More Shell, Commands & C

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

January 16, 2023

Today's lesson plan

1. Brief review of last week
2. Demos
   - Rupert's dotfiles
   - `ssh` configs
   - `top`/`htop`
   - Student-requested commands
3. Introduction to C

Section 1

Introduction to C

C at a Glance: What is it?

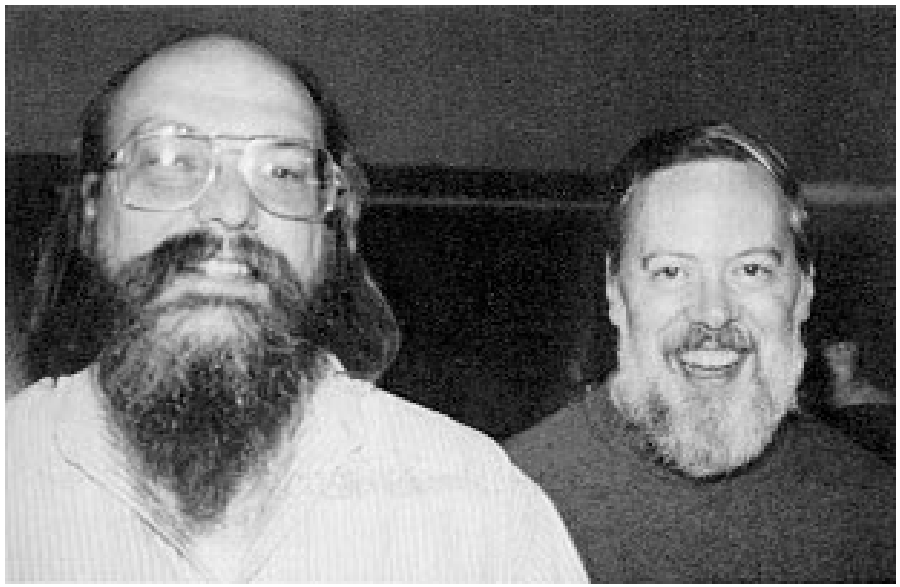

Figure 1: Thompson & Ritchie

- "High-level" systems programming language.
- Created by Dennis Ritchie in 1972.
- Later lead to C++, C#, Java, JavaScript, Perl, and much more.
- Many important applications are written in C, including Microsoft Windows, GNU/Linux, parts of MacOS, and many systems utilities.

C at a Glance: Why do we care?

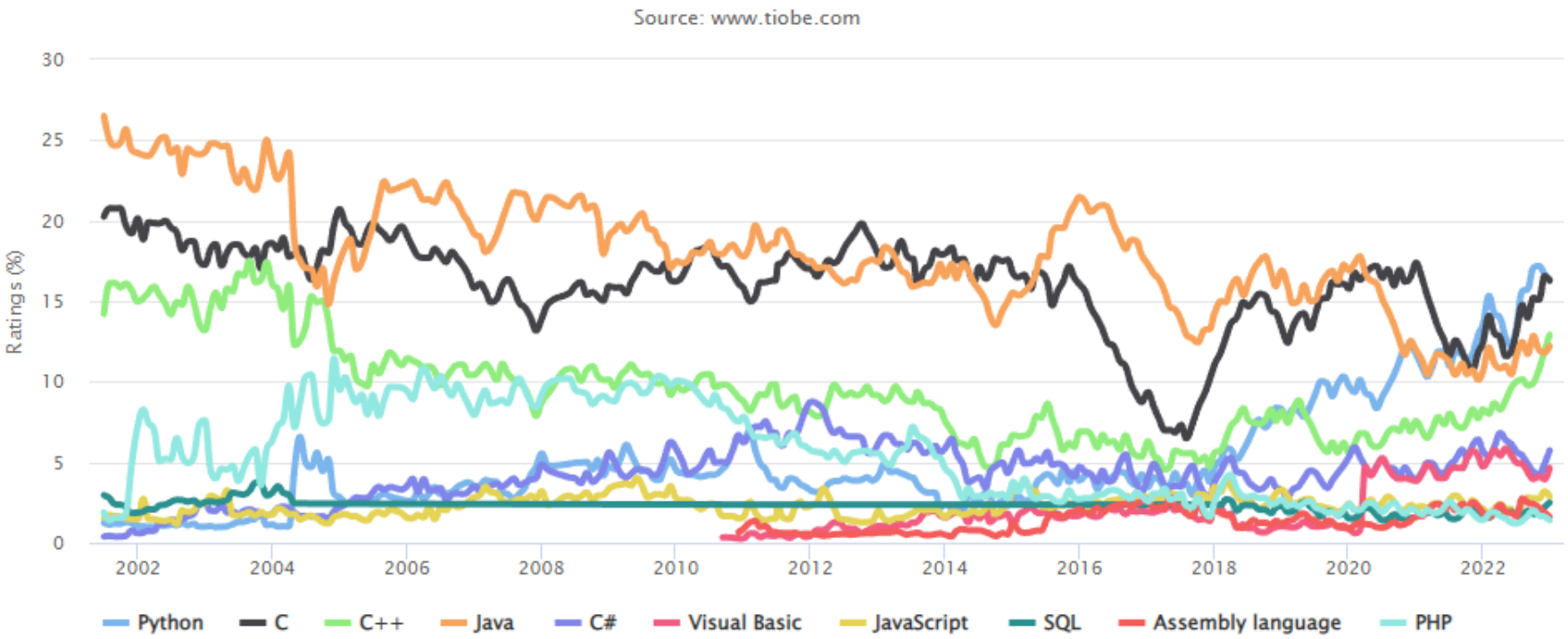

Figure 2: TIOBE Programming Community Index

Python, C, C++, Java, C# were the five most popular and recently growing languages! Notice any patterns?

C at a Glance: Why *should* we care?

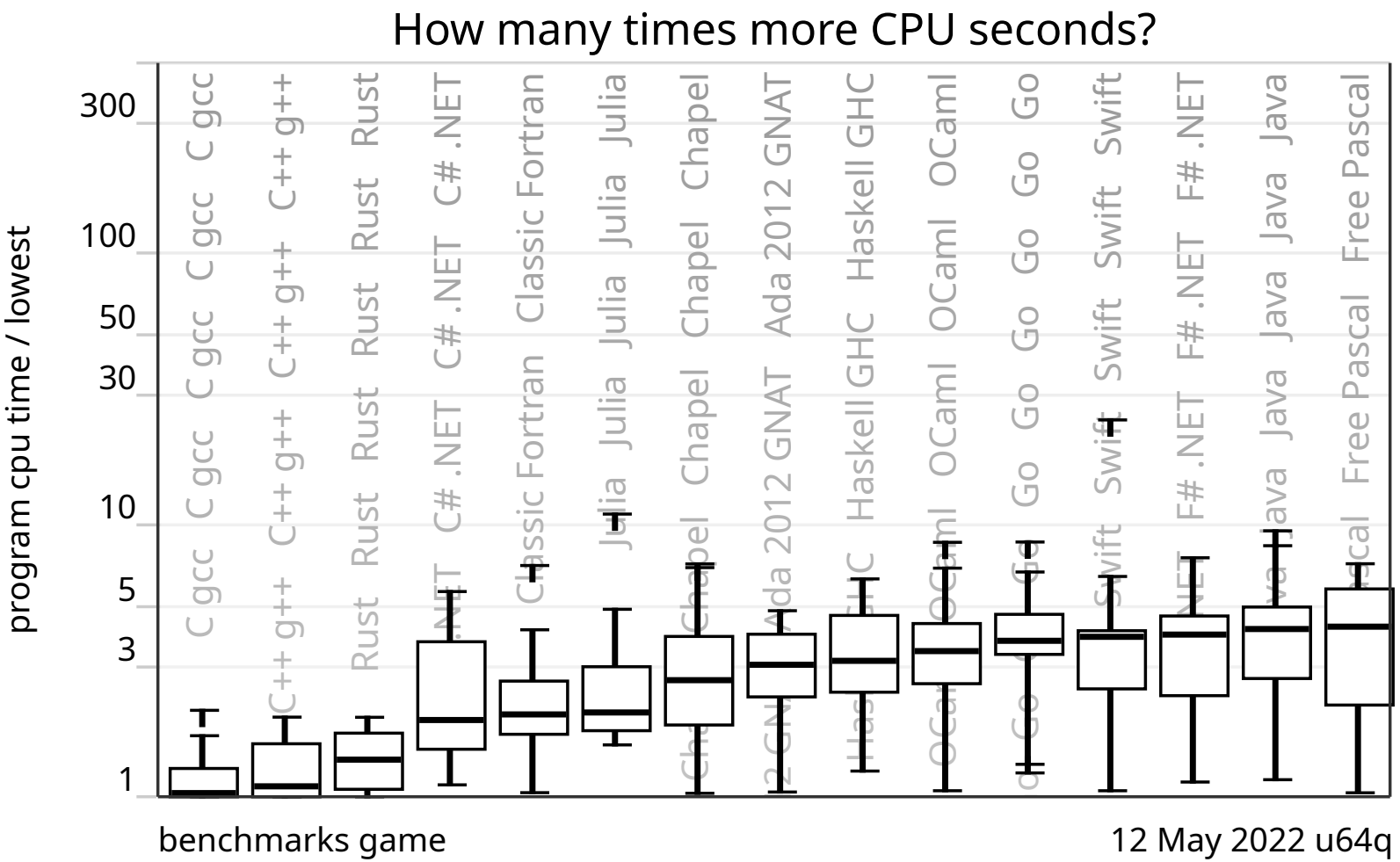

Figure 3: Benchmarks — Where's your favourite language?

C at a Glance: Why *should* we care?

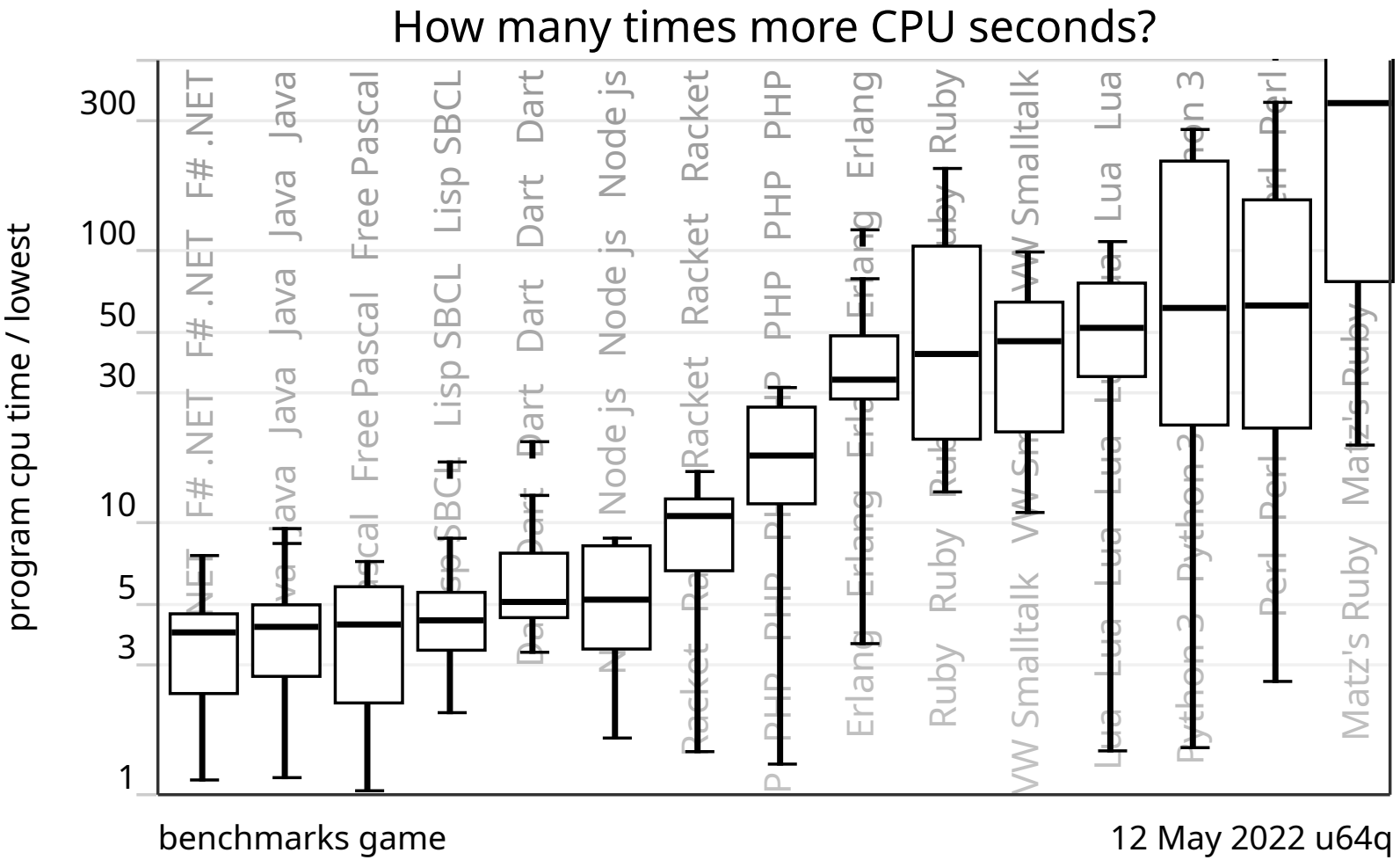

Figure 4: Benchmarks — Did you find it yet?

C at a Glance: Other advantages

- Greater control over memory/systems.
- Compiles to high performance code.
- Low systems requirements to develop on.
- Legacy and widespread use.

C at a Glance: What the future holds

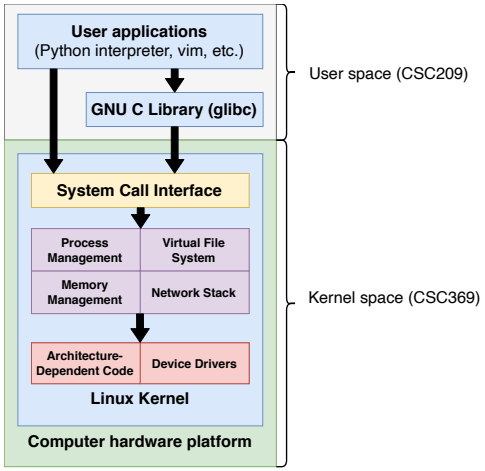

Figure 5: The Software Stack

- C/C++ are still the most widely used systems languages.
- Improvements to the computing stack call for better systems.
- Many modern high-level libraries are still implemented in C/C++.

Compiling C: GCC & LLVM

- A single language might have multiple implementations.
- GNU Compiler Collection provides `gcc`/`g++` to compile C/C++.
- LLVM Compiler Infrastructure provides `clang`/`clang++`.
  - Slower than `gcc`/`g++` but more flexible/modular.
  - Heavily used in developing modern languages.

Compiling C: GCC Pipeline

All input files (the last arguments) flow through the pipeline (depending on options) up to (and including)...

1. Preprocessing (`gcc -E`)
   - removes comments.
   - expands compiler directives (`#includes` statements).
   - expands macros.
2. Compilation (`gcc -S`) generates assembly code (`*.s`/`*.asm`).
3. Assembly (`gcc -c`) generates binary/machine code objects (`*.o`).
4. Linking (`gcc`) consolidates objects into a single application.
   - defaults to `a.out`, but you can use `-o <output>` to specify.

C: Source Files

File Type	Extension
C source (function implementations)	`.c`
C header (shared macros/declarations)	`.h`
C++ source/headers*	`.c*`, `.h*`

C++ is a superset of C $\implies$ all C code is valid C++.

Example (Makefile)

`github.com/rhubarbwu/Game-of-Life/blob/master/Makefile`

Compiling C: Macros & Header Files

- Macros are preprocessor substitutions; they are `#define`'d.
- You can `#include` headers from the system libraries or local directory. They typically contain function/variable/macro declarations and other `#include` statements.

```
#include <system.h>
#include "local.h"
#define MAX 100
void function(unsigned *array, char **strings);
```

Examples from `github.com/rhubarbwu/Game-of-Life`

- macros & headers: `field-cuda.cpp`
- multiline macros: `rules.h`
- lots of headers: `gol-cuda.cpp`

Syntax & Semantics of C: Hello, world!

```c
#include <stdio.h>

int main(int argc, char**argv){
    char end = '!';
    printf("Hello, world%c\n", end);
    return 0;
}
```

- `#include <stdio.h>` is like "importing" a library at preprocessing.
- `printf` stands for "print with formatting"
- `argc` counts the number of arguments.
- `argv` is an array of arguments...
- Return codes are explicit.

Compiling

```
$ gcc -o hello hello.c // -o to specify an output path
```

Syntax & Semantics of C: Types

Type	(Min) Bits	Format (unsigned format)
`char`†	8	`%c`/`%hhi`
`short`†	16	`%hi`/`%hd` (`%hu`)
`int`†	(16) 32	`%i`/`%d` (`%u`)
`long`†	(32) 64	`%li`/`%ld` (`%lu`)
`long long`†	64	`%lli`/`%lld` (`%llu`)
float	32*	`%f` and others
double	64*	`%lf` and others
long double	128*	`%Lf` and others

†- unsigned/signed variants exist

* - in general

Read more: `en.wikipedia.org/wiki/C_data_types`

Syntax & Semantics of C: Operators

C supports C-style operators:

- Arithmetic operators (`a+b`, `a-b`, `a*b`, `a/b`, `a%b`).
- Bitwise operators (`~x`, `x|y`, `x&y`, `x^y`, `x<<y`, `x>>y`).
  - Right bit-shift >> is logical on unsigned numbers and arithmetic on signed numbers.
- Comparators (==, !=, >, <, >=, <=)
- Assignment operators (=, -=, *=, /=, %=)
  - Bitwise counterparts (`x|=y`, `x&=y`, `x^=y`, `x<<=y`, `x>>=y`).
- Increment/decrement operators (`x++`, `y--`, `++w`, `--z`)
- Logical operators (`!p`, `p||q`, `p&&q`).
- Pointer operators.
  - `*ptr` dereferences the pointer `ptr` to get an object.
  - `&obj` takes the address of object `obj` to create a pointer.
  - More on pointers/arrays next week.

Syntax & Semantics of C: Variables

By default, variables are local to the scope; you can use `auto` to be explicit. For global variables, you can use macros, or better, `extern`.

```c
#include <stdio.h>
extern int x = 32;
int b = 8;
int main() {
   auto int a = 28;
   extern int b;
   printf("auto variable : %d", a);
   printf("extern variables x and b : %d,%d", x, b);
   x = 15;
   printf("modified extern variable x : %d", x);
   return 0;
}
```

www.tutorialspoint.com/extern-keyword-in-c

Syntax & Semantics of C: Control (1)

C-style languages typically have the same `if`/`else` blocks and `for` loops.

```c
#include <stdio.h>
char *is_div_six(int x) {
    int r2 = x % 2, r3 = x % 3;
    char *result = "three";
    if (r2 && r3) result = "no";
    else if (r2 || r3) result = "six";
    else if (!r2) result = "two";
    return result;
}
int main(int argc, char **argv) {
    int x=0;
    for (int x=0; x<10000; x+=13)
        printf("%s\n", is_div_six(x));
    return 0;
}
```

Syntax & Semantics of C: Control (2)

And also `switch` and `while`.

```c
#include <stdio.h>
void is_div_ten(int x) {
    switch (x % 10) {
        case 5: printf("five\n"); break;
        case 0: printf("ten\n");
        case 2: case 4: case 6: case 8:
            printf("two\n"); break;
        default: printf("no\n");
    }
}
int main(int argc, char **argv) {
    int x=0;
    while (x < 1000) { is_div_ten(x); x+=7; }
    return 0;
}
```

Syntax & Semantics of C: Control (3)

Less common is `do-while` which is `while` but checks conditions *after* each iteration.

```
int main(int argc, char **argv) {
    char result = '';
    int i = 0;
    do {
        i = i + 1;
        result = result + i;
    } while (i < argc);
}
```

Syntax & Semantics of C: Control (4)

Jump statements include `break`, `continue`, `return`. `goto` also exists but is bad practice.

```c
#include <stdio.h>

int main (int argc, char **argv) {
    if (argc == 0) return 1;
    int a = 10; // local variable
    LOOP:do { // do loop execution
        if (a == 7) { a *= 2; continue; }
        if (a == 15) { a += 1; goto LOOP; }
        printf("value of a: %d\n", a++);
        if (a >= 20) break;
    } while ( a < 20 );

    return 0;
}
```