Small Lie in Big O

Mateusz Pusz

November 15, 2016

SIMPLE PROGRAMMING PROBLEM

‘sample’ is a simple class containing 8-bit unsigned integral member and a few user defined methods. Integral member inside ‘sample’ class should be initialized during class construction.

Implement function ‘test’ that:

- *takes 2 integral arguments ‘n’ and ‘k’*
- *creates a collection of ‘n’ ‘sample’ objects with random values assigned*
- *sorts created collection*
- *inserts a new random value keeping the collection sorted (repeat ‘k’ times)*
- *returns average value from all the samples in the collection*

‘sample’ is a simple class containing 8-bit unsigned integral member and a few user defined methods

```
class sample {
  std::uint8_t value_;
public:

  // more interface...
};
```

Integral member inside ‘sample’ class should be initialized during class construction

```
class sample {
  std::uint8_t value_;
public:
  explicit constexpr sample(std::uint8_t value) noexcept : value_{value} {}
  // more interface...
};
```

implement a function ‘test’ that takes 2 integral arguments ‘n’ and ‘k’

```
class sample {
  std::uint8_t value_;
public:
  explicit constexpr sample(std::uint8_t value) noexcept : value_{value} {}
  // more interface...
};


std::uint8_t test(int n, int k)
{




}
```

implement a function ‘test’ that …

```
class sample {
  std::uint8_t value_;
public:
  explicit constexpr sample(std::uint8_t value) noexcept : value_{value} {}
  // more interface...
};


std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  // sort the collection
  // k times insert a new random sample keeping the collection sorted
  // return average value from samples in the collection
}
```

“sorting”, “comparing values”, … -> need to extend ‘sample’ interface

```
class sample {
  std::uint8_t value_;
public:
  explicit constexpr sample(std::uint8_t value) noexcept : value_{value} {}
  // more interface...

  // returns stored value
  constexpr operator std::uint8_t() const noexcept { return value_; }
};
```

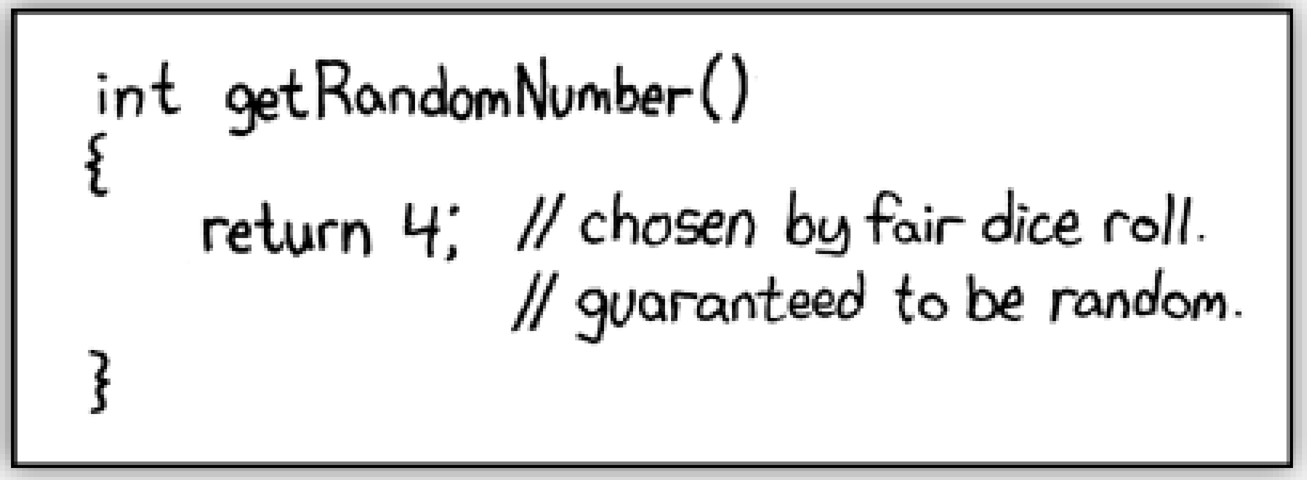
“class”, not a fundamental type -> dynamic memory allocation
“inserting in the middle”, “sorting”, … -> std::list

```
using elem_type = sample*;
using collection = std::list<elem_type>;
```

“random values” -> provide random numbers generator

```
std::uint8_t generate()
{
  ... // probably not enough time today to describe the implementation
}
```

XKCD way ...



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

... or use C++11 <random>

creates a collection of ‘n’ ‘sample’ objects with random values assigned

```
collection init_random(int n)
{
  collection samples;
  for(int i=0; i<num; ++i)
    samples.emplace_back(new sample{generate()});
  return samples;
}

std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  auto samples = init_random(n);

  // ...
}
```

O(n)

sorts created collection

```
std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  auto samples = init_random(n);

  // sort the collection
  values.sort([](const auto& l, const auto& r) { return *l < *r; });

  // ...
}
```

O(nlogn)

inserts a new random value keeping the collection sorted (repeat ‘k’ times)

```
std::uint8_t test(int n, int k)
{
  // ...

  // k times insert a new random sample keeping the collection sorted
  for(int i=0; i<k; ++i) {
    const sample new_sample{generate()};
    samples.emplace(find_if(begin(samples), end(samples),
                            [&](const auto& s) { return new_sample < *s; }),
                    new sample{new_sample});
  }

  // ...
}
```

O(k * (n+k))

returns average value from all the samples in the collection

```cpp
std::uint8_t test(int n, int k)
{
  // ...

  // return average value from samples in the collection
  return std::accumulate(begin(samples), end(samples), std::uint64_t{},
               [](const auto& sum, const auto& s){ return sum + *s; }) / samples.size();
}
```

O(n)

Whole code

Do you see a problem?

```
std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  auto samples = init_random(n);

  // sort the collection
  samples.sort([](const auto& l, const auto& r) { return *l < *r; });

  // k times insert a new random sample keeping the collection sorted
  for(int i = 0; i < k; ++i) {
    const sample new_sample{generate()};
    samples.emplace(find_if(begin(samples), end(samples),
                            [&](const auto& s) { return new_sample < *s; }),
                    new sample{new_sample});
  }

  // return average value from samples in the collection
  return std::accumulate(begin(samples), end(samples), std::uint64_t{},
                         [](const auto& sum, const auto& s){ return sum + *s; }) / samples.size();
}
```

Whole code

```
std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  auto samples = init_random(n);

  // sort the collection
  samples.sort([](const auto& l, const auto& r) { return *l < *r; });

  // k times insert a new random sample keeping the collection sorted
  for(int i = 0; i < k; ++i) {
    const sample new_sample{generate()};
    samples.emplace(find_if(begin(samples), end(samples),
                            [&](const auto& s) { return new_sample < *s; }),
                    new sample{new_sample});
  }

  // return average value from samples in the collection
  return std::accumulate(begin(samples), end(samples), std::uint64_t{},
                         [](const auto& sum, const auto& s){ return sum + *s; }) / samples.size();
}
```

Do you see a problem?

We have a memory leak here

Adding cleanup is simple, right?

```
std::uint8_t test(int n, int k)
{
  // ...

  // cleanup the collection
  for(auto& s : samples)
    delete s;
}
```

Whole code

Do you see a problem?

```cpp
std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  auto samples = init_random(n);

  // sort the collection
  samples.sort([](const auto& l, const auto& r) { return *l < *r; });

  // k times insert a new random sample keeping the collection sorted
  for(int i = 0; i < k; ++i) {
    const sample new_sample{generate()};
    samples.emplace(find_if(begin(samples), end(samples),
                            [&](const auto& s) { return new_sample < *s; }),
                    new sample{new_sample});
  }

  // return average value from samples in the collection
  auto avg = std::accumulate(begin(samples), end(samples), std::uint64_t{},
                             [](const auto& sum, const auto& s){ return sum + *s; }) / samples.size();

  // cleanup the collection
  for(auto& s : samples)
    delete s;

  return avg;
}
```

Whole code

```
std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  auto samples = init_random(n);

  // sort the collection
  samples.sort([](const auto& l, const auto& r) { return *l < *r; });

  // k times insert a new random sample keeping the collection sorted
  for(int i = 0; i < k; ++i) {
    const sample new_sample{generate()};
    samples.emplace(find_if(begin(samples), end(samples),
                            [&](const auto& s) { return new_sample < *s; }),
                    new sample{new_sample});
  }

  // return average value from samples in the collection
  auto avg = std::accumulate(begin(samples), end(samples), std::uint64_t{},
                             [](const auto& sum, const auto& s){ return sum + *s; }) / samples.size();

  // cleanup the collection
  for(auto& s : samples)
    delete s;

  return avg;
}
```

Do you see a problem?

The code is not exception safe!!!

RESOURCE ACQUISITION IS INITIALIZATION (RAII)

```
class resource {
  // resource handle
public:
  resource(/* args */)
  { /* obtain ownership of a resource and store the handle */ }
  ~resource()
  { /* reclaim the resource */ }
};
```

> *If you dissect the words of the RAII acronym (Resource Acquisition Is Initialization), you will think RAII is about acquiring resources during initialization. However the power of RAII comes not from tying* ***acquisition*** *to* ***initialization****, but from tying* ***reclamation*** *to* ***destruction****.*
>
> *-- Bjarne Stroustrup*

RAII USAGE IN C++ STANDARD LIBRARY

- **Smart Pointers**
  - std::unique_ptr<T, Deleter = std::default_delete<T>>
  - std::shared_ptr<T>
  - std::weak_ptr<T>
  - ~~std::auto_ptr<T>~~
- **Containers**
  - all STL containers manage ownership of their data
- **File streams**
- **Mutex locks**
  - std::lock_guard<Mutex>
  - std::unique_lock<Mutex>
- **More…**

Let's fix our code to be C++ exception safe

```
using elem_type = std::unique_ptr<sample>;
// ...

// replace all
new sample{...}
// with
std::make_unique<sample>(...)

std::uint8_t test(int n, int k)
{
  // ...

  ~~// cleanup the collection~~
  ~~for(auto& s : samples)~~
    ~~delete s;~~
}
```

Writing exception safe code is not that hard at all!!!

TWEAKING ‘N’ IN ‘F(N)’

Can we make our code faster?

Solution	test(10 000 000, 0)	test(0, 100 000)	test(100 000, 10 000)
Original	10,7s	104,5s	32,6s

TWEAKING ‘N’ IN ‘F(N)’

Can we make our code faster?

Solution	test(10 000 000, 0)	test(0, 100 000)	test(100 000, 10 000)
Original	10,7s	104,5s	32,6s
Change #1	9,3s	57,1s	16,4s
Change #2	0,75s	1,7s	0,37s
Change #3	0,75s	0,98s	0,21s
Speedup	**14x**	**106x**	**155x**

Yes, A LOT faster

“class”, not a fundamental type -> dynamic memory allocation
“inserting in the middle”, “sorting”, … -> std::list

```
using elem_type = sample*;
using collection = std::list<elem_type>;
```

WRONG!!!

C++ IS NOT C# OR JAVA

- Heap usage should be avoided if possible
  - allocation and deallocation of heap memory is slow
  - obtaining data from non-cached memory is slow
  - heap allocation can fail
  - allocation of many small objects causes huge memory fragmentation
- C++ loves value semantics
  - using pointers changes semantics of copy, assignment and equality
  - pointer dereferencing takes time
  - much easier to write thread-safe code
  - reference/pointer semantics causes aliasing problems
    - compiler optimizer cannot do its best -> slower code
  - Copy Elision (RVO) and Move Semantics improve things a lot

CHANGE #1

Avoid pointers

```
using elem_type = sample;
// ...

std::uint8_t test(int n, int k)
{
  // ...
  samples.sort(~~[](const auto& l, const auto& r) { return *l < *r; }~~);
  // ...
  auto avg = std::accumulate(begin(samples), end(samples), std::uint64_t{},
                             ~~[](const auto& sum, const auto& s){ return sum + *s; }~~) / samples.size();
  // ...
}
```

1.15x-2x speedup

CHANGE #2

Use std::vector<T> as a default container

```cpp
using collection = std::vector<elem_type>;

collection init_random(int n, int k)
{
  collection samples;
  samples.reserve(n + k);
  for(int i=0; i<n; ++i)
    samples.emplace_back(generate());
  return samples;
}
```

```cpp
std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  auto samples = init_random(n, k);
  // ...

  // sort the collection
  sort(begin(samples), end(samples));
  // ...
}
```

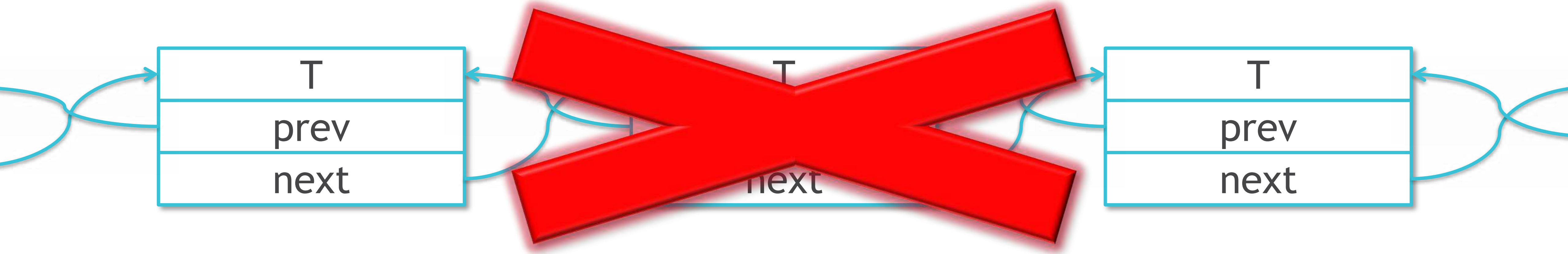
12x-44x speedup. Why?

ALGORITHMIC POINT OF VIEW

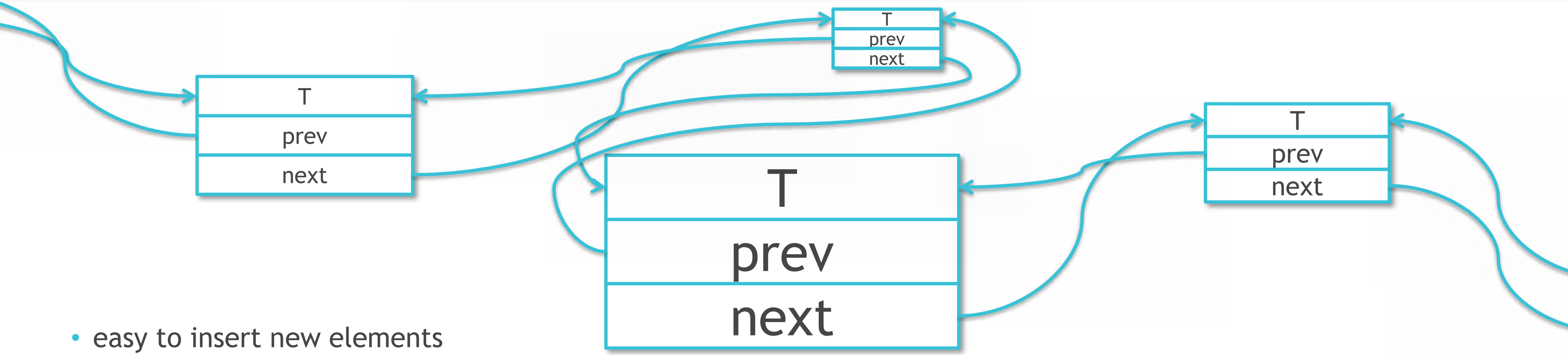
Operation	std::list<T>	std::vector<T>
create container of 'n' random values	O(n)	O(n)
sort container	O(nlogn)	O(nlogn)
'k' times insert a new random value keeping the container sorted	O(k * (n+k))	O(k * (n+k))
verify if sorted	O(n+k)	O(n+k)
Overall complexity	**O(k * (n+k))**	**O(k * (n+k))**

12x-44x speedup. Why?

std::list<T>

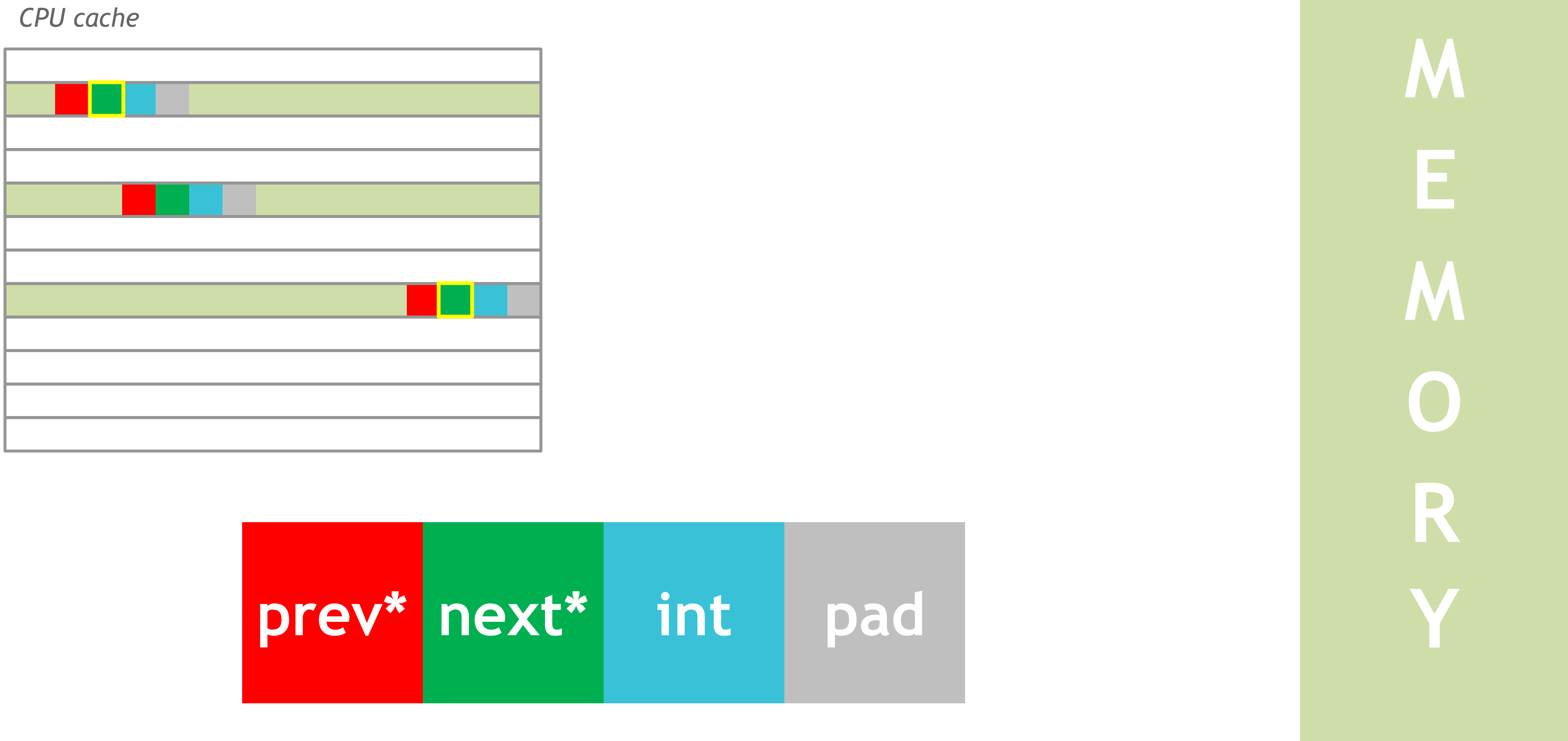


std::list<T>

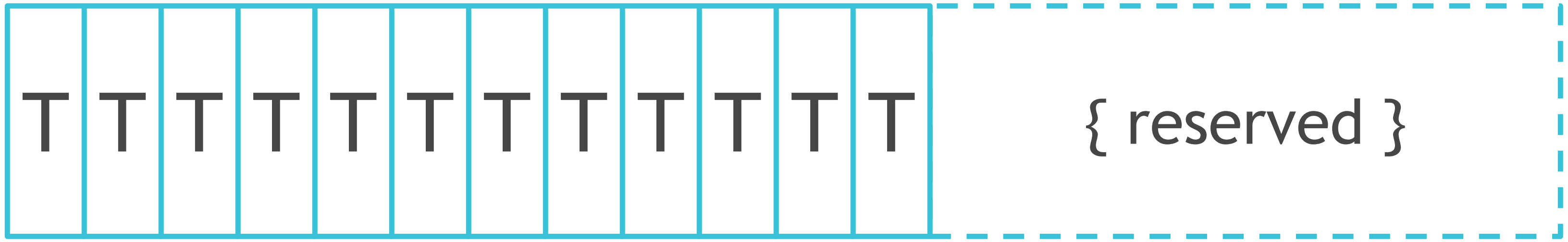


- easy to insert new elements
- easy to erase existing elements
- easy to reorder elements
- bidirectional iteration
- memory ineffective
  - for small T large memory overhead
  - a lot of dynamic memory allocations and deallocations
  - a lot of pointer dereferences during iteration
  - not cache friendly

std::list<int> ITERATION

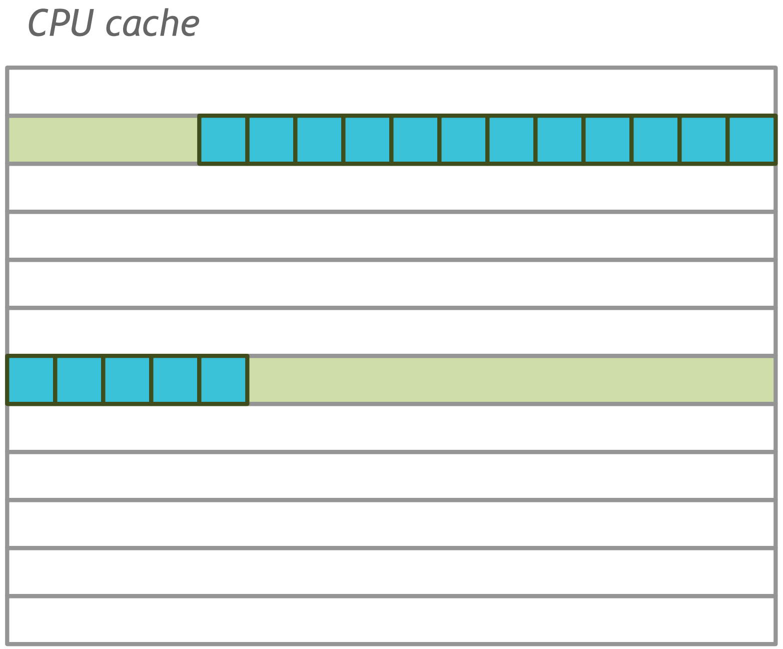


std::vector<T>



- inserting new elements
  - end – easy (unless buffer reallocation is needed)
  - begin, middle – hard
- erasing existing elements
  - end – easy
  - begin, middle – hard
- swapping values needed to reorder elements
- random access iteration
- memory effective
  - small memory overhead (reserved space)
  - nearly no dynamic memory allocations and deallocations
  - no pointer dereferences during iteration
  - cache friendly!!!

std::vector<int> ITERATION



Writing cache friendly code makes a huge difference!

LIST

```
Performance counter stats for './test_list':
      83991,294231      task-clock (msec)         #    0,999 CPUs utilized
      213901565783      cycles                    #    2,547 GHz
       27748493084      instructions              #    0,13  insn per cycle
        8472128537      branches                  #  100,869 M/sec
          61763769      branch-misses             #    0,73% of all branches
        8822212776      L1-dcache-loads           #  105,037 M/sec
        4799991469      L1-dcache-load-misses     #   54,41% of all L1-dcache hits
        4177763046      LLC-loads                 #   49,740 M/sec
         550506468      LLC-load-misses           #   13,18% of all LL-cache hits
             78237      page-faults               #    0,931 K/sec
```

VECTOR

```
Performance counter stats for './test_vector':
       2826,385943      task-clock (msec)         #    0,999 CPUs utilized
        7462656329      cycles                    #    2,640 GHz
       27117855932      instructions              #    3,63  insn per cycle
        7396349132      branches                  # 2616,893 M/sec
          39248559      branch-misses             #    0,53% of all branches
        6466518567      L1-dcache-loads           # 2287,911 M/sec
          77800147      L1-dcache-load-misses     #    1,20% of all L1-dcache hits
             89304      LLC-loads                 #    0,032 M/sec
             36469      LLC-load-misses           #   40,84% of all LL-cache hits
              1050      page-faults               #    0,371 K/sec
```

Latency Numbers Every Programmer Should Know

```
L1 cache reference                          0.5 ns
Branch misprediction                        5   ns
L2 cache reference                          7   ns    14x L1 cache
Mutex lock/unlock                          25   ns
Main memory reference                     100   ns    20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy            3,000   ns
Send 1K bytes over 1 Gbps network      10,000   ns
Read 4K randomly from SSD             150,000   ns
Read 1 MB sequentially from memory    250,000   ns
Round trip within same datacenter     500,000   ns
Read 1 MB sequentially from SSD     1,000,000   ns    4X memory
Disk seek                          10,000,000   ns    20x datacenter roundtrip
Read 1 MB sequentially from disk   20,000,000   ns    80x memory, 20X SSD
Send packet CA->Netherlands->CA   150,000,000   ns
```

-- Jeff Dean

A NEW POSSIBILITY FOR ALGORITHMIC OPTIMIZATION

```
std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  // sort the collection
  // k times insert a new random sample keeping the collection sorted
  // return average value from samples in the collection
}
```

§25.4.3.1
template<class ForwardIterator, class T>
*ForwardIterator **lower_bound**(ForwardIterator **first**, ForwardIterator **last**, const T& **value**);*

***1 Requires**: The elements e of [first,last) shall be partitioned with respect to the expression e < value*
***2 Returns**: The furthermost iterator i in the range [first,last] such that for every iterator j in the range [first,i) the following corresponding conditions hold: *j < value.*
***3 Complexity**: At most log2(last − first) + O(1) comparisons.*

CHANGE #3

Know and use C++ algorithms

```
std::uint8_t test(int n, int k)
{
  // ...
  // k times insert a new random sample keeping the collection sorted
  for(int i=0; i<k; ++i) {
    const sample new_sample{generate()};
    samples.emplace(lower_bound(begin(samples), end(samples), new_sample), new_sample);
  }
  // ...
}
```

1x-1.7x speedup

Whole code

```
collection init_random(int n, int k)
{
  collection samples;
  samples.reserve(n + k);
  for(int i=0; i<n; ++i)
    samples.emplace_back(generate());
  return samples;
}

std::uint8_t test(int n, int k)
{
  auto samples = init_random(n, k);
  sort(begin(samples), end(samples));

  // k times insert a new random sample keeping the collection sorted
  for(int i=0; i<k; ++i) {
    const sample new_sample{generate()};
    samples.emplace(lower_bound(begin(samples), end(samples), new_sample), new_sample);
  }

  return std::accumulate(begin(samples), end(samples), std::uint64_t{}) / samples.size();
}
```

Isn’t it a great example of efficient programming?

QUOTES FROM THE C++ COMMUNITY

> Using std::map is an exercise in slowing down code.
> If you think link list is bad... maps are worse.

std::multiset<sample> usage

	test(10 000 000, 0)		test(0, 100 000)		test(100 000, 10 000)	
Metric	vector	multiset	vector	multiset	vector	multiset
task-clock [msec]	765	9162				
Instructions [insn/cycle]	1,33	0,31				
L1-dcache-loads [M/sec]	645,8	189,6				
L1-dcache-load-misses	0,39%	29,91%				
LLC-loads [M/sec]	0,074	37,9				
LLC-load-misses	37,55%	18,61%				
page-faults [K/sec]	1	11				

std::multiset<sample> usage

	test(10 000 000, 0)		test(0, 100 000)		test(100 000, 10 000)	
Metric	vector	multiset	vector	multiset	vector	multiset
task-clock [msec]	765	9162				
Instructions [insn/cycle]	1,33	0,31	3,86	0,60		
L1-dcache-loads [M/sec]	645,8	189,6	2557,5	326,3		
L1-dcache-load-misses	0,39%	29,91%	1,03%	12,06%		
LLC-loads [M/sec]	0,074	37,9	0,035	17,7		
LLC-load-misses	37,55%	18,61%	22,45%	16,84%		
page-faults [K/sec]	1	11	0,14	23		

std::multiset<sample> usage

	test(10 000 000, 0)		test(0, 100 000)		test(100 000, 10 000)	
Metric	vector	multiset	vector	multiset	vector	multiset
task-clock [msec]	765	9162	978	46		
Instructions [insn/cycle]	1,33	0,31	3,86	0,60		
L1-dcache-loads [M/sec]	645,8	189,6	2557,5	326,3		
L1-dcache-load-misses	0,39%	29,91%	1,03%	12,06%		
LLC-loads [M/sec]	0,074	37,9	0,035	17,7		
LLC-load-misses	37,55%	18,61%	22,45%	16,84%		
page-faults [K/sec]	1	11	0,14	23		

std::multiset<sample> usage

	test(10 000 000, 0)		test(0, 100 000)		test(100 000, 10 000)	
Metric	vector	multiset	vector	multiset	vector	multiset
task-clock [msec]	765	9162	978	46	212	61
Instructions [insn/cycle]	1,33	0,31	3,86	0,60	3,70	0,59
L1-dcache-loads [M/sec]	645,8	189,6	2557,5	326,3	2598,5	367,6
L1-dcache-load-misses	0,39%	29,91%	1,03%	12,06%	1,45%	19,20%
LLC-loads [M/sec]	0,074	37,9	0,035	17,7	0,084	16,9
LLC-load-misses	37,55%	18,61%	22,45%	16,84%	21,07%	8,38%
page-faults [K/sec]	1	11	0,14	23	0,67	23

TWEAKING ‘N’ IN ‘F(N)’

Can we make our code faster?

Solution	test(10 000 000, 0)	test(0, 100 000)	test(100 000, 10 000)
Original	10,7s	104,5s	32,6s
Change #1	9,3s	57,1s	16,4s
Change #2	0,75s	1,7s	0,37s
Change #3	0,75s	0,98s	0,21s
Speedup	**14x**	**106x**	**155x**

TWEAKING ‘N’ IN ‘F(N)’

Can we make our code faster?

Solution	test(10 000 000, 0)	test(0, 100 000)	test(100 000, 10 000)
Original	10,7s	104,5s	32,6s
Change #1	9,3s	57,1s	16,4s
Change #2	0,75s	1,7s	0,37s
Change #3	0,75s	0,98s	0,21s
Bonus #1	8,9s	0,04s	0,05s
Bonus #2	1,7s	0,04s	0,02s
Speedup	**???**	**2 612x**	**1 630x**

Bonus #1

std::multiset<sample> usage

```
using collection = std::multiset<elem_type>;
// ...

std::uint8_t test(int n, int k)
{
  // ...
  ~~sort(begin(samples), end(samples));~~
  // ...
    samples.emplace(~~lower_bound(begin(samples), end(samples), new_sample)~~, new_sample);
  // ...
}
```

0.08x-24x speedup

Bonus #2

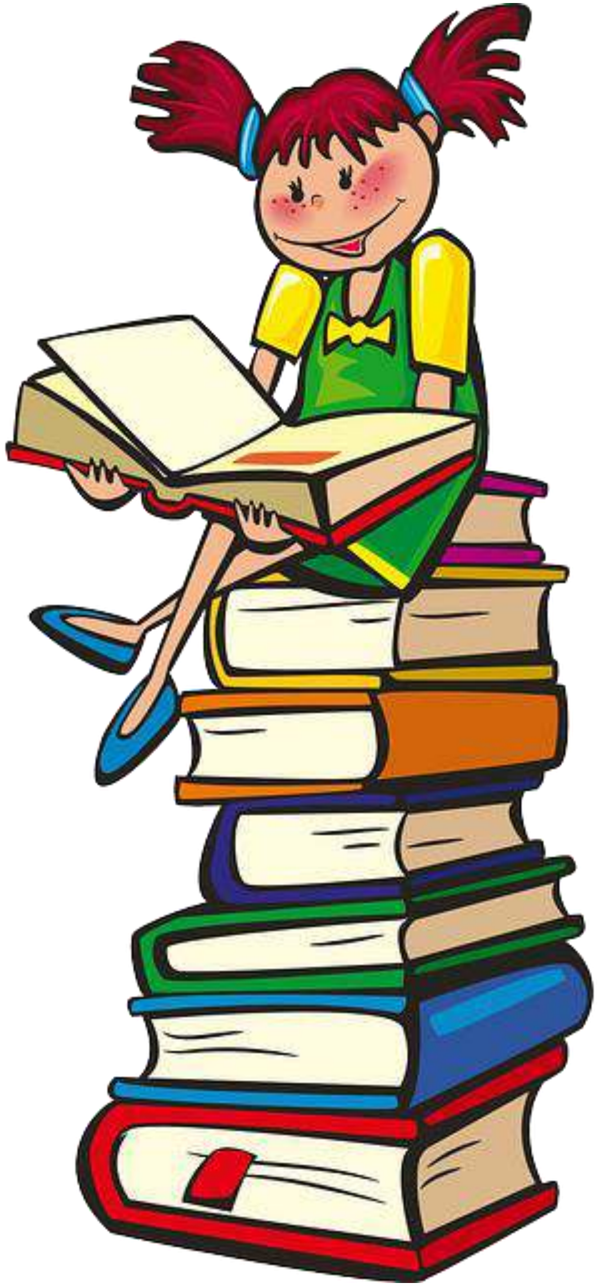
std::multiset<sample> + std::vector<sample> usage

```
using collection1 = std::vector<elem_type>;   using collection2 = std::multiset<elem_type>;
// ...

std::uint8_t test(int n, int k)
{
  // create collection of 'n' random samples
  collection1 samples1 = init_random(n, k);

  // sort the collection
  sort(begin(samples1), end(samples1));

  collection2 samples2{begin(samples1), end(samples1)};
  // ...
}
```

TAKE AWAYS

Learn C++ to be a more efficient programmer

- containers,
- algorithms,
- new C++11/C++14/C++17 features



Use RAII design pattern as your first resort tool for management of all resources

Use tools provided by the C++ standard or C++ community

<u>Do not reinvent the wheel!!!</u>

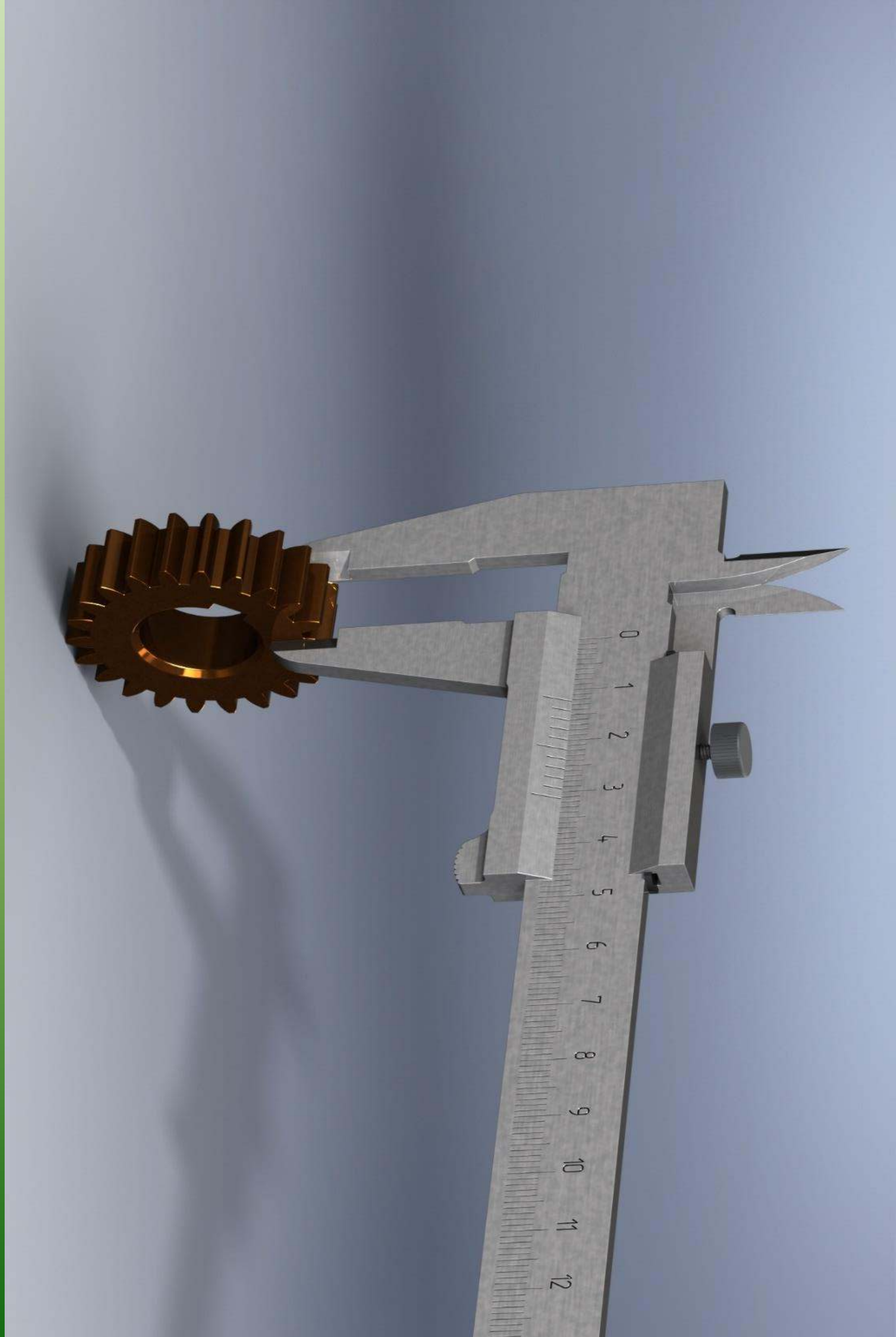
Remember that <u>memory access is the bottleneck</u> of many today’s applications!

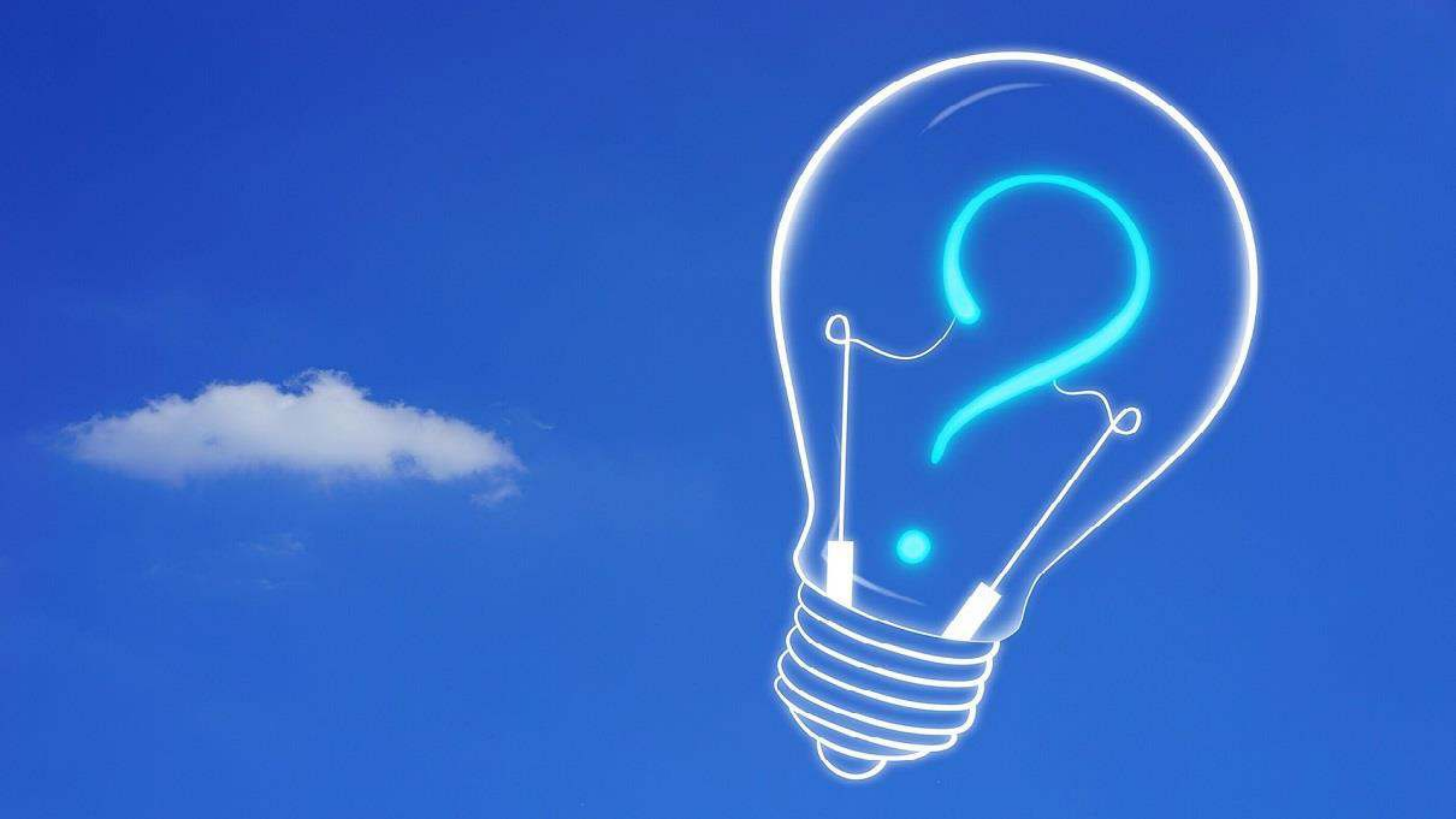
Limit pointers usage
Write cache friendly code

Photograph by Mauro Sartori

Always meassure your code if you care for performance!

Do not do premature optimizations based only on your assumptions (algorithms, CPU cache friendliness or others)





CAUTION

Programming is addictive (and too much fun)

BONUS SLIDE

PSEUDO-RANDOM NUMBERS GENERATOR

```
template<typename T>
auto make_random_int_generator(T min = std::numeric_limits<T>::min(),
                               T max = std::numeric_limits<T>::max(),
                               std::mt19937::result_type seed = std::mt19937::default_seed)
{
    static_assert(std::is_integral_v<T>);
    std::mt19937 gen{seed};
    std::uniform_int_distribution<T> distr{min, maxs};
    return [=]() mutable { return distr(gen); };
}

std::uint8_t generate()
{
    static auto generator = make_random_int_generator<std::uint8_t>();
    return generator();
}
```