HARDWARE (COMPILERS, OS, RUNTIME) AND C, C++, TO CUDA

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Date: 1 Nov 2017.
Key words and phrases. C, C++, CUDA, CUDA C/C++, Compilers, OS, runtime, classes, Object Oriented Programming, Segmentation Fault, memory, memory addresses.

Abstract. I review what, how, and why Segmentation Fault is and occurs in the specific context of C and hopefully to CUDA, virtual memory addresses in C++, and to CUDA.

1. Introduction; why I'm writing this

I didn't realize the importance of having a profound understanding of C/C++ and its relation to hardware, in particular memory (addresses), until I learned more specifically about the work done at NVIDIA from its news and job postings. Coming from a theoretical and mathematical physics background, I wanted to get myself up to speed as fast as possible, provide good textbook and online references, and directly relate these topics, which seem to be classic topics in computer science (at hardware level) and electrical engineering, to CUDA, to specifically GPU parallel programming with direct CUDA examples.

Note that there is a version of this LaTeX/pdf file in a "grande" format (not letter format but "landscape" format) that is exactly the same as the content here, but with different size dimensions.

I began looking up resources (books, texts, etc.) Recommend a text that explains the physical implementation of C (stack, heap, etc) [closed], and a Mehta recommended books including Hyde (2006) [4].

> "When programmers first began giving up assembly language in favor of using HLLs, they generally understood the low-level ramifications of the HLL statements they were using and could choose their HLL statements appropriately.Unfortunately, the generation of computer programmers that followed them did not have the benefit of mastering assembly language. As such, they were not in a position to wisely choose statements and data structures that HLLs could efficiently translate into machine code."

pp. 2, Ch. 1 of Hyde (2006) [4], where HLL stands for high-level language. I was part of that generation and knew of nothing of assembly.

Bryant and O'Hallaron (2015) [1]

2. 80x86 Assembly; 80x86 Architecture

cf. Ch. 3 of Hyde (2006) [4]

Main difference between complex instruction set computer (CISC) architectures such as 80x86 and reduced instruction set computer (RISC) architectures like PowerPC is the way they use memory. RISC provde relatively clumsy memory access, and applications avoid accessing memory. 80x86 access memory in many different ways and applications take advantage of these facilities.

2.1. Basic 80x86 Architecture.

Intel CPU generally classified as a *Von Neumann machine*, containing 3 main building blocks:

(1) CPU
(2) memory
(3) input/output (I/O) devices

These 3 components are connected together using the *system bus*. System bus consists of

- address bus
- data bus
- control bus

CPU communicates with memory and I/O devices by placing a numeric value on the address bus to select 1 of the memory locations or I/O device port locations, each of which has a unique binary numeric address.
Then the CPU, I/O, and memory devices pass data among themselves by placing data on the data bus.
Control bus contains signals that determine the direction of data transfer (to or from memory, and to or from an I/O device).

So

$$\textbf{Memory}$$

$$\mathrm{Obj}(\textbf{Memory}) = \text{ memory locations}$$

$$\textbf{I/O}$$

$$\mathrm{Obj}(\textbf{I/O}) = \text{ I/O device port locations}$$

and

$$\textbf{address bus}$$

$$\mathrm{Obj}(\textbf{address bus}) = \text{ unique binary numeric addresses}$$

Likewise,

$$\textbf{data bus}$$

$$\mathrm{Obj}(\textbf{data bus}) = \text{ data}$$

And so

$$\mathrm{CPU} : \textbf{Memory} \times \textbf{address bus} \to \mathrm{Hom}(\textbf{Memory}, \textbf{address bus})$$

$$\mathrm{CPU} : (\text{memory location}, \text{unique binary numeric address}) \mapsto \&$$

in the language of category theory.

2.2. **Registers (for 80x86).** cf. 3.3 Basic 80x86 Architecture, from pp. 23 of Hyde (2006) [4].

Example: to add 2 variables, x, y and store $x + y$ into z, you must load 1 of the variables into a register, add the 2nd. operand to the register, and then store register's value into destination variable. "Registers are middlemen in almost every calculation." Hyde (2006) [4].

There are 4 categories of 80x86 CPU registers:

- general-purpose
- special-purpose application-accessible
- segment
- special-purpose kernel-mode

Segment registers not used very much in modern 32-bit operating systems (e.g. Windows, Linux; what about 64-bit?) and special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools.

2.2.1. *80x86 General-Purpose Registers.* 80x86 CPUs provide several general-purpose registers, including 8 32-bit registers having the following names: (8 bits in 1 byte, 32-bit is 4 bytes)

$$EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP$$

where E is *extended*; 32-bit registers distinguished from 8 16-bit registers with following names

$$AX, BX, CX, DX, SI, DI, BP, SP$$

and finally

$$AL, AH, BL, BH, CL, CH, DL, DH$$

Hyde says that it's important to note about the general-purpose registers is they're not independent. 80x86 doesn't provide 24 separate registers, but overlaps the 32-bit registers with 16-bit registers, and overlaps 16-bit registers with 8-bit registers. e.g.

$$\&AL \neq \&AH, \text{ but}$$
$$\&AL, \&AH \in \&AX \text{and } \&AX \in \&EAX, \text{ so that}$$
$$\&AL, \&AH, \&AX \in \&EAX$$

3. Compilers; Compiler Operation and Code Generation

cf. Ch. 5 Compiler Operation and Code Generation, from pp. 62 and pp. 72- of Hyde (2006) [4]

4. gdp; good debugging processes (in C/C++/CUDA)

Learn C the hard way Lecture 4, Using Debugger (GDB)

5. Pointers in C; Pointers in C categorified (interpreted in Category Theory) and its relation to actual, physical, computer memory and (memory) addresses ((address) bus; pointers, structs, arrays in C

From Shaw (2015) [5], Exercise 15,

e.g. `ages[i]`, You're indexing into array `ages`, and you're using the number that's held in i to do it:

$$a : \mathbb{Z} \to \text{Type} \in \mathbf{Type} \quad \text{e.g.} \quad a : \mathbb{Z} \to \mathbb{R} \text{ or } \mathbb{Z}$$
$$a : i \mapsto a[i] \qquad\qquad a : i \mapsto a[i]$$

Index $i \in \mathbb{Z}$ is a location *inside* `ages` or a, which can also be called *address*. Thus, $a[i]$.

Indeed, from cppreference for Member access operators,

Built-in *subscript* operator provides access to an object pointed-to by pointer or array operand. And so `E1[E2]` is exactly identical to `*(E1+E2)`.

To C, e.g. `ages`, or a, is a location in computer's memory where, e.g., all these integers (of `ages`) start, i.e. where a starts.

$\mathbf{Memory}$, $\quad \text{Obj}(\mathbf{Memory}) \ni$ memory location. Also, to specify CPU,

$\mathbf{Memory}_{CPU}$, $\quad \text{Obj}(\mathbf{Memory}_{CPU}) \ni$ computer memory location

It's *also* an address, and C compiler will replace e.g. `ages` or array a, anywhere you type it, with address of very 1st integer (or 1st element) in, e.g. `ages`, or array a.

$$\mathbf{Arrays} \overset{\cong}{\longleftrightarrow} \mathbf{address}$$
$$\text{Obj}(\mathbf{Arrays}) \overset{\cong}{\longleftrightarrow} \text{Obj}(\mathbf{address})$$
$$a \overset{\cong}{\longleftrightarrow} \texttt{0x17}$$

"But here's the trick": e.g. "`ages` is an address inside the *entire computer*." (Shaw (2015) [5]).

It's not like i that's just an address inside `ages`. `ages` array name is actually an address in the computer.

"This leads to a certain realization: C thinks your whole computer is 1 massive array of bytes."

"What C does is layer on top of this massive array of bytes the concept of types and sizes of those types." (Shaw (2015) [5]).

Let

$$\mathbf{Memory}_{CPU} := 1 \text{ massive array of bytes}$$
$$\text{Obj}(\mathbf{Memory}_{CPU})$$

$$\mathbf{Type}$$
$$\text{Obj}(\mathbf{Type}) \ni \texttt{int}, \texttt{char}, \texttt{float}$$
$$\text{Obj}(\mathbf{Type}) \xrightarrow{\texttt{sizeof}} \mathbb{Z}^+$$
$$T \xmapsto{\texttt{sizeof}} \texttt{sizeof}(T)$$
$$\texttt{float} \xmapsto{\texttt{sizeof}} \texttt{sizeof}(\texttt{float})$$

How C is doing the following with your arrays:

- *Create* a block of memory inside your computer:

$$\text{Obj}(\mathbf{Memory}_{CPU}) \supset \text{ Memory block}$$

  Let $\text{Obj}(\mathbf{Memory}_{CPU})$ be an ordered set. Clearly, then memory can be indexed. Let $b \in \mathbb{Z}^+$ be this index. Then $\text{Memory block}(0) = \text{Obj}(\mathbf{Memory}_{CPU})(b)$.
- *Pointing* the name `ages`, or a, to beginning of that (memory) block.
  Entertain, possibly, a category of pointers, **Pointers** $\equiv$ **ptrs**.

$$\mathbf{ptrs}$$
$$\text{Obj}(\mathbf{ptrs}) \ni a, \text{ e.g. } \texttt{ages}$$
$$a \mapsto \text{Memory block}(0)$$
$$\text{Obj}(\mathbf{ptrs}) \to \text{Obj}(\mathbf{Memory}_{CPU})$$

- *indexing* into the block, by taking the base address of `ages`, or a

$$a \overset{\cong}{\longmapsto} \text{ base address } 0x17$$
$$\text{Obj}((T)\mathbf{array}) \overset{\cong}{\to} \text{Obj}(\mathbf{addresses})$$
$$a[i] \equiv a + i \overset{\cong}{\longmapsto} \text{ base address + i * } \texttt{sizeof}(T) \overset{*}{\mapsto} a[i] \in T \text{ where } T, \text{ e.g. } T = \mathbb{Z}, \mathbb{R}$$
$$\text{Obj}((T)\mathbf{array} \overset{\cong}{\to} \text{Obj}(\mathbf{addresses}) \to T$$

"A pointer is simply an address pointing somewhere inside computer's memory with a type specifier." Shaw (2015) [5]

C knows where pointers are pointing, data type they point at, size of those types, and how to get the data for you.

5.0.1. *Practical Pointer Usage.*

- Ask OS for memory block (chunk of memory) and use pointer to work with it. This includes strings and `structs`.
- Pass by reference - pass large memory blocks (like large structs) to functions with a pointer, so you don't have to pass the entire thing to the function.
- Take the address of a function, for dynamic callback. (function of functions)

"You should go with arrays whenever you can, and then only use pointers as performance optimization if you absolutely have to." Shaw (2015) [5]

5.0.2. *Pointers are not Arrays.* No matter what, pointers and arrays are not the same thing, even though C lets you work with them in many of the same ways.

From Eli Bendersky's website, Are pointers and arrays equivalent in C?

He also emphasizes that

5.0.3. *Variable names in C are just labels.* "A variable in C is just a convenient, alphanumeric pseudonym of a memory location." (Bendersky, [6]). What a compiler does is, create label in some memory location, and then access this label instead of always hardcoding the memory value.

"Well, actually the address is not hard-coded in an absolute way because of loading and relocation issues, but for the sake of this discussion we don't have to get into these details." (Bendersky, [6]) (EY : 20171109 so it's on the address bus?)

Compiler assigns label *at compile time*. Thus, the great difference between arrays and pointers in C.

5.0.4. *Arrays passed to functions are converted to pointers.* cf. Bendersky, [6].

Arrays passed into functions are always converted into pointers. The argument declaration `char arr_place[]` in

```
void foo(char arr_arg[], char* ptr_arg)
{
char a = arr_arg[7];
char b = ptr_arg[7];
}
```

is just syntactic sugar to stand for `char* arr_place`.

From Kernighan and Ritchie (1988) [7], pp. 99 of Sec. 5.3, Ch. 5 Pointers and Arrays,

> When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

Why?

The C compiler has no choice here, since,
array name is a label the C compiler replaces *at compile time* with the address it represents (which for arrays is the address of the 1st element of the array).

But function isn't called at compile time; it's called *at run time*.
At run time, (where) something should be placed on the stack to be considered as an argument.

Compiler cannot treat array references inside a function as labels and replace them with addresses, because it has no idea what actual array will be passed in at run time.

EY : 20171109 It can't anticipate the exact arguments that'll it be given *at run-time*; at the very least, my guess is, it's given instructions.

Bendersky [6] concludes by saying the difference between arrays and pointers does affect you. One way is how arrays can't be manipulated the way pointers can. Pointer arithmetic isn't allowed for arrays and assignment to an array of a pointer isn't allowed. cf. van der Linden (1994) [8]. Ch. 4, 9, 10.

Bendersky [6] has this one difference example, "actually a common C gotcha":

"Suppose one file contains a global array:"

```
char my_Arr[256]
```

Programmer wants to use it in another file, *mistakingly* declares as

```
extern char* my_arr;
```

When he tries to access some element of the array using this pointer, he'll most likely get a segmentation fault or a fatal exception (nomenclature is OS-dependent).

To understand why, Bendersky [6] gave this hint: look at the assembly listing

```
char a = array_place[7];

0041137E  mov  al,byte ptr [_array_place+7 (417007h)]
00411383  mov  byte ptr [a],al

char b = ptr_place[7];

00411386  mov  eax,dword ptr [_ptr_place (417064h)]
0041138B  mov  cl,byte ptr [eax+7]
0041138E  mov  byte ptr [b],cl
```

or my own, generated from `gdb` on Fedora 25 Workstation Linux:

```
0x00000000004004b1 <main+11>:    movzbl 0x200b8f(%rip),%eax        # 0x601047 <array_place+7>
0x00000000004004b8 <main+18>:    mov    %al, 0x1(%rbp)

0x00000000004004bb <main+21>:    mov    0x200be6(%rip),%rax        # 0x6010a8 <ptr_place>
0x00000000004004c2 <main+28>:    movzbl 0x7(%rax),%eax
0x00000000004004c6 <main+32>:    mov    %al, 0x2(%rbp)
```

"How will the element be accessed via the pointer? What's going to happen if it's not actually a pointer but an array?" Bendersky [6]

EY : 20171106. Instruction-level, the pointer has to

- `mov 0x200be6(%rip),%rax` - 1st., copy value of the pointer (which holds an address), into `%rax` register.
- `movzbl 0x7(%rax),%eax` - off that address in register '
- `mov %al,-0x2(%rbp)` - `mov` contents `-0x2(%rbp)`into register `%al`

If it's not actually a pointer, but an array, the value is copied into the `%rax` register is an actual `char` (or `float`, some type). *Not* an address that the registers may have been expecting!

5.1. **Structs in C.** From Shaw (2015) [5], Exercise 16,

`struct` in C is a collection of other data types (variables) that are stored in 1 block of memory. You can access each variable independently by name.

- The `struct` you make, i.e.g `struct Person` is now a *compound data type*, meaning you can refer to `struct Person` using the same kinds of expressions you would for other (data) types.
- This lets you pass the whole `struct` to other functions
- You can access individual members of `struct` by their names using `x->y` if dealing with a ptr.

If you didn't have `struct`, you'd have to figure out the size, packing, and location of memory of the contents. In C, you'd let it handle the memory structure and structuring of these compound data types, `structs`. (Shaw (2015) [5])

Part 1. C, Stack, Heap Memory Management in C

6. C, Stack and Heap Memory Management, Heap and Stack Memory Allocation

cf. Ex. 17 of Shaw (2015) [5]

Consider chunk of RAM called stack, another chunk of RAM called heap. Difference between heap and stack depends on where you get the storage.

Heap is all the remaining memory on computer. Access it with `malloc` to get more.

Each time you call `malloc`, the OS uses internal functions (EY : 20171110 address bus or overall system bus?) to register that piece of memory to you, then returns ptr to it.

When done, use `free` to return it to OS so OS can use it for other programs. Failing to do so will cause program to *leak* memory. (EY: 20171110, meaning this memory is unavailable to the OS?)

Stack, on a special region of memory, stores temporary variables, which each function creates as locals to that function. How stack works is that each argument to function is *pushed* onto stack and then used inside the function. Stack is really a stack data structure, LIFO (last in, first out). This also happens with all local variables in `main`, such as `char action`, `int id`. The advantage of using stack is when function exits, *C compiler* pops these variables off of stack to clean up.

Shaw's mantra: If you didn't get it from `malloc`, or a function that got it from `malloc`, then it's on the stack.

3 primary problems with stacks and heaps:

- If you get a memory block from `malloc`, and have that ptr on the stack, then when function exits, ptr will get popped off and lost.
- If you put too much data on the stack (like large structs and arrays), then you can cause a *stack overflow* and program will abort. Use the heap with `malloc`.
- If you take a ptr, to something on stack, and then pass or return it from your function, then the function receiving it will *segmentation fault*, because actual data will get popped off and disappear. You'll be pointing at dead space.

cf. Ex. 17 of Shaw (2015) [5]

7. Data segments; Towards Segmentation Fault

cf. Ferres (2010) [9]

When program is loaded into memory, it's organized into 3 *segments* (3 areas of memory): let executable program generated by a compiler (e.g. `gcc`) be organized in memory over a range of addresses (EY : 20171111 assigned to physical RAM memory by address bus?), ordered, from low address to high address.

- *text* segment or code segment - where compiled code of program resides (from lowest address); code segment contains code executable or, i.e. code binary.

  As a memory region, text segment may be placed below heap, or stack, in order to prevent heaps and stack overflows from overwriting it.

  Usually, text segment is sharable so only a single copy needs to be in memory for frequently executed programs, such as text editors, C compiler,

shells, etc. Also, text segment is often read-only, to prevent program from accidentally modifying its instructions. cf. Memory Layout of C Programs; GeeksforGeeks

- Data segment: data segment subdivided into 2 parts:
  - initialized data segment - all global, static, constant data stored in data segment. Ferres (2010) [9]. Data segment is a portion of virtual address of a program.
    Note that, data segment not read-only, since values of variables can be altered at run time.
    This segment can also be further classified into initialized read-only area and initialized read-write area.
    e.g. `char s[] = "hello world"` and `int debut = 1` *outside the main (i.e. global)* stored in initialized read-write area.
    `const char *string = "hello world"` in global C statement makes string literal "hello world" stored in initialized read-only area. Character pointer variable string in initialized read-write area. cf. Memory Layout of C Programs
  - uninitialized data stored in BSS. Data in this segment is initialized by kernel (OS?) to arithmetic 0 before program starts executing.
    Uninitialized data starts at end of data segment ("largest" address for data segment) and contains all global and static variables initialized to 0 or don't have explicit initialization in source code.
    e.g. `static int i;` in BSS segment.
    e.g. `int j;` global variable in BSS segment.
    cf. Memory Layout of C Programs
- Heap - "grows upward" (in (larger) address value, begins at end of BSS segment), allocated with `calloc, malloc`, "dynamic memory allocation".
  Heap area shared by all shared libraries and dynamically loaded modules in a process.
  Heap grows when memory allocator invokes `brk()` or `sbrk()` system call, mapping more pages of physical memory into process's virtual address space.
- Stack - store local variables, used for passing arguments to functions along with return address of the instruction which is to be executed after function call is over.
  When a new stack frame needs to be added (resulting from a *newly called function*), stack "grows downward." (Ferres (2010) [9])
  Stack grows automatically when accessed, up to size set by kernel (OS?) (which can be adjusted with setrlimit(`RLIMIT_STACK`,...)).

7.0.1. *Mathematical description of Program Memory (data segments), with **Memory**, **Addresses**.* Let **Address**, with Obj(**Address**) $\cong \mathbb{Z}^+$ be an ordered set.
Memory block $\subset$ Obj(**Memory**), s.t.

$$\text{Memory block} \overset{\cong}{\mapsto} \{ \text{ low address , low address+}\texttt{sizeof}(T), \dots \text{ high address } \} \equiv \text{addresses}_{\text{Memory block}} \subset \mathbb{Z}^+$$

where $T \in$ Obj(**Types**) and $\cong$ assigned by address bus, or the virtual memory table, and $\text{addresses}_{\text{Memory block}} \subset$ Obj(**Addresses**).

Now,
text segment, (initialized) data segment, (uninitialized) data segment, heap, stack, command-line arguments and environmental variables $\subset \text{addresses}_{\text{Memory block}}$, that these so-called data segments are discrete subsets of the set of all addresses assigned for the memory block assigned for the program.

Now, $\forall\, i \in$ text segment , $\forall\, j \in$ (initialized) data segment , $i < j$ and $\forall\, j \in$ (initialized) data segment , $\forall\, k \in$ (uninitiaized) data segment , $j < k$, and so on. Let's describe this with the following notation:

$$\begin{aligned} &\text{text segment} < \text{(initialized) data segment} < \text{(uninitialized) data segment} < \\ &< \text{heap} < \text{stack} < \text{ command-line arguments and environmental variables} \end{aligned} \tag{1}$$

Consider stack of variable length $n_{\text{stack}} \in \mathbb{Z}^+$. Index the stack by $i_{\text{stack}} = 0, 1, \dots n_{\text{stack}} - 1$. "Top of the stack" is towards "decreasing" or "low" (memory) address, so that the relation between "top of stack" to beginning of the stack and high address to low address is *reversed*:

$$i_{\text{stack}} \mapsto \text{high addreess} - i_{\text{stack}}$$

Call stack is composed of stack frames (i.e. "activation records"), with each stack frame corresponding to a subroutine call that's not yet termined with a routine (at any time).

The *frame pointer* `FP` points to location where stack pointer was.

Stack pointer usually is a register that contains the "top of the stack", i.e. stack's "low address" currently, Understanding the stack, i.e.

$$\text{eval}(RSP) = \text{high address} - i_{\text{stack}} \tag{2}$$

7.0.2. *Mathematical description of strategy for stack buffer overflow exploitation.* Let $n_{\text{stack}} = n_{\text{stack}}(t)$. Index the stack with i_{stack} (from "bottom" of the stack to the "top" of the stack):

$$0 < i_{\text{stack}} < n_{\text{stack}} - 1$$

Recall that $i_{\text{stack}} \in \mathbb{Z}^+$ and

$$i_{\text{stack}} \mapsto \text{high address } -i_{\text{stack}} \equiv x = x(i_{\text{stack}}) \in \text{Addresses}_{\text{Memory block}} \subset \text{Obj}(\textbf{Address})$$

Let an array of length L (e.g. `char` array) `buf`, with `\&buf` = `\&buf(0)` $\in \text{Obj}(\textbf{Address})$, be s.t. `&buf` $= x(n_{\text{stack}} - 1)$ (starts at "top of the stack and "lowest" address of stack at time t, s.t.

$$\texttt{\&buf}(j) = \texttt{\&buf}(0) + j\texttt{sizeof}(T)$$

with $T \in \textbf{Types}$).

Suppose return address of a function (such as `main`), eval(`RIP`) be

$$\text{eval}(\texttt{RIP}) = \texttt{\&buf} + L \text{or at least } \text{eval}(\texttt{RIP}) \geq \texttt{\&buf} + L$$

If we write to `buf` more values than L, we can write over eval(`RIP`), making eval(`RIP`) a different value than before.

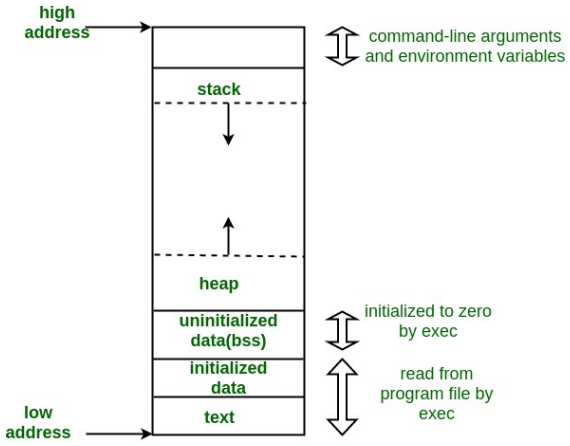

FIGURE 1. cf. https://www.geeksforgeeks.org/memory-layout-of-c-program/

7.1. **Stack.** cf. https://www.geeksforgeeks.org/memory-layout-of-c-program/

Traditionally, stack area adjoined heap area and they grow in opposite direction. When stack ptr meets heap ptr, free memory exhausted.
(with modern large address spaces, virtual memory techniques, they may be placed almost anywhere, but they still typically grow opposite direction)

stack area contains program stack, UFO structure, typically located in higher parts of memory.
On standard x86, grows toward address 0.
"stack ptr" register tracks top of stack, adjusted to point to top each time a value is "pushed" onto stack.
set of values pushed for 1 function call is "stack frame", stack frame consists at minimum of return address.

Newly called function allocates room on stack for its automatic and temporary variables.
$\implies$ recursion: each time recursive function calls itself, new stack frame is used, so 1 set of variables (in recursion) doesn't interfere with variables from another instance of the function.

cf. Where are static variables in C/C++? Tutorialspoint

static variables remain in memory; lifetime is entire program, stored in data segment of memory (near heap); data segment is part of virtual address space of a program.

asked here https://www.codingame.com/work/cpp-interview-questions/, what is a static variable

cf. Ferres (2010) [9]

Stack and functions: When a function executes, it may add some of its state data to top of the stack (EY : 20171111, stack grows downward, so "top" is smallest

address?); when function exits, stack is responsible for removing that data from stack.

In most modern computer systems, each thread has a reserved region of memory, stack. Thread's stack is used to store location of function calls in order to allow return statements to return to the correct location.

- OS allocates stack for each system-level thread when thread is created.
- Stack is attached to thread, so when thread exits, that stack is reclaimed, vs. heap typically allocated by application at runtime, and is reclaimed when application exits.
- When thread is created, stack size is set.
- Each byte in stack tends to be reused frequently, meaning it tends to be mapped to the processor's cache, making if very fast.
- Stored in computer RAM, *just like* the heap.
- Implemented with an actual stack data structure.
- stores local data, return addresses, used for parameter passing
- Stack overflow, when too much stack is used (mostly from infinite (or too much) recursion, and very large allocation)
- Data created on stack can be used without pointers.

Also note, for **physical location in memory**, because of Virtual Memory, makes your program think that you have access to certain addresses where physical data is somewhere else (even on hard disc!). Addresses you get for stack are in increasing order as your call tree gets deeper.

memory management - What and where are the stack and heap? Stack Overflow, Tom Leys' answer

7.2. **Stack overflow.** If you use heap memory, and you overstep the bounds of your allocated block, you have a decent chance of triggering a segmentation fault (not 100

On stack, since variables created on stack are always contiguous with each other; writing out of bounds can change the value of another variable. e.g. buffer overflow.

7.3. **Heap.** Heap contains a linked list of used and free blocks. New allocations on the heap (by `new` or `malloc`) are satisfied by creating suitable blocks from free blocks.

This requires updating list of blocks on the heap. This meta information about the blocks on the heap is stored *on the heap* often in a small area in front of every block.

- Heap size set on application startup, but can grow as space is needed (allocator requests more memory from OS)
- heap, stored in computer RAM, like stack.

7.3.1. *Memory leaks.* Memory leaks occurs when computer program consumes memory, but memory isn't released back to operating system.

"Typically, a memory leak occurs because dynamically allocated memory becomes unreachable." (Ferres (2010) [9]).

Programs `./Cmemory/heapstack/Memleak.c` deliberately leaks memory by losing the pointer to allocated memory.

Note, generally, the OS delays real memory allocation until something is written into it, so program ends when virtual addresses run out of bounds (per process limits).

7.4. **More Segmentation Faults.** The operating system (OS) is running the program (its instructions). Only from the hardware, with memory protection, with the OS be signaled to a memory access violation, such as writing to read-only memory or writing outside of allotted-to-the-program memory, i.e. data segments. On `x86_64` computers, this general protection fault is initiated by protection mechanisms from the hardware (processor). From there, OS can signal the fault to the (running) process, and stop it (abnormal termination) and sometimes core dump.

For virtual memory, the memory addresses are mapped by program called *virtual addresses* into *physical addresses* and the OS manages virtual addresses space, hardcare in the CPU called memory management unit (*MMU*) translates virtual addresses to physical addresses, and kernel manages memory hierarchy (eliminating possible overlays). In this case, it's the *hardware* that detects an attempt to refer to a non-existent segment, or location outside the bounds of a segment, or to refer to location not allowed by permissions for that segment (e.g. write on read-only memory).

7.4.1. *Dereferencing a ptr to a NULL ptr (in C) at OS, hardware level.* The problem, whether it's for dereferencing a pointer that is a null pointer, or uninitialized pointer, appears to (see the `./Cmemory/` subfolder) be at this instruction at the register level:

```
x000000000040056c <+38>:        movzbl (%rax),%eax

// or

0x00000000004004be <+24>:       movss  %xmm0,(%rax)
```

involving the register RAX, a temporary register and to return a value, upon assignment. And in either case, register RAX has trying to access virtual (memory) address `0x0` (to find this out in `gdb`, do `i r` or `info register`).

Modern OS's run user-level code in a mode, such as *protected mode*, that uses "paging" (using secondary memory source than main memory) to convert virtual addresses into physical addresses.

For each process (thread?), the OS keeps a *page table* dictating how addresses are mapped. Page table is stored in memory (and protected, so user-level code can't modify it). For every memory access, given (memory) address, CPU translates address according to the page table.

When address translation fails, as in the case that *not all addresses are valid*, and so if a memory access generates an invalid address, the processor (hardware!) raises a *page fault exception*. "This triggers a transition from *user mode* (aka *current privilege level (CPL) 3* on x86/x86-64) into *kernel mode* (aka CPL 0) to a specific location in the kernel's code, as defined by the *interrupt descriptor table* (IDT)." cf. [What happens in OS when we dereference a NULL pointer in C?](https://stackoverflow.com/questions/12645647/what-happens-in-os-when-we-dereference-a-null-pointer-in-c)

Kernel regains control and send signal (EY : 20171115 to the OS, I believe).

In modern OS's, page tables are usually set up to make the address 0 an invalid virtual address.

cf. [What happens in OS when we dereference a NULL pointer in C?](https://stackoverflow.com/questions/126 happens-in-os-when-we-dereference-a-null-pointer-in-c)

Part 2. C++

8. Free Store

GotW #9, Memory Management - Part I
cf. 11.2 Introduction of Ch. 11 Select Operations [10].

9. Copy vs. Move

cf. 17.1 Introduction of Ch. 17 Construction, Cleanup, Copy, and Move of Stroustrup [10].

Difference between *move* and *copy*: after a copy, 2 objects must have same value; whereas after a move, the source of the move isn't required to have its original value. So moves can be used when source object won't be used again.

Refs.: Sec. 3.2.1.2, Sec. 5.2, notion of moving a resource, Sec. 13.2-Sec.13.3, object lifetime and errors explored further in Stroustrup [10]

5 situations in which an object is copied or moved:

- as source of an *assignment*
- as object initializer
- as function argument
- as function return value
- as an exception

9.1. Copy constructor.

cf. Copy constructors, cppreference.com

Copy constructor of class T is non-template constructor whose 1st parameter is `T&`, `const T&`, `volatile T&`, or `const volatile T&`.

```
class_name ( const class_name & )
class_name ( const class_name & ) = default;
class_name ( const class_name & ) = delete;
```

9.1.2. *Explanation.*

(1) Typical declaration of a copy constructor.
(2) Forcing copy constructor to be generated by the compiler.
(3) Avoiding implicit generation of copy constructor.

Copy constructor called whenever an object is **initialized** (by **direct-initialization** or **copy-initialization**) from another object of same type (unless **overload resolution** selects better match or call is **elided** (???)), which includes

- initialization `T a = b;` or `T a(b);`, where b is of type T;
- function argument passing: `f(a);`, where a is of type T and f is `void f(T t)`;
- function return: `return a;` inside function such as `T f()`, where a is of type T, which has no **move constructor**.

```
struct A
{
int n;
A(int n = 1) : n(n) { }
A(const A& a) : n(a.n) { } // user defined copy ctor
};

struct B : A
{
// implicit default ctor B::B()
// implicit copy ctor B::B(const B&)
};

int main()
{
A a1(7);
A a2(a1); // calls the copy ctor
B b;
B b2 = b;
A a3 = b; // conversion to A& and copy ctor
}
```

i.e. cf. Copy Constructor in C++

Definition 1. ***Copy constructor*** *is a member function which initializes an object using another object of the same class.*

9.1.4. *When is copy constructor called?*

(1) When object of class returned by value
(2) When object of class is passed (to a function) by value as an **argument**.
(3) When object is constructed based on another object of same class (or overloaded)
(4) When compiler generates temporary object

However, it's not guaranteed copy constructor will be called in all cases, because C++ standard allows compiler to optimize the copy away in certain cases.

9.1.5. *When is used defined copy constructor needed? shallow copy, deep copy.* If we don't define our own copy constructor, C++ compiler creates default copy constructor which does member-wise copy between objects.

We need to define our own copy constructor only if an object has pointers or any run-time allocation of resource like file handle, network connection, etc.

9.1.6. *Default constructor does only shallow copy.*

9.1.7. *Deep copy is possible only with user-defined copy constructor.* We thus make sure pointers (or references) of copied object point to new memory locations.

```
MyClass t1, t2;
MyClass t3 = t1;          //     > (1)
t2 = t1;                    //      > (2)
```

Copy constructor called when new object created from an existing object, as copy of existing object, in (1). Assignment operator called when already initialized object is assigned a new value from another existing object, as assignment operator is called in (2).

9.1.9. *Why argument to a copy constructor should be const?* cf. Why copy constructor argument should be const in C++?, geeksforgeeks.org

(1) Use `const` in C++ whenever possible so objects aren't accidentally modified.
(2) e.g.

```
#include <iostream>

class Test
{
/* Class data members */
public:
Test(Test &t)   { /* Copy data members from t */ }
Test()                  { /* Initialize data members */ }
};

Test fun()
{
Test t;
return t;
};

int main()
{
Test t1;
Test t2 = fun();  error: invalid initialization of non const reference of type Test& from an rvalue of
}
```

`fun()` returns by value, so compiler creates temporary object which is copied to t2 using copy constructor (because this temporary object is passed as argument to copy constructor since compiler generates temp. object). Compiler error is because **compiler-created temporary objects cannot be bound to non-const references**.

9.2. **Move Constructor.** For a class, to control what happens when we move, or move and assign object of this class type, use special member function *move constructor*, *move-assignment operator*, and define these operations. Move constructor and move-assignment operator take a (usually nonconst) rvalue reference, to its type. Typically, move constructor moves data from its parameter into the newly created object. After move, it must be safe to run the destructor on the given argument. cf. Ch. 13 of Lippman, Lajole, and Moo (2012) [12]

10. vtable; virtual table

I was given this answer to a question I posed to a 20 year C++ veteran and it was such an important answer (as I did not know a virtual table existed, at all before), that I will copy this, repeat this and explore this extensively:

"The keyword you're looking for is virtual table: " How are virtual functions and vtable implemented?, stackoverflow

Original question, from Brian R. Bondy:

10.1. **How are virtual functions and vtable implemented?** We all know what virtual functions are in C++, but how are they implemented at a deep level?

Can the vtable be modified or even directly accessed at runtime?

Does the vtable exist for all classes, or only those that have at least one virtual function?

Do abstract classes simply have a NULL for the function pointer of at least one entry?

Does having a single virtual function slow down the whole class? Or only the call to the function that is virtual? And does the speed get affected if the virtual function is actually overwritten or not, or does this have no effect so long as it is virtual.

Answer from *community wiki*:

10.1.1. *How are virtual functions implemented at a deep level?* From "Virtual Functions in C++"

Whenever a program has a virtual function declared, a v-table is constructed for the class. The v-table consists of addresses to the virtual functions for classes that contain one or more virtual functions. The object of the class containing the virtual function contains a virtual pointer that points to the base address of the virtual table in memory.

Whenever there is a virtual function call, the v-table is used to resolve to the function address.

An object of the class that contains one or more virtual functions contains a virtual pointer called the vptr at the very beginning of the object in the memory. Hence the size of the object in this case increases by the size of the pointer. This vptr contains the base address of the virtual table in memory.

Note that virtual tables are class specific, i.e., there is only one virtual table for a class irrespective of the number of virtual functions it contains. This virtual table in turn contains the base addresses of one or more virtual functions of the class. At the time when a virtual function is called on an object, the vptr of that object provides the base address of the virtual table for that class in memory. This table is used to resolve the function call as it contains the addresses of all the virtual functions of that class. This is how dynamic binding is resolved during a virtual function call.

cf. "Virtual Functions in C++"

10.1.2. *What is a Virtual Function?* A virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. It is one that is declared as virtual in the base class using the virtual keyword. The virtual nature is inherited in the subsequent derived classes and the virtual keyword need not be re-stated there. The whole function body can be replaced with a new set of implementation in the derived class.

10.1.3. *What is Binding?* Binding is associating an object or a class with its member. If we call a method `fn()` on an object `o` of a class `c`, we say that object `o` is binded with method `fn()`.

This happens at *compile time* and is known as *static* - or *compile-time* binding. Calls to virtual member functions are resolved during *run-time*. This mechanisms is known as *dynamic-binding.*

The most prominent reason why a virtual function will be used is to have a different functionality in the derived class. The difference between a non-virtual member function and a virtual member function is, the non-virtual member functions are resolved at compile time.

10.1.4. *How does a Virtual Function work?* When a program (code text?) has a virtual function declared, a **v-table** is *constructed* for the class.

The v-table consists of addresses to virtual functions for classes that contain 1 or more virtual functions.
The object of the class containing the virtual function *contains a virtual pointer* that points to the base address of the virtual table in memory. An object of the class that contains 1 or more virtual functions contains a virtual pointer called the **vptr** at the very beginning of the object in the memory. (Hence size of the object in this case increases by the size of the pointer; "memory/size overhead.")

This vptr is added as a hidden member of this object. As such, compiler must generate "hidden" code in the **constructors** of each class to initialize a new object's vptr to the address of its class's vtable.

Whenever there's a virtual function call, vtable is used to resolve to the function address. This vptr contains base address of the virtual table in memory.

Note that virtual tables are class specific, i.e. there's only 1 virtual table for a class, irrespective of number of virtual functions it contains, i.e.

vtable is same for all objects belonging to the same class, and typically is shared between them.

This virtual table in turn contains base addresses of 1 or more virtual functions of the class.

At the time when a virtual function is called on an object, the vptr of that object provides the base address of the virtual table for that class in memory. This table is used to resolve the function call as it contains the addresses of all the virtual functions of that class. This is how dynamic binding is resolved during a virtual function call, i.e.

class (inherited or base/parent) cannot, generally, be determined *statically* (i.e. **compile-time**), so compiler can't decide which function to call at that (compile) time. (Virtual function) call must be dispatched to the right function *dynamically* (i.e. **run-time**).

10.1.5. *Virtual Constructors and Destructors.* A constructor cannot be virtual because at the time when constructor is invoked, the vtable wouldn't be available in memory. Hence, we can't have a virtual constructor.

A virtual destructor is 1 that's declared as virtual in the base class, and is used to ensure that destructors are called in the proper order. Remember that destructors are called in reverse order of inheritance. If a base class pointer points to a derived class object, and we some time later use the delete operator to delete the object, then the derived class destructor is not called.

Finally, the article "Virtual Functions in C++" concludes, saying, "Virtual methods should be used judiciously as they are slow due to the overhead involved in searching the virtual table. They also increase the size of an object of a class by the size of a pointer. The size of a pointer depends on the size of an integer." I will

have to check this with other references, because, first of all, how then would class inheritance be otherwise implemented?

cf. How are virtual functions and vtable implemented?, stackoverflow

10.1.6. *Can the vtable be modified or even directly accessed at runtime? No.* "Universally, I believe the answer is "no". You could do some memory mangling to find the vtable but you still wouldn't know what the function signature looks like to call it. Anything that you would want to achieve with this ability (that the language supports) should be possible without access to the vtable directly or modifying it at runtime. Also note, the C++ language spec does not specify that vtables are required - however that is how most compilers implement virtual functions."

10.1.7. *Do abstract classes simply have a NULL for the function pointer of at least one entry? Some do place NULL pointer in vtable, some place pointer to dummy method; in general, undefined behavior.* The answer is it is unspecified by the language spec so it depends on the implementation. Calling the pure virtual function results in undefined behavior if it is not defined (which it usually isn't) (ISO/IEC 14882:2003 10.4-2). In practice it does allocate a slot in the vtable for the function but does not assign an address to it. This leaves the vtable incomplete which requires the derived classes to implement the function and complete the vtable. Some implementations do simply place a NULL pointer in the vtable entry; other implementations place a pointer to a dummy method that does something similar to an assertion.

Note that an abstract class can define an implementation for a pure virtual function, but that function can only be called with a qualified-id syntax (ie., fully specifying the class in the method name, similar to calling a base class method from a derived class). This is done to provide an easy to use default implementation, while still requiring that a derived class provide an override.

10.1.8. *Does having a single virtual function slow down the whole class or only the call to the function that is virtual? No, but space overhead is there.* "This is getting to the edge of my knowledge, so someone please help me out here if I'm wrong!

I believe that only the functions that are virtual in the class experience the time performance hit related to calling a virtual function vs. a non-virtual function. The space overhead for the class is there either way. Note that if there is a vtable, there is only 1 per class, not one per object."

10.1.9. *Does the speed get affected if the virtual function is actually overridden or not, or does this have no effect so long as it is virtual? No, but space overhead is there.* "I don't believe the execution time of a virtual function that is overridden decreases compared to calling the base virtual function. However, there is an additional space overhead for the class associated with defining another vtable for the derived class vs the base class."

10.2. **pImpl, shallow copy, deep copy.** cf. Item 22: "When using the Pimpl Idiom, define special member functions in the implementation file," pp. 147 of Meyers (2014) [13].

```
class Widget {                    // still in header "widget.h"
public:
Widget();
~Widget();                  // dtor is needed see below
...
```

```
private:
struct Impl;     // declare implementation struct
Impl *pImpl;     // and pointer to it
};
```

Because `Widget` no longer mentions types `std::string, std::vector`, and `Gadget, Widget` clients no longer need to `\#include` headers for these types. That speeds compilation.

incomplete type is a type that has been declared, but not defined, e.g. `Widget::Impl`. There are very few things you can do with an incomplete type, but declaring a pointer to it is 1 of them.

`std::unique_ptr`s is advertised as supporting incomplete types. But, when `Widget w;`, `w`, is destroyed (e.g. goes out of scope), destructor is called and if in class definition using `std::unique_ptr`, we didn't declare destructor, compiler generates destructor, and so compiler inserts code to call destructor for `Widget`'s data member `m_Impl` (or `pImpl`).

`m_Impl` (or `pImpl`) is a `std::unique_ptr<Widget::Impl>`, i.e., a `std::unique_ptr` using default deleter. The default deleter is a function that uses `delete` on raw pointer inside the `std::unique_ptr`. Prior to using `delete`, however, implementations typically have default deleter employ C++11's `static_assert` to ensure that raw pointer doesn't point to an incomplete type. When compiler generates code for the destruction of the `Widget w`, then, it generally encounters a `static_assert` that fails, and that's usually what leads to the error message.

To fix the problem, you need to make sure that at point where code to destroy `std::unique_ptr<Widget::Impl>` is generated, `Widget::Impl` is a complete type. The type becomes complete when its definition has been seen, and `Widget::Impl` is defined inside `widget.cpp`. For successful compilation, have compiler see body of `Widget`'s destructor (i.e. place where compiler will generate code to destroy the `std::unique_ptr` data member) only inside `widget.cpp` after `Widget::Impl` has been defined.

For compiler-generated move assignment operator, move assignment operator needs to destroy object pointed to by `m_Impl` (or `pImpl`) before reassigning it, but in the `Widget` header file, `m_Impl` (or `pImpl`) points to an incomplete type. Situation is different for move constructor. Problem there is that compilers typically generate code to destroy `pImpl` in the event that an exception arises inside the move constructor, and destroying `pImpl` requires `Impl` be complete.

Because problem is same as before, so is the fix - *move definition of move operations into the implementation file.*

For copying data members, support copy operations by writing these functions ourselves, because (1) compilers won't generate copy operations for classes with move-only types like `std::unique_ptr` and (2) even if they did, generated functions would copy only the `std::unique_ptr` (i.e. perform a *shallow copy*), and we want to copy what the pointer points to (i.e., perform a *deep copy*).

If we use `std::shared_ptr`, there'd be no need to declare destructor in `Widget`.

Difference stems from differing ways smart pointers support custom deleters. For `std::unique_ptr`, type of deleter is part of type of smart pointer, and this makes it possible for compilers to generate smaller runtime data structures and faster runtime code. A consequence of this greater efficiency is that pointed-to types must be complete when compiler-generated special functions (e.g. destructors or move

operations) are used. For `std::shared_ptr`, type of deleter is not part of the type of smart pointer. This necessitates larger runtime data structures and somewhat slower code, but pointed-to types need not be complete when compiler-generated special functions are employed.

Part 3. Integers, numeric encodings, number representation, binary representation, hexidecimal representation

11. Introduction:

`int` $\neq \mathbb{Z}$, `float` $\neq \mathbb{R}$

cf. Systems 1, Introduction to Computer Systems, Introduction lecture at University of Texas, CS429; Don Fussell.

`int` $\neq \mathbb{Z}$, `float` $\neq \mathbb{R}$

e.g. $x^2 \geq 0$?

`float`: Yes!

`int`: $40000 \times 40000 \rightarrow 1600000000$

$50000 \times 50000 \rightarrow$??

Is $(x + y) + z = x + (y + z)$?

`unsigned` and `signed int`: Yes!

`float`'s

$$(1e20 + -1e20) + 3.14 \rightarrow 3.14$$

$$1e20 + (-1e20 + 3.14) \rightarrow ??$$

Can't assume "usual" properties due to finiteness of representations, so that integer operations satisfy "ring" properties - commutativity, associativity, distributivity

floating point operations satisfy "ordering" properties - monotonicity, values of signs

12. Bits and Bytes

cf. Lecture 2, Bits and Bytes, CS429.

Consider a (number) system with radix or base b ($b > 1$),
string of digits $d_1, \ldots, d_n$,

$$c = \sum_{i=1}^{n} d_i b^{n-i}, \qquad 0 \leq d_i < b \qquad \forall\, i = 1 \ldots n \tag{3}$$

with notation $c \equiv c_b$ to help denote the base or "radix", e.g. 15213_{10}, 00000000_2.

Byte $\equiv B$, 1 Byte $= 8$ bits. Note that $2^8 = 256$

e.g. Binary 00000000_2 to 11111111_2
Decimal: 0_{10} to 255_{10}
Hexadecimal: 00_{16} to FF_{16}

12.1. **Machine Words.** Machine has "Word size".

Nominal size of integer-valued data, including addresses.

12.2. **Word-oriented Memory Organization.** Addresses specify Byte locations.
Addresses of successive words differ by 4 (32-bit) or 8 (64-bit).
32-bit words:
Addr = 0000, Addr = 0004
64-bit words:
Addr = 0000, Addr = 0008

12.3. **Byte-ordering: Big-Endian, Little-Endian.** $b = 8$

$$c = \sum_{i=0}^{n-1} d_i 8^{n-1-i} \tag{4}$$

Big-Endian: Least significant byte has highest address.
For address a,

$$a \mapsto d_0$$
$$a+1 \mapsto d_1$$
$$\vdots$$
$$a+n-1 \mapsto d_{n-1}$$

Little-Endian: Least significant byte has lowest address

$$a \mapsto d_{n-1}$$
$$a+1 \mapsto d_{n-2}$$
$$\vdots$$
$$a+n-1 \mapsto d_0$$

12.4. **Representing Strings.** C Strings
- represented by array of characters, each character encoded in ASCII format - string should by null-terminated meaning final character = 0

12.5. **Machine-Level Code Representation.** Encode program as sequence of Instructions.
Each simple operation - arithmetic operation, read or write memory, conditional branch. - instructions encoded as bytes, i.e. programs are byte sequences too!

12.6. **Boolean Algebra.** Boole's Algebraic representation of logic.
And:
$A\&B = 1$ when both $A = 1$ and $B = 1$
Or:
$A|B = 1$ when either $A = 1$ or $B = 1$
Not:
$\sim A = 1$ when $A = 0$,
$\sim A = 0$ when $A = 1$,

Exclusive-Or (Xor) $A\hat{}B = 1$ when either $A = 1$ or $B = 1$, but not both

$\hat{}$	0	1
0	0	1
1	1	0

12.6.1. *Application of Boolean Algebra.* Claude Shannon 1937 MIT Master's Thesis. Encode closed switch as 1, open switch as 0. Connection when $A\& \sim B|\sim A\&B = A \wedge B$

12.6.2. *Integer Algebra.* $\langle \mathbb{Z}, +, *, -, 0, 1 \rangle$ forms a ring

12.6.3. *Boolean Algebra.* $\langle \{0,1\}, |, \&, \sim, 0, 1 \rangle$ forms a "Boolean algebra"

Or (|) is "sum" operation.
And (&) is "product" operation.
$\sim$ is "complement" operation.
0 is additive identity.
1 is multiplicative identity.

cf. https://en.wikipedia.org/wiki/Bitwise_operation

Bitwise OR may be used to set to 1 selected bits of register. e.g. fourth bit of 0010 may be set by performing bitwise OR with pattern with only 4th bit set.

Bitwise AND is equivalent to multiplying corresponding bits. Thus, if both bits in compared position are 1, bit in resulting binary representation is $(1 \times 1 = 1)$; otherwise result is 0 ($1 \times 0 = 0$ and $0 \times 0 = 0$).

Operation may be used to determine whether particular bit is set (1) or clear (0). e.g. given bit pattern 0011 (decimal 3), to determine whether 2nd bit is set, use bitwise AND with bit pattern containing 1 only in 2nd. bit. Because result 0010 is non-zero, we know 2nd. bit in original pattern is set. This is often called *bit masking* (by analogy, use of masking tape covers, or *masks*, portions that are not of interest)

Bitwise AND may be used to clear selected bits (or flags) of a register in which each bit represents an individual Boolean state.

- This technique is an efficient way to store a number of Boolean values using as little memory as possible.
- Also, easy to check parity (even or odd?) of binary number by checking value of lowest valued bit.

Bitwise XOR may be used to invert selected bits in a register (also called toggle or flip). Any bit may be toggled by XORing with 1. e.g. given bit pattern 0010, 2nd and 4th bits may be toggled by bitwise XOR with bit pattern containing 1 in 2nd. and 4th. positions.

Consider Boolean Ring:

$$\langle \{0,1\}, \wedge, \&, I, 0, 1 \rangle \simeq \mathbb{Z}_2 \equiv \text{ integers } \mod 2$$

where I is identity operation $I(A) = A$,
and $A \wedge A = 0$ is the additive inverse (existence) property.

12.6.4. *De Morgan's Laws.*

$$A\&B = \sim (\sim A|\sim B)$$
$$A|B = \sim (\sim A\& \sim B)$$

12.6.5. *Exclusive-Or using Inclusive Or.*

$$A \wedge B = (\sim A\&B)|(A\& \sim B)$$
$$A \wedge B = (A|B)\& \sim (A\&B)$$

All properties of Boolean Algebra apply to bit vectors.

12.7. **Representing and Manipulating Sets; bit vectors as sets.** Width w bit vector represents subsets of $\{0, \dots, w-1\}$, s.t. for $a \in \{0,1\}^w$, $a_j \in \{0,1\}$, $\forall\, j = 0, 1, \dots w-1$.

If $a_j = 1, j \in A$; otherwise,
If $a_j = 0$, $j \notin A$

12.7.1. *Operations.*
$\&\rightarrow$ intersection
$|\rightarrow$ union
$\wedge \rightarrow$ symmetric difference
$\sim\rightarrow$ complement

Logic operations in C: &&, ||, !
Shift operations
cf. Systems 1, Integers lecture at University of Texas, CS429 is a **very good lecture**; both *mathematically rigorous* and full of useful, *clear* examples.

13. Unsigned integers

Consider $\mathbb{Z}^+_{2^w} \rightarrow \mathbb{Z}^+$, where $\mathbb{Z}^+_{2^w}$ represent "unsigned" integers.

$$\begin{aligned} \mathbb{Z}^+_{2^w} &\rightarrow \mathbb{Z}^+ \\ (x_0, x_1, \dots, x_{w-1}) &\mapsto \sum_{i=0}^{w-1} x_i \cdot 2^i \end{aligned} \tag{5}$$

To recap using CS429's notation, for an unsigned integer x,
$B2U \equiv$ base-2-unsigned

$$B2U(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

where $B2U : "\mathbb{Z}^+" \rightarrow \mathbb{Z}^+_{2^w}$
So w is the total number of "bits" to represent x.
For hexidecimal (-based) numbers, observe this relationship:

$$2^w = 16^v = 2^{4v} \text{ so } w = 4v \tag{6}$$

So for a hexidecimal number, v hexidecimal numbers represent $w = 4v$ bits.
Observe that

$$\mathbb{Z}^+_{2^w} = \{0, \dots 2^w - 1\} \tag{7}$$

Also, **modular addition** $(\mathbb{Z}^+_{2^w}, +)$ forms an **abelian group**.

14. Two's complement

cf. Systems 1 Integers
Denote the notation for so-called "two's complement" numbers as $\mathbb{Z}_{2^w}$, which is a representation for integers in such a manner:

$$\mathbb{Z}_{2^w} \to \mathbb{Z}$$

$$(x_{w-1}, x_{w-2}, \dots x_0) \mapsto -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \tag{8}$$

with

$$\mathbb{Z}_{2^w} = \{-2^{w-1} \dots 2^{w-1} - 1\} \tag{9}$$

To recap using CS429's notation, for integer $x \in \mathbb{Z}$,
$B2T \equiv$ base-2-Two's complement

$$B2T(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

where $B2T : "\mathbb{Z}" \to \mathbb{Z}_{2^w} = \{-2^{w-1} \dots 2^{w-1} - 1\}$

Observe that

$$\begin{aligned} \max\left(\mathbb{Z}_{2^w}^+\right) &= 2 \max \mathbb{Z}_{2^w} + 1 \\ |\min \mathbb{Z}_{2^w}| &= \max \mathbb{Z}_{2^w} + 1 \end{aligned} \tag{10}$$

To recap and to clean up and unify the notation, recall the following:
radix (i.e. base) b
string of digits $d_1, \dots d_n$,

$$c = \sum_{i=1}^{n} d_i b^{n-i}, \quad 0 \le d_i < b \quad \forall\, i = 1 \dots n$$

cf. wikipedia, "Radix" i.e.

$$\{0, 1, \dots b-1\}^n \to \mathbb{Z}$$

$$(d_1, \dots d_n) \mapsto c = \sum_{i=1}^{n} d_i b^{n-i}$$

or (alternative notation),

$$\{01, , \dots b-1\}^w \to \mathbb{Z}$$

$$(d_{w-1}, d_{w-2}, \dots, d_0) \mapsto c = \sum_{i=0}^{w-1} d_i b^i$$

In Fussell's notation, Fussell (2011) [14],

$$\{0, 1, \dots b-1\}^w \to \mathbb{Z} \text{ or } \mathbb{Z}_{2^w}^+ \to \mathbb{Z}^+$$

$$(x_{w-1}, x_{w-2}, \dots x_0) \mapsto c = \sum_{i=0}^{w-1} x_i 2^i,\ x_i < 2$$

$$B2U(X) = B2U(x_{w-1}, \dots x_0) = \sum_{i=0}^{w-1} x_i 2^i$$

I will also use this notation:

$$B2U(x_w, x_{w-1}, \dots x_1) = \sum_{i=1}^{w} x_i 2^{i-1}$$

For the *Two's Complement*,

$$B2T(X) = B2T(-x_{w-1}, x_{w-2}, \dots x_0) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$
$$\{0, 1, \dots b-1\}^w \to \mathbb{Z}$$

or with the notation I'll use,

$$B2T(-x_w, x_{w-1}, \dots x_1) = -x_w 2^{w-1} + \sum_{i=1}^{w-1} x_i 2^{i-1}$$

Note that for hexadecimal,

$$c = \sum_{i=1}^{w} x_i 2^{i-1} = \sum_{i=1}^{v} y_i (16)^{i-1} = \sum_{i=1}^{v} y_i 2^{4i-4}$$

v hexadecimal digits $\leftrightarrow 4v = w$ bits.

14.1. **Numeric Ranges.** For unsigned values,

$$\text{UMin} = 0$$
$$000\dots0$$
$$\text{UMax} = 2^w - 1$$

For why $\text{UMax} = 2^w - 1$, consider the geometric progression:

$$\sum_{k=1}^{n} ar^{k-1}$$
$$(1-r)\sum_{k=1}^{n} ar^{k-1} = a - ar^n \text{ or } \sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r}$$
$$\implies \sum_{i=1}^{w} 2^{i-1} = \frac{1-2^w}{1-2} = 2^w - 1$$

$$\text{TMin} = -2^{w-1}$$
$$100\dots0$$
$$\text{TMax} = 2^{w-1} - 1011\dots1$$

$-1 \leftrightarrow 111\dots1$

14.2. **Negating with complement.**

$$\begin{gathered} \sim: \mathbb{Z}_{2^w} \to \mathbb{Z} \text{ s.t.} \\ \sim x + 1 = -x \quad \forall\, x \in \mathbb{Z}_{2^w} \\ \sim: \mathbb{Z}^+_{2^w} \to \mathbb{Z} \text{ s.t.} \\ \sim x = 2^w - 1 - x \end{gathered} \tag{11}$$

14.3. **Power of 2 Multiply with shift (left shift bitwise operation).** For both $\mathbb{Z}^+_{2^w}$, $\mathbb{Z}_{2^w}$,

$$u << k = u \cdot 2^k \qquad \forall\, u \in \mathbb{Z}^+_{2^w} \text{ or } \mathbb{Z}_{2^w} \text{and } k \in \mathbb{Z} \tag{12}$$

14.4. **Unsigned power-of-2 Divide, with shift (right shift bitwise operator).**

$$u >> k = \lfloor u/2^k \rfloor \qquad \forall\, u \in \mathbb{Z}^+_{2^w}, \quad k \in \mathbb{Z} \tag{13}$$

15. Endianness

Consider *address* $\in \mathbb{Z}^+_{2^3}$. Consider a *value* $\in \mathbb{Z}^+_{2^3}$ or $\mathbb{Z}_{2^3}$.
$\forall\, (x_0, x_1, \dots x_{w-1}) \in \mathbb{Z}^+_{2^3}$ or $\mathbb{Z}_{2^3}$
Given address a,

15.1. **Big-endian.**

Definition 2 (Big-endian)**.** *Most-significant byte value is at the lowest address, i.e.*

$$\begin{aligned} a &\mapsto x_{w-1} \\ a+1 &\mapsto x_{w-2} \\ a+2 &\mapsto x_{w-3} \\ &\vdots \\ a+w-1 &\mapsto x_0 \end{aligned} \tag{14}$$

15.2. **Little-endian.**

Definition 3 (Little-endian)**.** *Least-significant byte value is at the lowest address, i.e.*

$$\begin{aligned} a &\mapsto x_0 \\ a+1 &\mapsto x_1 \\ &\vdots \\ a+w-1 &\mapsto x_{w-1} \end{aligned} \tag{15}$$

For Little-endian $\to \mathbb{Z}$,
$(\ addresses\ ) \in \mathbb{Z}^+_{2^3} \mapsto \mathbb{Z}$,
$a \mapsto (x_0, x_1, \dots x_{w-1})$
For $\mathbb{Z} \to$ *Big-endian,*
$n \mapsto (x_{w-1}, x_{w-2}, \dots x_0)$.
Instead, to tackle the confusing problem of byte ordering, think about the integer as itself first, and then consider mapping multibyte binary values to memory.

$$\mathbb{Z} \to \left(\mathbb{Z}_{2^w} \longleftarrow \mathbb{Z}_{(2^3)^w} = \boldsymbol{addresses}\right) \tag{16}$$

$$\mathbb{Z} \ni (x_{w-1}, x_{w-2}, \dots x_1, x_0) \xrightarrow{\textit{Little-Endian}} ((x_0, x_1, \dots x_{w-2}, x_{w-1}) \longleftarrow\!\shortmid\ (a, a+1, \dots a+w-2, a+w-1))$$

$$\mathbb{Z} \ni (x_{w-1}, x_{w-2}, \dots x_1, x_0) \xrightarrow{\textit{Big-Endian}} ((x_{w-1}, x_{w-2}, \dots x_1, x_0) \longleftarrow\!\shortmid\ (a, a+1, \dots a+w-2, a+w-1))$$

Part 4. Floating Point Numbers

16. Floating point, IEEE Floating

IEEE Standard 754
Recall our previous notation for the sum representation of a number c:

$$c = \sum_{i=1}^{n} d_i b^{n-i}; \qquad 0 \leq d_i < b, \quad \forall\, i = 1 \dots n \tag{17}$$

e.g.

$$c = 42_{10} = 4 \cdot 10^{2-1} + 2 \cdot 10^{2-2} = d_1 b^{n-1} + d_2 b^{n-2}$$

By writing the exponent to be $n - i$, then we can write the digit representing the largest value in the summation "starting from the left" as such:

$$c = (d_1, d_2, \dots d_{n-1}, d_n)$$

e.g. $(4, 2) = 42$.

However, the CS429h lectures uses the following notation for the `unsigned int`:

$$B2U(X) = \sum_{i=0}^{w-1} x_i 2^i \text{ or } (x_{w-1}, x_{w-2}, \dots x_1, x_0) \mapsto \sum_{i=0}^{w-1} x_i 2^i$$

e.g. $(x_3, x_2, x_1, x_0) = (1, 0, 1, 1) \mapsto x_0 2^0 + x_1 2^1 + x_2 2^2 + x_3 2^3$

16.1. **Fractional Binary Numbers.**

16.1.1. *Representation.* Bits to right of "binary point" represent fractional powers of 2.

Binary number is used to represent a rational number, which in turn is a floating-point *representation*.

$$\sum_{k=-j}^{i} b_k \cdot 2^k \equiv \sum_{k=-M}^{N} b_k 2^k \tag{18}$$

$$b \equiv (b_N, b_{N-1}, \dots b_1, b_0, b_{-1}, \dots b_{-M}) \mapsto \sum_{k=-M}^{n} b_k 2^k \equiv c \tag{19}$$

i.e.

$$\sum_{k=-M}^{N} b_k 2^k = b_N \cdot 2^N + b_{N-1} 2^{N-1} + \dots + b_1 \cdot 2 + b_0 \cdot 1 + b_{-1} \cdot \frac{1}{2} + \dots + b_{-M} \cdot 2^{-M}$$

e.g. $101.11_2 \mapsto 5\frac{3}{4}$

Divide by 2 by shifting right:

$$\frac{c}{2} = \sum_{k=-M}^{N} b_k 2^k / 2 = \sum_{k=-M-1}^{N-1} b_{k+1} 2^k \tag{20}$$

$$(b_N, b_{N-1}, \dots b_{-M}) \xrightarrow{\cdot \frac{1}{2}} (0, b_N, b_{N-1}, \dots b_{-M-1}, b_{-M})$$

Multiply by 2 by shifting left:

$$2 \cdot c = \sum_{k=-M}^{N} b_k \cdot 2^k \cdot 2 = \sum_{k=-M+1}^{N+1} b_{k-1} 2^k \tag{21}$$

$$(b_N, b_{N-1}, \dots b_{-M}) \xrightarrow{\cdot 2} (b_N, b_{N-1}, \dots b_{-M}, 0)$$

This implies that we first choose the "most significant bit", the digit b_k representing the largest value in the sum representation to start from the *left*.

Numbers of form $0.111111\ldots_2$ just below 1.0 (remember those are bits, 0 or 1).

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^M}+\ldots\rightarrow 1.0$$

Use $1.0-\epsilon$ notation.

16.1.2. *Limitation of representable numbers.* Can only exactly represent numbers of form $\frac{x}{2^k}$

The other numbers have repeating bit representations.

Value	Representation
$\frac{1}{3}$	$0.0101010101[01]\ldots_2$
$\frac{1}{5}$	$0.001100110011[0011]\ldots 2$
$\frac{1}{10}$	$0.0001100110011[0011]\ldots_2$

16.1.3. *Floating Point Representation. Numerical Form*:

$$-1^s\,M\,2^E \tag{22}$$

where

sign bit s, $+1$ or -1
significand (mantissa, or coefficient) M, normally a fractional value in range $[1.0, 2.0)$
Exponent E weights value by power of 2.

From wikipedia,

$$\frac{s}{b^{p-1}}\cdot b^e \tag{23}$$

where s significand (ignoring an implied decimal point), p precision (number of digits in significand), base b.

e.g.

$$1.528535047\times 10^5=\frac{1528535047}{10^9}\times 10^5$$

Another example:

$$\left(\sum_{n=0}^{p-1}c_n 2^{-n}\right)\times 2^e=$$
$$(1\times 2^{-0}+1.2^{-1}+0\cdot 2^{-2}+0\cdot 2^{-3}+1\cdot 2^{-4}+\cdots+1\cdot 2^{-23})\times 2^1$$
$$\approx 1.5707964\times 2\approx 3.1415928$$

For the encoding of this numerical form,

Most significant bit (MSB) is sign bit.

exp field encodes E
frac field encodes M

$$s \qquad \text{exp} \qquad\qquad \text{frac}$$

Sizes for the encoding:

Single precision: 8 exp bits, 23 frac bits; 31 bits + 1 s bit = 32 bit
and so
2^{8-1}

$$\sum_{i=0}^{M-1} c_i 2^{-i} \xrightarrow{\max} 2^{M-1} = 2^{23-1}$$

Double precision: 11 exp bits, 52 frac bits; 63 bits + 1 s bit = 64 bits
$2^{\exp-1} = 2^{11-1} = 1024$. This is the max value of e in decimal.
$2^{M-1} = 2^{52-1}$

16.1.4. *"Normalized" Numeric Values.* Condition:

$$\exp \neq 000\ldots0 \text{and } \exp \neq 111\ldots1$$

Exponent coded as biased value:

$$E = \text{Exp} - \text{Bias} \tag{24}$$

Exp: unsigned value determined by exp
Bias: Bias value

Note that E is the actual value desired, e.g. $2^{13} = 2^E$, whereas Exp is the actual binary representation.

- Single precision: 127 (Exp: $1\ldots254$, E: $-126\ldots127$)
- Double precision: 1023 (Exp: $1\ldots2046$, E: $-1022\ldots1023$)

In general

$$\text{Bias} = 2^{e-1} - 1 \tag{25}$$

where e is number of exponent bits.

wikipedia says this: the exponent is "biased" in the engineering sense of the word - value stored is offset from actual value by the **exponent bias**. Biasing is done because exponents have to be signed value in order to be able to represent both tiny and huge values. It's not represented in two's complement, usual representation for signed values; it'd make comparison harder.

Exponent is stored as unsigned value which is suitable for comparison, and when interpreted, it's converted into an exponent within a *signed* range by subtracting the bias.

wikipedia uses this notation:

$$2^{k-1} - 1 = \text{ bias for floating point number}$$

where k is number of bits in exponent.

Number of normalized floating-point numbers in system (B, P, L, U) where

- base B
- precision of system to P numbers
- L smallest exponent representable in system
- U largest exponent used in system

$$2(B-1)(B^{P-1})(U-L+1)+1 \tag{26}$$

Smallest positive normalized floating-point number.

Underflow level = UFL = B^L

TODO: Understand all of this in https://en.wikipedia.org/wiki/Floating-point_arithmetic

Significand coded with implied leading 1:
Get extra leading bit for "free".
$xxx\ldots x$: bits of frac
Minimum when $000\ldots0$ ($M = 1.0$) Maximum when $111\ldots1$ ($M = 2.0 - \epsilon$)

16.1.5. *Denormalized Values and other Special Values.* Consider "denormalized" values, the special case with the following condition:

$$\exp = 000\ldots0$$

The Value is this:
Exponent value $E = -\text{Bias} + 1$ (compare this to Eq. **??**).
Significand value $M = 0.xxx\ldots x_2$, i.e. xxx...x: bits of frac.

Consider these special cases:

- $\exp = 000\ldots0$, frac $= 000\ldots0$, which represents value 0
  Note that there are distinct values for $+0$ and -0
- $\exp = 000\ldots0$, frac $\neq 000\ldots0$
  These represent numbers that are very close to 0.0. They lose precision as they get smaller. There's "gradual underflow". TODO: understand what it means for "gradual underflow", cf. https://www.cs.utexas.edu/users/fussell/courses/cs429h/lectures/Lecture_4-429h.pdf, pp. 10

Consider special values with the condition that

$$\exp = 111\ldots1$$

They include the following cases:

- $\exp = 111\ldots1$, frac $= 000\ldots0$. This represents the value at infinity, ∞. Operation that overflows. There are positive infinity and negative infinity. e.g. $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- $\exp = 111\ldots1$, frac $\neq 000\ldots0$ This is Not-a-Number (NaN). It represents case when no numeric value can be determined. e.g. `sqrt(-1)`, $\infty - \infty$.

Part 5. Complexity, Data Structures, Algorithms

17. Complexity, Big-O

Also note, let N = number of elements in a collection.

$$\log_b N = n \leftrightarrow b^n = N$$

$N \log_b N = n \leftrightarrow b^n = N^N$ but rather, use Stirling's approximation:

$$N \ln N \cong \log_b N! = \sum_{i=1}^{N} \log_b i = n$$

$O(N \log_b N) = O(\log_b N!)$ via Stirling's approximation: $\ln N! = N \ln N - N + O(\ln N)$

17.1. **Multi-Part Algorithms: Add vs. Multiply, e.g. $O(A+B)$ vs. $O(A*B)$.** If your algorithm is "do this (A), then, when completed, do that (B)" then add runtimes: $O(A + B)$.

If your algorithm is "do this for each time you do that", then multiply runtimes: $O(A * B)$.

17.2. **Amoritized Time (e.g. dynamically resizing array "ArrayList" or maybe `std::vector`).** Dynamically resize array x: if there are N elements, s.t. $N =$ max. capacity at N, $|x|= N \mapsto |x|= 2N$ and there are N copies to be made ($T = N$).

Let $X =$ number of elements to be inserted.

Suppose when $x = 1, 2, 4, 8, 16, \dots 2^j, \dots X$, $x = 2^j$ copies are made.

total number of copies:

$$\sum_{j=0}^{\lfloor \log_2 X \rfloor} 2^j = \sum_{j=0}^{\lfloor \log_2 X \rfloor} 2^{\lfloor \log_2 X \rfloor - j} = \sum_{j=0}^{\lfloor \log_2 X \rfloor} \frac{X}{2^j} = X \left(\frac{1 - 2^{-(\lfloor \log_2 X \rfloor + 1)}}{1 - 1/2} \right) = 2X \left(1 - 2/X\right)$$
$$\cong 2X \text{ if } X \text{ large}$$

Note that $X = 2^y$

$\log_2 X = y$

X insertions take $O(2X)$. "Amoritized" time for each insertion is $O(1)$.

17.3. $\log N$ **runtimes ($O(\log N)$).** e.g. binary search. find element (example) c in N-element *sorted* array. $x = x_i$, $i = 0, 1, \dots N-1 \mapsto i' = 1, 2 \dots N$.

First, compare c to midpoint of array.

If $c = x_{\lceil \frac{N}{2} \rceil}$, done.

So let $n_1 = N$.

if $c < x_{\lceil \frac{N}{2} \rceil}$, consider $x_1, x_2, \dots x_{\lfloor \frac{N}{2} \rfloor}$, if $c > x_{\lceil \frac{N}{2} \rceil}$, consider $x_{\lceil \frac{N}{2} \rceil + 1}, \dots x_N$

Let $n_2 = \lfloor \frac{N}{2} \rfloor$.

By induction, $n_j = \lfloor \frac{N}{2^{j-1}} \rfloor$

We stop when we either find the value c or we're down to just 1 element.

$\implies$ total runtime is the a matter of how many steps (dividing N by 2 each time).

$J =?$ s.t. $n_J = 1 = \lfloor \frac{N}{2^{J-1}} \rfloor \implies 0 = \log_2 N + (-J + 1)$ or

$$J = \log_2 N$$

When you see a problem where number of elements $n_0 = N, \dots n_j$ gets halved each time, it'll likely be $O(\log_2 N)$ runtime.

Finding an element in a **balanced binary search tree**: $O(\log N)$; with each comparison, we go either left or right.

17.4. **Recursive run times $O(2^N)$.**

Let $n_1 = N$		2^1 $f(n_1 - 1) = f(N-1)$ calls
$n_2 = N - 1$		$4 = 2^2$ $f(n_2 - 1)$ calls
$\dots$	$\mapsto$	$\dots$
$n_j = N - j + 1$		2^j $f(n_j - 1)$ calls
until $j = N$		2^N

$$\sum_{j=1}^{N} 2^j = \sum_{j=0}^{N-1} 2^{N-j} = 2^N \sum_{j=0}^{N-1} 2^{-j} = 2^N \left(\frac{1 - 2^{-N}}{1 - 1/2} \right) = 2^{N+1}(1 - 2^{-N}) \cong 2^N \qquad (N \text{ large})$$
$$\implies 2^{N+1} - 1 \text{ nodes}$$

Try to remember this pattern. When you have a recursive function that makes multiple calls, runtime will often (not always) look like

$$\boxed{O(\text{branches}^{\text{depth}})} \tag{27}$$

branches = number of times each recursive call branches.

space complexity is $O(N)$, only $O(N)$ nodes exist at any time.

cf. VI Big O, pp. 46, McDowell, Example 1

```
for (int i=0; i < N; i++)
{ sum += array[i]; }
```

$O(N)$.

cf. VI Big O, pp. 46, McDowell, Example 2

```
for (int i = 0; i < N; i++)
{
for (int j = 0; j < N; j++)
{
std::cout << i << j;
}
}
```

$O(N * N) = O(N^2)$ Or see it as printing $O(N^2)$ total number of pairs.

cf. VI Big O, pp. 46, McDowell, Example 3

```
for (int i = 0; i < N; i++)
{
for (int j = i + 1; j < N; j++)
{
std::cout << i << j;
}
}
```

$$i = 1, 2 \dots N$$
$$j = i + 1, \dots N$$
$$\sum_{j=i+1}^{N} 1 = N - i$$

$$\sum_{i=1}^{N}(N-i) = N \cdot N - \sum_{i=1}^{N} i = N^2 - \frac{N(N+1)}{2} = N^2 - \frac{N^2}{2} - \frac{N}{2} = \frac{N^2 - N}{2} \cong N^2 \quad \text{(large)}$$

cf. VI Big O, pp. 46, McDowell, Example 4

```
for (int i = 0; i < NA; i++)
{
for (int j = i + 1; j < NB; j++)
{
if (a[i] < b[j])
{
std::cout << a[i] << b[j];
}
}
}
```

if statement within j's for loop is $O(1)$ time since it's just a sequence of constant time statements.

$\implies O(NA \cdot NB)$

cf. VI Big O, pp. 47, McDowell, Example 5

```
for (int i = 0; i < NA; i++)
{
for (int j = i + 1; j < NB; j++)
{
for (int k = 0; k < 1000000; k++)
{
//...
}
}
}
```

$\implies O(NA \cdot NB \cdot 1000000) = O(100000(NA)(NB)) \cong O(NA \cdot NB)$

100,000 units of work is still constant, so run time is $O(NA \cdot NB)$.

cf. VI Big O, pp. 48, McDowell, Example 6

```
void reverse(int array[], const int N)
{
for (int i = 0; i < N / 2; i++)
{
int other = N - i - 1;
int temp = array[i];
array[i] = array[other];
array[other] = temp;
}
}
```

$O(N)$ time. The fact that it only goes through half of the array (in terms of iterations) doesn't impact big O time.

cf. VI Big O, pp. 47, McDowell, Example 8

Let the longest string be of length L.

sort each string $\rightarrow L \log L$ (merge sort or best "worst" case for a sort)

N_a = number of strings in the array of strings.

$N_a L \log L$ = total number of sorts.

Sort the full array.

You should also take into account that you need to compare the strings.

Each string comparison takes $O(L)$ time (compare each string element at each position).

$O(N_a \log N_a)$ comparisons (sort the full array of strings)

$$\implies \boxed{O(N_a L \log L) + O(L N_a \log N_a) = O(N_a L(\log L N_a))}$$

Operation	Average	Amoritized Worst Case
Copy	$O(N)$	$O(N)$
Append(1)	$O(1)$	$O(1)$
pop last	$O(1)$	$O(1)$
insert	$O(N)$	$O(N)$
Get item	$O(1)$	$O(1)$
Set item	$O(1)$	$O(1)$
Delete item	$O(N)$	$O(N)$
Get slice	$O(k)$	$O(k)$

Insert and delete is $O(N)$. Need to move every element with index greater than k, and reindex each element.

Get array length is $O(1)$. cf. https://stackoverflow.com/questions/21614298/what-is-the-runtime-of-array-length

18. Data Structures

18.1. Linked Lists, $O(1)$ insertion.

18.1.1. *Linked Lists vs. arrays.* main difference: Linked lists and arrays store difference information in each element.

Both cases: each element stores a value.
Both cases: stores 1 more information.

Array: index as a number.
Linked List: reference to next element, e.g. store addresses of next element, e.g. $x0123$
e.g.

$x0122$	$x0122$
value : 8	value : 8
next : $x0123$	next : nullptr

Trick to remember: Given elements at addresses 1, 2, 3. If you delete the next reference for 1 (to 2), and replace it with a new object, you'll lose your reference to 3.
Assign your next reference to object at 2 before assigning next reference to 1.

Insertion takes constant time since you're just shifting around (constant, finite number of) pointers, instead of iterating every element of the list.

Doubly linked list: Also have pointers to previous element, e.g.

1	2	3
value:8	value:2	value:6
next:2	next:3	next:4
prev:0	prev:1	prev:2

cf. https://en.wikipedia.org/wiki/Linked_list

Disadvantages of Linked Lists:

- Use more memory than arrays because storage used by their pointers
- Nodes in linked list must be read in order from beginning: inherently sequential access

- nodes stored in continguously (possibly)
- difficult to reverse traverse, doubly linked list helps, but memory consumed in allocating space for back-ptr

Linked lists vs. dynamic arrays:
List data structures comparison

	Linked list	Array	Dynamic Array	Balanced Tree
indexing	$O(N)$	$O(1)$	$O(1)$	$O(\log N)$
insert/delete at beginning	$O(1)$		$O(N)$	$O(\log N)$
insert/delete at end	$O(1)$ when last element is known $O(N)$ when last element is unknown		$O(1)$ amoritized	$O(\log N)$
insert/delete in middle	search time $+O(1)$		$O(N)$	$O(\log N)$
wasted space (average)	$O(N)$	0	$O(N)$	$O(\log N)$

head of list is 1st. node.
tail of 1st. last node in 1st. (or rest of list)

18.2. **Stack, $O(1)$ pop, $O(1)$ push.** "Stack Details", Udacity, Data Structures and Algorithms

top element $\mapsto$ Linked List head
push top element, pop top element. Last in, first out.
Last element you put in (new head), is the First out (when you pop).

18.3. **Queue, $O(1)$ insert, $O(1)$ delete, $O(N)$ search, $O(N)$ space complexities.** https://en.wikipedia.org/wiki/Queue_(abstract_data_type) Lesson 2: List-Based Collections, 11. Queues

Oldest element (tail) comes out first.

First In, First Out.
first element you put in (tail) is the first out (when you pop).

18.3.1. *Queue as Linked List.* head. "oldest element in the queue", "first"
tail. "newest element in queue", "last",
(I guess it grows from the tail)
add element to tail, enqueue.
Dequeue, remove head.
peek - peek at head.

Save references to head and tail.

18.3.2. *Deque. double-ended queue.* One can enqueue and dequeue from either end.

18.3.3. *Priority Queue.* . Assign each element in queue with a priority.
Remove oldest and highest priority element first.

18.4. **STL Containers.** cf. Ch. 31, STL Containers, Bjarne Stroustrup (2013) [10]

pp. 886 Stroustrup (2013) [10] `std::vector<T, A>` contiguous allocated sequence of `T`s;

cf. pp. 888 Sec. 31.2.1 "Container Representation", [10]

`vector` element data structure is most likely an array:

`vector`: rep ← elements — free space

`list` : likely represented by sequence of links pointing to elements and number of elements; doubly-linked list of `T`; use when you need to insert and delete elements without moving existing elements.

`map` : likely implemented as (balanced) tree of nodes pointing to (key, value) pairs:

`unordered_map` likely implemented as hash table

`unordered_map` rep ← hash table ←, ←, ... (k, v), (k, v) , ...

`string` for short strings, characters are stored in the string handle itself, for longer strings elements are stored contiguously on free-store (like `vector` elements). Like `vector`, `string` can grow into "free space" allocated to avoid repeated reallocations.

`string`: rep ← characters – free space

Like a built-in array, `array` is simply sequence of elements, with no handle, `array`: elements

cf. pp. 894, Sec. 31.3, Operations Overview, Stroustrup (2013) [10]

Standard Container Operation Complexity

	`[]` Sec. 31.2.2	List Sec. 31.3.7	Front Sec. 31.4.2	Back Sec. 31.3.6	Iterators Sec. 33.1.
`vector`	const	$O(n)$+		const +	Ran
`list`		const	const	const	Bi
`forward_list`		const	const		For
`deque`	const	$O(N)$	const	const	Ran
`stack`				const	
`queue`			const	const	
`priority_queue`			$O(\log(n))$	$O(\log(n))$	
`map`	$O(\log(n))$	$O(\log(n))$+			Bi
`set`		$O(\log(n))$+			Bi
`unordered_map`	const+	const+			For
`unordered_set`		const+			For
`string`	const	$O(n)$+	$O(n)$+	const+	Ran
`array`	const				Ran

Ran - random-access iterator, "For" - "forward iterator", "Bi" - "bidirectional iterator"

18.5. **vector.**

18.5.1. *vector and growth.* Layout of `vector` : elem ← "front" of elements,
space ← "front" of extra space (meets end of elements)
last ← end of extra space
alloc

Use of both size (number of elements), and capacity (number of available slots for elements without reallocation) makes growth through `push_back()` reasonably efficient: there's not an allocation operation each time we add an element, only every time we exceed capacity (Sec. 13.6)
adding half the size is common, standard doesn't specify by how much capacity is increased

18.5.2. *vector and Nesting.* `vector` (and similarly contiguously allocated data structures) has 3 major advantages:

- elements of `vector` compactly stored; no per-element memory overhead (contiguous on memory address)
  amount of memory consumed by `vec` of type `vector<X>` roughly `sizeof(vector<X>)+vec.size()*sizeof(X)`; `sizeof(vector<X>)` is about 12 bytes, insignificant for large vectors
- fast traversal, consecutive access, to get to next element, code doesn't have to indirect through a pointer
- simple and efficient random access, makes sort and binary search efficient

vs. doubly-linked list, `list`, incurs 4-words-per-element memory overhead (2 links plus free-store allocation header)

be careful don't unintentional compromise efficiency of access, e.g. 2-dim. matrix don't do `vector<vector<double>>`,
do `vector<double>` and compute locations from the indices

18.5.3. *vector vs. array.* `vector` resource handle, i.e. allows it to be resized and enable efficient move semantics,
disadvantage to arrays that don't rely on storing elements separately from handle; keeps sequence of elements on stack or in another object.

18.6. **Trees.** value, left pointer, right pointer,

levels - how many connections it takes to reach the root +1 child can only have 1 parent height - number of edges between it and farthest leaf
depth - number of edges to root.

height, depth are inverse

18.7. **Graphs.** edges can store data too.
Directed graph edges have a sense of direction.
A = set of ordered pairs of vertices.
undirected graph.
acyclic (no cycles)
DAG Directed graph with no cycles.

18.7.1. *Connectivity.* connected graph has only 1 connected component. $\exists$ path $\forall$ pair of vertices.
minimum number of elements (edges) to remove, to disconnect a component.

weakly connected directed graph if replacing all its directed edges with undirected edges produces a connected (undirected) graph.

18.7.2. *Graph Representation.* vertex object:
list of edges.

Edge Object
vertices.

Edge List: $= E$

Adjacency list $l = l(i)$ s.t. $\forall i \in V$, $l(i) \in$ **Set** s.t. $l(i)$ = set of all adjacent vertices.

Adjacent Matrix.

Let V = set of all vertices.
$\forall\, v \in V$, label them: $v = v(i)$, $i \in \mathbb{N}$

$\forall\, v_i \in V$, so $\forall\, i = 0, 1, \ldots N-1$,
Consider $\forall\, w = w_j \in V$, so $\forall\, j = 0, 1, \ldots N-1$.

If v_i, w_j are adjacen, let $a(i,j) = 1$, otherwise $w(i,j) = 0$. (adjacent means $\exists$ edge s.t. $E = \{v_i, w_j\}$)
$\implies a$ is an adjacency matrix.

Adjacency list.

$\forall\, v = v_i \in V$, $i = 0, 1, \ldots N-1$
$E = \{(v_o, v_t) | v_o, v_t \in V\}$
if $\exists\, e \in E$ s.t. $v_i = e(0) = v_o$, then $e(1) = v_t$ is adjacent to it.

Adjacency list useful for counting number of edges that a node has or number of adjacencies.

18.7.3. *Graph Traversal.* Depth-first search (DFS)
Breadth-first search (BFS)

DFS.
implementation: stack.
keep 2 structures: 1. Seen list, 2. stack.

If seen before, pop stack and go back.
$O(|E|+|V|)$ visit every edge twice $O(2|E|+|V|) = O(|E|+|V|)$.
$O(|V|)$ time to look up a vertex.

More on procedure:

begin with any node v_1. Put $v(v_1)$ into seen. Put v_1 on stack.
if $\exists\, \{e\} \subset E$ s.t. for $e = (v_o, v_t)$, $v_o = v_1$, then, put $v(v_t)$ into seen. Put v_t on stack.

Then consider v_t.
If $v(v_t) \in$ seen, consider another edge.
If you run out of edges with new node, pop stack.

Eulerian path $O(|E|)$

Hamiltonian path

19. Search

19.1. **Binary Search** $O(\log(N))$**.** https://classroom.udacity.com/courses/ud513/lessons/7123524086/concepts/71154040750923

$O(\log(N)+1) = O(\log(N))$ Binary search efficiency.

Given array $a = a_i$, $i = 0, 1, \ldots N-1$, N elements in an array, s.t. $a(i) \leq a(j)$ if $i \leq j$, $i, j \in 0, 1 \ldots N-1$ (i.e. sorted array)
Given x to search for,
Consider m_j s.t.

For $j = 1$, given N, $m_1 := \begin{cases} \frac{N}{2} & \text{if } N \text{ odd} \\ \frac{N}{2} - 1 & \text{if } N \text{ even} \end{cases}$

m_1 = midpoint to compare against.

if $x = a(m_1)$ done.
if $x < a(m_1)$, consider a_i s.t. $i = 0, \dots m_1 - 1$
if $x > a(m_1)$, consider a_i s.t. $i = m_1 + 1 \dots N - 1$

For j, given a_i, $i = l, \dots r$ $l \leq r$, (l, r are included in the range),
Let $L_j := r - l + 1$

$$m_j := \begin{cases} \frac{L_j}{2} + l & \text{if } N \text{ odd} \\ \frac{L_j}{2} - 1 + l & \text{if } N \text{ even} \end{cases}$$

If $x = a(m_j)$ done,
if $x < a(m_j)$, consider a_i s.t. $i = 0, \dots m_j - 1$
If $x > a(m_j)$, consider a_i s.t. $i = m_j + 1, \dots N - 1$
Stop when $L_j = 0$.

19.2. **Breadth-first search (BFS).** https://en.wikipedia.org/wiki/Breadth-first_search

starts at tree root, or some arbitrary node, and explores all neighbor nodes at present depth prior to moving on to nodes at next depth level.

Worst-cast performance $O(|V|+|E|) = O(b^d)$, Worst-case space complexity $O(|V|) = O(b^d)$

https://www.quora.com/What-are-the-advantages-of-using-BFS-over-DFS-or-using-DFS-over-BFS

Pro:
1. Solution definitely found out by BFS, 2. never get trapped in blind alley, unwanted nodes, 3. if there are more than 1 solution, will find solution with shortest steps

Con:
1. Memory constraints: as it stores all nodes of present level to go for next level 2. if solution far away, consumes time

Application of BFS:
1. Find shortest Path, 2. Check graph with bipertiteness

19.3. **Depth-first search.** Pre-order - check off nodes as soon as you see it, before seeing children.
root check it off, pick first left child, ...

In-order.
check off node once left child is seen.
went left most to right, went through all nodes in order.

Post-order.
Check off leaf, don't check off parent, check right child.

Delete $O(N)$. insert.

When do we want to use these structures? pros and cons.

19.4. **Binary Search Tree (BST), $O(\log N)$ height of the tree, run-time complexity, worst case $O(N)$.** Given node x, functions l, r, v s.t. $l(x), r(x) \in$ **Nodes** $\cup \emptyset$, $v(x)$ is some ordered value, so $v(x) \in$ **OrderedSet**.

Binary Search tree has the condition that $\forall\, x$,

$$v(l(x)) \leq v(x) \leq v(r(x))$$

Given search value y,
if $v(x) = y$, done,
if $v(x) < y$, $x_1 = r(x)$
if $v(x) > y$, $x_1 = l(x)$

height of tree is $O(\log N)$, run-time complexity.
insert $O(\log(N))$
delete $O(\log(N))$

Unbalanced, distribution of nodes skewed, worse case $O(N)$ search, insert, delete
e.g. $5 \to 10 \to 15 \to 20$

19.5. **Heaps.** max-heap: parent must always have bigger value than its child.
min-heap: parent must always have smaller value than its children.
Unlike binarytree, heap can have any number of children.

Pro:
1. heap finds max, min (root) 2. heap data structure efficiently use graph algorithm, e.g. Dijkstra

Con:
1. takes more time to compare and execute.
heap = max. efficient implementation of priority queue.

Heap	find min	delete min	insert	decrease-value
Binary Heap	$O(1)$	$O(\log N)$	$O(\log N)$	$O(\log N)$

e.g. max binary heap.
"complete" all levels except last are completely full.
if not, continue adding values from left to right.

Search: $O(N)$ no guarantee child is $\lessgtr$. end up searching entire tree.
improved search: if $x >$ root, quit search (for max-heap)
In general, if v(node) = node value $< x$, no need to check children of node.

Search: worse cast: $O(N)$
Average case: $O(N/2) = O(N)$

19.5.1. *Heapify (Worst case $O(\log N)$).* stick new element in open spot of the tree.
heapify - reorder tree based on heap property: $\forall$ given node C, if parent node P of C, $v(P) \geq v(C)$ (max heap)

Swap P and C when $v(C) > v(P)$
Extract; remove root. replace root with right most element. Then swap when necessary.

Heapify: worst case $O(\log N)$.
As many operations as height of the tree.

19.5.2. *Heap implementation (array).* max-binary heap: since we know how may children (2), $\forall$ parent,
Use some math to find next mode. If level l, nodes per level $l = 2^{l-1}$
Sorted array into tree.

Let sorted array be $a = a_i$ s.t.

$$a_0 \mapsto l = 1$$
$$a_1, a_2 \mapsto l = 2$$
$$a_3, a_4, a_5, a_6 \mapsto l = 3$$

$\left(\sum_{m=1}^{l-1} 2^{m-1}\right) - 1 = k$ index to start from.
Tree vs. Array
if tree uses nodes, need pointers: Arrays save space.

19.6. **Self-balancing tree.** Balanced minimize number of levels to use

19.7. **Red-Black Tree; search** $O(\log N)$**, space** $O(N)$**, insert** $O(\log N)$**, delete** $O(\log N)$**.** root black,
level 1 black
level 2 red
level 3 black

null leaf nodes must be colored black (i.e. all leaves are black)
Rule 4 (optional) root must be black.
Rule 5 every node; every path descending down must contain same number of black nodes. i.e. $\forall$ path from given node, to any of its descendant NIL nodes (leaf) must contain same number of black nodes.
red-black tree = self balancing binary search tree ($\forall$ node x, $v(l(x)) \leq v(x) \leq v(r(x))$)

19.7.1. *Red-Black Trees, insertion.* Insert Red Nodes Only.
runtime for insertion worst case $O(\log N)$. Binary search tree worse case $O(N)$ because BST could be unbalanced.

20. Recursion

cf. https://en.wikipedia.org/wiki/Recursion, Udacity
Recursion

- must call itself at some pt.
- base case
- alter input parameter

Theorem 1 (Recursion (from set theory))**.** *Given* $X \in \boldsymbol{Set}$, $a \in X$, $f : X \to X$
$\exists!\ F : \mathbb{N} \to X$ *s.t.*
$F(0) = a$
$F(n+1) = f(F(n))$

e.g. $Fib(0) = 0$
$Fib(1) = 1$
$\forall\, n > 1,\ n \in \mathbb{Z},\ Fib(n) := Fib(n-1) + Fib(n-2) = f(Fib(n-1), Fib(n-2))$

21. Sorting

21.1. **Bubble Sort, $O(N^2)$ in time, $O(1)$ in space.** cf. Efficiency of Bubble Sort, Lesson 3: Searching and Sorting, Data Structures and Algorithms, Python

Time complexity of bubble sort:
$\forall$ iteration, there were $N-1$ comparisons. Worst case is $N-1$ iterations to order all N elements.
$\implies O(N^2)$ worse case
$O(N^2)$ average case
$O(N)$ best case (it was already sorted!)

Space complexity $O(1)$ no extra arrays needed. Sort was "in-place."

From wikipedia, "Bubble Sort" (Optimizing Bubble Sort), cf. https://en.wikipedia.org/wiki/Bubble_sort, Observe that for an array of elements a indexed by $i = 1, 2 \dots N$,
$j = 1 \mapsto a(N)$ largest (otherwise it would not have been swapped; contradiction)
$j = 2 \mapsto a(N-1)$ 2nd. largest. Iterated on $2, \dots N-1$ $N-2$ total (don't include the first). $\vdots$
$j \mapsto a(N-j+1)$ jth largest, iterated on $2, \dots N-j+1$, $N-j$ total.

21.2. **Merge Sort, $O(N \log N)$ in time, $O(N)$ in space.** cf. Efficiency of Merge Sort, Lesson 3: Searching and Sorting, Data Structures and Algorithms, Python

Use approximation (approximate to the *worse case*!) to count the number of comparisons (operations) in a "pass" (iteration)
multiply each iteration with number of comparisons per iteration.

Number of iterations $\cong \log N$ since $2^J - 1 =$ array size.

N comparisons for $\log(N)$ steps $\implies O(N \log N)$

Space complexity. Auxilliary space $= O(N)$ at each step we need total arraying size N to copy into.

21.3. **Quick Sort.**

21.3.1. *First element pivot Implementation.* cf. https://stackoverflow.com/questions/22504837/how-to-implement-quick-sort-algorithm-in-c

Consider array $a = a_i$, $i = 0, 1, \dots N-1$; $|a| = N$ (array of length N).

Given $l, r \in 0, 1, \dots N-1$,
Let $p := a(l)$ (pivot value)
$i = i_i$; $i_1 = l$
Let $j = j_j$ s.t. $j_1 = l+1, \dots, j_{r-l-1} = r-1$ (range over $r-1-(l+1)+1 = r-l-1$)
if $a(j_j) \leq p$
$i_{j+1} = i_j + 1$
$a(i_{j+1}) = a(i_j + 1) \leftrightarrow a(j_j)$.

$a(i_{r-l-1}) \leftrightarrow a(l)$
$\mapsto i_{r-l-1}$.

Let's work out a table for the first few steps:

iteration	i	j	swaps	condition	example
1	l	$l+1$			
	$l+1$	$l+1$	$a(l+1) \leftrightarrow a(l+1)$	$p \geq a(l+1)$	5 3 4
2	l	$l+2$			
	$l+1$	$l+2$			
	$l+1$	$l+2$	$a(l+1) \leftrightarrow a(l+2)$	$p \geq a(l+2)$	5 6 3
	$l+2$	$l+2$	$a(l+2) \leftrightarrow a(l+2)$	$p \geq a(l+2)$	5 3 4
3	$l+1$	$l+3$	$a(l+1) \leftrightarrow a(l+3)$	$p \geq a(l+3)$	5 8 6 3 $\mapsto$ 5 3 6 8

21.3.2. *Complexity of Quick Sort.* Worst case if *all* pivots are in place, e.g. $1, 2, 8, 13$
Cannot "split" (partition) by pivots since it's already sorted $\implies O(N^2)$.
Average case $O(N \log N)$ (N iterations (for each element) $\log N$ comparisons (via split))
$O(1)$ space complexity.

Best case; move pivot to middle and divide each partition by 2. Bad case; array already somewhat sorted, hard to move pivot to middle.

21.4. **Time Complexities of all Sorting Algorithms.** https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

"Best" complexity (best of worst case) sort is $O(N \log N)$.

21.5. **Hashing, $O(1)$ lookup, or $O(n)$ bucket lookup.**

21.5.1. *Hash maps.* key $\xrightarrow{\text{Hash function}}$ Hash: Key
Value:

$\langle$Key, Value$\rangle$ $\xrightarrow{\text{Hash function on Key}}$ Hash: $\langle K, V \rangle$
Value:

Use keys are inputs to has function.
Store Key, Value pair in map $\langle K, V \rangle$

21.5.2. *Hash Functions.* Value $\xrightarrow{\text{Hash function}}$ Hash Value (often index in an array)

Fast $O(1)$ look up
e.g. ticket number
$8675309 \xrightarrow{?}$ some index.
Take last two digits of a big number
$01234956 \mapsto 56 \mapsto 56/100 = 5$
"last few digits are the most random"
collisions

e.g. $0123456 \mapsto 56\%10$
$6543216 \mapsto 16\%10$

To deal with collisions 1) change value(s used in hash function) in hash function or change hash function completely, or 2) change structure of array $\mapsto$ buckets.

Normal Array vs. Bucket
Value collection

(1) $X\%1000000$, $O(1)$
(2) Bucket, iterate through bucket $O(n)$, n allocated size of each bucket
    (a) Hash function inside

load factor - how "full" has table is.
Load factor = number of entries / number of buckets
e.g. 10 values in 1000 buckets, load factor = 0.01, majority of buckets in table will be empty.
waste memory with empty buckets, so rehash i.e. come up with new hash function with less resulting buckets.

closer load factor to 1, better to rehash and add more buckets
any table with load value greater than 1, guaranteed to have collisions.
100 numbers N
a number is a multiple of 5: $x = 5n$
Number of values = 100 numbers = N
Number of buckets = 100 (0 to 99)
$\implies 100/100 = 1$ load factor.
$100/107 < 1$ (good)
125 is a multiple of 5.
$x = 5n$. $\frac{x}{125} = \frac{5n}{125} = \frac{n}{25} \mapsto$ loss of collisions.
$\frac{100}{87} > 1 \mapsto$ collisions.
$\frac{100}{1000} = 0.1$ but waste memory, ton of empty buckets.

21.5.3. *Hash Table.* Constant time lookup important to speed up.
String Keys.
ASCII values from letters.
30 or less words, Use ASCII.

e.g. "UD" $\mapsto U = 85$
$D = 68$
$31 = (32 : 326)$
Huge has values for 4-letter strings
31
$s(0)31^{n-1} + s(1)32^{n-2} + \cdots + s(n-1)$

22. References for Data Structures and Algorithms

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/ https://www.tu-ilmenau.de/en/institute-of-theoretical-computer-science/lehre/lehre-ss-2017/aud/
Lecture Notes for Data Structures and Algorithms. Revised each year by John Bullinaria. School of Computer ScienceUniversity of BirminghamBirmingham, UK
https://www.tu-ilmenau.de/fileadmin/public/iti/Lehre/AuD/WS19Ing/AuD1-Kap-2-statisch2.pdf
https://www.tu-ilmenau.de/en/institute-of-theoretical-computer-science/

lehre/lehre-ws-20192020/aud-1/

Part 6. Linux System Programming

23. File Descriptors

cf. https://medium.com/@copyconstruct/nonblocking-i-o-99948ad7c957

Let byte $b \in (\mathbb{Z}_2)^8 \equiv B \equiv o$ (when referring to exactly 8 bits).

The fundamental building block of I/O in Unix is a sequence of bytes $(b_n)_{n\in\mathbb{N}}$, $b_n \in B$.

Most programs work with an even simpler abstraction - a stream of bytes or an I/O stream.

File descriptors $\equiv fd$. fd references I/O streams. $fd \in \mathbb{N}$ (by definition and the Linux programmer's manual).

$$\mathbb{N} \to B^{\mathbb{N}}$$

$$fd \overset{\text{surj}}{\longmapsto} (b_n)_{n\in\mathbb{N}}$$

e.g. Pipes, files, FIFOs, POSIX IPC's (message queues, semaphores, shared memory), event queues are examples of I/O streams referenced by a fd, i.e. $fd \overset{\text{surj}}{\longmapsto} (b_n)_{n\in\mathbb{N}}$.

23.1. Creation and Release of Descriptors.

fd's are either

- explicitly created by system calls like `open, pipe, socket`, etc. or
- inherited from parent process

fd's released when

- process exists
- calling `close` system call
- implicitly after an `exec` when fd marked as **close on exec**; i.e. fd automatically (and atomically) closed when `exec`-family function succeeds

24. Unix execution model

cf. https://stackoverflow.com/questions/4204915/please-explain-the-exec-function-and-its-fa
37558902

24.1. Processes and Programs.

A process is something in which a program executes. $pid \in \mathbb{N} \overset{\text{surj}}{\longmapsto}$ process

These 2 operations that you can do is the entire UNIX execution model:

`fork` - creates new process containing duplicate of current program, including its state. There are differences between the processes so to be able to distinguish which is parent, which is child:

$$pid_P \overset{\text{surj}}{\longmapsto} pid_C > 0$$

$$\text{prog}_i \text{ s.t. } \text{prog}_i \in pid_P \mapsto \text{prog}_i \in pid_C$$

`exec` - replaces program in current process with brand new program:

$$\text{prog}_i \in pid_i \mapsto \text{prog}_{i'} \in pid_i$$

$$\text{s.t. } \text{prog}_i = \emptyset,\ \text{prog}_i \neq \text{prog}_{i'}$$

More details:

`fork()` makes a near duplicate of current process, identical in almost every way; 1 process calls `fork()` while 2 processes return from it:

$$pid_P \xmapsto{\texttt{fork()}} pid_P, pid_C > 0$$

Examples:

- When `fork`, `exec` used in sequence: In shells, `find`:

$$pid_{\text{shell}} \xmapsto{\texttt{fork()}} pid_{\text{shell}}, pid_C \qquad \text{prog}_j, pid_C \xmapsto{\texttt{exec()}} \text{prog}_{j'} \in pid_C$$

Shell forks, then child loads `find` program into memory, setting up all command line arguments, standard I/O, etc.

- `fork()` only. Daemons simply listening on TCP port and fork copy of themselves to process specific request, while parent goes back to listening.

Notice that nominally, parent waits for child to exit.

close-on-exec flag, if set, will have fd automatically be closed during a successful **execve** (or `exec`) (cf. http://man7.org/linux/man-pages/man2/fcntl.2.html)

24.2. **File entry.** Every fd points to a data structure called *file entry* in the kernel.

$$fd \mapsto \text{ file entry}$$

i.e. $\forall\, fd$, $\exists$ file entry and pointer to that file entry.

open file description - an entry in system-wide table of open files.
open file description records file offset and file status flags.

$n_{\text{offset}} \equiv$ file offset, per fd, from the beginning of file entry.

`fork()` results in both parent and child processes using same fd and reference to same offset n_{offset} in file entry.

If process was last to reference file entry, kernel then deallocates that file entry.

Each file entry contains

- the type
- array of function pointers, that translates generic operations on fds to file-type specific implementations.

`open` call implementation greatly varies for different file types, even if higher level API exposed is the same.

e.g. sockets - Same `read, write` used for byte-stream type connections, but different system calls handle addressed messages like network datagrams.

For a process pid, consider fd table of fds. $\forall$ fd, $fd \mapsto$ open file description $\in$ open file table (system-wide).

$\forall$ open file description $\in$ open file table,
open file description $= n_{\text{offset}}$, status flags, and
open file description $\xmapsto{\text{inode ptr}}$ inode table.

A blocking system call is 1 that suspends or puts calling process on wait until an event occurs [1]

[1] https://stackoverflow.com/questions/19309136/what-is-meant-by-blocking-system-call, https://www.quora.com/What-is-the-difference-between-a-blocking-system-call-and-an-non-blocking-system-call

24.3. **Readiness of fds.** cf. https://medium.com/@copyconstruct/nonblocking-i-o-99948ad7c957

A fd is considered *ready* if process can perform I/O operation on fd without blocking (i.e. without process having to wait or be suspended). For fd to be considered "ready", it doesn't matter if **the operation actually transfer any data** - all that matters is that the I/O operation can be performed without blocking (i.e. the calling process won't wait or be suspended).

A fd changes into a *ready* state when an I/O *event* happens, such as the arrival of new input or completion of a socket connection or when space is available on previously full socket send buffer after TCP transmits queued data to the socket peer.

There are 2 ways to find out about the readiness status of a descriptor - edge triggered and level-triggered.

24.4. **Level Triggered.** cf. https://medium.com/@copyconstruct/nonblocking-i-o-99948ad7c957

To determine if fd is ready, the process tries to perform a non-blocking I/O operation. Also seen as "pull" or "poll" model. (User "pull" on the fd).

At time t_0, process pid tries I/O operation on non-blocking fd. If I/O operation blocks, system call returns error `err`.
At time $t_1 > t_0$, pid tries I/O on fd (again). Say call blocks again, and returns `err`.
At time $t_2 > t_1$, pid tries I/O on fd (again). Assume call blocks again, and returns `err`.
At time $t_3 > t_2$, pid polls status of fd, and fd is ready. pid can then choose to actually perform entire I/O operation (e.g. read all data available on socket).
At time $t_4 > t_3$, assume pid polls status of fd, and fd isn't ready; call blocks (again), and I/O operation returns `err`.
At time $t_5 > t_4$, pid polls for status of fd, and fd is ready. pid can then choose to only perform a partial I/O operation (e.g., reading only half of all data available)
At time $t_6 > t_5$, pid polls status of fd, and fd is ready. This time, pid can choose to perform no I/O.

24.5. **Edge Triggered.** Process *pid* receives a notification only when fd is "ready" (usually when there's any new activity on fd since it was last monitored). This can be seen as the "push" model, in that a notification is pushed to the process about readiness of a fd.

process pid only notified that fd is ready for I/O, but not provided additional information like for instance how many bytes arrived on socket buffer.

Thus, pid only armed with incomplete data as it tries to perform any subsequent I/O operation e.g. at $t_2 > t_0$, pid gets notification about fd being ready.

byte stream $(b_n)_{n\in\mathbb{N}}$ available for I/O stored in buffer.

Assume $N_b = 1024$ bytes available for reading, when pid gets notification at $t = t_2$.

$$\implies (b_n)_{n<N_b} \subset (b_n)_{n\in\mathbb{N}}$$

Assume pid only reads $N_b' = 500$ bytes $< N_b$

That means for $t = t_3, t_4, \dots t_{i_2}$, there are still $N_b - N_b' = 524$ bytes available in $(b_n)_{n<N_b}$ that pid can read without blocking.

But since pid can only perform I/O once it gets next notification, these $N_b - N_b'$ bytes remain sitting in buffer for that duration.

Assume pid gets next notification at $t_{i_2} = t_6$, when additional N_b bytes have arrived in buffer.

Then total amount of data available on $(b_n)_{n\in\mathbb{N}} = 2N_b - N_b' = 1548 = 524+1024$.

Assume pid reads in $N_b = 1024$ bytes.

This means that at end of 2nd. I/O operation, $N_b - N_b' = 524$ bytes still remain in $(b_n)_{n\in\mathbb{N}}$ that pid won't read before next notification arrives.

While it might be tempting to perform all I/O immediately once notification arrives, doing so has consequences.

- large I/O operation on single fd has potential to starve other fd's
- Furthermore, even with case of level-triggered notifications, an extremely large Write or send call has potential to block.

Multiplexing I/O on fd's.

In above, we only described how pid handles I/O on a single fd.

Often, pid wants to handle I/O on more than 1 fd.

e.g. program prog needs to log to `stdout`, `stderr`, while accept socket connections, and make outgoing RPC connections to other services.

There are several ways of multiplexing I/O on fds.

- nonblocking I/O (fd itself is marked as non-blocking; operations may finish partially)
- signal-drive I/O (pid owning fd is notified when I/O state of fd changes)
- polling I/O (e.g. `select`, `poll`, both provide level-triggered notifications about readiness of fd)

24.6. **Multiplexing I/O with Non-blocking I/O.** We could put all fds in nonblocking mode.

pid can try to perform I/O operations on fds to check if any I/O operations result in error.

kernel performs I/O operation on fd and returns err, or partial output, or result of I/O operation if it succeeds.

Cons:

- Frequent checks
- Infrequent checks. If such operations are conducted infrequently, then it might take pid unacceptably long time to respond to I/O event that's available.

When does this approach make sense?

Operations on output fds (e.g. writes) don't generally block.

- In such cases, it might help to try to perform I/O operation first, and revert back to polling when operation returns err.

It might also make sense to use non-blocking approach with edge-triggered notifications, where fds can be put in nonblocking mode, and once pid gets notified of I/O event, it can repeatedly try I/O operations until system calls would block with an `EAGAIN` or `EWOULDBLOCK`.

24.7. **Multiplexing I/O via signal driven I/O.** .

Kernel instructed to send the pid a signal when I/O can be performed on any of the fds.

pid will wait for signals to be delivered when an of the fds is ready for I/O operation.

Cons: Signals are expensive to catch, rendering signal drive I/O impractical for cases where large amount of I/O is performed.

Typically used for "exceptional conditions."

24.8. **Multiplexing I/O via polling I/O.** .
fds placed in non-blocking mode.
pid uses **level triggered mechanism** to ask kernel by means of system call (`select` or `poll`) which fds are capable of performing I/O.

24.8.1. *`poll`, and `select` vs. `poll`.* With `select`, we pass in 3 sets of fds we want to monitor for reads, writes, and exceptional cases.
With `poll`, we pass in a set of fds, **each marked with the events it specifically needs to track**.

24.8.2. *What happens in the kernel (with `select` and `poll`)?* Both `select` and `poll` are *stateless*, meaning,
every time a `select` or `poll` system call is made, the kernel checks *every fd*, in the input array passed as 1st. argument, for the occurrence of an event and return the result to the pid.
$\implies$ this means that the cost of `select`/`poll` is O(N), where N is the number of fds monitored.
The implementation of `select` and `poll` comprises 2 tiers:

- Specific top tier which decodes incoming request, as well as several device or socket specific bottom layers
- bottom layers comprise of *kernel poll functions* used both by `select`, `poll`

25. `epoll` Event poll

cf. "The method to epoll's madness" by Cindy Sridharan[15]
`epoll`, a Linux specific construct, allows for a pid to monitor multiple fds and get notifications when I/O is possible on them.

25.1. `epoll` **Syntax.** *epoll*, unlike *poll*, is not a system call. It's a kernel data structure that allows a pid to multiplex I/O on multiple fds.
This data structure can be created, modified, deleted by 3 system calls: `epoll_create`, `epoll_ctl`, `epoll_wait`.

26. References and Resources for Linux System Programming

http://igm.univ-mlv.fr/~yahya/progsys/linux.pdf
http://www.cs.fsu.edu/~baker/opsys/examples/forkexec/fork_wait.c
http://cs241.cs.illinois.edu/coursebook/ https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf https://www.cs.rutgers.edu/~pxk/417/notes/content/02r-sockets-programming-slides.pdf https://www.cs.cmu.edu/~srini/15-441/S10/lectures/r01-sockets.pdf
https://db.in.tum.de/teaching/ss19/c++praktikum/

Part 7. C++ Template Metaprogramming

From this question on stack overflow, comes this tutorial.
cf. Vandevoorde, Josuttis, and Gregor (2017) [16].

Part 8. Functional Programming and Category Theory

Best discussion of the relation to monads in category theory to monads in "computer science": https://ncatlab.org/nlab/show/monad+%28in+computer+science%29

Best article on practical implementation (real daily programming, real programming experiences) of functional programming without pulling punches on Category Theory:

https://nalaginrut.com/archives/2019/10/31/8%20essential%20patterns%20you%20should%20know%20about%20functional%20programming%20in%20c%2B%2B14

More links:

https://nbviewer.jupyter.org/github/ivanmurashko/articles/blob/master/cattheory/cattheory.pdf https://wiki.ifs.hsr.ch/SemProgAnTr/files/mof_categories_final.pdf

27. Actors, Concurrent Systems

https://sourceforge.net/projects/sobjectizer/ https://github.com/Stiffstream/sobjectizer#helloworld-example

Čukić (2018) [17]

Ch. 12, pp. 250, Čukić (2018) [17]

Consider that multiple (e.g. person) objects shouldn't have any shared data - real people *share* data by *talking* to each other; they don't have shared variables everyone can access and change.

actors - in actor model, actors are completely isolated entities that share nothing but can send messages to 1 another.

- actor receives messages and processes them 1 by 1, message → actor
- reaction to message can be change the actor itself or send message to another actor, actor → another actor in system

design actors as follows:

- Actors can receive only messages of a signle type, and send messages of a single (not necessarily same) type. If you need to support multiple different types for input or output messages, use `std::variant` or `std::any`, as in Ch. 9, Čukić (2018) [**?**]
- Leave choice of whom to send message to external controller so can compose actors in a functional way. External controller will schedule which sources an actor should listen to
- leave it up to external controller to decide which messages should be processed asynchronously and which shouldn't

https://cs.lmu.edu/~ray/notes/messagepassing/ Message passing notes

Part 9. Technical Interviews

"Interview Introduction", Data Structures and Algorithms in Python, Udacity

- Clarifying the Question
- Generating Inputs and Outputs
- Generating Test Cases
- Brainstorming

- Runtime Analysis
- Coding
- Debugging

Part 10. Embedded Systems

Lee and Seshia (2016) [18].

cf. Ch. 3 Discrete Dynamics, Lee and Seshia (2016) [18].

showed how state machines can be used to model discrete dynamics.

cf. Sec. 3.1., Lee and Seshia (2016) [18].

pp. 43, Ex. 3.1., Lee and Seshia (2016) [18]. e.g. Consider system that counts number of cars that enter and leave parking garage in order to keep track of how many cars are in garage at any time.

Ignore for now how to design sensors that detect entry or departure of cars.

Assume `ArrivedDetector` actor produces an event when car arrives, and `DepartureDetector` actor produces an event when car departs.

`Counter` actor keeps running count, starting from initial value i.

Each time count changes, it produces an output event that updates display.

Each entry or departure modeled as **discrete event**. Discrete event **occurs at** an instant of time rather than over time.

EY: So there are *3* actors:

(1) Arrival Detector $\rightarrow$ produces event when car arrives
(2) Departure Detector $\rightarrow$ produces event when car departs
(3) Counter actor (keeps running count, each time count changes (so it's a FSM)) -¿ produces output event that updates a display

Signal u into `up` input port (from `ArrivalDetector`, upon arrival) is function of form

$$u : \mathbb{R} \rightarrow \{\text{absent}, \text{present}\}, \text{ i.e.}$$

$\forall$ time $t \in \mathbb{R}$, $u(t)$ either absent, i.e. there's no event at that time, or present, meaning there is a

pure signal - signal or function of form $u : \mathbb{R} \rightarrow \{$ absent, present$\}$, carries no value, but instead provides all its information by either being present or absent at any given time.

Counter: when **event** is present at up input port, increment count and produce output of new count value.
When event present at down input, decrements count and produces output new count value.
At other times, produces no output (count output is absent)

Hence,

$$c : \mathbb{R} \rightarrow \{ \text{ absent } \} \bigcup \mathbb{Z}$$

Counter signal is not pure.

Input to counter is pair of discrete signals that at certain times have an event, at other times have no event. Output is also discrete signals.

signals $\xrightarrow{\text{counter}}$ output signal

EY: separate FSM logic from this signal or message passing function.

Definition 4 (Signal and Discrete Signal)**.** *Signal is a function of the form* $e : \mathbb{R} \to \{absent\} \bigcup X$, $X \in Obj\mathbf{Set}$ *s.t.*
$\exists$ *1-to-1* $f : T \to \mathbb{N}$, *s.t.* f *order preserving, i.e.* $\forall t_1, t_2 \in T$, *if* $t_1 \leq t_2$, *then* $f(t_1) \leq f(t_2)$, *where* $T = T_{present} = \{t \in \mathbb{R} | e(t) \neq absent\}$.
e *is* ***discrete*** *if* $\exists$ *1-to-1* $f : T \to \mathbb{N}$ *such that* f *order preserving.*

cf. pp. 52 Lee and Seshia (2016) [18]

For example, in Figure 3.5, for the thermostat with hysteresis (so that it avoids **chattering**, heater would turn on and off rapidly when temperature is close to setpoint temperature), the finite state machine (FSM) could be **event triggered**, like the garage counter, in which case it'll react whenever a *temperature* input is provided.

Alternatively, it could be **time triggered** - it reacts at regular time intervals.

The definition of FSM doesn't change in these two cases: it's up to the environment in which an FSM operates when it should react.

cf. pp. 52, Sec. 3.3.2 "When a Reaction Occurs", Lee and Seshia (2016) [18]

Recall the notion of "reaction": from pp. 48, Sec. 3.2 "The Notion of State", formally define state to be encoding of everything about past that has an effect on system's reaction to current or future inputs. From Sec. 3.3 "Finite-State Machines",

Definition 5 (State Machine, Sec. 3.3 Finite State Machines, Lee and Seshia (2016)[18], pp. 48)**.** ***state machine*** *is a model of a system with discrete dynamics that at each* reaction *maps valuations of inputs to valuations of outputs, where map may depend on its current state.* ***finite-state machine*** *(FSM) is a state machine where the set* States *of possible states is finite.*

However, *nothing in the definition of a state machine constrains* ***when*** *it reacts.* The environment determines when machine reacts. Chapters 5, 6 of Lee and Seshia (2016) [18] describe a variety of mechanisms and give precise meaning to terms like **event triggered** and **time triggered**.

cf. Sec. 3.4 Extended State Machines, Lee and Seshia (2016) [18]

Pushes problem of large number of states into a variable, and there's only 1 state.

e.g. garage counter of Fig. 3.4, in Fig. 3.8

pp. 62, "Extended state machines can provide a convenient way to keep track of the passage of time." but merely pushes tracking time to input variable. e.g. Ex. 3.9, traffic light, Fig. 3.10.

Lee and Seshia mixes extended state machine state with discrete state and variables; but variables act as input.

Sec. 3.5 Nondeterminism, Fig. 3.11, of Lee and Seshia (2016)[18] nondeterministic model of pedestrians (as FSM)

state machine nondeterministic if more than 1 initial state (sufficient condition) or
if $\forall$ state, $\exists!$ 2 distinct transitions with guards that can evaluate to true in same

reaction

Sec. 3.6 Behaviors and Traces, Lee and Seshia (2016)[18],
Consider port p of state machine with type V_p
Consider sequence of reactions $\in (V_p \bigcup \{ \text{ absent })^\infty$, represented

$$s_p : \mathbb{N} \to V_p \bigcup \{ \text{ absent } \}$$

This is signal received on that port (if it's an input), or produced on that port (if it's an output).
Behavior may be represented as sequence of valuations called **observable trace**.
Let x_i represent valuation of input ports,
y_i represent valuation of output ports at reaction i
Then observable trace is

$$((x_0, y_0), (x_1, y_1), \dots)$$

Observable trace is really just another representation of behavior.
Ch. 3. Discrete Dynamics, Exercise 8:

Exercise 8.

(a) $x : \mathbb{R} \to \{a, p\}$, $a \equiv$ absent, $p \equiv$ present

$$x(t) = \begin{cases} p & \text{if } t \in \mathbb{Z}^+ \\ a & \text{otherwise, } \forall\, t \in \mathbb{R} \end{cases}$$

Recall $T := \{t \in \mathbb{R} | x(t) \neq a\}$. Let $t_1, t_2 \in T$ s.t. $t_1 \leq t_2$. Then since $t_1, t_2 \in T$, $t_1, t_2 \in \mathbb{Z}^+$ (by def. of given x). Let $f : T \to \mathbb{N}$ by identity $t \mapsto t$. Then $f(t_1) = t_1 \leq t_2 = f(t_2)$

(b) Consider pure signal $y : \mathbb{R} \to \{p, a\}$ given by

$$y(t) = \begin{cases} p & \text{if } t = 1 - 1/n \,\forall\, n \in \mathbb{Z}^+ \\ a & \text{otherwise} \end{cases} \qquad \forall\, t \in \mathbb{R}$$

Let $T := \{t | y(t) \neq a\}$, let $t_1, t_2 \in T$ s.t. $t_1 \leq t_2$. Consider $t_1 = 1 - \frac{1}{n_1}$, $t_2 = 1 - \frac{1}{n_2}$.
Now

$$1 - \frac{1}{n_1} \leq 1 - \frac{1}{n_2} \text{ or } \frac{-1}{n_1} \leq -\frac{1}{n_2} \text{ or } \frac{1}{n_1} \geq \frac{1}{n_2} \text{ or } n_2 \geq n_1$$

Let $f(t) = \frac{1}{1-t}$ if $t_1 \leq t_2$. $f(t_1) = \frac{1}{1-\left(1-\frac{1}{n_1}\right)} = n_1 \leq n_2 = \frac{1}{1-t_2} = f(t_2)$

(1) Signal w is a merge of x and y, i.e. if $w(t) = p$ if either x or $y = p$, $w = a$ otherwise

References

[1] Randal E. Bryant, David R. O'Hallaron. **Computer Systems: A Programmer's Perspective** (3rd Edition). ISBN-13 : 978-0134092669 ISBN-10 : 013409266X Pearson; 3rd Edition (March 12, 2015)

[2] Bertrand Meyer. **Object-Oriented Software Construction** (Book/CD-ROM) (2nd Edition) 2nd Edition. Prentice Hall; 2 edition (April 13, 1997). ISBN-13: 978-0136291558

[3] Randall Hyde. **Write Great Code: Volume 1: Understanding the Machine** October 25, 2004. No Starch Press; 1st edition (October 25, 2004). ISBN-13: 978-1593270032

[4] Randall Hyde. **Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level**. 1st Edition. No Starch Press; 1 edition (March 18, 2006). ISBN-13: 978-1593270650

[5] Zed A. Shaw. **Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)** (Zed Shaw's Hard Way Series) 1st Edition. Addison-Wesley Professional; 1 edition (September 14, 2015) ISBN-13: 978-0321884923.

[6] Eli Bendersky. Are pointers and arrays equivalent in C?

[7] Brian W. Kernighan, Dennis M. Ritchie. **C Programming Language**, 2nd Ed. 1988.

[8] Peter van der Linden. **Expert C Programming: Deep C Secrets** 1st Edition. Prentice Hall; 1st edition (June 24, 1994) ISBN-13: 978-0131774292

[9] Leo Ferres. "Memory management in C: The heap and the stack". Memory management in C: The heap and the stack

[10] Bjarne Stroustrup. **The C++ Programming Language**, 4th Edition. Addison-Wesley Professional; 4 edition (May 19, 2013). ISBN-13: 978-0321563842

[11] Carlo Ghezzi, Mehdi Jazayeri. **Programming Language Concepts** 3rd Edition. Wiley; 3 edition (June 23, 1997). ISBN-13: 978-0471104261 https://vowi.fsinf.at/images/7/72/TU_Wien-Programmiersprachen_VL_(Puntigam)_-_E-Book_SS08.pdf

[12] Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. **C++ Primer** (5th Edition). Addison-Wesley Professional; 5 edition (August 16, 2012) ISBN-13: 978-0321714114

[13] Scott Meyers. **Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14**. 1st Edition. O'Reilly Media; 1 edition (December 5, 2014) ISBN-13: 978-1491903995

[14] Prof. Donald S. Fussell (Instructor). CS429H (378H) - Systems I (Honors) (Computer Organization, Architecture, and Programming). Spring 2011. University of Texas. https://www.cs.utexas.edu/users/fussell/courses/cs429h/

[15] Cindy Sridharan. "The method to epoll's madness." *Medium*. Oct. 29, 2017. https://medium.com/@copyconstruct/the-method-to-epolls-madness-d9d2d6378642

[16] David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor. **C++ Templates: The Complete Guide** (2nd Edition). Addison-Wesley Professional; 2 edition (September 18, 2017). ISBN-13: 978-0321714121

[17] Ivan Čukić. **Functional Programming in C++: How to improve your C++ programs using functional techniques**. 1st Edition. 2018

[18] Edward Ashford Lee, Sanjit Arunkumar Seshia. **Introduction to Embedded Systems: A Cyber-Physical Systems Approach** (The MIT Press) Second Edition. The MIT Press; Second edition (December 30, 2016). ISBN-10: 0262533812. ISBN-13: 978-0262533812