

Tools Seminar

Week 3 - C/C++ Toolchain

Hongzheng Chen

Nov 29, 2019

- 1 Computer System Overview
- 2 C/C++ Programming Language
 - C/C++ Introduction
 - Compiling and Running C/C++
 - Auto Building
 - Debugging
 - Testing
 - Profiling
- 3 Summary

1

Computer System Overview

Some Introductory Books & Courses You Need to Know

UC Berkeley CS Major Lower Division Degree Requirements (61 Series)

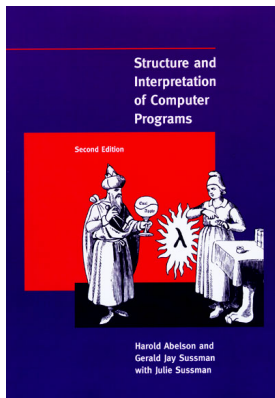
- 61A: [Structure and Interpretation of Computer Programs](#)
- 61B: [Data Structures](#)
- 61C: [Great Ideas in Computer Architecture \(Machine Structures\)](#)

MIT EECS General Institute Requirements

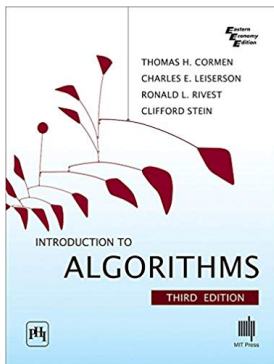
- 6.001: [Structure and Interpretation of Computer Programs](#)
→ 6.0001 Python
- 6.006: [Introduction to Algorithms](#)
- 6.004: [Computation Structures](#)

Some Introductory Books & Courses You Need to Know

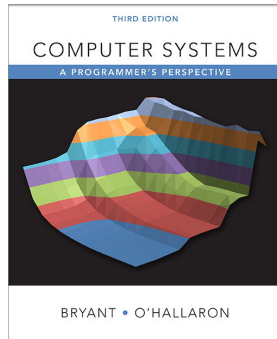
SICP



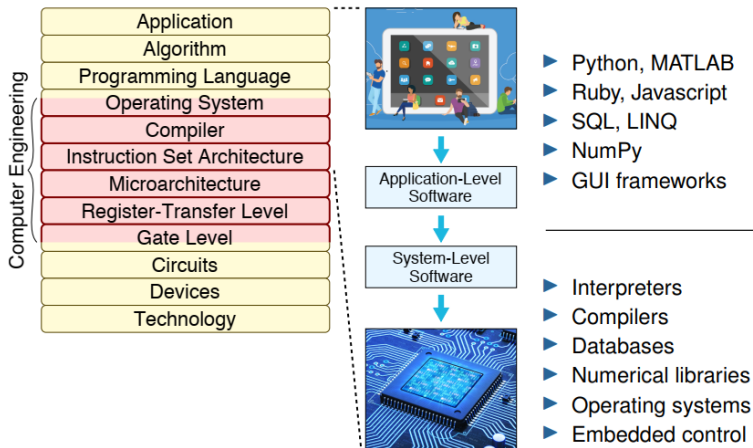
CLRS



★CS:APP



Computer Systems Programming is Diverse



Source: Christopher Batten, [Cornell ECE 2400: Computer Systems Programming](#), Fall 2019

2

C/C++ Programming Language

2.1

C/C++ Introduction

TIOBE Ranking

Nov 2019 Ranking

Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.246%	-0.50%
2	2		C	16.037%	+1.64%
3	4	▲	Python	9.842%	+2.16%
4	3	▼	C++	5.605%	-2.68%
5	6	▲	C#	4.316%	+0.36%
6	5	▼	Visual Basic .NET	4.229%	-2.26%
7	7		JavaScript	1.929%	-0.73%
8	8		PHP	1.720%	-0.66%
9	9		SQL	1.690%	-0.15%
10	12	▲	Swift	1.653%	+0.20%

Source: <https://www.tiobe.com/tiobe-index/>

C Programming Language

Actually in 9102, most of the CS top schools in US use Python as their introductory language.

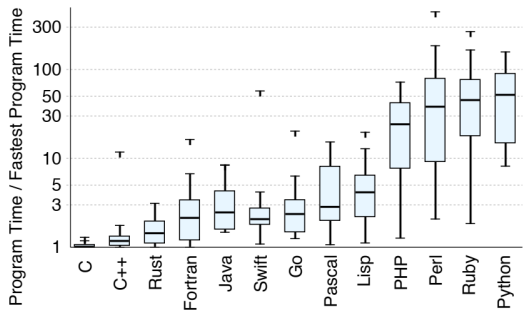
C Programming Language

Actually in 9102, most of the CS top schools in US use Python as their introductory language.

So why we still use C?

C Programming Language

- Speed: Very close to hardware, extremely fast
→ *High performance computing (HPC)
 - Parallel computing: OpenMP, MPI
 - Specific accelerator: cuda, Verilog

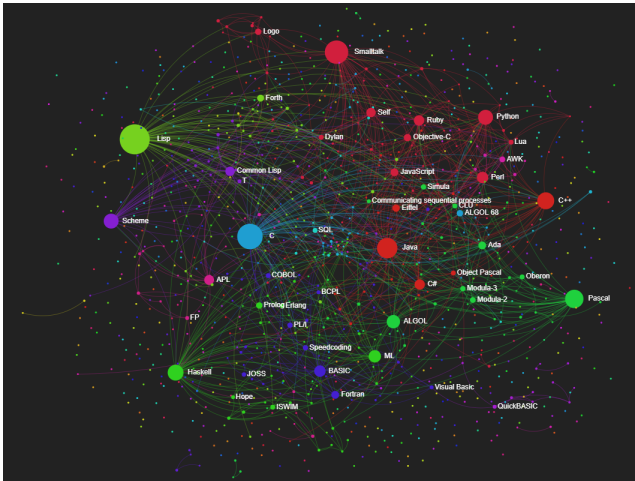


The Computer Language Benchmarks Game

C Programming Language

- Portability:
 - OS
 - Networking
 - Embedded systems
- Stability: Static type & rigorous spec
 - Compiler / Interpreter

C influences lots of PLs which all has C-like grammar (Java, C#)



◀ ▶ ◀ ▶ ▶ 🔍 ↺

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

- Object-Oriented Programming (OOP) support

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

- Object-Oriented Programming (OOP) support
 - **Abstraction & Encapsulation**

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

- Object-Oriented Programming (OOP) support
 - **Abstraction & Encapsulation**
 - Enable to distribute works in a large project

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

- Object-Oriented Programming (OOP) support
 - **Abstraction & Encapsulation**
 - Enable to distribute works in a large project
- Modern language features

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

- Object-Oriented Programming (OOP) support
 - **Abstraction & Encapsulation**
 - Enable to distribute works in a large project
- Modern language features
 - Templates and Generic Programming: Polymorphism

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

- Object-Oriented Programming (OOP) support
 - **Abstraction & Encapsulation**
 - Enable to distribute works in a large project
- Modern language features
 - Templates and Generic Programming: Polymorphism
 - C++11 standard

So what about C++?

- Superset of C, but not only C

C++ \neq C + STL!

- Object-Oriented Programming (OOP) support
 - **Abstraction & Encapsulation**
 - Enable to distribute works in a large project
- Modern language features
 - Templates and Generic Programming: Polymorphism
 - C++11 standard
- Also lightweight, fast, and high-performance

2.2

Compiling and Running C/C++

Some Compilers for C/C++

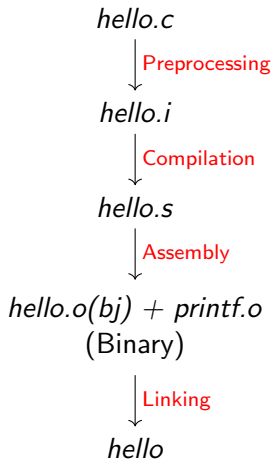
- gcc/g++: GNU Compiler Collection
 - [MinGW](#) (Minimalist GNU for Windows)
- clang/clang++
 - Low Level Virtual Machine (LLVM) [UIUC, CGO'04]
 - → commonly used for building your own compiler for DSL
- Visual Studio C++
 - Mainly for Windows applications development
- icc/[icpc](#): Intel C/C++ Compiler
 - Extremely high-performance when compiling to Intel CPU architecture
 - Need a student account

Four stages of Compilation

```
// hello.c
#include <stdio.h>

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

```
; hello.s
.LC0:
.string "Hello world!"
main:
; protect state
push    rbp
mov     rbp, rsp
; call function
mov     edi, OFFSET FLAT:.LC0
call    puts
; restore state & return
mov     eax, 0
pop     rbp
ret
```



Preprocessing

Try this:

```
// student.c
#include "name.txt"
#include "number.txt"
```

Use C preprocessor (cpp) to generate output

```
$ cpp student.c -o student.out
$ cat student.out
```

Preprocessing

You can include several times

```
#include "name.txt"  
#include "name.txt"  
#include "name.txt"  
#include "number.txt"  
#include "number.txt"  
#include "number.txt"
```

But it's very dangerous due to dependency → function redefinition

Preprocessing

Make sure the file is only included once:

```
#ifndef NAME_TXT
#define NAME_TXT
Alice
Bob
Carlo
#endif // NAME_TXT
```

Preprocessing

Make sure the file is only included once:

```
#ifndef NAME_TXT
#define NAME_TXT
Alice
Bob
Carlo
#endif // NAME_TXT
```

* Preprocessor macros are very ugly! C++20 proposes Modules (import)

Compilation & Assembly

```
$ gcc avg.c -c -o avg.o  
$ objdump -dC avg.o
```

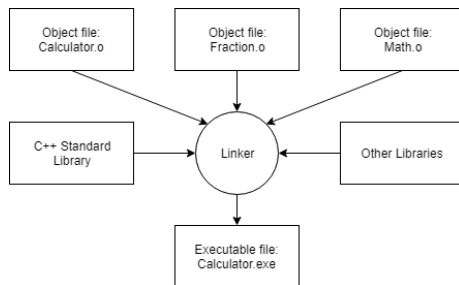
- Online editor and compiler: [Repl.it](#)
- Online assembly code generation: [GodBolt](#)

Ref: [ECE 2400](#)

Linking

```
$ gcc avg.o -o avg
```

- Static library: .a (Linux), .lib (Windows)
- Dynamic library: .so (Linux), .dll (Windows)



Source: [LearnCpp](#)

Compilation Flags

When you configure C environment in VS Code, you need to add them

- `-o`: output to file
- `-O`: `-O0`, `-O1`, `-O2`, `-O3` (See [gcc optimization options](#))
- `-Wall`: with all warnings
- `-Werror`: all warnings to errors
- `-D[FLAG]` or `-D[FLAG]=VALUE`: pass preprocessor flag `#if FLAG ...`
- `-std=c++11`: standard
- `-I`: `[/path/to/header-files]` (`.h`, `.hpp`)
- `-L`: `[/path/to/shared-libraries]`
- `-l`: Links to shared library or shared object (`.so`, `.dll`)

Ref: <https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html>

2.3

Auto Building

Existing Problems

- Programs with hard dependency, say projects with 100k LoC
- Intractable to enter a bunch of commands
- Some unmodified files will be regenerated

Existing Problems

- Programs with hard dependency, say projects with 100k LoC
- Intractable to enter a bunch of commands
- Some unmodified files will be regenerated

We need **auto compilation** tools!

Makefile

Put previous commands together

```
avg: avg.o
    gcc avg.o -o avg

avg.o:
    gcc avg.c -c -o avg.o
```

Just type make!

Makefile - Grammar

```
target : prerequisite0 prerequisite1 prerequisite2  
<TAB>command
```

- First finish prerequisite
 - If pre does not change, skip it
 - If pre changes, re-generate it
- When all the pre are finished, do command
 - Can be ANY Linux command!

Makefile - Variables

```
CC = gcc
```

```
avg: avg.o  
    $(CC) avg.o -o avg
```

```
avg.o:  
    $(CC) avg.c $(FLAGS) -c -o avg.o
```

Makefile - Auto Variables

- `$@`: Target file
- `^`: all prerequisite
- `<`: The first prerequisite
- `%`: wildcard

```
CC = gcc
FLAGS = -O3

% : %.c
    $(CC) $(FLAGS) $< -o $@
```

Compared with

```
avg:
    $(CC) avg.c $(FLAGS) -o avg.o
```

Makefile - .PHONY

```
.PHONY : clean  
clean :  
    -rm edit $(objects)
```

- .PHONY: not associated with physical files; target is always out-of-date
- Put it at the back of Makefile

* For complete usage of Makefile,

please refer to <https://chhzh123.github.io/2019-02-24-makefile/>

CMake

```
cmake_minimum_required(VERSION 2.8)
enable_language(C)
add_executable( avg avg.c )
```

Need to install CMake first (apt-get install)

```
$ cmake .
$ make target
```

2.4

Debugging

Debugging

The GNU Debugger (GDB)¹

- `gcc -g`: Add debug flag
- `gdb program`: Debug program (from the shell)
- `run -v`: Run the loaded program with the parameters
- `bt`: Backtrace (in case the program crashed)
- `info registers`: Dump all registers
- `break 10`: Breakpoint at line 10

¹source: [Wiki](#)

2.5

Testing

CTest

```
#include <stdio.h>
#include "avg.h"
#include "utst.h"

int main()
{
    UTST_ASSERT_INT_EQ( avg( 10, 20 ), 15 );
    return 0;
}
```

CTest

```
cmake_minimum_required(VERSION 2.8)
enable_language(C)
enable_testing()

add_executable( avg avg.c )
add_executable( avg-test avg-test.c )
add_test( avg-test avg-test )
```

Type

```
$ make avg-test
$ make test
```

Change the number and regenerate it

Other Tools

- Google Unit Test: [Google Test](#)
- Code Coverage: `gcov`
- Continuous Integration: [Travis-CI](#), [Codecov.io](#)

2.6

Profiling

Profiling

Timing:

- `<time.h>`

```
void timing()
{
    time_t start, stop;
    start = time(NULL);
    foo(); // do something
    stop = time(NULL);
    printf("Time:%ld\n", (stop-start));
}
```

- `time <command>`

- **Real:** Wall clock time - time from start to finish of the call
- **User:** Amount of CPU time spent in user-mode code (outside the kernel)
- **Sys:** Amount of CPU time spent in the kernel

Profiling

- [perf](#) (Linux): CPU, cache, memory etc.
- [pcm](#) (Intel): CPU, cache, memory etc.
- [valgrind](#) [PLDI'07]: memory management

3

Summary

Week 2 - C/C++ Toolchain

- Compiling: gcc, clang
- Auto building: Makefile, CMake
- Debugging: gdb
- Testing: CTest
- Profiling: time, perf, valgrind