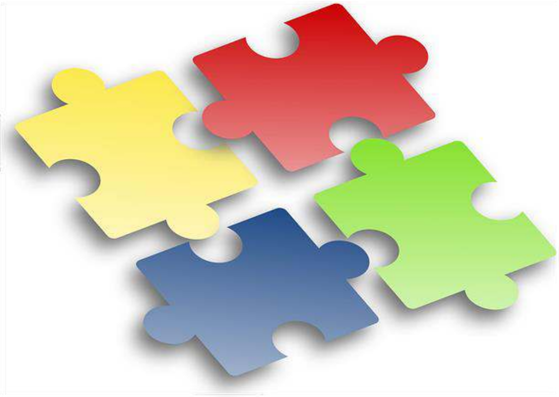


A short talk about efficient C++ programming

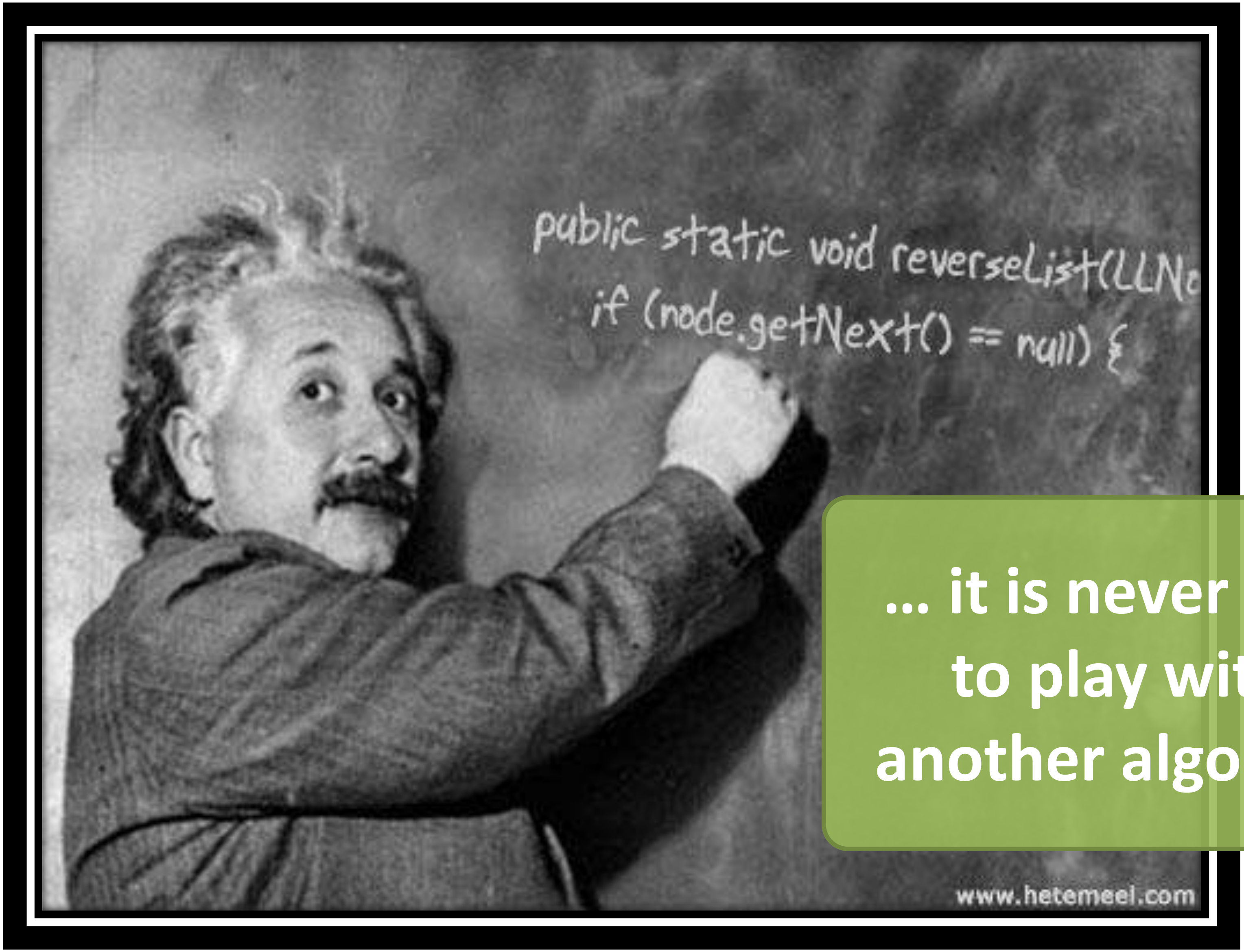
„C++11 feels like a new language:
The pieces just fit together better than they used to
and I find a higher-level style of programming
more natural than before and as efficient as ever.”
Bjarne Stroustrup

Mateusz Pusz
Intel Technology Poland

Plan for next 45 minutes

- **We will scope on efficient programming in C++ with special attention on C++11**
  - Can C++ be efficient at all???
- **I will not bore you with C++ reference documentation for any specific C++11 feature**
  - There are already good books, blogs and Wikipedia out there
- **However I will showcase you a lot of new C++11 features**
- **I want to show you how different aspects of C++ computer programming interact with each other**
  - C++ is a programming language and not just a collection of separate features
- **Hopefully we will have some fun too!**
- **And …**

*



```
public static void reverseList(LLNo
  if (node.getNext() == null) {
```

… it is never too late to play with yet another algorithm ☺

Simple programming problem

‘my_int’ is a simple class containing ‘int’ as a member and a few user defined methods. ‘int’ member inside ‘my_int’ class should be initialized during class construction.

Implement a function ‘test’ that:

- *takes 2 integral arguments ‘n’ and ‘k’*
- *creates a container of ‘n’ ‘my_int’ objects with random values assigned*
- *sorts created container*
- *inserts a new random value keeping the container sorted (repeat ‘k’ times)*
- *verifies if the resulting container is sorted*

‘my_int’ is a simple class containing ‘int’ as a member and a few user defined methods

```cpp
class my_int {
  int _value;
public:

  // some methods...
};
```

‘int’ member inside ‘my_int’ class should be initialized during class construction

```
class my_int {
  int _value;
public:
  explicit constexpr my_int(int value): _value{value} {}
  // some methods...
};
```

generalized constant expression

uniform initialization

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

```
class my_int {
  int _value;
public:
  explicit constexpr my_int(int value): _value{value} {}
  // some methods...
};


void test(int n, int k)
{




}
```

implement a function ‘test’ that …

```
class my_int {
  int _value;
public:
  explicit constexpr my_int(int value): _value{value} {}
  // some methods...
};


void test(int n, int k)
{
  // create container of 'n' random values
  // sort container
  // k times insert a new random value keeping the container sorted
  // verify if sorted
}
```

“sorting”, “comparing values”, … –> I need to extend ‘my_int’ interface

```
class my_int {
  int _value;
public:
  explicit constexpr my_int(int value): _value{value} {}
  // some methods...

  // returns stored value
  explicit constexpr operator int() const { return _value; }

  // user-defined less-comparator needed for sorting
  friend constexpr bool operator <(const my_int& left,
                                   const my_int& right)
  { return left._value < right._value; }
};
```

explicit conversion operator

“class”, not a fundamental type –> operator new

“inserting in the middle”, “sorting”, … –> std::list

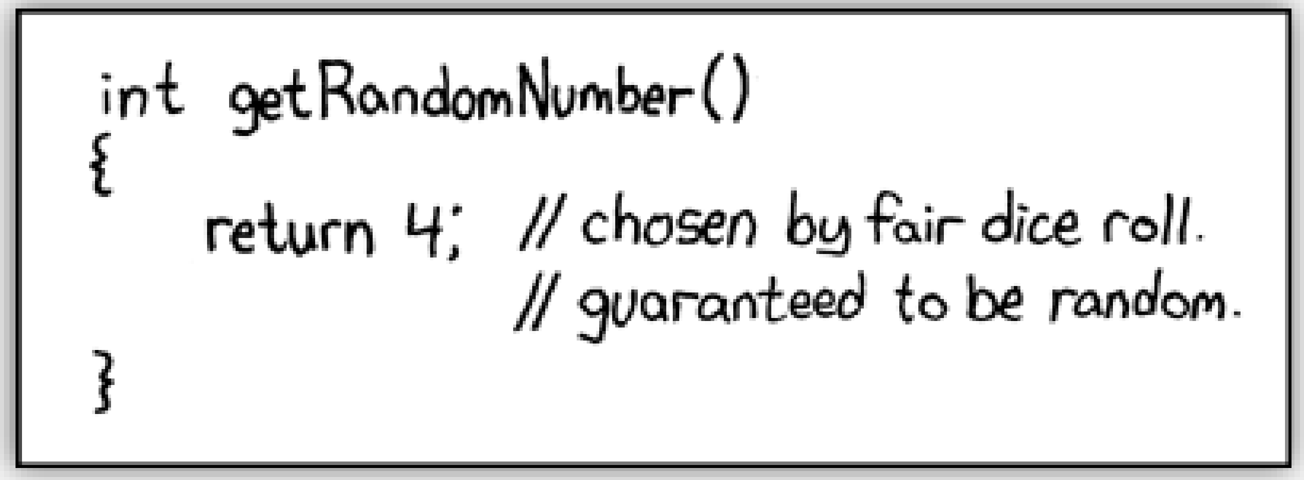
```
using elem_type = my_int*;
using container = std::list<elem_type>;
```

type aliases

“random values” –> provide random numbers generator

```
int generate()
{
  ... // 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 container of ‘n’ ‘my_int’ objects with random values assigned

```
int generate();       // implementation provided in Backup

void init_random(int num, container& values)
{
  for(int i=0; i<num; ++i)
    values.emplace_back(new my_int{generate()});
}

void test(int n, int k)
{
  // create container of 'n' random values
  container values;
  init_random(n, values);
  // ...
}
```

faster container operations based on Perfect Forwarding

O(n)

sorts created container

```
void test(int n, int k)
{
  // ...

  // sort container
  auto ptr_cmp = [](const elem_type& l, const elem_type& r)
                 { return *l < *r; };
  values.sort(ptr_cmp);

  // ...
}
```

type inference

lambda -> [captures](args){ body; }

O(nlogn)

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

```
void test(int n, int k)
{
  // ...

  // k times insert a new random value keeping the container sorted
  for(int i=0; i<k; ++i) {
    const int val = generate();
    values.emplace(find_if(begin(values), end(values),
                           [&](const elem_type& v)
                           { return static_cast<int>(*v) > val; }),
                   new my_int{val});
  }

  // ...
}
```

faster container operations based on Perfect Forwarding

O(k * (n+k))

verifies if the resulting container is sorted

```
void test(int n, int k)
{
  // ...
  auto ptr_cmp = [](const elem_type& l, const elem_type& r)
                 { return *l < *r; };
  // ...

  // verify if sorted
  std::cout << (is_sorted(begin(values), end(values), ptr_cmp) ?
                "PASS\n" : "FAIL\n");
}
```

new C++11 algorithm

O(n)

```
$./test
PASS
```

Are we there yet?

Whole code

Do you see a problem?

```
void test(int n, int k)
{
  // create container of 'n' random values
  container values;
  init_random(n, values);

  // sort container
  auto ptr_cmp = [](const elem_type& l, const elem_type& r){ return *l < *r; };
  values.sort(ptr_cmp);

  // k times insert a new random value keeping the container sorted
  for(int i=0; i<k; ++i) {
    const int val = generate();
    values.emplace(find_if(begin(values), end(values),
                           [&](const elem_type& v)
                           { return static_cast<int>(*v) > val; }),
                   new my_int{val});
  }

  // verify if sorted
  std::cout << (is_sorted(begin(values), end(values), ptr_cmp) ? "PASS\n":"FAIL\n");
}
```

Whole code

Do you see a problem?

```
void test(int n, int k)
{
  // create container of 'n' random values
  container values;
  init_random(n, values);

  // sort container
  auto ptr_cmp = [](const elem_type& l, const elem_type& r){ return *l < *r; };
  values.sort(ptr_cmp);

  // k times insert a new random value keeping the container sorted
  for(int i=0; i<k; ++i) {
    const int val = generate();
    values.emplace(find_if(begin(values), end(values),
                           [&](const elem_type& v)
                           { return static_cast<int>(*v) > val; }),
                   new my_int{val});
  }

  // verify if sorted
  std::cout << (is_sorted(begin(values), end(values), ptr_cmp) ? "PASS\n":"FAIL\n");
}
```

We have a memory leak here

Adding cleanup is simple, right?

```
void test(int n, int k)
{
  // ...

  // cleanup the container
  for(auto& elem : values)
    delete elem;
}
```

range-based for loop

Whole code

Do you see a problem?

```
void test(int n, int k)
{
  // create container of 'n' random values
  container values;
  init_random(n, values);

  // sort container
  auto ptr_cmp = [](const elem_type& l, const elem_type& r){ return *l < *r; };
  values.sort(ptr_cmp);

  // k times insert a new random value keeping the container sorted
  for(int i=0; i<k; ++i) {
    const int val = generate();
    values.emplace(find_if(begin(values), end(values),
                           [&](const elem_type& v)
                           { return static_cast<int>(*v) > val; }),
                   new my_int{val});
  }

  // verify if sorted
  std::cout << (is_sorted(begin(values), end(values), ptr_cmp) ? "PASS\n“ : "FAIL\n");

  // cleanup the container
  for(auto& elem : values)
    delete elem;
}
```

Whole code

Do you see a problem?

```
void test(int n, int k)
{
  // create container of 'n' random values
  container values;
  init_random(n, values);

  // sort container
  auto ptr_cmp = [](const elem_type& l, const elem_type& r){ return *l < *r; };
  values.sort(ptr_cmp);

  // k times insert a new random value keeping the container sorted
  for(int i=0; i<k; ++i) {
    const int val = generate();
    values.emplace(find_if(begin(values), end(values),
                           [&](const elem_type& v)
                           { return static_cast<int>(*v) > val; }),
                   new my_int{val});
  }

  // verify if sorted
  std::cout << (is_sorted(begin(values), end(values), ptr_cmp) ? "PASS\n“ : "FAIL\n");

  // cleanup the container
  for(auto& elem : values)
    delete elem;
}
```

The code is not exception safe!!!

Resource Acquisition Is Initialization (RAII)

```
class MyResource {
  // private resource
public:
  MyResource(/* args */)
  { // do whatever is needed to obtain ownership over resource }
  ~MyResource()
  { // do whatever is needed to reclaim the resource }
  // more class stuff here
};
```

> *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
  - std::string also became a regular container in C++11
- 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<my_int>;
// ...

void test(int n, int k)
{
  // ...

  ~~// cleanup the container~~
  ~~for(auto &elem : values)~~
    ~~delete elem;~~
}
```

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

Tweaking ‘n’ in ‘f(n)’

Can we make our code faster?

Solution	test(100 000, 10 000)	test(10 000 000, 0)	test(0, 100 000)
Original	25,5s	14,3s	63,2s
Change 1	9,75s	9,6s	30,4s
Change 2	0,52s	0,88s	2,4
Change 3	0,33s	0,88s	1,5s
Speedup	**77x**	**16x**	**42x**

Yes, A LOT faster

“class”, not a fundamental type –> operator new

“inserting in the middle”, “sorting”, … –> std::list

```
using elem_type = my_int*;
using container = 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 = my_int;
// ...

void test(int n, int k)
{
  // ...
  ~~auto ptr_cmp = [](const elem_type& l, const elem_type& r)~~
                  ~~{ return *l < *r; };~~
  // ...
}
```

1.5x-2x speedup

Change 2

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

```
using container=std::vector<elem_type>;

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

```
void test(int n, int k)
{
  // create container of 'n' random values
  auto values = init_random(n, k);

  // sort container
  sort(begin(values), end(values));

  // ...
}
```

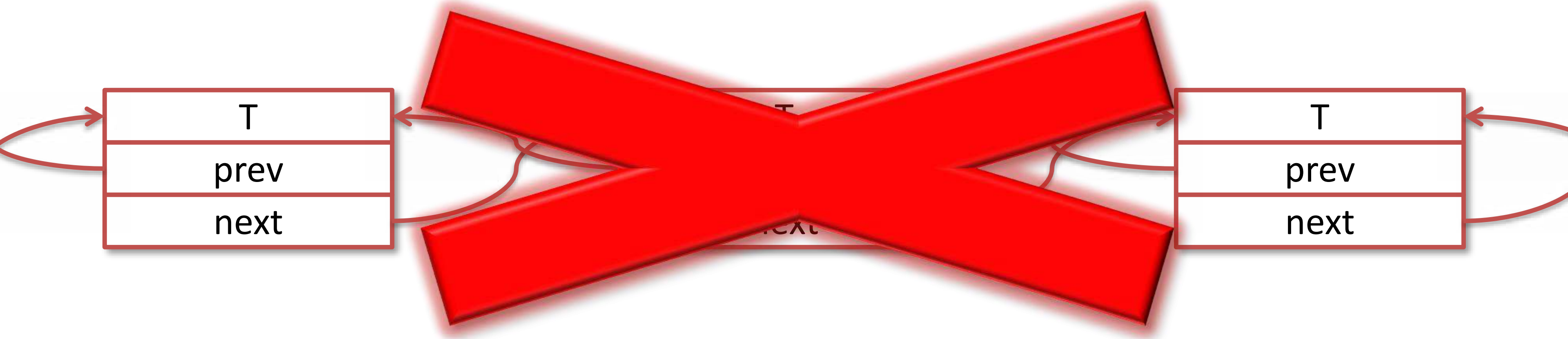
10x-18x speedup. Why?

Algorithmic point of view

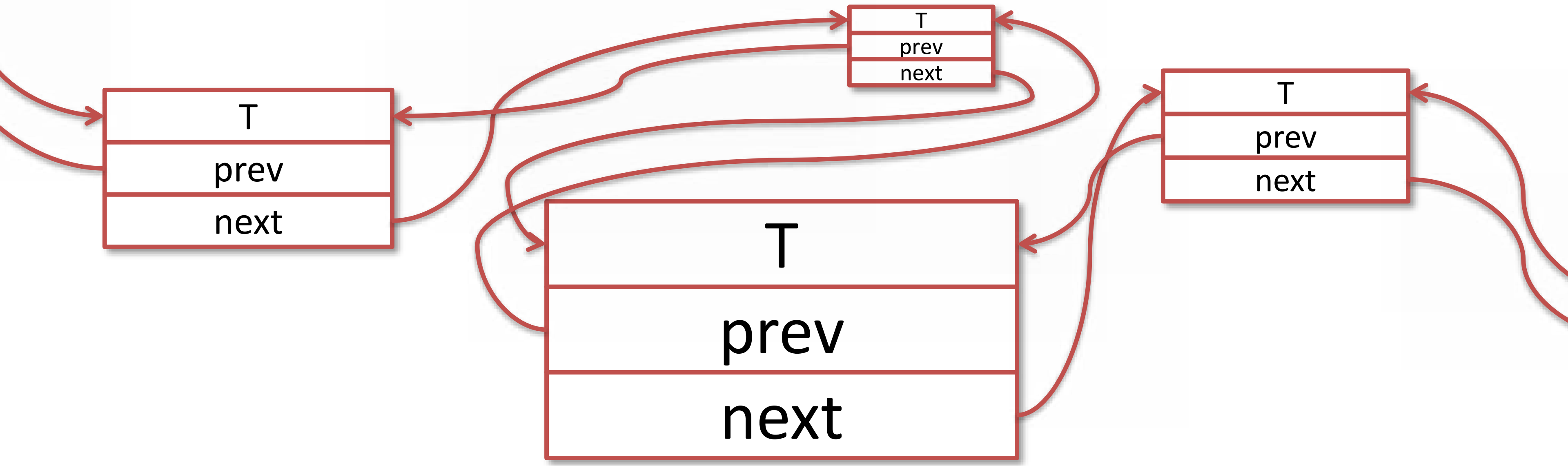
Operation	std::list<T>	std::vector<T>
create container of 'n' random values	O(n)	O(n)
sort container	O(nlogn)	O(nlogn)
'k' times insert a new random value keeping the container sorted	O(k * (n+k))	O(k * (n+k))
verify if sorted	O(n+k)	O(n+k)
Overall complexity	**O(k * (n+k))**	**O(k * (n+k))**

10x-18x speedup. Why?

std::list<T>



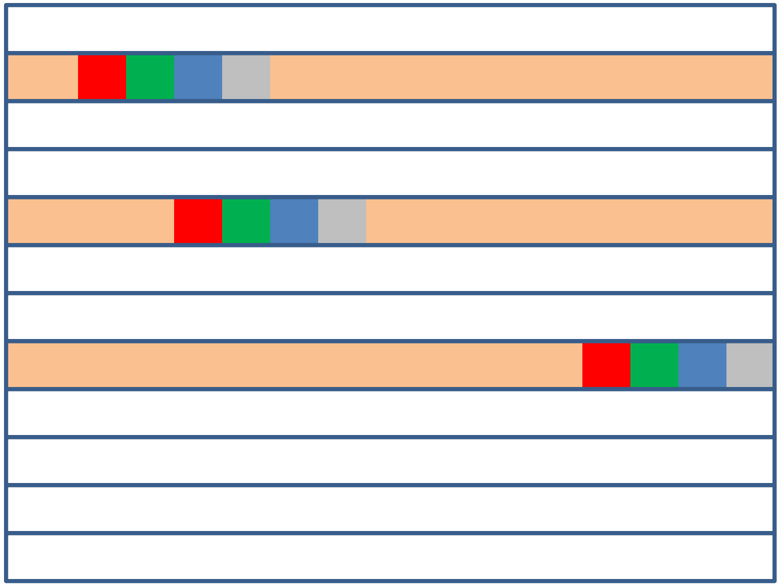
std::list<T>

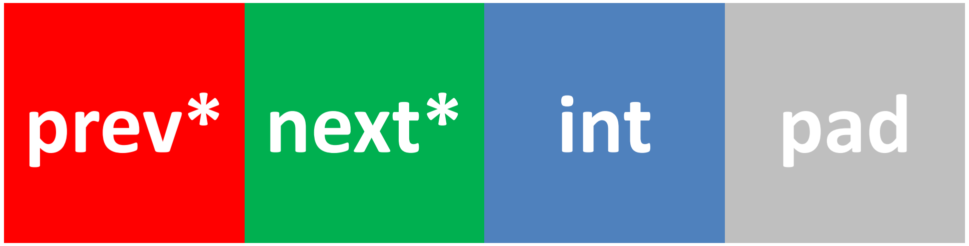


- **easy to insert new elements**
- **easy to erase existing elements**
- **easy to reorder elements**
- **bidirectional iteration**
- **memory ineffective**
  - for small T large memory overhead
  - a lot of dynamic memory allocations and deallocations
  - a lot of pointer dereferences during iteration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

std::list<int> iteration

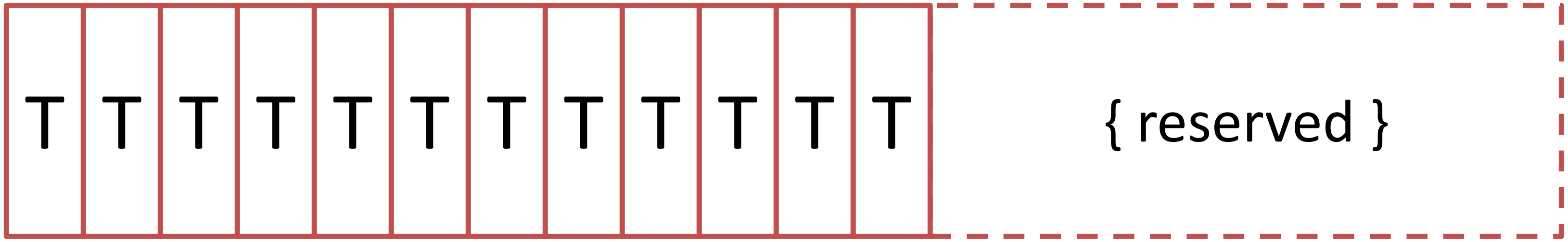
CPU cache





M
E
M
O
R
Y

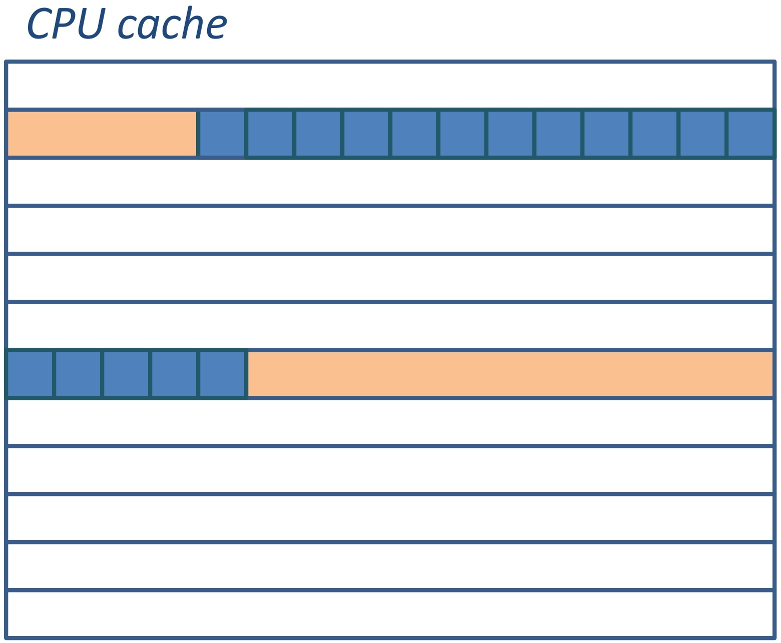
std::vector<T>



T T T T T T T T T T T T { reserved }

- **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!

M E M O R Y

A new possibility for algorithmic optimization

```
void test(int n, int k)
{
  // create container of 'n' random values
  // sort container
  // k times insert a new random value keeping the container sorted
  // verify if sorted
}
```

§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

...

3 ***Complexity:*** At most log2(*last* – *first*) + O(1) comparisons.

Change 3

Know and use C++ algorithms

```cpp
void test(int n, int k)
{
  // ...
  // k times insert a new random value keeping the container sorted
  for(int i=0; i<k; ++i) {
    const my_int val{generate()};
    values.emplace(lower_bound(begin(values), end(values), val),
                   val);
  }
  // ...
}
```

1x-1.6x speedup

Whole code

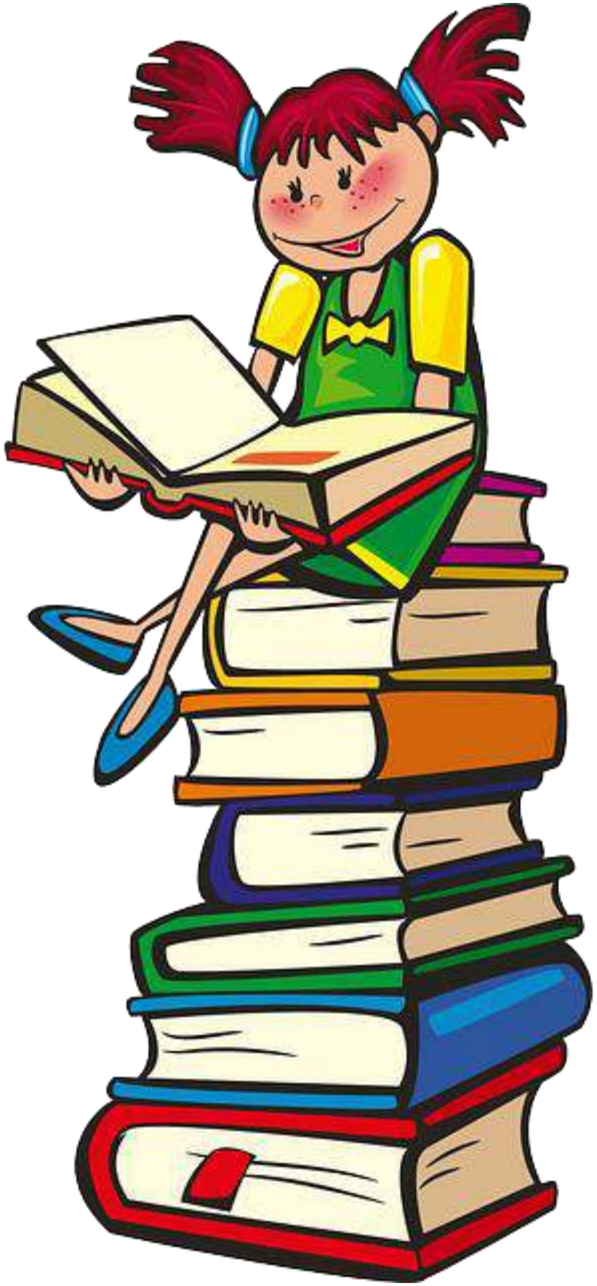
```cpp
std::vector<my_int> init_random(int n, int k)
{
  std::vector<my_int> values;
  values.reserve(n + k);
  for(int i=0; i<n; ++i)
    values.emplace_back(generate());
  return values;
}

void test(int n, int k)
{
  auto values = init_random(n, k);
  sort(begin(values), end(values));

  // k times insert a new random value keeping the container sorted
  for(int i=0; i<k; ++i) {
    const my_int val{generate()};
    values.emplace(lower_bound(begin(values), end(values), val), val);
  }

  std::cout << (is_sorted(begin(values), end(values)) ? "PASS\n" : "FAIL\n");
}
```

Isn't it a great example of efficient programming?

TAKE AWAYS

Learn C++ to be a more efficient programmer

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



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

Use tools provided by the C++ standard.

Do not reinvent the wheel!!!

*

Remember that memory access is the bottleneck of many today's applications

- limit pointers usage
- write cache friendly code

Photograph by Mauro Sartori

Read about
Copy Elision,
Move Semantics,
Perfect Forwarding,
Constant Expressions
to write faster
C++ code

Questions?

Thank you

Happy coding!!!

Backup

```
std::function<int()> make_random_int_generator(
    int min = std::numeric_limits<int>::min(),
    int max = std::numeric_limits<int>::max(),
    std::mt19937::result_type seed = std::mt19937::default_seed)
{
  std::mt19937 gen{seed};
  std::uniform_int_distribution<int> distr{min, max};
  return [=]() mutable { return distr(gen); };
}


int generate()
{
  static auto generator = make_random_int_generator();
  return generator();
}
```