CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Introduction to C
  - Why C?
  - Hello, world!

Your Goals

- Compile a C program using gcc

Aside: GNU

GNU's Not Unix

- Goal is to be completely free
  - Mostly licensed under 'GNU General Public License (GPL)'
- Many software tools
- Typically used with a linux kernel
- Managed by the Free Software Foundation
- https://www.gnu.org/home.en.html

C v. Scripting

C

- Compiled
- Highly structured, data-typed
- Strings have library processing
- Data structures and libraries
- Good for large complex programs
  - Java, with object-oriented programming, is even better for complex programs

Scripting

- Interpreted
- Esoteric variable access
- Everything is a string
- Easy access to files and program
- Good for quick & interactive programs
  - Do one thing and do it well

C v. Java

C

- Lower level (closer to assembly)
- No guaranteed memory safety
- Procedural
- Compiled (not interpreted like bash)
- Conditional controls (if, while)
- Modern syntax (human readable)
- Small standard library

Java

- Higher level (lots of compilation)
- Safe (sand-boxed in jvm, compiled limits)
- Object Oriented
- Compiled
- Conditional controls (if, while)
- Modern syntax (human readable)
- Large standard library, huge extended libraries

C v. Rust

C

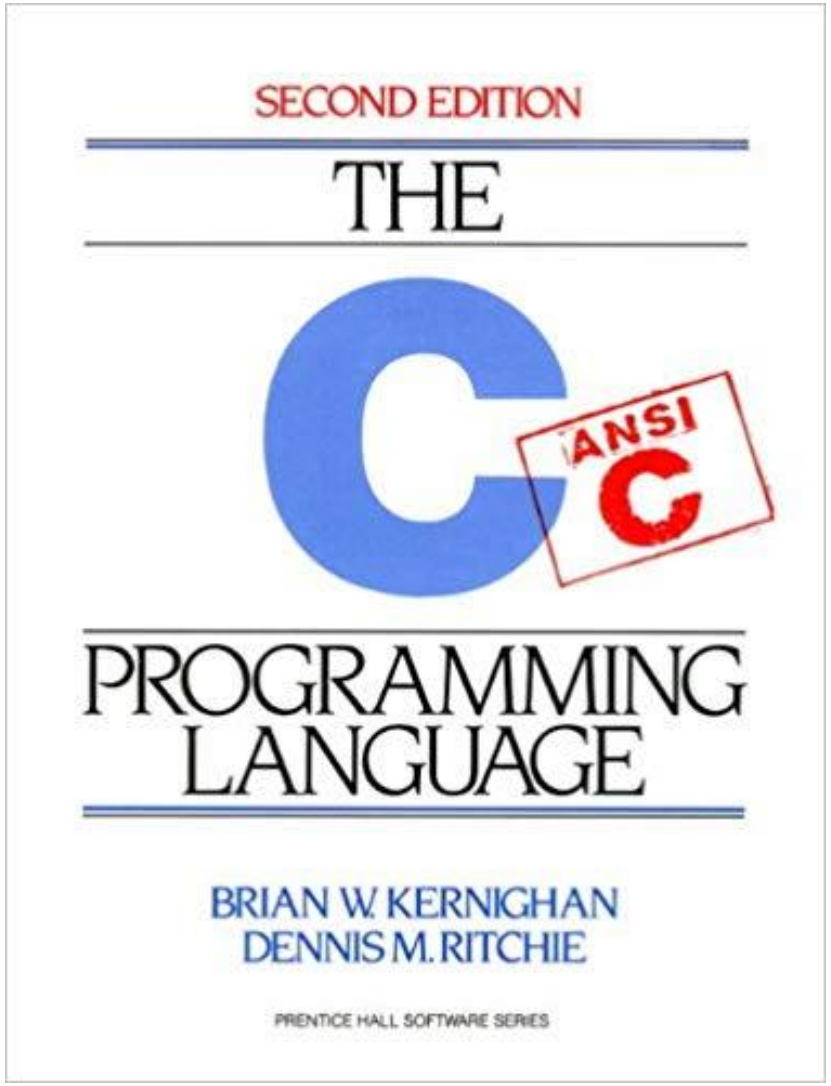
- Relies on user management for memory safety
- C++ Exceptions for error management
- C++ extensive std::lib
- C – procedural, c++ object-oriented
- Extensive legacy code

Rust

- Built in memory-safety
  - Single owner of memory
- Uses Result-type
- Designed for thread safety
- Not object-oriented but offers similar features.
- 30 years younger than C++

Why C?

- C is a fairly compact language - fewer features than Java, but easier to implement efficiently
- Provides lower level (closer to assembly) language
- Understanding C can give insight into how computers (and memory) work
- Still used for
  - Embedded programming
  - Systems programming
  - High-performance code
  - GPU Programming
  - C/C++ has more legacy code than any language except COBOL (by some estimates)



C References

- K&R C – available on kindle and in the UW library – **the** language reference
- Essential C: http://cslibrary.stanford.edu/101/EssentialC.pdf
- Cplusplus web: http://www.cplusplus.com/
- O'Reilly books (C in a Nutshell, etc.) – available through the UW libraries.

Hello World

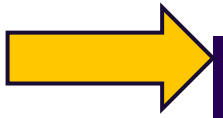
- Compile at a bash command line with `gcc hello.c`
  - Creates an executable `a.out`
- `gcc –Wall –std=c11 –o hello hello.c`
  - `Wall` turns on all warnings
  - `C11` specifies C11 standard
  - Creates executable hello
- Run: `./a.out` or `./hello`
- Exits with `0` (`return 0;`)

```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World - #include

- Includes the `stdio.h` (the standard IO library – `printf`, etc.)
- Could include other standard libraries
  - `stdlib.h`, `math.h`, `assert.h`, etc.
- You can also include developer files
  - `#include "myFile.h"`

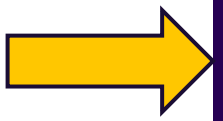


```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World - #define

- Lines with `#` are pre-processor commands.
- Can also `define` constants or other macros.

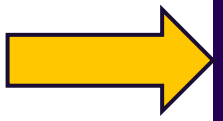


```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World - comments

- This is a comment block.
- `/* Long form comments */`
- `// in-line comments`

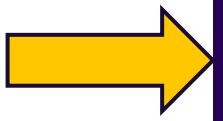


```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World - functions

- Main function
- Looks a lot like Java functions
  - Return type, arguments, etc.
  - Code block surrounded by { and }.
- Programs run through 'main'
  - BUT, its not part of a class!
- Return values from 'main' will exit the program.
  - >> echo "$?"

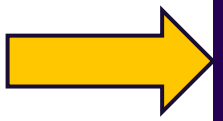


```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World – logical constructs

- There are also logical constructs that look a lot like Java!

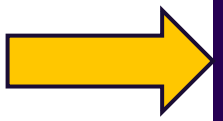


```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World - variables

- Also, there are variables that look a like like Java
  - Data type assigned at declaration
  - `int, long, float, double, char`



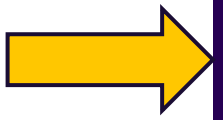
```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World - strings

“Hello, World!\n”

- A string of length 15
  - \n is one character
  - Also contains ‘null terminator’ - \0
- A ‘string literal’ evaluates to a global, immutable array



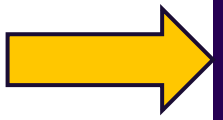
```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World - printf

`printf`

- Function defined in stdio.h
- Prints to stdout



```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

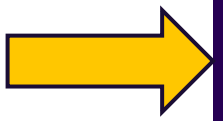
Source File Structure

```
// includes for functions & types defined elsewhere
#include <stdio.h>
#include "localstuff.h"
// symbolic constants
#define MAGIC 42
// global variables (if any)
static int days_per_month[ ] = { 31, 28, 31, 30, …};
// function prototypes
// (to handle "declare before use")
 void some_later_function(char, int);
// function definitions
void do_this( ) { … }
char *return_that(char s[ ], int n) { … }
int main(int argc, char ** argv) { … }
```

Logical Control

- `if while switch for`
- `Break continue`

- https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Statements
- No built-in Boolean type!
  - Use integers, and/or declare constants
  - Generally 0/NULL -> False
  - Anything else -> True
- But could `#include <stdbool.h>`



```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

I/O: `printf, scanf`

- `printf` (print-format)
- `int printf(const char *format, ...)`
- 'Format' is a string that can contain format tags
- + additional arguments to match tags
- Number of arguments better match number of format specifiers
- Corresponding arguments match types (%d, int; %f, float; %e, float (prints scientific); %s, \0-terminated char*; … Compiler might check, but not guaranteed
  - best case scenario: you crash
- `printf("%s: %d %g\n", p, y+9, 3.0)`

- `scanf` (gets input, formatted)
- `int scanf(const char *format, ...)`
- 'Format' is a string that can contain format tags
- + additional arguments to match tags - should be pointers to the right data type so input can be stored in them
- `scanf("%d %s", &n, str);`
- `scanf("%*s %d", &a);`
  - %*s ignores string until space, then reads in an integer

Pointers

- Also, there are variables DO NOT LOOK like Java
  - Data type assigned at declaration
- BUT these variables a storing addresses in memory, not values.

```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int arg[illegible] **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Computers & Memory

- CPU - the 'central processing unit': computer circuitry that follows computer instructions with simple logic, arithmetic, and I/O
- Hard disc storage (modernly often solid-state memory instead of traditional drive): holds long-term memory which can persist across re-starts
- RAM (memory) : where data is stored during operation - short term memory

Working Memory

Address space: list of bytes addressed in order

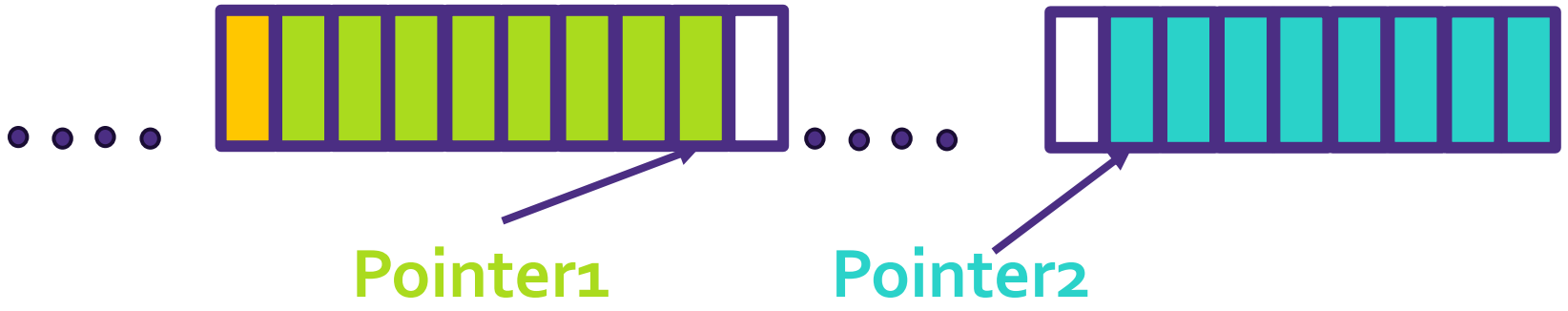
- Programs are said to have access to this 2^{64} byte space
  - '64 bit' system refers to needing 64 bits to index the space
  - But really don't - many other things are also using this space
  - Addressing is also usually 'virtual', but, this is a good model for us
- Location in array is the 'address' of a byte
- Programs keep track of addresses of each of their pieces of memory
- Accessing unused address causes a 'segmentation fault'

Working Memory - zones

Code Globals Heap-> <-Stack

- Lowest memory stores program instructions, then global variables (static constants, string literals)
- 'Heap' holds dynamically allocated variables ('new' or 'malloc' variables)
- 'Stack' holds current instructions, each function in a frame
  - 'Stack' memory implies that a frame is added, and then the last frame added is removed first
- The heap and stack grow dynamically. Meet in the middle ?= 'out of memory' error

So, what is a pointer?



- Point to an address in memory

Code	Meaning
`int x = 4;`	Variable called 'x' of type 'int' storing value of '4'
`int *xPtr = &x;`	Variable called 'xPtr' ot type 'pointer to int', given value of 'the location of x'
`int xCopy = *xPtr;`	Variable called 'xCopy' of type 'int' storing value at the location pointed to by xPtr
`int *noPtr = NULL;`	Variable called 'noPtr' of type 'pointer to int' correctly set to 'no location'