Basic Memory Introduction

An introduction to low-level programming and how to avoid

```
$./a.out
105
Core Dumped.
```

going over why it happens, How to diagnose compiled (C/C++) programs & how to fix it.

I wrote assuming the reader has experience in some garbage collected language like Python or Javascript.

[1] WHY

In many languages deemed high level or easy to use the main thing allowing this easy development without directly caring about the underlying memory & addresses is a garbage collector. On a very high level they work like this:

```
int func( ) {
    String x = "APPLE";   ⟶ "APPLE" allocated
    func2(x);             ⟶ Copy of x made & tracked.
    x = "ORANGE";         ⟶ "APPLE" removed
                              & "ORANGE" allocated
}
```

This is a very rough example to give out an idea of its purpose, there are many different ways a garbage collector can function See the working of the Java or the C# GC if you're interested in the engineering behind them.

<Talk about various GC strategies>

All in all, GCs are fantastic bits of engg. that make the vast majority of coding relatively hassel free by removing the work of the coder to manage memory.

But if you're reading this, you already decided you don't use one or for the following use cases might be suboptimal,

- Byte Wrangling: You're moving around a lot of

large buffers around & you can't afford any overhead in copying & in general want to interact with the bytes correctly.

- Latency sensitive code: When you're working on is real-time or cannot handle a GC pause at any time.
  - ↓
  - microseconds your program's execution could be paused to rec() objects.

- High Performance: This is a bad reason to chose after C, C++, Rust for, you can get fast code in all the languages but it's true that compiled languages tend to be closer to the hardware so can be optimized at a lower level.

*: Be careful of choosing a hard lang. simply b/c you think something like C# would be too slow, chances are good C# code can outperform the equivalent code in Rust (eg.).

It all depends on the algorithms & the data structures you employ.
↓
O(n)

How

* OV - Oversimplified for expl.
■ - Additional comment

Primer

Im going to over a few concepts here so that I can use the terms Im comfortable lorter.

Byte → A collection of 8 bits. [uint8_t]

Slice / Span → A struct consisting of a ptr & a length (bytes).
↳ struct → A collection of multiple diff data types together.

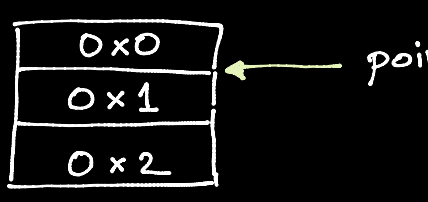
Pointer → An offset into program memory, can encode the type of object its pointing to.

void * → Anything

int * → Integer(s)

Could point to multiple objects in sequence but you cant infer that from its type.

Program memory can be thought of a long (only long) tape where each slot in that tape can hold a byte (Byte addressable)



A pointer simply points to one of these bytes.

OY

Heap / Stack:

You can think of this tape as being divided into 2 kinds, A fast tape c/ limited qty & one bulk cheap tape but its slower.

The fast tape is the **Stack**, the other is **Heap**

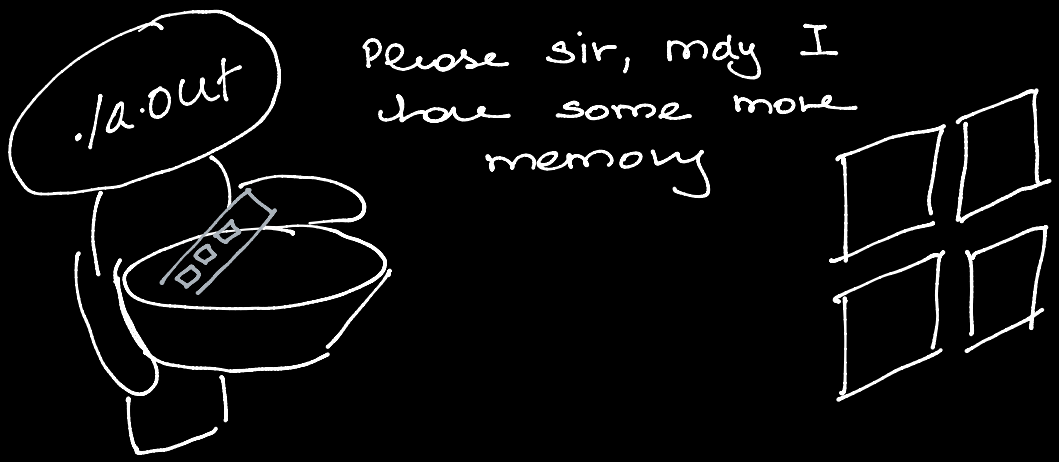
↓ Its already managed for you

↓ You manually manage this one.

Allocating / Freeing:

This mostly applies to the heap & not the stack since that one is already managed for you.

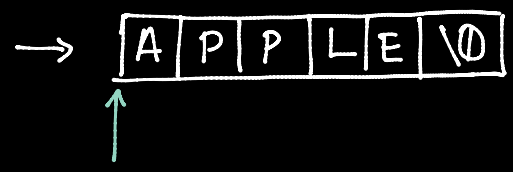
When you want to put say 500 numbers on the heap, you need a pointer to 500×4 bytes of free space, You can't simply point a pointer to any memory address that is not given to your program by the OS, So you have to humbly request the OS for these $2e^3$ bytes of free space, this process of asking for memory from the operating system is called **allocation**.



The opposite process, returning the memory to the operating system (ie giving up ownership) is called **freeing**.

Segmentation Fault (SIGSEV):
When your program tries to access memory it doesn't have access to, the OS kills the process & raises a segfault error.

String:
Since I'm mostly writing from the perspective of a C programmer, the string I will be referring to is a C-string.



The string is stored w/ no length & is terminated with a NULL byte (\0)

Reverse

I feel the easiest way to go at will be to re-tread common issues you could face & see whats going on & how to solve it.

Returning a stack string.

I specify stack since a heap pointer can be passed around till the pointer is freed.

Lets take the following:

```
1. char* getTimestampStr( ) {
2.     char timestamp[100];
3.     snprintf(timestamp, sizeof(timestamp), "%llu", time(NULL));
```

```
4.      return timeStamp;
5.  }
```

In line 2, we declare a variable 'timestamp' to be 100 chars [notice theres no allocation so this a stack variable]

In this we put the UNIX timestamp into that variable, → Using sprintf()

sizeof(X) → Gives the size of a data type as known at compile time, Not useful for pointers since itll return the no of bytes to store the pointer (8 bytes)

If you replace the data type of timestamp w/ this youll get 8 also → sizeof((char*)timestamp);

↓

Casting away the known array type.

Then after insertion, the function returns the variable as a pointer.

↓

C doesnt have proper 'ARRAY' types so any array is fundamentally a pointer to memory.

But this will fail since the pointer returned by the function will be invalid. This will be easiest to show with the assembly:

The following ASM is for AMD64 (SysV calling convention)

getTimestampStr:

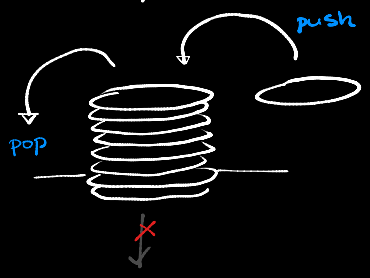
```
push   rbx      → Store callee address on stack
sub    rsp, 112 → Reserve 100 bytes on the stack      (1)
xor    edi, edi → Zero out EDI register
call   time@PLT → Call time(EDI = 0)
lea    rdx, [rip + .L.str] → Store &"%lld"
mov    rbx, rsp → Store address of stack space        (2)
mov    esi, 100 → Store sizeof(timestamp)
mov    rdi, rbx → Copy buffer address
mov    rcx, rax → Store output of time (NULL)
xor    eax, eax → Mark as multiple variadic arguments
call   snprintf@PLT → Call snprintf(rdi, rsi, rdx, rcx)
mov    rax  rbx → RAX = RBX = Buffer pointer.
add    rsp, 112 → "Unreserve" stack space             (3)
pop    rbx      → Restore callee address

ret
```

If you haven't messed w/ assembly registers, here's what you need to understand

Stack (Data Structure)

Think of a stack of plates. (plates)



You can push new elements to the end of the stack

You can only pop the last inserted element → I.e the element at the top of the stack.

Wikibook Link

The CPU Stack (from w/h we derive stack memory from) is basically some scratch space (think like a rough notebook) whenever it needs some memory & It can't use its registers (insufficient space / data to be stored for later); So stack memory really refers to data stored on this stack.

The CPU has a stack register (ESP) & instructions only for pushing & poping data from the stack.

Pushing data increments the SP

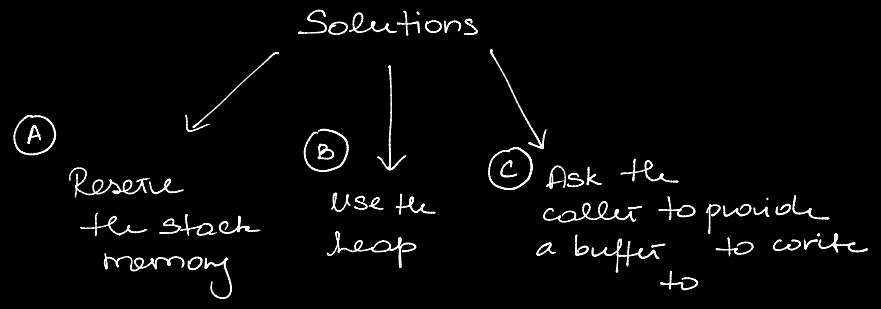
Poping data decrements the SP

Now we can look at the instructions,

- *1 → This reserves 112 bytes of space on the stack (112 → 100+ rounding to 16b boundry)
- *2 → Sets the address of the array to the current stack pointer address.
- *3 → Unreserves the stack space (cleanup).

I hope the problem is very visible now, the stack space isn't reserved at all so any future code that modifies the stack will clobber the char array so corrupting our string → Makes the returned pointer pointless.

There's 3 ways we can solve this, you should've thought of 1 already,



Once you're done / you'll start to notice almost all C apis using (B) or (C) in order to pass non primative data back to the caller.

(A) This is done by making the array static

```
.LC0:
    .string "%llu"
    .text
    .globl  getTimestampStr
    .type   getTimestampStr, @function
getTimestampStr:
    endbr64
    push    rbx #
# t.c:6:     snprintf(timestamp, sizeof(timestamp), "%llu", time(NULL));
    mov edi, 0  #,
    call    time@PLT    #
    mov r9, rax # tmp86,
# /usr/include/x86_64-linux-gnu/bits/stdio2.h:54:   return __builtin___snprintf_chk (__s, __n, __USE_FORTIFY_LEV
    lea r8, .LC0[rip]   #,
    mov ecx, 100    #,
    mov edx, 2  #,
    mov esi, 100    #,
    lea rbx, timestamp.0[rip]   # tmp84 → Function static storage
    mov rdi, rbx    #, tmp84
    mov eax, 0  #,
    call    __snprintf_chk@PLT  #
# t.c:8: }
    mov rax, rbx    #, tmp84
    pop rbx #
    ret
    .size   getTimestampStr, .-getTimestampStr
    .local  timestamp.0
    .comm   timestamp.0,100,32
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
    .long   4f - 1f
    .long   5
```

} The "%llu" local

} Reserve space for static variable.

Direct output from GCC

Also this is impractical for large objects like several megabytes in size. Its best for singletons or for function returns that are expected to be used then discarded before the next function call.

The main problem w/ this approach is that you can't use it on-demand.

The storage is attached to this storage so if you did this.

getTimestampStr()
↳ Stores "T+0" in static variable space & returns pointer to it

getTimestamp()
↳ Stores "T+1" in static variable

After the 2nd call, the first functions output is removed / replaced.

(B) This is done by manually allocating memory & manually freeing it.

Say you have an object w/ some max or known size, & the size is > than say 4kB (4096 bytes), Its a good candidate to be heap allocated, Its also really useful if the object size could change in the future / it needs to be passed around

In our example, we have a `timestamp` array, that we've determined to be max 100 bytes (chars) in size, we can get a pointer to 100 bytes of space on the heap using malloc( )

MAN

```
char * timestamp = (char*) malloc(sizeof(char) * 100);
```

Notice that we can no longer keep the array size w/ its type since the compiler is unaware of what malloc() is doing, It only knows it will return a void* pointer.

So its up to us to make sure we are always in bounds of our heap variables or we can run into segfaults.

So since sizeof(timestamp) no longer knows the size of our array, we have to hardcode it.

```
snprintf(timestamp, 100, "%llu", time(NULL));
```

For this reason of often associating the size of our obtained pointer its very common to see a struct like this:

```
struct slice_t {
    void * ptr;
    size_t len;
}
```
↳ The correctly sized int type to hold any possible memory size

This is very often used type in Zig or other languages.

Once we are done using the heap memory object, Its common to free() it. This returns the memory to the OS for further use.

If you continually allocate memory without freeing it, that's called a memory leak & eventually the system won't allocate any more memory (OOM)

```
free(timestamp);
```

Remember to only call free() once after the memory is used. If you call free() again on an already freed variable that is a double free error.

The pointer that is freed is no longer valid after its freed, Using a freed variable call cause a use after free error or a segfault

If you want to resize the block given by malloc() use realloc()

```
timestamp = realloc(timestamp, 2000);
```

↓

timestamp is now 2kB. Original data preserved.

(3) This is a way of simply not making it your problem, you make it the callers responsibility to allocate & give you a large enough buffer which can be either heap or stack allocated.

The function can be changed to

```
void getTimestampStr(char* buf, size_t buf_s) {
    snprintf(buf, buf_s, "%llu", time(NULL));
}
```

& it could be called like this

```
int main() {
    char ts[100];
    getTimeStampStr(&ts[0], sizeof(ts));
}
```

STACK

```
int main() {
    char ts* = (char*) malloc(1024);
    getTimestampStr(ts, 1024);
}
```

HEAP

This one is best if you want the buffer size to be configurable or if the function directly writes to a byte array.

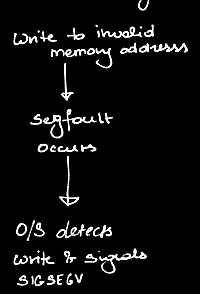
DEBUGGING INTRODUCTION

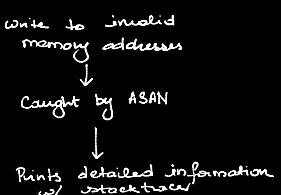
Even following best practices, you can very easily reach a segfault & it will make you a lot more confident if you can triage / diagnose & remedy it.

The diagnosis & remedy will depend on your specific project but I can show you some ways to triage issues.

-faddress = sanitizer

This compiler flag enables the adress sanitizer or ASAN, How it works isn't exactly relevant but you can think of it like this:





So when your program crashes, you get a nice helpful message