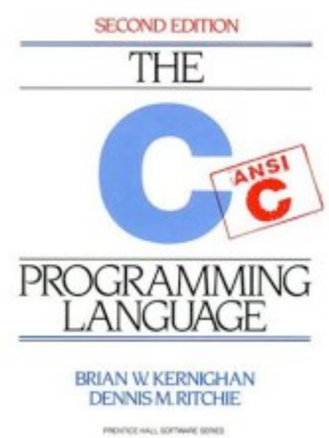
CS240



C Programming

Suresh Jagannathan

http://www.cs.purdue.edu/homes/cs240

C


```c
int a[]={0,1,2,3,4};

r((int)a, 0, 5);


void r(int a, int i, int E) {
  printf("%d\n",*((int*)a+i));
  if (++i < E) r(a, i, E);
}
```

Learning objectives

CS240 is your introduction to *low-level* programming

You will learn ...

- to solve problems computationally
- to design, implement, test, debug and evaluate complex algorithms
- about language-level and machine-level representations of control and data
- to use production-level tools (C, Unix, Emacs, gdb, make, shell...)

Our programming language of choice is C, because

- it is widely used in the industry

  complex systems (from web browsers to operating systems) written in C/C++/Objective-C
- it gives you fine-grain control over resources

  whereas Java hides everything from you
- it allows you to explore the interaction between software and hardware

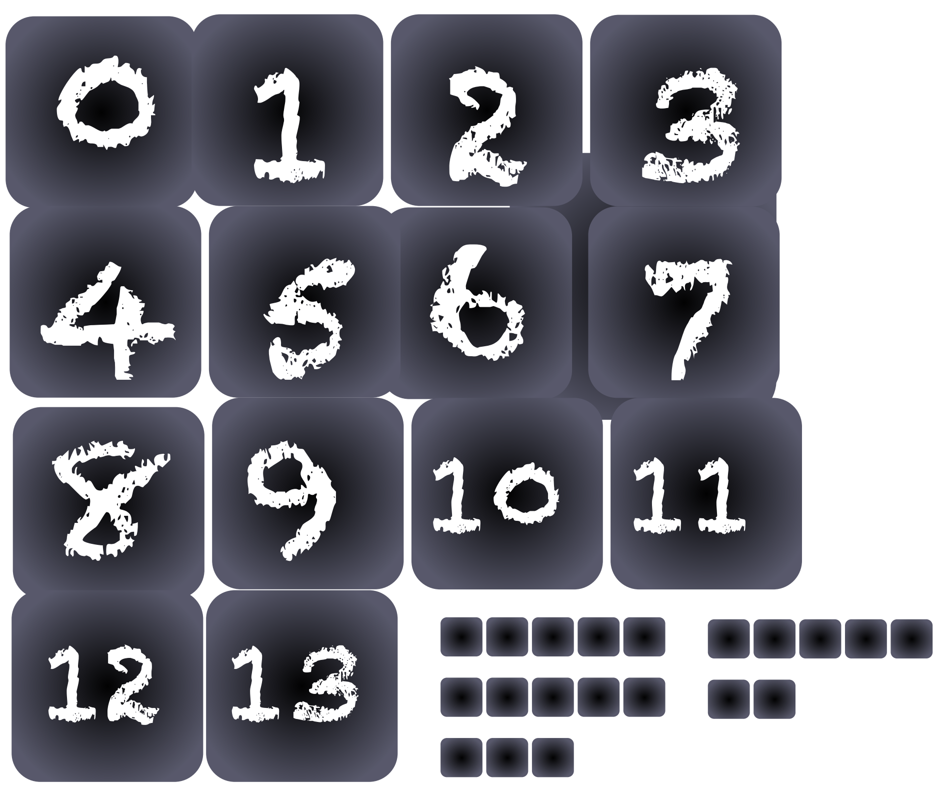
  C exposes architectural features

Workload

13 homeworks (programming assignments), ~13 take-home quizzes, ~7 in-class quizzes, 2 midterms, 1 final

Grade mix:

- homeworks and quizzes 45%, midterms 30%, final 25%



Piazza

Main forum for interacting with instructors, TAs, and other students

- Link: www.piazza.com/CS240

Used to ask and clarify any issues pertaining to the course:

- lectures
- labs and projects

Late policy

None.

Rationale: In a large class it is not possible to accommodate requests for extensions

All assignments are handed on time.

- A penalty of 5% per quarter hour will be charged to all assignments submitted after 9:00 pm on the day the assignment is due.
- No assignment will be accepted after 11:59 PM on the due date.

Academic integrity

Any case of cheating will handled by the Dean of students

You are encouraged to discuss problems and approaches but:

- Sharing solution is not allowed.
- Buying solutions is not allowed.
- Copying code from the internet is not allowed.
- Copying code from other students is not allowed.
- Copying partial code from other students is not allowed.

`http://homes.cerias.purdue.edu/~spaf/cpolicy.html`

Each year we catch students… they end up with an F… and a record… is it worth it?

Generative AI

- Useful as an additional resource
- *Not as a substitute student*
- Won't be able to use them on exams!

Attendance

Class attendance attendance is mandatory

- If you miss class, get someone else's notes...

Lab attendance is optional

- If you miss a lab, try to get in on another session during the same week

Rationale:

- *Slides may not be complete*
- *To prepare for exams, trust your notes and the book*

Office hours

Lab sessions are your first line of defense

- Ask as many questions as you can…

Piazza is your second best bet

- Can ask a question either in public or private

Quizzes

Short tests of your understanding of the lecture material

- Take-home: 24 hours to submit, 1 hour to complete once started
- In-class: 5-7 minutes

Quizzes cover material directly found in the book

It is your responsibility to read the book and ask questions ahead of class in case something is unclear

In-class quizzes will take place in the first five minutes of class and are usually unannounced

Questions

Use the following algorithm

- Ask on Piazza
- Ask TA at lab
- Ask Prof during class

Rationale: This focuses the interaction and ensure best use of your time

Regrading questions

- Regrading will only be done **the week following** release of the grade
- Contact the TA responsible for the assignment/exam (see syllabus)
- Midterm/Final issues are dealt by the Instructor, project/labs are dealt by your TA and can be escalated to the Faculty member

It is your responsibility to check your grades!

Questions

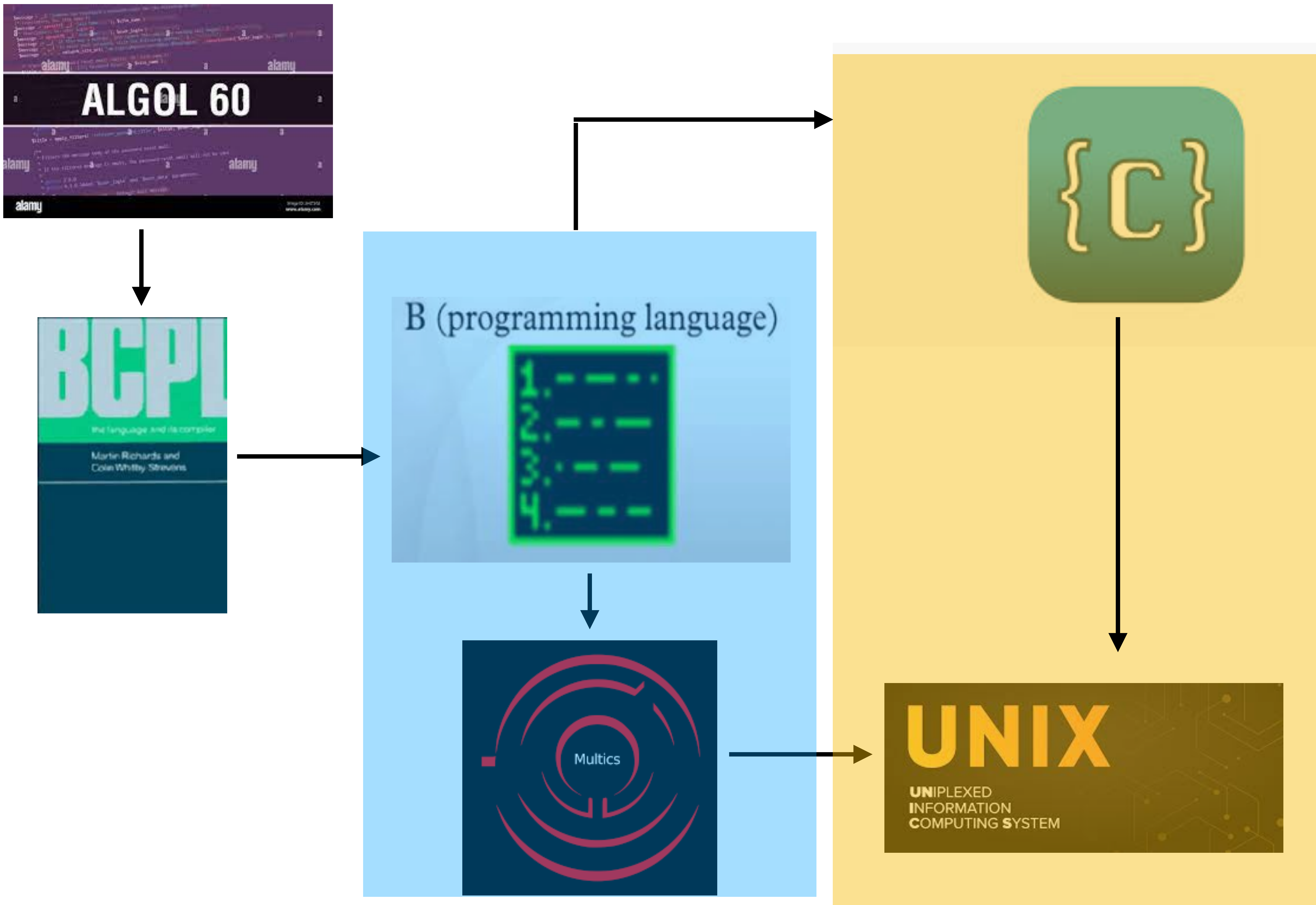
How to ask a question on Piazza:

- Read the book, slides, notes
- Describe the problem clearly, using the right terms
– Add output from compiler
- Add any other relevant information

Be polite and respectful of TAs' time and we'll do the same

Avoid anonymous questions...

History


ALGOL 60
BCPL
the language and its compiler
Martin Richards and
Colin Whitby Strevens
B (programming language)
{C}
Multics
UNIX
UNIPLEXED
INFORMATION
COMPUTING SYSTEM

Programming

Is programming a craft? an art? a science?

- There are many ways to express some task
- How do we know which is best?

We need to understand the tradeoffs…

- e.g. iteration vs. recursion

```
int a[]={0,1,2,3,4};
```

```
void r(int* a, int i,
int E) {
printf("%d\n",*(a+i));
  if (++i < E)
      r(a, i, E);
 }
```

```
for (int i=0; i<5; i++)
  printf("%d\n",a[i]);
```

Programming


```
int a[]={0,1,2,3,4};
for
  (int i=0;
   i<5;
   i++)
    printf(
      "%d\n",
      a[i]);
```

create a 5 element array

loop from zero to five

step by one

print i-th elem followed by newline

… and in Java


```
int[] a = new int[]{0,1,2,3,4};
for (int i=0; i<5; i++)
  System.out.println(a[i]);
```

```
int a[]={0,1,2,3,4};
for (int i=0; i<5; i++)
  printf("%d\n",a[i]);
```

Why C?

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Energy Efficiency across Programming Languages

How Do Energy, Time, and Memory Relate?

Rui Pereira
HASLab/INESC TEC
Universidade do Minho, Portugal
ruipereira@di.uminho.pt

Marco Couto
HASLab/INESC TEC
Universidade do Minho, Portugal
marco.l.couto@inesctec.pt

Francisco Ribeiro, Rui Rua
HASLab/INESC TEC
Universidade do Minho, Portugal
fribeiro@di.uminho.pt
rrua@di.uminho.pt

Jácome Cunha
NOVA LINCS, DI, FCT
Univ. Nova de Lisboa, Portugal
jacome@fct.unl.pt

João Paulo Fernandes
Release/LISP, CISUC
Universidade de Coimbra, Portugal
jpf@dei.uc.pt

João Saraiva
HASLab/INESC TEC
Universidade do Minho, Portugal
saraiva@di.uminho.pt

Abstract

This paper presents a study of the runtime, memory usage and energy consumption of twenty seven well-known software languages. We monitor the performance of such languages using ten different programming problems, expressed in each of the languages. Our results show interesting findings, such as, slower/faster languages consuming less/more energy, and how memory usage influences energy consumption. We show how to use our results to provide software engineers support to decide which language to use when energy efficiency is a concern.

CCS Concepts • **Software and its engineering** → **Software performance**; **General programming languages**;

Keywords Energy Efficiency, Programming Languages, Language Benchmarking, Green Software

ACM Reference Format:
Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136031

1 Introduction

Software language engineering provides powerful techniques and tools to design, implement and evolve software languages. Such techniques aim at improving programmers productivity - by incorporating advanced features in the language design, like for instance powerful modular and type systems - and at efficiently execute such software - by developing, for example, aggressive compiler optimizations. Indeed, most techniques were developed with the main goal of helping software developers in producing faster programs. In fact, in the last century *performance* in software languages was in almost all cases synonymous of *fast execution time* (embedded systems were probably the single exception).

In this century, this reality is quickly changing and software energy consumption is becoming a key concern for computer manufacturers, software language engineers, programmers, and even regular computer users. Nowadays, it is usual to see mobile phone users (which are powerful computers) avoiding using CPU intensive applications just to save battery/energy. While the concern on the computers' energy efficiency started by the hardware manufacturers, it quickly became a concern for software developers too [28]. In fact, this is a recent and intensive area of research where several techniques to analyze and optimize the energy consumption of software systems are being developed. Such techniques already provide knowledge on the energy efficiency of data structures [15, 27] and android language [25], the energy impact of different programming practices both in mobile [18, 22, 31] and desktop applications [26, 32], the energy efficiency of applications within the same scope [2, 17], or even on how to predict energy consumption in several software systems [4, 14], among several other works.

An interesting question that frequently arises in the software energy efficiency area is whether *a faster program is also an energy efficient program*, or not. If the answer is yes, then optimizing a program for speed also means optimizing it for energy, and this is exactly what the compiler construction community has been hardly doing since the very beginning of software languages. However, energy consumption does not depends only on execution time, as shown in the equation $E_{nergy} = T_{ime} \times P_{ower}$. In fact, there are several research works showing different results regarding

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10...$15.00
https://doi.org/10.1145/3136014.3136031

What conclusions can we draw from these results?

What are the characteristics of a programming language that influence these characteristics?

Getting started


```
#include <stdio.h>
int main() {
  printf("Hello World!\n");
}
```

```
public class Hello {
  public static void main(String[] s) {
    System.out.println("Hello World!”);
}
```

Compilation


```c
#include <stdio.h>
#define HELLO "Hello World!\n"
int main() {
  printf(HELLO);
}
```

Start with a human readable file containing your source program and possibly some references to libraries

Call the C compiler to obtain an executable file, which is a sequence of numbers that can run on the target hardware

```
gcc -std=c99 -c hello.c

gcc -o a.out hello.o


./a.out
```

```
33 % hexdump a.out
0000000 cf fa ed fe 07 00 00 01 03 00 00 80 02 00 00 00
0000010 0d 00 00 00 20 06 00 00 85 00 20 00 00 00 00 00
0000020 19 00 00 00 48 00 00 00 5f 5f 50 41 47 45 5a 45
0000030 52 4f 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000040 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
0000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000060 00 00 00 00 00 00 00 00 19 00 00 00 28 02 00 00
0000070 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
0000080 00 00 00 00 01 00 00 00 00 10 00 00 00 00 00 00
```

Compilation


```c
#include <stdio.h>
#define HELLO "Hello World!\n"
int main() {
  printf(HELLO);
}
```

```
gcc -std=c99 -E hello.c
```

One of the first steps is to expand macro definitions and include external declarations.

The compiler works in phases that transform the program into the executable one step at a time.

```
# 1 "h.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "h.c"
# 37 "/usr/include/i386/_types.h" 3 4
typedef signed char __int8_t;
typedef unsigned char __uint8_t;

...

# 238 "/usr/include/stdio.h" 3 4
int fprintf(FILE * , const char * , ...) __attribute__
);
void perror(const char *);
int printf(const char * , ...) __attribute__((__format
int puts(const char *);

...

# 500 "/usr/include/stdio.h" 2 3 4
# 2 "h.c" 2

int main() {
  printf("Hello World!\n");
}
```

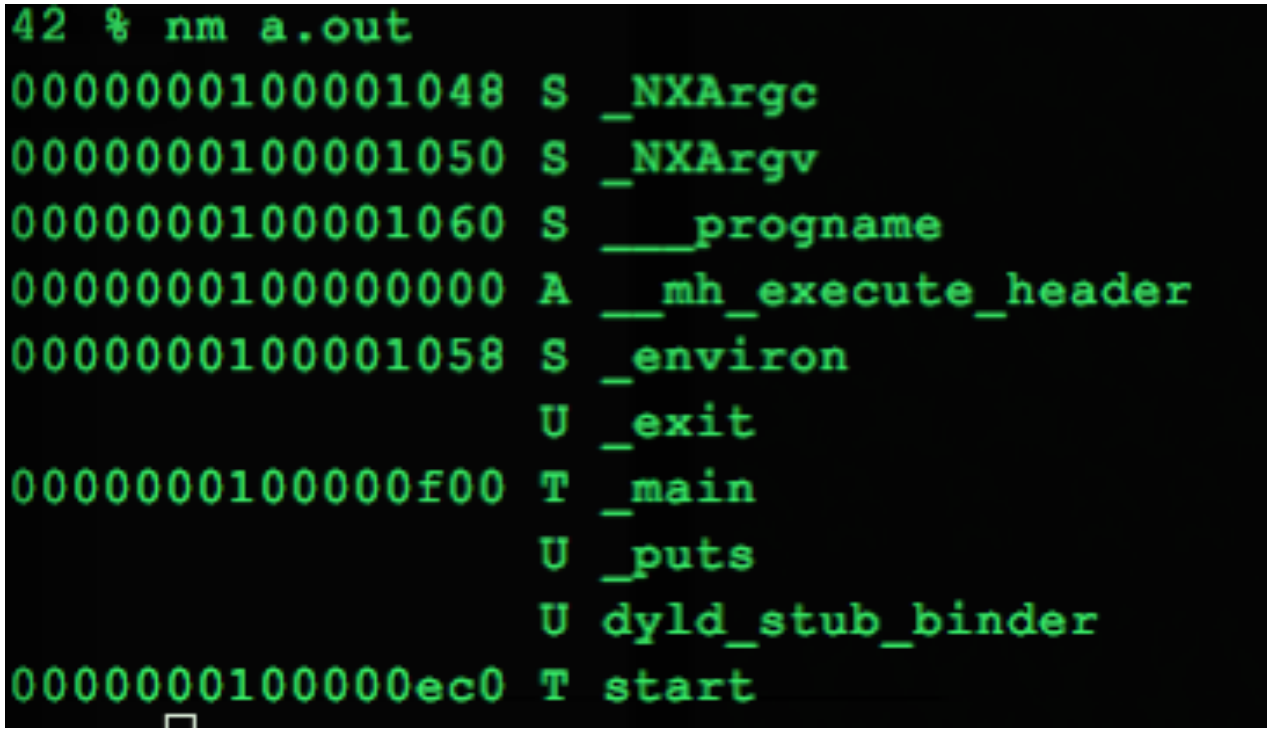
Compilation


```c
#include <stdio.h>
#define HELLO "Hello World!\n"
int main() {
  printf(HELLO);
}
```

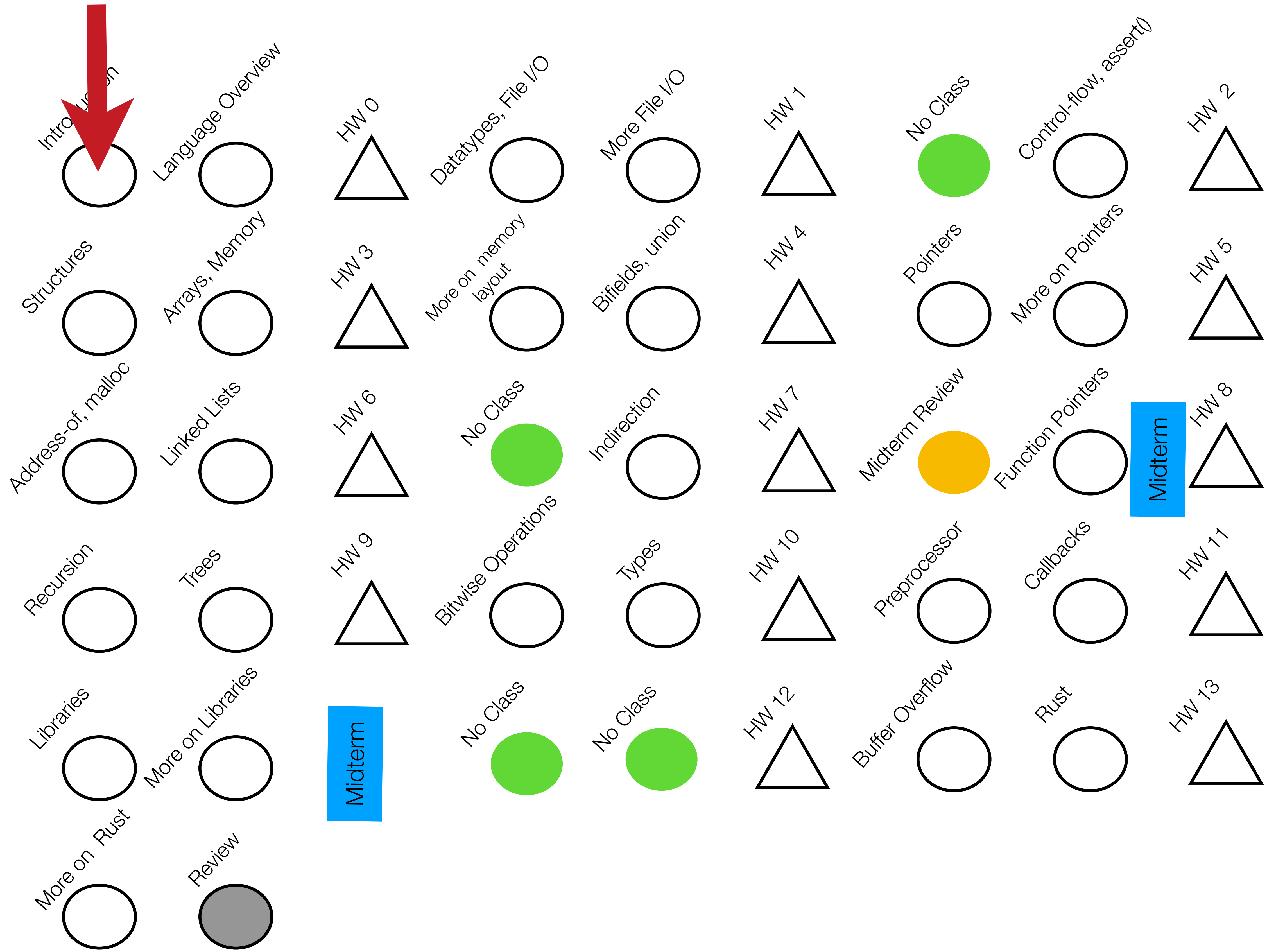
```
gcc -c hello.c
gcc -o a.out hello.o
./a.out
```

The compiler works in phases that transform the program into the executable one step at a time.

The last step links libraries, e.g. I/O, to create a stand alone executable binary

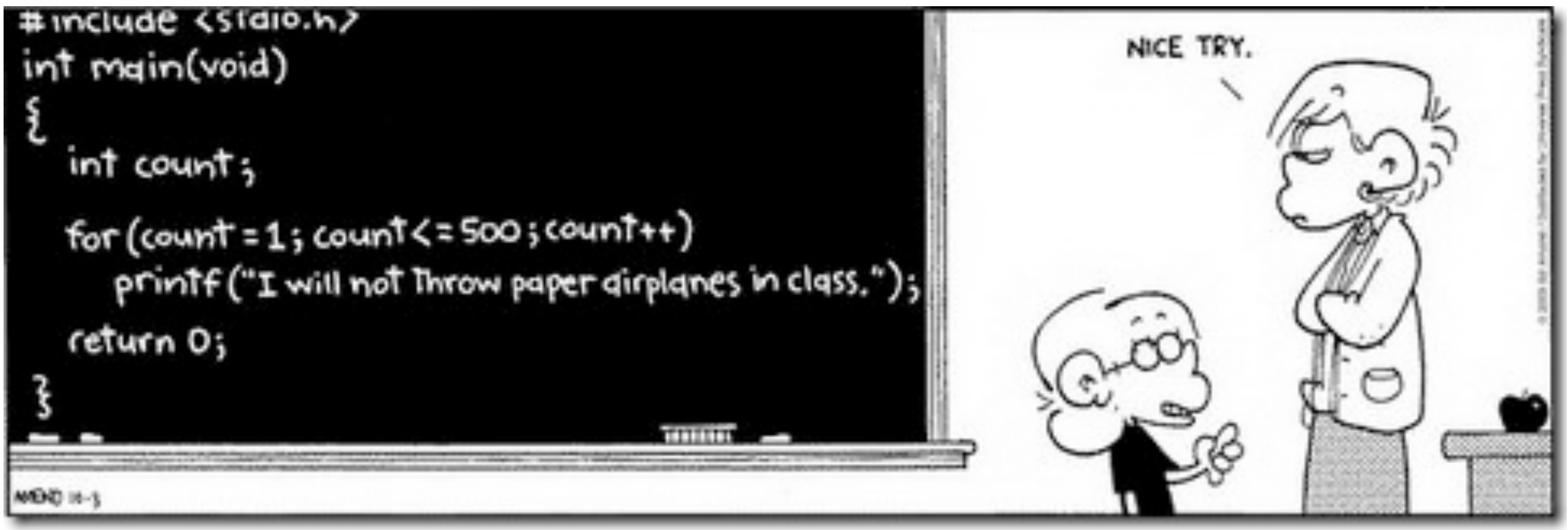


Roadmap



Language Overview
HW 0
Datatypes, File I/O
More File I/O
HW 1
No Class
Control-flow, assert()
HW 2
Structures
Arrays, Memory
HW 3
More on memory layout
Bitfields, union
HW 4
Pointers
More on Pointers
HW 5
Address-of, malloc
Linked Lists
HW 6
No Class
Indirection
HW 7
Midterm Review
Function Pointers
Midterm
HW 8
Recursion
Trees
HW 9
Bitwise Operations
Types
HW 10
Preprocessor
Callbacks
HW 11
Libraries
More on Libraries
Midterm
No Class
No Class
HW 12
Buffer Overflow
Rust
HW 13
More on Rust
Review

Goal?


#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count <= 500; count++)
printf("I will not Throw paper airplanes in class.");
return 0;
}
NICE TRY.