



DYNAMIC MEMORY ALLOCATION CHALLENGES IN SAFETY CRITICAL SYSTEMS

XAVIER BONAVENTURA

BMW Group

ABOUT ME

- Where to find me:
 -  @limdor
 -  @xbonaventurab
- Working in C++
 - since 2011
 - in safety critical systems since 2018



- I represent BMW at the MISRA C++ working group and at the ISO C++ committee meetings

ABOUT YOU

- How many of you are not allowed to use dynamic memory allocation?
- How many of you do not know what are the issues with dynamic memory allocation?

DEFINITION OF A SAFETY CRITICAL SYSTEM

- A system whose failure could lead to:
 - Loss of life or serious injuries
 - Significant property damage
 - Significant environmental damage



FUNCTIONAL SAFETY IS KEY IN SAFETY CRITICAL SYSTEMS

- What is functional safety?
 - “Protecting a user from technology”
 - “Systems that lead to the freedom from unacceptable risk of injury or damage to the health of people by the proper implementation of one or more automatic protection functions (often called safety functions). A safety system consists of one or more safety functions.”

TÜV SÜD

FUNCTIONAL SAFETY IS KEY IN SAFETY CRITICAL SYSTEMS

- Safety in a kettle
 - Automatic switch off
 - Burning risk
 - Boil dry protection
 - Fire hazard due to catch fire
 - Pouring without splashing
 - Burning risk



THERE ARE MULTIPLE STANDARDS TO ENSURE FUNCTIONAL SAFETY

- One standard to rule them all

IEC 61508 – General

- One standard for each industry

IEC 61511 – Industrial
processes

IEC 62061 –
Machinery

IEC 60880 – Nuclear power
plants

ISO 26262 – Automotive

IEC 62304 – Medical
devices

EN 50128 – Railway

NOT EVERYTHING IS EQUALLY CRITICAL (ISO 26262 – AUTOMOTIVE)

- Multiple categories to define criticality (Hazard and Risk Analysis)
 - Severity
 - If something happens, how bad is it?
 - Exposure
 - How frequently are you exposed to that situation?
 - Controllability
 - If something bad happens, how hard is for the user to control the situation?

ASIL LEVELS IN ISO 26262

Severity	Exposure	Controllability		
		Simple	Normal	Difficult
Light and moderate injuries	Very low	QM	QM	QM
	Low	QM	QM	QM
	Medium	QM	QM	A
	High	QM	A	B
Severe and life-threatening injuries	Very low	QM	QM	QM
	Low	QM	QM	A
	Medium	QM	A	B
	High	A	B	C
Life threatening injuries and fatal injuries	Very low	QM	QM	A or QM
	Low	QM	A	B
	Medium	A	B	C
	High	B	C	D

REQUIREMENTS PROVIDED BY THE STANDARD

- ISO 26262-6:2018 - Road vehicles — Functional safety — Part 6: Product development at the software level

ASIL				
	A	B	C	D
Entry 1	o	+	+	++
Entry 2	+	+	++	++
Entry 3	++	+	o	o

- ++ Method is highly recommended
- + Method is recommended
- o No recommendation for or against

REQUIREMENTS PROVIDED BY THE STANDARD

- ISO 26262-6:2018 - Road vehicles — Functional safety — Part 6: Product development at the software level
 - General topics for the product development at the software level (Clause 5)
 - Requires use of coding guidelines that covers (5.4.3)
 - Use of language subsets with the goal of excluding language constructs that
 - can be interpreted different by different developers
 - experience tell us that they can lead to mistakes
 - can lead to unhandled run-time errors

REQUIREMENTS PROVIDED BY THE STANDARD

- ISO 26262-6:2018 - Road vehicles — Functional safety — Part 6: Product development at the software level
 - Software architectural design (Clause 7)
 - To avoid systematic faults should consider (7.4.3)
 - Appropriate management of shared resources

REQUIREMENTS PROVIDED BY THE STANDARD

- ISO 26262-6:2018 - Road vehicles — Functional safety — Part 6: Product development at the software level
 - Software unit design and implementation (Clause 8)
 - To achieve (8.4.5)
 - Correct order of execution of subprograms and functions
 - Correctness of data and control flow
 - The following principles should be applied
 - No dynamic objects or variables, or else online test during their creation
 - Avoid global variables or else justify their usage
 - No hidden data flow or control flow

DYNAMIC MEMORY ALLOCATION IN CODING GUIDELINES

- JSF++ Joint Strike Fighter Air Vehicle C++ coding standards for the system development and demonstration program
- AUTOSAR Guidelines for the use of the C++14 language in critical and safety-related systems
- MISRA C++:2023 Guidelines for the use of C++17 in critical systems

Do not use dynamic memory
allocation

Do not use dynamic memory
allocation after initialization
phase

If you have to, they provide
guidelines

DYNAMIC MEMORY ALLOCATION HAS POTENTIAL PROBLEMS

Memory leaks

Out of memory

Allocation/deallocation
mismatch

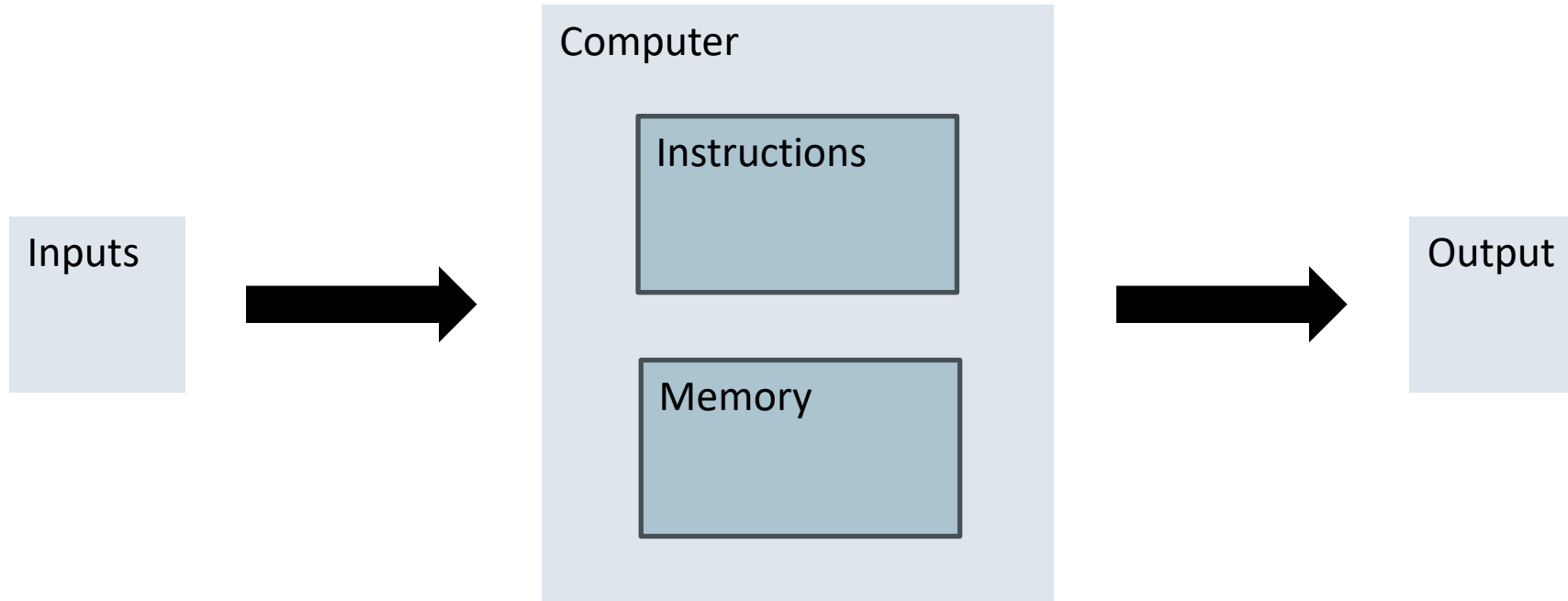
Memory fragmentation

Non-deterministic
runtime

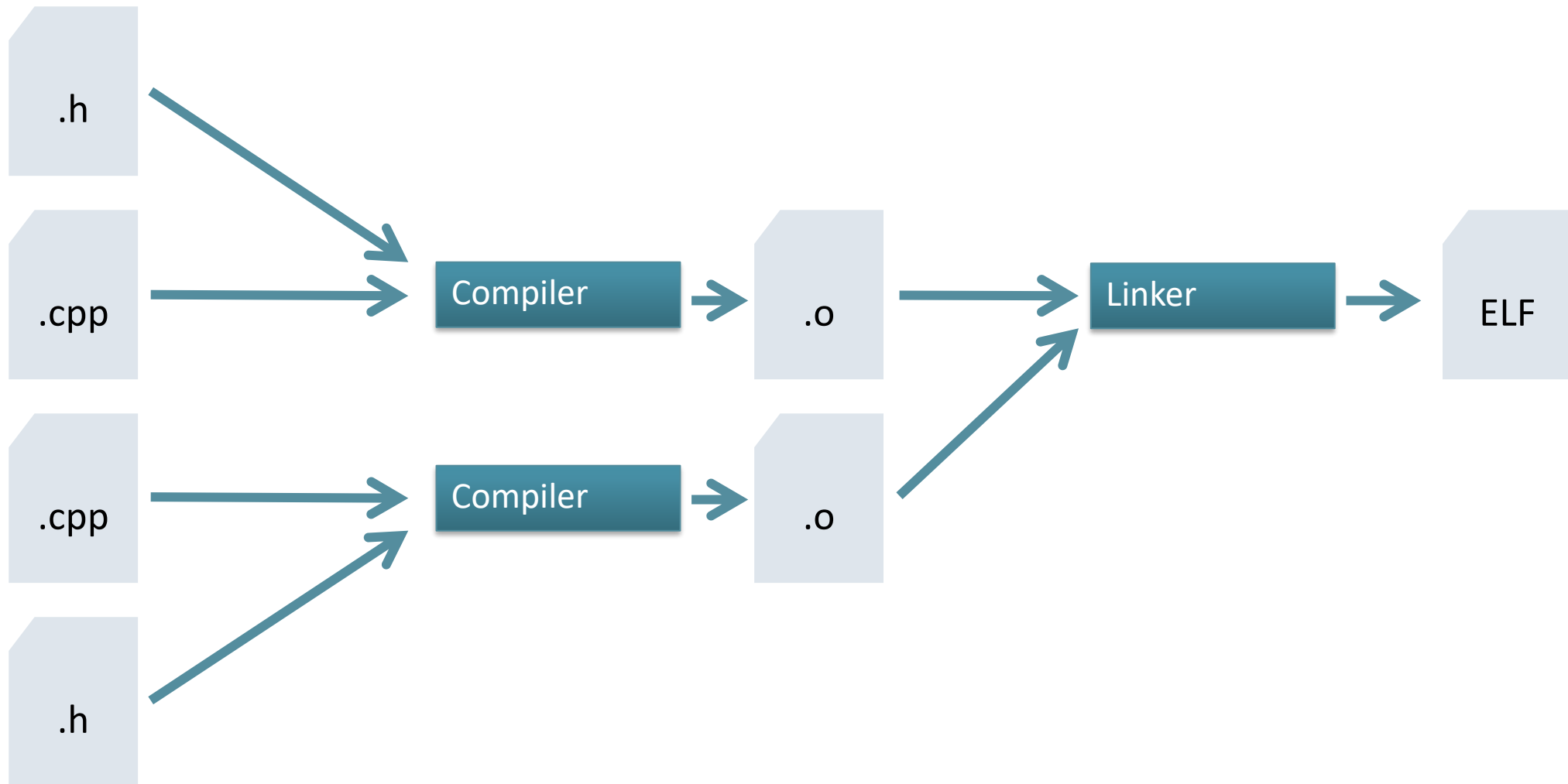
Use after free

ANATOMY OF A COMPUTER PROGRAM

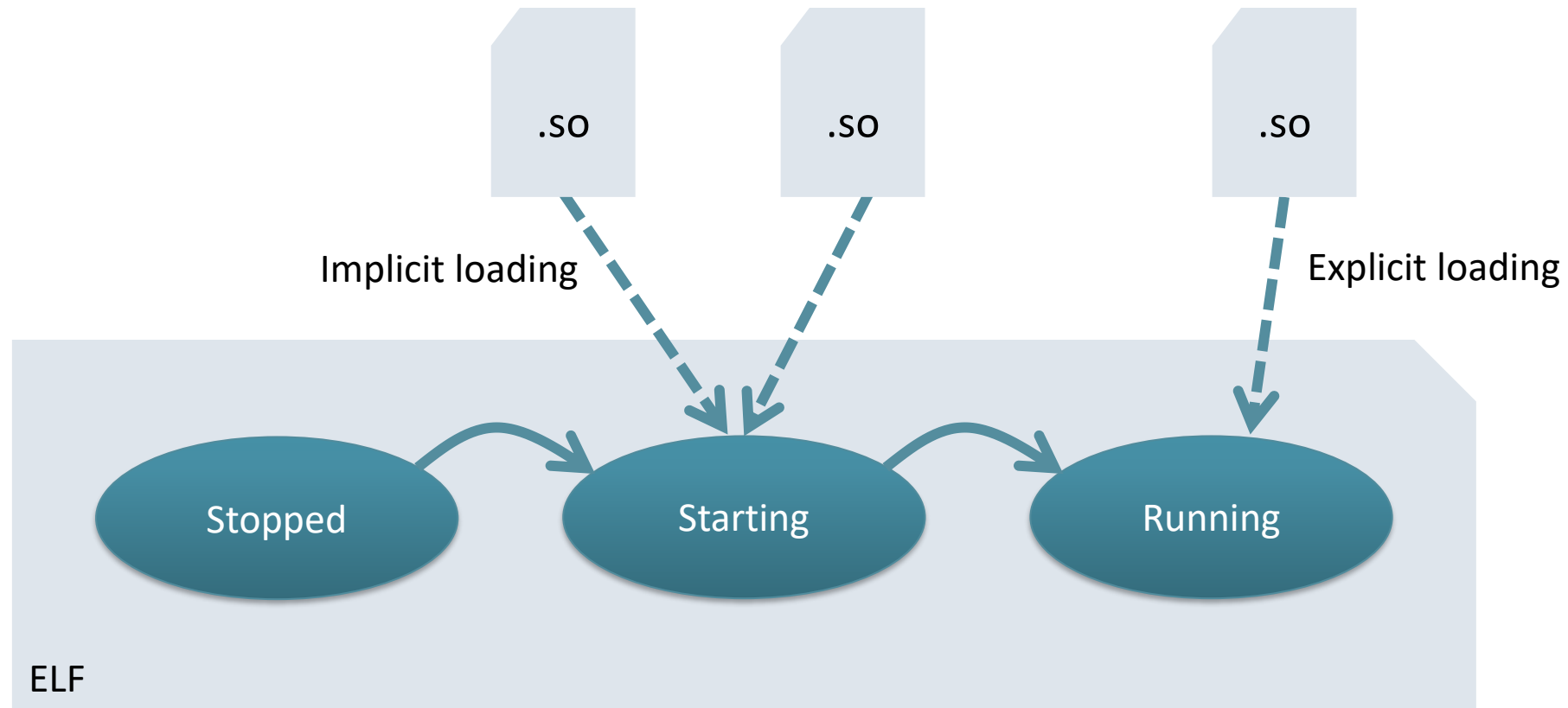
- A program is nothing else than a bunch of instructions modifying chunks of memory based on some inputs to produce some outputs



PROCESS LAYOUT IN MEMORY



PROCESS LAYOUT IN MEMORY



EXECUTABLE LAYOUT – ELF FORMAT

- Demo:

- https://github.com/limdor/dynamic_memory/tree/master/dynamic_memory/elf_format
- `bazel build //dynamic_memory/elf_format:main -c dbg`
- `readelf bazel-bin/dynamic_memory/elf_format/main --all`



EXECUTABLE LAYOUT – ELF FORMAT

```
constexpr auto c_foo = "NDC TechTown 2025";  
static constexpr auto sc_foo = "NDC TechTown 2025";  
  
int bar;  
static int s_bar;  
  
int d = 42;  
static int s_d = 42;  
  
const int c_d = 42;  
static const int sc_d = 42;  
  
int main()  
{  
    return 0;  
}
```

EXECUTABLE LAYOUT – ELF FORMAT

```
:~/dynamic_memory$ readelf bazel-bin/dynamic_memory/elf_format/main --sections
```

There are 39 section headers, starting at offset 0x1bc0:

Section Headers:

[Nr]	Name	Type	Address	Offset	Size	EntSize	Flags	Link	Info	Align
[0]		NULL	0000000000000000	00000000	0000000000000000	0000000000000000		0	0	0
[1]	.interp	PROGBITS	0000000000000270	00000270	000000000000001c	0000000000000000	A	0	0	1
[2]	.note.gnu.pr[...]	NOTE	0000000000000290	00000290	0000000000000030	0000000000000000	A	0	0	8
[3]	.note.ABI-tag	NOTE	00000000000002c0	000002c0	0000000000000020	0000000000000000	A	0	0	4
[4]	.note.gnu.bu[...]	NOTE	00000000000002e0	000002e0	0000000000000024	0000000000000000	A	0	0	4
[5]	.dynsym	DYNSYM	0000000000000308	00000308	0000000000000090	0000000000000018	A	6	1	8
[6]	.dynstr	STRTAB	0000000000000398	00000398	00000000000000a1	0000000000000000	A	0	0	1
[7]	.gnu.hash	GNU_HASH	0000000000000440	00000440	000000000000001c	0000000000000000	A	5	0	8
[8]	.gnu.version	VERSYM	000000000000045c	0000045c	000000000000000c	0000000000000002	A	5	0	2
[9]	.gnu.version_r	VERNEED	0000000000000468	00000468	0000000000000030	0000000000000000	A	6	1	4
[10]	.rela.dyn	RELA	0000000000000498	00000498	00000000000000f0	0000000000000018	A	5	0	8
[11]	.rela.plt	RELA	0000000000000588	00000588	0000000000000018	0000000000000018	AI	5	13	8
[12]	.init	PROGBITS	00000000000005a0	000005a0	000000000000001b	0000000000000000	AX	0	0	4
[13]	.plt	PROGBITS	00000000000005c0	000005c0	0000000000000030	0000000000000010	AX	0	0	16
[14]	.text	PROGBITS	00000000000005f0	000005f0	00000000000000f8	0000000000000000	AX	0	0	16
[15]	.fini	PROGBITS	00000000000006e8	000006e8						

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
D (mbind), l (large), p (processor specific)

EXECUTABLE LAYOUT – ELF FORMAT

```
:~/dynamic_memory$ readelf bazel-bin/dynamic_memory/elf_format/main --symbols
```

Symbol table '.symtab' contains 41 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	000000000000002c0	32	OBJECT	LOCAL	DEFAULT	3	__abi_tag
2:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
3:	00000000000002018	0	OBJECT	LOCAL	DEFAULT	26	__TMC_LIST__
4:	00000000000000620	0	FUNC	LOCAL	DEFAULT	14	deregister_tm_clones
5:	00000000000000650	0	FUNC	LOCAL	DEFAULT	14	register_tm_clones
6:	00000000000000690	0	FUNC	LOCAL	DEFAULT	14	__do_global_dtors_aux
7:	00000000000002018	1	OBJECT	LOCAL	DEFAULT	27	completed.0
8:	00000000000001d88	0	OBJECT	LOCAL	DEFAULT	20	__do_global_dtor[...]
9:	000000000000006d0	0	FUNC	LOCAL	DEFAULT	14	frame_dummy
10:	00000000000001d90	0	OBJECT	LOCAL	DEFAULT	21	__frame_dummy_in[...]
11:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	main.cpp
12:	00000000000001d78	8	OBJECT	LOCAL	DEFAULT	19	_ZL5c_foo
13:	00000000000001d80	8	OBJECT	LOCAL	DEFAULT	19	_ZL6sc_foo
14:	00000000000002020	4	OBJECT	LOCAL	DEFAULT	27	_ZL5s_bar
15:	00000000000002014	4	OBJECT	LOCAL	DEFAULT	25	_ZL3s_d
16:	00000000000000714	4	OBJECT	LOCAL	DEFAULT	16	_ZL3c_d
17:	00000000000000718	4	OBJECT	LOCAL	DEFAULT	16	_ZL4sc_d
18:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c

EXECUTABLE LAYOUT – ELF FORMAT

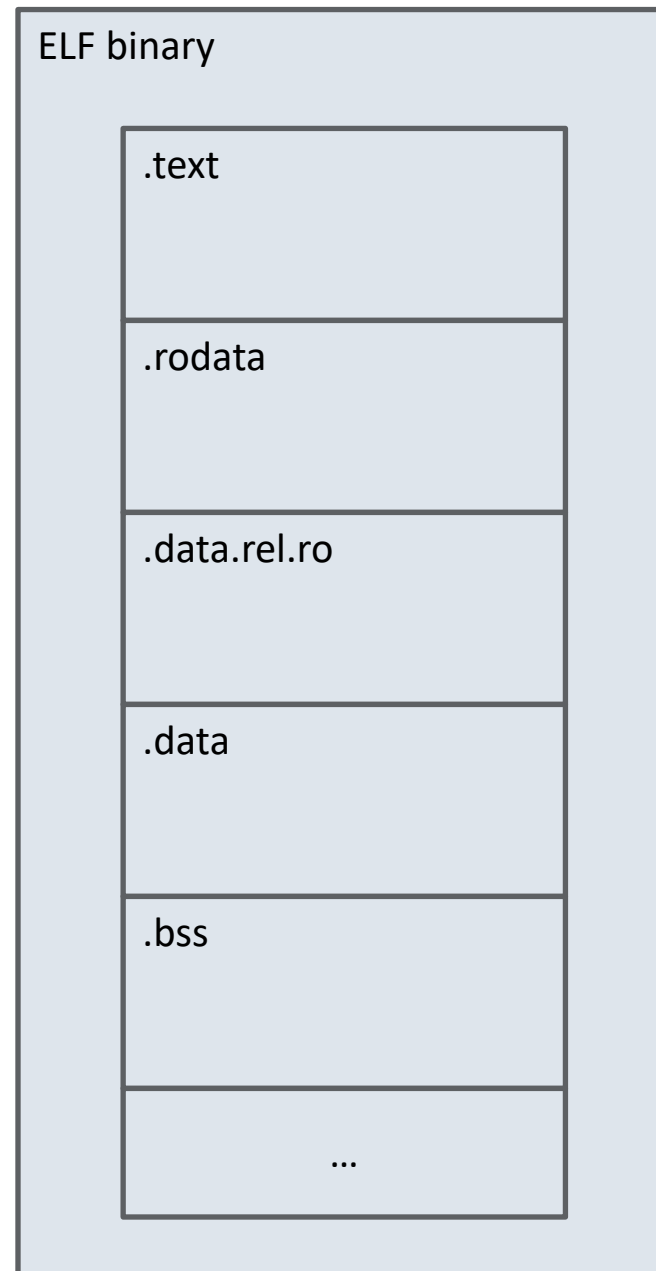
```
constexpr auto c_foo = "NDC TechTown 2025";  
static constexpr auto sc_foo = "NDC TechTown 2025";
```

```
int bar;  
static int s_bar;
```

```
int d = 42;  
static int s_d = 42;
```

```
const int c_d = 42;  
static const int sc_d = 42;
```

```
int main()  
{  
  
    return 0;  
}
```



EXECUTABLE LAYOUT – ELF FORMAT

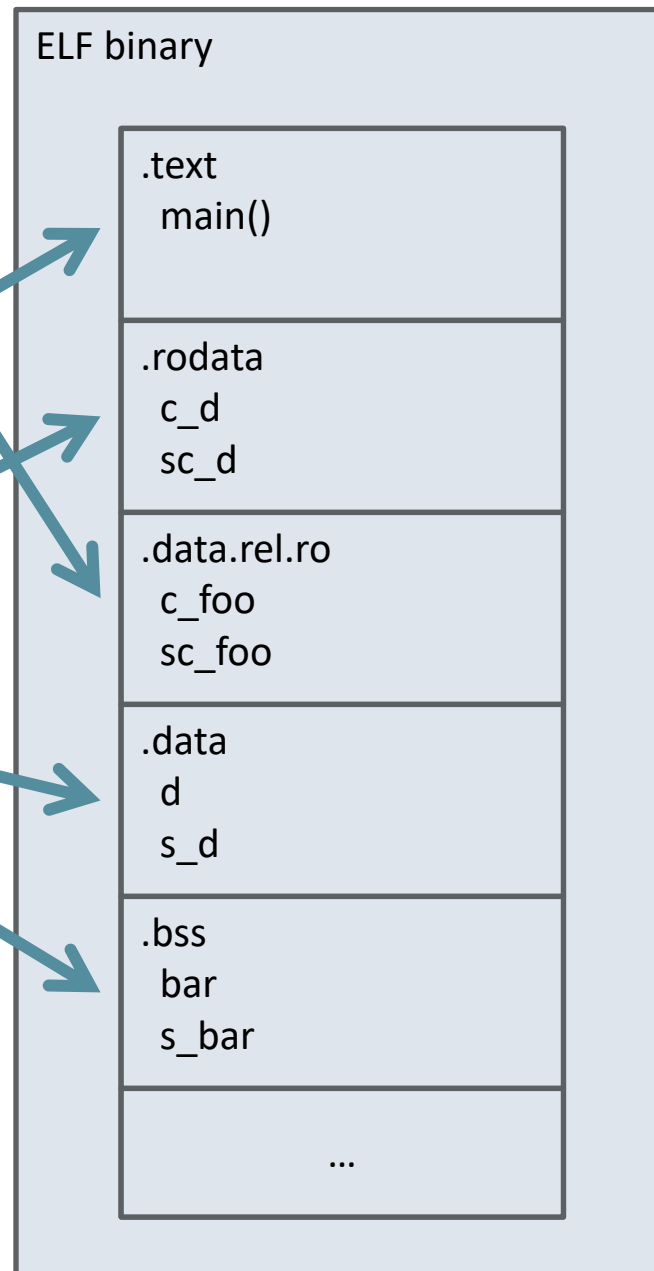
```
constexpr auto c_foo = "NDC TechTown 2025";  
static constexpr auto sc_foo = "NDC TechTown 2025";
```

```
int bar;  
static int s_bar;
```

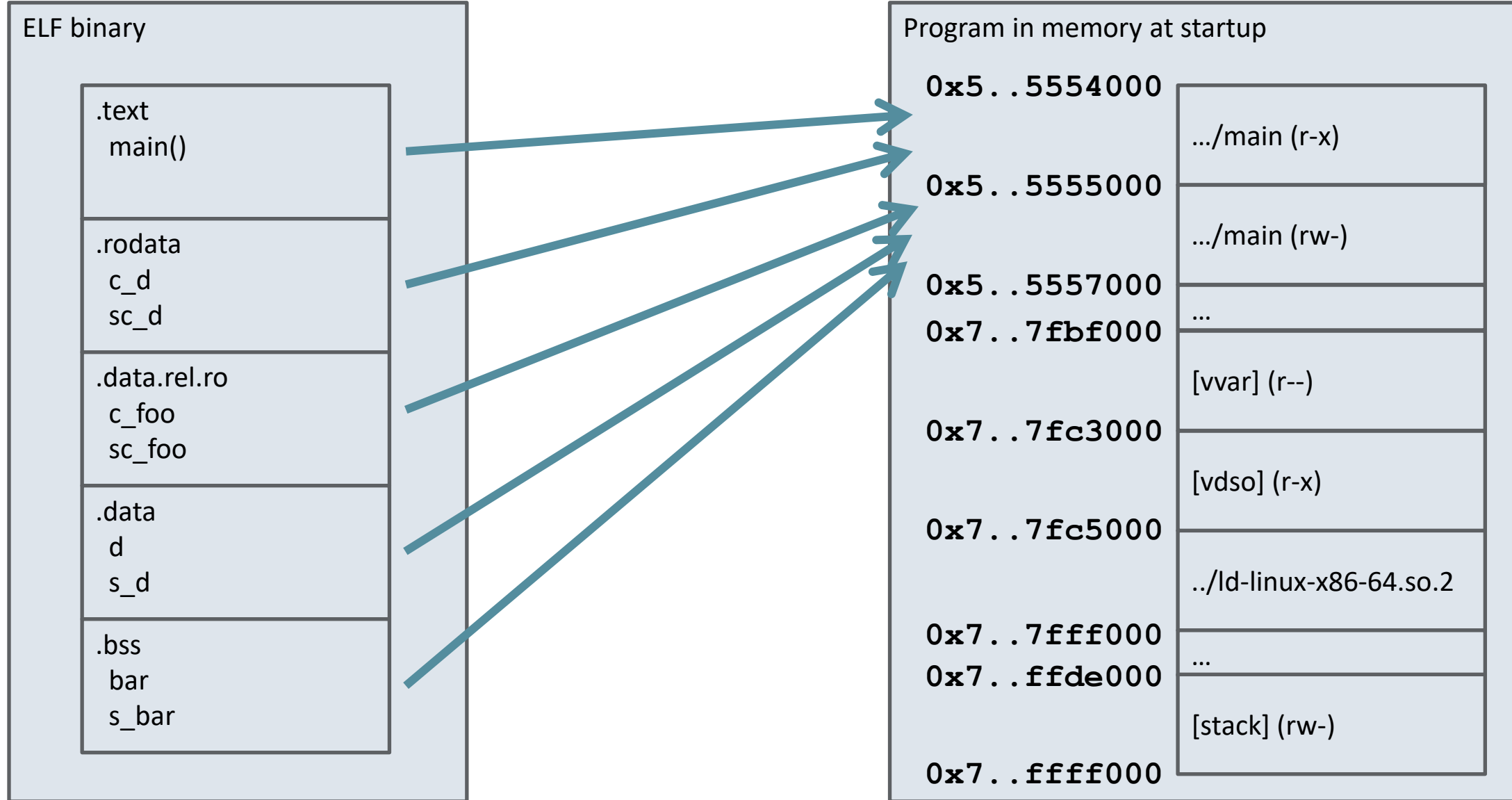
```
int d = 42;  
static int s_d = 42;
```

```
const int c_d = 42;  
static const int sc_d = 42;
```

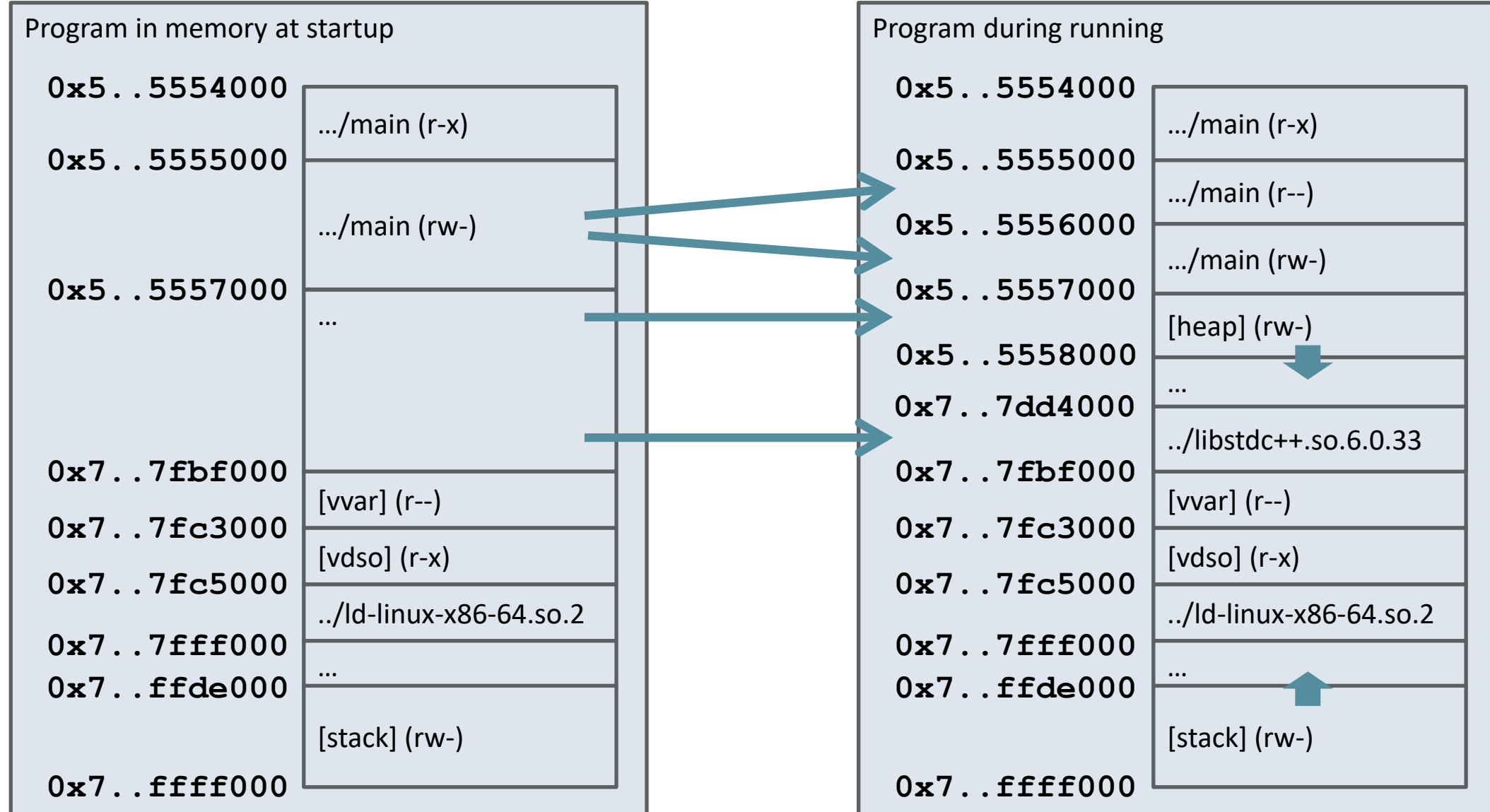
```
int main()  
{  
    return 0;  
}
```



PROCESS LAYOUT IN MEMORY – LINUX



PROCESS LAYOUT IN MEMORY – LINUX



GROWING STACK

- Demo:

- https://github.com/limdor/dynamic_memory/tree/master/dynamic_memory/growing_stack
- `bazel build //dynamic_memory/growing_stack:main -c dbg`
- `gdb bazel-bin/dynamic_memory/growing_stack/main`
 - `break main.cpp:5 if n == 0`



GROWING STACK

```
unsigned long compute(unsigned long n)
{
    if (n == 0)
    {
        return 0;
    }
    return 1 + compute(n - 1);
}

int main()
{
    int n;
    do
    {
        std::cout << "Number to be computed: ";
        std::cin >> n;
        std::cout << "Computed number: " << compute(n) << "\n";
    } while (n != 0);
    return 0;
}
```


GROWING STACK

```
(gdb) info proc mappings
```

```
process 24235
```

```
Mapped address spaces:
```

Start Addr	End Addr	Size	Offset	Perms	objfile
0x555555554000	0x555555555000	0x1000	0x0	r-xp	/growing_stack/main
0x555555555000	0x555555557000	0x2000	0x0	rw-p	/growing_stack/main
0x7ffff7fbf000	0x7ffff7fc3000	0x4000	0x0	r--p	[vvar]
0x7ffff7fc3000	0x7ffff7fc5000	0x2000	0x0	r-xp	[vdso]
0x7ffff7fc5000	0x7ffff7fc6000	0x1000	0x0	r--p	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fc6000	0x7ffff7ff1000	0x2b000	0x1000	r-xp	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ff1000	0x7ffff7ffb000	0xa000	0x2c000	r--p	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ffb000	0x7ffff7fff000	0x4000	0x36000	rw-p	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ffde000	0x7ffff7fff000	0x21000	0x0	rw-p	[stack]



132 KB

GROWING STACK

```
unsigned long compute(unsigned long n)
{
    if (n == 0)
    {
        return 0;
    }
    return 1 + compute(n - 1);
}

int main()
{
    int n;
    do
    {
        std::cout << "Number to be computed: ";
        std::cin >> n;
        std::cout << "Computed number: " << compute(n) << "\n";
    } while (n != 0);
    return 0;
}
```

0x21000 0x0 rw-p [stack]



132 KB

Number to be computed: 20000

0x9e000 0x0 rw-p [stack]



632 KB

Number to be computed: 0

0x9e000 0x0 rw-p [stack]



632 KB

GROWING HEAP

- Demo:

- https://github.com/limdor/dynamic_memory/tree/master/dynamic_memory/growing_heap
- `bazel run //dynamic_memory/growing_heap:main`



GROWING HEAP

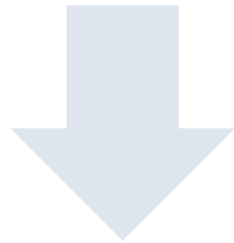
```
--- Heap memory region info for pid 41035 (start) ---  
Pathname: [heap]          Size: 135168 bytes  
Start address: 93999983702016 (0x557e13a33000)  
End address: 93999983837184 (0x557e13a54000)  
--- Heap memory region info (end) ---
```

What would you like to do?

- a - Allocate
- i - Random initialize memory
- p - Print pointer addresses
- d - Deallocate
- q - Quit

GROWING HEAP

```
--- Heap memory region info for pid 41035 (start) ---  
Pathname: [heap]          Size: 135168 bytes  
Start address: 93999983702016 (0x557e13a33000)  
End address: 93999983837184 (0x557e13a54000)  
--- Heap memory region info (end) ---
```

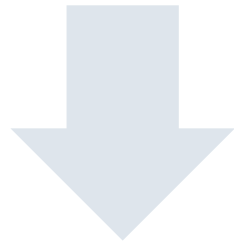


Allocate 50.000 bytes

```
--- Heap memory region info for pid 41035 (start) ---  
Pathname: [heap]          Size: 270336 bytes  
Start address: 93999983702016 (0x557e13a33000)  
End address: 93999983972352 (0x557e13a75000)  
--- Heap memory region info (end) ---
```

GROWING HEAP

```
--- Heap memory region info for pid 41035 (start) ---  
Pathname: [heap]          Size: 270336 bytes  
Start address: 93999983702016 (0x557e13a33000)  
End address: 93999983972352 (0x557e13a75000)  
--- Heap memory region info (end) ---
```

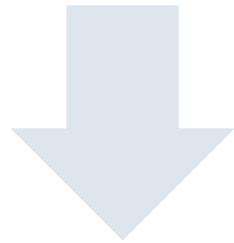


Allocate 100.000 bytes (150.000 bytes until now)

```
--- Heap memory region info for pid 41035 (start) ---  
Pathname: [heap]          Size: 270336 bytes  
Start address: 93999983702016 (0x557e13a33000)  
End address: 93999983972352 (0x557e13a75000)  
--- Heap memory region info (end) ---
```

GROWING HEAP

```
--- Heap memory region info for pid 41035 (start) ---  
Pathname: [heap]      Size: 270336 bytes  
Start address: 93999983702016 (0x557e13a33000)  
End address: 93999983972352 (0x557e13a75000)  
--- Heap memory region info (end) ---
```



Allocate 200.000 bytes (350.000 bytes until now)

```
--- Heap memory region info for pid 41035 (start) ---  
Pathname: [heap]      Size: 270336 bytes  
Start address: 93999983702016 (0x557e13a33000)  
End address: 93999983972352 (0x557e13a75000)  
--- Heap memory region info (end) ---
```

GROWING HEAP

What would you like to do?

a - Allocate

→ i - Random initialize memory

→ p - Print pointer addresses

d - Deallocate

q - Quit

50000 bytes at **0x557e13a47cc0**

Pathname: [heap] Size: 270336 bytes

Start address: 93999983702016 (0x557e13a33000)

End address: 93999983972352 (0x557e13a75000)

100000 bytes at **0x557e13a54020**

Pathname: [heap] Size: 270336 bytes

Start address: 93999983702016 (0x557e13a33000)

End address: 93999983972352 (0x557e13a75000)

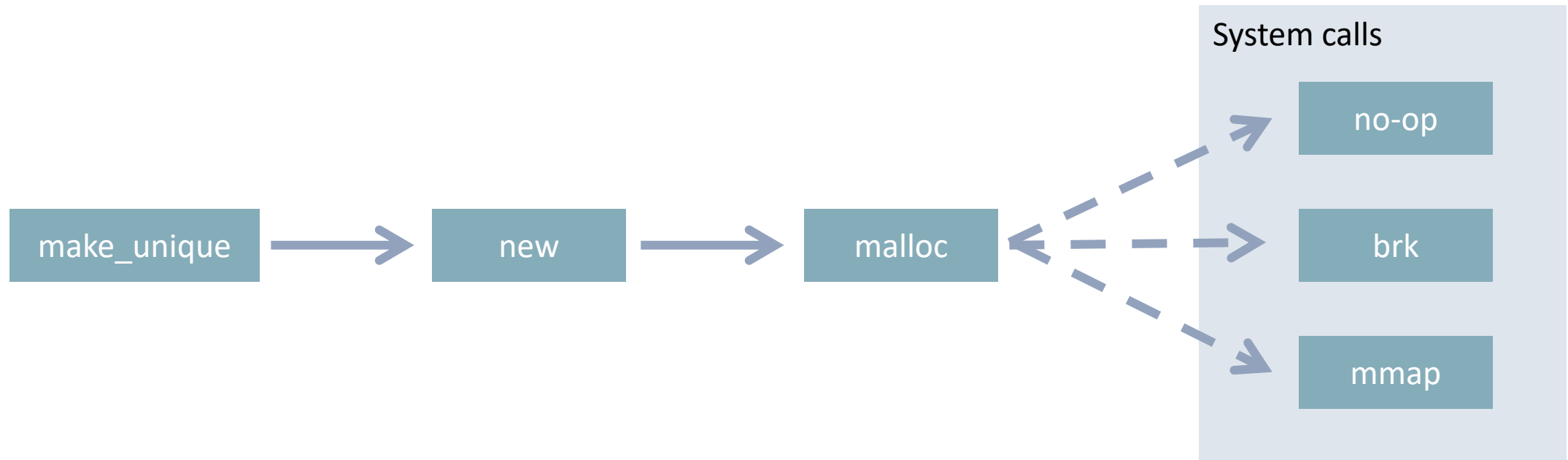
200000 bytes at **0x7f4781bc8010**

Pathname: Size: 221184 bytes

Start address: 139945096019968 (0x7f4781bc8000)

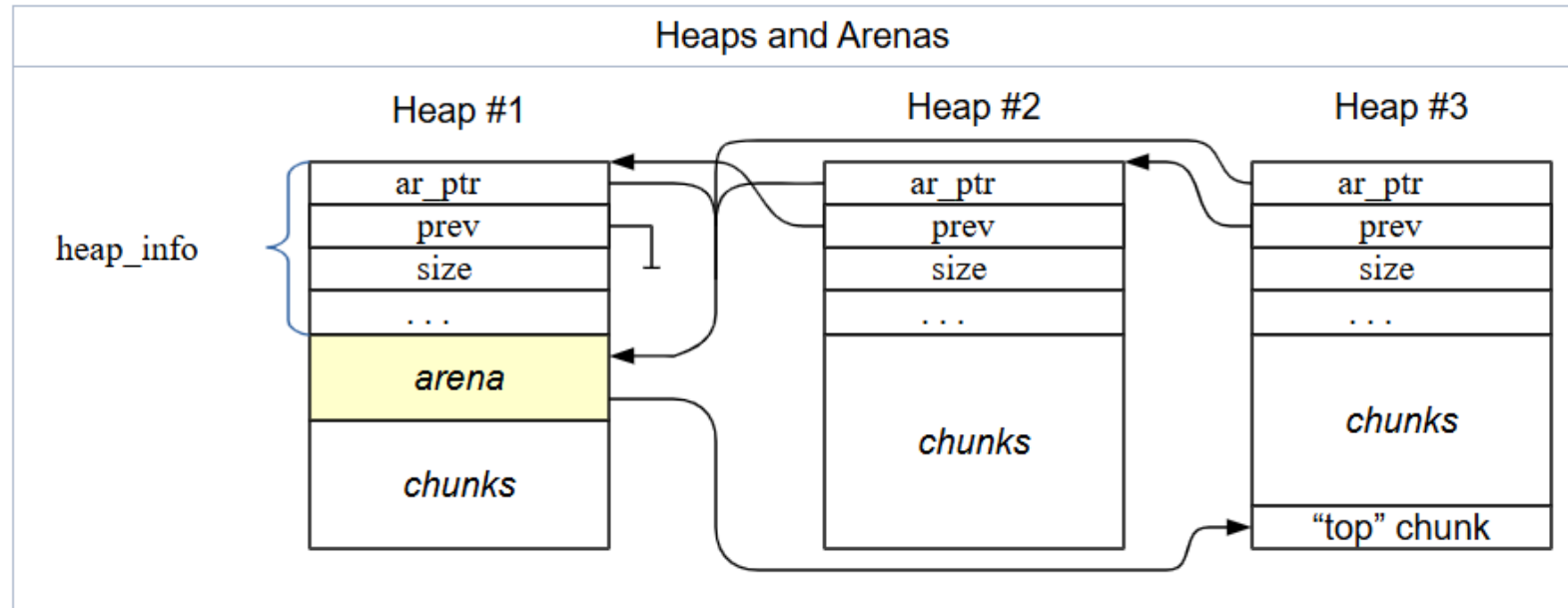
End address: 139945096241152 (0x7f4781bfe000)

THERE IS NO SINGLE HEAP MEMORY REGION



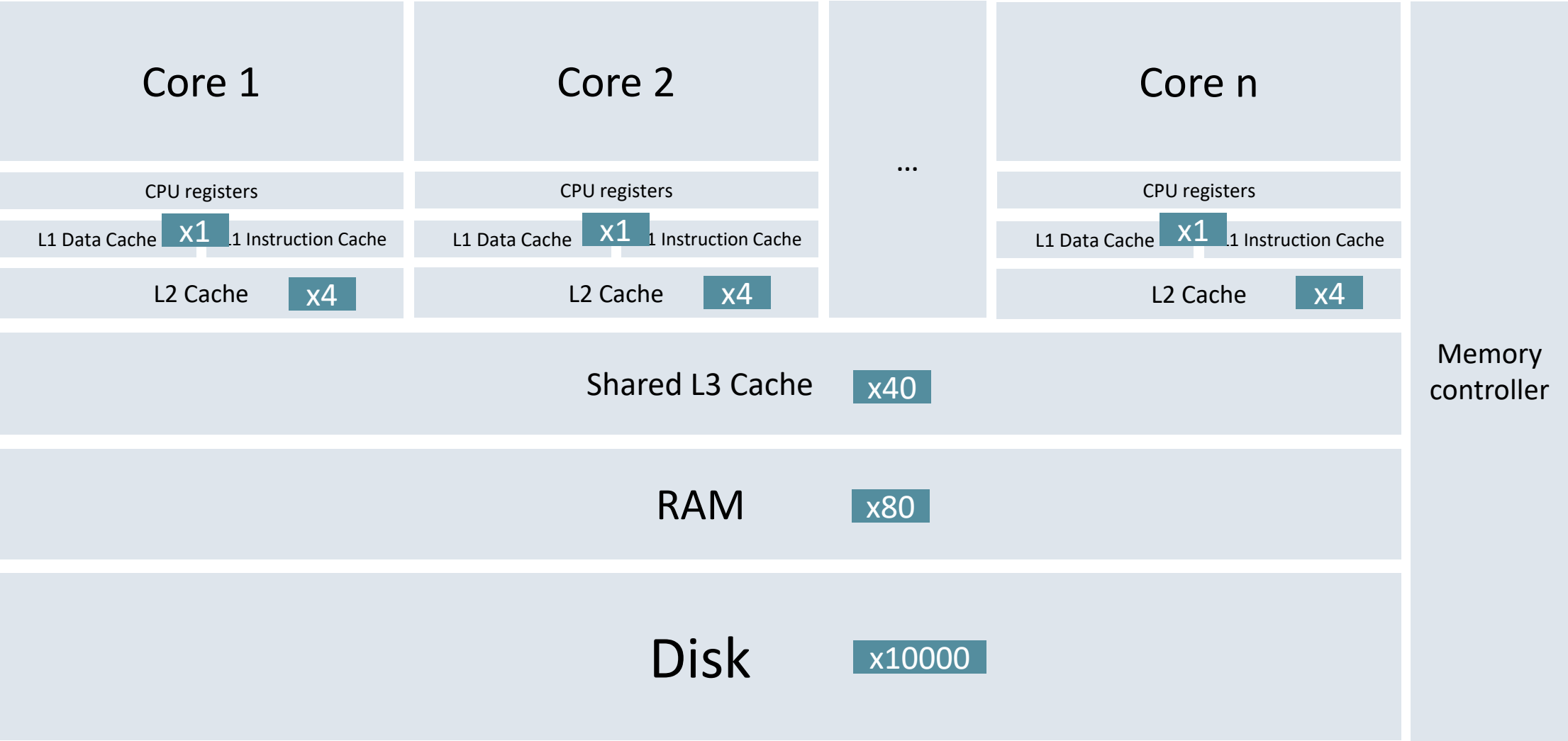
THERE IS NO SINGLE HEAP MEMORY REGION

- Behavior of malloc depends on the implementation

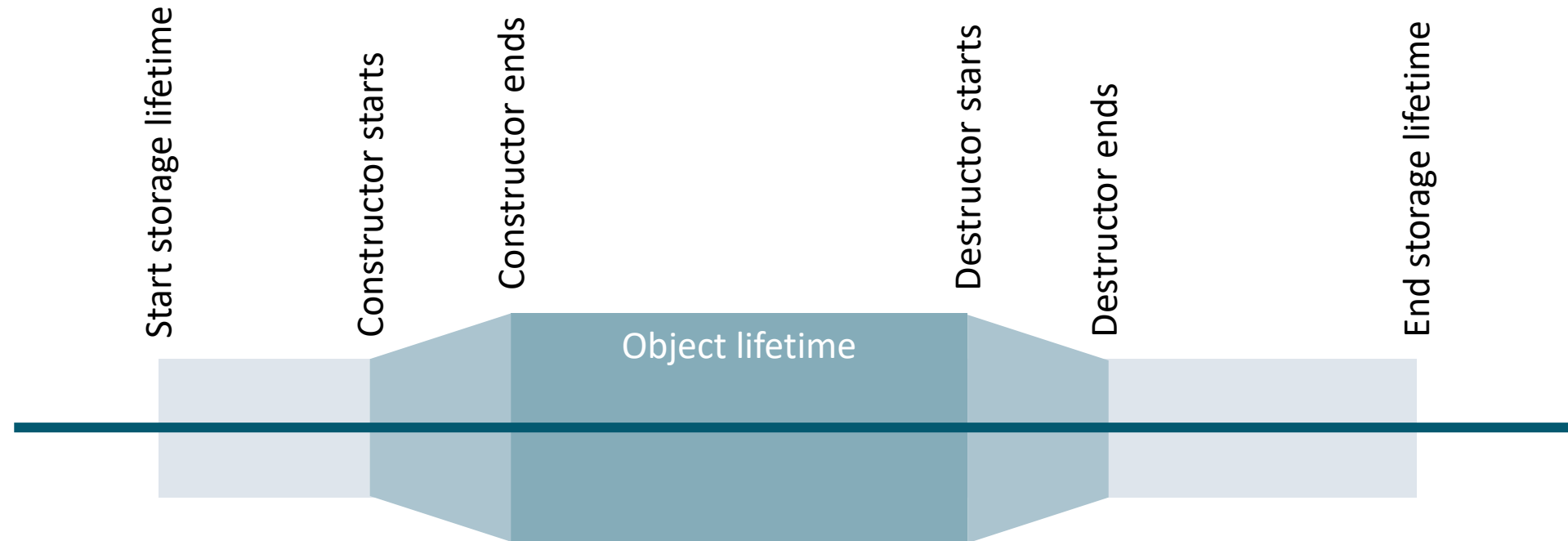


- <https://sourceware.org/glibc/wiki/MallocInternals>

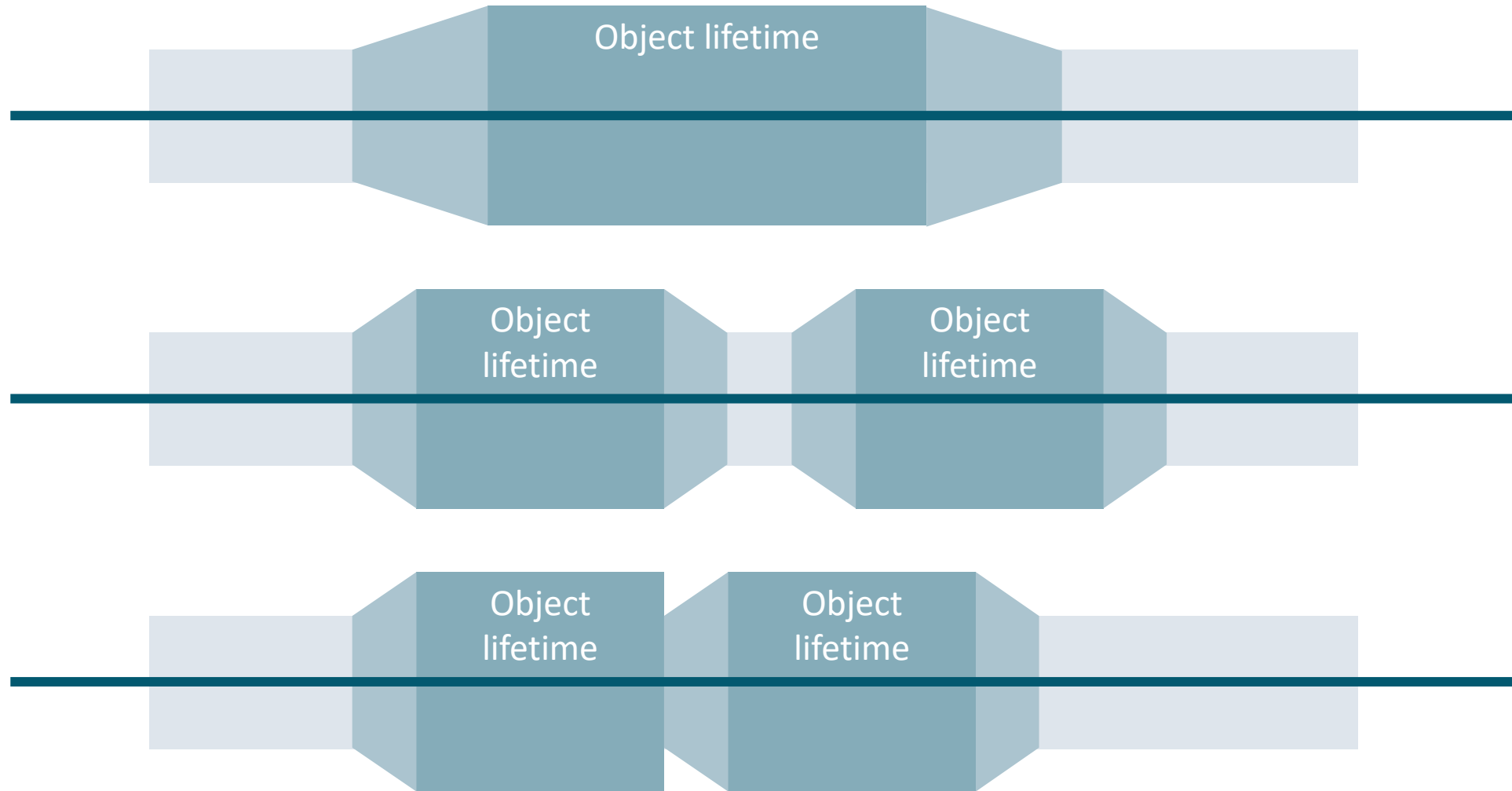
COMPUTER ARCHITECTURE



OBJECT AND STORAGE ARE DIFFERENT THINGS



OBJECT AND STORAGE ARE DIFFERENT THINGS



STORAGE DURATION

Static storage duration

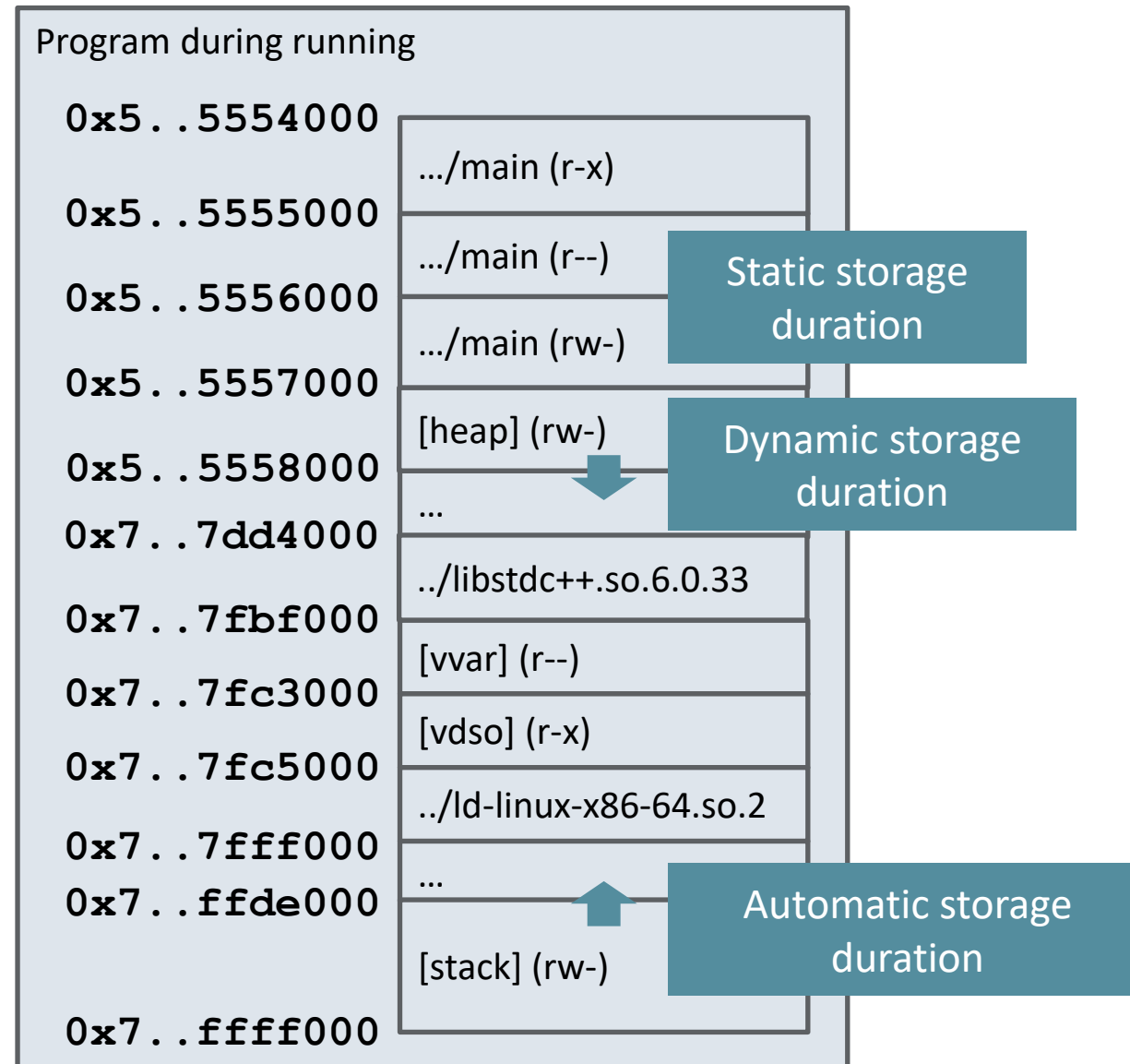
Thread storage duration

Automatic storage
duration

Dynamic storage
duration*

*Note: Special Storage for exceptions

STORAGE DURATION



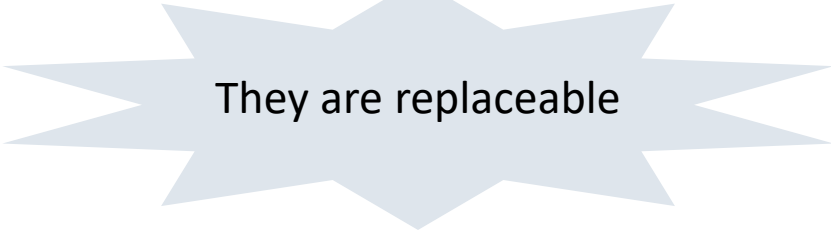
ALLOCATING/DEALLOCATING OBJECTS IN THE HEAP WITH NEW AND DELETE

- New and delete expressions can be used to allocate and deallocate objects on the heap
- How many forms of the new operator are available?

```
void* operator new(std::size_t size);  
void* operator new(std::size_t size, std::align_val_t alignment);  
void* operator new(std::size_t size, const std::nothrow_t&) noexcept;  
void* operator new(std::size_t size, std::align_val_t alignment, const std::nothrow_t&) noexcept;  
void* operator new[](std::size_t size);  
void* operator new[](std::size_t size, std::align_val_t alignment);  
void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;  
void* operator new[](std::size_t size, std::align_val_t alignment, const std::nothrow_t&) noexcept;
```

- How many forms of the delete are available?

```
void operator delete(void* ptr) noexcept;  
void operator delete(void* ptr, std::size_t size) noexcept;  
void operator delete(void* ptr, std::align_val_t alignment) noexcept;  
void operator delete(void* ptr, std::size_t size, std::align_val_t alignment) noexcept;  
void operator delete(void* ptr, const std::nothrow_t&) noexcept;  
void operator delete(void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;  
void operator delete[](void* ptr) noexcept;  
void operator delete[](void* ptr, std::size_t size) noexcept;  
void operator delete[](void* ptr, std::align_val_t alignment) noexcept;  
void operator delete[](void* ptr, std::size_t size, std::align_val_t alignment) noexcept;  
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;  
void operator delete[](void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;
```



They are replaceable

ALLOCATING/DEALLOCATING OBJECTS IN THE HEAP WITH NEW AND DELETE

■ New and delete expression

```
::_opt new new-placement_opt new-type-id new-initializer_opt  
::_opt new new-placement_opt (type-id) new-initializer_opt
```

```
::_opt delete cast-expression  
::_opt delete[] cast-expression
```

■ Parameters of the new operator

```
...new (std::size_t size, std::align_val_t alignment, const std::nothrow_t&)
```



Number of bytes to allocate



Alignment to use



Do not throw when not enough memory

ALLOCATING/DEALLOCATING OBJECTS IN THE HEAP WITH NEW AND DELETE

■ Allocate an integer

```
auto p = new int;  
auto p = new int(2);  
auto p = new int{3};
```

```
auto p = new (int);  
auto p = new (int)(2);  
auto p = new (int){3};
```

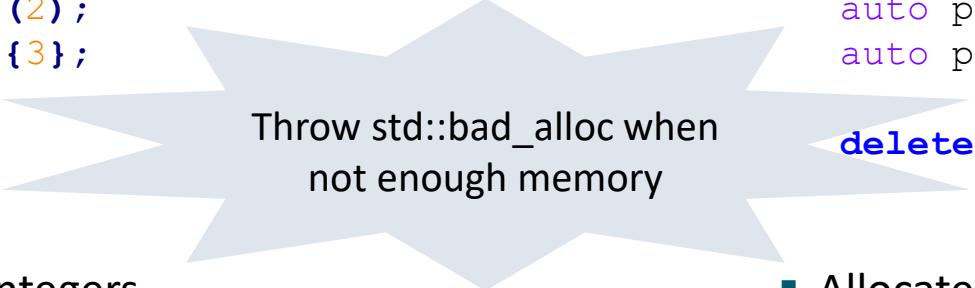
```
delete p;
```

■ Allocate a class

```
auto p = new Foo;  
auto p = new Foo(2);  
auto p = new Foo{3};
```

```
auto p = new (Foo);  
auto p = new (Foo)(2);  
auto p = new (Foo){3};
```

```
delete p;
```



Throw std::bad_alloc when
not enough memory

■ Allocate an array of integers

```
auto p = new int[]{3,4,5};  
auto p = new int[3];  
auto p = new (int[]){3,4,5};  
auto p = new (int[3]);
```

```
delete[] p;
```

■ Allocate an array of classes

```
auto p = new Foo[]{3,4,5};  
auto p = new Foo[3];  
auto p = new (Foo[]){3,4,5};  
auto p = new (Foo[3]);
```

```
delete[] p;
```

SELECTION OF THE NEW/DELETE VERSION

■ Which new version is selected?

```
auto p = new int;  
auto p = new int(2);  
auto p = new int{3};  
  
auto p = new (int);  
auto p = new (int)(2);  
auto p = new (int){3};
```

```
➡ ... new(std::size_t);  
➡ ... new(std::size_t, std::align_val_t);  
... new(std::size_t, const std::nothrow_t&) noexcept;  
... new(std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;
```

■ Which delete version is selected?

```
delete p;
```

```
➡ ... delete(void* ptr) noexcept;  
➡ ... delete(void* ptr, std::size_t) noexcept;  
➡ ... delete(void* ptr, std::align_val_t) noexcept;  
➡ ... delete(void* ptr, std::size_t, std::align_val_t) noexcept;  
... delete(void* ptr, const std::nothrow_t&) noexcept;  
... delete(void* ptr, std::align_val_t, const std::nothrow_t&) noexcept;
```

ALLOCATING WITH NEW WITHOUT THROWING

- Allocating with new without throwing

`::opt new new-placementopt new-type-id new-initializeropt`



```
auto p = new (std::nothrow) int;  
auto p = new (std::nothrow) int{3};  
  
auto p = new (std::nothrow) int[]{3,4,5};  
auto p = new (std::nothrow) int[3];
```

- Which new version is selected?

```
... new/new[] (std::size_t);  
... new/new[] (std::size_t, std::align_val_t);  
➡ ... new/new[] (std::size_t, const std::nothrow_t&) noexcept;  
➡ ... new/new[] (std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;
```

ALLOCATING WITH NEW WITHOUT THROWING

- Deallocating with delete after allocating with new without throwing

```
delete p;  
delete[] p;
```

- Which delete version is selected?

```
➡ ... delete(void* ptr) noexcept;  
➡ ... delete(void* ptr, std::size_t) noexcept;  
➡ ... delete(void* ptr, std::align_val_t) noexcept;  
➡ ... delete(void* ptr, std::size_t, std::align_val_t) noexcept;  
... delete(void* ptr, const std::nothrow_t&) noexcept;  
... delete(void* ptr, std::align_val_t, const std::nothrow_t&) noexcept;
```

- No placement new version of delete

```
::_opt delete cast-expression  
::_opt delete[] cast-expression
```

ALLOCATING WITH EXTENDED ALIGNMENT

- Allocating with extended alignment

```
auto p = new (std::align_val_t{64}) int{3};
```

```
auto p = new (std::align_val_t{64}) int[]{3,4,5};
```

- Can we?

GCC 

Clang 

MSVC 

error C2956:

usual deallocation function 'void operator delete(void *,std::align_val_t) noexcept'
would be chosen as placement deallocation function.

predefined C++ types (compiler internal)(37):


note: see declaration of 'operator delete'

ALLOCATING WITH EXTENDED ALIGNMENT



- Allocating with extended alignment

```
auto p = new (std::align_val_t{64}) int{3};  
  
delete p;
```

- Can we?



```
... new(std::size_t);  
... new(std::size_t, std::align_val_t);  
... new(std::size_t, const std::nothrow_t&) noexcept;  
... new(std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;
```

```
... delete(void* ptr) noexcept;  
... delete(void* ptr, std::size_t) noexcept;  
... delete(void* ptr, std::align_val_t) noexcept;  
... delete(void* ptr, std::size_t, std::align_val_t) noexcept;  
... delete(void* ptr, const std::nothrow_t&) noexcept;  
... delete(void* ptr, std::align_val_t, const std::nothrow_t&) noexcept;
```

ALLOCATING WITH EXTENDED AND NOTHROW

- Allocating with extended alignment

```
auto p = new (std::align_val_t{64}, std::nothrow) int{3};
```

```
auto p = new (std::align_val_t{64}, std::nothrow) int[]{3,4,5};
```

- Can we?

GCC



Clang



MSVC



- Still same issue with delete

```
➡ ... delete(void* ptr) noexcept;  
➡ ... delete(void* ptr, std::size_t) noexcept;  
... delete(void* ptr, std::align_val_t) noexcept;  
... delete(void* ptr, std::size_t, std::align_val_t) noexcept;  
... delete(void* ptr, const std::nothrow_t&) noexcept;  
... delete(void* ptr, std::align_val_t, const std::nothrow_t&) noexcept;
```


ALLOCATING WITH EXTENDED ALIGNMENT (THE RIGHT WAY)

- Make the alignment requirement part of the type

```
struct alignas(64) Foo          auto p = new Foo{3};
{
    int bar{42};                delete p;
};
```

→

```
... new(std::size_t);
... new(std::size_t, std::align_val_t);
... new(std::size_t, const std::nothrow_t&) noexcept;
... new(std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;
```

→

```
... delete(void* ptr) noexcept;
... delete(void* ptr, std::size_t) noexcept;
→ ... delete(void* ptr, std::align_val_t) noexcept;
→ ... delete(void* ptr, std::size_t, std::align_val_t) noexcept;
... delete(void* ptr, const std::nothrow_t&) noexcept;
... delete(void* ptr, std::align_val_t, const std::nothrow_t&) noexcept;
```



ALLOCATING WITH EXTENDED ALIGNMENT (THE RIGHT WAY)

- Make the alignment requirement part of the type



```
struct alignas(64) Foo
{
    int bar{42};
};

auto p = new (std::nothrow) Foo{3};

delete p;
```



```
... new(std::size_t);
... new(std::size_t, std::align_val_t);
... new(std::size_t, const std::nothrow_t&) noexcept;
... new(std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;
```



```
... delete(void* ptr) noexcept;
... delete(void* ptr, std::size_t) noexcept;
... delete(void* ptr, std::align_val_t) noexcept;
... delete(void* ptr, std::size_t, std::align_val_t) noexcept;
... delete(void* ptr, const std::nothrow_t&) noexcept;
... delete(void* ptr, std::align_val_t, const std::nothrow_t&) noexcept;
```

DYNAMIC MEMORY ALLOCATION USING C++ STD LIBRARY

■ Using the C API

```
void *malloc( size_t size );  
void* calloc( size_t num, size_t size );  
void *realloc( void *ptr, size_t new_size );  
void *aligned_alloc( size_t alignment, size_t size );  
  
void free( void *ptr );
```

■ Using smart pointers

```
template< class T, class... Args >  
constexpr unique_ptr<T> make_unique( Args&&... args );  
shared_ptr<T> make_shared( Args&&... args );  
  
template< class T >  
constexpr unique_ptr<T> make_unique_for_overwrite();  
shared_ptr<T> make_shared_for_overwrite();
```

DYNAMIC MEMORY ALLOCATION USING C++ STD LIBRARY

- Using indirect and polymorphic (planned for C++26)
 - Dynamic allocated objects with value semantics
 - <https://eel.is/c++draft/indirect>
 - <https://eel.is/c++draft/polymorphic>
 - Reference implementation
 - https://github.com/jbcoo/value_types

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
auto pointer = std::make_unique<int>(2);
```

Dynamic allocation: Yes

Number of allocations: 1

Allocated bytes: 4

```
auto pointer = std::make_shared<int>(2);
```

Dynamic allocation: Yes

Number of allocations: 1

Allocated bytes: 24

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
const std::string small_string = "my_small_string";
```

Dynamic allocation: No

```
const std::string big_string = "my_big_string_that_does_not_fit_for_small_string_optimization";
```

Dynamic allocation: Yes

Number of allocations: 1

Allocated bytes: 62

```
const std::string_view big_string = "my_big_string_that_does_not_fit_for_small_string_optimization";
```

Dynamic allocation: No

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
const std::array array_of_ints = {4, 3, 5, 6, 7, 8, 9, 10};
```

Dynamic allocation: No

```
const std::vector<std::string> default_constructed_vector{};
```

Dynamic allocation: No

```
const std::vector<int> vector_of_ints = {4, 3, 5, 6, 7, 8, 9, 10};
```

Dynamic allocation: Yes

Number of allocations: 1

Allocated bytes: 32

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
void answer_to_live_or_abort(int value)
{
    if (value != 42)
    {
        std::terminate();
    }
}
```

```
std::function<void(int)> f = answer_to_live_or_abort;
```

Dynamic allocation: No

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
struct universe_t
{
    void answer_or_abort(int value)
    {
        ::answer_to_live_or_abort(value);
    }
};

universe_t universe;
std::function<void(int)> f = std::bind(&universe_t::answer_or_abort, &universe, _1);
```

Dynamic allocation: Yes

Number of allocations: 1

Allocated bytes: 24

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
std::thread thread{answer_to_live_or_abort, 42};  
thread.join();
```

Dynamic allocation: Yes
Number of allocations: 1
Allocated bytes: 24

```
std::jthread thread{answer_to_live_or_abort, 42};
```

Dynamic allocation: Yes
Number of allocations: 2
Allocated bytes: 48

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
std::promise<void> promise;
```

Dynamic allocation: Yes
Number of allocations: 2
Allocated bytes: 64

```
auto future = promise.get_future();
```

Dynamic allocation: No

```
promise.set_value();
```

Dynamic allocation: No

```
std::future<int> future;
```

Dynamic allocation: No

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
std::any a = 42;
```

Dynamic allocation: No

```
const std::string small_string = "my_small_string";  
std::any a = small_string;
```

Dynamic allocation: Yes

Number of allocations: 1

Allocated bytes: 32

IMPLICIT DYNAMIC MEMORY ALLOCATION IN THE C++ STD LIBRARY



- Does it dynamically allocate memory? And how much?

```
std::stacktrace::current();
```

Dynamic allocation: Yes

Number of allocations: 1

Allocated bytes: 512

HOW TO KNOW IF SOMETHING DYNAMICALLY ALLOCATES?

- Class satisfies `AllocatorAwareContainer`

`std::vector` (for `T` other than `bool`) meets the requirements of *Container*, *AllocatorAwareContainer* (since C++11), *SequenceContainer*, *ContiguousContainer* (since C++17) and *ReversibleContainer*.

- One template parameter is an Allocator

```
template< class BidirIt, class Alloc, class CharT, class Traits >
bool regex_match( BidirIt first, BidirIt last,
                  std::match_results<BidirIt, Alloc>& m,
                  const std::basic_regex<CharT, Traits>& e,
                  std::regex_constants::match_flag_type flags =
                    std::regex_constants::match_default );
```

(1) (since C++11)

Snapshots from <https://en.cppreference.com/w/cpp>

HOW TO KNOW IF SOMETHING DYNAMICALLY ALLOCATES?

- Throws `bad_alloc` if memory could not be allocated

Exceptions

1) `std::bad_alloc` if memory could not be allocated for the stop-state.

- You know that the only way to implement it is dynamically allocating memory
- You provide a replacement for the global operator `new` and `delete` and you see if it dynamically allocates

Snapshots from <https://en.cppreference.com/w/cpp>

CUSTOMIZING DYNAMIC MEMORY ALLOCATION

- Global new and delete
 - One new/delete to rule them all.
 - Multiple overloads that should follow same semantics. If you define one, define them all.
 - It replaces the default new/delete. Once replaced you have no longer access to the default ones.
 - If you do not follow the allocation/deallocation semantics required by the standard, the behavior is undefined.

```
void *operator new(std::size_t count)
{
    if (g_tracking)
    {
        g_allocated_bytes += count;
        g_number_of_allocations++;
    }
    void *pointer_to_storage = malloc(count);
    if (!pointer_to_storage)
    {
        throw std::bad_alloc();
    }
    return pointer_to_storage;
}
```


CUSTOMIZING DYNAMIC MEMORY ALLOCATION

- Class new and delete
 - Custom new/delete per type.
 - All instance of the same type, will use the same allocator.
 - Only called when the object is allocated on the heap (new, make_unique, make_shared, etc.).
 - It does not propagate to the members. If a member requires dynamic memory allocation, it will not use the class allocator.

```
class Foo
{
public:
    static void *operator new(std::size_t size)
    {
        if (free_slot + size >= storage.size())
        {
            throw std::bad_alloc();
        }
        void *pointer = &storage[free_slot];
        free_slot += size;
        return pointer;
    }
    static void operator delete(void *) {}

private:
    int bar;
};
```



CUSTOMIZING DYNAMIC MEMORY ALLOCATION

- Custom allocator
 - Multiple instances for the same type allocator.
 - It needs to follow the Allocator requirements.
 - Several traits are optional.
 - Default traits are provided via `allocator_traits`.
 - When copy/move a type with a custom allocator, you can choose at allocator level if allocator gets propagated.

```
template <class T>
class MallocAllocator
{
public:
    using value_type = T;

    [[nodiscard]] T *allocate(std::size_t size)
    {
        auto p = std::malloc(sizeof(T) * size);
        return static_cast<T *>(p);
    }
    void deallocate(T *pointer, std::size_t size)
    {
        std::free(pointer);
    }
};
```



CUSTOMIZING DYNAMIC MEMORY ALLOCATION

- Polymorphic allocator (pmr) (since C++17)
 - Allocation behavior can be configured at runtime.
 - Functionality is divided between `polymorphic_allocator` and `memory_resource`.
 - Standard library provides multiple memory resources.
 - Including null memory resource and new and delete memory resource
 - Polymorphic allocators do not propagate on move/copy assignment of the container.
 - Swapping two containers that allocators do not compare equal is undefined behavior.
 - It does propagate to the members that use polymorphic allocators.

```
std::array<std::byte, 50> storage;  
  
std::pmr::monotonic_buffer_resource resource{  
    storage.data(), storage.size(),  
    std::pmr::null_memory_resource()  
};  
  
std::pmr::vector<int> vector{  
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},  
    &resource  
};
```



IN SUMMARY, DYNAMIC MEMORY ALLOCATION IS HARD

- The end goal is to achieve functional safety
- The C++ mental model is one part, but the implementation plays a big role
 - OS, std library, compiler, etc.
- When using allocators, still remember what you wanted to solve
 - Memory leaks
 - Out of memory
 - Allocation/deallocation mismatch
 - Memory fragmentation
 - None deterministic runtime
 - Use after free

BMW
GROUP



DYNAMIC MEMORY ALLOCATION CHALLENGES IN SAFETY CRITICAL SYSTEMS

XAVIER BONAVENTURA

BMW Group

FURTHER READING

- PE Format (Windows executables)
 - <https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail>
 - <https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/march/inside-windows-an-in-depth-look-into-the-win32-portable-executable-file-format-part-2>