C/C++ PRIMER

LECTURE 9: GENERIC PROGRAMMING AND C++ TEMPLATES

Fabian Wermelinger
Harvard University

OUTLINE

- Standard Template Library (STL)

WHY THE STL?

- Every line of code you do not write does not contain bugs.
- When you write code you often need to sort data of find a minimum/maximum value for example. You should not (repetitively) implement such algorithms.
- The STL is well tested and optimized. It is hard to beat it performance wise.
- The STL is a template library (header only) and can be used with any types you wish.

https://en.wikipedia.org/wiki/Standard_Template_Library

WHAT IS THE STL?

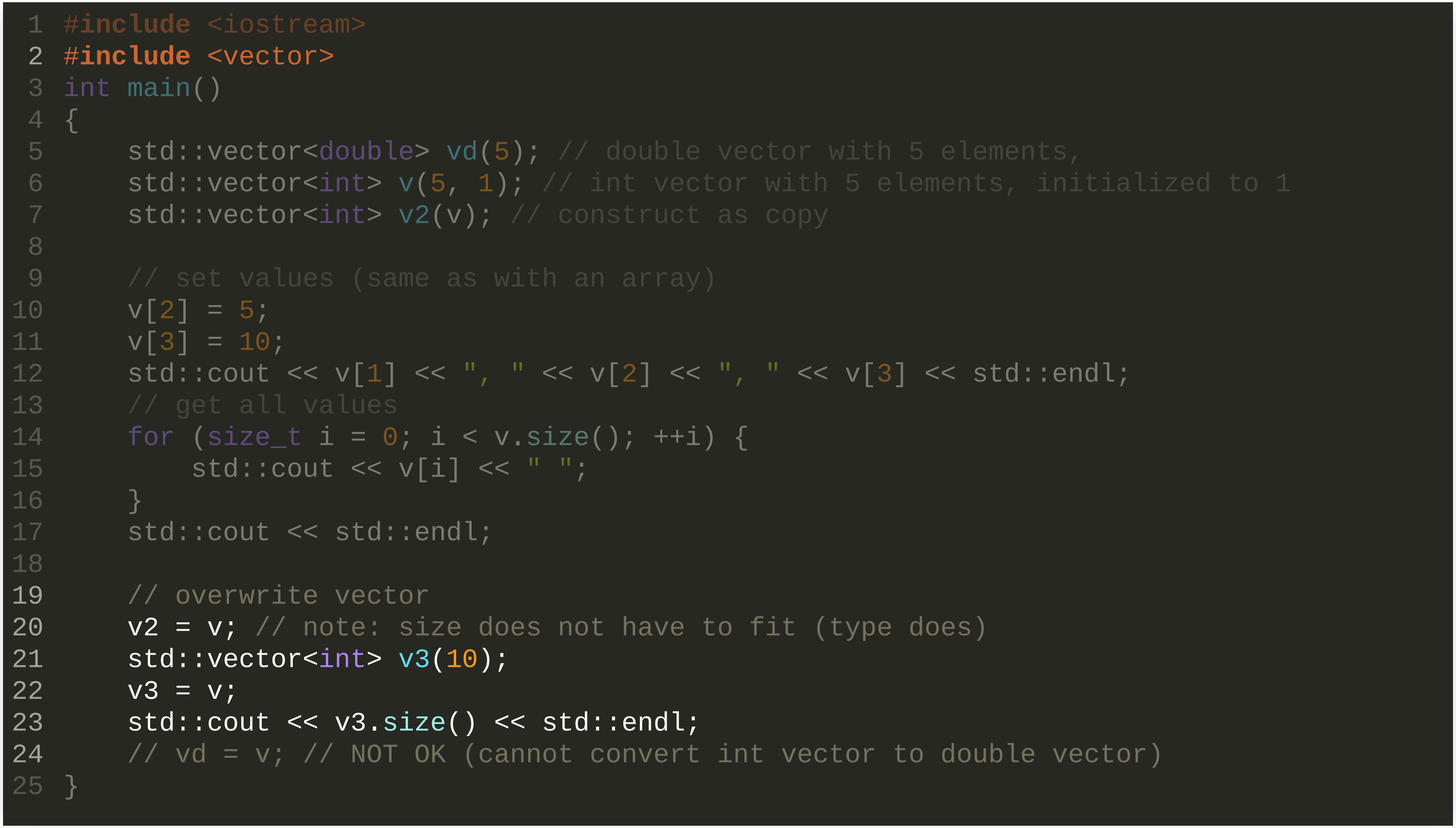
- The STL is a set of general-purpose generic classes and functions provided by `C++`.
- The STL is usually shipped with the compiler, you do not need to download it yourself.
- The STL is based on three components:
  1. ***Containers***: Objects that hold other objects. Examples are `std::array`, `std::vector`, `std::forward_list` (singly-linked list), `std::list` (doubly-linked list), `std::map`, `std::queue` and others.
  2. ***Iterators***: Iterators are a design pattern to iterate over containers without exposing the container internals for element retrieval.
  3. ***Algorithms***: Frequently used algorithms that act on containers using iterators. Examples are sorting, searching or converting data.

CONTAINERS: `std::vector`

- The `std::vector` class is one you will need often.
- It dynamically allocates *coalesced* memory on the heap.
- A `std::vector` has $\mathcal{O}(1)$ random access complexity. What does this mean?
- Any STL container can be copied, overwritten, and removed like any other built-in types (this is a consequence of RAII).

CONTAINERS: std::vector

Examples: https://en.cppreference.com/w/cpp/container/vector



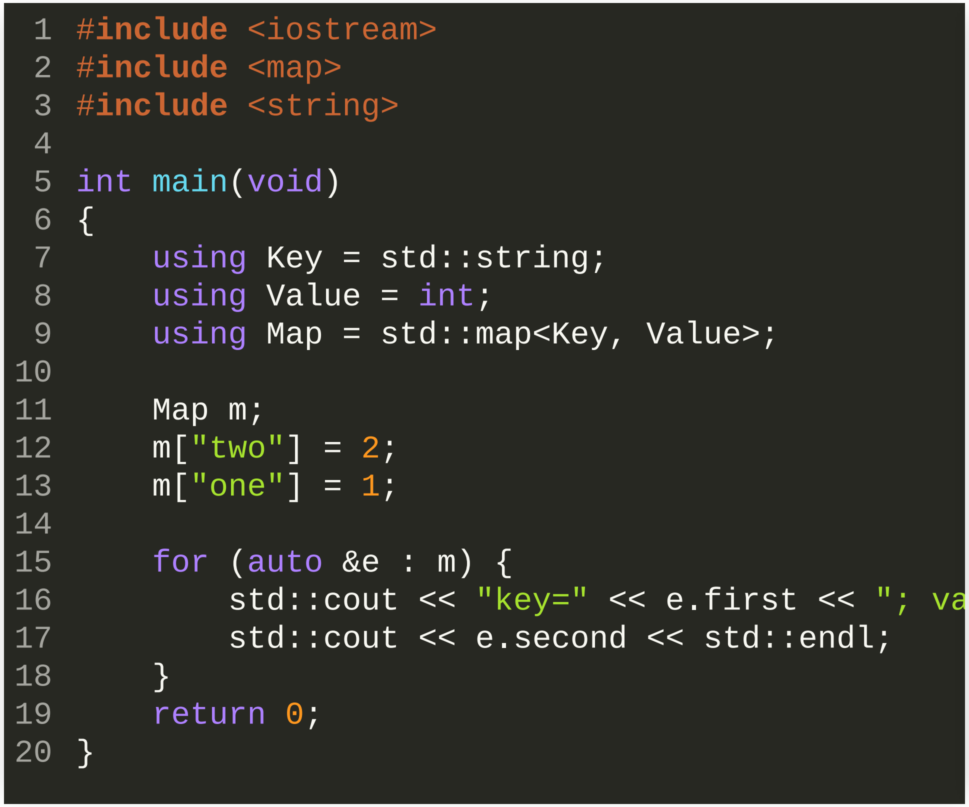
```cpp
#include <iostream>
#include <vector>
int main()
{
    std::vector<double> vd(5); // double vector with 5 elements,
    std::vector<int> v(5, 1); // int vector with 5 elements, initialized to 1
    std::vector<int> v2(v); // construct as copy

    // set values (same as with an array)
    v[2] = 5;
    v[3] = 10;
    std::cout << v[1] << ", " << v[2] << ", " << v[3] << std::endl;
    // get all values
    for (size_t i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;

    // overwrite vector
    v2 = v; // note: size does not have to fit (type does)
    std::vector<int> v3(10);
    v3 = v;
    std::cout << v3.size() << std::endl;
    // vd = v; // NOT OK (cannot convert int vector to double vector)
}
```

CONTAINERS: `std::map`

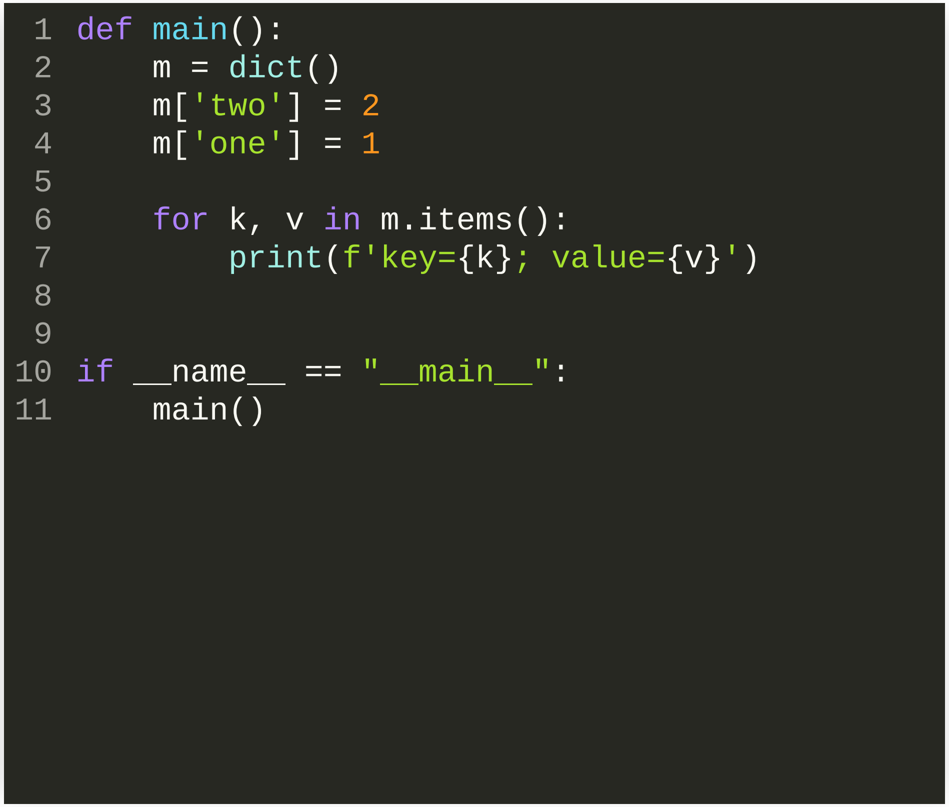
- This is a sorted associative container.
- It behaves similar to a `dict` in `python`
- A `map` is usually implemented as a red-black tree.
- It maps a key of type `Key` to a value of type `T`.

CONTAINERS: std::map

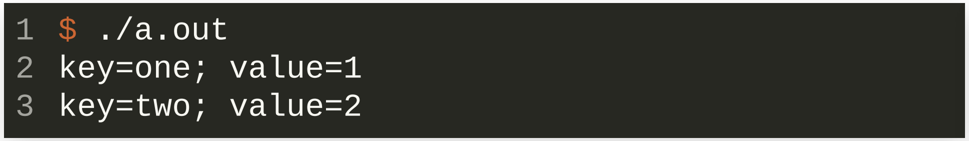
Examples: https://en.cppreference.com/w/cpp/container/map



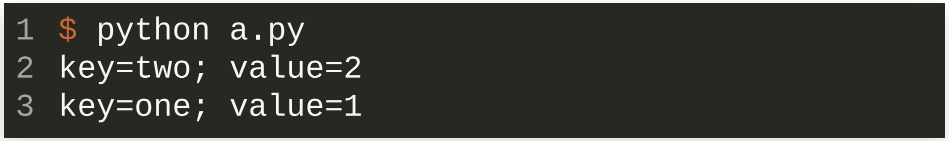
```cpp
#include <iostream>
#include <map>
#include <string>

int main(void)
{
    using Key = std::string;
    using Value = int;
    using Map = std::map<Key, Value>;

    Map m;
    m["two"] = 2;
    m["one"] = 1;

    for (auto &e : m) {
        std::cout << "key=" << e.first << "; va
        std::cout << e.second << std::endl;
    }
    return 0;
}
```



```python
def main():
    m = dict()
    m['two'] = 2
    m['one'] = 1

    for k, v in m.items():
        print(f'key={k}; value={v}')


if __name__ == "__main__":
    main()
```



```
$ ./a.out
key=one; value=1
key=two; value=2
```



```
$ python a.py
key=two; value=2
key=one; value=1
```

MORE STL CONTAINERS

- `std::list`: doubly-linked list. Only *sequential* (linear) access.
- `std::queue`: First-In-First-Out (FIFO) queue. No random access for this data structure.
- `std::deque`: Double ended queue with random access ($\mathcal{O}(1)$ access time). `std::vector` does have `push_front` method but `std::deque` is more general for this purpose
- `std::priority_queue`: Sorted queue. The position of insertion is defined by element values. Usually implemented on top of a `std::vector`.
- `std::stack`: Last-In-First-Out (LIFO) data structure. Implemented on top of a `std::deque`.
- `std::set`: Is an associative container that contains a sorted set of *unique* objects.

ITERATORS

- Not all STL containers allow for $\mathcal{O}(1)$ random access complexity.
- But you can always iterate over the elements in the data structure. For example, in a `std::forward_list` (singly-linked list) we can only iterate in *forward* direction.
- The STL defines iterators, a design pattern in OOP that provides a unique interface for element iteration of container types, hiding the underlying details of how the elements are retrieved. (Element retrieval for a `std::vector` is not the same as for a `std::list`.)
- STL Iterators behave like pointers in `C++`. They can be incremented, some can be decremented like pointers.
- However, the memory layout for some data structures may not be contiguous (e.g. `std::vector` is contiguous, `std::list` is not) but we do not worry about it when working with iterators as all of these details are hidden.

BASIC USAGE OF ITERATORS

- Iterators are `class`es usually nested within the container type:

```cpp
std::vector<int> v(10); // int vector with 10 elements

// loop over v with an iterator `it`
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << std::endl; // iterators behave like pointers!
}
```

- Here the type of the iterator is `std::vector<int>::iterator`. Note that the `iterator` type is nested within the `std::vector` container class template.
- To get the *value* of the iterator, we have to dereference it, same as with pointers: `*it`.
- If the iterator is pointing to the end of the container (`it == v.end()`) we stop iteration.
- It is convention in the STL that `begin()` and `end()` return *iterators* to the beginning and one past the last element of the container, respectively.

BASIC USAGE OF ITERATORS

- Range based `for`-loops were introduced in `C++11` and make our typing-life a bit easier (nested types in class templates means a lot of typing. The `auto` keyword also comes in handy here). In the previous slide we used *raw* iterators, here the range based `for`-loop hides raw iterator usage from user-level code:

```
std::vector<int> v(10); // int vector with 10 elements

// loop over v with an iterator `it` (shorter typing with `auto`)
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << std::endl; // iterators behave like pointers!
}

// range based for-loop (uses iterators under the hood)
for (auto e : v) {
    std::cout << e << std::endl; // we get the dereferenced element in e
}
```

- Range based `for`-loops are convenient and somewhat more aligned with how you write a `for`-loop in `python`. (This does not necessarily mean it is always better to use range based loops.)

DIFFERENT KINDS OF ITERATORS

The basic iterator categories are the following:

1. ***Random access iterators:*** can be incremented/decremented arbitrarily. For example, `i+=3`, `++i`, `--i`. This type of iterator can be used with `std::vector` for example.
2. ***Bidirectional iterators:*** can be incremented or decremented, that is, `++i` and `--i`. An example container for this iterator is `std::list` (doubly-linked list).
3. ***Forward iterators:*** can be incremented only, that is, `++i`. An example for this iterator is `std::forward_list` (singly-lined list).
4. ***Input iterators:*** can be incremented only (`++i`) and is *read-only*. Examples are any STL container that is `const` qualified.

STL ALGORITHMS

https://en.cppreference.com/w/cpp/algorithm

- The STL provides templated implementations of numerous algorithms.
- All algorithms in the STL operate with *iterators* which in turn are defined for the STL container types.
- A selection of these algorithms include:
  - Copying
  - Sorting
  - Searching (sorted and unsorted sequences)
  - Replacing/removing elements
  - Reordering/partitioning
  - Merging sequences
  - Set operations
  - Heap operations
  - Permutations
  - Numeric operations (scan, reductions, inner products)
- You need to include `<algorithm>` in your code.

STL ALGORITHMS

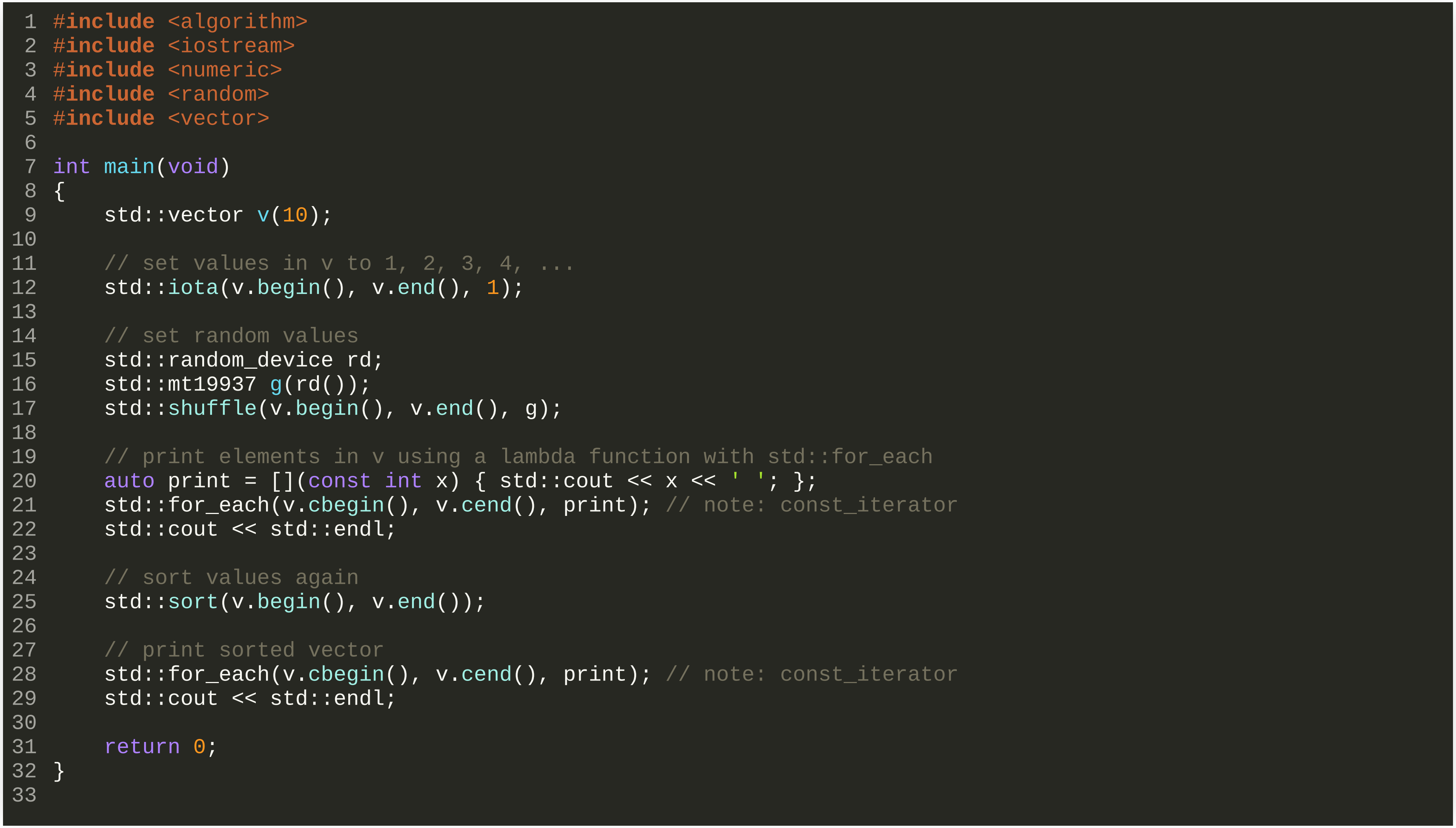
We use the `<algorithm>` standard library in order not to "re-invent" the wheel. The STL is well tested and optimized.

Examples: value initialization, `for_each` transformations, accumulation

```cpp
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>

int main(void)
{
    std::vector<int> v(10);

    // set all values in v to 42
    std::fill(v.begin(), v.end(), 42);

    // set values in v to 0, 1, 2, 3, ...
    std::iota(v.begin(), v.end(), 0);

    // print elements in v using a lambda function with std::for_each
    auto print = [](const int x) { std::cout << x << '\n'; };
    std::for_each(v.cbegin(), v.cend(), print); // note: const_iterator

    // increment elements in v using a lambda function with std::for_each
    auto incr = [](int &x) { ++x; };
    std::for_each(v.begin(), v.end(), incr); // can not use const_iterator here

    // accumulate the values in v
    const int sum = std::accumulate(v.begin(), v.end(), 0);

    return 0;
}
```

STL ALGORITHMS

Examples: random shuffles, sorting



```cpp
#include <algorithm>
#include <iostream>
#include <numeric>
#include <random>
#include <vector>

int main(void)
{
    std::vector v(10);

    // set values in v to 1, 2, 3, 4, ...
    std::iota(v.begin(), v.end(), 1);

    // set random values
    std::random_device rd;
    std::mt19937 g(rd());
    std::shuffle(v.begin(), v.end(), g);

    // print elements in v using a lambda function with std::for_each
    auto print = [](const int x) { std::cout << x << ' '; };
    std::for_each(v.cbegin(), v.cend(), print); // note: const_iterator
    std::cout << std::endl;

    // sort values again
    std::sort(v.begin(), v.end());

    // print sorted vector
    std::for_each(v.cbegin(), v.cend(), print); // note: const_iterator
    std::cout << std::endl;

    return 0;
}

```

STL CONCLUSION

- The STL offers many commonly used data structures and algorithms.
- The implementation in the STL is efficient.
- Do not re-invent the wheel. The STL is used everywhere, well tested and has a large community behind it, most of these people are profession `C++` developers.
- There is much more functionality in the STL that we can not cover here due to its vast size.
- See https://en.cppreference.com/w/cpp
  - Strings library
  - Containers library
  - Iterator library
  - Algorithms library
  - Numerics library

HANDS-ON: STL

Goal: we want to find an element in a vector of `std::string`'s and add a new string with a different value than the ones in the vector *before* it.

Steps:

- `std::string` allows for comparison with `==`.
- Setup a small `std::vector` using `std::string` and initialize the strings to identical values. Set one element in the vector to a different value.
- Check https://en.cppreference.com/w/cpp/algorithm for something that you can use to *find* elements in a container.
- Solve the problem using an algorithm from `<algorithm>` and possibly `insert` on the container. Find the element in the string with the value you changed above and insert a new string with yet another value before it. Print your result to `stdout`.

Write your solution in `hands-on/01/main.cpp`.