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

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

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 the usual "letter" format that is exactly the same as the content here, `HrdwCCppCUDA.pdf`.

Meyer (1997) [1]

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.

> "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) [3], where HLL stands for high-level language. I was part of that generation and knew of nothing of assembly.

2. 80x86 Assembly; 80x86 Architecture

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

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

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

and

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

Likewise,

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

And so

$$\text{CPU} : \textbf{Memory} \times \textbf{address bus} \to \text{Hom}(\textbf{Memory}, \textbf{address bus})$$
$$\text{CPU} : (\text{memory location}, \text{unique binary numeric address}) \mapsto \&$$

in the language of category theory.

2.2. Registers (for 80x86).

cf. 3.3 Basic 80x86 Architecture, from pp. 23 of Hyde (2006) [3].

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) [3].

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) [3]

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) [4], 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:

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

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.

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

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.

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

"But here's the trick": e.g. "`ages` is an address inside the *entire computer*." (Shaw (2015) [4]).
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) [4]).
Let

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

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

How C is doing the following with your arrays:

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

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

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

  $$\textbf{ptrs}$$
  $$\text{Obj}(\textbf{ptrs}) \ni a, \text{ e.g. } \texttt{ages}$$

  $$a \mapsto \text{Memory block}(0)$$
  $$\text{Obj}(\textbf{ptrs}) \to \text{Obj}(\textbf{Memory}_{CPU})$$
- *indexing* into the block, by taking the base address of `ages`, or a

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

"A pointer is simply an address pointing somewhere inside computer's memory with a type specifier." Shaw (2015) [4]
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) [4]

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, [5]). 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, [5]) (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, [5].
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) [6], 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 [5] 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) [7]. Ch. 4, 9, 10.
Bendersky [5] 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 [5] 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 [5]

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) [4], 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, `struct`s. (Shaw (2015) [4])

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) [4]

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) [4]

7. Data segments; Towards Segmentation Fault

cf. Ferres (2010) [8]

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) [8]. 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) [8])

  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*:

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

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, cf. 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)` $\in$ Obj(**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 $T \in$ **Types**).

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

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

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

7.1. **Stack.** cf. Ferres (2010) [8]

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

cf. `https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap/80113#80113`

Stack is memory set aside for a thread of execution. Each thread gets a stack, while there's typically only 1 heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

"

> To what extent are they controlled by the OS or language runtime?

The OS allocates the stack for each system-level thread when the thread is created. Typically the OS is called by the language runtime to allocate the heap for the application.

> What is their scope?
>
> The stack is attached to a thread, so when the thread exits the stack is reclaimed. The heap is typically allocated at application startup by the runtime, and is reclaimed when the application (technically process) exits.
>
> What determines the size of each of them?

The size of the stack is set when a thread is created. The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system).

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) [8]).

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/12645647/what-happens-in-os-when-we-dereference-a-null-pointer-in-c)

Part 2. C++

8. static

9. Free Store

cf. GotW #9, Memory Management - Part I

Free store is 1 of the 2 dynamic memory areas, allocated/freed by new/delete (free store and heap). Object lifetime can be less than the time the storage is allocated.

Let $t_{0,\text{obj}}, t_{f,\text{obj}}, t_{f,\text{obj}} \geq t_{0,\text{obj}}$,
$t_{0,\text{alloc}}, t_{f,\text{alloc}}, t_{f,\text{alloc}} \geq t_{0,\text{alloc}}$,
with the condition that

$$t_{0,\text{alloc}} \leq t_{0,\text{obj}} \leq t_{f,\text{obj}} \leq t_{f,\text{alloc}}$$

.

For $t \in (t_{0,\text{alloc}}, t_{0,\text{obj}})$, storage may be accessed and manipulated through a `void*` but non of the proto-object's *nonstatic* members or member functions may be accessed, have their addresses taken, or be otherwise manipulated.

cf. 11.2 Introduction of Ch. 11 Select Operations [9].

10. Copy vs. Move

cf. 17.1 Introduction of Ch. 17 Construction, Cleanup, Copy, and Move of Stroustrup [9].

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 [9]

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

10.1. **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;
```

10.1.2. *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.*

10.1.4. *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.

10.1.5. *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.

10.1.6. *Default constructor does only shallow copy.*

10.1.7. *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).

10.1.9. *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 a
}
```

`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**.

10.2. **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) [11]

11. 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:

11.1. **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*:

11.1.1. *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++"

11.1.2. *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.

11.1.3. *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.

11.1.4. *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**).

11.1.5. *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

11.1.6. *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."

11.1.7. *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.

11.2. **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) [12].

```
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.

References

[1] Bertrand Meyer. **Object-Oriented Software Construction** (Book/CD-ROM) (2nd Edition) 2nd Edition. Prentice Hall; 2 edition (April 13, 1997). ISBN-13: 978-0136291558
[2] 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
[3] 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
[4] 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.
[5] Eli Bendersky. Are pointers and arrays equivalent in C?
[6] Brian W. Kernighan, Dennis M. Ritchie. **C Programming Language**, 2nd Ed. 1988.
[7] Peter van der Linden. **Expert C Programming: Deep C Secrets** 1st Edition. Prentice Hall; 1st edition (June 24, 1994) ISBN-13: 978-0131774292
[8] Leo Ferres. "Memory management in C: The heap and the stack". Memory management in C: The heap and the stack
[9] Bjarne Stroustrup. **The C++ Programming Language**, 4th Edition. Addison-Wesley Professional; 4 edition (May 19, 2013). ISBN-13: 978-0321563842
[10] 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
[11] Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. **C++ Primer** (5th Edition). Addison-Wesley Professional; 5 edition (August 16, 2012) ISBN-13: 978-0321714114
[12] 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