

Department I - C Plus Plus

Modern and Lucid C++
for Professional Programmers

Week 3 - Iterators and Algorithms

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 02.10.2018
HS2018



Recap Week 2



- **iosfwd contains only the declarations for `std::ostream` and `std::istream`**
 - In header files (.h) this is usually sufficient when the streams are only used in function declarations
- **istream and ostream contain the implementation of the corresponding stream, operators**
 - Usually, these are required in source files (.cpp) when the streams are actually used in functions
- **iostream contains all of the above and additionally `std::cout`, `std::cin`, `std::cerr`**
 - This is only required in the source file containing the `main()` function, because only there the global standard IO variables shall be used
- **General advice: only use the minimally required header**

```
#include <iostream>
#include <string>

void askForName(std::ostream & out) {
    out << "What is your name? ";
}

std::string inputName(std::istream & in) {
    std::string name{};
    in >> name;
    return name;
}

void sayGreeting(std::ostream & out, std::string name) {
    out << "Hello " << name << ", how are you?\n";
}

int main() {
    askForName(std::cout);
    sayGreeting(std::cout, inputName(std::cin));
}
```

```
void askForName(std::ostream & out)
```

- **std::string and built-in types represent values**
 - Can be copied and passed-by-value
 - No need to allocate memory explicitly for storing the chars
- **Some objects aren't values, because they can not be copied:**
 - Streams representing the program's I/O
- **Functions taking a stream object must take it as a reference, because they provide a side-effect to the stream (i.e., output characters)**
- **Reference parameters are marked with '&' (ampersand)**
- **In Java all objects are passed as references! (not the same kind of references as in C++!)**
 - Same name, different concept

- **Statements are sequenced by ; (semicolon)**
- **Within a single expression, such as a function call, sequence of evaluation is unspecified! (except for the comma operator ,)**
 - C++17 introduced more defined sequencing relations

```
void sayGreeting(std::ostream & out,  
                std::string name1,  
                std::string name2){  
    out << "Hello " << name1 << ", do you love " << name2 << "?\n";  
}  
  
int main() {  
    askForName(std::cout);  
    sayGreeting(std::cout,  
                inputName(std::cin),  
                inputName(std::cin));  
}
```



`std::vector<T>` and its Iterators



Goals:

- You can use an `std::vector` in your code
- You know how to get and use iterators of an `std::vector`

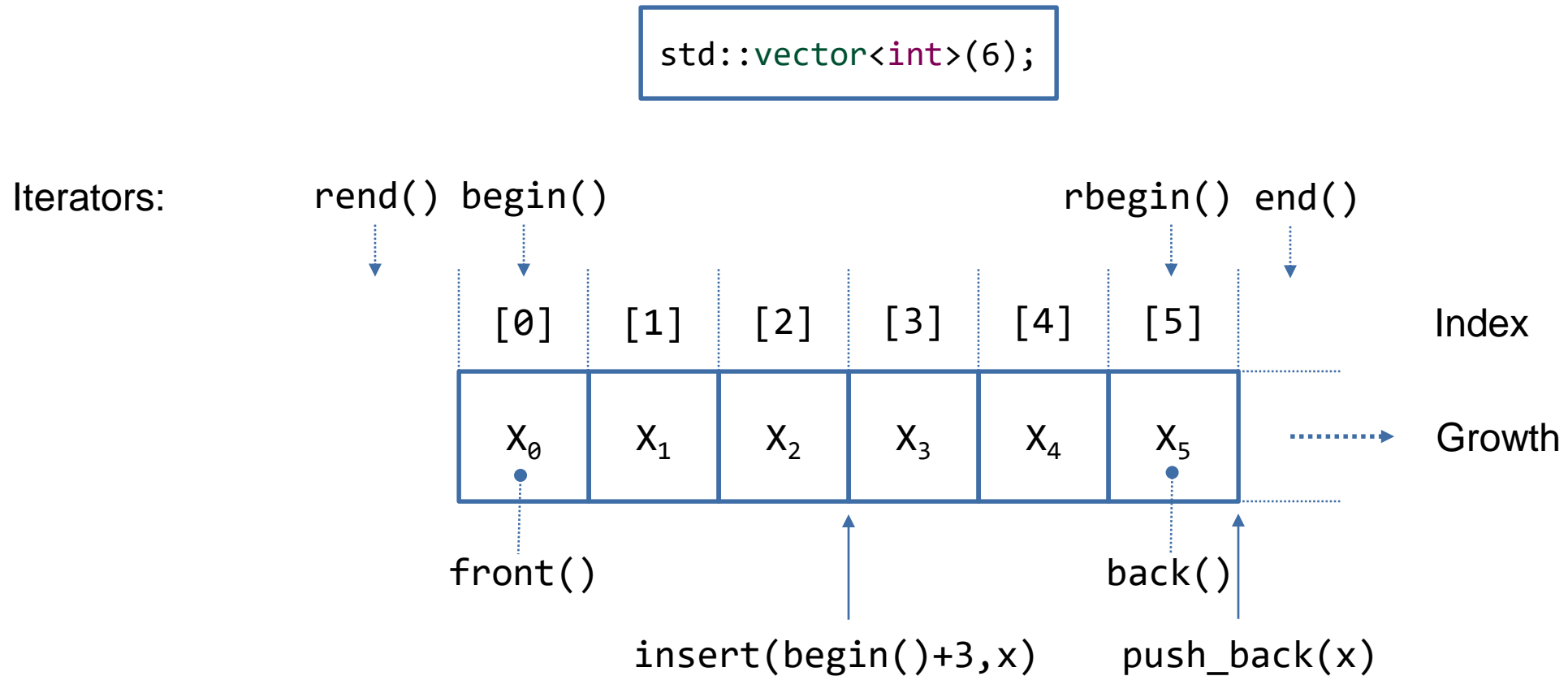
```
std::vector<int>{1, 2, 3, 4, 5};
```

- C++'s `std::vector<T>` is a **Container = contains its elements of type T (no need to allocate them)**
 - `java.util.ArrayList<T>` is a collection = keeps references to T objects (must be “new”ed)
 - T is a *template type parameter* (= placeholder for type)
- **`std::vector` can be initialized with a list of elements**
 - Otherwise it is empty: `std::vector<double> vd{};`
 - Other construction means might need parentheses (legacy)
- **When an initializer is given, the element type can be deduced!**

```
std::vector{1, 2, 3, 4, 5};
```

```
std::vector{};
```





- **Parenthesis at definition allow providing initial size, when type of elements is a number**
 - `std::vector<std::string> words{6};` works

```
for (size_t i = 0; i < v.size(); ++i) {  
    std::cout << "v[" << i << "] = " << v[i] << '\n';  
}
```

- You can index a vector like an array

- CAUTION: No bounds check!
- Accessing an element outside the valid range is Undefined Behavior



- Index variable type is "unsigned"

- size_t or std::vector<T>::size_type

- Accessing elements with at() checks bounds

- std::out_of_range exception is thrown when accessing an invalid index

```
for (size_t i = 0; i < v.size(); ++i) {  
    std::cout << v.at(i) << '\n';  
}
```

- Print all elements except the last

```
void printButLast(std::vector<char> const & values) {
    for (size_t i = 0; i < values.size() - 1; ++i) {
        std::cout << "v[" << i << "] = " << values[i] << '\n';
    }
}

int main() {
    std::vector letters{'a', 'b', 'c', 'd'};
    printButLast(letters);

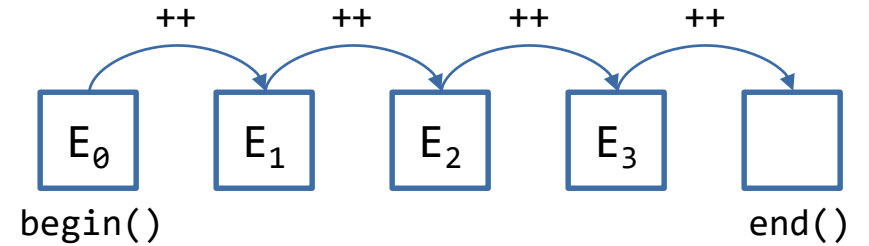
    std::vector<char> empty{};
    printButLast(empty);
}
```



- Index-based iteration is used only if the actual index value is required!

- **Advantage: No index error possible**
- **Works with all containers, even value lists {1, 2, 3}**

	const: <ul style="list-style-type: none">• element cannot be changed	non-const: <ul style="list-style-type: none">• element can be changed
reference: <ul style="list-style-type: none">• element in vector is accessed	<pre>for (auto const & cref : v) { std::cout << cref << '\n'; }</pre>	<pre>for (auto & ref : v) { ref *= 2; }</pre>
copy: <ul style="list-style-type: none">• loop has own copy of the element	<pre>for (auto const ccopy : v) { std::cout << ccopy << '\n'; }</pre>	<pre>for (auto copy : v) { copy *= 2; std::cout << copy << '\n'; }</pre>



- **Each container provides iterators**
- **There is always a pair of iterators denoting begin and end of an iteration**
 - `std::begin(v)` and `std::end(v)`
 - `v.begin()` and `v.end()`
- **C++ iterators don't know the end of an iteration (no `hasNext()` member)**
- **Operations:**
 - Comparison: You have to compare the current iterator to end
 - Accessing the current element: `*` operator
 - Step to the next element: `++` operator

```
iterator != std::end(v)
```

```
*iterator
```

```
++iterator
```

```
for (auto it = std::begin(v); it != std::end(v); ++it) {  
    std::cout << (*it)++ << ", ";  
}
```

- **Start with `std::begin(v)`**
- **Compare against `std::end(v)`**
- **Access element with `*iterator`**
 - Changing the element in a non-const container is possible in this way
- **Guarantee to just have read-only access with `std::cbegin()` and `std::cend()`**

```
for (auto it = std::cbegin(v); it != std::cend(v); ++it) {  
    std::cout << *it << ", ";  
}
```

- **This kind of iteration is only useful if the position (the iterator) is required in the loop**

Using Iterators with Algorithms



Goals:

- You know some basic algorithms of the standard library
- You can apply the algorithms to an `std::vector` with its iterators
- You should want to avoid writing hand-written loops

- **Each algorithm takes iterator arguments**
 - The range(s) of elements to apply an algorithm to is specified by iterators
- **The algorithm does what its name tells us**
- **Example: Counting values**
 - Algorithm `std::count` returns the number of occurrences of a value in range
 - Works with all ranges denoted by a pair of iterators

```
size_t count_blanks(std::string s) {
    size_t count{0};
    for (size_t i = 0; i < s.size(); ++i) {
        if (s[i] == ' ') {
            ++count;
        }
    }
    return count;
}
```

```
//The implementation is so simple it
//is not even necessary to create
//a separate function

size_t count_blanks(std::string s) {
    return std::count(s.begin(), s.end(), ' ');
}
```


- **Summing up all values in a vector (with `std::accumulate`)**

- Applies + operator to elements
- Requires the initial value

```
std::vector<int> v{5, 4, 3, 2, 1};  
std::cout << std::accumulate(std::begin(v), std::end(v), 0) << " = sum\n";
```

- **Number of elements in range (with `std::distance`)**


- Containers provide a `size()` member function
- Useful if you only have iterators

```
void printDistanceAndLength(std::string s) {  
    std::cout << "distance: " << std::distance(s.begin(), s.end()) << '\n';  
    std::cout << "in a string of length: " << s.size() << '\n';  
}
```

```
void print(int x) {
    std::cout << "print: " << x << '\n';
}
void printAll(std::vector<int> v) {
    std::for_each(std::cbegin(v), std::cend(v), print);
}
```

- Like for statement: Executes an action for each element in a range
- Last argument is a function ("first class value" in C++) that takes one parameter of the element type
- Using `std::cout` outside main is discouraged
 - What can we do if we want to print to a given `std::ostream`?

```
void print(int x, std::ostream & out) {
    out << "print: " << x << '\n';
}
void printAll(std::vector<int> v, std::ostream & out) {
    std::for_each(std::cbegin(v), std::cend(v), print(?, out));
}
```

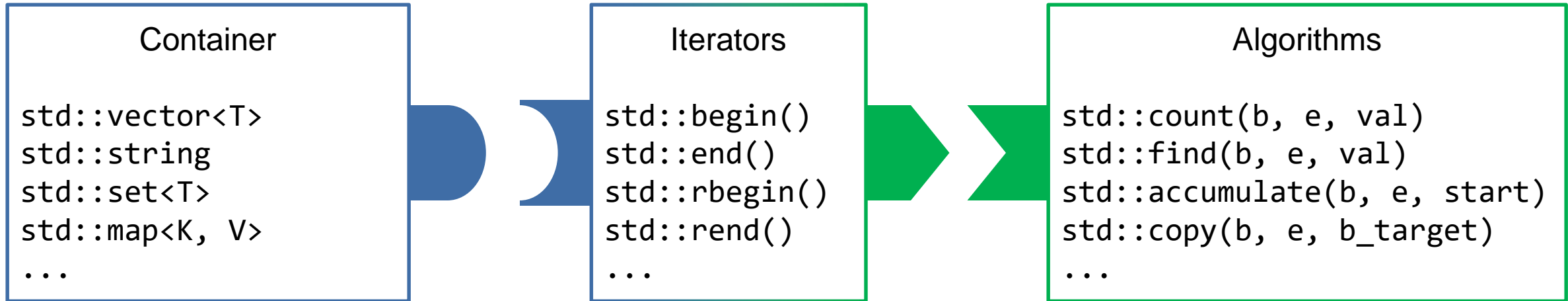


```
void printAll(std::vector<int> v, std::ostream & out) {  
    std::for_each(std::crbegin(v), std::crend(v), [&out](auto x) {  
        out << "print: " << x << '\n';  
    });  
}
```

Lambda structure:

```
[<capture>](<parameters>) -> <return-type> {  
    <statements>  
}
```

- A lambda expression creates a function object on the fly that can be passed to an algorithm
 - The created function is called from within the algorithm
 - Capture names variables taken from the surrounding scope, or define new ones (= copy, & -> reference, rename possible, type deduced)
- Parameters are like function parameters, if any, but you can use auto
- The return_type can be omitted if void or consistent return statements in the body (-> compiler knows)



- **Containers cannot be used with algorithms directly**
 - Iterators connect containers and algorithms

- Inserting elements into an `std::vector<T>`

- Append: `v.push_back(<value>);`
- Insert anywhere: `v.insert(<iterator-position>, <value>);`

- When using the `std::copy` algorithm the target has to be an iterator too

```
std::copy(<input-begin-iterator>, <input-end-iterator>, <output-begin-iterator>);
```

- Can we do the following?

```
std::vector<int> source{1, 2, 3}, target{};  
std::copy(source.begin(), source.end(), target.end());
```



Undefined
Behavior

- We need an `std::back_inserter` or an `std::inserter`

```
std::vector<int> concat(std::vector<int> first, std::vector<int> second) {  
    std::copy(second.begin(), second.end(), std::back_inserter(first));  
    return first;  
}
```

- Filling a vector with `std::fill` requires a vector with existing elements to be overwritten

```
std::vector<int> v{};
v.resize(10);
std::fill(std::begin(v), std::end(v), 2);
```

```
std::vector<int> v(10);
std::fill(std::begin(v), std::end(v), 2);
```

Caution: Requires round parentheses in case of a vector with numeric elements, otherwise it would get 1 element whose value is 10

- Or create a vector directly filled with 10 2s
 - The element type is deduced to be `int` (from 2)

```
std::vector v(10, 2);
```

- The algorithms `std::generate()` and `std::generate_n()` fill a range with computed values
 - Either use `std::back_inserter` or a non-empty container

```
std::vector<double> powerOfTwos{};
double x{1.0};
std::generate_n(std::back_inserter(powerOfTwos),
                5,
                [&x] {return x *= 2.0;})
);
```

```
std::vector<double> powerOfTwos(5);
double x{1.0};
std::generate(powerOfTwos.begin(),
              powerOfTwos.end(),
              [&x] {return x *= 2.0;})
);
```

- The `std::iota()` algorithm fills a range with subsequent values (1, 2, 3, ...) #include <numeric>

```
std::vector<int> v(100);
std::iota(std::begin(v), std::end(v), 1);
```

- **std::find() and std::find_if() return an iterator to the first element that matches the value or condition**
 - If no match exists the end of the range is returned

```
auto zero_it = std::find(std::begin(v), std::end(v), 0);  
if (zero_it == std::end(v)){  
    std::cout << "no zero found \n";  
}
```

- **Similarly std::count() and std::count_if() return the number of matching elements in a range**

```
std::cout << std::count(v.begin(), v.end(), 42) << " times 42\n";  
std::cout << std::count_if(begin(v), end(v), [](int x) {  
    return isEven(x);  
}) << " even numbers\n";
```


- **Writing readable code is about expressing intentions**

- For many intentions there is matching iterator-based algorithm in the standard library

- **It is superior to use the corresponding algorithm (function call) instead of coding your own loop**

- Correctness
- Readability
- Performance

```
bool find_with_loop(std::vector<int> const & values, int const v) {  
    auto const end = std::end(values);  
    for (auto it = std::begin(values); it != end; ++it) {  
        if (*it == v) {  
            return true;  
        }  
    }  
    return false;  
}
```

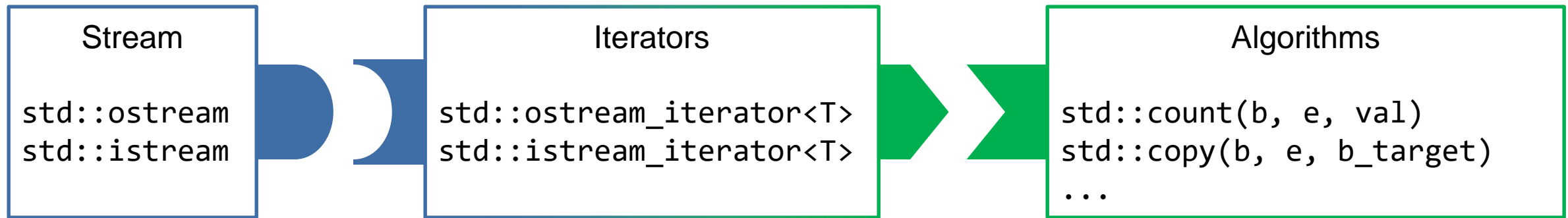
```
bool find_with_algorithm(std::vector<int> const & values, int const v) {  
    auto const pos = std::find(std::begin(values), std::end(values), v);  
    return pos != std::end(values);  
}
```

Iterators for I/O



Goals:

- You can create iterators for `std::istream` and `std::ostream`
- You can specify ranges on streams with stream iterators



- Streams (`std::istream` and `std::ostream`) cannot be used with algorithms directly

```
std::copy(std::begin(v), std::end(v), std::ostream_iterator<int>{std::cout, ", "});
```

- `std::ostream_iterator<T>` outputs values of type T to the given `std::ostream`
 - No `end()` marker needed for output, it ends when the input range ends
- `std::istream_iterator<T>` reads values of type T from the given `std::istream`
 - End iterator is the default constructed `std::istream_iterator<T>{}`
 - It ends when the stream is no longer good()

- The (stream) iterators have a very unpleasant name length, even with auto-completion
- A type alias can help to abbreviate that

```
using <alias-name> = <type>;
```

- Useful if long type names occur more than once
- Example
 - Copy strings from standard input to standard output

```
using input = std::istream_iterator<std::string>;  
input eof{};  
input in{std::cin};  
std::ostream_iterator<std::string> out{std::cout, " "};  
std::copy(in, eof, out);
```

- **`std::istream_iterator` uses operator `>>` for input**
 - Disadvantage: It skips white space
- **For an exact copy, we also need the rest**
- **`std::istreambuf_iterator<char>` uses `std::istream::get()` to get every character**
 - This only works with char-like types

```
using input = std::istreambuf_iterator<char>;
input eof{};
input in{std::cin};
std::ostream_iterator<char> out{std::cout, " "};
std::copy(in, eof, out);
```

- To fill a vector from a stream you can either use `copy` with `std::back_inserter(v)`

- It uses `v.push_back()` internally

```
using input = std::istream_iterator<int>;
input eof{};
std::vector<int> v{};
std::copy(input{std::cin}, eof, std::back_inserter(v));
```

- Or, construct the `std::vector<T>` directly from two iterators

```
using input = std::istream_iterator<int>;
input eof{};
std::vector<int> const v{input{std::cin}, eof};
```

- **Output can be done using ostream, i.e., `std::cout` and `<<`**
- **Input uses istream, i.e., `std::cin` and `>>` to an lvalue**
- **Streams have a state for eof and format errors on input**
- **Use algorithms over hand-written loops whenever possible**
- **Iterators specify ranges in C++ and connect streams/containers with algorithms**