CSE 410: Systems Programming

C and POSIX

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

C and POSIX

In this course, we will use:

- The C programming language
- POSIX APIs

C is a programming language designed for systems programming.

POSIX is a standardized operating systems interface based on UNIX.

The POSIX APIs are C APIs.

Why C?

There are dozens of programming languages. Why C?

C is “high level” — but not very.

- C provides functions, structured programming, complex data types, and many other powerful abstractions
- …yet it also exposes many architectural details

Most operating system kernels are written in C.

Why POSIX?

POSIX is a standardization of the UNIX API.

- Portable Operating System Interface …X?

In the 80s and 90s, UNIX implementations became very fragmented.

Interoperability suffered, and a standard was developed.

POSIX is probably the most widely implemented OS API.

- All UNIX systems (Linux, BSD, *etc.*)
- macOS
- Many real-time operating systems (QNX, RTEMS, eCos, …)
- Microsoft Windows (sort of)

But really …why?

We had to choose something.

C fits the layer of abstraction we wanted to learn about.

POSIX is widely available, well-documented, and simple.[1]

The text specifically considers Linux on X86-64.
We will, too.

[1] err …ish?

C Syntax

C will look familiar to you, as Java syntax is based on C.

However, there are some large semantic differences.

- Syntax: how something is written
- Semantics: what something means

First of all, there are no classes in C.
However, C functions and Java methods look similar.

main()

Every C program starts with the function `main()`.[2]

```
int main(int argc, char *argv[]) {
    return 0;
}
```

This should look pretty familiar, except for that `*`.

A C function takes zero or more arguments and returns a single value.

All arguments are pass-by-value (unlike Java).

[2]Sort of …

C flow control

C flow control looks a lot like Java.

However, iterator syntax is not supported.

```
if (condition) { /* true body */ } else { /* false
    body */ }

while (condition) { /* body */ }

for (setup; condition; iterate) { /* body */ }

switch (integer) {
  case value: /* body */ break;
  default: /* body */ break;
}
```

C Types

C is statically typed.

Every variable is declared with a type.

Every assignment to a variable must honor its type.
(However, C will perform conversions in some cases.)

Valid:

```c
int x = 0;
long y = 0;
x = 37;
y = x;
```

Invalid:

```c
int x = 0;
x = "Hello, world!";
```

C Gotchas

Some gotchas coming from Java:

- C variables are not initialized when declared.
- There is no garbage collector.
- There are no constructors or destructors.

You must set up and clean up after yourself in C.

POSIX Overview

POSIX defines a lot of things.

- An API for interacting with the OS kernel
- A command line interpreter (shell)
- A set of available commands
- Filesystem semantics
- …

We're *mostly* concerned with the first.

POSIX API

The POSIX interface to the OS facilities forms an API: Application Programming Interface.

This API is a set of C language functions and variables.

On a UNIX system, the functions are mostly system calls.

System calls are requests to the OS kernel.

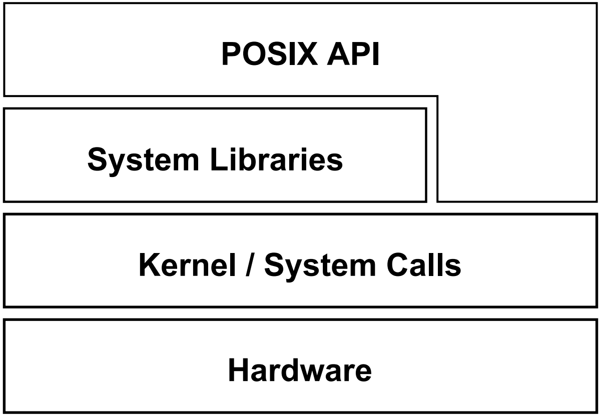
The kernel is the privileged core of the OS.

Example POSIX functions

- `open()`: Opens a file for reading or writing
- `fork()`: Creates a new process
- `connect()`: Creates a network connection
- `exit()`: Gracefully terminates the current process
- `tcsetattr()`: Configures a serial (or virtual) terminal
- `time()`: Get the current time

Some POSIX functions overlap the C standard.

The POSIX Stack



Man pages

The POSIX manual is online on most POSIX systems, accessible via the `man` command.

The manual is divided into sections [2]:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions eg /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)

The C Toolchain

A C program consists of one or more source files.

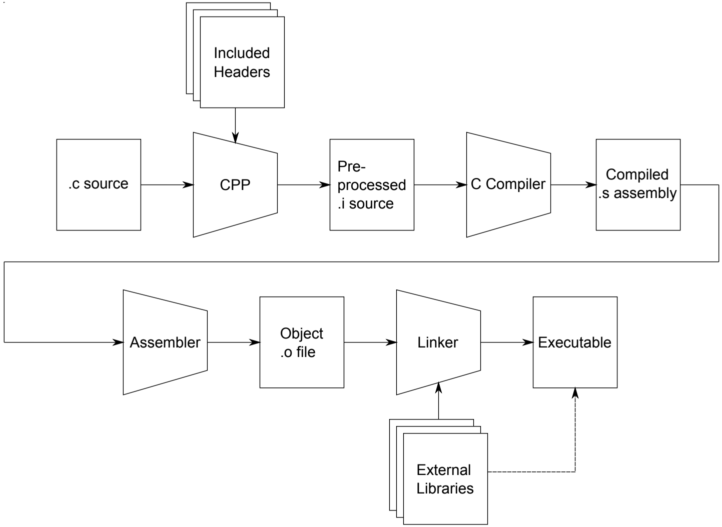
The C compiler driver passes the source code through several stages to translate it into machine code.

A source file[3] is sometimes called a translation unit.

Each stage may be invoked individually …more later.

[3]Plus some other stuff

The C Toolchain



The C Preprocessor

The preprocessor does just what it sounds like.

It performs certain source code transformations before the C is processed by the compiler.

It doesn't understand C, and can be used for other things!

The C Compiler

The compiler transforms C into machine-dependent assembly code.

It produces an object file via the assembler.

An object file contains:

- **Constant data:** Data of unchanging value used by the code in the object file
- **Static symbols:** Locally-defined variables and functions that are not available outside of this translation unit
- **Locally-defined globals:** Globally visible variables and functions that have complete definitions
- **Unresolved symbols:** Globally visible variables and functions that are defined in another translation unit.

The Assembler

The assembler takes assembly code and transforms it into machine-executable instructions.

While the assembler is a powerful tool, in a modern C compiler toolchain it performs a more-or-less mechanical transformation.

The Linker

The linker joins object files into an executable.

It maintains a symbol table for each object file.

Unresolved symbols in one object file may be found (and thus resolved) in other object files.

Other unresolved symbols may be found in libraries.

A Warning

These slides attempt to be precise, but simplify some things.

Usually this is because the details:

- are unnecessarily confusing, or
- require knowledge you are not expected to have.

If something here conflicts with the standard or the compiler, the standard or compiler wins.

C Syntax

Now for a rundown of C syntax.

We'll talk about both preprocessor and language syntax.

We will revisit C throughout the semester.

I recommend The C Programming Language [3]

The C Preprocessor

The C preprocessor applies preprocessor directives and macros to a source file, and removes comments.

Directives begin with #.

- `#include`: (Preprocess and) insert another file
- `#define`: Define a symbol or macro
- `#ifdef`/`#endif`: Include the enclosed block only if a symbol is defined
- `#if`/`#endif`: Include only if a condition is true
- …

Preprocessor directives end with the current line (not a semicolon).

Including headers

The `#include` directive is primarily used to incorporate headers.

There are two syntaxes for inclusion:

- `#include <file>`
  Include a file from the system include path (defined by the compiler)
- `#include "file"`
  Include a file from the current directory

Defining Symbols and Macros

The #define directive defines a symbol or macro:

```
#define PI 3.14159

#define PLUSONE(x) (x + 1)

PLUSONE(PI) /* Becomes (3.14159 + 1) */
```

Macros are expanded, not calculated!
The expansion will be given directly to the compiler.

Conditional Compilation

The various `#if` directives control conditional compilation.

```
#ifdef ARGUMENT
/* This code will be included only if ARGUMENT is
   a symbol defined by the preprocessor --
   regardless of its expansion */
#endif
```

The `#ifndef` directive requires `ARGUMENT` to be undefined.

The `#if` directive requires `ARGUMENT` to evaluate to true.

Using the Preprocessor

The preprocessor can be invoked as gcc -E.

Using the preprocessor correctly and safely is tricky.

For now, it is best to limit your use of the preprocessor.

We'll talk more about cpp later.

Keywords

Keywords (as of C99):

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`inline`	`int`	`long`	`register`
`restrict`	`return`	`short`	`signed`	`sizeof`
`static`	`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`	
`_Bool`	`_Complex`	`_Imaginary`		

Reserved words (simplified rules):

- Identifiers beginning with underscore (frequently violated)
- Certain macros (you are unlikely to notice)
- Standard methods and variables (you may notice)

Integer Types

Platform-specific integer types you should know:

- `char`: One character.
- `short`: A short (small) integer
- `int`: An “optimally sized” integer
- `long`: A longer (bigger) integer
- `long long`: An *even longer* integer

Their sizes are: 8 bits $\leq$ `char` $\leq$ `short` $\leq$ `int` $\leq$ `long` $\leq$ `long long`

Furthermore:
`short`, `int` $\geq$ 16 bits, `long` $\geq$ 32 bits, `long long` $\geq$ 64 bits

Whew!

Integer Modifiers

Every integer type may have modifiers.

Those modifiers include `signed` and `unsigned`.

All unmodified integer types *except* `char` are signed.
`char` may be signed or unsigned!

The keyword `int` may be elided for any type except `int`.
These two declarations are equivalent:

```
long long nanoseconds;
signed long long int nanoseconds;
```

Integers of Explicit Size

The confusion of sizes has led to explicitly sized integers.

They live in `<stdint.h>`

Exact-width types are of the form `intN_t`.
They are exactly *N* bits wide; *e.g.*: `int32_t`.

Minimum-width types are of the form `int_leastN_t`.
They are at least *N* bits wide.

There are also unsigned equivalent types, which start with `u`:
`uint32_t`, `uint_least8_t`

N may be: 8, 16, 32, 64.

So Many Integers!

All of this about integers is to drive home the following:
C is a rather low-level language.

You will worry more about architecture details in C than you have in other languages, previously.

Constants

We talked about defined constants previously.

C also has const-qualified types:

```
const int CALLNUMBER = 410;
```

A const-qualified type may be linked as a variable, but the compiler will emit an error if it can detect a change.

The weasel-words are important there:
`const` in C is not as powerful as in some languages.

Still, `const` can keep you out of trouble!

Arrays

Arrays in C are declared and dereferenced with `[]`.

Declaring an array of 10 integers:

```c
int array[10];
```

An array represents a contiguous block of memory that contains its elements back-to-back.
(This will be important later!)

Array declarations are statically sized.

This means the compiler must know exactly how many elements there are.

Array Sizing

Illegal:

```
const int x = 47;
int array[x];
```

Recall that `const` has weasel words.

The compiler cannot guarantee that `x` is 47!

Legal:

```
#define X 47
int array[X];
```

Implicit Array Sizes and Array Constants

An array constant uses curly brackets.

If the compiler can figure out how big an array should be, it doesn't need an explicit size.

```
int array[] = { 1, 2, 3 };
```

This creates an array of size 3.

Array Dereferencing

Array dereferencing works (for now) just like you'd expect:

```
int val = array[7];
```

The type of an array dereference is the type of the array.
(This will be important later.)

Array indices for dereference need not be constant:

```
int x = 47;
int y = array[x];
```

C Strings

A “C string” is an array of `char`s, with the last byte being 0.

The C compiler will generate such a string when you use `""`.

String variables are usually a pointer to `char`.
(We’ll talk a lot more about pointers later!)

They can also be a constant array.

```
char str[] = "CSE 410";
```

This statement will create an array of length 8, containing:

- The individual characters `'C'`, `'S'`, …
- A byte of value 0

C String Termination

The zero byte at the end of a string is called a NUL terminator.

(No, NUL is not a typo! That's the ASCII name for a zero byte!)

To iterate a C string, you use the NUL to find the end:

```c
char str[] = "C is awesome!";
for (int i = 0; str[i] != 0; i++) {
    if (str[i] == '!') {
        printf("An exclamation!\n");
    }
}
```

C String Escapes

C strings can contain escapes starting with \.

These escapes will be converted to specific ASCII characters.

Some escapes you should know:

Escape	Expansion
`\\`	Literal backslash
`\r`	Carriage return
`\n`	Platform-dependent end of line
`\t`	ASCII Tab
`\"`	Literal double quote
`\0`	ASCII NUL (string terminator)
`\000`	(Any 3 octal digits); byte value given
`\x00`	(x and any 2 hex digits); byte value given

Pointers

The last major data type we'll discuss today is the pointer.

Pointers do not appear in Java, Python, *etc.*.
They are most similar to object references in these languages.

A C pointer contains the address of a memory location.

It will also have an associated type for what is at that location.

Pointer Concepts

A pointer:

- Contains an address
- Allows the memory at that address to be manipulated
- Associates a type with the manipulated memory

To the computer, memory is just bits.

Programmers supply the meaning.

(Memory representations will be our next major topic.)

The special pointer value `NULL` represents an invalid address.

Pointer Syntax — Declaration

A pointer variable is marked with *.

```
char *str;
```

This is a pointer to char.
(`char *` is the idiomatic string type in C.)

A pointer may be marked const, in which case the memory it points to is const.[4]

```
const char *str;
```

[4] There is another type of constant pointer that we won't talk about now.

Pointer Syntax — Taking Addresses

A pointer may be created from a variable using &.
This is sometimes called the address-of operator.

```
int x = 42;
int *px = &x;
```

px is now a pointer to x.
(More on the implications of this later.)

Pointer Syntax — Dereferencing

A pointer is dereferenced with `*`, `->`, or `[]`.
(More on `->` when we get to structures.)

```
int *px = &x;
int y = *px;
```

- The variable `px` is created as a pointer to `x`, an integer.
- The variable `y` is created as an integer.
- `y` is assigned the value of `x` by dereferencing `px` with `*`.

A pointer can also be dereferenced like an array.

```
y = px[0];
```

- This is exactly the same as `y = *px;`.

Pointers and Arrays

Arrays and pointers are closely related in C.

You can often think of an array variable as a pointer to the first array element, and a pointer variable as an array.

However, they are not the same.

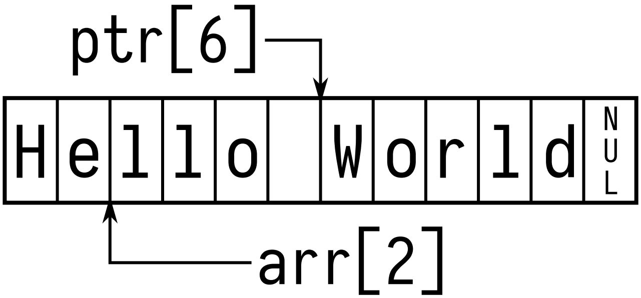
In both cases, dereferencing with `[i]` says

…add `i` times the size of the type of this variable to the base address (first element of the array or pointer value), then treat the memory at that location as if it is of the type of this variable.

Pointers and Arrays

Consider:

```
char arr[] = "Hello World";
char *ptr = arr;
```



Arrays Are Not Pointers

```
char arr[] = "string";
char arr2[] = arr;
```

“error: invalid initializer”

```
char arr[] = "Hello World";
char *ptr = arr;
```

`ptr` points to arr[0].

C/POSIX Text I/O

C and POSIX provide a wide variety of I/O facilities.

Among those are some convenient functions for reading and writing text.

There are also functions for binary data, and there is overlap.
(We'll discuss binary data later.)

This API is defined in the header `stdio.h`:

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

But first, a diversion for the standard I/O streams.

Standard I/O

POSIX I/O streams have the type `FILE *`.

These are both specific “files” that are open in every program, and facilities to manipulate those and other files.

Every POSIX process has three files that are always present:

- `stdin`
- `stdout`
- `stderr`

They may be closed, but they are defined.

Each stream has an underlying file descriptor, which we will discuss later.

stdin, stdout, stderr

- `stdin`: The process's standard input.
  This is the default location for reads. It is often the terminal, but may be any file or device on the system.
- `stdout`: The process's standard output.
  This is the default destination for non-error writes from the process. It is also often the terminal.
- `stderr`: The process's standard error.
  This is the default destination for error messages from the process. It is often the same destination as `stdout`, but need not be.

  The rules for `stderr` are slightly different. (More later.)

Basic Text I/O

There are several functions provided for basic textual I/O:

- `puts(str)`: prints a string to `stdout` with a trailing newline
- `fputs(str, stream)`: prints a string to a specified stream
- `printf(format, ...)`: prints a string to `stdout`, providing sophisticated formatting capabilities
- `fprintf(stream, format, ...)`: prints a string to `stream`, providing sophisticated formatting capabilities
- `gets()`: this function is dangerous, do not use it.
- `fgets(str, size, stream)`: Read a single line of text from the specified stream, but no more than `size - 1` bytes.
- `fscanf(stream, format, ...)`: Read complex formatted data from the specified stream

puts

```
#include <stdio.h>

int puts(const char *s);
int fputs(const char *s, FILE *fp);
```

The `puts()` functions write a simple string to a stream.
(`puts()` writes to `stdout`, `fputs()` is specified.)

The `puts()` version also automatically writes a trailing newline.

printf

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *fp, const char *format, ...);
```

The `printf()` family[5] is the workhorse of C textual output.

Like `puts()`, the unqualified version writes to `stdout`.

`printf()` accepts a specially formatted string that describes its output, plus a variable number of arguments.

The `...` token declares a function with a variable number of arguments. (Don't worry about the details for now.)

[5]And there are a ton of them!

Format Strings

Format strings can contain conversion specifiers starting with `%`.
(The special sequence `%%` represents a literal percent sign.)

Any other character (mostly) will print as itself.

`%d`	Integer
`%hd`	Short integer
`%ld`	Long integer
`%f`	Double
`%s`	String
`%c`	Character
`%x`	Hexadecimal Integer
`%p`	Pointer
…	Many others

Conversion Specifiers

Conversion specifiers can be very specific.

Their full form is: *%jfsw.plt*

j	Justification; use `-` for right-justified
f	Fill; use `0` to zero-fill, or space to space-fill
s	Sign; use `+` to display sign for all integers
w	Width; characters to reserve for a numeric
.p	Precision; digits after the decimal to print
l	Length; size of an integer word (`h`, `l`, `ll`)
t	Type

Example: `"%+02.2f"`

Print a `double` with two digits after the decimal, and if there are fewer than two digits before the decimal, print leading zeroes.

Some Conversions

```
printf("Pi is %.3f\n", 3.141592654d);

Pi is 3.142
```

Some Conversions

```
printf("Pi is %.3f\n", 3.141592654d);
```

```
Pi is 3.142
```

```
printf("printf: %p\n", &printf);
```

```
printf: 0x7f0c43536190
```

Some Conversions

```
printf("Pi is %.3f\n", 3.141592654d);

Pi is 3.142
```

```
printf("printf: %p\n", &printf);

printf: 0x7f0c43536190
```

```
char *name = "Ethan";
printf("My name is %s\n", name);

My name is Ethan
```

fgets

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *fp);
```

Reads up to `size - 1` bytes from `fp`, or until it reads a newline.

This is a convenient way to read lines of data, although if the buffer is not large enough, more than one read may be required.

`fgets()` returns `NULL` for errors or end of file.
Otherwise it returns `s`, which is non-`NULL`.

fscanf

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

Reads complex input according to the specified format string.

Conversions in the format string are similar to `printf()`.

Destination arguments are pointers and incoming text is type-converted.

The integer return value is the number of successful conversions.

Formatted I/O Example

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        double d;
        int conversions;

        printf("Input a floating-point number: ");
        conversions = scanf("%lf", &d);

        printf("There were %d successful
           conversions\n", conversions);
        printf("You entered: %f\n", d);
        return 0;
}
```

More on Text I/O

See the man pages!

The formatted I/O functions, in particular, have lots of options.

Recommended man pages:

```
man stdio
man printf
man scanf
```

Don’t try to use these functions for binary data!

The C Compiler Driver

First, we will ignore most stages of compilation.

The C compiler driver can take a `.c` source file and produce an executable directly.

We'll look at that with Hello World:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Compiling Hello World

We compile Hello World as follows:

```
gcc -Wall -Werror -O2 -g -std=c99 -o helloworld helloworld.c
```

This command says:

- `-Wall`: Turn on all warnings
- `-Werror`: Treat all warnings as errors
- `-O2`: Turn on moderate optimization
- `-g`: Include debugging information
- `-std=c99`: Use the 1999 ISO C Standard
- `-o helloworld`: Call the output `helloworld`
- `helloworld.c`: Compile the file `helloworld.c`

Compiling Hello World II

The C compiler driver ran all of the steps necessary to build an executable for us.

- The C preprocessor handled including a header
- The compiler produced assembly
- The assembler produced object code
- The linker produced `helloworld`

```
[elb@westruun]~/.../posix$ ./helloworld
Hello, world!
```

Compiling in Steps

The compiler driver can be used to invoke each step of the compilation individually.

It can also be used to invoke up to a step.

The starting step is determined by the input filename.

The ending step is determined by compiler options.

Recall that `-E` invoked the preprocessor.

Compiling to Assembly

Let's compile to assembly using `-S`:

```
$ gcc -Wall -Werror O2 -std=c99 -S helloworld.c
```

On the next slides, we'll examine the output from `helloworld.s`.

helloworld.s I

```
        .file   "helloworld.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Hello, world!"
        .section        .text.startup,"ax",@progbits
        .p2align 4,,15
        .globl  main
        .type   main, @function
```

We'll get to the details later, but for now notice:

- `.LC0:` is a local label
- `.string` declares a string constant (no newline!)
- The `.globl` and `.type` directives declare that we're defining a global function named `main`

helloworld.s II

```
main:
.LFB11:
        .cfi_startproc
        leaq    .LC0(%rip), %rdi
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
        call    puts@PLT
        xorl    %eax, %eax
        addq    $8, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

We'll skip the postamble, for now.

The Generated Code

First of all, you aren't expected to understand the code (yet).

```
leaq    .LC0(%rip), %rdi
```

This code loads the string constant's address (from .LC0).

Then, later:

```
call    puts@PLT
```

…it calls `puts()` to output the string.

Note that the C compiler:

- Noticed we were outputting a static string
- Noticed it ended in a newline
- Replaced the (complicated) `printf()` with the (simpler) `puts()` *and a modified string*

Compiling to an Object File

You may wish to compile to an object file.

This is used when multiple source files will be linked.

In this case, use `-c`:

```
$ gcc -Wall -Werror -O2 -std=c99 -c helloworld.c
```

This will produce `helloworld.o`.

Linking

Compiling any input files without an explicit output stage will invoke the linker.

```
gcc -Wall -Werror -O2 -std=c99 -o helloworld helloworld.o
```

This command will link `helloworld.o` with the system libraries to produce `helloworld`.

You can view the linkage with `ldd`:

```
[elb@westruun]~/.../posix$ ldd helloworld
  linux-vdso.so.1 (0x00007ffe34d1a000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f24dacbb00
  /lib64/ld-linux-x86-64.so.2 (0x00007f24db25c000)
```

Summary

- C is statically typed.
- C exposes many architecture details.
- C has no garbage collector, constructors, or destructors.
- The POSIX API is based on UNIX.
- POSIX provides an interface to the OS kernel.
- A C string is an array of characters.
- C and POSIX provide a rich text-based I/O API.
- Pointers allow direct access to memory by address.
- The “C compiler” is actually a chain of operations.

Next Time ...

- In-memory data representations
- C structures
- More on pointers

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 1: Intro, 1.1-1.3, Chapter 3: Intro, 3.2, Chapter 10: 10.10. Pearson, 2016.

Optional Readings

[2] John W. Eaton et al. *man — an interface to the on-line reference manuals*. man(1).

[3] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Second Edition. Prentice Hall, 1988.

License