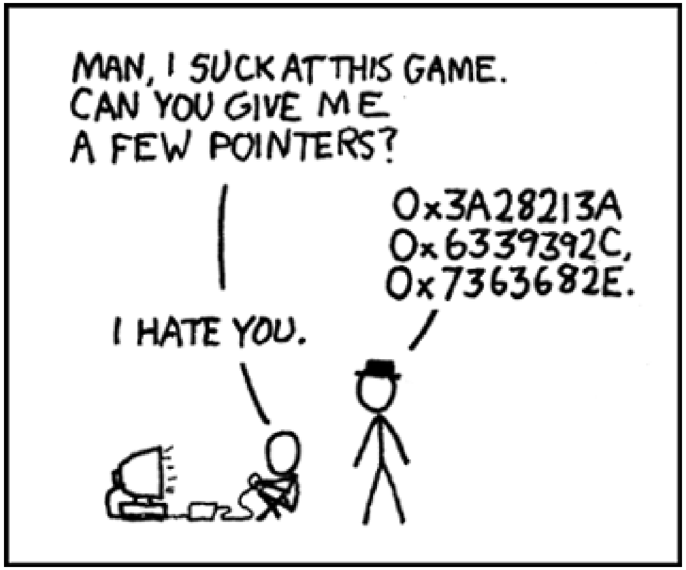
CS 261
Fall 2016

Mike Lam, Professor

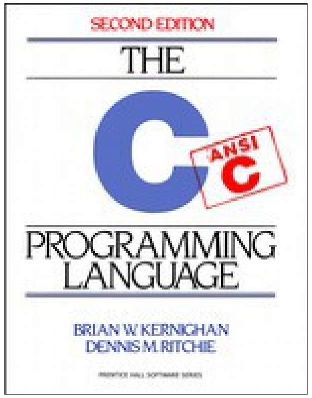


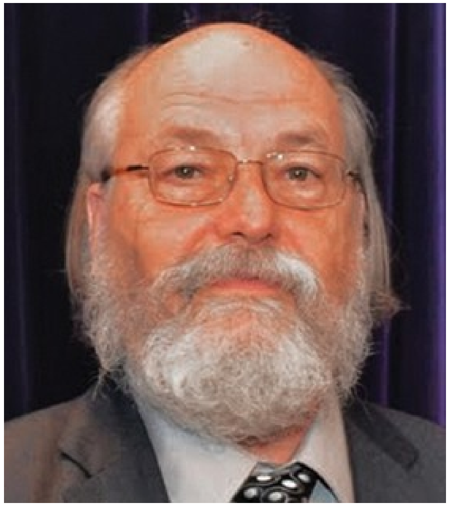
C Introduction

Address Spaces and Pointers

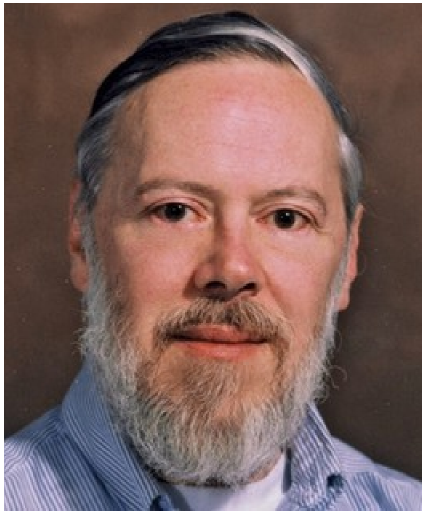
The C Language

- Systems language originally developed for Unix
- Imperative, compiled language with static typing
- “High level” at the time; now considered low-level
- Provides pointers and allows direct access to memory
- Many compilers and standards: we’ll use GNU and C99

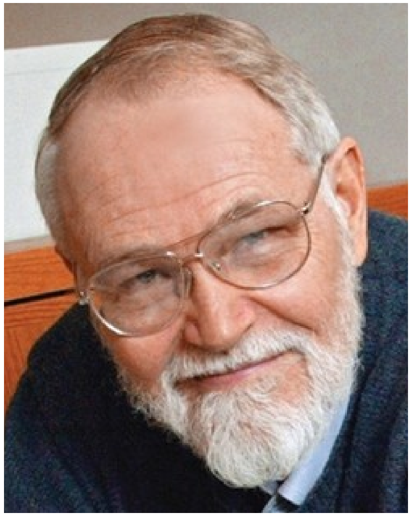




Ken Thompson



Dennis Ritchie



Brian Kernighan

Compilation

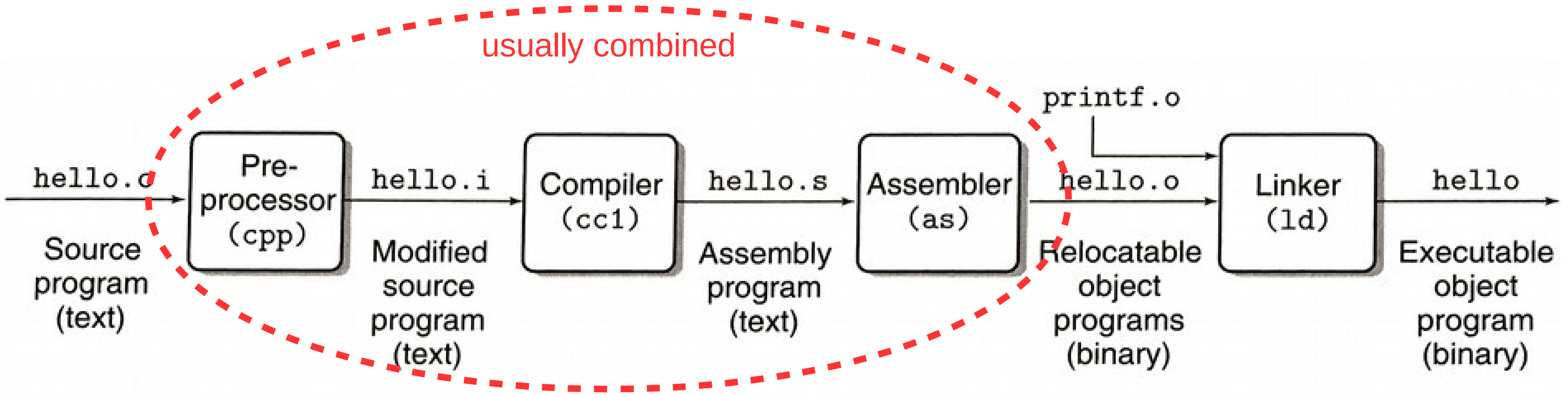


Figure 1.3 The compilation system.

```
linux> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

Makefiles

- The compilation process is usually streamlined using a build system: Make, CMake, Ant, Maven
- In this class, we will use Make
- Provide a “Makefile” that contains targets, dependencies, and build commands
- Example Makefile:

target → dependency

```
hello: hello.c
       gcc -g -O0 -o hello hello.c
```

build command

Hello, World

- How is this different from Java?

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Similarities to Java

- Semicolons!
- Comments
- Basic types: int, char, float, double
- Loops: do, while, for
- Switch statements
  - Parameter must be integer
- Method/function definitions
- Fixed-sized arrays

Differences from Java

- Additional fixed-width types: uint32_t, int32_t, size_t (in stdint.h)
- Booleans are “bool” (in stdbool.h)
  - Actually integers: 0 is “false”, anything else is “true”
- No objects (but it does have structs)
- No built-in string type (C strings are just arrays of chars)
- No built-in exceptions
- Different I/O functions: printf, fgets, scanf (in stdio.h)
- No standard container framework
- Functions must be declared before use (declaration vs. definition)
- Interface (.h) vs implementation (.c)
- Preprocessor macros (#include, #define)

Pointers

- **A pointer is a variable that contains a memory address**
- Declared with “*” operator
  - `int *p;`
  - `int **p;     // yes, this works`
- Often initialized using the “&” operator (“address of”)
  - `int x;`
  - `p = &x;`
- Dereferenced with “*” operator (“follow the pointer”)
  - `*p = 7;`
- C does NOT check pointers before dereferencing them!
  - `int *p = 0; *p = 123;   // this will segfault!`

Process address spaces

- Static: created at load time, destroyed on exit (fixed size)
- Stack: created/destroyed at function calls (fixed size)
- Heap: allocated/deallocated with malloc/free (variable size)
    - Watch for memory leaks; you may not leak memory in this course!

Every process has its own address space

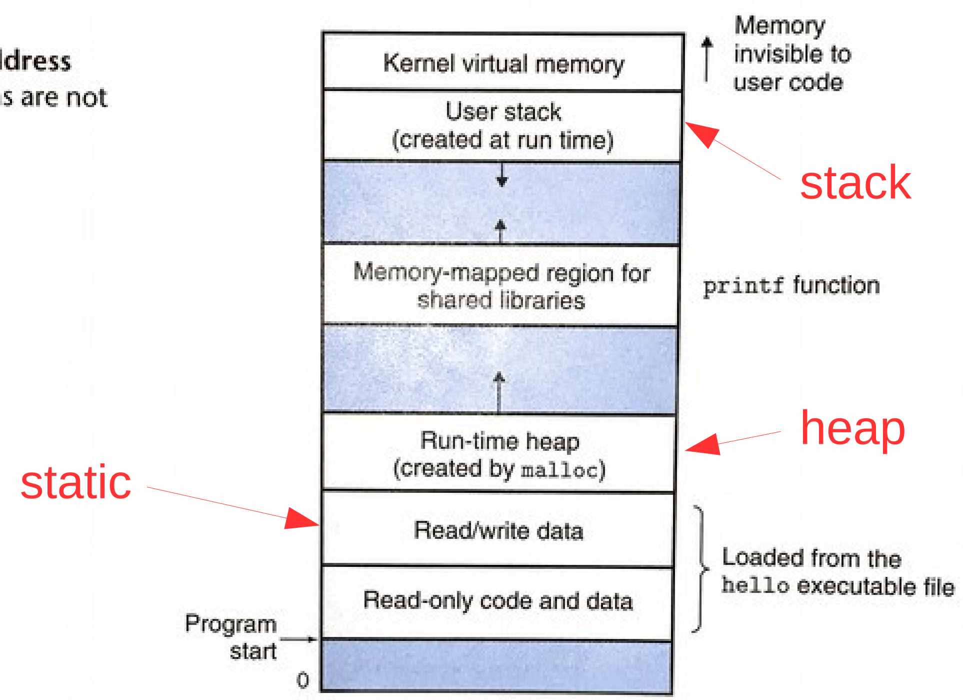


Figure 1.13
Process virtual address space. (The regions are not drawn to scale.)

Process address spaces

```
int global_var;

void foo()
{
    static int foo_st_var;
    int foo_var;
}

int main()
{
    int main_var;
    int *malloc_var = (int*)malloc(sizeof(int));
    foo();
    return 0;
}
```

For each of the following variables, classify them as static (C), stack (K), or heap (H):

- `global_var`
- `foo_st_var`
- `foo_var`
- `main_var`
- `malloc_var`

Does this program leak memory? If so, where?