UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Lecturer
Justin Yokota

Introduction to the C Programming Language

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)
```

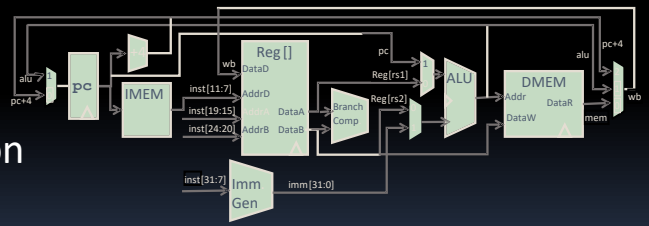
Anything can be represented as bits, i.e., data or instructions

Assembler

Machine Language Program (RISC-V)

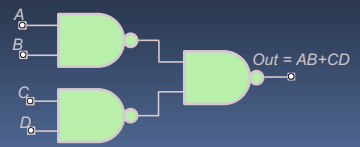
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

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



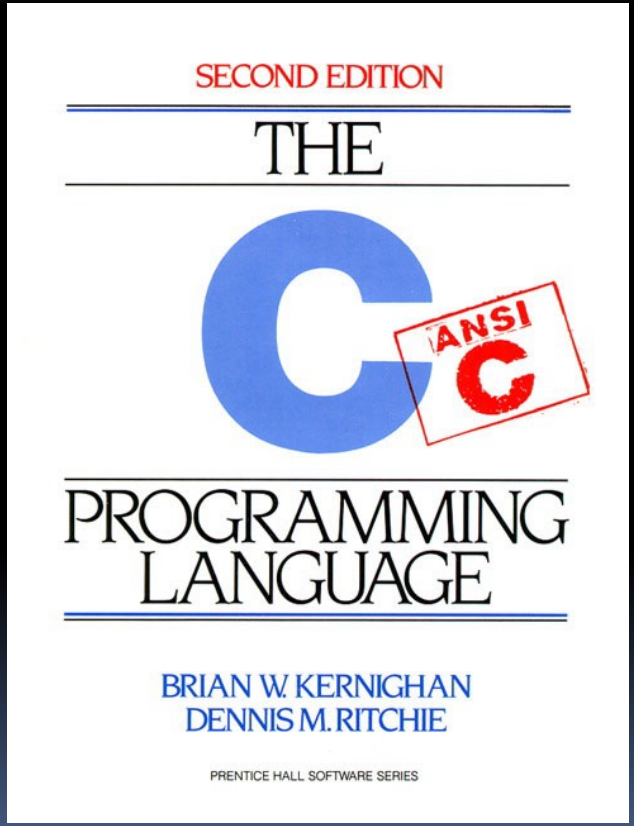
Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to C (1/2)

- **Kernighan and Ritchie**
  - *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.*
- **Enabled first operating system not written in assembly language!**
  - UNIX - A portable OS!



Introduction to C (2/2)

- **Why C?**
  - We can write programs that allow us to exploit underlying features of the architecture
    - memory management, special instructions, parallelism
- **C and derivatives (C++/Obj-C/C#) still one of the most popular programming languages after >40 years!**
- **If you are starting a new project where performance matters use either Go or Rust**
  - 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 to take advantage of modern multi-core microprocessors

Disclaimer

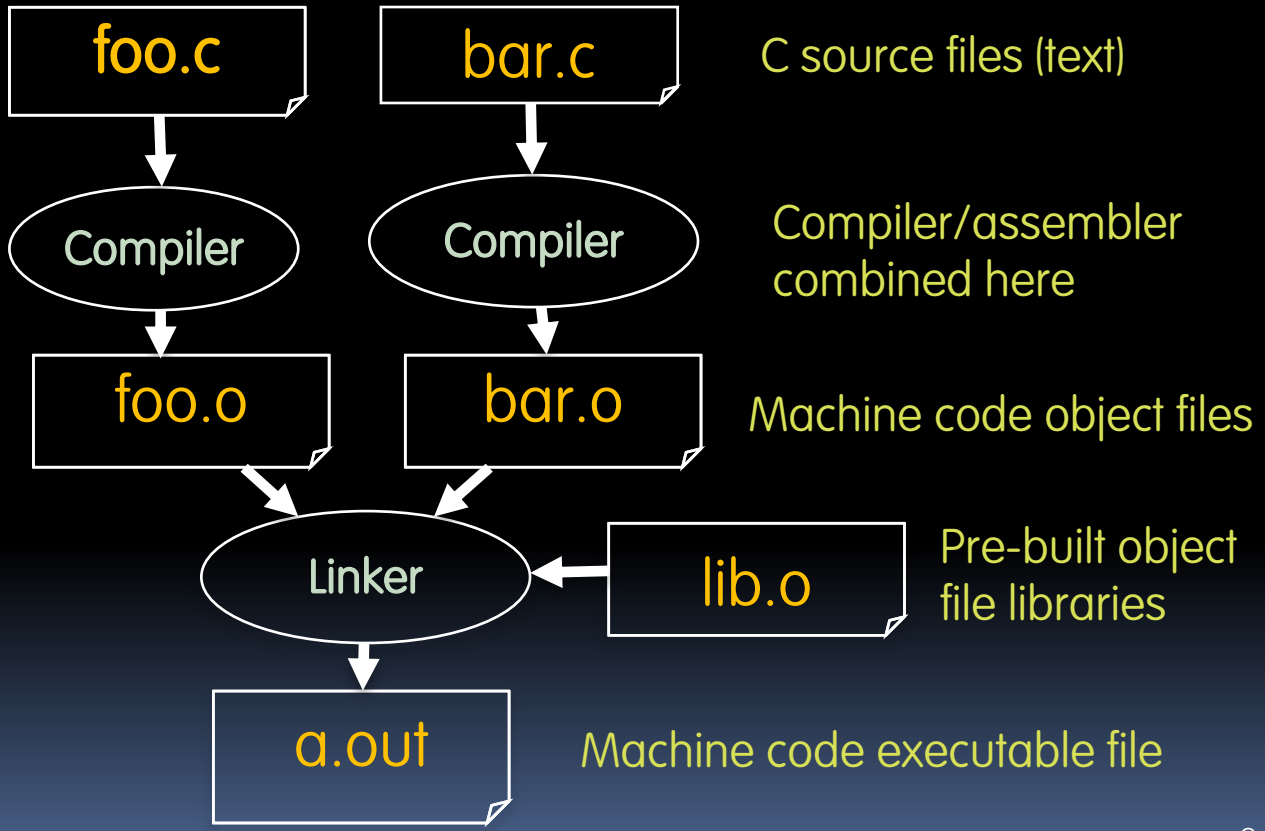
- **You will not learn how to fully code in C in these lectures! You'll still need your C reference**
  - K&R is a *must-have*
  - Useful Reference: "JAVA in a Nutshell," O'Reilly
    - Chapter 2, "How Java Differs from C"
  - Brian Harvey's helpful transition notes
    - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
- **Key C concepts: Pointers, Arrays, Implications for Memory management**
  - Key security concept: All of the above are *unsafe*: If your program contains an error in these areas it might not crash immediately but instead leave the program in an inconsistent (and often exploitable) state

Compile v. Interpret

Compilation: Overview

- **C compilers map C programs directly into architecture-specific machine code (string of 1s and 0s)**
  - Unlike Java, which converts to architecture-independent bytecode that may then be compiled by a just-in-time compiler (JIT)
  - Unlike Python environments, which converts to a byte code at runtime
    - These differ mainly in exactly when your program is converted to low-level machine instructions ("levels of interpretation")
- **For C, generally a two part process of compiling .c files to .o files, then linking the .o files into executables;**
  - Assembling is also done (but is hidden, i.e., done automatically, by default); we'll talk about that later in the CALL lecture

C Compilation Simplified Overview (more later)



Compilation: Advantages

- **Reasonable compilation time: enhancements in compilation procedure (`Makefiles`) allow only modified files to be recompiled**
- **Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)**
  - But these days, a lot of performance is in libraries:
  - Plenty of people do scientific computation in **Python!?!**
    - they have good libraries for accessing GPU-specific resources
    - Also, many times python allows the ability to drive many other machines very easily … wait for Spark™ lecture
    - Also, Python can call low-level C code to do work: Cython

Compilation: Disadvantages

- **Compiled files, including the executable, are architecture-specific, depending 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**
  - I.e., "porting your code" to a new architecture
- **"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

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/`

CPP Macros: A Warning...

- **You often see C preprocessor macros 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)));` ☑

C vs Java

C vs. Java (1/3)

	C	Java
Type of Language	Function Oriented	Object Oriented
Programming Unit	Function	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 and executes program	`java Hello` interprets bytecodes

hello, world	`#include <stdio.h>`
	`int main(void)`
	`{`
	`  printf("Hi\n");`
	`  return 0;`
	`}`
		`public class HelloWorld {`
		`  public static void`
		`main(String[] args) {`
		`  System.out.println("Hi");`
		`} }`

Storage	Manual (`malloc`, `free`)	New allocates & initializes, Automatic (garbage collection) frees

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

C vs. Java (2/3)

	C	Java
Comments (C99 same as Java)	`/* ... */`	`/* ... */` or `//` … end of line
Constants	`#define, const`	`final`
Preprocessor	Yes	No
Variable declaration (C99 same as Java)	At beginning of a block	Before you use it
Variable naming conventions	`sum_of_squares`	`sumOfSquares`
Accessing a library	`#include <stdio.h>`	`import java.io.File;`

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

C vs. Java (3/3) …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: `? :`

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 Syntax: main

- To get the `main` function to accept arguments, use this:
  - `int main (int argc, char *argv[])`
- What does this mean?
  - `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:

    ```
    unix% sort myFile
    ```

  - `argv` is a pointer to an array containing the arguments as strings (more on pointers later).

C Syntax

C Syntax: True or False?

- **What evaluates to FALSE in C?**
  - `0` (integer)
  - `NULL` (pointer: more on this later)
  - Boolean types provided by C99's `stdbool.h`
- **What evaluates to TRUE in C?**
  - …everything else…
  - Same idea as in Scheme
    - Only `#f` is false, everything else is true!

Typed Variables in C

- **Must declare the type of data a variable will hold**
  - Types can't change. E.g, `int var = 2;`

Type	Description	Example
`int`	Integer Numbers (including negatives) At least 16 bits, can be larger	`0, 78, -217, 0x7337`
`unsigned int`	Unsigned Integers	`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'`
`long`	Longer `int`, Size >= `sizeof(int)`, at least 32b	`0, 78, -217, 301720971`
`long long`	Even longer `int`, size >= `sizeof(long)`, at least 64b	`31705192721092512`

Integers: Python vs. Java vs. C

- **C: `int` should be integer type that target processor works with most efficiently**
- **Only guarantee:**
  - `sizeof(long long)` ≥ `sizeof(long)` ≥ `sizeof(int)` ≥ `sizeof(short)`
  - Also, `short` >= 16 bits, `long` >= 32 bits
  - All could be 64 bits
  - This is why we encourage you to use `intN_t` and `uintN_t`!!

Language	`sizeof(int)`
Python	>=32 bits (plain `int`s), infinite (long `int`s)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

Consts and Enums in C

- Constant 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

- Enums: a group of related integer constants. E.g.,

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

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; }
```

Structs in C

- **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;
```

Dot notation: `x.y = value`

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

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

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!

First Big C Program: Compute Sines table

```c
#include <stdio.h>
#include <math.h>
int main(void)
{
    int    angle_degree;
    double angle_radian, pi, value;

    printf("Compute a table of the sine function\n\n");
    pi = 4.0*atan(1.0); /* could also just use pi = M_PI */
    printf("Value of PI = %f \n\n", pi);
    printf("Angle\tSine\n");
    angle_degree = 0;/* initial angle value */
    while (angle_degree <= 360) { /* loop til angle_degree > 360 */
        angle_radian = pi * angle_degree / 180.0;
        value = sin(angle_radian);
        printf ("%3d\t%f\n ", angle_degree, value);
        angle_degree += 10; /* increment the loop index */
    }
    return 0;
}
```

```
PI = 3.141593

Angle  Sine
  0    0.000000
 10    0.173648
 20    0.342020
 30    0.500000
 40    0.642788
 50    0.766044
 60    0.866025
 70    0.939693
 80    0.984808
 90    1.000000
… etc …
```

C Syntax: Variable Declarations

- **Similar to Java, but with a few minor but important differences**
  - All variable declarations must appear before they are used
  - 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;…`

An Important 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

And In Conclusion, …

- C chosen to exploit underlying features of HW
- Key C concepts
  - Pointers, arrays, implications for mem management
- 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 bool types)
  - Use `intN_t` and `uintN_t` for portable code!
  - Unitialized variables contain garbage
    - "Bohrbugs" (repeatable) vs "Heisenbugs" (random)