Computing in the News

www.tiobe.com/tiobe-index

Aug 2025	Aug 2024	Change	Programming Language	Ratings	Change
1	1		Python	26.14%	+8.10%
2	2		C++	9.18%	-0.86%
3	3		C	9.03%	-0.15%
4	4		Java	8.59%	-0.58%
5	5		C#	5.52%	-0.87%

"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."

CS61C

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

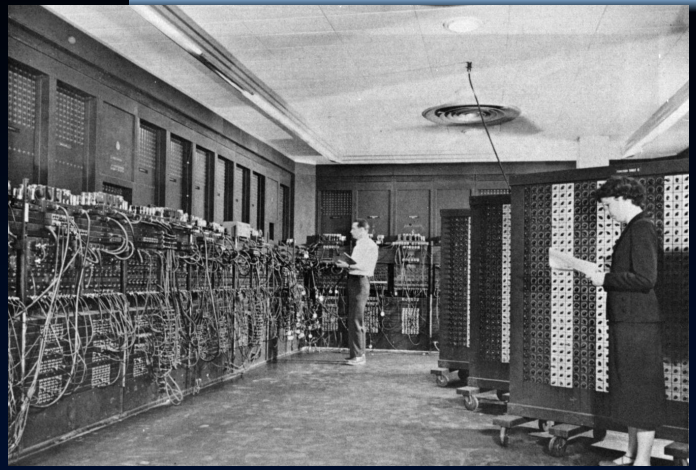
Teaching Professor
Dan Garcia

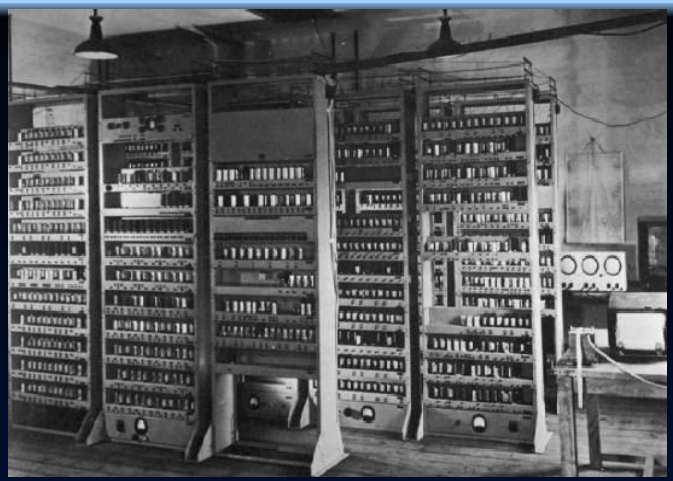
The dept allowed for class expansion to support ALL 83 concurrent enrollment students, welcome!!!

Introduction to the C Programming Language

(lecture demo code in Drive, instructions slide)

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

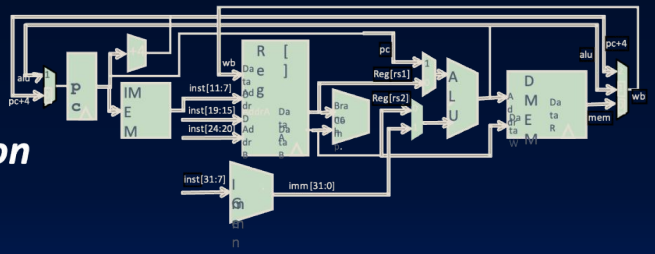
Machine Language Program (RISC-V)

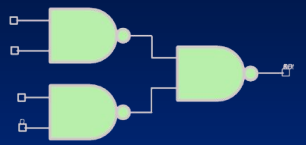
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

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

Architecture Implementation

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
- More C Features
- [Extra] Reference Slides

"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)

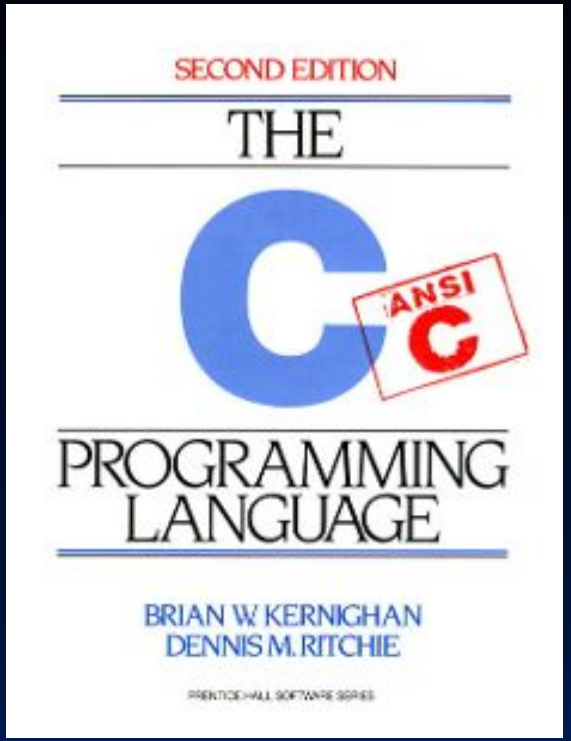
Mildly agree (I've coded a fair bit in C)

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 Language Is Constantly Evolving

- The C programming language standard has had several significant revisions since its inception in 1972.
  - Just like Python2 vs Python 3 – same language, but slightly different syntax/features

- From StackOverflow:
  - Pre-1989: K&R C (note K&R 1st ed 1978, 2nd ed 1988)
  - 1989/1990: ANSI C
  - 1999: C99
  - 2011: C11
  - 2017: C17
  - 2024: C23
    - New functions that are memory-safe, legacy-compatible
    - Syntax changes improve C++ compatibility.

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! More references at the end of these slides
  - Brian Harvey's helpful transition notes: https://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf

- CS61C will teach **key C concepts**: Pointers, Arrays, Implications for memory management
  - Key security concept: Memory management in C is **unsafe**.
  - If your CS61C program contains an error, it might not crash immediately but instead leave the program in an inconsistent (and often exploitable) state. Take CS161/CS162 for more!
- Today we'll go at a more leisurely pace, what we don't cover you should read as a reference.

Agenda

Hello World

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- More C Features
- [Extra] Reference Slides

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
- More C Features
- [Extra] Reference Slides

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
- More C Features
- [Extra] Reference Slides

C vs. Java

[for reference]

	C	Java
Language Paradigm	Function Oriented (programming unit: function)	Object Oriented (programming unit: Class = Abstract Data Type)
Compilation	`gcc hello.c` creates machine language code	`javac Hello.java` creates Java virtual machine language bytecode
Execution	`./a.out` loads, executes program	`java Hello` interprets bytecodes
Dynamic Memory Management	Manual (`malloc`, `free`) (more later)	Automatic garbage collection; `new` both allocates and initializes

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

C vs. Java: Syntax

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

	C	Java
Variable declaration	Typed declaration; declare before you use it	
Function declaration	Use curly braces. `void` means no return value	
Accessing a library	`#include <stdio.h>`	`import java.io.File;`
Comments	Multiline: `/* … */` , end-of-line: `// …`	
Similar operators	Arithmetic, Assignment: `+,-,*,/,%, =, +=,-=, …` Boolean: `!, &&, ||` Comparators: `==, !=, <, <=, >, >=` increment and decrement: `++` and `--` Bitwise operators: `<<,>>,~,&,|,^` Subgroup expressions: `()`	

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)
```

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

- Control flow (if-else, for, etc.) allow some omission of curly braces.
  - ...note function defs. must always have braces
- **Single-line statements can omit** curly braces. (same as in Java, but we didn't tell you)
- ⚠ However, subsequent lines are considered outside of the body
  - Leads to many debugging errors (StackOverflow)
  - "Just because you can, doesn't mean you should")

```
1  #include <stdio.h>
2  int main(int argc, char *argv[]) {
3      int x = 0;
4      if (x == 0)
5          printf("x is 0\n");
6      if (x != 0) // careful!
7          printf("x not 0 line 1\n");
8          printf("x not 0 line 2\n");
9      return 0;
10 }
```

Agenda

C Variables and Basic Types

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- More C Features
- [Extra] Reference Slides

C Basic Types

Type	Description	Example
`int`	Integer Numbers (including negatives)	`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	`'a', 'D', '\n'`
`short`	Shorter int	`-7`
`long`	Longer int	`0, 78, -217, 301720971`
`long long`	Even longer int	`3170519272109251`

https://en.wikibooks.org/wiki/C_Programming/Language_Reference#Table_of_data_types

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 integer types, other than char!
  - It only guarantees **relative** sizes:
    `sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)`
- C: `int` size **depends on computer**; what integer type is most efficient w/processor
- Instead **explicitly declare how many bits for integers**! (`stdint.h`)
  - Use `intN_t` and `uintN_t` for portable code! (`N` is in bits: 8,16,32,64)
  - E.g., `int32_t x;` instead of `int x;` (for demos we'll still use `int`)

Language	size of integer (in bits)
Python	>=32 bits (plain ints), infinite (long ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

https://en.wikibooks.org/wiki/C_Programming/Language_Reference#Table_of_data_types

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

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

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


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

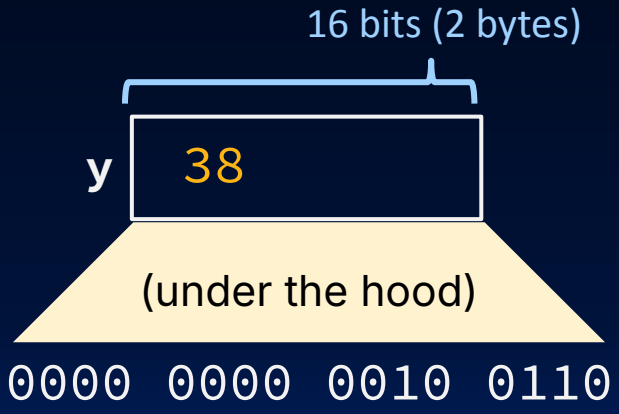
Why are variables typed in C?

- C variables are typed.
  - Types of ***variables*** can't change, e.g., y cannot later store a float in the same block.
  - However, you can **typecast *value*s** (more later)

- 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.

```
uint16_t y = 38;
```

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



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)

Agenda

More C Features

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- More C Features
- [Extra] Reference Slides

Consts, Enums, #define in C

- Constant, **`const`**, is assigned a typed value once in the declaration.
  - Value can't change during entire execution of program.

```
const float  golden_ratio = 1.618;
const int    days_in_week = 7;
const double the_law      = 2.99792458e8;
```

  - You can have a constant version of any of the standard C variable types.
- `#define PI (3.14159)` is a CPP (C Preprocessor) Macro.
  - Prior to compilation, preprocess by performing string replacement in the program based on all #define macros.
  - Replace all `PI` with `(3.14159)` □ In effect, makes **`PI`** a "constant"
- Enums: a group of related integer constants.

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};
```

CPP Macros: A Warning…

- You often see C preprocessor s defined to create small "functions"
  - But they aren't actual functions, instead it just changes the ***text*** of the program
  - In fact, all `#define` does is ***string replacement***
  - `#define min(X,Y) ((X)<(Y)?(X):(Y))`
- This can produce, umm, interesting errors with macros, if `foo(z)` has a side-effect
  - `next = min(w, foo(z));`
  - `next = ((w)<(foo(z))?(w):(foo(z)));`

More Typing: Typedefs and Structs

- `typedef` allows you to define new types.

```
typedef uint8_t BYTE;
BYTE b1, b2;
```

- `structs` are structured groups of variables, e.g.,

```
typedef struct {
    int length_in_seconds;
    int year_recorded;
} SONG;

SONG song1;
song1.length_in_seconds  =  213;
song1.year_recorded      = 1994;

SONG song2;
song2.length_in_seconds  =  248;
song2.year_recorded      = 1988;
```

SONG is an alias for `typedef struct {int length_in_seconds; int year_recorded; }`

Dot notation: `x.y = value`

Structs are not objects!
The **dot (.)** operator is **not** a method call! (more later)

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 Reference Slides

- Intro to C
- Hello World
- Compilers (vs. Interpreters)
- C vs. Java: Syntax
- C Variables and Basic Types
- More C Features
- [Extra] Reference Slides

gcc/gdb commands

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

# 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
```

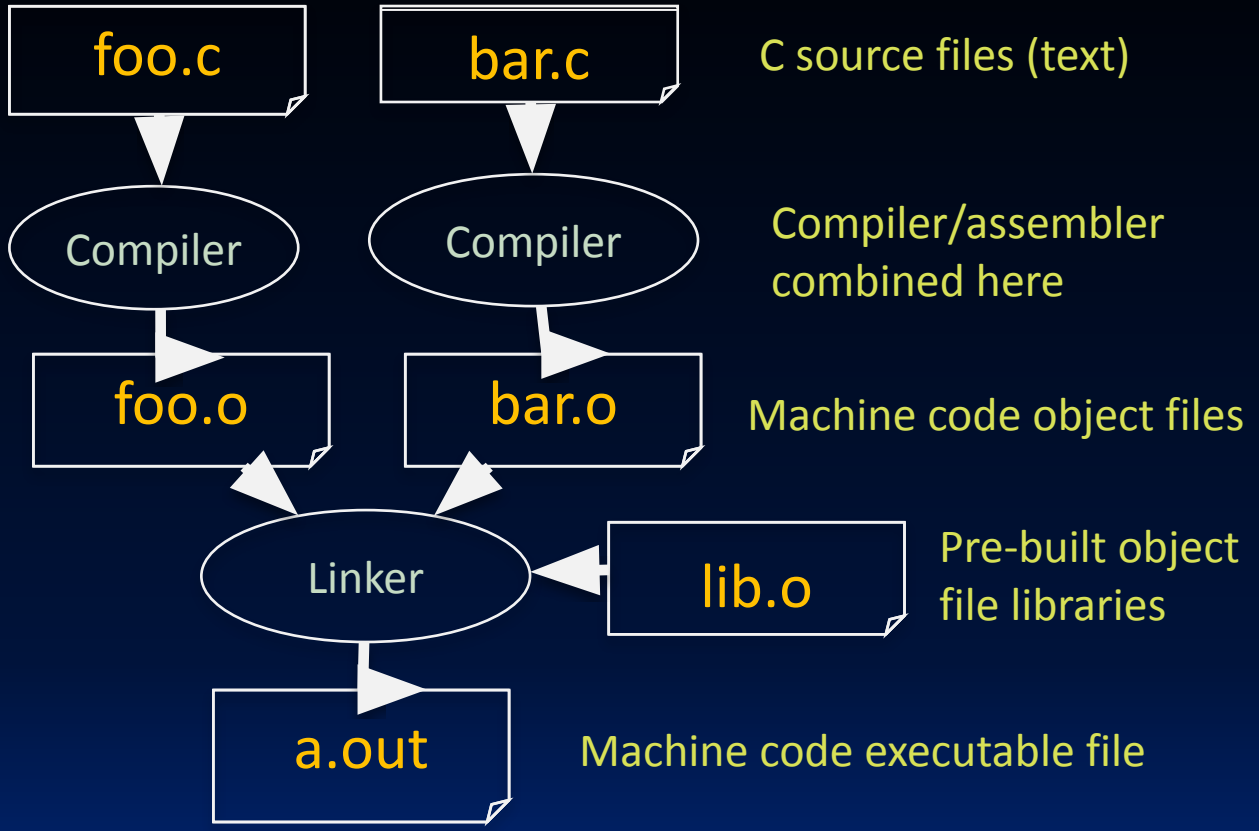
Has there been an update to ANSI C?

- Yes! It's called the "C99" or "C9x" std
  - To be safe: "`gcc -std=c99`" to compile
  - `printf("%ld\n", __STDC_VERSION__);` → `199901`
- References
  - `en.wikipedia.org/wiki/C99`
- Highlights
  - Declarations in `for` loops, like Java
  - Java-like `//` comments (to end of line)
  - Variable-length non-global arrays
  - `<inttypes.h>`: explicit integer types
  - `<stdbool.h>` for boolean logic def's

Has there been an update to C99?

- Yes! It's called the "C11" (C18 fixes bugs…)
  - You need "`gcc -std=c11`" (or `c17`) to compile
  - `printf("%ld\n", __STDC_VERSION__);` → `201112L`
  - `printf("%ld\n", __STDC_VERSION__);` → `201710L`
- References
  - `en.wikipedia.org/wiki/C11_(C_standard_revision)`
- Highlights
  - Multi-threading support!
  - Unicode strings and constants
  - Removal of `gets()`
  - Type-generic Macros (dispatch based on type)
  - Support for complex values
  - Static assertions, Exclusive create-and-open, …

C Compilation Simplified Overview (more later)



C Pre-Processor (CPP)

- C source files first pass through macro processor, CPP, before compiler sees code
- CPP replaces comments with a single space
- CPP commands begin with "#"
  - #include "file.h" /* Inserts file.h into output */
  - #include <stdio.h> /* Looks for file in standard location, but no actual difference! */
  - #define PI (3.14159) /* Define constant */
  - #if/#endif /* Conditionally include text */
- Use –save-temps option to gcc to see result of preprocessing
  - Full documentation at: http://gcc.gnu.org/onlinedocs/cpp/

C vs. Java…operators nearly identical

- arithmetic: `+, -, *, /, %`
- assignment: `=`
- augmented assignment: `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
- bitwise logic: `~, &, |, ^`
- bitwise shifts: `<<, >>`
- boolean logic: `!, &&, ||`
- equality testing: `==, !=`
- subexpression grouping: `()`
- order relations: `<, <=, >, >=`
- increment and decrement: `++` and `--`
- member selection: `., ->`
  - Slightly different than Java because there are both structures and pointers to structures, more later
- conditional evaluation: `? :`

Typed Functions in C

- You have to declare the type of data you plan to return from a function
- Return type can be any C variable type, and is placed to the left of the function name
- You can also specify the return type as `void`
  - Just think of this as saying that no value will be returned
- Also need to declare types for values passed into a function
- Variables and functions MUST be declared before used

```
int number_of_people    () {  return 3;     }
float dollars_and_cents () {  return 10.33; }
```

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs (shows Java's legacy) for control flow
  - A statement can be a {} of code or just a standalone statement
- If-else
  - `if (expression) statement`
    - `if (x == 0) y++;`
    - `if (x == 0) {y++;}`
    - `if (x == 0) {y++; j = j + y;}`
  - `if (expression) statement1 else statement2`
    - There is an ambiguity in a series of if/else if/else if you don't use {}s, so use {}s to block the code
    - In fact, it is a bad C habit to not always have the statement in {}s, it has resulted in some amusing errors…
- while
  - `while (expression) statement`
  - `do statement while (expression);`

C Syntax : Control Flow (2/2)

- For

```
for (initialize; check; update) statement
```

- switch

```
switch (expression){
  case const1:    statements
  case const2:    statements
  default:        statements
}
break;
```

- Note: until you do a `break` statement things keep executing in the switch statement

C also has `goto`

- But it can result in spectacularly bad code if you use it, so don't!

C Syntax: Variable Declarations

- Similar to Java, but with a few minor but important differences
  - All variable declarations must appear before they are used
  - ANSI C: All must be at the beginning of a block.
  - A variable may be initialized in its declaration; ***if not, it holds garbage!***
    - the contents are undefined…
- Examples of declarations:
  - Correct: `{ int a = 0, b = 10; …`
  - Incorrect in ANSI C: `for (int i=0; …`
  - 🌟 **Correct in C99** (and beyond): `for (int i=0;…`