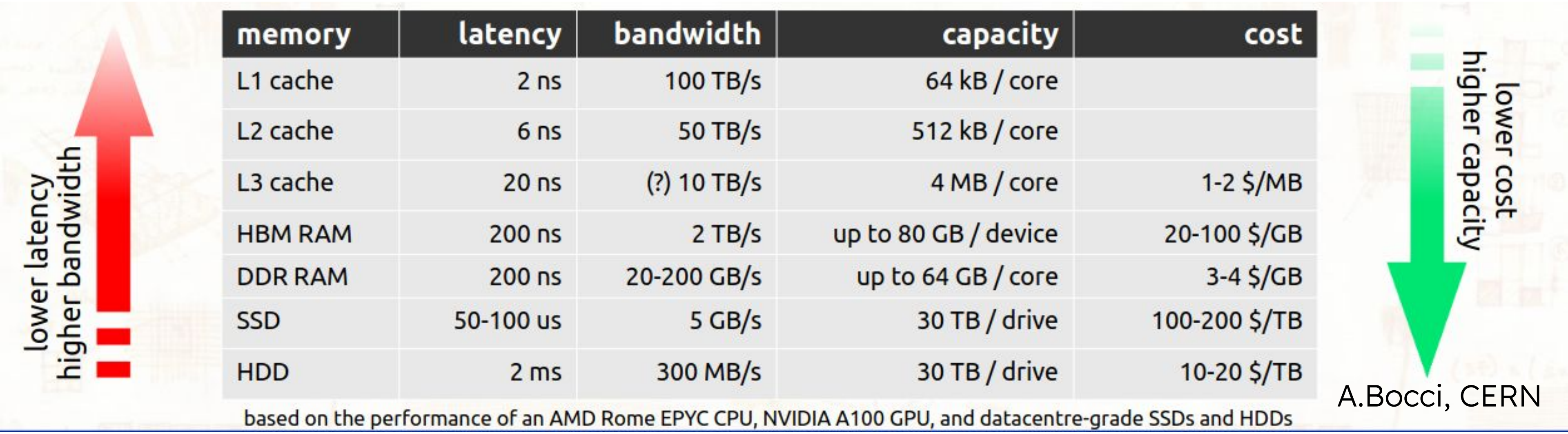
Efficient Memory Management

Simone Balducci, Wahid Redjeb
CERN

What is memory?

- In general, memory refers to the storage a program uses to write and read data
  - RAM
  - GPU memory
  - HBM
  - Disk space (HDD, SDD)
  - Caches

Different types of memory - Latency


lower latency
higher bandwidth
lower cost
higher capacity

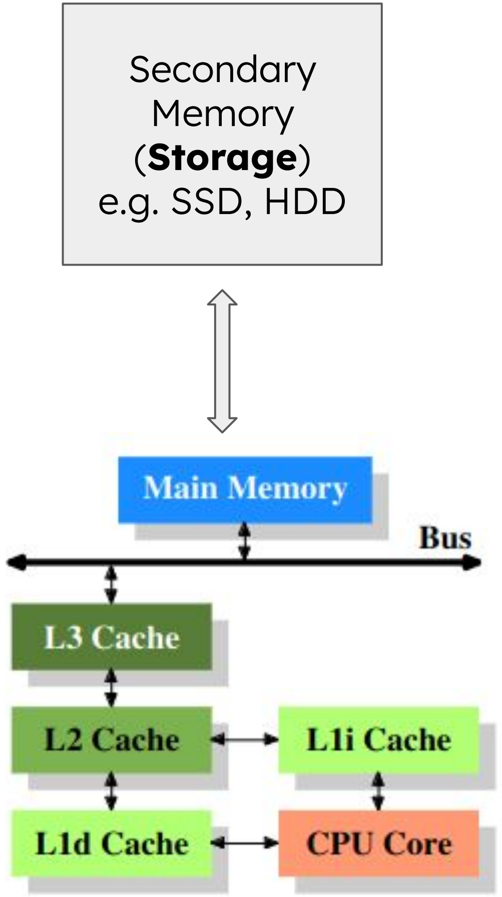
memory	latency	bandwidth	capacity	cost
L1 cache	2 ns	100 TB/s	64 kB / core	
L2 cache	6 ns	50 TB/s	512 kB / core	
L3 cache	20 ns	(?) 10 TB/s	4 MB / core	1-2 $/MB
HBM RAM	200 ns	2 TB/s	up to 80 GB / device	20-100 $/GB
DDR RAM	200 ns	20-200 GB/s	up to 64 GB / core	3-4 $/GB
SSD	50-100 us	5 GB/s	30 TB / drive	100-200 $/TB
HDD	2 ms	300 MB/s	30 TB / drive	10-20 $/TB

based on the performance of an AMD Rome EPYC CPU, NVIDIA A100 GPU, and datacentre-grade SSDs and HDDs

A.Bocci, CERN

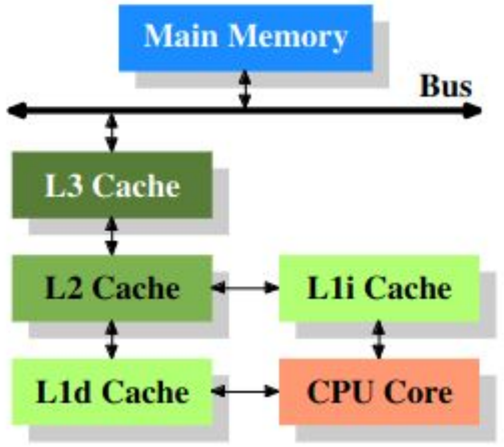
Different types of memory

- Secondary Memory (SSD, HDD) [variable storage]
- Main Memory (RAM) [usually tens of GBs]
- 3 levels of cache
  - Small [32/64kB] separate L1 (I+D) caches for each core.
  - Medium [256kB - 6MB] combined L2 cache, perhaps shared among some cores.
  - Large [4 - 20MB] combined L3 cache shared between all cores



Caches

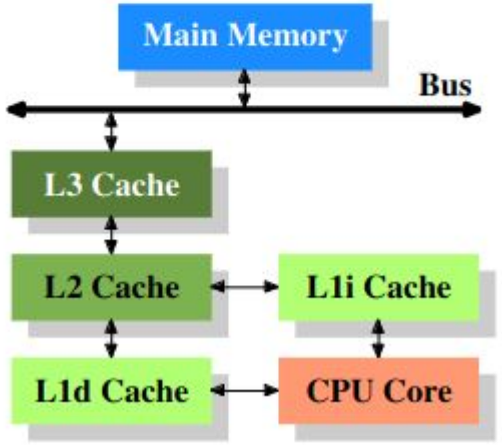
- CPU looks for data in L1 -> L2 -> L3 -> RAM
- Data area loaded in cache in unit of **cache lines**
    - **Usually 64 bytes, but depends on architecture**
- Core cache controllers + replacement policy; hardware prefetchers may prefetch based on **access patterns.**
    - **Cache locality**
    - Cache lines might be promoted or demoted depending on these patterns



Caches

The functioning of caches is based on two principles:

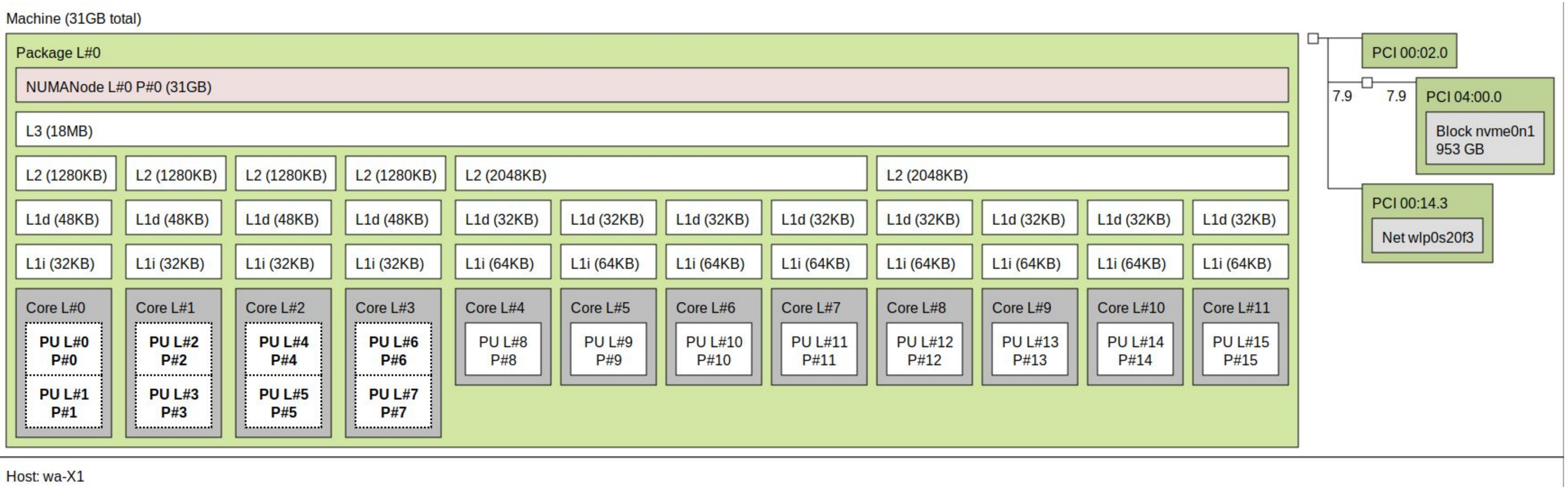
- **time locality**
  - If a program accesses one memory address, there is a good chance that it will access the same address again after a short amount of time.
    - E.g loops (variable `sum` continuously updated)
- **spatial locality**
  - If a program accesses one memory address, there is a good chance that it will also access other nearby addresses.



Different types of memory

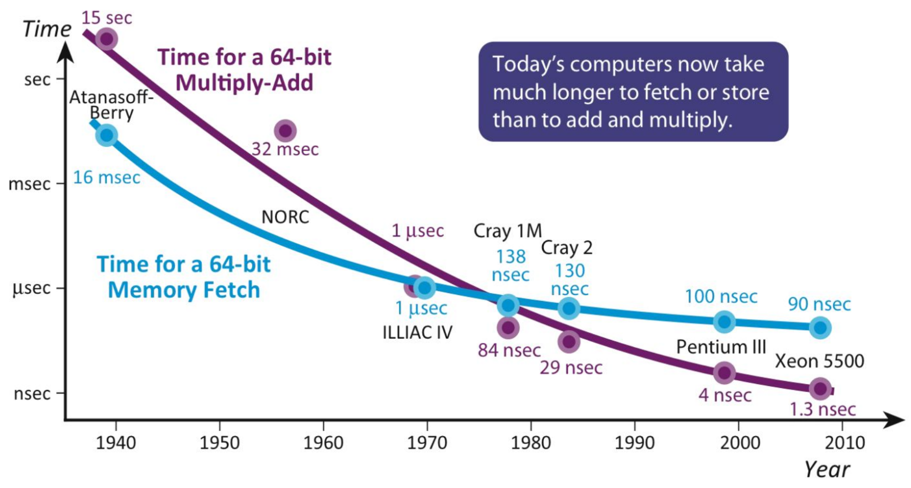
Have a look at your system

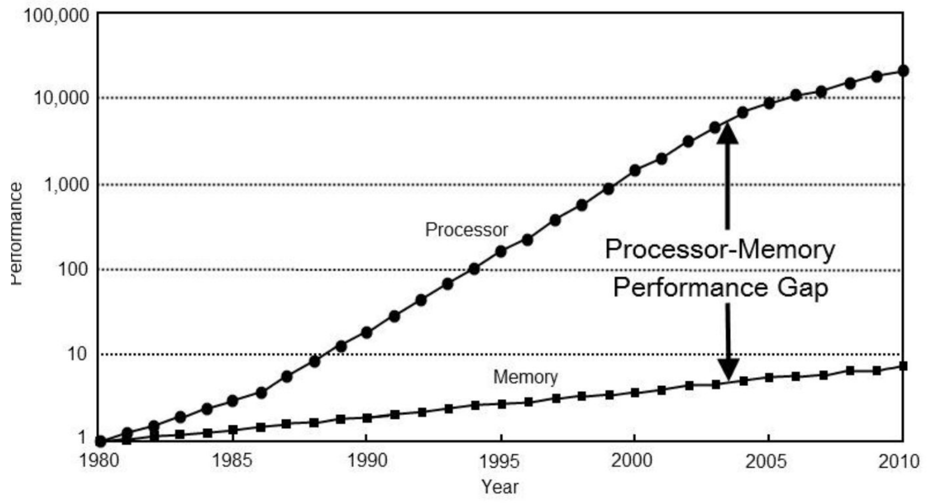
- `lscpu`
- `lstopo`



Why are we interested in memory?

- Most of the memory is *very* **slow** compared to CPU operations





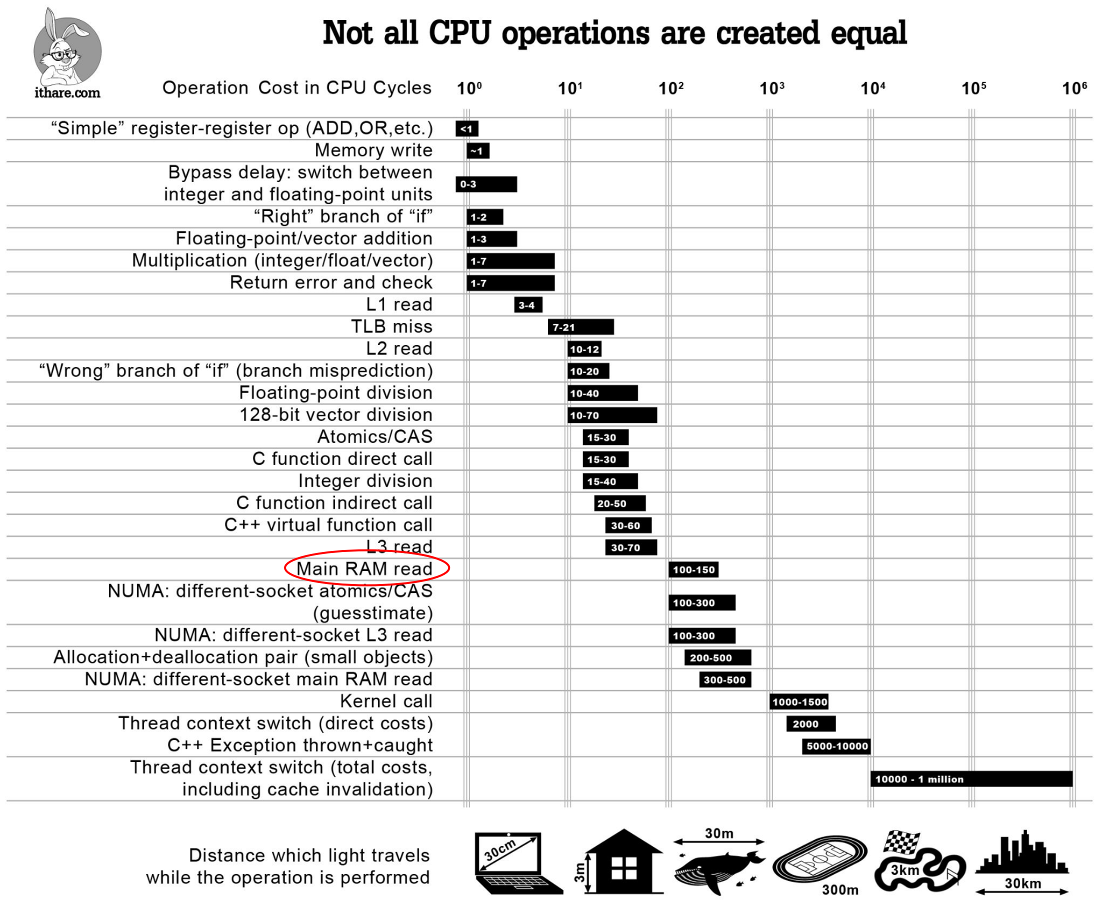
Why are we interested in memory?

Everything here is better than reading from main memory

When writing efficient code, the most important thing to address is memory

But there's no general rule, the best solution to adopt depends on your **data**

- **Know your data**



Not all CPU operations are created equal

Operation	Cost in CPU Cycles
"Simple" register-register op (ADD,OR,etc.)	<1
Memory write	~1
Bypass delay: switch between integer and floating-point units	0-3
"Right" branch of "if"	1-2
Floating-point/vector addition	1-3
Multiplication (integer/float/vector)	1-7
Return error and check	1-7
L1 read	3-4
TLB miss	7-21
L2 read	10-12
"Wrong" branch of "if" (branch misprediction)	10-20
Floating-point division	10-40
128-bit vector division	10-70
Atomics/CAS	15-30
C function direct call	15-30
Integer division	15-40
C function indirect call	20-50
C++ virtual function call	30-60
L3 read	30-70
Main RAM read	100-150
NUMA: different-socket atomics/CAS (guesstimate)	100-300
NUMA: different-socket L3 read	100-300
Allocation+deallocation pair (small objects)	200-500
NUMA: different-socket main RAM read	300-500
Kernel call	1000-1500
Thread context switch (direct costs)	2000
C++ Exception thrown+caught	5000-10000
Thread context switch (total costs, including cache invalidation)	10000 - 1 million

ithare.com
10⁰
10¹
10²
10³
10⁴
10⁵
10⁶
Distance which light travels while the operation is performed
30cm
3m
30m
300m
3km
30km

Data oriented design

- **Data temporal locality**
  - Exploit data that has just been read or written to memory
  - Exploit data that is “hot” in the processor cache
- **Data spatial locality**
  - Fully exploit cache line: work on adjacent data!
  - Avoid pointers chasing if possible
    - Pointers to pointers to pointers …
  - AoS → SoA
- **Hide memory latency**
  - Prefetch data in advance while working on previous data
  - Keep the processor busy while more data is fetched
  - Common strategy on GPU
- **If possible avoid dynamic allocations**
  - Remember: understand your data
  - Custom allocators
- **Avoid high level abstraction**

Size of Data Types

Size of a type corresponds to the number of bytes needed to store an object of that type

- Use `sizeof()` operator to get the size of your type
  - Try it yourself with some common types
  - `char`, `int`, `float`, `double`, `int *`, `std::vector<double>`, `std::vector<int>`
- Define your own Class / Struct with different members and get the size of your class
  - Try to change the order of the members
  - Try to add a bool to your members

Size of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

Size of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

$\xrightarrow{\text{13 bytes}}$

```
sizeof(MyStruct) -> 24
```

Size of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

$\xrightarrow{\text{13 bytes}}$

Alignment of data types

- To have a more efficient memory access from the CPU data types are ***aligned***
- *Alignment is an integer value representing the number of bytes between successive addresses at which objects of this type can be allocated.*
    - e.g. if a type has an alignment of 4, it can be allocated only every 4 bytes: 0x…00, 0x…04, 0x…08, 0x…0c, 0x…10, …

- The valid alignment values are **non-negative integral powers of two.**
- The operator **`alignof`**`()` gives you the alignment of a type
- You can request stricter alignment using **`alignas`**`()` specifier
- The alignment of any class object is given by the largest of the alignment of its members

Alignment of data types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a																				

Alignment of Data Types

```cpp
struct MyStruct {

    int a; //4 bytes

    double b; //8 bytes

    bool c; // 1 byte

};
```

```cpp
alignof(int) -> 4

alignof(double) -> 8
```

```cpp
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a					b	b	b	b	b	b	b	b								

Alignment of Data Types

```
struct MyStruct {

    int a; //4 bytes

    double b; //8 bytes

    bool c; // 1 byte

};
```

```
alignof(int) -> 4

alignof(double) -> 8
```

```
alignof(bool) -> 1
```

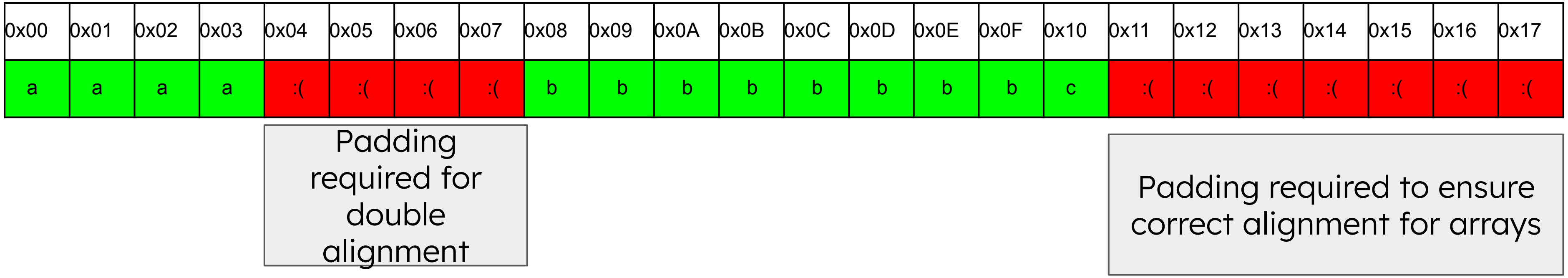
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a					b	b	b	b	b	b	b	b	c							

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```



0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a	:(	:(	:(	:(	b	b	b	b	b	b	b	b	c	:(	:(	:(	:(	:(	:(	:(

Padding required for double alignment

Padding required to ensure correct alignment for arrays

Alignment of Data Types

```
struct MyStruct {

    double b; //8 bytes

    int a; //4 bytes

    bool c; // 1 byte

};
```

```
alignof(double) -> 8

alignof(int) -> 4
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
b	b	b	b	b	b	b	b	a	a	a	a	c	:(	:(	:(

`sizeof(MyStruct) = 16`

Padding required to ensure correct alignment for arrays

Alignment of Data Types - Optimize memory design

```cpp
struct MyStruct {
    double b; //8 bytes
    int a; //4 bytes
    bool c; // 1 byte
};
```

Put data members in decreasing size order
Group data members based on their size and alignment

- Dedicate some time to understand if you are introducing padding and if you can avoid it

Group data members based on their usage

- Better to have data members that are used together within a single cache line!
  - Cache line usually are 64 bytes.

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
b	b	b	b	b	b	b	b	a	a	a	a	c	:(	:(	:(

`sizeof(MyStruct) = 16`

Padding required to ensure correct alignment for arrays

Exercise

- Create a Class or struct for a Particle with the following members
  - 1 `const std::string` to hold the particle's name;
  - 3 `double`s for the x, y, z velocities
  - 3 `bool`s to mark if there has been a collision along the x, y z directions
  - 1 `float` for the mass
  - 1 `float` for the energy
  - 3 `double`s for the px, py, pz coordinates
  - 1 `const int` for the particle's id
- What is the best order for your members?

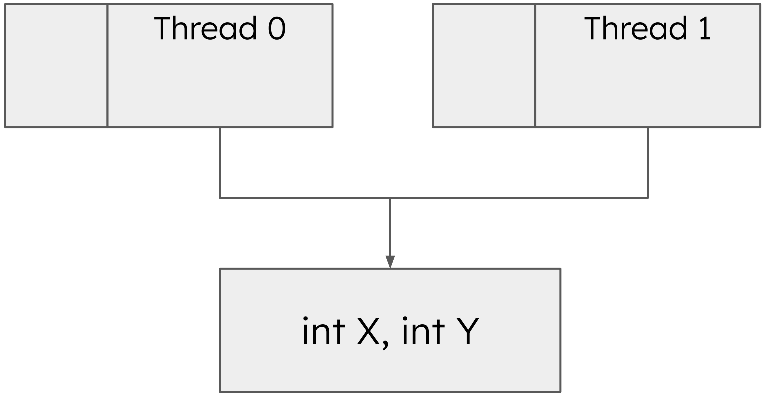
False sharing

- False sharing is a performance-degrading usage pattern that happens in multi-threaded application
- If two cores are accessing different elements that are in the same cache line
  - Each core has its own copy of the cache line
- Core0 reads the value X from the cache line
  - It marks the cache line as **exclusive**
- Core1 reads the value Y from the its copy of the same cache line
  - Both core mark the cache line as **shared**
- Core0 decides to write in address space of X
  - Marks its cache line as **updated**
    - It has to send a message to Core1 saying it has updated the cache line
- Core1 marks its cache line as **invalid**
  - Has to re-read the cache line from main memory
- Core0 has to immediately return the result back to main memory

This process for keeping caches in coherence can be extremely expensive!

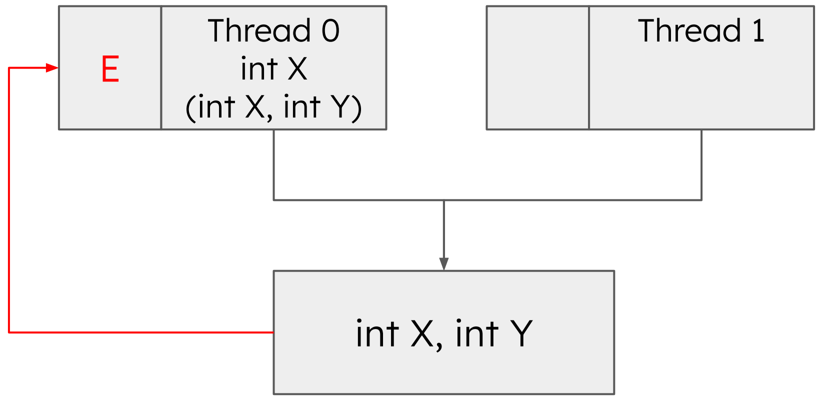
False sharing

Initial State



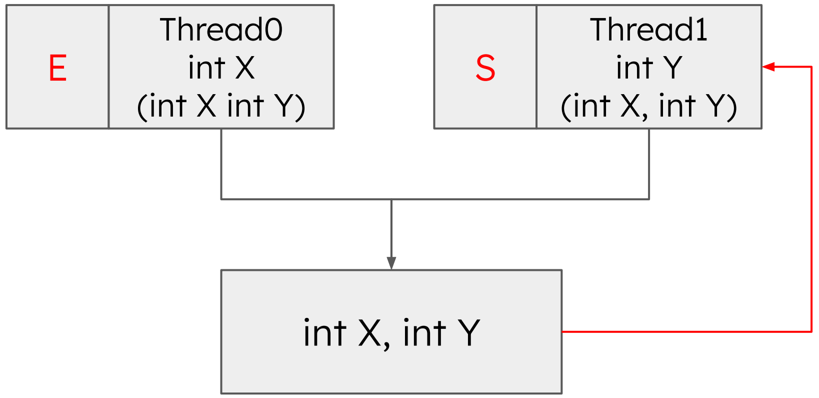
False sharing

Thread 0 wants to read X
→ Loads cache line (X, Y)
→ Marks cache as Exclusive



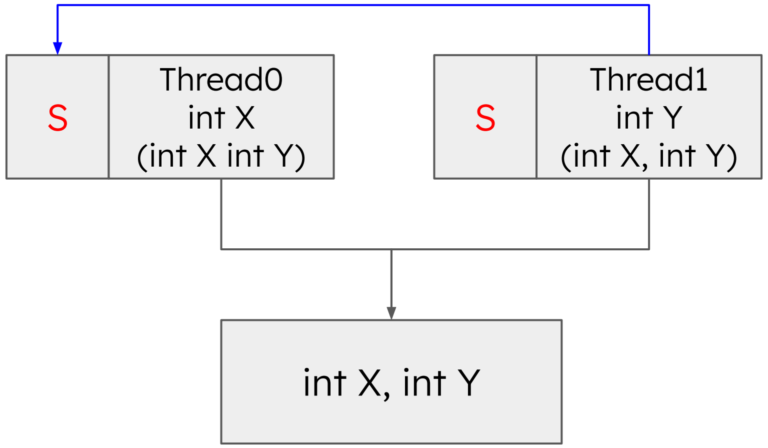
False sharing

Thread 1 wants to read Y
→ Loads cache line (X, Y)



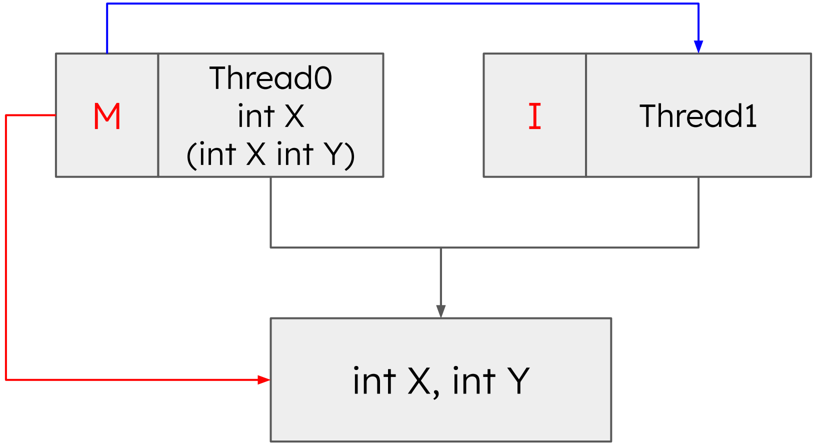
False sharing

Thread 1 wants to read Y
→ Loads cache line (X, Y)
→ Marks cache as shared and tells thread 0 to mark cache as shared as well



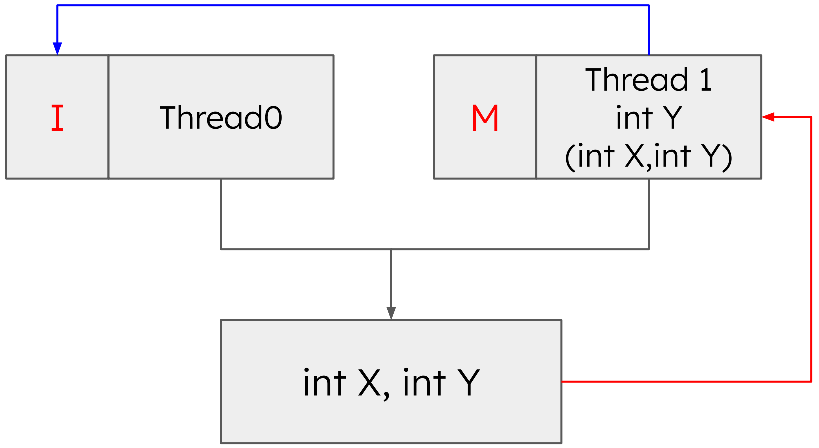
False sharing

Thread 0 does some operations and then wants to update X
→ Flags the cache line as modified and notify thread 1
→ Thread 0 update the cache line → writes back (coherence write-back)
-> Thread 1 now has to invalidate its cache and throw it away



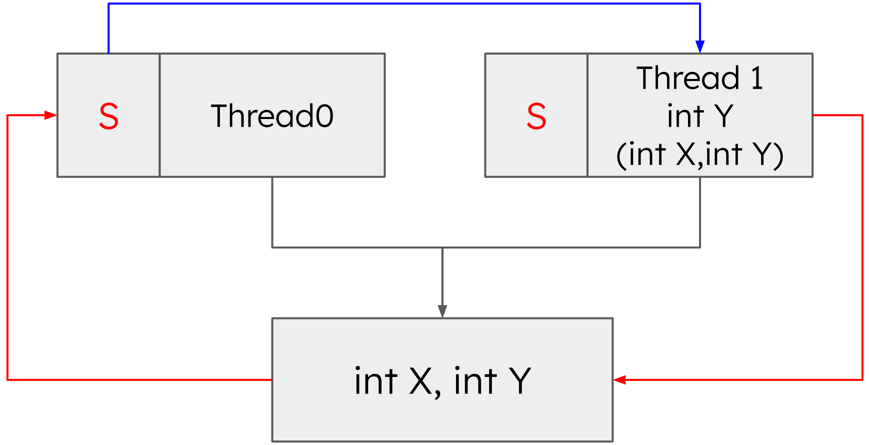
False sharing

Thread 1 now wants to update Y
→ Has to reload the cache
→ Flags the cache line as modified and notify thread 0
→ Thread 0 now has to invalidate its cache and throw it away



False sharing

Thread 0 now wants to update again Y
→ Has to reload the cache
→ But thread 1 needs to update the cache line (coherence write-back)
→ Flag cache as shared



False Sharing

- False sharing is a performance-degrading usage pattern that happens in multi-threaded application
  - Triggers mechanism of cache coherency (MESI protocol)
    - Caches are continuously getting evicted → low cache usage
      - High cache misses

Small exercise: hands-on/memory/false_sharing/false_sharing.cpp

```
g++  false_sharing.cc -pthread
```

How to avoid false sharing?

- If you can, use “local” data
  - Local to each thread, if you need to store some results, store it in a local variable and then gather the results of the multiple threads at the end
  - E.g.: sum of vector entries
    - You split the vector in different blocks and then you assign each block to a different thread
      - Each thread will perform the summation on its own block
      - Don’t update an entry of the block with the partial sum, store the partial sum in a local variable —> don’t touch the cached data!
    - This will get much more clear with the lectures on parallelism
- Align your data to the cache line size
  - Such that each thread will load a different cache line to avoid interference between the two threads
  - Add padding to your data structure or use `alignas(CACHE_SIZE)`

Querying the cache size

You can ask for the cache size by using

```
#include <new>
```

```
auto cls_constructive = std::hardware_constructive_interference_size;
```

Use constructive to create “true sharing” —> keep stuff you are going to use together in the same cache line

```
auto cls_destructive = std::hardware_destructive_interference_size;
```

Use destructive to avoid false sharing

They are the same on x86 architecture, but different on ARM:

- Constructive → 256 bytes
- Destructive → 64 bytes

Memory operations - Constructing objects

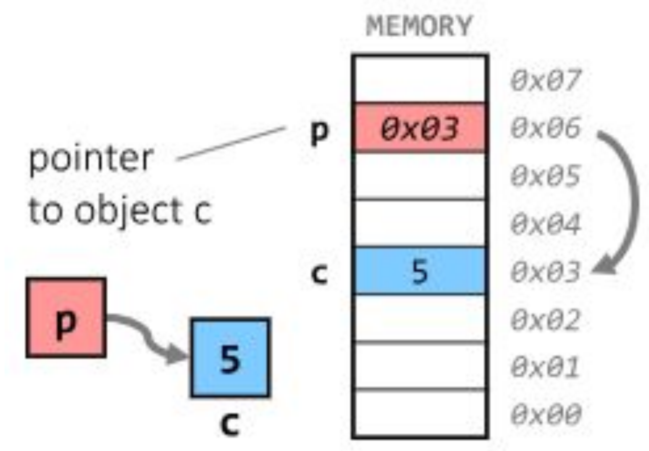
- `T* new T(args…)`
  - Allocates and creates object `T`
- `T* new T[N]{args…}`
  - Constructs `N` object of type `T` using its constructor `T::T(args…)`
    - The object is created in the allocated memory at `ptr`
- `T* new(std::align_val_t(alignment)) T{args…}`
  - Constructs an object of type `T` using its constructor `T::T(args…)`
    - Memory is aligned to alignment bytes
    - The object is created in the allocated memory at `ptr`

Remember: After allocating memory with **`new`**, you must always deallocate it with **`delete`**

Recap: pointers and references

- A **pointer** is an object containing the address to a memory region
    - It can be NULL, i.e. not pointing to anything
    - The address that it points to can be changed
    - The data in the pointed memory can be accessed with the * operator
- A **reference** is a mapping to another object
    - It is bound to the object that it's constructed from
        - It cannot be null and cannot be changed to reference another object
    - Modifying the reference modifies the referenced object
    - The referenced is used like any object
    - Creating a reference doesn't copy any memory
- So, a pointer maps to a memory region, a reference maps to a pre-existing object

```
int* p = new int(5);
std::cout << p << std::endl;     // 0x1221b2b0
std::cout << *p << std::endl;    // 5
int x;
int& rx = x;
std::cout << x << std::endl;     // 5
```



Memory Leak

As we noticed, both malloc and new requires the programmer to free/destroy the object manually to avoid memory leaks.

Memory Leak: *Failure to release unreachable memory, which can no longer be allocated again by any process during execution of the allocating process.* A memory leak occurs when a program allocates memory on the heap but it fails to deallocate the memory when not needed, losing the reference to the allocated memory (unreachable). Results in an increasing memory usage that slows down the program.

Detecting memory leaks

- Detecting memory leaks can be tricky
- Luckily there are tools meant to help with this task
- To mention two:
  - Address sanitizer
  - Valgrind

AddressSanitizer

There are tools for looking at your memory management and for debugging memory problems:

- AddressSanitizer
  - Instrument the program at compile time
    - `g++ program.cc -fsanitize=address`
      - Enable AddressSanitizer → usually is enough
    - Much more options can be enabled https://github.com/google/sanitizers/wiki/addresssanitizerflags
  - Much faster and usually more precise
    - It has all the symbols available at compilation time

Valgrind

There are tools for looking at your memory management and for debugging memory problems:

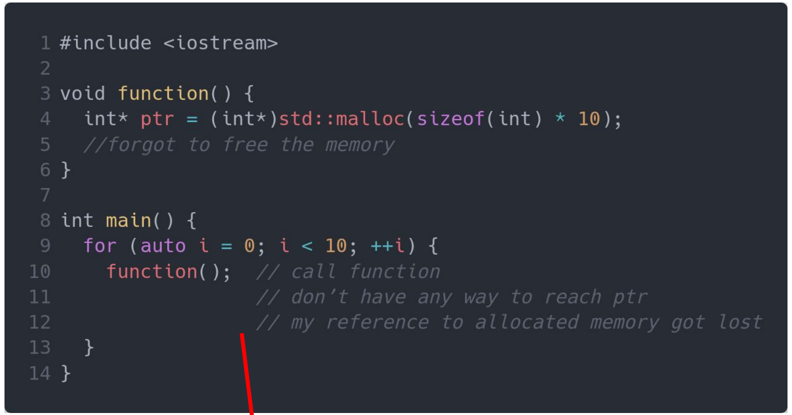
- Valgrind
  - Instrument the binary → runtime
  - `valgrind -tool=memcheck ./a.out`
    - Valgrind is a tool suite, so you can enable different other tools
      - Cachegrind → performs cache analysis
      - Massif → performs heap analysis
      - Helgrind → thread debugging tools → race conditions
  - Much slower

Which one to use?

- I would say AddressSanitizer → faster
- But you should use both, especially in case you want to look at performance improvements

Test them in
https://github.com/infn-esc/esc25/tree/main/hands-on/memory/asan

Memory Leak - Example - ASan



```cpp
#include <iostream>

void function() {
  int* ptr = (int*)std::malloc(sizeof(int) * 10);
  //forgot to free the memory
}

int main() {
  for (auto i = 0; i < 10; ++i) {
    function();  // call function
                 // don't have any way to reach ptr
                 // my reference to allocated memory got lost
  }
}
```

- The memory is not stored in a long-lived variable
- The pointer object goes out-of-scope at the end *function*
- The memory is **never deallocated**

```
g++ memory_leak.cc -fsanitize=address
```

```
./a.out
```

```
=================================================================
==19662==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 400 byte(s) in 10 object(s) allocated from:
        #0 0x7389410b4887 in   interceptor malloc
../../../../src/libsanitizer/asan/asan malloc linux.cpp:145
        #1 0x5cd7bc97325e in function() (/home/wa/Documents/Wahid/Bertinoro/a.out+0x125e)
        #2 0x5cd7bc97327f in main (/home/wa/Documents/Wahid/Bertinoro/a.out+0x127f)
        #3 0x738940829d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: 400 byte(s) leaked in 10 allocation(s).
```

Memory Leak - Example - Valgrind

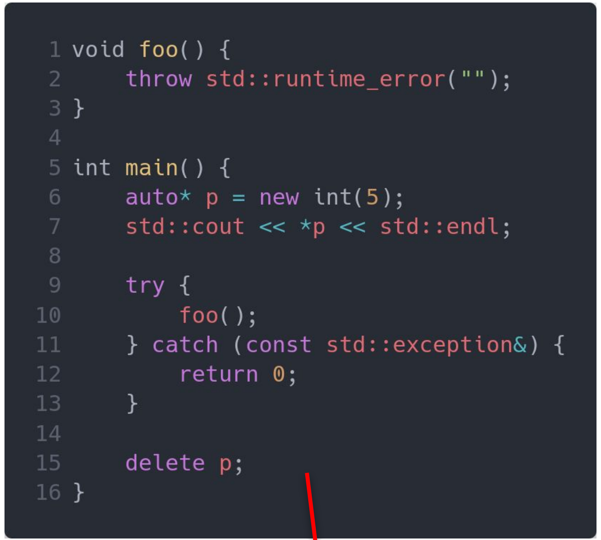
```cpp
#include <iostream>

void function() {
  int* ptr = (int*)std::malloc(sizeof(int) * 10);
  //forgot to free the memory
}

int main() {
  for (auto i = 0; i < 10; ++i) {
    function();  // call function
                 // don’t have any way to reach ptr
                 // my reference to allocated memory got lost
  }
}
```

- The memory is not stored in a long-lived variable
- The pointer object goes out-of-scope at the end *function*
- The memory is **never deallocated**

```
g++ memory_leak.cc

valgrind --leak-check=full ./a.out
```

```
==17482== Memcheck, a memory error detector
==17482== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17482== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==17482== Command: ./a.out
==17482==
==17482==
==17482== HEAP SUMMARY:
==17482==     in use at exit: 400 bytes in 10 blocks
==17482==   total heap usage: 11 allocs, 1 frees, 73,104 bytes allocated
==17482==
==17482== 400 bytes in 10 blocks are definitely lost in loss record 1 of 1
==17482==    at 0x4848899: malloc (in usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==17482==    by 0x10919E: function() (in /home/wa/Documents/Wahid/Bertinoro/a.out)
==17482==    by 0x1091BF: main (in /home/wa/Documents/Wahid/Bertinoro/a.out)
==17482==
==17482== LEAK SUMMARY:
==17482==    definitely lost: 400 bytes in 10 blocks
==17482==    indirectly lost: 0 bytes in 0 blocks
==17482==      possibly lost: 0 bytes in 0 blocks
==17482==    still reachable: 0 bytes in 0 blocks
==17482==         suppressed: 0 bytes in 0 blocks
==17482==
```

Memory Leak - Example of unreachable delete



```cpp
 1 void foo() {
 2     throw std::runtime_error("");
 3 }
 4 
 5 int main() {
 6     auto* p = new int(5);
 7     std::cout << *p << std::endl;
 8 
 9     try {
10         foo();
11     } catch (const std::exception&) {
12         return 0;
13     }
14 
15     delete p;
16 }
```

```
=================================================================
==1==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 4 byte(s) in 1 object(s) allocated from:
   #0 0x75e550f3a73b in operator new(unsigned long)
(/opt/compiler-explorer/gcc-15.2.0/lib64/libasan.so.8+0x12273b) (BuildId:
620b620e803590cdaf76f7a713f6c544a53408b6)
   #1 0x0000004012cb in main /app/example.cpp:13
   #2 0x75e550829d8f  (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId:
4f7b0c955c3d81d7cac1501a2498b69d1d82bfe7)

SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1 allocation(s).
```

- Here we are correctly deleting the memory that we allocate
- However, an exception is thrown before the deletion, which terminates the program
- The delete is never reached, and the memory is never deallocated

Smart Pointers

A smart pointer is an object that works like a pointer, but it also manages the lifetime of the object it is pointing to.

Exploits the RAII (Resources acquisition is initialization) idiom:

- Resources are acquired in the constructor
- Resources are released in the destructor

In short: remove the need of manually freeing the allocated memory

Unique Pointer: Implementation example

```cpp
#include <iostream>

template <typename Pointee>
class UniquePtr {
  Pointee* m_p;

public:
  explicit UniquePtr(Pointee* p) : m_p{p} {}  // aquire the resource
  ~UniquePtr() { delete m_p; }                // delete the resource

  UniquePtr(UniquePtr const&) = delete;             // more in Francesco’s lectures
  UniquePtr& operator=(UniquePtr const&) = delete;  // more in Francesco’s lectures

  Pointee* operator->() { return m_p; }
  Pointee& operator*() { return *m_p; }
};

int main() {
  UniquePtr<int> p{new int{42}};
  std::cout << *p << '\n';
}
```

Smart pointers in the STL

- The STL provides three types of smart pointers
  - unique_ptr
  - shared_ptr
  - weak_ptr
- **Unique pointers** have the **sole ownership** of the memory they allocate
- **Shared pointers** allocate memory that can be owned by multiple other shared pointers
  - The number of shared pointers owning the same memory region is tracked by the **reference counter**
- Weak pointers point to memory owned by shared pointers, but have **no ownership** over it
  - The creation of a weak pointer **doesn't** increase the reference counter

Example: unique_ptr

```cpp
#include <iostream>
#include <memory>

int main() {
  auto p = std::unique_ptr<int>(new int(10));    // Initialization via constructor
  auto ptr = std::make_unique<int>(42);          // Initialization via factory function (preferred)

  std::cout << *ptr << std::endl;                // Smart pointers provide the same dereference
  *ptr = 100;                                    // dereference as regular pointers

  auto* raw_ptr = ptr.get();                     // Get the raw pointer

  auto array = std::make_unique<float[]>(size);  // Can also allocate arrays
  for (size_t i = 0; i < size; ++i) {
    array[i] = static_cast<float>(i) * 1.5f;     // Access elements like regular arrays
  }
}
```

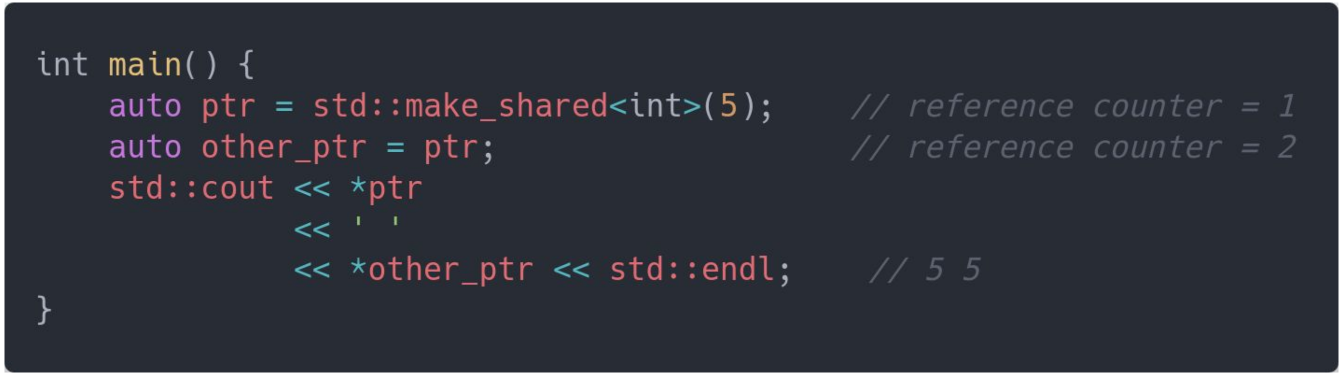
unique_ptr

- The memory allocated by a unique_ptr can only be owned by that unique_ptr
- This means that unique pointers cannot be **copied**
- However, they can be **moved**
  - See Francesco’s lecture
- When passing unique_ptr to a function, either move or pass by **const-ref**

```cpp
#include <iostream>
#include <memory>

void foo(std::unique_ptr<int> p) { std::cout << "Value of the pointee: " << *p << std::endl; }

void bar(const std::unique_ptr<int>& p) { std::cout << "Value of the pointee: " << *p << std::endl; }

int main() {
  auto ptr = std::make_unique<int>(100);

  foo(ptr);             // This will not compile
  foo(std::move(ptr));  // This will compile
  bar(ptr);             // This will compile
}
```

shared_ptr

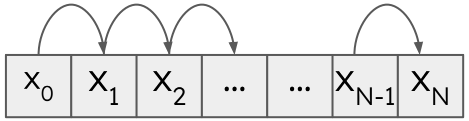
- Shared pointers can be constructed in the same way as unique pointers
  - constructor
  - `std::make_shared` factory function
- The memory they allocate can be owned by multiple shared pointers
- They can be copied, unlike unique pointers
- When they get copied, the reference counter of all the pointers pointing to the same memory region increases
- Then the reference count goes to zero, the memory gets deallocated



```cpp
int main() {
    auto ptr = std::make_shared<int>(5);    // reference counter = 1
    auto other_ptr = ptr;                   // reference counter = 2
    std::cout << *ptr
              << ' '
              << *other_ptr << std::endl;    // 5 5
}
```

Sequential Memory Access

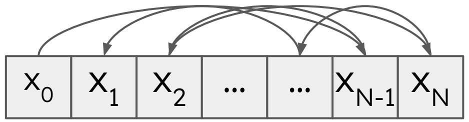
- Consecutive element access
- Good cache locality
- Good memory bandwidth
- Each cycle can read consecutive memory area
    - Cached Memory Access
- Good use of prefetcher



Perfect memory access pattern for CPUs!

Random Memory Access

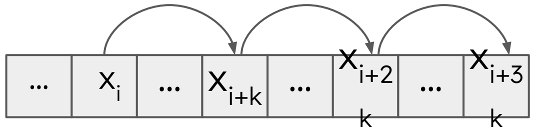
- Elements are accessed in random order
- Cache locality not ensured anymore
- Bad memory bandwidth
- Impossible to prefetch data
- Prefetcher not used



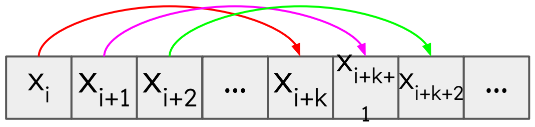
Never use this!

Strided Memory Access

- Elements are accessed at fixed intervals
- Good use of prefetcher
  - Pattern easy to predict



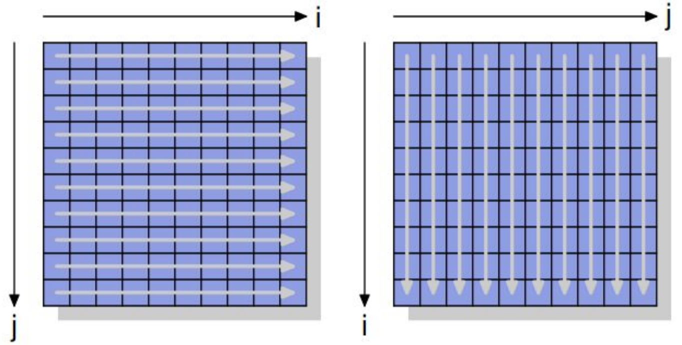
- Very common pattern on GPU
  - Stride size = Grid Size
  - Coalesced memory access
    - Good cache locality and bandwidth



Optimize Memory Access - Example

$$(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{i(N-1)}b_{(N-1)j}$$

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += A[i][k] * B[k][j]
```



A is accessed sequentially → good!
B is not → everytime I jump to another row. For each iteration in the k-loop I get a cache-hit-miss. → Bad spatial locality!

What Every Programmer Should Know About Memory, Ulrich Drepper

Optimize Memory Access - Example

$$(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b^{\mathrm{T}}_{jk} = a_{i1} b^{\mathrm{T}}_{j1} + a_{i2} b^{\mathrm{T}}_{j2} + \cdots + a_{i(N-1)} b^{\mathrm{T}}_{j(N-1)}$$

```
double Bt[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    Bt[i][j] = B[j][i];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += A[i][k] * Bt[j][k]
```

A is accessed sequentially → good!
Bt is accessed sequentially → good!

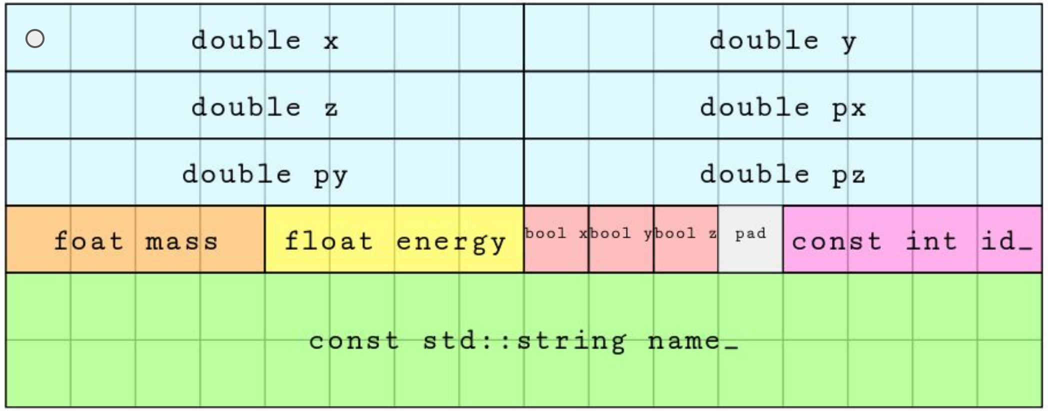
	Original	Transposed
Cycles	16,765,297,870	3,922,373,010
Relative	100%	23.4%

What Every Programmer Should Know About Memory, Ulrich Drepper

Memory Access - Data Structures

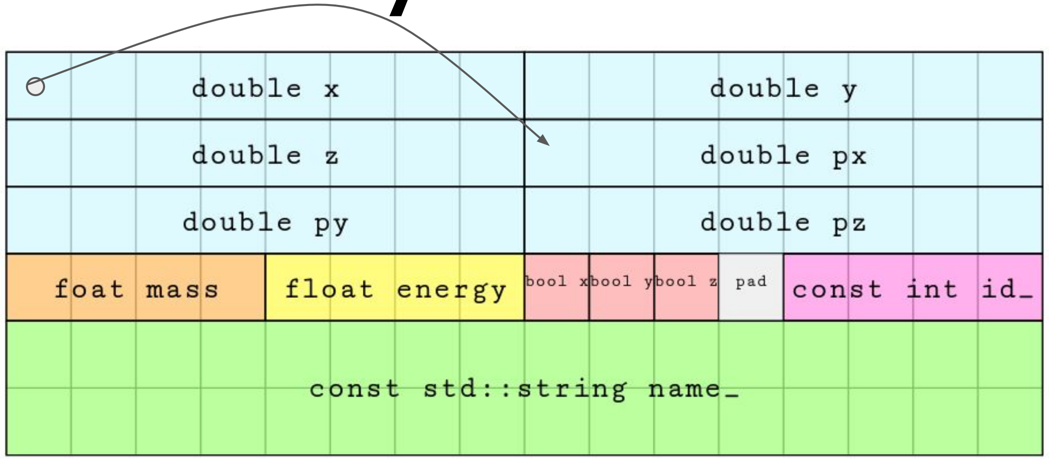
- The way you access memory is not only driven by the algorithm, but it strongly depends on how you designed your data structure
- Let’s investigate our GoodParticle datastructure
    - Exercise hands-on/memory/datastructures/aos_soa.cpp
- Write a function to initialize a collection of N GoodParticles
    - Assign some value to each member of GoodParticle
    - Pick a x_max value
    - And a time value t
- Write another function that takes as input the collection, and x_max
- Iterate over the elements of this collection and for each element:
    - Update the position x → x = x + px / mass * t
    - If x < 0 or x > x_max → set hit_x to true
        - Else, set it to false and change the sign of px

GoodParticle memory access



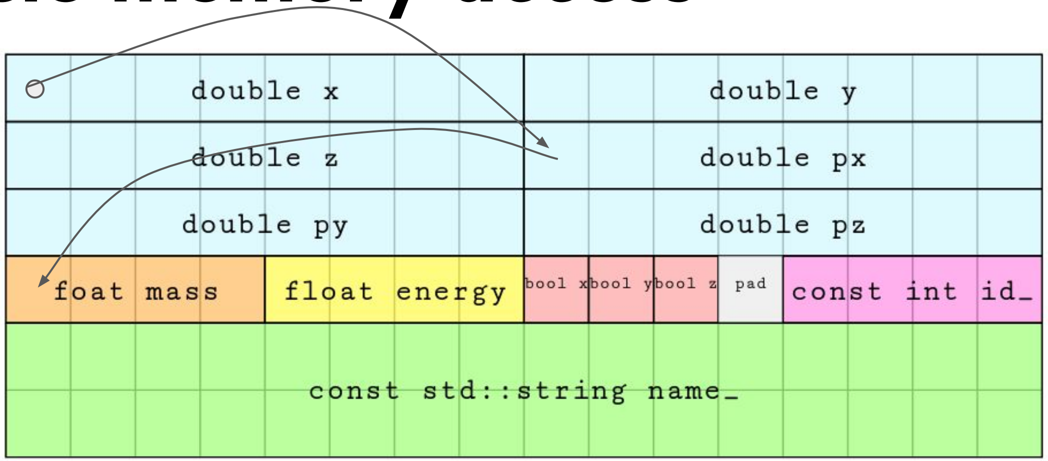
x +=

GoodParticle memory access



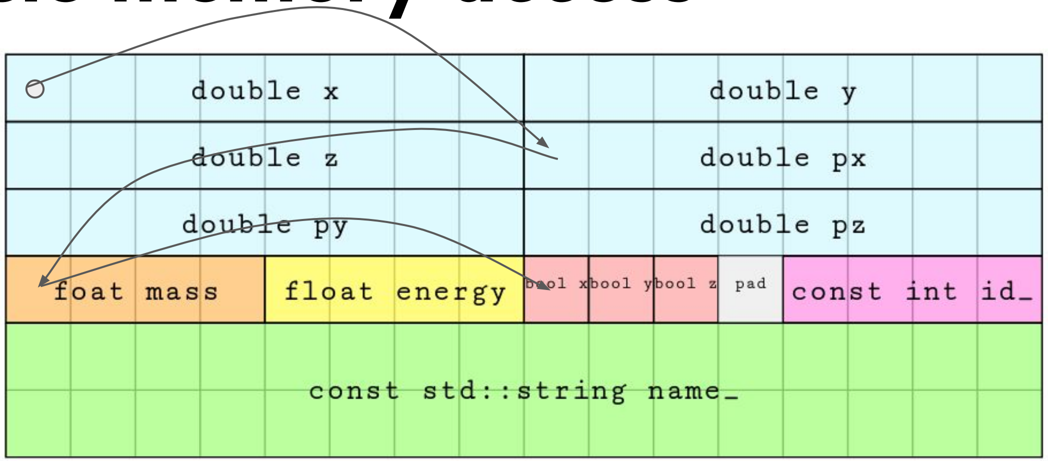
x += px

GoodParticle memory access



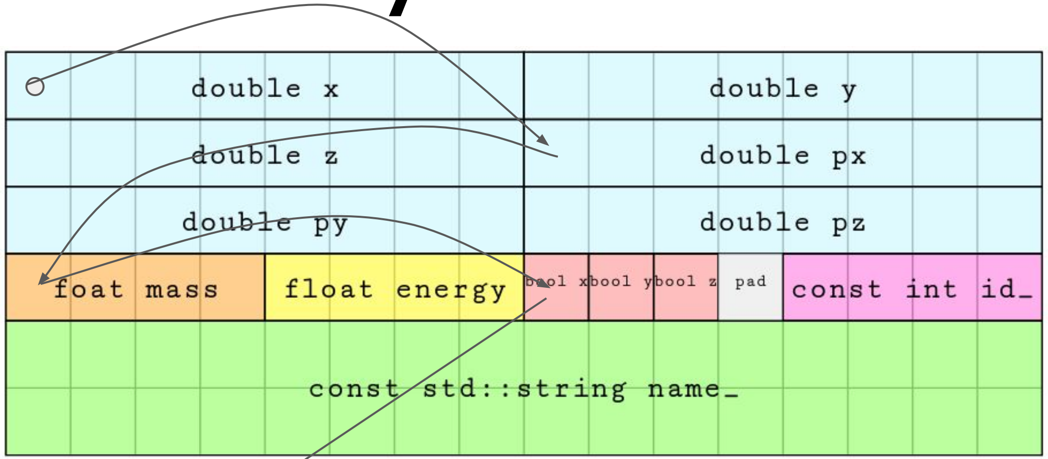
x += px/m * t

GoodParticle memory access



p.x += p.px/p.m * t
p.hit_x = statement? true : false

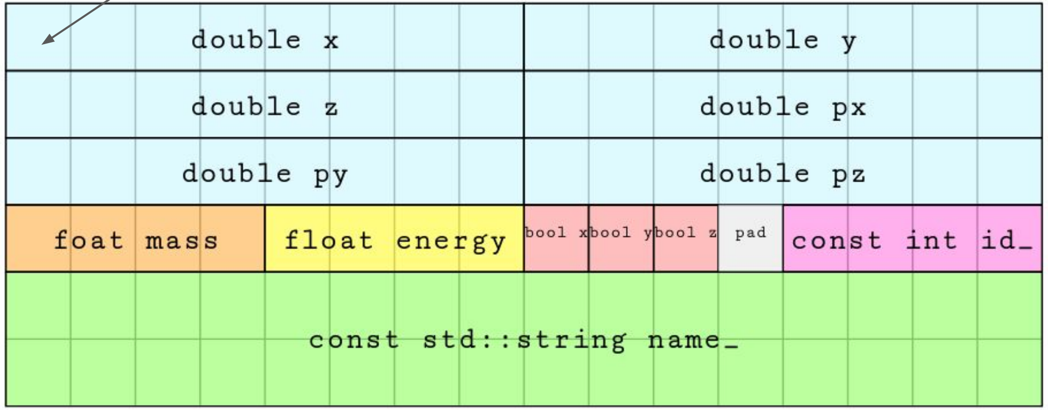
GoodParticle memory access



Particle 1

p.x += p.px/p.m * t
p.hit_x = statement? true : false

Next iteration



Particle 2

GoodParticle memory access

- Our problem needs only some members of our class GoodParticle
  - We are paying the price of loading the full object for accessing its members
  - `sizeof(GoodParticle) = 96bytes`
  - `sizeof`(double$_{x}$) + `sizeof`(double$_{px}$) + `sizeof`(bool$_{hit\_x}$) + `sizeof`(float$_{mass}$) = 21bytes
    - We are using only 22% of what we are reading!
- Our `std::vector<GoodParticle>` is commonly called Array of Struct
  - Very common dastracture coming from Object Oriented Programming (OOP)
    - Self contained objects
      - Bad cache locality and bad memory bandwidth
    - Commonly used because it easy to represent the reality
      - Not great for manipulating data in some scenario
- In principe we would like to have a data structure that allow us to use only what we need in a specific piece of code

Array of Structs vs Struct of Arrays

```
struct Particle {
    double x;
    double y;
    double z;
     …
};

std::vector<Particle> particles;
```

- All data fields for each element are stored together in a contiguous block of memory.
- Cache locality might be loss if not all the elements are used

```
struct ParticleSoA {
    std::vector<double> x;
    std::vector<double> y;
    std::vector<double> z;
     …
};

ParticleSoA particles;
```

- Each data field of all elements is stored in separate arrays.
- This layout is beneficial when you need to perform operations on some fields for all elements concurrently

AoS vs SoA

- Take the last exercise
    - Implement an SoA version of GoodParticle
    - Add two more functions, one for initializing the SoA collection and one to perform the operation previously discussed
- Try to time it
    - Try to use compiler optimization (-O1 -O2 -O3)
    - What happens?
- What memory access pattern are we using now?
- Is your data structure interface that different?

AoS vs SoA

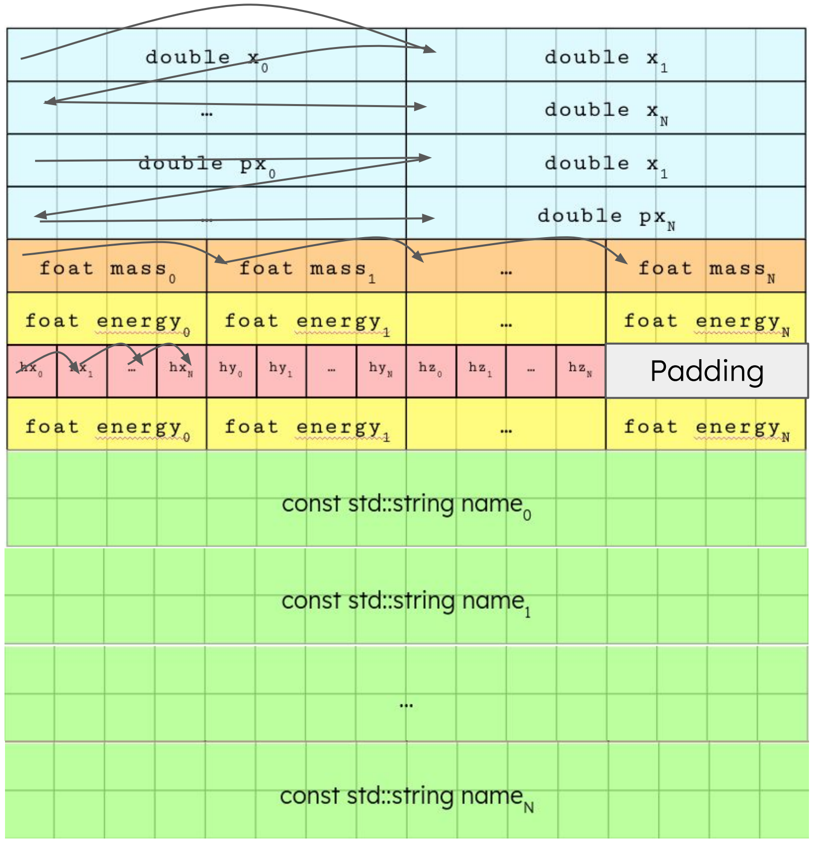
- Sequential access pattern on each member of our object!
- Use only what you need
    - You can pass to your function only the members you are going to use

```
int N = 100;
std::vector<GoodParticleAoS> particles(N);
```

96 bytes * 100 = 9600 bytes
9600 bytes / 64 bytes/cacheline = **150 cache lines**

```
ParticleSoA particles(N);
```

21 bytes * 100 = 2100 bytes
2100 / 64 bytes/cacheline = **33 cache lines**



More on SoA

- So far our SoA uses std::vector, which is useful to be able to resize our datastructure
- However, resizing is quite expensive
- Better to have fixed sized SoA
    - If you don't know your exact size, better to put a Max Value
        - Knowing the size (and alignment) at compile time helps the compiler to optimize your code
            - Especially true for vectorization!
- Moreover, you can use single memory buffers to allocate and deallocate memory in one go, or to transfer it to accelerators
    - And you could also reuse the same memory!

Exercise

- Modify your ParticleSoA struct such that:
    - Contains a single memory buffer and a single size
    - Contains M pointers pointing to the beginning of each “column”
    - Explicit constructor that takes the number of particle you want to allocate
        - Allocates the needed memory with a single operation
        - Set each pointer to the beginning of the column
            - Remember alignment
    - Note → if you allocate the buffer with std::malloc it will give you a void* pointer
        - You can use reinterpret_cast<T*> to cast your pointer to a different type

```
g++ -Wall -Wextra -fsanitize=alignment,address
your_program.cc
```

To check if gcc is happy with your alignment!

Why single allocation?

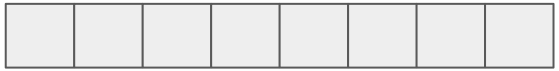
- Easier booking of underlying allocator → the allocator has to keep track of all the allocation and find new memory space → could hurt CPU
- Spatial locality → single block of contiguous memory
- Less fragmentation. Many small allocations fragment the heap, while a monolithic block keeps it compact.

But

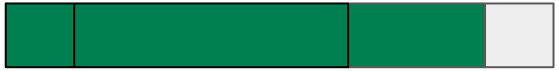
- Less Lifetime flexibility : can’t free/reuse parts independently — all or nothing.
- Manual memory management: error prone
- A single huge block may still span many pages, not all of which are touched
- Concurrency: one shared block may require extra synchronization in multithreaded programs.
- Modern allocators: jemalloc, tcmalloc, etc. already optimize fragmentation and locality, so benefits may be smaller in practice.

Memory Fragmentation

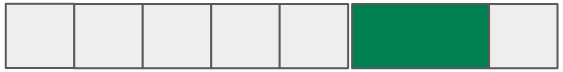
- UNIX system uses the glibc memory allocator



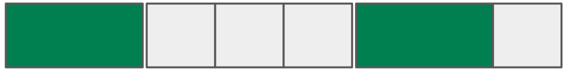
Allocate 1kB, 4kB, 2kB



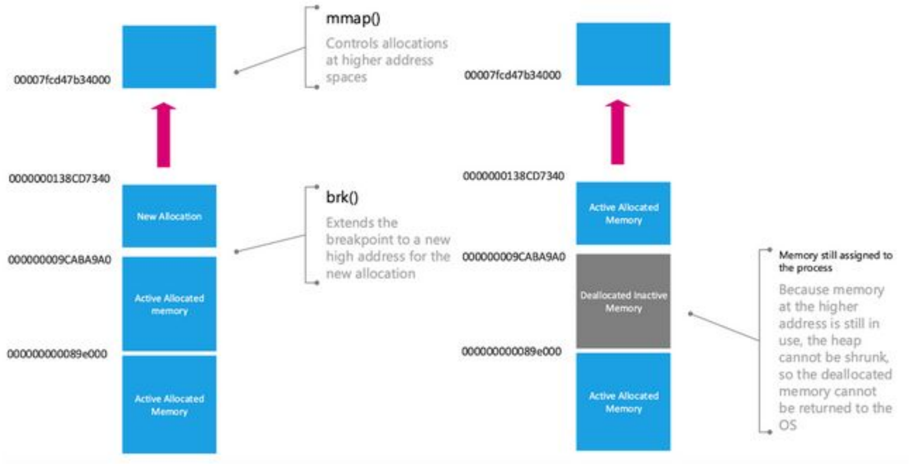
Deallocate 1kB, 4kB



Allocate 2kB



Allocate 4kB → Unable

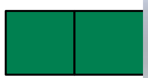

00007fcd47b34000
0000000138CD7340
000000009CABA9A0
000000000089e000
New Allocation
Active Allocated memory
Active Allocated Memory
mmap()
Controls allocations at higher address spaces
brk()
Extends the breakpoint to a new high address for the new allocation
00007fcd47b34000
0000000138CD7340
000000009CABA9A0
000000000089e000
Active Allocated Memory
Deallocated Inactive Memory
Active Allocated Memory
Memory still assigned to the process
Because memory at the higher address is still in use, the heap cannot be shrunk, so the deallocated memory cannot be returned to the OS

4kB are available, but not of contiguous memory
→ Memory Fragmentation

Memory Fragmentation

- UN
  me

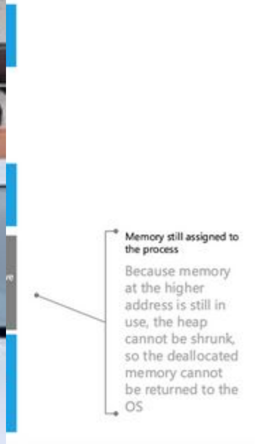
Allocate



Dealloc

Allocate

Allocate


Memory still assigned to the process
Because memory at the higher address is still in use, the heap cannot be shrunk, so the deallocated memory cannot be returned to the OS

Jemalloc and TCMalloc

If your program allocates and deallocates objects with different life times, you get memory fragmentation and the process might not be able to return the memory to the OS

- Alternative allocators
    - Might give you better performance and reduce memory fragmentation
        - But detailed studies are necessary on the full application
- Jemalloc
    - Used by Mozilla Firefox, Facebook, …
    - Tries to avoid memory fragmentation
- TCMalloc
    - Developed by Google
    - Fast C implementation of malloc and new, multithreaded

```
# jemalloc
g++ yourProgam.cpp -O3 -pthread -ljemalloc
#tc malloc
g++ yourProgam.cpp -O3 -pthread -ltcmalloc_minimal
```

Some hints

- Design your datastructures together with your algorithms
  - Don’t try to represent reality with code
    - If some data need different treatment, try to separate them from the rest
- Remember caches
  - E.g.: take some time in writing your for loops
    - Can I design my loop to have a better access pattern?
    - Can I redesign my Datastructure layout to have a better access pattern?
    - Is there anything I can bring out of my loop?
- Try to avoid re-allocation
  - E.g.: std::vector<>.reserve()
  - Custom allocators
- Keep in mind false sharing! → alignment, local variables
- Try to reduce allocation size → reduce your data types (double —> float)
- Avoid copying → pass by ref instead of value, std::span (passing the view of your container)

Take Away Message

- Memory is what keeps you away from running code efficiently
- Keep memory always in mind when you are developing your software
- Remember to understand your hardware and map what you are programming on it
- Investigate your data before developing your data structure and try to understand the memory footprint and how to better access the memory
- Profile profile profile
  - perf, **ASan**, **valgrind**, intel VTune

Reference

- Thanks Andrea Bocci for all the inputs and help in preparing the lecture!
- Reducing memory footprint using jemalloc
  - https://twiki.cern.ch/twiki/bin/view/LCG/VIJemalloc
- **What Every Programmer Should Know About Memory**
  - https://akkadia.org/drepper/cpumemory.pdf
- What Programmers Should Know About Memory Allocation - S. Al Bahra, H. Sowa, P. Khuong - CppCon 2019
  - https://www.youtube.com/watch?v=gYfd25Bdmws&t
- CppCon 2014: Mike Acton "Data-Oriented Design and C++"
  - https://www.youtube.com/watch?v=rX0ItVEVjHc&t=2838s
- Computer Architecture A Quantitative Approach - Fifth Edition -J. Hennessy, D. Patterson
- jemalloc