Lecture 3:
Its time to see the C...

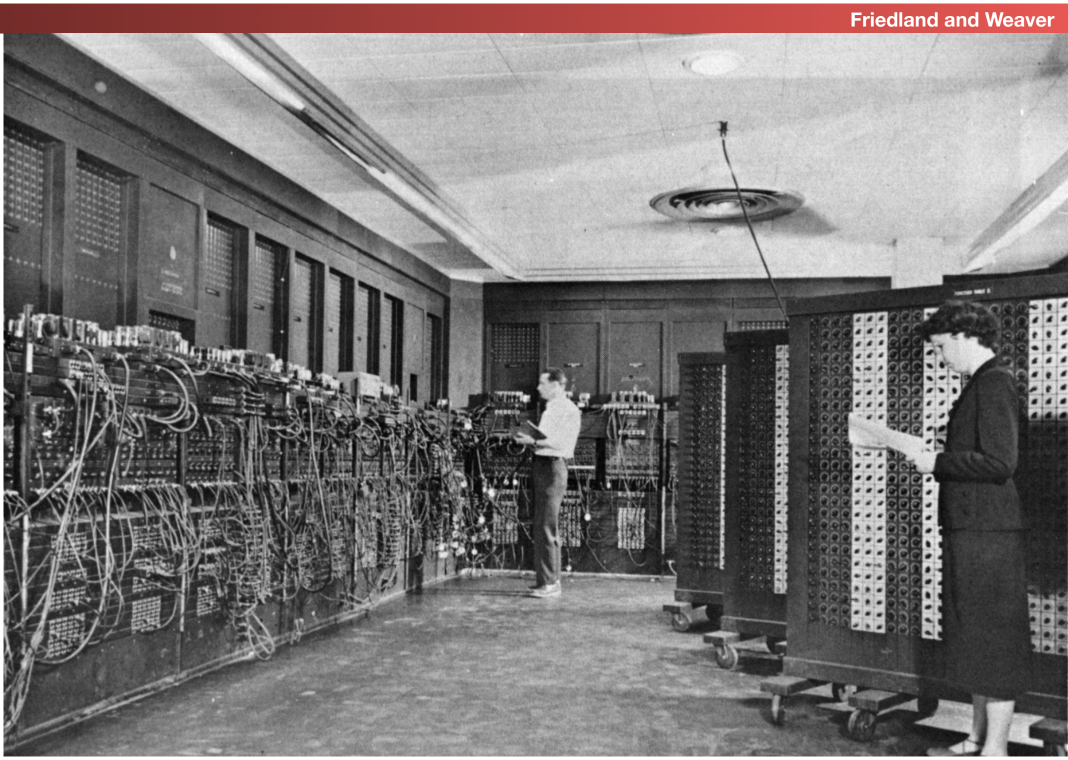
Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Agenda

- Computer Organization
- Compile vs. Interpret
- C vs Java

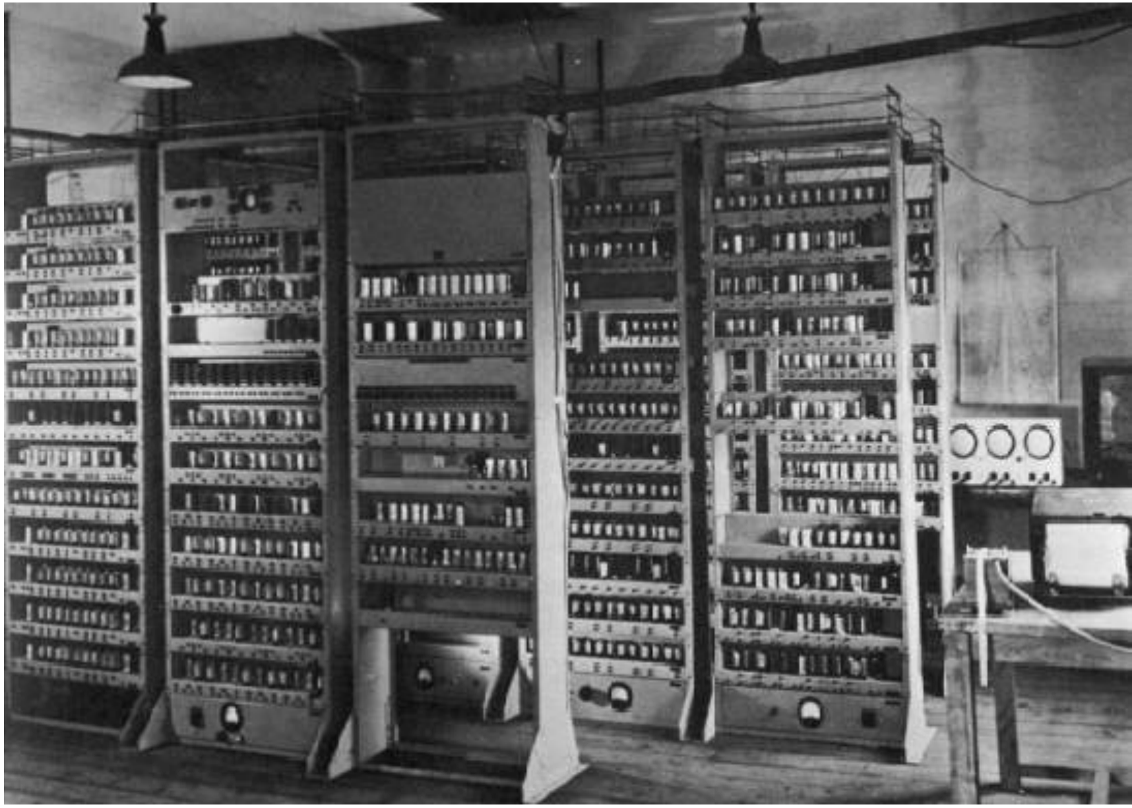
ENIAC (U.Penn., 1946)
First Electronic General-Purpose Computer

- Blazingly fast (multiply in 2.8ms!)
  - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches
  - At that time & before, "computer" mostly referred to ***people*** who did calculations

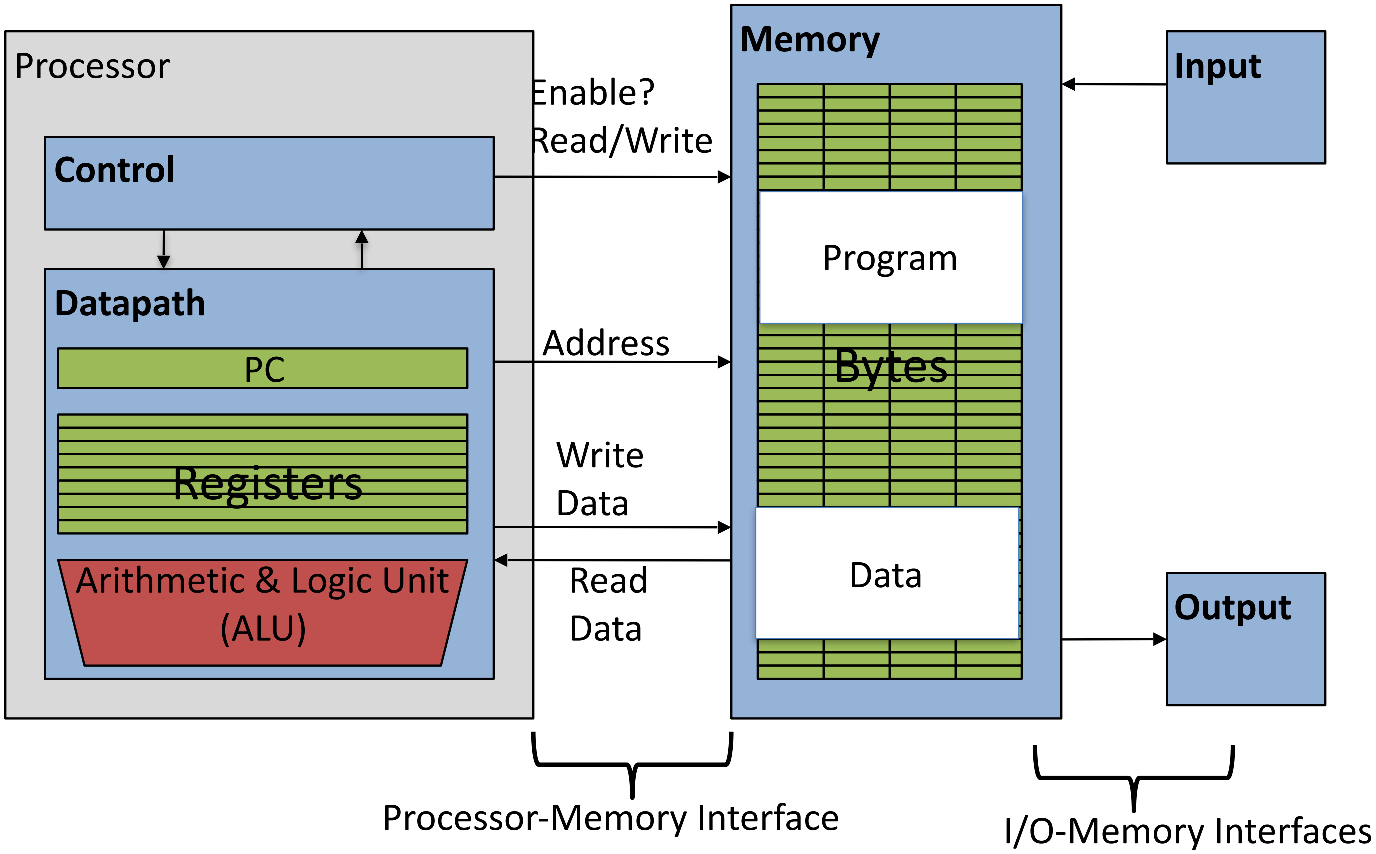


EDSAC (Cambridge, 1949)
First General Stored-Program Computer

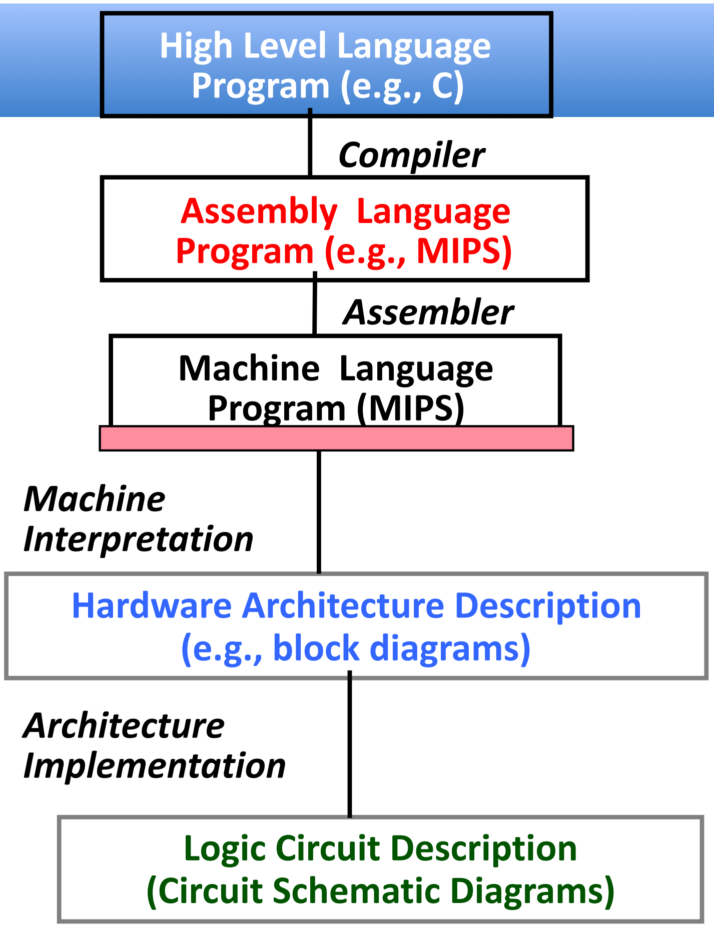
- Programs held as numbers in memory
  - This is the revolution:
    It isn't just programmable, but the program is just the same type of data that the computer computes on
- 35-bit binary 2's complement words



Components of a Computer



Great Idea: Levels of Representation/Interpretation



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

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

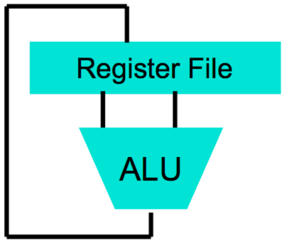
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

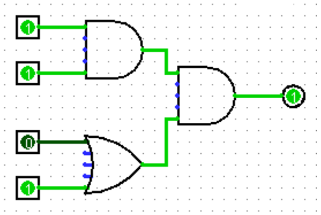
We are here!

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

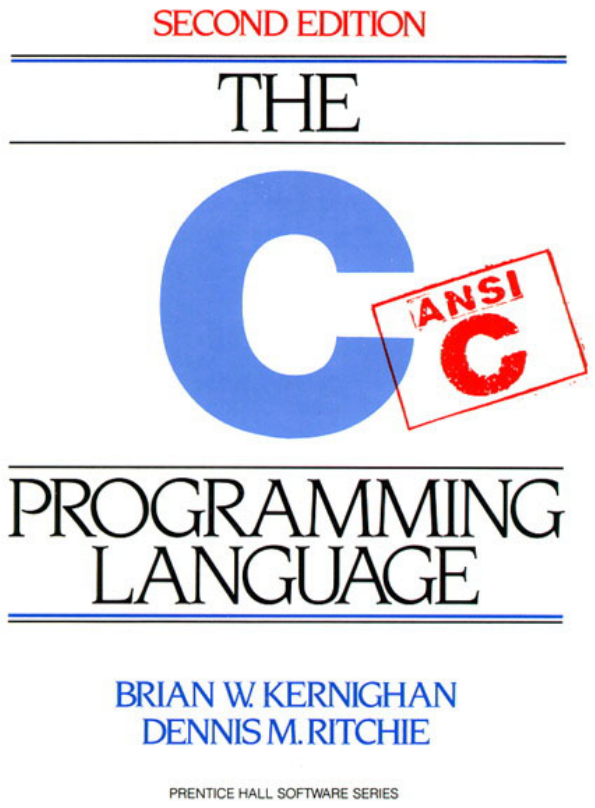
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





Introduction to C
“The Universal Assembly Language”

- Class pre-req included classes teaching Java
  - “Some” experience is required before CS61C
    - C++ or Java OK
- Python used in two labs
- C used for everything else "high" level
- Almost all low level assembly is MIPS
  - But Project 4 will require touching 64b Arm assembly which is very similar



Language Poll

- Please raise your hand for the first one you can say yes to:
- I have programmed in C, C++, C#, or Objective-C
- I have programmed in Java
- I have programmed in Swift, Go, Rust, etc
- None of the above

Intro to C

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

Intro to C

- 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 application programming languages after >40 years!

Disclaimer

- You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course
  - K&R is a ***must-have***
  -]"JAVA in a Nutshell," O'Reilly
    - Chapter 2, "How Java Differs from C"
    - http://oreilly.com/catalog/javanut/excerpt/index.html
  - Brian Harvey's helpful transition notes
    - On CS61C class website: pages 3-19
    - 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

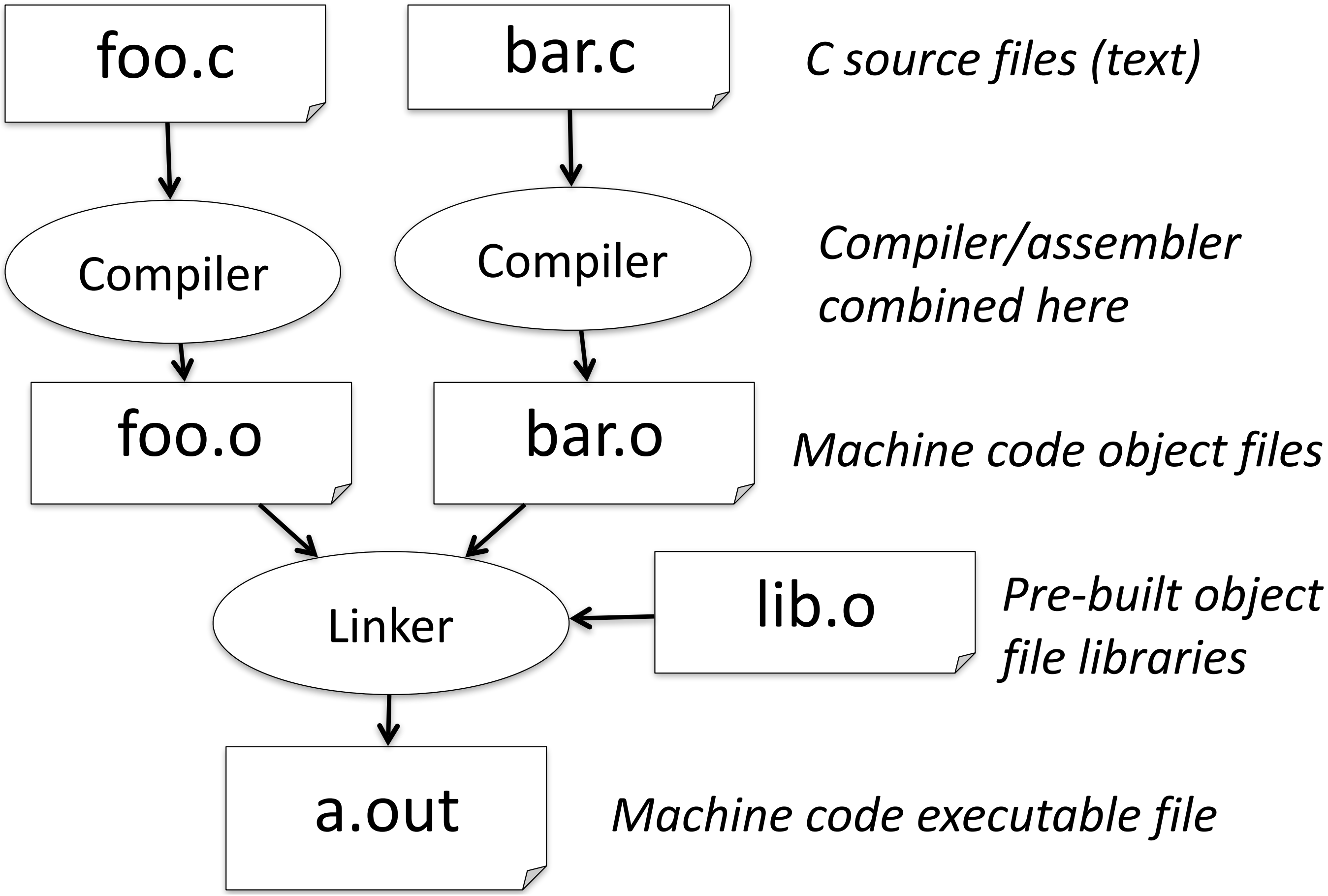
Agenda

- Computer Organization
- Compile vs. Interpret
- C vs Java

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

C Compilation Simplified Overview
(more later in course)



foo.c

bar.c

C source files (text)

Compiler

Compiler

Compiler/assembler combined here

foo.o

bar.o

Machine code object files

Linker

lib.o

Pre-built object file libraries

a.out

Machine code executable file

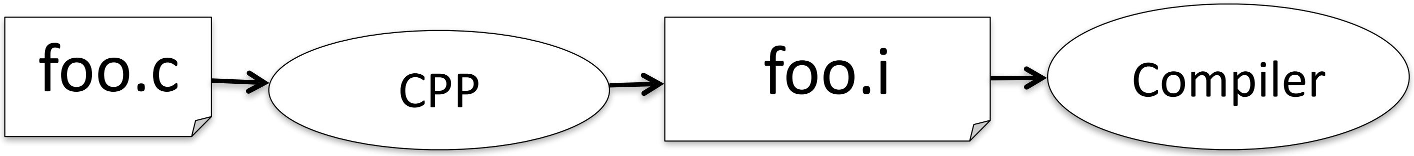
Compilation: Advantages

- 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!?!***, because they have good libraries for accessing GPU-specific resources
- Reasonable compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., MIPS vs. RISC-V) and the operating system (e.g., Windows vs. Linux)
- 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 do compiles in parallel (**`make -j X`**)
    - 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 M_PI (3.14159)` /* Define constant */
- `#if/#endif /* Conditional inclusion of 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
- This can produce, umm, interesting errors
  - `#define twox(x) (x + x)`
  - `twox(y++);`
  - `(y++ + y++);`

C vs. Java

	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("Hello\n");`
	`   return 0;`
	`}`
		`public class HelloWorld {`
		`   public static void main(String[] args) {`
		`   System.out.println("Hello");`
		`   }`
		`}`

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

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

C vs. Java

	C	Java
Comments	/* … */	/* … */ or // … end of line
Constants	#define, const	final
Preprocessor	Yes	No
Variable declaration	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

Typed Variables in C

```
int   variable1   = 2;
float variable2   = 1.618;
char  variable3   = 'A';
```

- Must declare the type of data a variable will hold
  - Types can't change

Type	Description	Examples
int	integer numbers, including negatives	0, 78, -1400
unsigned int	integer numbers (no negatives)	0, 46, 900
float	floating point decimal numbers	0.0, 1.618, -1.4
char	single text character or symbol	'a', 'D', '?’
double	greater precision/big FP number	10E100
long	larger signed integer	6,000,000,000

Integers: Python vs. Java vs. C

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

- 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

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;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants. Ex:

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

Typed Functions in C

```
int number_of_people ()
{
  return 3;
}

float dollars_and_cents ()
{
  return 10.33;
}

int sum ( int x, int y)
{
   return x + y;
}
```

- 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 necessary to declare types for values passed into a function
- Variables and functions MUST be declared before they are used

Structs in C

- 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;
```

Dot notation: **`x.y = value`**

A First C Program: Hello World

`Original C:`

```
main()
{
  printf("\nHello World\n");
}
```

`ANSI Standard C:`

```
#include <stdio.h>

int main(void)
{
  printf("\nHello World\n");
  return 0;
}
```

C Syntax: `main`

- When C program starts
  - C executable a.out is loaded into memory by operating system (OS)
  - OS sets up stack, then calls into C runtime library,
  - Runtime 1st initializes memory and other libraries,
  - then calls your procedure named main ()
- We'll see how to retrieve command-line arguments in main() later…

A Second C Program: Compute Table of Sines

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    int    angle_degree;
    double angle_radian, pi, value;
    /* Print a header */
    printf("\nCompute a table of the
sine function\n\n");
    /* obtain pi once for all       */
    /* or just use pi = M_PI, where */
    /* M_PI is defined in math.h    */
    pi = 4.0*atan(1.0);
    printf("Value of PI = %f \n\n",
            pi);
    printf("angle     Sine \n");
```

```
 angle_degree = 0;
 /* initial angle value */
 /* scan over angle     */
 while (angle_degree <= 360)
 /* loop until angle_degree > 360 */
    {
       angle_radian = pi*
             angle_degree/180.0;
       value = sin(angle_radian);
       printf (" %3d      %f \n ",
              angle_degree, value);
       angle_degree += 10;
       /* increment the loop index */
    }
 return 0;
}
```

Second C Program
Sample Output

```
Compute a table of the sine function

Value of 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
 100      0.984808
 110      0.939693
 120      0.866025
 130      0.766044
 140      0.642788
....
```

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: `for (int i = 0; i < 10; i++) { ...`

Newer C standards are more flexible about this, more later

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs (shows Java's legacy) in terms of control flow
  - A statement can be a {} of code or just standalone
- 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
- 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!

    ```
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;  /* MISTAKE! THIS LINE SHOULD NOT HAVE BEEN HERE */
    ```

C Syntax: True or False

- What evaluates to FALSE (aka #AlternateTrue) in C?
  - 0 (integer)
  - NULL (a special kind of pointer that is also 0: more on this later)
  - No explicit Boolean type
    - Often you see #define bool (int)
- What evaluates to TRUE in C?
  - ***Anything*** that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true!

C and 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: ., ->
  - This is slightly different than Java because there are both structures and pointers to structures
- conditional evaluation: ? :

Nick's Tip of the Day...

- Why valgrind
  - Easy to install on a Raspberry Pi:
    `sudo apt-get install valgrind`
  - Instructions on other platforms are implementation dependent
- Valgrind turns most unsafe "heisenbugs" into "bohrbugs"
  - It adds almost all the checks that Java does but C does not
  - The result is your program ***immediately*** crashes where you make a mistake
- Nick's scars from 60C:
  - First C project, spent an entire day tracing down a fault...
  - That turned out to be a <= instead of a < in initializing an array!