CS61C

Great Ideas
in
Computer Architecture
(a.k.a. Machine Structures)

Assistant Teaching
Professor
Lisa Yan

EECS department has approved ALL 58 concurrent enrollment students. Welcome!!!

Discussion starts today!

Introduction to the C Programming Language

(lecture demo code in Drive, instructions slide)

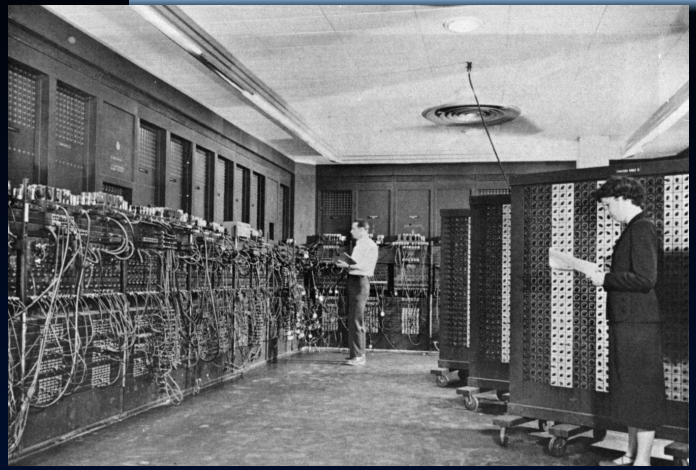
Computing in the News

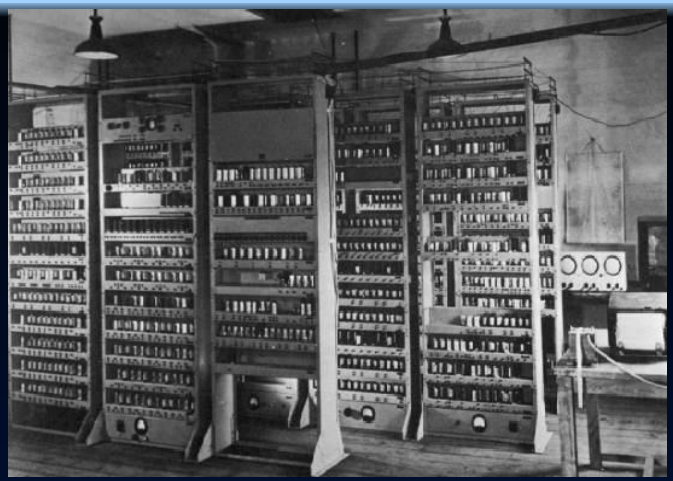
www.tiobe.com/tiobe-index

Jan 2026	Jan 2025	Change	Programming Language	Ratings	Change
1	1		Python	22.61%	-0.68%
2	4	▲	C	10.99%	+2.13%
3	3		Java	8.71%	-1.44%
4	2	▼	C++	8.67%	-1.62%
5	5		C#	7.39%	+2.94%

"The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular web sites Google, Amazon, Wikipedia, Bing and more than 20 others are used to calculate the ratings."

From ENIAC (1946) to EDSAC (1949)





ENIAC: First Electronic General-Purpose Computer

- Needed 2-3 days to set up new program
- Programmed with patch cords and switches
  - At that time & before, "computer" mostly referred to people who did calculations
  - Mostly women! (See Hidden Figures, 2016)

EDSAC: First General Stored-Program Computer

- Programs held as **numbers in memory**
  - Revolution! Program is also data!
- 35-bit binary Two's complement words

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

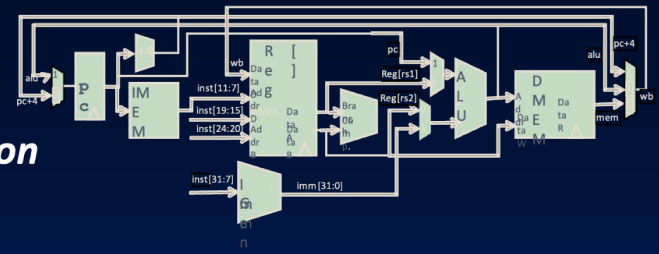
Assembler

Machine Language Program (RISC-V)

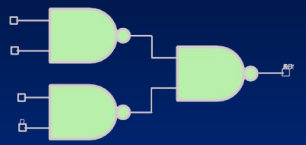
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Architecture Implementation

Hardware Architecture Description (e.g., block diagrams)



Logic Circuit Description (Circuit Schematic Diagrams)



Anything can be a ***number***—data, instructions, etc.

Agenda

Intro to C

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- [Extra] C Demo Setup

"Before this class, I (student) would say I am a solid C programmer"

Strongly disagree (never coded in C, and I don't know Java or C++) 0%

Mildly disagree (never coded in C, but I do know Java and/or C++) 0%

Neutral (I've coded a little in C) 0%

Mildly Agree (I've coded a fair bit in C) 0%

Strongly Agree (I've coded a lot in C) 0%

L03 Before this class, I (student) would say I am a solid C programmer

Strongly disagree (never coded in C, and I don't know Java nor C++)

Mildly disagree (never coded in C, but I do know Java and/or C++)

Neutral (I've coded a little in C)

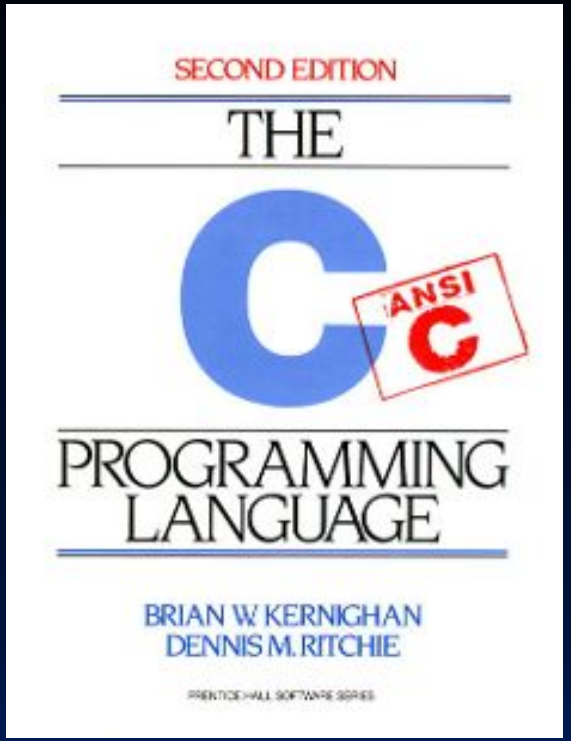
Mildly agree (I've coded a fair bit in C)

Strongly Agree (I wrote the C standard)

Why C? (1/2)

Kernighan and Ritchie (K&R):

> C is not a "very high-level" language, nor a "big" one, and is not specialized to any particular area of application. But its **absence of restrictions and its generality** make it more convenient and effective for many tasks than supposedly more powerful languages.



Why C? (2/2)

- We can write programs that allow us to exploit underlying features of the architecture
  - **memory management**, special instructions, parallelism
- Enabled first operating system not written in assembly language!
  - **UNIX** - A portable OS!

- C and derivatives (C++/Obj-C/C#) still one of the most popular programming languages after >40 years!
  - The C standard is constantly evolving! Latest standards are more memory-safe, legacy-compatible, and more C++ compatible

We're teaching the current C17 standard in this course (hive gcc)

🌟Disclaimer(s)

- **You will not learn how to fully code in C in these lectures!**
  You'll still need your C reference.
  - K&R is a must-have!
  - Course notes, Professor Emeritus Brian Harvey's helpful 61A transition notes

- CS61C will teach **key C concepts**: Pointers, Arrays, and Implications for **memory management**
  - Key security concept: Memory management in C is **unsafe**.

- Today we'll go at a more "leisurely" pace, but you should read more for HW

Agenda

Hello World

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- [Extra] C Demo Setup

Our very first C program

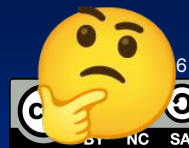
hello_world.c

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello World!\n");
  return 0;
}
```

HelloWorld.java

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

What do you notice? What's similar? What's different?



Our very first C program

hello_world.c

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello World!\n");
  return 0;
}
```

HelloWorld.java

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

- Import library uses #include
  - For `printf()`
- Main function
  - Unlike Java, not an object method!
    - C is **function-oriented**.
  - Function **return type is integer**, not void.
    - **0 on success**??
- Data types seem similar enough
  - But why are there two command-line arguments, `argc` and `argv`? (later)

Our very first C program

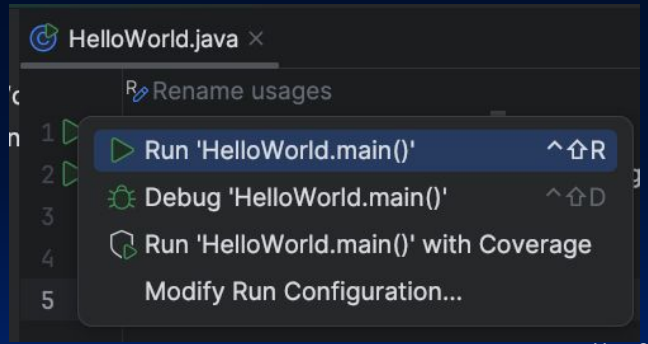
hello_world.c

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello World!\n");
  return 0;
}
```

(demo;
lecture code in Drive)

HelloWorld.java

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```



Our very first C program

hello_world.c

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello World!\n");
  return 0;
}
```

Command-line compiler, **gcc**:

```
~/lec02 $ gcc hello_world.c
~/lec02 $ ./a.out
Hello World!
```

gcc is a C compiler. It is a program that transforms C programs into executable machine code.

HelloWorld.java

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Agenda

Compilers (vs. Interpreters)

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- [Extra] C Demo Setup

Compilation, an Overview

- C compilers map C programs directly into **architecture-specific** machine code (string of 1s and 0s).
  - Differs mainly in **when** your program is converted to low-level machine instructions
    - Java: converts to architecture-independent bytecode, which is then compiled by a just-in-time compiler (JIT)
    - Python: converts to byte code at runtime ("interpreted")

- For C, compiling is colloquially the full process of using a compiler to translate C programs into executables.
  - But actually multiple steps: **compiling** .c files to .o files, automatic **assembling**, then **linking** the .o files into executables. (more later)

Command-line compiler, **gcc**:

```
~/lec02 $ gcc hello_world.c
~/lec02 $ ./a.out
Hello World!
```

Compilation: Advantages

- **Reasonable compilation time**: enhancements in compilation procedure (**Makefiles**) allow only modified files to be recompiled
  - (See Project 1, L02 Drive for example)
- Generally **much faster runtime performance** vs. Java for comparable code, because compilation optimizes for a given architecture.
  - (But these days, a lot of performance is in libraries)

Compilation: Disadvantages

- Compiled files, including the executable, are **architecture-specific**.
  - Depends on processor type (e.g., MIPS vs. x86 vs. RISC-V) and the operating system (e.g., Windows vs. Linux vs. MacOS)
  - Executable must be rebuilt on each new system.
    - "Porting your code" to a new architecture means copying the .c file, then recompiling using gcc

- "Change → Compile → Run [repeat]" iteration cycle can be slow during development.
  - but `make` only rebuilds changed pieces, and can compile in parallel: `make -j`
  - linker is sequential though → Amdahl's Law
  - (more later)

So…why C?

- C is much more “low level” than other languages you’ve seen…
  - Inherently unsafe, has terrible keyword conventions/scope, …
  - But all things considered, reasonable for teaching comp. architecture
  - In practice (and in 2025), you have many better options!
- If performance matters:
  - **Rust**, “C-but-safe”: By the time your C is (theoretically) correct w/all necessary checks it should be no faster than Rust
  - **Go**, “Concurrency”: Practical concurrent programming takes advantage of modern multi-core microprocessors
- If scientific computation matters:
  - **Python** has good **libraries** for accessing GPU-specific resources.
    - Python can call low-level C code to do work: Cython
    - Pytorch, a popular Python library for machine learning, uses C++
  - The Python interpreter is written in C.
  - **Spark** can manage many other machines in parallel. (more later)

Agenda

C vs. Java: Syntax

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- [Extra] C Demo Setup

C vs. Java

http://www.cs.princeton.edu/introcs/faq/c2java.html

C style: Use `snake_case`, NOT `camelCase`.

Demo Files

- Command-line arguments
  - `args.c`
- Use your braces
  - `braces.c`
- [at home] Number literals (hex, binary)
  - `hello_numbers.c`

C syntax: Command-line arguments

demo: args.c

- Combined, **argc** and **argv** get **main()** to accept arguments.
  - **argc**: # of strings on the command line
    - executable counts as one
  - **argv**: pointer to an array containing the arguments as strings
    - (More later re: pointers, arrays, strings.)

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Received %d args\n", argc);
  for(int i = 0; i < argc; i++) {
    printf("arg %d: %s\n",
           i, argv[i]);
  }
  return 0;
}
```

```
$ ./args 61C rocks
Received 3 args
arg 0: ./args
arg 1: 61C
arg 2: rocks
```

C: Use your braces

demo: braces.c

- **Single-line statements** in control flow body (if-else, for, while) **can omit** curly braces.
- ⚠ However, subsequent lines are considered outside of the body
  - Leads to many debugging errors (StackOverflow)
  - "Just because you can, doesn't mean you should")

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int x = 0;
    if (x == 0)
        printf("x is 0\n");
    if (x != 0) // careful!
        printf("x not 0 line 1\n");
        printf("x not 0 line 2\n");
    return 0;
}
```

C number literals

(from last time)

```c
#include <stdio.h>
int main() {
     const int N = 1234;
     printf("Decimal: %d\n",N);
     printf("Hex:     %x\n",N);
     printf("Octal:   %o\n",N);

     printf("Literals (not supported by all compilers):\n");
     printf("0x4d2         = %d (hex)\n", 0x4d2);
     printf("0b10011010010 = %d (binary)\n", 0b10011010010);
     printf("02322         = %d (octal, prefix 0 - zero)\n", 0x4d2);
     return 0;
}
```

Output

```
Decimal: 1234
Hex:     4d2
Octal:   2322
Literals (not supported by all compilers):
0x4d2         = 1234 (hex)
0b10011010010 = 1234 (binary)
02322         = 1234 (octal, prefix 0 - zero)
```

Format strings

Agenda

C Variables and Basic Types

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- [Extra] C Demo Setup

C Basic Types

Type	Description	Example
`int`	Signed integers (i.e., including negatives) Two's complement starting in C23	`0, 78, -217, 0x7337`
`unsigned int`	Unsigned Integers (i.e., non-negatives)	`0, 6, 35102`
`float`	Floating point decimal	`0.0, 3.14159, 6.02e23`
`double`	Equal or higher precision floating point	`0.0, 3.14159, 6.02e23`
`char`	Single character (ASCII)	`'a', 'D', '\n'`
`short`	Shorter int	`-7`
`long`	Longer int	`0, 78, -217, 301720971`
`long long`	Even longer int	`3170519272109251`
`bool`	(as of C23) 1 (true) or 0 (false) (pre-C23): #include <stdbool.h>	`true, false`

WikiBooks

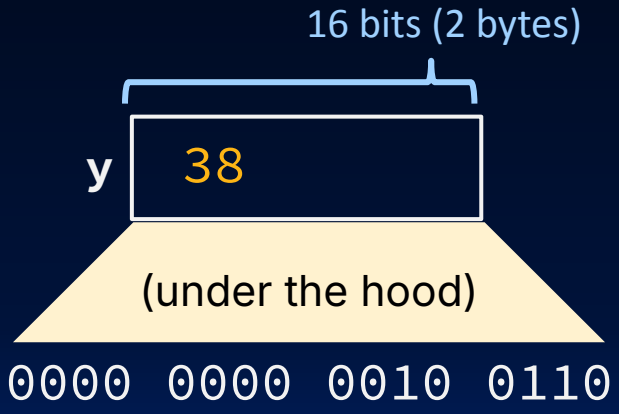
Why are variables typed in C?

- C variables are typed.
  - Types of ***variables*** can't change.
  - However, you can implicitly/explicitly **typecast** values.

```
uint16_t y = 38;
```

(unsigned 16-bit integer, see `stdint.h`)

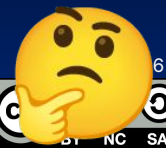
- A variable's type helps the compiler determine **how to translate the program to machine code** designed for the computer's architecture:
  - **How many bytes** the variable takes up in memory, and
  - **What operators** the variable supports, etc.



sizeof(int)

- `sizeof(arg)`: compile-time operator; gives size in **bytes** (of type or variable).
- Which of the following is true about the C `int` data type? Select all that apply. Notes:
  - $2^{15} = 32768$

A. Signed integer data type
B. Capable of containing **at least** the [−32767, +32767] range
C. `sizeof(int) = 2`
D. `sizeof(int) = 16`
E. `sizeof(int) = 4`
F. `sizeof(int) = 64`
G. None of the above



Integer types (C vs. Python vs. Java)

- The C standard does not define the absolute size of most integer types! [only relative sizes]
- C: `int` size **depends on computer**.
  - What integer type is most efficient w/processor

- TIP: **Explicitly declare how many bits for integers**!

```
#include <stdint.h>
```

  - `intN_t` and `uintN_t` for portable code!
    - `N` is in bits: 8, 16, 32, 64
  - Instead of `int x;`
    use `int32_t x;` (for some demos we'll still use `int`)

WikiBooks

A Cautionary Note: Undefined Behavior...

- A lot of C has "Undefined Behavior"
  - This means it is often unpredictable behavior
    - It will run one way on one computer...
    - But some other way on another
    - Or even just be different each time the program is executed!
- Often characterized as "**Heisenbugs**"
  - Bugs that seem random/hard to reproduce, and seem to disappear or change when debugging
  - Cf. "**Bohrbugs**" which are repeatable

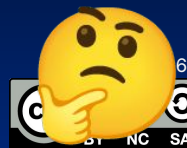
What will this program output?

demo: declaration.c

```
1  #include <stdio.h>
2  int main(int argc, char *argv[]) {
3      int x = 0;
4      int y;
5
6      printf("before: x=%d, y=%d, ",
               x, y);
7      x++;
8      y += x;
9      printf(" after: x=%d, y=%d\n",
               x, y);
10     return 0;
11 }
```

%d Placeholder for argument, where
d: format value as decimal numeral

A. `before: x=0, y=0, after: x=1, y=1`
B. `before: x=0, y=???, after: x=1, y=???`
C. undefined
D. compiletime error
E. runtime error
F. something else



What will this program output?

before: x=0, y=0 after: x=1, y=1 — 0

before: x=0, y=??? after: x=1, y=??? — 0

undefined — 0

compiletime error — 0

runtime error — 0

something else — 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Declaring a C Variable Does Not Also Initialize It

- ⚠ **Danger** ⚠: Bugs sometimes may only manifest after you've built other parts of your program.

- Variables are **not automatically initialized to default values**!
- If a variable is not initialized in its declaration, **it stores garbage**!
  - The contents are undefined…
  - …but C still lets you use uninitialized variables!

```
int x; // declaration
…
x = 42; // initialization
```


```
// OK
int y = 38;
```

The C `bool`: `true` or `false`?

see: bool.c

- Originally, the boolean was not a built-in type in C! Instead:
- FALSE:
  - **0** (integer, i.e., all bits are 0)
  - **NULL** (pointer) (more later)
- TRUE:
  - **Everything else!**
  - (Note: Same is true in Python)
- Nowadays:
  - `true` and `false` provided by `stdbool.h`.
  - Built-in type as of C23 (but we are using C17)

```
if (42) {
  printf("meaning of life\n");
}
```

A. `meaning of life`
B. (nothing)

More C Features

Read course notes and K&R:

- consts, enums, #define
- structs
- typedefs

And In Conclusion, …

- C chosen to exploit underlying features of HW
  - Pointers, arrays, implications for Mem management
  - We'll discuss this in a LOT more detail next time!!
- C compiled and linked to make executables
  - Pros (speed) and Cons (slow edit-compile cycle)
- C looks mostly like Java except
  - no OOP, ADTs defined through structs
  - 0 (and NULL) FALSE, all else TRUE (C99 onwards has `bool` types)
  - Use `intN_t` and `uintN_t` for portable code!
- **Uninitialized variables contain garbage**.
  - "Bohrbugs" (repeatable) vs "Heisenbugs" (random)

Agenda

[Extra] C Demo Setup

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- [Extra] C Demo Setup

gcc/gdb commands

```
# first download zip on local machine
$ scp -r lec03_code.zip hive3:.    # on local
$ unzip lec03_code.zip             # then on hive

# simple compile
$ gcc hello_world.c
$ ./a.out # why a.out?

# rename binary file
$ gcc -o hello_world hello_world.c

# generate debugging symbols for gdb
$ gcc -d hello_world.c
$ gdb a.out

# we wrote a makefile for you to more easily compile different files
$ make hello_world
$ ./hello_world
```