

# Basic Memory Introduction

An introduction to low-level programming  
and how to avoid

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

Note: I've pretty much  
only covered the  
kinds of errors you'd  
encounter in a single-  
threaded program.

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.

- Reference Counter → Increment a counter each time the obj is referenced & decrease it when it's not.
- Mark & Sweep → mark objects for cleanup.
- Generational Collection → Divide into young & old gens of objects.

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

But if you're reading this, you already decided you won't use one or for the following reasons 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: what 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 free() objects.
- High Performance: This is a bad reason to choose 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 bad 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 (e.g.).

It all depends on the algorithms & the data structures you employ.

# HOW

\*

ov - oversimplified  
for exple.■ - additional  
commentXYZ - link

## Primer

I'm going to over a few concepts here so that I can use the terms I'm comfortable later.

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 can't infer that from its type.

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

0x0
0x1
0x2

pointer = 0x1

A pointer simply points to one of these bytes.

QV

## Heap / Stack :

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

The fast tape is the Stack, the other is Heap

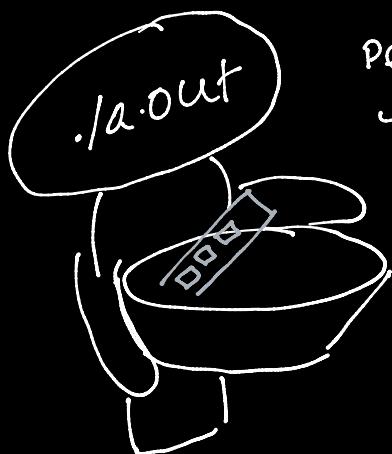
↓  
It's already managed for you

↓  
You manually manage this one.

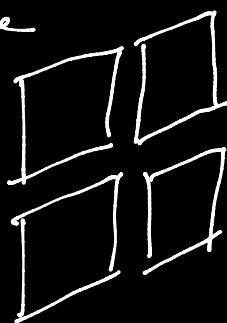
## Allocating / Freeing:

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  $500 \times 4$  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.



Please sir, may I  
have some more  
memory



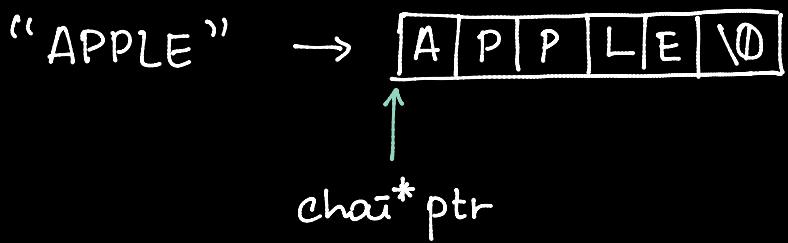
The opposite process, returning the memory to the operating system (ie giving up ownership) is called freeing.

## 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-read common issues you could face & see what's 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.

Let's take the following:

1. char\* getTimestampStr() {
2. char timestamp[100];
3. sprintf(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 is a stack variable ]

\* In this we put the UNIX timestamp into that variable,

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

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

↓  
Casting away the known array type.

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

↓  
C doesn't 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)

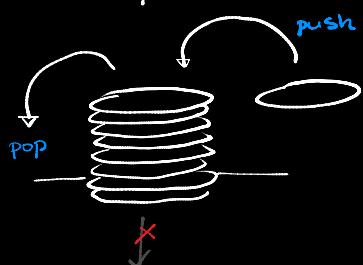
### getTimestampStr:

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

Pushing data increments the SP

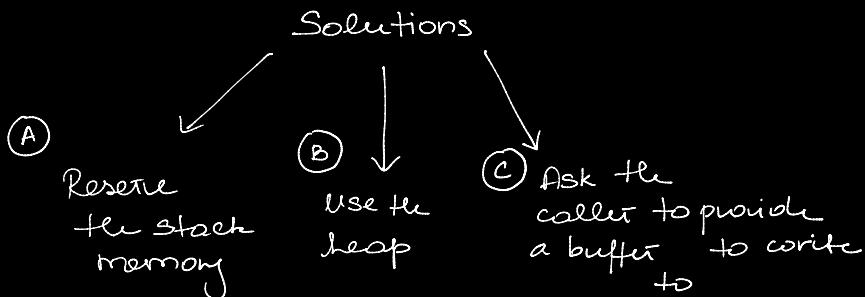
Popping data decrements the SP

Now we can look at the instructions.

- <sub>1</sub> → This reserves 112 bytes of space on the stack ( $112 \rightarrow 100 + \text{padding}$  to 16b boundary)
- <sub>2</sub> → Sets the address of the array to the current stack pointer address.
- <sub>3</sub> → 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 opis using (B) or (C) in order to pass non primitive data back to the caller.

(A) This is done by marking the array `static`

```
10 .LC0:  
11     .string "%llu"  
12     .text  
13     .globl gettimeofdayStr  
14     .type gettimeofdayStr, @function  
15 gettimeofdayStr:  
16     endbr64  
17     push rbx #  
18 # t.c:6: sprintf(timestamp, sizeof(timestamp), "%llu", time(NULL));  
19     mov edi, 0 #,  
20     call time@PLT #  
21     mov r9, rax # tmp86,  
22 # /usr/include/x86_64-linux-gnu/bits/stdio2.h:54: return __builtin_snprintf_chk __s, __n, __USE_FORTIFY_LEVEL  
23     lea rbx, .LC0[rip] #,  
24     mov ecx, 100 #,  
25     mov edx, 2 #,  
26     mov esi, 100 #  
27     lea rbx, [timestamp.0[rip]] -----> Function static storage  
28     mov rdi, rbx #, tmp84  
29     mov eax, 0 #,  
30     call __snprintf_chk@PLT #  
31 # t.c:8: }  
32     mov rax, rbx #, tmp84  
33     pop rbx #  
34     ret  
35     .size gettimeofdayStr, .getTimestampStr  
36     .local timestamp.0  
37     .comm timestamp.0,100,32  
38     .section .note.GNU-stack,"",@progbits  
39     .section .note.gnu.property,"a"  
40     .align 8  
41     .long 1f - 0f  
42     .long 4f - 1f  
43     .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. It's 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 cont. lease 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 2<sup>nd</sup> call, the first function's output is removed/replaced.

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

Say you have an object w/ some known size, & the size is > than say 4kB (4096 bytes), Its a good candidate to be leap 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're determined to be more <sup>(class)</sup> 100 bytes 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 select 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, this causes a memory leak & eventually the system won't allocate any more memory (oom).

```
free(timestamp);
```

Remember to only call `free()` once of the memory is used. If you call `free()` again on an already freed variable that is a double free error.  
The pointer itself is no longer valid after its freed, using a freed variable will 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 caller's responsibility to allocate & give you a large enough buffer which can be either heap or stack deallocated.

The function can be elongated to

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

If it could be called like this

```

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

```

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

STACK

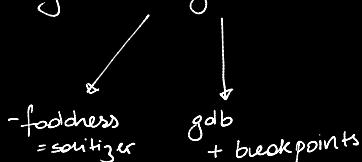
## HEAP

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

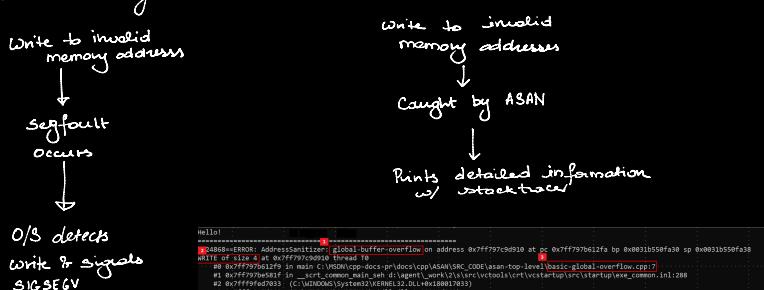
Even following best practices, you can very easily reach a segfault & it will make you a lot more confident if you can triage / diagnose & remedy it.

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



-f address = sanitizer

This complaint ~~feigns~~ enables the address sanitizer or ASAN, how it works isn't exactly relevant but you can think of it like this:



Generally mapping a breakpoint to source code requires the program to be compiled w/ debugging information. ie → it should not be stripped.

↳ Hoe 'entia'  
info. vermaend.

So when you program crashes,  
you get a nice helpful message  
like this:

4

As you can see, it provides the line no. of where the memory error occurred & a stacktrace.

Alternatively you can use a debugger. I'm not going to go very in-depth on this since there is simply so much you can do using a debugger but for 'basic' debugging, what you need is:

1) The executable must be built with debug symbols. Generally this is done with '-g'.

Note: The optimization level you compile with will determine the kind of assembly you inspect, use '-O0' for the closest equivalent of your C code.

2) Run your program

3) When an exception occurs, inspect the backtrace & the stack / variables

You can also set breakpoints for the prog. to auto-stop when reached & watch variables for when they are read or written to.

## Misc

Tools like Clang w/ CppFormat turned on can help you catch errors, early

Simple Rule of Thumb is whenever passing a pointer to MULTIPLE elements, pass in a length parameter as well.

\* I apologize for the constant shifts in the size of my handwriting. It took me a while to figure that I should keep it large & consistent for legibility.

I would've typed but if I did that I would've never finished it hehe.

