Library study group

Camiel Verdult & Marnix Laar

What did we do for this topic?

We went to the Library

Studied


That's all Folks!

All jokes aside, what did we do?

Standard Library? What the heck is that?

Standard Library? What the heck is that?

std::string

std::cout

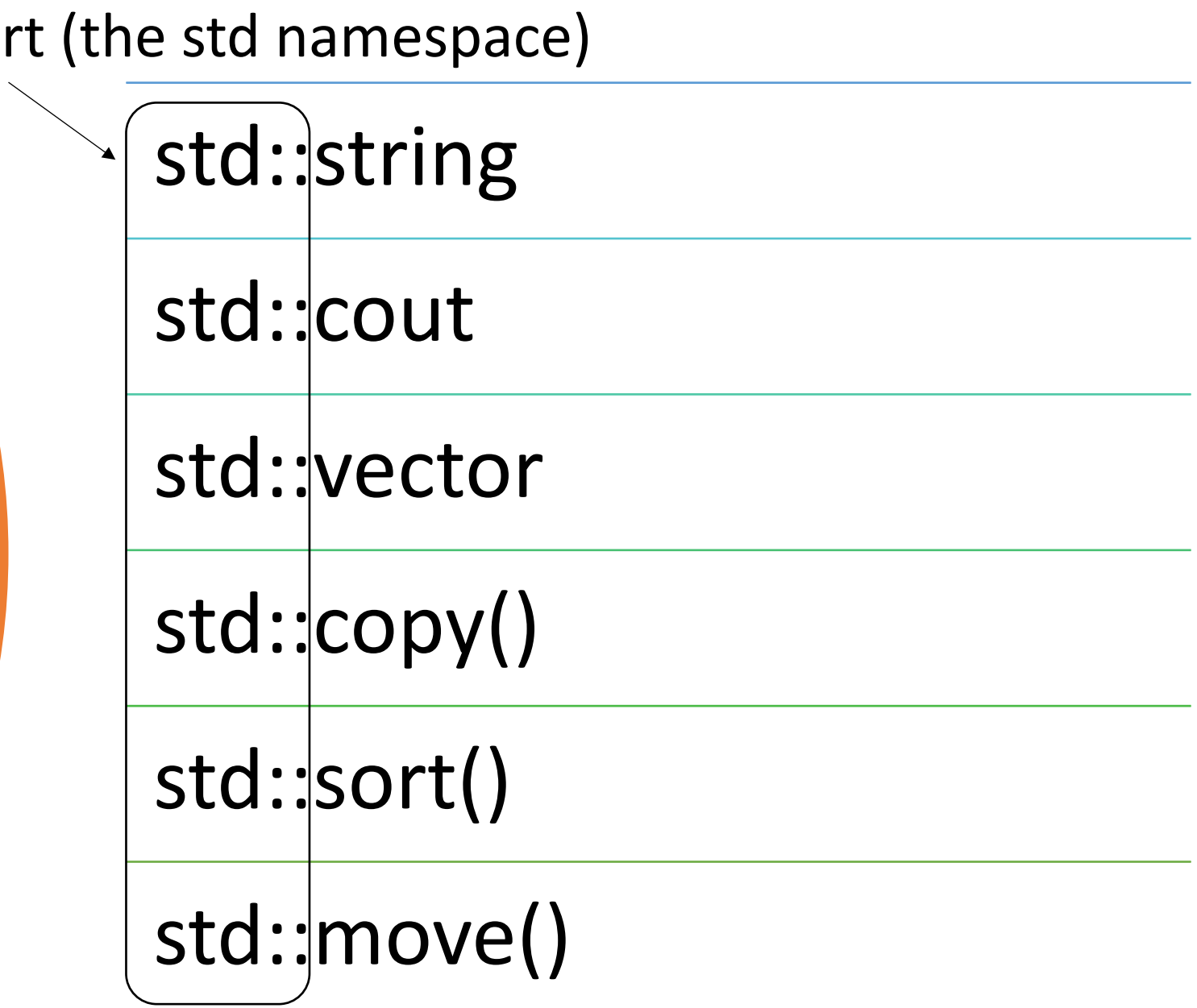
std::vector

std::copy()

std::sort()

std::move()

Standard Library? What the heck is that?



Who makes the standard library?

- ISO standardized
- Dictates:
  - The declaration and expected behavior
  - The side effects and conditions
- Implementations' underlying code may differ

Example from §21.4.2:

```
basic_string(const basic_string& str,
             size_type pos, size_type n = npos,
             const Allocator& a = Allocator());
```

Requires: `pos <= str.size()`

Throws: `out_of_range` if `pos > str.size()`.

Effects: Constructs an object of class `basic_string` and determines the effective length `rlen` of the initial string value as the smaller of `n` and `str.size() - pos`, as indicated in Table 65.

Standard Library – implementations

- GNU C++ Standard Library (libstdc++)
- LLVM C++ Standard Library (libc++) (one of the most popular, comes with clang and mingw)
- NVIDIA C++ Standard Library (libcu++)
- Microsoft C++ Standard Library (MSVC STL)
- Electronic Arts Standard Template Library (EASTL)
- Abseil (Google)
- Folly (Facebook)

LLVM Standard Library vector implementation

(aka template hell)

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
  class vector : protected _Vector_base<_Tp, _Alloc>
  {
#ifdef _GLIBCXX_CONCEPT_CHECKS
    ...
#endif

#if __cplusplus >= 201103L
    static_assert(is_same<typename remove_cv<_Tp>::type, _Tp>::value,
    "std::vector must have a non-const, non-volatile value_type");
# if __cplusplus > 201703L || defined __STRICT_ANSI__
    static_assert(is_same<typename _Alloc::value_type, _Tp>::value,
    "std::vector must have the same value_type as its allocator");
# endif
#endif

    typedef _Vector_base<_Tp, _Alloc>         _Base;
    typedef typename _Base::_Tp_alloc_type        _Tp_alloc_type;
    typedef __gnu_cxx::__alloc_traits<_Tp_alloc_type> _Alloc_traits;

  public:
    typedef _Tp                   value_type;
    typedef typename _Base::pointer           pointer;
    typedef typename _Alloc_traits::const_pointer const_pointer;
    typedef typename _Alloc_traits::reference     reference;
    typedef typename _Alloc_traits::const_reference   const_reference;
    typedef __gnu_cxx::__normal_iterator<pointer, vector> iterator;
    typedef __gnu_cxx::__normal_iterator<const_pointer, vector>
    const_iterator;
```

```
  Creates a %vector with no elements.
#if __cplusplus >= 201103L
      vector() = default;
#else
      ...
#endif

      Creates a %vector with no elements.

      Parameters
          __a An allocator object.

      explicit
      vector(const allocator_type& __a) _GLIBCXX_NOEXCEPT
      : _Base(__a) { }

#if __cplusplus >= 201103L
      Creates a %vector with default constructed elements.

      Parameters
          __n The number of elements to initially create.
          __a An allocator.

      This constructor fills the %vector with __n default constructed elements.
      explicit
      vector(size_type __n, const allocator_type& __a = allocator_type())
      : _Base(_S_check_init_len( n: __n,  a: __a), __a)
      { _M_default_initialize( n: __n); }
```

Looking at the std::vector implementation

Our own (readable) vector “implementation”

```cpp
namespace mc {

    template <typename T>
    class vector {

    public:
        // It’s a good idea to define those. Others can then easily probe a vector object in generic code.
        using value_type = T;
        using pointer = T*;
        using reference = T&;
        using const_reference = const T&;

        vector(std::size_t capacity):
                m_data{ new T[capacity]},
                m_cap{ capacity },
                m_sz{ 0 } {}

        vector(): vector(DEFAULT_CAP) {};

        // Copy constructor
        vector(const vector& other): m_data{ new T[other.capacity()]},
                                     m_cap{ other.size() },
                                     m_sz{ 0 } {}

        ~vector(){
            delete[] m_data;
        }

        [[maybe_unused]] std::size_t capacity() {
            return m_cap;
        }

        [[maybe_unused]] std::size_t size() {
            return m_sz;
        }

        [[maybe_unused]] [[nodiscard]] std::size_t capacity() const {
            return m_cap;
        }

        [[maybe_unused]] [[nodiscard]] std::size_t size() const {
            return m_sz;
        }
```

Our own (readable) pair “implementation”

```cpp
// marnix camiel namespace
namespace mc {

    template <typename T1, typename T2>
    class pair {

    public:
        // It’s a good idea to define those. Others can then easily probe a vector object in generic code.
        using first_type = T1;
        using first_pointer = T1*;
        using first_reference = T1&;
        using first_const_reference = const T1&;

        using second_type = T2;
        using second_pointer = T2*;
        using second_reference = T2&;
        using second_const_reference = const T2&;

        // Empty constructor
        pair()= default;

        // Default constructor
        pair(T1 first, T2 second): m_first {first}, m_second {second} {}

        // Copy constructor
        pair(const pair& other): pair{other.first(), other.second()} {}

        // Copy assignment operator
        pair& operator=(pair other) {
            m_first = other.first();
            m_second = other.second();
            return *this;
        }

        // > compare operator
        bool operator>(pair other) {
            if (m_first == other.first()) {
                return m_second > other.second();
            }
            return m_first > other.first();
        }
```

Our own (readable) map “implementation”

```cpp
namespace mc {

    template<typename TKey, typename TValue>
    class map { // This whole class is a wrapper around an mc::vector<mc::pair>
    public:

        using first_type = TKey;
        using first_pointer = TKey*;
        using first_reference = TKey&;
        using first_const_reference = const TKey&;

        using second_type = TValue;
        using second_pointer = TValue*;
        using second_reference = TValue&;
        using second_const_reference = const TValue&;

        using pair_template = mc::pair<TKey, TValue>;
        using vector_template = mc::vector<pair_template>;

        // Default constructor
        [[maybe_unused]] explicit map() : m_vector {} {}

        // Copy constructor
        map(const map &other) : m_vector {other.raw()} {}

        [[maybe_unused]] std::size_t capacity() {
            /* This function returns the capacity of the underlying vector */
            return m_vector.capacity();
        }

        [[maybe_unused]] std::size_t size() {
            /* This function returns the size of the underlying vector */
            return m_vector.size();
        }

        [[maybe_unused]] [[nodiscard]] std::size_t capacity() const {
            /* This function returns the capacity of the underlying vector as a const */
            return m_vector.capacity();
        }

        [[maybe_unused]] [[nodiscard]] std::size_t size() const {
            /* This function returns the size of the underlying vector as a const */
            return m_vector.size();
        }
```

Google Tests

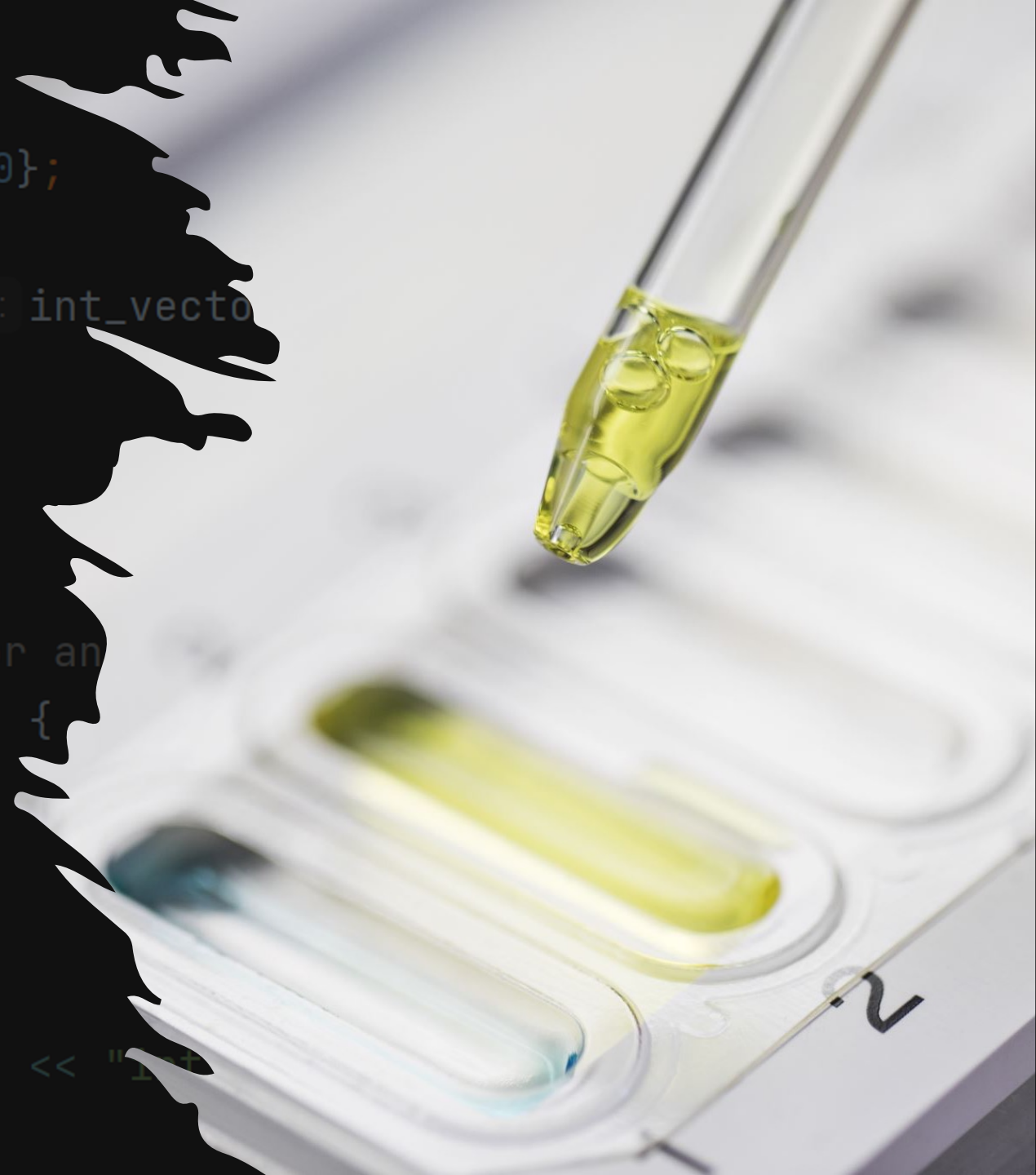
```
TEST(vector, int_array_compare) {
    // Initialise empty vector
    mc::vector<int> int_vector{ capacity: 1000};

    std::iota( first: int_vector.begin(), last: int_vecto

    // Initialise int array
    int numbers[1000];

    // Add 1000 elements to both the vector an
    for (std::size_t i = 0; i < 1000; i++) {
        numbers[i] = i;
    }

    // Make sure their content matches
    ASSERT_EQ(*int_vector.raw(), *numbers) << "I
}
```



Missing functionality

- mc::vector:
  - Initializer list constructor
- mc::map:
  - Red-black tree data storage

Things we have implemented so far

mc::vector:

- begin() and end() for std algorithm compatibility
- sort() as a built-in function
- print() and operator<< for debugging
- most things you might use an std::vector for

mc::pair:

- I think everything defined in the standard?

mc::map:

- Storage datatype is mc::vector instead of Red-Black Tree (would be out of scope for this topic)
- Implementation based on mc::vector