C Guide

By Richard Yin

C is a static, compiled, non-objected-oriented low-level programming language – base functions in C are written to translate very efficiently to machine instructions, making C one of the fastest languages. However, they are also incredibly limited and basic, so it generally takes much longer to program the same task in C versus other languages.

My experience with learning C is suffering. The syntax is worse than Java's, the built-in functions are less readable, the code is harder to debug since C doesn't stop when errors are encountered, the productivity is magnitudes lower, and the code is 50% error checking.

Disclaimer: For the sake of legibility and conciseness, I have simplified typing for many functions – instead of short/long or custom types, I might just write int. I remove const indicators to save space too.

C Standard Library

Library	Usage
`#include <stdio.h>`	Input/output (print, get user input, open file)
`#include <stdlib.h>`	Dynamic memory, convert strings to numbers, sorting, simple math (*div*, *mod*, *abs*),
`#include <string.h>`	String operations (comparison, length, concatenate, find)
`#include <unistd.h>`	Pipe, fork, exec, read/write, sleep
`#include <signal.h>`	Signals
`#include <sys/….h>`	Miscellaneous
`#include <arpa/inet.h>`	Internet-related operations
`#include <math.h>`	Trigonometry, *e* and *log*, powers, rounding

Statements starting with # are not C code. They're processed before regular code compilation by C's preprocessor and usually at the top of the document.

Basic Data Types

There are four basic types with different sizes (bytes of space taken up) as given by the `sizeof` function:

Data Type	Symbol	Range/Precision	`sizeof`(Symbol) (in bytes)	Format Specifier
Character	`char`	−128 to 127	1	`%c`
Integer	`int`	−32768 to 32767	4	`%d`
Float	`float`	32-bit / ≈ 7 decimals	4	`%f`
Double	`double`	64-bit / ≈ 15 decimals	8	`%f`

Format Specifier: Of a type, a symbol representing it that must be used in order to print a variable of that type.

- eg. To print integer `a` and character `b`, write `printf("%d, %c", a, b)`

There are no **Boolean** types – 0 is false and non-0 is true.
There are no dictionaries or sets, not even in the standard library; only fixed-size arrays.
There is a special **null** value returned on errors by most built-in functions that usually return addresses *(see pointers)*.

There are four type modifiers (see here for more detail):

Symbol	Description	Example
`signed`	Allows positives & negatives; the default.	`signed int`
`unsigned`	Allows only positives – doubles the largest representable value.	`unsigned int`
`long`	Use double the bytes to allow for more precision.	`long int`
`short`	Use half the bytes to save space.	`short int`

Basic Operations

Most C operations and syntax are the same as Java, so use that as your reference point.
But, note that `NULL` is an address, which is different from an uninitialized variable.

Python	C	Explanation
`x // y`	`x / y`	C truncates division (ie. rounds towards 0).
`x / y`	`(float) x / y`	You must cast to a float or double.
`x ** y`	`pow(x, y)`	Exponents only exist as a math library function that works with doubles.

`a = b`
`if b == c:`
`  …`
	`if (a = b == c)`
	`  …`
		Braindead syntax that re-assigns `a` to `b` before checking `b == c`.
		Not to be confused from `a = b == c;` outside the if statement, which would set `a` to `b == c`.

Logical Operator	Description
`x && y`	AND operator.
`x || y`	OR operator.
`!x`	NOT operator.
`x & y`	Perform AND bit/element-wise on `x, y` in binary, convert output to decimal
`x | y`	Perform OR bit/element-wise on `x, y` in binary, convert output to decimal
`~x`	Perform NOT bit/element-wise on `x, y` in binary, convert output to decimal
`x ^ y`	Perform XOR bit/element-wise on `x, y` in binary, convert output to decimal
`x << n`	Shift `n` bits left the bits in `x`. Deletes left-most bits, inserts 0 on right.
`x >> n`	Shift `n` bits right the bits in `x`. Deletes right-most bits, inserts 0s on left.

Pointers and Arrays

Pointer: The address of an object in memory, which is an 8-byte hexadecimal number. Reuses the symbol *.

`int i;`	`i`'s uninitialized value is junk: whatever was previously in the address that's now allocated for `i`
`&i`	The address of `i`, AKA a pointer to `i`. Reuses the symbol `&`.
`int* p = &i;`	Set `p`, a pointer to an integer, to `&i`, the address of `i`. Conventionally, this is notated "`int *p`", but I believe this is braindead syntax.
`int** pp = &p;`	Set `pp`, a pointer to a pointer to an integer, to `&p`, the address of the pointer to `i`.
`int** pp = &&i;`	Error. The result `&i` isn't stored in memory yet, so calling `&(&i)` in the same line fails.
`sizeof(pp); // 8`	All pointers have 8 bytes allocated to them.
`*p`	Dereference `p`; equivalent to `i`. Reuses the symbol * again.
`*pp`	Dereference `pp` once; equivalent to `p`
`**pp`	Dereference `pp` twice; equivalent to `*p` or `i`
`*p = 9;`	Set to 9 whatever `p` is pointing to (mutating operation to primitives!); equivalent to `i = 9;`
`*p = *p + 2;`	Increase by 2 whatever `p` is pointing to (mutating operation)
`p = p + 2;`	Increase `p` by the size of 2 `int`'s in memory (ie. `2 * sizeof(int)`). `p` doesn't point to `i` now.
`int* q = *pp;`	Equal to `p`. The two * do different things! Garbage C syntax.
`i**(*pp+*p)*i`	A valid expression in C and why its syntax is garbage. Add `*pp` (integer pointer) and `*p` (integer), dereference the result, then multiply by `i` twice.

Array: An immutable ordered collection of one type of object. Assigned a fixed, contiguous block of memory.

`int hi[3];`	Creates array of length 3 with uninitialized junk values. Similar to Java syntax, except [] goes after variable name. It's stupid.
`int hi[3] = {1, 2, 3};` `int hi[] = {1, 2, 3};`	Memory addresses of elements are 4 bytes apart. This is a shortcut.
`sizeof(hi); // 12`	There are 3 elements, 4 bytes per element, thus 12 bytes total
`int* p = hi;`	When you refer to an array, you don't. You refer to item 0's address: `hi` is `&hi[0]`. So `int[]` is an integer array, and `int*` can be an integer pointer or integer array – no way to tell without looking at following code. Thanks, C.
`int p[3] = hi;`	Error!! Cannot set an int array of size 3 to an integer pointer. Use a for loop to copy...
`*p, p[0], hi[0]`	Refers to the first item of `hi`. `p[0]` only works if `p` is an array index (the program will know).
`*(p + 1), p[1], hi[1]`	Refers to the second item of `hi` Note that `p[i] = *(p + (i * sizeof(int)))`, so indexing is fancy dereferencing.
`hi[4]`	Not flagged by C as out-of-bounds; you'll access an unknown part of memory (unsafe)!
`p = p + 1;` `*p, p[0]`	Will now refer to the second item of `hi`

Strings

String: An array of characters with a hidden *null terminator* "`\0`" marking the final letter (ignore characters after it).

String Literal: A constant char pointer to a read-only part of memory containing the string (mutation causes **bus error**).

`char hi[5] = {'h', 'e', 'l', 'l', 'o'};` `char hi[5] = "hewwo";`	Create printable 5-character array `['h', 'e', 'l', 'l', 'o']` that isn't a string. String operations may still work on them.
`hi = "hello";`	Error. `Hi` is `%hi[0]`, so a memory address can't be set to a string.

`char hi[10];`
`hi[0] = 'h'; hi[1] = 'e'; hi[2] = 'w';`
`hi[3] = 'w'; hi[4] = 'o';`
 	
`hi[5] = '\0';`
	Create 10-character array. It is not a string right now.
	It starts with "hewwo".
	It ends with 5 characters of unitialized garbage.
	
	This line will turn `hi` into a string.

`char hi[] = "hewwo";`
`char hi[6] = "hewwo\0";`
`strlen(hi); // 5`
`sizeof(hi); // 6`
	Create a 6-character string of length 5.

`char hi[10] = "hewwo";`	Create 10-character string `"hewwo"` of length 5, where

```
strlen(hi);  // 5
sizeof(hi);  // 10
```
uninitialized characters are all set to `\0`

```
char* index = &hi[3];
index - hi
```
The number of characters between indices 0 and 3 of `hi`, 3.

```
char* hi = "hewwo";
hi = "hello";
```
Create string literal `hi` with memory address "hewwo"
Set `hi` to another string literal "hello".

`char[]` is an array of characters (and usually a string).
`char*` can either be a pointer to a character, a string literal, or an array of characters (and usually a string). Very fun.

Note that a `char s[]` can be passed to a function that accepts `char*` – and the function can mutate `s` if the code is written to do that. But if a `char* s` is passed instead, there will be an error.

Below are string functions. Many of them are unsafe if `s2` is longer than `s1` memory-wise.

Function	Return	Description
`strlen(char* s)`	`int`	Return string length, not including `\0`
`strcmp(char* s1,char* s2)`	`int`	String equality, but returns 0 if strings are equal instead of 1. So, `s1 == s2` in C looks like `!strcmp(s1, s2)`, which is stupid as hell.
`strtol(char* s, char** endptr, int base)`	`int`	Ignores initial white space, returns 1st number in string `s` as an integer in base `base`. Sets `endptr` to a pointer to the first character after the number if it is not null.
`strcpy(char* to, char* from)`	`char*`	Copy `from` into `to` (mutating it). Returns `to`.
`strncpy(char* to, char* from, int n)`	`char*`	Copy the first `n` characters of `from` into `to` (mutating it). Returns `to`.
`strcat(char* to, char* from)`	`char*`	Append `from` to the end of `to` (mutating it). Removes the first `\0` of `to`, adds `from`, then adds `\0` at the end. Returns `to`.
`strncat(char* to, char* from, int n)`	`char*`	Append the first `n` characters of `from` to the end of `to` (mutating it). Returns `to`.
`strchr(char* s, char c)`	`char*`	Find character `c` in `s`. Returns a pointer to `c` in `s`. The index can be found with `strchr(s, c) - s`.
`strstr(char* s, char* sub)`	`char*`	Finds substring `sub` in `s`. Returns a pointer to `sub` in `s`. Index can be found same way as `strchr`.
`memset(void* str, char c, int n)`	`void`	Copies char `c` to the first `n` characters of `str`.

Functions

```
int main() {
    // Do something
    return 0;
}
```
C's main function, if it takes no parameters from the command prompt.

The return is the **exit code**, which is 0 if all good, and not 0 if bad.

```
int main(int argc, char** argv) {
    // Do something
    return 0;
}
```
C's main function, if it takes parameters when ran from the command prompt in the format "`./cool_file param1 param2 …`"

`argc` is the number of parameters (including `cool_file`)
`argv` is the array of parameters (as strings).

Note that "`cool_file`" counts as a parameter.

```
int sum(int* A, int size) {
    int total = 0;
    for (int i = 0; i < size; i++) {
        total += A[i];
    }
    return total;
}
```
Recall as you pass an array to a function, you don't – you pass it as a pointer to the array's first index.

Arrays don't store size; it must be separately passed.

There is also no for-each loop in C.

```c
void largest(int** A, int size, int* largest_pt) {
    *largest_pt = **A;
    for (int i = 1; i < size; i++) {
        if (*A[i] > *largest_pt) {
            *largest_pt = *A[i];
        }
    }
}
```

This function mutates `largest_pt` to be the largest element pointed to in `A`, an array of pointers `int*`.

A non-mutating Python equivalent:

```python
largest_pt = A[0]
for i in range(1, len(A) + 1):
    if A[i] > largest_pt:
        largest_pt = A[i]
```

```c
int add(int i) {
    return i + 27;
}
int apply(int i, int (*f)(int)) {
    return f(i);
}
// Somewhere else
apply(20, add);   // 47
```

When you pass a function to a function, you don't – you pass it as a pointer.

The syntax is very hard to read, but it's essentially just a normal function syntax with (`*func_name`) in the middle and no names for input types.

```c
void (*f(char a))(int) {
   // Something
}
```

The coup de grace to C syntax's redeemability.
A function `f` that takes `char` input & returns a pointer to a function that takes `int` input & returns `void`.

```c
int (*g)(int) = add;
```

A function pointer.

```c
(*g)(10); // 37
```

Applying a function pointer.

```c
void (*f)(char* s, int c);
```

Pointer to a function that takes an input character and integer and returns nothing, called `f`.

```c
void *f(char* s, int c);
(void *) f(char* s, int c);
```

Function that takes an input character and integer and returns a void pointer, called `f`.

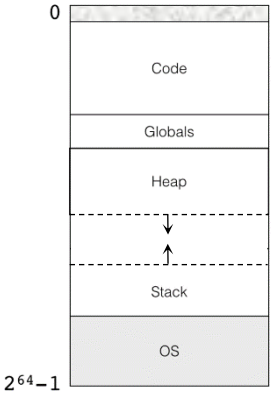
Names to the function inputs are optional.

```c
int increment() {
    static int i = 1;
    i += 1;
    return i;
}
// Somewhere else
int i = increment();   // i = 2
i = increment();       // i = 3
```

Static variables inside functions remember their value across multiple of those function calls.

Memory Model

In C, memory is a super-long array of all data, its address divided into segments:



Top: Temporary storage area (eg. for input/output AKA IO)
Code: Stores our program's code (in machine code form)
Globals: Stores global/static/constant variables, string literals – anything outside of a function or class. Also stores uninitialized variables.

Heap: Stores dynamically-allocated data that is manually allocated & freed at run-time.
Stack: Stores each function call (and its local variables) as an individual **stack frame**

- Top-most frame of stack is the currently-running function
- Heaps fill out top-to-bottom, stacks fill out bottom-to-top – **out of memory error** if available space is exceeded

OS: Stores memory for the OS and is inaccessible by most programs.

Segmentation Fault: An error when the program attempts to access forbidden parts of memory.
Bus Error: An error when the program tries access an invalid/non-existent memory address *(eg. changing a string literal)*
Buffer Overflow: An overflow of space in any part of the memory model *(eg. stack overflow, heap overflow)*
Stack Smashing: Error that C throws when user input (IO) exceeds buffer capacity

Dynamic Memory

Static Memory Allocation: When memory is automatically allocated/deallocated on the stack during compile-time.

- eg. Creating an array, which has to be fixed length in C

Dynamic Memory Allocation: When memory is manually allocated/deallocated on the heap during running-time

- eg. Creating linked lists, adding/removing items from it during running-time

```c
int* i_pt = malloc(sizeof(int));
*i_pt = 5;
```

Allocates `sizeof(int)` bits on the heap, exactly enough space for an `int`.

Returns a pointer to newly-allocated address (`NULL` if error) of type `void*`, as the program doesn't know what type we're allocating for.

```c
free(i_pt);
```

Deallocates all memory allocated for `i_pt`.
Doesn't change value stored on `i_pt`'s address, so `i_pt` still contains 5.

Memory Leak: When we lose the address to data stored on the heap – risks out of memory error, unsafe!
Dangling Pointer: A pointer to freed memory – value stored in address could change, unsafe!
Use-After-Free Error: When we dereference a dangling pointer
Double Free Error: When we free the same pointer twice

Structures

Composite Data Type: A data type constructable using primitive data types and other composite types.

```c
struct country {
    char continent[50];
    int population;
    float gdp;
};
```

This is a data class in Python, or a record in Java.
The fields are called **members** of the struct.
Arrays in structs must be fixed-size – you must dynamically allocate memory for varied sizes.

```c
struct country argentina;
```

We have to use `struct country` to refer to this type every time. Annoying! (See later for a fix)

```c
argentina.continent = "South America";
argentina.population = 45810000;
argentina.gdp = 487200000000;
```

Use dot notation like in classes.

```c
struct country argentina = {
    .continent = "South America",
    .population = 45810000,
    .gdp = 487200000000
}
```

Alternate way to instantiate a class.

`void cause_recession(struct country* c) {`
`  (*c).gdp /= 2;`
`  // c->gdp /= 2; alternate notation`
`}`
`cause_recession(argentina);`
`argentina.gdp  // 243600000000`
	Structs are passed as **copies**, unlike Python and Java classes.
	
	It is computationally faster to mutate attributes by passing pointers to structs.

`struct node {`
`  int item;`
`  struct node* next;`
`};`
`struct node* new_node(int item, struct node* next) {`
`  struct node* new = malloc(sizeof(node));`
`  (*new_node).item = item;`
`  (*new_node).next = next;`
`}`
`struct node* start = NULL;`
`start = new_node(3, start);`
`start = new_node(2, start);`
`start = new_node(1, start);`
	Structs allow recursive definitions. This is an example node of a linked list.
	
	This function builds a new node and returns a pointer to it.
	
	Initialize the last node (points to `NULL`) as `start`, then repeatedly setting `start` to the previous node.

Header Files

Prototype: A function without a body specified by `{}` that ends with `;`. Similar to Java interface functions. Used to import functions written in different files

Header File (.h): A file with prototypes, type definitions, and variables to be instantiated, like a Java interface. The type of the standard C library files.

Code	Description
`#include "filename.h"`	Non-header files with this line become one big file – share functions, variables, etc.
`#define FLAG`	Defines a macro/constant variable `FLAG` as 1 (default value).

`#ifdef FLAG`
`// compile this`
`#else`
`// compile that`
`#endif`
	Helps direct the compiler to compile the right functions based on whether `FLAG` is defined. This is useful, for compiling based on OS type, or to avoid duplicate #define statements in different files (which will not compile!).
	
	Note that `#ifdef FLAG` is equivalent to `#if defined(FLAG)`

`#ifndef FLAG`
`#define FLAG`
`// compile this`
`#endif`
	A safe way to start header files so that the file isn't compiled twice in a row.

`extern`	Keyword for variables that must be instantiated elsewhere (ie. in another file)
`static`	Reduces the scope of global variables to the file it's in – a get-around for duplicate global variable names in two connected files

Definitions and Macros

Name	Works On	Description
`typedef`	Data Types	Defines an alias/shortcut name
`#define`	Values, Functions	Defines a **macro** – a rule/pattern for replacing inputs with outputs. • Can be used to define aliases and constants. • C's preprocessor replaces all whole/space-separated instances of a macro's name (including those in comments, unless surrounded by "") with the macro's value, before the rest of the code is compiled. • Many predefined types in C have _t suffixes (eg. size_t for strlen(), sizeof())

`#define SCREEN_WIDTH 800`	Create a **constant** for the value 800, `SCREEN_WIDTH`

`#define true 1`
`#define false 0`
`typedef unsigned char bool;`
 	
`bool x = false;`
`unsigned char x = 0;`
	Create an **alias/shortcut name** for the value 1, `true`
	Create an **alias/shortcut name** for the value 0, `false`
	Create an **alias/shortcut name** for `unsigned char` (0 to 255), `bool`.
	
	Equivalent expressions!
	If you want a bool, don't actually do this. See here instead.

`#define MULT(x) ((x) * 1.2)`	Create a macro that's a function. • You need () around `x` and the whole expression so that order of operations still hold and `MULT(5 + 3) -> ((5 + 3) * 1.2)`. Expressions are not evaluated first!

`#if __APPLE__`
`char OS[] = "trash";`
`#elif __gnu_linux__`
`char OS[] = "ok";`
`#else`
`char OS[] = "based";`
`#endif`
	Predefined macros on the computer exist for telling apart OS systems (equal to 1 if the OS system is right). Might differ by computer.
	
	Undefined macros have truth values of false.
	
	The C preprocessor can execute different statements based on the macro values, or whether they are defined. For multi-line C, use \ like / in Python. Good for debugging levels, OS-dependent behaviour, etc.

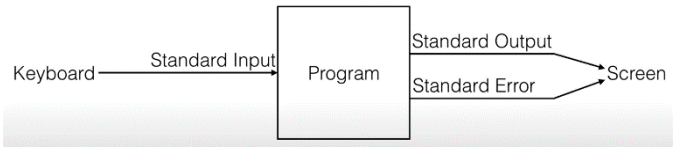
`void (*f(char a))(int) {`
`  // Something`
`}`
 	
`typedef void (*func)(int);`
`func f(char a) {`
`  // Something`
`}`
	Recall C's shit syntax. We can relieve our eyes using a `typedef`.

`typedef struct country_1 {`
`  char continent[50];`
`  int population;`
`  float gdp;`
`} country_2;`
 	
`struct country_1 yugoslavia;`
`country_2 yugoslavia;`
	Example of `typedef` used to shorten calls to structs.
	
	Equivalent! You can omit "`country_1`" in the `typedef`, but then only the second line "`country_2 yugoslavia;`" will work.

Streams

System Call: A function that directly requests a service from the OS (eg. `exit()`, `read()`, `write()`)
Library Call: A standard function accessible by default in a language. May include system calls (eg. IO functions).
Stream: The flow of information through a program

Stream	Default Location
Standard Input (`stdin`)	Keyboard
Standard Output (`stdout`)	Screen
Standard Error (`stderr`)	Screen



- In the shell, streams can be redirected to networks, devices, files, etc.
- The `FILE*` object includes `stdin`, `stdout`, and `stderror`

Below are some IO functions. Many are unsafe if not enough space is given to store inputs.

Function	Return	Description
`fopen(char filename[], char mode[])`	`FILE*`	Open `filename` as a pointer to a `FILE` object. `Mode` is "`r`" , "`w`", or "`a`" for reading, writing (creating new file, overwrites original content), and appending (editing), or "`r+`" as an example of combined modes.???? Add "`b`" at the end for binary files.
`fclose(FILE* stream)`	`int`	Closes a file, clear buffer memory, returns 0 if success.
`printf(char format[], …)`	`int`	Print `format`. To print non-string variables, format specifiers are used; variables must be put as extra arguments in the order they appear in `format`. Returns the number of characters printed.
`fprintf(FILE* stream, char format[], …)`	`int`	Identical to `fprintf`, but output is specified by `stream`.
`scanf(char format[], …)`	`int`	Takes user input of type `format` (use format specifiers!), and mutates what pointers in the … part point to. Returns number of successfully-matched variables. If piped by command prompt, can take file inputs. If called many times, `scanf` continues where it left before. Ignores whitespaces before the first non-whitespace. %*d means to find a digit and ignore it
`fscanf(FILE* stream, char format[], …)`	`int`	Identical to `fscanf`, but input is specified by `stream`.
`fgets(char str[], int n, FILE* stream)`	`char[]`	Takes first `n - 1` characters from `stream` (stop if end of file or `\n`), mutates `str` to store it there, adds `\0` at end. Returns `str` if successful, `NULL` if error.
`fgetc(FILE* stream)`	`int`	Returns the next character from `stream`. Casting to a char is recommended. Returns `EOF` (end-of-file) if empty.
`fwrite(void* ptr, int size, int nmemb, FILE* stream)`	`int`	Write `*ptr` to `stream`. Requires the size of each element in `*ptr` and how many values are being written (eg. for a length-5 integer array, arguments are `sizeof(int)`, `5`). Returns # of elements written (error if return ≠ `nmemb`)
`fread(void* ptr, int size, int nmemb, FILE* stream)`	`int`	Read `nmemb` items of size `size` in `stream`, store it in `*ptr`. Returns number of successfully-read elements (thus there's an error if return is not `nmemb`)
`fseek(FILE* stream, int offset, int whence)`	`int`	Change position of file `stream` that we are reading at. `whence` is one of three constants: ➢ `SEEK_SET`: Position 0 + `offset` ➢ `SEEK_CUR`: Current position + `offset` ➢ `SEEK_END`: End of file − `offset` Note that the read/get functions automatically increment the position we are reading at.
`rewind(FILE* stream)`	`void`	Shortform for `fseek(stream, 0, SEEK_SET)`

- Beware of output types like `EOF` (end-of-file) in special cases.
- When you write to a file, you write to the buffer part of memory, which is controlled by the OS and only periodically written to the disk (when? Depends on the stream).
    - If program ends unexpectedly, changes to a file might not be made.
    - `int fflush(FILE* stream)` manually gets OS to write everything on buffer to disk. Rarely used.
- Different computers may store binary data differently

Errors

Errno: Global integer variable used to store error types. Is mutated upon system call errors.

- System calls return values of `-1` *(for ints)* and `NULL` *(for pointers)* mean error
- C has a built-in header file with codes, numbers, and descriptions for different error types.

Function	Return	Description
`exit(int status)`	`void`	Exit the program with a given status code. Its use is encouraged as its purpose is clearer than `return 1` to programmers. It can also end the program from any function.
`perror(char[] s)`	`void`	Print `s`, followed by the current errno's message to standard error. Use after a system call error; otherwise it prints an undefined error.

Signals and Processes

Process: A running instance of a program *(ie. a task in the task manager)*

- **Running:** A process, if a CPU is executing it.
- **Ready:** A process, if it is ready to be executed and waiting for a CPU to become available.
- **Blocked:** A process, if it waiting for data to arrive from somewhere else *(eg. sleep, scan, file access)*
- Usually much more active processes than computer processes (or CPUs) – our OS gives the illusion of running them all simultaneously by switching between running/ready quickly.

Process Table: In Linux, a data structure containing all currently-running processes (ie. all PCBs)
Task/Process Control Block (PCB): A data structure with info about the state of a process

- **Process Identifier (PID):** A unique ID for a process
- **Stack Pointer (SP):** Pointer to the top of the stack
- **Program Counter (PC):** Pointer to next instruction in code to execute
- **Open File Table:** Table of all files open
- **Signal Table:** Table of all signals (see here for more). Stored as constants in `<signal.h>`

Core Dump: When a program prints a recorded state of working memory, usually in response to a crash
Signal: A software-generated interrupt sent to a process by the OS. Defined as constants in `<signal.h>`

#	Signal	Name	Default Action	Intended Purpose	Command
2	Interrupt	`SIGINT`	Terminate	Interrupt process.	Ctrl+C
3	Quit	`SIGQUIT`	Core dump	Terminate process normally.	Ctrl+\
6	Abort	`SIGABRT`	Core dump	Terminates a process abnormally.	`abort()`
9	Kill	`SIGKILL`	Terminate	Terminate process at all costs. Last resort.	
10	Bus	`SIGBUS`	Core dump	Throw bus error	
11	Seg Fault	`SIGSEGV`	Core dump	Throw segmentation default	
13	Pipe	`SIGPIPE`	Terminate	Throw error from writing on a pipe with no reader	
15	Terminate	`SIGTERM`	Terminate	Terminate process, allow time for some cleanup	
17	Stop	`SIGSTOP`	Pause	Pause a process; not ignorable.	
18	User Stop	`SIGTSTP`	Pause	Pause a process.	Ctrl+Z
19	Continue	`SIGCONT`	Ignore	Continue after a stop signal.	

- Signals, like errors, can be caught and ignored, with the default action rerouted to something else
    - Kill and Stop are special – they cannot be ignored, and their default action cannot be changed

Signal Mask: A set of signals the process will block/ignore until later unblocked.

Code	Return	Description

`struct sigaction {`
`  void (*sa_handler)(int);`
`  void (*sa_sigaction)(int, siginfo_t*, void*);`
`  sigset_t sa_mask;`
`  int sa_flags;`
`  void (*sa_restorer)(void);`
`}`
		A struct detailing how to handle a signal. Don't worry too much on the details.
		`sa_handler` is the function called when the signal is received (whose input is the signal number). Allows `SIG_DFL` (default action) & `SIG_IGN` (ignore).
		The signal mask should be set empty with sigemptyset() – basically, signals this handler should ignore in case other signals arrive during execution of the handler function
		The signal flags should be set to 0 (default)

`sigaction(int signum,`
`  struct sigaction* action,`
`  struct sigaction* old_action)`
	`int`	If `action` is not `NULL`, modify signal action struct of signal with number `signum` to `act`.
		If `old_action` is not `NULL`, store the old signal action struct in `*oldact`.
		Return 0 if success, -1 if error (and set errno).

`sigemptyset(sigset_t* set);`	`int`	Initialize `*set`, a set of signals, to be empty. Return 0.
`sigaddset(sigset_t* set, int signum);`	`int`	Add the signal with number `signum` to `*set`. Return 0 on success, -1 on failure.
`sigprocmask(int how, sigset_t* set,` `  sigset_t* oldset);`	`int`	`how` takes on three possible values: `SIG_BLOCK` – Add signals in `set` to blocked signals `SIG_UNBLOCK` – Delete signals in `set` from blocked sigs `SIG_SETMASK` – Make `set` the set of blocked signals If `set` is `NULL`, nothing happens. If `oldset` is not `NULL`, mutate it to be the previous set of blocked values. Return 0 on success, -1 if error. If sent, blocked signals will be ignored by the program regardless of handler functions.

Function	Return	Description
`fork()`	`int`	Creates child process in exact same state, same PCB info, except with different PID. If successful, returns 0 in child, and child process's PID (positive int) in parent. If failed, returns -1 in the parent. There is no guarantee for which process runs first. The OS decides that.
`getpid()`	`int`	Return the current process's PID
`getppid()`	`int`	Return the current process's parent's PID
`getuid()`	`int`	Return the user ID running the current process
`getgid()`	`int`	Return the group ID of the user ID running the current process
`usleep(int usec)`	`int`	Suspend execution for at least `usec` microseconds (1,000,000 microseconds = 1 second). Only allows inputs of at least 1,000,000.

`waitpid(int pid,`
`  int* status,`
`  int options)`
	`int`	Suspend execution until the…
		`pid > 0`: Termination of the child process with PID `pid`.
		`pid == 0`: Termination of any child process with the same process group ID
		`pid == -1`: Termination of any child process
		`pid < -1`: Termination of any child process with `abs(process group ID) == pid`
		If `status != NULL`, sets `*status` equal to status info of terminated child. If child returns 0, `status` is 0. Otherwise, use macros to help read the status info here.
		`options` can take on four possible values:
		`0`: Default settings
		`WNOHANG`: Tries to return status info *immediately*, without suspending execution

		`WUNTRACED:` Suspend execution until a child process terminates *or stops* `WCONTINUED:` Suspend execution until a child process terminates *or resumes* If successful, return pID of the child whose status info is obtained If `WNOHANG` and some child's status info is not available, return 0. If unsuccessful, return -1 and set errno.
`wait(int* status)`	`int`	Suspend execution until any child terminates. Same as `waitpid(-1, status, 0)`.
`WIFEXITED(int status)`	`int`	Returns 1 if the status is a normal exit, 0 otherwise.
`WEXITSTATUS(int status)`	`int`	Returns exit status of status

`execlp(char file[],`
`char arg0[],`
`…, NULL)`
	`int`	Calls the linux command/run file `file` with arguments `arg0, arg1, …`
		Must have `NULL` as the last argument for some reason.
		
		Does not create a new process. Passes control to the OS to overwrites new code into the code region of memory and runs it with a new stack frame. So pre-existing code disappears and any line of code after `execlp` does not run.
		
		If successful, the command executes. If unsuccessful, return -1 and set errno.
		There're many variations of this command; see here for more.
		- v: pass arguments as an an array of strings, ending with NULL
		- p: don't use PATH environment variable – folders must be specified by full location
		- l:

Init Process: The first process launched by an OS when it loads. Has PID of 1.
Orphan: A child process that exits **after** the parent exits. The child's parent becomes the init process.
Zombie: A child process that exits **before** the parent collects termination status. Will stay in process table until…

- Termination status is collected by the parent process (using `wait`)
- Parent processes ends, zombie is orphaned, and init process collects termination statuses and closes them

`int c = fork();`
`if (c > 0) {`
`  // Parent process`
`} else if (c == 0) {`
`  // Child process`
`} else {`
`  // Error happened!`
`}`
	Basic outline of using a fork.

`int status;`
`int child_pid = wait(&status);`
`if (WIFEXITED(status)) {`
`  int child_status = WEXITSTATUS(status);`
`  // Do something`
`} else {`
`  // Error happened in child!`
`}`
	Example of code in the parent process that finds the first child process that ends, checks if it ended properly, and does something using the exit status of the child.

Pipes

File Descriptor (fd): An integer assigned by the OS as a key for a specific open file/communication channel.

- 0 is standard input
- 1 is standard output
- 2 is standard error

Pipe: An object outside a process that allows one-way communication between parent/child processes.

- Two way works, but it wonky and bad and unreliable and don't do it

`int fd[2];` `pipe(fd);`	Create a pipe, setting `fd[0]` to the file descriptor for reading the pipe and `fd[1]` to the fd for writing to the pipe.
`int x = 50;` `write(fd[1], &x, sizeof(int));`	Write `x` to the writing part of the pipe. `write`/`close` are called by other IO functions (eg. `fread`) as a helper. Write calls will fail if the read end is closed.

`int y;` `read(fd[0], &y, sizeof(int));`	Read `x` from the reading part of the pipe and store it in `y`. Returns `EOF` if the pipe is empty and `0` if the write end of the pipe has been closed.

`int c = fork();`
`if (c > 0) {  // Parent process`
`  close(fd[0]);  // Close read`
`  // Write something to child!`
`  close(fd[1]);`
`} else if (c == 0) { // Child process`
`  close(fd[1]);  // Close write`
`  // Read something from parent!`
`  close(fd[0]);`
`} else {`
`  // Error happened`
`}`
	Using a pipe with a child process.
	The parent closes the pipe's reading end – it only writes to the pipe.
	The child closes the pipe's writing end – it only reads from the pipe.
	
	Pipes are one-way – you must have exactly 1 read/write end open when calling write/read, otherwise there is an error.
	
	Close pipes after use – there's a limit to fds openable at once
	
	The OS makes sure read/write calls do not occur simultaneously. Read calls block if the pipe is empty and write calls block if the pipe is full.
	
	Pipe ends are automatically closed when the program returns.

`int filefd = open("hi.txt", O_RDONLY);`	Open `"hi.txt"` with read-only permission, storing its fd in `filefd`. Flags are separated with |. See this for more flags.
`dup2(fd, STDOUT_FILENO);`	Changes file descriptor for standard output to the file descriptor `fd`. In other words, redirects standard output to `fd`. Also consider using `STDIN_FILENO`, `fileno(stdin)`, `fileno(stdout)`.

`fd_set read_fds;`
`fd_set write_fds;`
`fd_set except_fds;`
	An `fd_set` is a set of file descriptors implemented as a bit array *(ie. a big binary number, where digit k is 1 if value/fd k is in the set)*

`FD_ZERO(&read_fds);`
`FD_SET(a, &read_fds);`
`FD_SET(b, &read_fds);`
	Initializes the bit array to be empty.
	Add file descriptor `a` to the bit array.
	Add file descriptor `b` to the bit array.

`select(n, &read_fds, &write_fds,` `  &except_fds, NULL);`	Block until the timer (last argument, `struct timeval*`) expires (if it is not `NULL`), OR until there is activity in the first `n` fds of the given bit arrays (`NULL` to not check activity) – ie. when fds in read/write sets are ready for reading/writing & fds in except set have exceptional conditions. Modifies relevant sets to only contain the file descriptors with changed status. Returns number of ready fds, -1 if error.

`for (int fd = 0; fd <= n; fd++) {`
`  if (FD_ISSET(fd, &read_fds))`
`  // Do something`
`}`
	Checks if file descriptors `fd` are inside set `read_fds`. Helps minimize errors where parent reads many pipes to many children, gets stuck waiting for a child to write while another is finished.

Sockets

Internet Protocol (IP) Address: Address of a computer connected to the internet. Can be used to send messages to it.

- Machines have different addresses per network
- Machine has address for itself – localhost (127.0.0.1)

Port: Location of a program in a computer connected to the internet.

- **Port Number:** An extension to IP address representing a port's address. Ranges from 0 – 65535
  - **Private/Dynamic Ports:** For personal use, used on per-request basis, 49,152 – 65,535
  - **Registered Ports:** Reserved by *Internet Assigned Numbers Authority* (IANA), 1024 – 49,151
  - **Well-Reserved Ports:** Reserved for common TCP applications, 0 – 1023
    - eg. http has 80, https is 443

Packet: Messages sent between computers. Contain an address and contents, but no exact route.
Router: Devices that direct/facilitate transfer of packets across networks

Server/Host: Programs running usually on a predefined port of a machine. Receives messages from clients
Client: Programs that interact with a server – can send message, start conversation/communication channel, etc.

Protocol: A defined system of rules for transmitting data
TCP Protocol: A standard that defines how to establish/maintain a network conversation

Socket: An endpoint of a communication link between two programs. Uses `#include <sys/socket.h>`

Stream Socket: A connection-mode socket (ie. it establishes a communication session before transferring useful data) built on TCP protocol. Guarantees no loss, in-order delivery of information.

Byte Ordering: The way the bytes are sorted in machines

- **Little Endian:** Ordering with least significant byte first – right-to-left ordering of bytes
- **Big Endian:** Ordering with most significant byte first – left-to-right ordering of bytes
  - Binary numbers are stored as $2 + 4 + 8 + \cdots$; the "least significant" byte is the byte representing 2.

Network Byte Order: Byte order used conventionally for transmitting data over the network. Used when giving ports/addresses to socket functions. Big-endian.

Host Byte Order: Byte order that is most natural for the host software/hardware. Can be either ordering.

Network Newline: Convention for text data transmitted over networks where lines end with "`\r\n`"

Function	Return	Description
`socket(int domain, int type, int protocol)`	`int`	Create a socket with `type` and communication channel `domain` and the `protocol` it follows (0 for default), usually constants. Once connected, you can read/write/close a socket like a pipe. Our focus is `SOCK_STREAM` (ie. stream socket) for type and `AF_INET` (ie. internet) for the domain. Return the socket's file descriptor if successful, -1 otherwise.
`htons(int hostshort)`	`int`	Convert short integer in host byte order to network byte order. Requires `#include <arpa/inet.h>`
`struct in_addr { int s_address; };`		A struct representing an internet address. Use `INADDR_ANY` for any address, `AF_INET`, or something else.
`struct sockaddr_in { int sin_family; int sin_port; struct in_addr sin_addr; char sin_zero[8]; };`		A struct that is a socket's internet address, consisting of its ➢ IP address (always set to AF_INET) ➢ Port number – use `htons` here! ➢ Address to connect to (as a client) or accept (as a server) ➢ 8 characters of padding – for security, since we send this info over the internet, set it to 0 first!
`bind(int socket_fd, struct sockaddr* address, int address_len)`	`int`	Assign `address` to the socket with file descriptor `socket_fd`. `struct sockaddr` is a generic form of `struct sockaddr_in`, so we need to cast in order to use it. Last argument is address' size, `sizeof(struct sockaddr_in)`. Return 0 on success, -1 on failure – important to check if port is already used for something else!
`listen(int socket_fd, int backlog)`	`int`	Open `socket_fd` to accept new connections. It takes time to setup server/client connections – `backlog` is the maximum pending connections `socket` should hold. Return 0 on success, -1 on failure.
`accept(int socket_fd, struct sockaddr* address, int* address_len)`	`int`	Blocks until a connection is detected to socket `socket_fd`, and accepts it. Stores client's address in `address` and its length in `*address_len`, whose initial value you should have initialized to `sizeof(struct sockaddr_in)`. On success, return the file descriptor for the newly-generated socket for communicating with the client. On failure, -1.
`struct addrinfo { int ai_flags; int ai_family; int ai_socktype; int ai_protocol;`		Struct containing info about an internet address: ➢ Flags (more here) ➢ Desired address family (eg. `AF_INET`) ➢ Preferred socket type ➢ Protocol

`int ai_addrlen;`
`struct sockaddr* ai_addr;`
`char* ai_canonname;`
`struct addrinfo* next;`
`};`
		➢ Size of socket's address
		➢ Pointer to socket's address
		➢ Official name of host
		➢ The next `addrinfo` struct (linked list structure)

`getaddrinfo(char* host,`
`char* service,`
`struct addrinfo* hints,`
`struct addrinfo* result)`
	`int`	Allocates and initializes linked list for each network address matching `host` (eg. "www.whatever.com") and `service` (the port number as a string), filtering with `hints.ai_flags`, setting `*result` to the linked list's front.
		Arguments can be specified `NULL` for nothing to happen.
		Return 0 on success, -1 on failure.

`freeaddrinfo(struct addrinfo* result)`	`void`	Free the linked list structure `addrinfo`.
`inet_pton(int af, char* from,` `in_addr* to)`	`int`	Converts IP address `from` into an `in_addr` object, storing it in `to`. `af` is either `AF_INET` or `AF_INET6`. Return 1 on success, 0 if bad IP address, -1 if bad address family (af).

`connect(int socket_fd,`
`struct sockaddr* address,`
`int address_len)`
	`int`	Initiate a connection from socket `socket_fd` to a known address `address` with length `address_len`.
		Return 0 on success, -1 on failure.

```c
int soc = socket(AF_INET, SOCK_STREAM, 0);

// Initialize data of server to connect to
struct sockaddr_in server;
memset(&(server.sin_zero), 0, sizeof(addr.sin_zero));  // Padding
server.sin_family = AF_INET;      // Set connection type to internet
server.sin_port = htons(69);      // Set server port to 69

// Get server info in order to find IP address
struct addrinfo* info;
getaddrinfo("some_server.something", NULL, NULL, &info);
server.sin_addr = ((struct sockaddr_in *) info->ai_addr)->sin_addr;
freeaddrinfo(info);

connect(soc, (struct sockaddr *) &server, sizeof(struct sockaddr_in));
```

Client-side code

The port number and website link must be known in advance.

Get website info, find IP of website and store it in `server.sin_addr`

Real code would also test for errors (-1 values).

```c
int soc = socket(AF_INET, SOCK_STREAM, 0);

// Initialize server address settings
struct sockaddr_in address;
memset(&server.sin_zero, 0, 8);
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(69);

// Link server to address, open socket to connections
bind(soc, (struct sockaddr *) &address, sizeof(struct sockaddr_in));
listen(soc, 5);

// Create variables to store client data
struct sockaddr_in client_address;
client_address.sin_family = AF_INET;
unsigned int client_len = sizeof(struct sockaddr_in);

// Block until a connection is received
int client_soc = accept(soc, (struct sockaddr *) &client_address,
                        &client_len);
```

Server-side code

To summarize,

Server-side:

- Socket – create socket
- Bind – bind socket to local address, port
- Listen – listen for new connections
- Accept – establish the new connection
- Read/Write – talk!
- Close – shutdown

Client-side:

- Socket – create socket
- Connect – connect to server
- Read/Write – talk!
- Close – shutdown