Introduction to C, C++, and Unix/Linux

CS 60

Lecture 2: History

Today

→ C, Unix/Linux and C++

→ Compilation Process…

→Reading for next class: K&R ch. 1-2 & 7.1-7.4.

Comment on the lectures

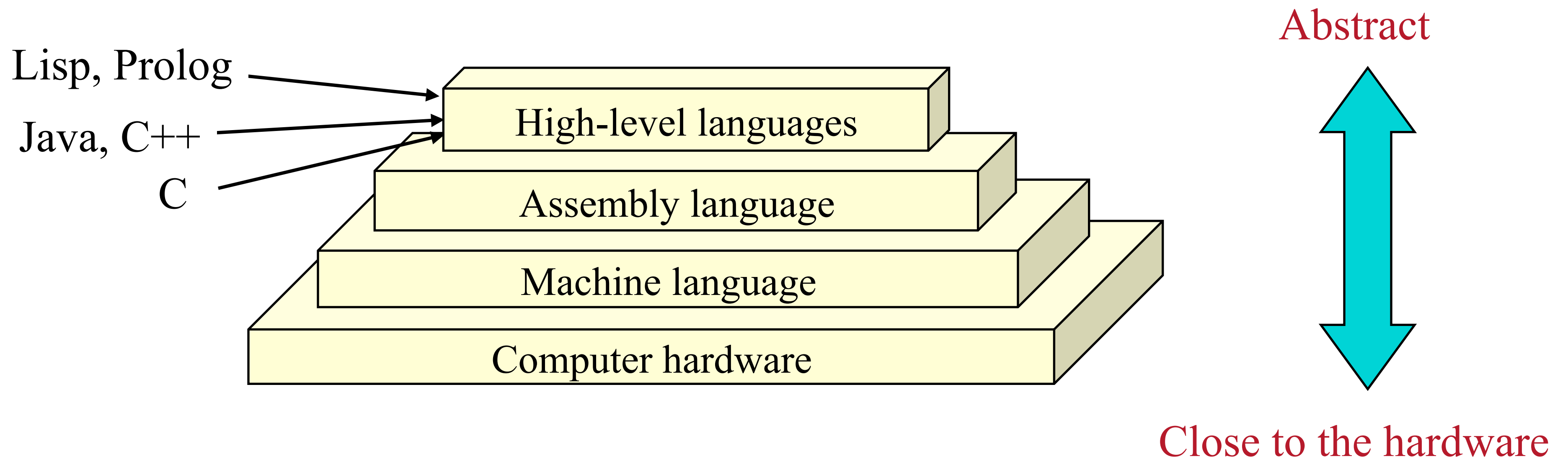
- I won't be going over every section of the books
- Rather, I'll be highlighting some things in the books, and presenting other material to complement what you're learning from the books
  - But feel free to ask questions from the books
- Soon we'll start focusing much more on C details
  - Make sure you know the basics and explore the language constructs.

The C programming language

- C is a procedural (or imperative) programming language
  - It specifies an explicit sequence of steps (instructions) to follow: do this, then this, then this…
- C is a compiled (not interpreted) language
- Java and C++ are object-oriented programming (OOP) languages
  - Forget objects for now – C does ***functions***

The C programming language

- C is a “low-level high-level” programming language



The C programming language

- Some languages like Lisp or Prolog attempt to completely divorce the programmer of any knowledge of the actual hardware running the program
- Other languages are more closely tied to the hardware
- C is very close to the hardware and some programs require knowledge of how the computer actually works internally
- One may compare C to a "sharp knife": A great and efficient tool for precision tasks, but DANGEROUS!

C vs. Java

- On the surface, C is much simpler to deal with
  - Almost everything is permissible!
- Simplicity comes at a high price
  - Freedom encourages lack of structure
  - Not as much compile-time validation
  - Cross-platform portability uncertain
    - E.g., size of ***int*s** and ***float*s**
- Closer to the "bare metal"
  - Micro-managing I/O operations
  - Memory management and pointers
  - Little run-time error checking

Why C?

- The language is ubiquitous.
- C is “fast” relative to interpreted languages.
- C is “fast” compared with any language that has a lot of built-in overhead to keep you out of trouble
- It’s available on almost every platform, and with recent standards, it’s become possible to write (fairly) portable code.

Portability of C

- C is probably the lowest level language where some portability can be assured across architectures (platforms)
  - Thus, it's the lowest level high level language…

A short history of C and Unix

- C was originally created as a language with which one could write operating systems (like Unix)
- All of this took place in the early 1970s when:
  - Hardware was much slower and much more expensive
  - Memory was incredibly expensive (> $0.01/bit!)
  - Operating systems and compilers were expensive
  - Operating systems were difficult to use (e.g., JCL)
  - On-line access was limited
  - Serious coding was done in assembler
  - Writing operating system code in assembler was necessary

Predecessor of Unix

- In the 1960s, several big players teamed up to create the greatest operating system that never was, known as “Multics”
- Multics was good, but it was too big – it didn’t fit well on any existing hardware
- It didn’t work out…

A short history of C and Unix (cont.)

- Two researchers are AT&T decided to use some of the technologies developed in the Multics project in a new OS (this was a home project)
  - All because they wanted to run a Space Travel game, actually
- They called it “Unix” as a parody of Multics, since it was much simpler, and light weight.

Ken Thompson

Dennis Ritchie

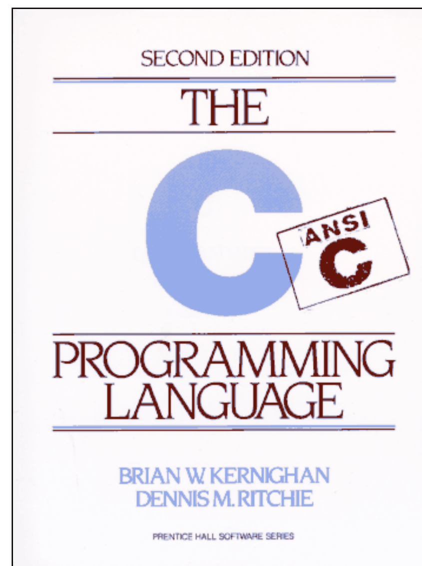
A short history of C and Unix (cont.)

- Ritchie and Thompson found they needed a new language to quickly implement the UNIX operating system
- Thompson created "B" by modifying an existing language called BCPL (created by London and Cambridge Univ., UK), in order to fit onto the PDP-7's 8K word memory size
- Later they decided to switch to the new PDP-11 and created a new version of B….

Thus was born

A short history of C and Unix (cont.)

- In 1978, Dennis Ritchie and Brian Kernighan published a book describing the C language
  - The book was the de facto standard for C
  - It became the best selling computer book ever (~500,000 copies sold)



← Second edition, 1988

A short history of C and Unix (cont.)

- By the early 1980's, C and Unix became very widely used, especially on the very popular and cheap (at that time) DEC Vax machines
- The PC makers soon realized C's advantages over BASIC, and provided implementations for the PC market
- Other implementations arose, and the C language began to change under corporate influences
- Soon it became clear that a formal standard was needed, just like every other language had

A short history of C and Unix (cont.)

- In December, 1989, ANSI formally adopted "Standard X3.159-1989", or "ANSI-C"
- C is now governed by various standards (e.g., ANSI)
  - But they got a late start – there was already a lot of C code in existence!
- Rewrite everything in order to make standardization easier???
  - NO….

A short history of C and Unix (cont.)

- C and Unix have a very intertwined history
- **Unix** was first implemented in assembly on the PDP-7 and PDP-11, and later rewritten in C when a compiler was available

First Users

- Bell Labs Patent Dept. (nroff and troff)
- Universities (free OS and compilers)
- Later on UC Berkeley Grads Jumped into it
- Two main threads: Bell Labs Unix ----- BSD Unix
- Newer Unix has features from both
- Multi users were allowed concurrently

Unix

- Provided hierarchical directory structure
- Allowed files and processes have a location in the directory structure
- Provided services to create, modify & destroyed processes & files
- Allowed sharing of resources (CPU, memory, disk space)
- Communication pipe (medium speed) and sockets (different machines)

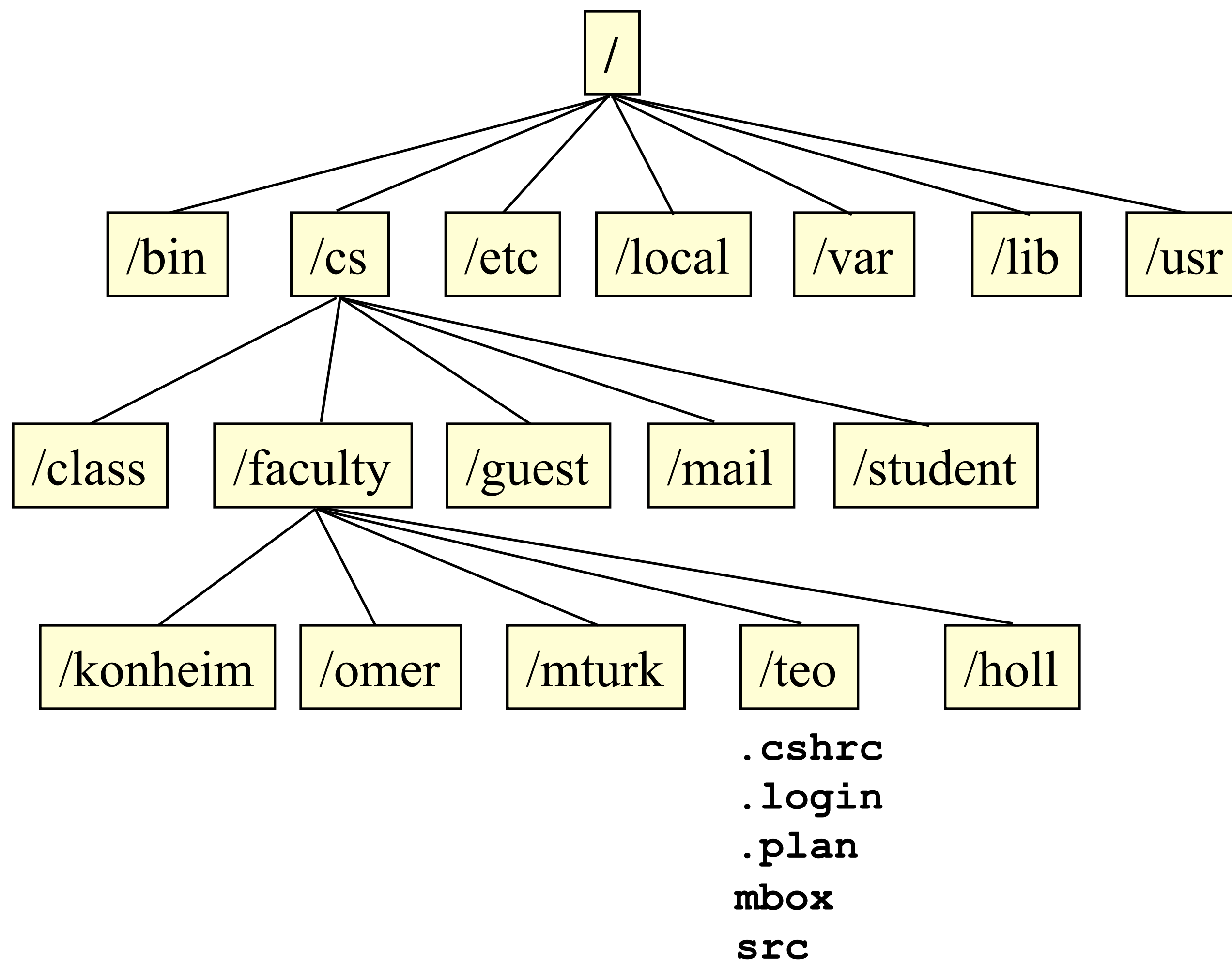
Unix (cont)

- Utilities (editors, compilers, shells, GUI, sorting, etc)
- Open system (source code available)
- From C you can access everything (parallel processing, interprocess communication, file handling
- Large number of utilities
- Portable OS

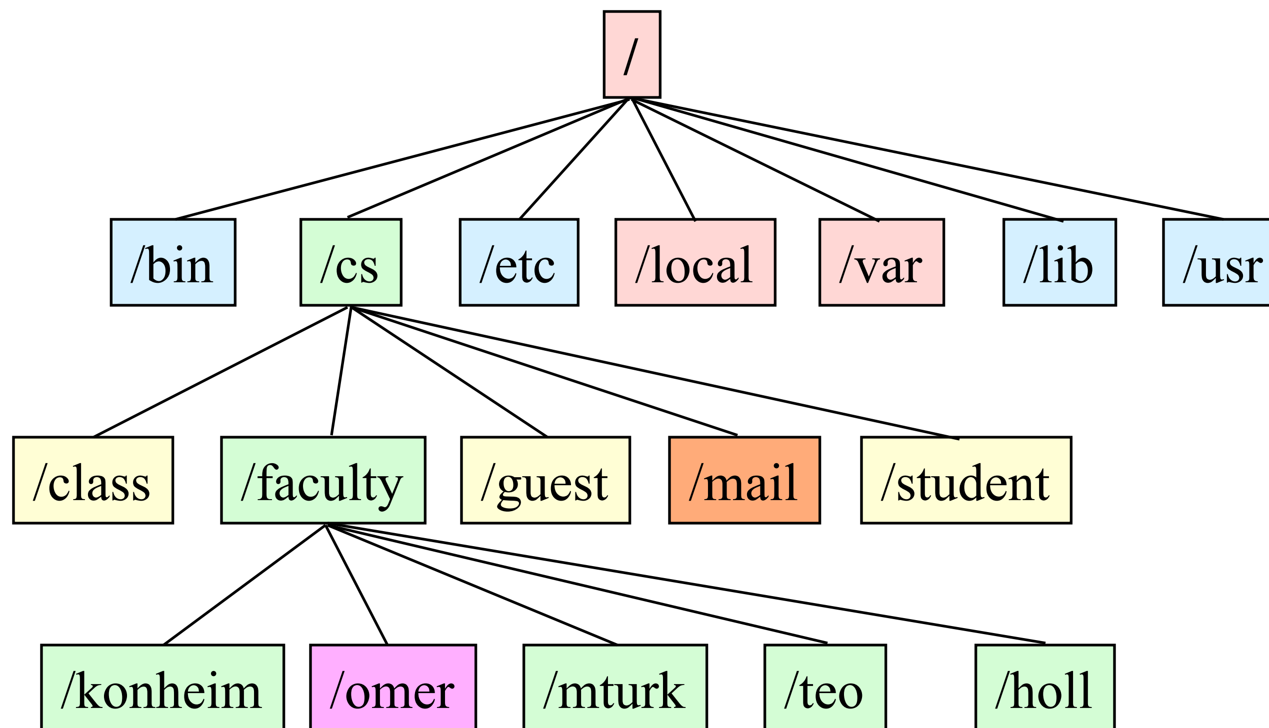
Unix file structure, directories, etc.

- Files and directories (“folders” in Windows)
  - **Files** contain text, data, programs, etc.
  - **Directories** contain files (and other directories, a.k.a. subdirectories)
- Directories are structured in a tree, with the / directory at the root of the tree
  - That’s the directory named “/”
  - Directories have only one parent directory (well…)
  - “.” refers to the current directory
  - “..” refers to the parent directory

File structure, directories, etc. (cont.)



In a networked environment…



The files and directories might physically reside on many different machines (color-coded here)

Unix Philosophy

- Utilities: Do one thing well and then combine utilities via a pipe (who -> sort -> terminal)
- Solve problem with multiple utilities
- Super-User and partial super-users (may be dangerous)

Unix and Linux

- Linus Torvalds, working on a CS degree in Finland in 1991 decided to write his own operating system for the Intel (i386) platform, basing it on Tanenbaum's "Minix" (a small UNIX look-alike). Not the first to do this, but it was FREE.

- The rest is history….

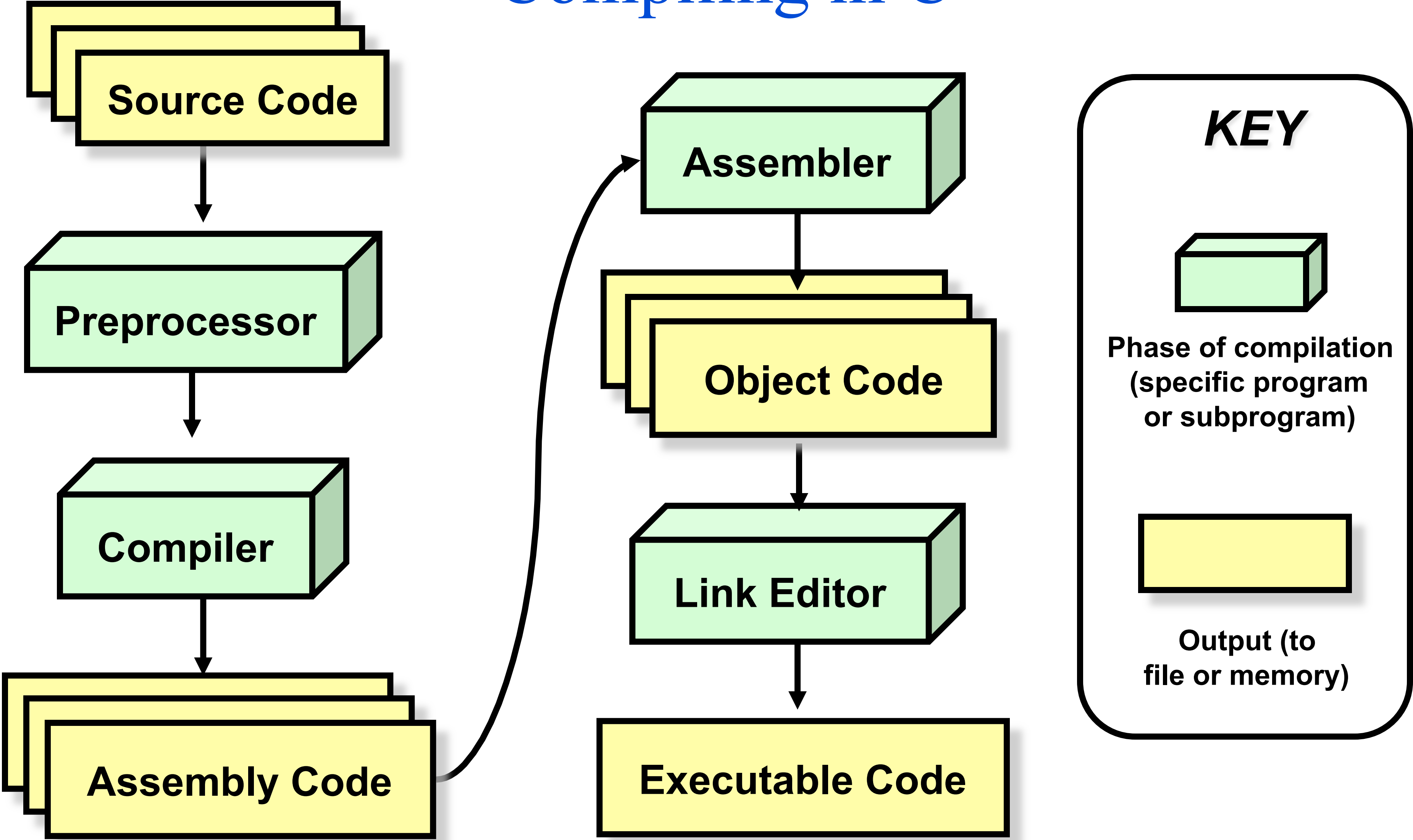
Object Oriented Programming

- Emphasis on Data -> Fit Language to problem (rather than fit problem to procedural language)
- Classes and objects
- C++ :
  - Information hiding (safeguards from improper use)
  - Polymorphism (multiple defs for opers and functions)
  - Inheritance (derive new classes from other ones)
  - Generic programming (type ind)

C++

- Bjarne Stroustrup (early 80s)
- Easier to write good programs
- Based on C and inspired by Simula 67
- Portability (main goal)
- g++ prog.C

Compiling in C



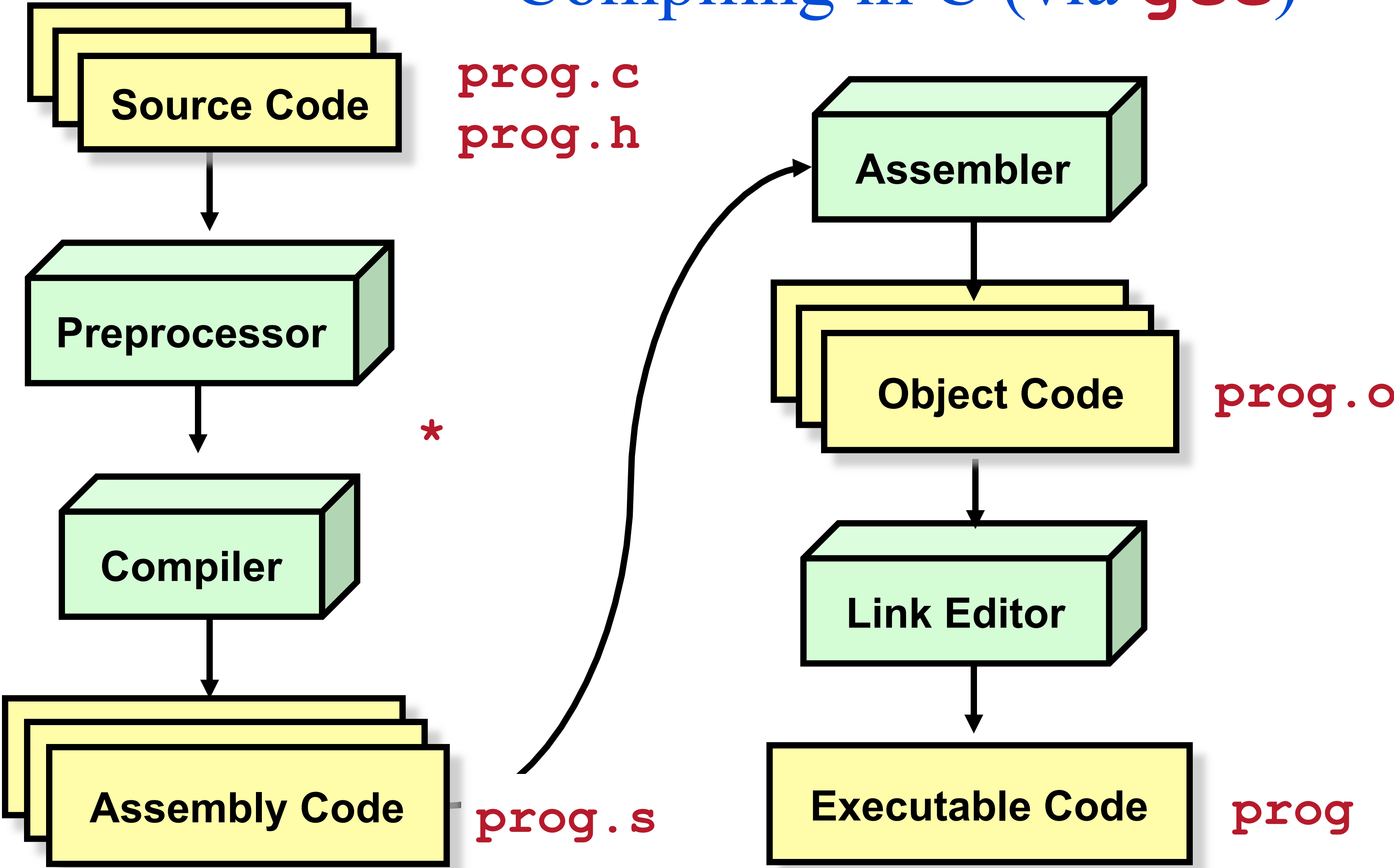
Source code

- C source code is written in text files. By convention, these files are given an appropriate name and a “.c” extension.
- Header files (“include files”) also contain C source code, but they should not include actual instructions, functions, variables, etc.
  - Definitions, constants, function declarations, compiler directives….

Intermediary files

- **`gcc -E`** produces source code after preprocessing
  - Prints to standard output
- **`gcc -S`** produces assembly code (.s)
- **`gcc -c`** produces object (machine) code (.o)
- **`gcc`** produces an executable file
  - **`-o prog`** produces a program called "prog"
  - **`./prog`** runs the program

Compiling in C (via gcc)



C preprocessing

Lines starting with # are Pre-Processor directives. They are not statements, hence do not end in ';'

```
#include <stdio.h>
main()
{
 if (1)
     printf("Its summer!\n");
}
```

The C preprocessor has many features – and pitfalls.

We will cover this later on.