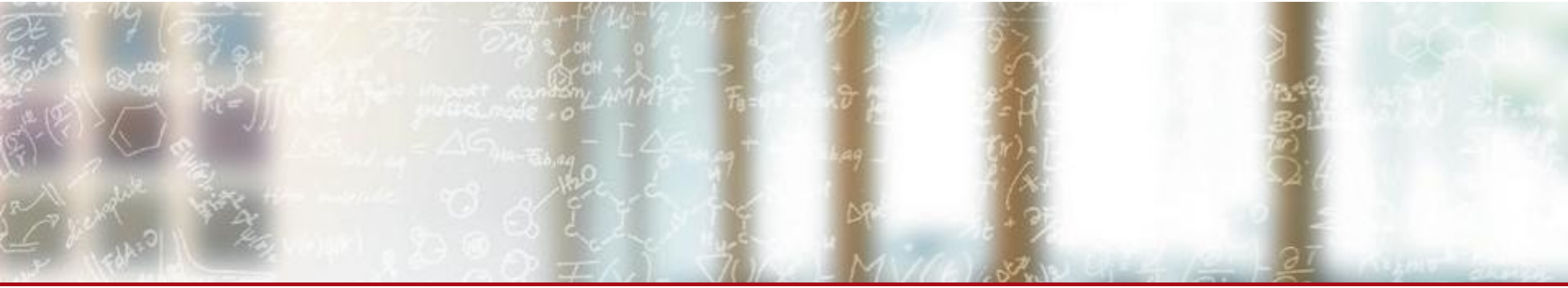
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Advanced C++ for HPC: STL

Anton Afanasyev, CSCS (afanasyev@cscs.ch)
2-4 October, 2017

Resources:

Definition

- STL stands for Standard Template Library
- Originally was designed by Alexander Stepanov
- The spec and implementation went to public in 1994
- Became one of the key stones of the C++ Standard Library
- Consists of containers, algorithms, iterators, functors and related stuff

Design principles

- Concepts serve as generalized interfaces
  - Each STL (template) class implements a concept or several concepts
  - For each STL template class parameter it is defined what concept(s) it should satisfy
- Orthogonal and extensible
  - Iterators are concepts
  - Containers implement iterators
  - Algorithms operate on iterators
  - Applying algorithms to containers (or parts of them) creates combinatorial number of usage possibilities
- No runtime overhead
- Everything is thread compatible but not thread safe

Iterator Concept

Iterators should behave as pointers within C-array

```
int data[] = { 1, 2, 3, 4 };
for (auto i = data; i != data + 4; ++i)
    std::cout << *i << "\n";
```

Iterator Concepts

Iterators should behave as pointers within C-array

```
std::vector data = { 1, 2, 3, 4 };
for (auto i = data.begin(); i != data.end(); ++i)
    std::cout << *i << "\n";
```

Iterator Concepts

Iterators should behave as pointers within C-array

```
std::vector data = { 1, 2, 3, 4 };
for (auto i = data.begin(); i != data.end(); ++i)
    std::cout << *i << "\n";
```

`++i, i++`	move forward
`--i, i--`	move backward
`i+5, i-5, i-j, i+=5, i-=5`	move anywhere
`*i`	dereference
`i == j, i != j`	compare
`i = j`	(cheap) copy

Iterator Concepts

Iterators should behave as pointers within C-array

```
std::vector<int> data = { 1, 2, 3, 4 };
for (auto i = data.begin(); i != data.end(); ++i)
    std::cout << *i << "\n";
```

`++i, i++`	move forward
`--i, i--`	move backward
`i+5, i-5, i-j, i+=5, i-=5`	move anywhere
`*i`	dereference
`i == j, i != j`	compare
`i = j`	(cheap) copy

RandomAccessIterator

Iterator Concepts

Iterators should behave as pointers within C-array

```
std::vector<int> data = { 1, 2, 3, 4 };
for (auto i = data.begin(); i != data.end(); ++i)
    std::cout << *i << "\n";
```

`++i, i++`	move forward
`--i, i--`	move backward
`i+5, i-5, i-j, i+=5, i-=5`	move anywhere
`*i`	dereference
`i == j, i != j`	compare
`i = j`	(cheap) copy

BidirectionalIterator

Iterator Concepts

Iterators should behave as pointers within C-array

```
std::vector<int> data = { 1, 2, 3, 4 };
for (auto i = data.begin(); i != data.end(); ++i)
    std::cout << *i << "\n";
```

`++i, i++`	move forward
`--i, i--`	move backward
`i+5, i-5, i-j, i+=5, i-=5`	move anywhere
`*i`	dereference
`i == j, i != j`	compare
`i = j`	(cheap) copy

ForwardIterator

Iterator Concepts

Iterators should behave as pointers within C-array

```
std::vector<int> data = { 1, 2, 3, 4 };
for (auto i = data.begin(); i != data.end(); ++i)
    std::cout << *i << "\n";
```

`++i, i++`	move forward, **but only once**
`--i, i--`	move backward
`i+5, i-5, i-j, i+=5, i-=5`	move anywhere
`*i`	dereference, **but only for reading**
`i == j, i != j`	compare
`i = j`	(cheap) copy

InputIterator

Iterator Concepts

Iterators should behave as pointers within C-array

```
std::vector<int> data = { 1, 2, 3, 4 };
for (auto i = data.begin(); i != data.end(); ++i)
    std::cout << *i << "\n";
```

`++i, i++`	move forward, **but only once**
`--i, i--`	move backward
`i+5, i-5, i-j, i+=5, i-=5`	move anywhere
`*i`	dereference, **but only for writing**
`i == j, i != j`	compare
`i = j`	(cheap) copy

OutputIterator

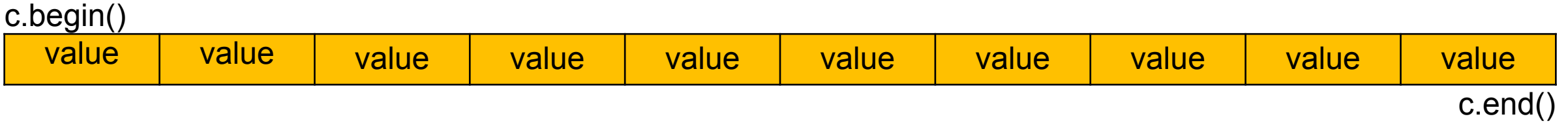
Functors – yet another concept

Functors should behave like pointers to functions

`f(42)`	callable
`g = f`	(cheap) copy
	an invocation doesn't change the functor

Container Concept (simplified)

Container owns a sequence of values and provides access to them via iterators



c.size(), c.empty() etc

Container Concept (possible alternative)

Container owns a sequence of values and provides access to them via iterators

begin(c)

end(c)

size(c), empty(c) etc

SequenceContainers

`std::array<T, N>`	static contiguous array
`std::vector<T>`	dynamic contiguous array
`std::deque<T>`	double-ended queue
`std::forward_list<T>`	singly-linked list
`std::list<T>`	doubly-linked list

std::array

is like a C-array but better

```
// can be returned from a function
std::array<int, 42> foo();

// can act as a tuple
using obj_t = std::array<int, 3>;
obj_t obj = { 3, 4, 5 };
auto second = std::get<1>(obj);
static_assert(std::tuple_size<obj_t>{} == 3, "");
static_assert(std::is_same<std::tuple_element<0, obj_t>::type, int>{}, "");

// is 'true' sequence container
std::sort(obj.begin(), obj.end());
```

std::vector

Consists of a pointer to the buffer, logical size and capacity of the buffer

When capacity is not enough to fit newly inserted elements, a new buffer is allocated with doubled capacity; the data is moved into the new buffer and the old data is released.

```
std::vector<int> a;
for (int i = 0; i != N; ++i) a.push_back(i);
```

How many times each element is moved on average?

vector

Consists of a pointer to the buffer, logical size and capacity of the buffer

When capacity is not enough to fit newly inserted elements, a new buffer is allocated with doubled capacity; the data is moved into the new buffer and the old data is released.

```
std::vector<int> a;
for (int i = 0; i != N; ++i) a.push_back(i);
```

How many times each element is moved on average? **2**

std::vector: Raw Access

```
size_t get_results_size();
void get_results(size_t size, struct Result* buf);

std::vector<Result> wrapper() {
    std::vector<Result> res(get_results_size());
    get_results(res.size(), res.data());
    return res;
}
```

std::vector<bool>

possibly space-efficient specialization

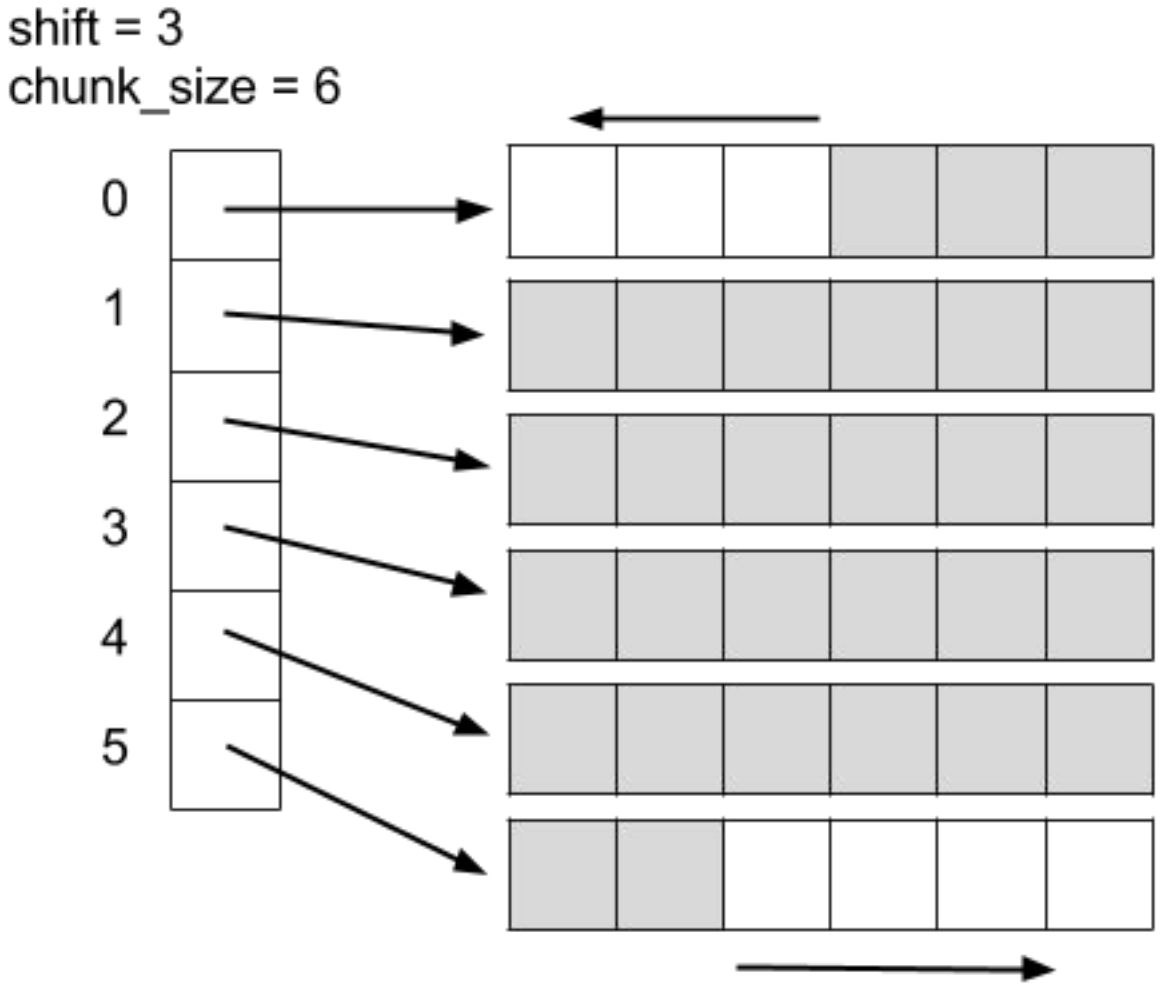
- elements are not stored in contiguous buffer
- not thread compatible
- was added to the library as a proof of concept for template specialization and proxy pattern

avoid using it without a solid reason

std::deque

data is in chunks; chunks are referenced from TOC

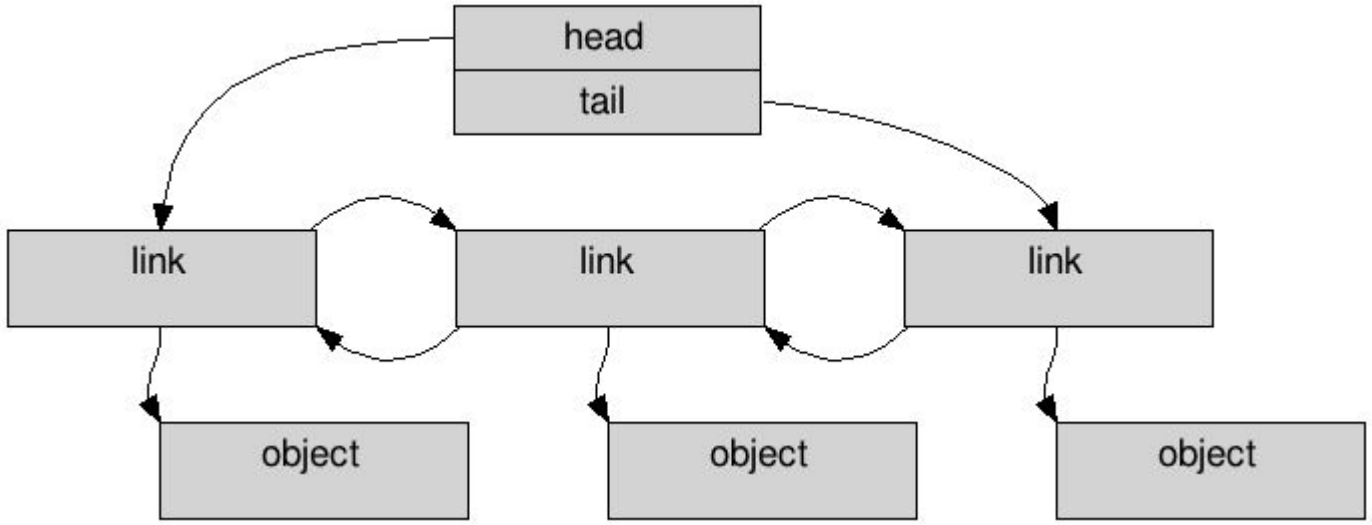
- effective push_front/push_back without invalidating refs and iterators
- random access requires two derefs



std::list

doubly-linked list; size is cashed

- fast insertions/deletions, has splice()
- no random access
- refs and iterators remains valid even when splicing between list
- some algorithms are implemented as methods



Sequence containers: what to choose

- use `std::vector` as a default – it is the fastest and consumes less memory
- use `std::deque` if you are implementing some sort of queue (pop_front/push_front)
- use `std::list` if performance doesn't matter much but your design requires references and iterator to be always valid.

AssociativeContainer

- `std::set<T>`
- `std::map<K, V>`
- `std::multiset<T>`
- `std::miltimap<K, V>`
- `std::unordered_set<T>`
- `std::unordered_map<K, V>`
- `std::unordered_multiset<T>`
- `std::unordered_multimap<K, V>`

AssociativeContainer

- `std::set<T, Comp>`
- `std::map<K, V, Comp>`
- `std::multiset<T, Comp>`
- `std::miltimap<K, V, Comp>`
- `std::unordered_set<T>`
- `std::unordered_map<K, V>`
- `std::unordered_multiset<T>`
- `std::unordered_multimap<K, V>`

AssociativeContainer

- `std::set<T, Comp>`
- `std::map<K, V, Comp>`
- `std::multiset<T, Comp>`
- `std::miltimap<K, V, Comp>`
- `std::unordered_set<T, Hash, Eq>`
- `std::unordered_map<K, V, Hash, Eq>`
- `std::unordered_multiset<T, Hash, Eq>`
- `std::unordered_multimap<K, V, Hash, Eq>`

AssociativeContainer requirements

- if you use non-default `Compare` functor, verify that it is sane
- values in `std::set` should be virtually immutable

insert vs operator[] vs at

```
std::map<std::string, int> m = {{"a", 1}, {"b", 2}, {"c", 3}};
m.insert({"a", 11});
m.at("b") = 22;
m["c"] = 33;
m["d"] = 44;
// m.at("e") = 55;
for(auto && v : m)
  std::cout << v.first << ":" << v.second << ”\n";
```

insert vs operator[] vs at

```
std::map<std::string, int> m = {{"a", 1}, {"b", 2}, {"c", 3}};
m.insert({"a", 11});
m.at("b") = 22;
m["c"] = 33;
m["d"] = 44;
// m.at("e") = 55;
for(auto && v : m)
  std::cout << v.first << ":" << v.second << "\n";
```

```
a:1
b:22
c:33
d:44
```

insert vs operator[] vs at

```
std::map<std::string, int> m = {{"a", 1}, {"b", 2}, {"c", 3}};
m.insert({"a", 11});
m.at("b") = 22;
m["c"] = 33;
m["d"] = 44;
m.at(”e") = 55;
for(auto && v : m)
  std::cout << v.first << ":" << v.second << ”\n";
```

```
std::out_of_range: map::at:  key not found
```

Algorithms

- `std::sort`
- `std::stable_sort`
- `std::transform`
- `std::merge`
- `std::partition`
- `std::accumulate`
- `std::iota`
- `std::remove`
- …

using `std::remove`

```
std::vector v = {1, 2, 3, 4};
std::remove(v.begin(), v.end(), 2);
assert(v.size() == 3);
```

using `std::remove`

```
std::vector v = {1, 2, 3, 4};
v.erase(std::remove(v.begin(), v.end(), 2), v.end());
assert(v.size() == 3);
```

How to use `std::swap`?

```
struct A {
    bool val;
    friend void swap(A&, A&) { std::cout << "I refuse to swap!\n"; }
};

int main() {
    bool x = false, y = true;
    std::swap(x, y);
    assert(x);
    A a{false}, b{true};
    std::swap(a, b);
    assert(!a.val);
}
```

How to use `std::swap`?

```
struct A {
    bool val;
    friend void swap(A&, A&) { std::cout << "I refuse to swap!\n"; }
};

int main() {
    using std::swap;
    bool x = false, y = true;
    swap(x, y);
    assert(x);
    A a{false}, b{true};
    swap(a, b);
    assert(!a.val);
}
```

`std::queue, std::stack, std::priorityqueue`

- they are container adaptors (use other containers underneath)
- `std::queue`: FIFO, push into one end and pop from the other
- `std::stack`: FILO, push and pop into/from one end
- `std::priority_queue`
  - ordered container, get the largest value in constant time
  - slow insertion/removal (logarithmic)

What is default container for those adaptors?
Why pop() returns void?

New Features: `std::initializer_list` Support

```
std::vector<int> eval(std::vector<int> src) {
    // do smth with src.
    return src;
}

int main() {
    // automatic conversion to the right type
    std::vector<double> a = {1., true, 3};
    // assign operator also works with IL
    a = {};
    // IL can contain any expressions. The execution order is defined.
    int i = 0;
    std::vector v = {++i, ++i, ++i};
    for(auto x : v) std::cout << x << "\n";
    // nested IL for nested containers work
    std::set<std::vector<int>> b = {
        {},
        {1, 2},
        {3}
    };
    // IL ctors are implicit. If function param is container, IL can be used as an args.
    auto c = eval({1, 2, 3});
}
```

New Features: Move Semantics Support

- Container element type requirements are soften from copyable to movable
- If the type is not even movable, it is still possible to store it in STL containers by wrapping into `std::unique_ptr`.
- There is no more reason to store raw pointers in containers anymore.

New Features: Range Based Loop Syntax

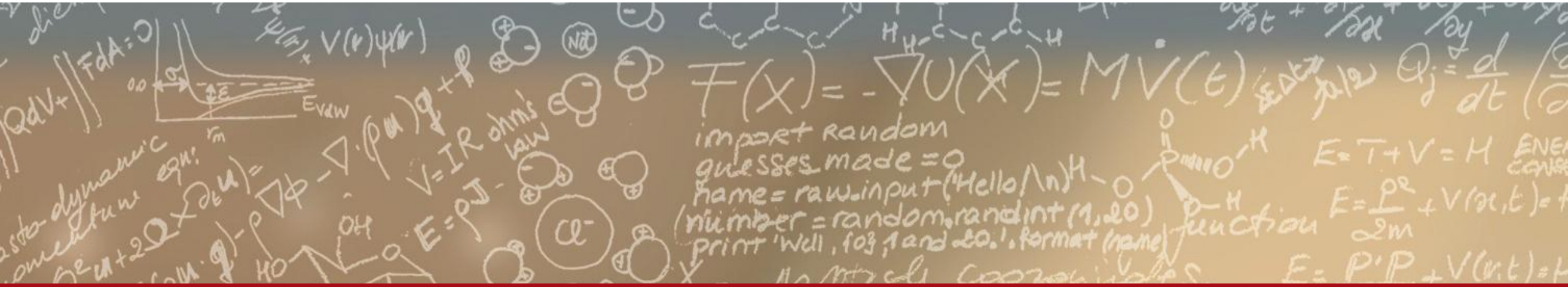
- this language feature is tailored to use with STL by design
- it implicitly defines a new concept
- makes simple algorithms like `std::for_each` and `std::transform` useless

New Features: Perfect Forwarding

all insertion methods now have emplace variations

```
struct A {
    A(int a, int b, int c) {}
};

int main() {
    std::vector<A> v;
    v.push_back(A{1, 2, 3});
    v.emplace_back(1, 2, 3);
}
```



Thank you for your attention.