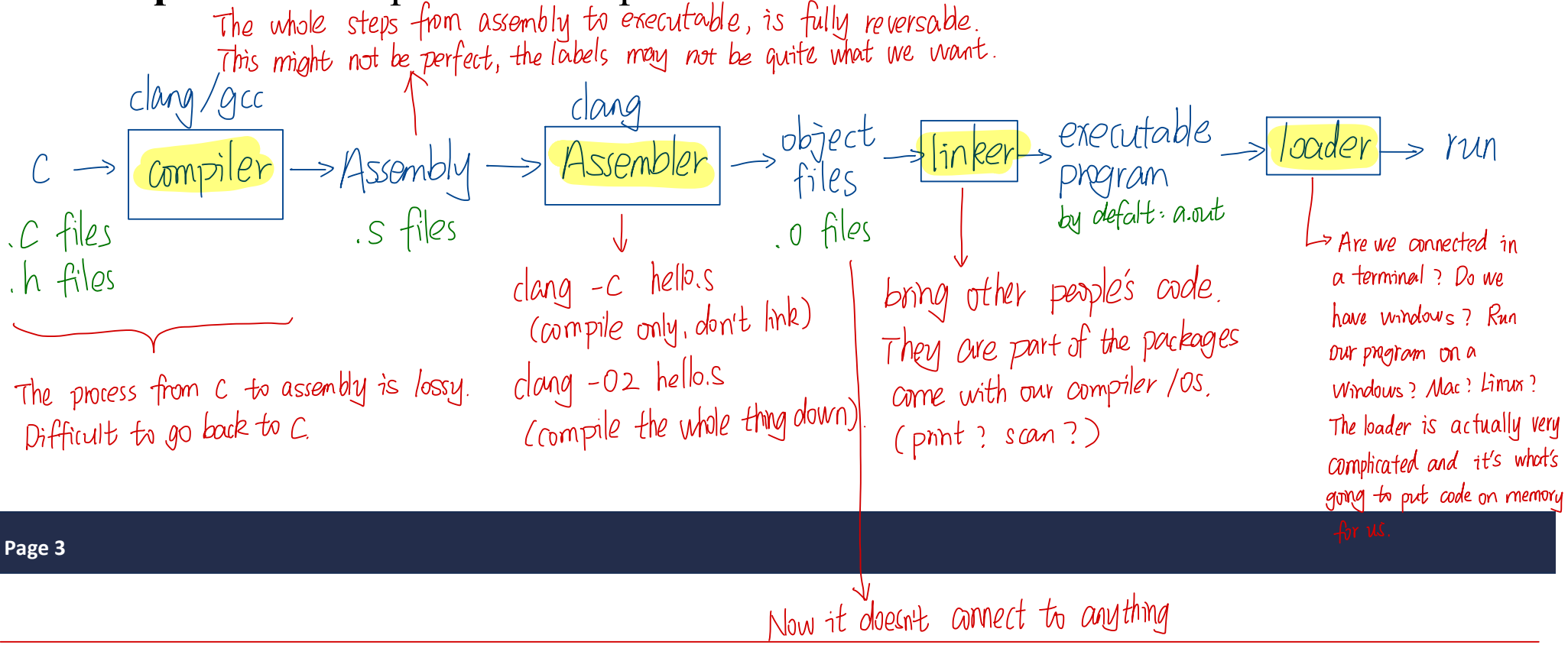
- Homework 6 due Monday at 11:59pm
- Midterm 2 is 2 weeks away (April 3)
  - Similar format to midterm 1
  - Schedule with SDAC early if needed

Compilation Pipeline

We want to bridge the gap between the assembly and C.

Turning our code into something that runs

- **Pipeline** - a sequence of steps in which each builds off the last



ls /usr/lib : lots of compiled things ready to be used.

ls /usr/lib64 | wc -l : show how many files.

C

C is a thin wrapper around assembly

- This is by design!
- Invented to write an operating system
  - Can write inline assembly in C
- Many other languages decided to look like C

Simple C Example

```
int main() {
    int y = 5;
    return 0;
}
```

Compilation Pipeline

Earlier, we saw:

- C files (`.c`) compiled to assembly (`.s`)
- Assembly (`.s`) assembled into object files (`.o`)
- Object files (`.o`) linked into a program / executable

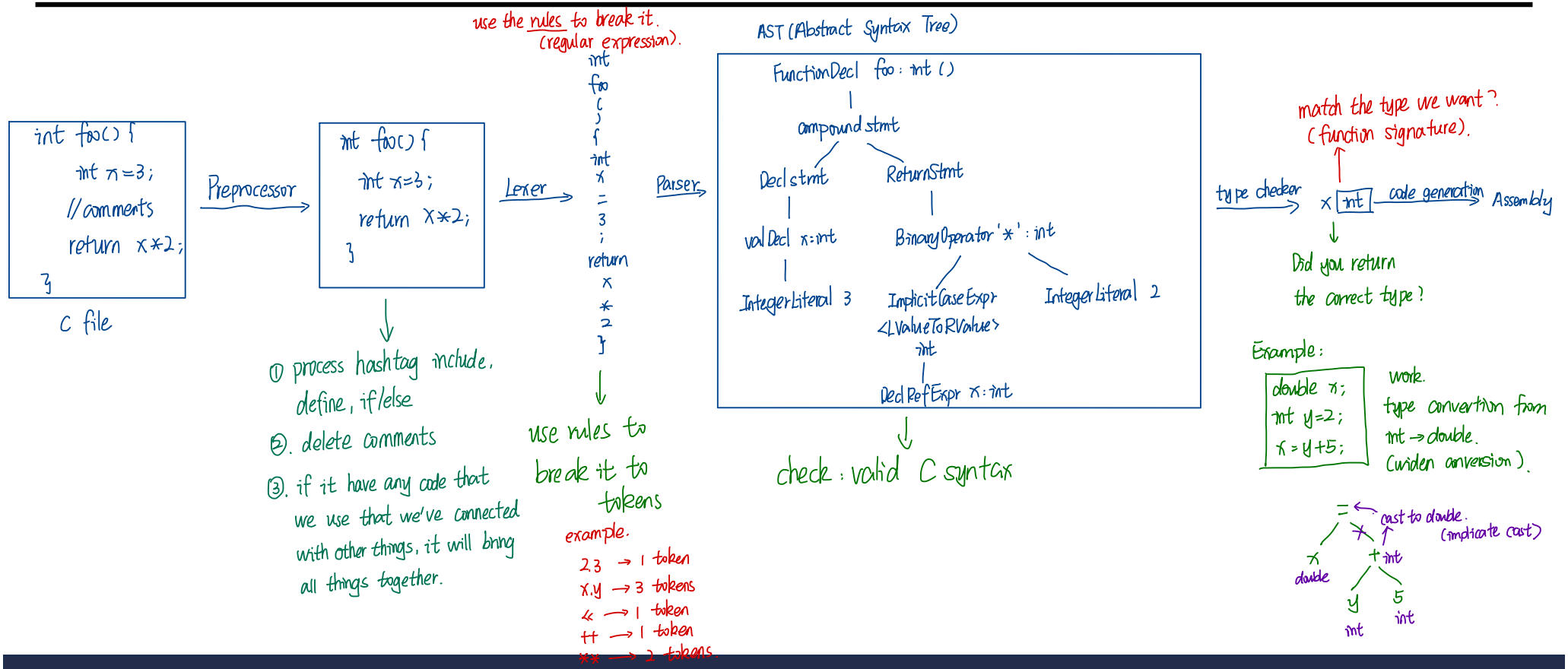
Compiling C to Assembly

Multiple stages to compile C to assembly

- Preprocess - produces C
  - C is actually implemented as 2 languages: C preprocessor language, C language
  - Removes comments, handles preprocessor directives (#)
  - `#include`, `#define`, `#if`, `#else`, …
- Lex - breaks input into individual tokens
- Parse - assembles tokens into intended meaning (parse tree)
- Type check - ensures types match, adds casting as needed
- Code generation - creates assembly from parse tree

→ has a lot of things like hashtags

Compiling C to Assembly



Compiling C to Assembly

Errors

Compile-time errors

- Errors we can catch during compilation (this process)
- **Before** running our program

Runtime errors (Example: Segmentation fault).

- Errors that occur when running our programs

clang → better error message

gcc

Simple C Example (we demonstrated an example in portal. check recording).

```
int main() {
    return 0;
}
```

The `main` function

- Start running the `main()` function
- `main` must return an integer - **exit code**
  - 0 = everything went okay
  - Anything else = something went wrong
- There *should* be arguments to main

```c
int main() {
    return 0;
}
```

clang -O0 -c hello.c -Wno-implicit-function-declaration -o hello.o

-c → compile only, no link.

-O0 → No optimization

-Wno-implicit-function-declaration → I don't want the warning. (for example, I call a function but didn't declare it.)

llvm-objdump -d hello.o

-d → disassemble.

```
hello.o:        file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       b: 31 c0                         xorl    %eax, %eax
       d: 5d                            popq    %rbp
       e: c3                            retq
```

This is the disassemble code for main. start from address 0.

This is an unnecessary local variable initialization (a dead store), artificially introduced by compiler under -O0 for simplicity and debuggability.

1. why xorl %eax, %eax ?

① Can we use movl $0, %eax ? Yes, but xor is shorter and faster.

② Why not xorq %rax, %rax ?

The compiler is choosing to use this as an easy way — a short way — to zero out the return register. It's only two bytes to clear that register with zero. Performing a 32-bit xor on eax automatically clears the upper 32 bits to zero.

2. For different return types:

Regardless of whether the return type is int, short, long, long long. according to the x86-64 calling convention. return values are passed via the rax register. For floating-point types, such as "float" and "double", use xmm0. (Completely different hardware).

```c
int main() {
    return 0;
}

long foo(){
    return 0;
}
```

```
0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       b: 31 c0                         xorl    %eax, %eax
       d: 5d                            popq    %rbp
       e: c3                            retq
       f: 90                            nop

0000000000000010 <foo>:
      10: 55                            pushq   %rbp
      11: 48 89 e5                      movq    %rsp, %rbp
      14: 31 c0                         xorl    %eax, %eax
      16: 5d                            popq    %rbp
      17: c3                            retq
```

```c
int main() {
    puts("It's Friday!");
    return 0;
}
```

```
hello.o:        file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 48 83 ec 10                   subq    $0x10, %rsp
       8: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       f: 48 8d 3d 00 00 00 00          leaq    (%rip), %rdi
      # 0x16 <main+0x16>
      16: b0 00                         movb    $0x0, %al
      18: e8 00 00 00 00                callq   0x1d <main+0x1d>
      1d: 31 c0                         xorl    %eax, %eax
      1f: 48 83 c4 10                   addq    $0x10, %rsp
      23: 5d                            popq    %rbp
      24: c3                            retq
```

→ It calls something it doesn't know about. Because I haven't actually told it.

The linker will eventually overwrite and tell it where to actually call "puts".

```
mrq9gz@portal03:~/examples1$ clang hello.c -o hello
hello.c:2:5: error: call to undeclared function 'puts'; ISO C99 and later do not support implicit function declarations [-Wimplicit-function-declaration]
    2 |     puts("It's Friday!");
      |     ^
1 error generated.
```

```c
int puts(const char * string);

int main() {
    puts("It's Friday!");
    return 0;
}
```

manually declare puts.

Declaration → let compiler know the signature of the function.

```c
#include <stdio.h>

int main() {
    puts("It's Friday!");
    return 0;
}
```

Also works, and better

if I use "cpp hello.c | less", I can see all things in stdio.h is actually copied to the hello.c file.

cpp: c preprocessor