Standard Library I

The Standard Library

Provides a collection of useful C++ classes and functions

- Is itself implemented in C++
- Part of the ISO C++ standard
  - Defines interface, semantics and contracts the implementation has to abide by (e.g. runtime complexity)
  - Implementation is *not* part of the standard
  - Multiple vendors provide their own implementations
  - Best known: `libstdc++` (used by gcc) and `libc++` (used by llvm)
- All features are declared within the `std` namespace
- Functionality is divided into sub-libraries each consisting of multiple headers
- Includes parts of the C standard library
  - For backward compatibility
  - Headers begin with "c" (e.g. `cstring`)
  - Never use them unless you absolutely know what you are doing!

The Standard Library - Feature Overview (1)

Most important library features:

- Utilities
    - Memory management (`new`, `delete`, `unique_ptr`, `shared_ptr`)
    - Error handling (exceptions, `assert()`)
    - Time (clocks, durations, timestamps, ...)
    - Optionals, Variants, Tuples, ...
- Strings
    - String class
    - String views
    - C-style string handling
- Containers: `array`, `vector`, lists, maps, sets
- Algorithms: (stable) sort, search, max, min, ...
- Iterators
- Numerics
    - Common mathematatic functions (`sqrt`, `pow`, `mod`, `log`, ...)
    - Complex numbers
    - Random number generation

The Standard Library - Feature Overview (2)

- I/O
  - Input-/Output streams
  - File streams
  - String streams
- Threads
  - Thread class
  - (shared) mutexes
  - futures
- And much more
  - Localization
  - Regex
  - Atomics
  - Filesystem support
  - ...

`std::string`

`std::string` is a class encapsulating character sequences

- Manages its own memory (so no need for `new`/`malloc`/`unique_ptr`)
- Has a wide array of member functions, making string manipulation easier
- Knows its own length: No need to worry about null termination!
- Contents are guaranteed to be stored in memory contiguously
- Can be used like a C-style `char` pointer
- Access to the underlying C-style `char` pointer via `c_str()`

`std::string` is defined in the `<string>` library header

- It is a typedef to `std::basic_string<char>`
- `std::basic_string` also has specializations for 16- and 32-bit character strings
- Specialization of `std::basic_string` with custom character types possible

`std::string` should *always* be preferred over `char` pointers!

Creating a std::string

The default constructor creates an empty string of length 0

```
std::string s;
s.size(); // == 0
```

Creation from a string literal via constructor argument or assignment

```
std::string s_constructed("my literal");
std::string s_assigned = "my literal";
```

Take care for strings that contain null-bytes:

```
std::string s = "null\0byte!";
std::cout << s << std::endl; // prints "null"

std::string s_with_size("null\0byte!", 10);
std::cout << s_with_size << std::endl; // prints "nullbyte!"
```

Accessing contents of std::string (1)

Single characters can be accessed with at() or array notation

```
std::string s = "Hello World!";
std::cout << s.at(4) << s[6] << std::endl; // prints "oW"
```

Since both functions return a reference, this can be used to modify the string

```
std::string s = "Hello World!";
s.at(4) = 'x';
s[6] = 'Y';
s[10] = s.at(9);
std::cout << s << std::endl; // prints "Hellx Yorll!"
```

Out of bounds access with array notation results in undefined behaviour, at() throws an exception

Accessing contents of std::string (2)

Iterators allow iteration over contents

```
std::string s = "Hello World!";
for (auto iter = s.begin(); iter != s.end(); ++iter) {
    ++(*iter);
}
std::cout << s << std::endl; // prints "Ifmmp!Xpsme"
```

For backwards compatibility: `c_str()` returns null-terminated char pointer

```
int i_only_know_c(const char* str) {
    int len = 0;
    while (str) { str++; len++; }
    return len;
}

std::string i_am_modern_cpp = "Hello World!";
int len = i_only_know_c(i_am_modern_cpp.c_str()); // 12
```

Comparing `std::string`

The `std::string` class provides a `compare()` function, comparing two strings (or substrings) lexicographically

```
std::string s1 = "Hello World!";
std::string s2 = "Goodbye World!";

std::cout << s1.compare(s2); // 1, G before H
//For substrings:
std::cout << s1.compare(6, 5, s2, 8, 5); //0, both are "World"
```

If three-way- or substring comparison is not needed, the standard relational operators <, ==, <=, ... can be used instead

```
std::string u0510 = "breezy badger";
std::string u1804 = "bionic beaver";
std::string u1904 = "disco dingo";

assert(u1904 > u1804); //okay, d after b
assert(u1804 > u0510); //fails, bi before br. Why, Ubuntu?!
```

`std::string` Operations

The standard library provides features a modern string library is expected to have, such as:

- `size()` or `length()`: The number of characters in the string
- `empty()`: Returns true if the string has no characters
- `append()` and `+=`: Appends another string or character. No need for manual memory allocations!
- `+` concatenates two strings
- `find()`: Returns the offset of thie first occurence of the substring, or the constant `std::string::npos` if not found
- `substr()`: Returns a new `std::string` that is a substring at the given offset and length. Be careful! Most of the times, you probably want a string view instead of a substring!

`std::string_view` (1)

Copying strings and creating substrings is expensive

- Whenever a substring is created, data is essentially duplicated
- Huge overhead when handling large amounts of data (e.g. parsing large JSON files)

`std::string_view` help avoid expensive copying

- Read-only views on already existing strings
- Internally: Just a pointer and a length
- Creation, substring and copying in constant time (vs. linear for strings)

`std::string_view` is defined in the `<string_view>` library header

- Creation: `std::string` (and optionally size) as constructor argument, from a char pointer with a length, or from a string literal
- Also has all (read-only) member functions of `std::string`
- Substring creates another string view in O(1)

Use `std::string_view` over `std::string` whenever possible!

std::string_view (2)

Example

```
std::string s = "garbage garbage garbage interesting garbage";

std::string sub = s.substr(24,11); // With string: O(n)

// With string view:
std::string_view s_view = s; // O(1)
std::string_view sub_view = s_view.substr(24,11); // O(1)

// Or in place:
s_view.remove_prefix(24); // O(1)
s_view.remove_suffix(s_view.size() - 11); // O(1)

// Also useful for function calls:
bool is_eq_naive(std::string a, std::string b) {return a == b; }
bool is_eq_views(std::string_view a, std::string_view b) {
     return a == b; }

is_eq_naive("abc", "def"); // 2 allocations at runtime
is_eq_with_views("abc", "def"); // no allocation at runtime
```

String Literals

There are also special literals to construct `std::string_view` and `std::string` objects that deal with null bytes correctly.
To use them, you have to use
`using namespace std::literals::string_view_literals` or
`using namespace std::literals::string_literals`.

```
using namespace std::literals::string_view_literals;
using namespace std::literals::string_literals;

auto s1 = "string_view\0with\0nulls"sv; // s1 is a string_view
auto s2 = "string\0with\0nulls"s; // s2 is a string

std::cout << s1; // prints "string_viewwithnulls"
std::cout << s2; // prints "stringwithnulls"
```

`std::optional`

`std::optional` is a class encapsulating a value that might or might not exist

- Defined in the header `<optional>`
- Some functions might fail or return without a valid result (e.g. looking up a non-existing file)
- It's unfavourable to encode such failures with a value of the function domain (e.g. an empty string when file could not be read)
- `std::optional` helps to express such results:
  - At any point in time, an optional either has a value, or it doesn't
  - If the computation succeeded, it returns an optional containing a value
  - If it failed, it returns an optional without a value
- The template parameter `T` denotes, of which type the optional may contain a value (e.g. `optional<int>` might contain an `int`)
- Guarantees to not dynamically allocate any memory when being assigned a value
- Is an object, despite supporting the dereference operators `*` and `->`
- Internally implemented as an object with a member of type `T` and a boolean

std::optional: Creation

Optionals are created through its constructor or with `std::make_optional`:

```
std::optional<std::string> might_fail(int arg) {
    if (arg == 0) {
        return std::optional<std::string>("zero");
    } else if (arg == 1) {
        return "one"; // equivalent to the case above
    } else if (arg < 7) {
        //std::make_optional takes constructor arguments of type T
        return std::make_optional<std::string>("less than 7");
    } else {
        return std::nullopt; // alternatively: return {}
    }
}
```

The value of an optional can be read whith `value()` (throws exception when empty) or dereferenced with `*` or `->` (undefined behavior when empty)

```
might_fail(3).value(); // "less than 7"
might_fail(8).value(); // throws std::bad_optional_access

*might_fail(3); // "less than 7"
might_fail(6)->size(); // 11
might_fail(7)->empty(); // undefined behavior
```

std::optional: Checking and Accessing

There are multiple ways to check whether an optional has a value:

```
might_fail(3).has_value(); // true
might_fail(8).has_value(); // false

// Or even simpler:
std::optional<std::string> opt5 = might_fail(5)
if (opt5) { //contextual conversion to bool
    opt5->size(); // 11
}
```

Providing a default value without boilerplate:

```
might_fail(42).value_or("default"); // "default"
```

Clearing an optional:

```
std::optional<std::string> opt5 = might_fail(5)
opt5.has_value(); // true
opt5.reset(); // Clears the value
opt5.has_value(); // false
```

std::pair

`std::pair<T, U>` is a template class that stores exactly one object of type `T` and one of type `U`.

- Defined in the header `<utility>`
- Constructor takes object of `T` and `U`
- Pairs can also be constructed with `std::make_pair()`
- Objects can be accessed with `first` and `second`
- Can be compared for equality and inequality
- Can be compared lexicographically with <, <=, >, and >=

```
std::pair<int, double> p1(123, 4.56);
p1.first; // == 123
p1.second; // == 4.56
auto p2 = std::make_pair(456, 1.23);
// p2 has type std::pair<double, int>
p1 < p2; // true
```

std::tuple

`std::tuple` is a template class with *n* type template parameters that stores exactly one object of each of the *n* types.

- Defined in the header `<tuple>`
- Constructor takes all objects
- Tuples can also be constructed with `std::make_tuple()`
- The ith object can be accessed with `std::get<i>()`
- Just like pairs, tuples define all relational comparison operators

```
std::pair<int, double, char> t1(123, 4.56, 'x');
std::get<1>(t1); // == 4.56
auto p2 = std::make_tuple(456, 1.23, 'y');
// p2 has type std::tuple<int, double, char>
p1 < p2; // true
```

std::tie()

Tuples can also contain values of reference type. They can be constructed with `std::tie()`.

- Can be used to easily “decompose” a tuple into existing variables
- Can also be used to quickly do lexicographic comparison on different objects

```
auto t = std::make_tuple(123, 4.56);
int a; double b;
std::tie(a, b) = t; // "decompose" t into a and b
// a is now 123, b is 4.56
int x = 456; double y = 1.23;
// Lexicographic comparison on (a, b) and (x, y):
std::tie(a, b) < std::tie(x, y); // true
```

Structured Bindings and Tuples

- Often, using structured bindings is easier than using `std::tie()`
- For tuples, `auto [a, b, c] = t;` initializes a, b, and c with `std::get<0>(t)`, `std::get<1>(t)`, and `std::get<2>(t)`, respectively
- Also works with `auto&` and `const auto&` in which case a, b, and c become references
- Also works with `std::pair`

```
auto t = std::make_tuple(1, 2, 3);
auto [a, b, c] = t; // a, b, c have type int
auto p = std::make_pair(4, 5);
auto& [x, y] = p; // x, y have type int&
x = 123; // p.first is now 123
```

Containers - A Short Overview

A container is an object that stores a collection of other objects

- Manage the storage space for their elements
- Generic: The type(s) of elements stored are template parameter(s)
- Provide member functions for accessing elements directly, or through iterators
- (Most) member functions shared between containers
- Make guarantees about the complexity of their operations:
    - Sequence containers (e.g. `std::array`, `std::vector`, `std::list`): Optimized for sequential access
    - Associative containers (e.g. `std::set`, `std::map`): Sorted, optimized for search ($O(\log n)$)
    - Unordered asscoiative containers (e.g. `std::unordered_set`, `std::unordered_map`): Hashed, optimized for search (amortized: $O(1)$, worst case: $O(n)$)

Use containers whenever possible! When in doubt, use `std::vector`!

`std::vector`

Vectors are arrays that can dynamically grow in size

- Defined in the header `<vector>`
- Elements are still stored contiguously
- Elements can be inserted and removed at any position
- Preallocates memory for a certain amount of elements
- Allocates new, larger chunk of memory and moves elements when memory is exhausted
- Memory for a given amount of elements can be reserved with `reserve()`
- Time complexity:
  - Random access: $O(1)$
  - Insertion and removal at the end: Typically $O(1)$, worst case: $O(n)$ due to possible reallocation
  - Insertion and removal at any other position: $O(n)$
- Access to the underlying C-style data array with `data()` member function

std::vector: Accessing Elements

Vectors are constructed just like arrays:

```
std::vector<int> fib = {1,1,2,3};
```

Access elements via C-style array notation, via `at()`, or through a raw pointer:

```
fib.at(0) // == 1;
fib[1] // == 1;

int* fib_ptr = fib.data();
fib_ptr[2] // == 3;
```

Update elements via C-style array notation, via `at()`, or through a raw pointer:

```
fib[3] = 43;
fib.at(2) = 42;
fib.data()[1] = 41; // fib is now 1, 41, 42, 43
```

Note: It is not possible to insert new elements this way! You can only update existing ones.

std::vector: Inserting and Removing Elements

Insert or remove elements at the end in constant time:

```
fib.push_back(5); // fib is now 1, 1, 2, 3, 5
int my_fib = fib.back(); // my_fib is 5
fib.pop_back(); // fib is 1, 1, 2, 3
```

Inserted or remove elements anywhere with an iterator pointing at the element after insertion, or the element to be erased respectively:

```
auto it = fib.begin(); it += 2;
fib.insert(it, 42); // fib is now 1, 1, 42, 2, 3

// insertion invalidated the iterator, get a new one
it = fib.begin(); it +=2;
fib.erase(it); // fib is now again 1, 1, 2, 3
```

Empty the whole vector with `clear`:

```
fib.clear();
fib.empty(); // true
fib.size(); // == 0
```

std::vector: Emplacing elements

Construct elements in place to avoid expensive moving around of data:

```
struct ExpensiveToCopy {
    ExpensiveToCopy(int id, std::string comment) :
        id(id), comment(std::move(comment)) {}
    int id;
    std::string comment;
};

std::vector<ExpensiveToCopy> vec;

// The expensive way:
ExpensiveToCopy e1(1,"my comment 1");
vec.push_back(e1); // need to copy e1!
// Better way, use std::move:
vec.push_back(std::move(e1));

// The best way:
vec.emplace_back(2,"my comment 2");

// Also works at any other position:
auto it = vec.begin(); it++;
vec.emplace(it, 3, "my comment 3");
```

std::vector: Reserving memory

If the final size of a vector is already known, give the vector a hint to avoid unnecessary reallocations:

```
std::vector<int> vec;
vec.reserve(1000000); //enough space for 1000000 elements is allocated
vec.capacity(); // == 1000000
vec.size(); // == 0, do not mix this up with capacity!

for (int i = 0; i < 1000000; i++) {
    vec.push_back(i); // no reallocations in this loop!
}
```

If the vector won't grow in the future, reduce its capacity to save unused space:

```
std::vector<int> vec;
vec.reserve(100); // Reserve some space to be sure

// Turns out, we only needed a capacity of 10
for (int i = 0; i < 10; i++) {
    vec.push_back(i);
}

// Free the space for the other 90 elements we reserved but didn't need
vec.shrink_to_fit();
```

std::unordered_map

Maps are associative containers consisting of key-value pairs

- Defined in the header `<unordered_map>`
- Keys are required to be unique
- At least two template parameters: `Key` and `T` (type of the values)
- Is internally a hash table
- Amortized $O(1)$ complexity for random access, search, insertion, and removal
- No way to access keys or values in order (use `std::map` for that!)
- Accepts custom hash- and comparison functions through third and fourth template parameter

Use `std::unordered_map` if you need a hash table, but don't need ordering

std::unordered_map: Accessing Elements

Maps can be constructed pairwise:

```
std::unordered_map<std::string,double>
    name_to_grade {{"maier", 1.3}, {"huber", 2.7}, {"schmidt", 5.0}};
```

Lookup the value to a key with C-style array notation, or with `at()`:

```
name_to_grade["huber"]; // == 2.7
name_to_grade.at("schmidt"); // == 5.0
```

A pair can also be searched for with `find`:

```
auto search = name_to_grade.find("schmidt");

if (search != name_to_grade.end()) {
    // Returns an iterator pointing to a pair!
    search->first; // == "schmidt"
    search->second; // == 5.0
}
```

To check if a key exists, use `count`:

```
name_to_grade.count("schmidt"); // == 1
name_to_grade.count("blafasel"); // == 0
```

`count()` either returns 0 or 1.

std::unordered_map: Insertion and Removal

Update or insert elements like this:

```
name_to_grade["musterfrau"] = 3.0;
```

In contrast to vectors, the array-notation also allows the insertion of new KV-pairs!

Maps also allow the direct insertion of pairs:

```
std::pair<std::string, double> pair("mueller", 1.0);
name_to_grade.insert(pair);

// Or simpler:
name_to_grade.insert({"mustermann", 3.7});

// Emplace also works:
name_to_grade.emplace("gruber", 1.7);
```

Erase elements with `erase()` or empty the container with `clear()`:

```
auto search = name_to_grade.find("schmidt");
name_to_grade.erase(search); // removes the pair with "schmidt" as key

name_to_grade.clear(); // removes all elements of name_to_grade
```

std::map

In contrast to unordered maps, the keys of `std::map` are sorted

- Defined in the header `<map>`
- Interface largely the same to `std::unordered_map`
- Optionally accepts a custom comparison function as template parameter
- Is internally a tree (usually AVL- or R/B-Tree)
- $O(log\ n)$ complexity for random access, search, insertion, and removal

`std:map` also allows to search for ranges:

`upper_bound()` returns an iterator pointing to the first *greater* element:

```
std::map<int, int> x_to_y = {{1, 1}, {3, 9}, {7, 49}};
auto gt3 = x_to_y.upper_bound(3);

while (gt3 != x_to_y.end()) {
    std::cout << gt3->first << "->" << gt3->second << ","; // 7->49,
}
```

`lower_bound()` returns an iterator pointing to the first element *not lower*:

```
auto geq3 = x_to_y.lower_bound(3);

while (geq3 != x_to_y.end()) {
    std::cout << geq3->first << "->" << geq3->second << ","; // 3->9, 7->49,
}
```

`std::unordered_set`

Sets are associative containers consisting of keys

- Defined in the header `<unordered_set>`
- Keys are required to be unique (as is expected of a set)
- Template parameter `Key` for the type of the elements
- Is internally a hash table
- Amortized $O(1)$ complexity for random access, search, insertion, and removal
- No way to access keys in order (use `std::set` for that!)
- Elements must not be modified! If an element's hash changes, the container might get corrupted
- Accepts custom hash- and comparison functions through second and third template parameter

std::unordered_set: Checking for Elements

Sets can be constructed just like arrays:

```
std::unordered_set<std::string>
    shopping_list {"milk", "bread", "butter"};
```

Look for an element with `find()`:

```
auto search = shopping_list.find("milk");

if (search != shopping_list.end()) {
    // Returns an iterator pointing to the element!
    *search; // == "milk"
}
```

Or with `count()` (returning either 0 or 1):

```
shopping_list.count("bread"); // == 1
shopping_list.count("blafasel"); // == 0
```

Check the number of the elements with `size()`:

```
shopping_list.size(); // == 3
shopping_list.empty(); // false
```

std::unordered_set: Insertion and Removal

Update or insert elements like this:

```
shopping_list.insert("lettuce");

//Emplace also works:
shopping_list.emplace("milk");
```

insert returns a std::pair<iterator,bool> indicating if insertion succeeded:

```
auto result = shopping_list.insert("milk");

result.second; // false, as "milk" is already an element of shopping_list
*result.first; // "milk", iterator points to element preventing insertion

result = shopping_list.insert("broccoli");
result.second; // true, "broccoli" was added
*result.first; // "broccoli", iterator points to newly inserted element
```

Erase elements with erase() or empty it with clear:

```
auto search = shopping_list.find("milk");
shopping_list.erase(search); // "milk" is no longer an element of shopping_list

shopping_list.clear(); // removes all elements of shopping_list
```

std::set

In contrast to unordered sets, the elements of `std::set` are sorted

- Defined in the header `<set>`
- Interface largely the same to `std::unordered_set`
- Optionally accepts a custom comparison function as template parameter
- Is internally a tree (usually AVL- or R/B-Tree)
- $O(log\ n)$ complexity for random access, search, insertion, and removal

`std:set` also allows to search for ranges:

`upper_bound()` returns an iterator pointing to the first *greater* element:

```
std::set<int> x = {1, 3, 7};
auto gt3 = x.upper_bound(3);

while (gt3 != x.end()) {
    std::cout << x << ","; // 7,
}
```

`lower_bound()` returns an iterator pointing to the first element *not lower*:

```
std::set<int> x = {1, 3, 7};
auto geq = x.lower_bound(3);

while (geq != x.end()) {
    std::cout << x << ","; // 3, 7,
}
```

Containers: Thread Safety

Containers give some thread safety guarantees:

- Two different containers: All member functions can be called concurrently by different threads (i.e. different containers don't share state)
- The same container: All *const* member functions can be called concurrently. `at()`, `[]` (expect in associative containers), `data()`, `front()`/`back()`, `begin()`/`end()`, `find()` also count as *const*
- Iterator operations that only read (e.g. incrementing or dereferencing an iterator) can be run concurrently with reads of other iterators and `const` member functions
- Different elements of the same container can be modified concurrently
- Be careful: As long as the standard does not explicitly require a member function to be sequential, the standard library implementation is allowed to parallelize it interally (see e.g. `std::transform` vs. `std::for_each`)

Rule of thumb: Simultaneous reads on the same container are always okay, simultaneous read/writes on *different* containers are also okay. Everything else requires synchronization.

Iterators: A Short Overview

Iterators are objects that can be thought of as pointer abstractions

- Problem: Different element access methods for each container
- Therefore: Container types not easily exchangable in code
- Solution: Iterators abstract over element access and provide pointer-like interface
- Allow for easy exchange of underlying container type
- The standard library defines multiple iterator types as containers have varying capabilities (random access, traversable in both directions, ...)

Be careful: When writing to a container, all existing iterators are invalidated and can no longer be used (some exceptions apply)!

Iterators: An Example (1)

All containers have a begin and an end iterator:

```
std::vector<std::string> vec = {"one", "two", "three", "four"};
auto it = vec.begin();
auto end = vec.end();
```

The begin iterator points to the first element of the container:

```
std::cout << *it; // prints "one"
std::cout << it->size(); // prints 3
```

The end iterator points to the first element *after* the container. Dereferencing it results in undefined behavior:

```
*end; // undefined behavior
```

An iterator can be incremented (just like a pointer) to point at the next element:

```
++it; // Prefer to use pre-increment
std::cout << *it; // prints "two"
```

Iterators: An Example (2)

Iterators can be checked for equality. Comparing to the end iterator is used to check whether iteration is done:

```
 // prints "three,four,"
while (it != end) {
 std::cout << *it << ",";
 it++;
}
```

This can be streamlined with a range-based for loop:

```
for (auto elem : vec) {
    std::cout << elem << ","; // prints "one,two,three,four,"
}
```

Such a loop requires the *range expression* (here: `vec`) to have a `begin()` and `end()` member.
`vec.begin()` is assigned to an internal iterator which is dereferenced, assigned to the *range declaration* (here: `auto elem`), and then incremented until it equals `vec.end()`.

Iterators: An Example (3)

Iterators can also simplify dynamic insertion and deletion:

```
for (it = vec.begin(); it != vec.end(); ++it) {
    if (it->size == 3) {
        it = vec.insert(it,"foo");
        // it now points to the newly inserted element
        ++it;
    }
}
//vec == {"foo", "one", "foo", "two", "three", "four"}

for (it = vec.begin(); it != vec.end(); ++it) {
    if (it->size == 3) {
        it = vec.erase(it);
        // erase returns a new, valid iterator
        // pointing at the next element
    }
}
//vec == {"three", "four"}
```

InputIterator and OutputIterator

Input- and OutputIterator are the most basic iterators. They have the following features:

- Equality comparison: Checks if two iterators point to the same position
- Dereferencable with the `*` and `->` operators
- Incrementable, to point at the next element in sequence
- A dereferenced `InputIterator` can *only* by read
- A dereferenced `OutputIterator` can *only* be written to

As the most restrictive iterators, they have a few limitations:

- Single-pass only: They cannot be decremented
- Only allow equality comparison, <, >=, etc. not supported
- Can only be incremented by one (i.e. `it + 2` does *not* work)

Used in single-pass algorithms such as `find()` (`InputIterator`) or `copy()` (Copying from an `InputIterator` to an `OutputIterator`)

`ForwardIterator` and `BidirectionalIterator`

`ForwardIterator` combines `InputIterator` and `OutputIterator`

- All the features and restrictions shared between input- and output iterator apply
- Dereferenced iterator can be read and written to

`BidirectionalIterator` generalizes `ForwardIterator`

- Additionally allows decrementing (walking backwards)
- Therefore supports multi-pass algorithms traversing the container multiple times
- All other restrictions of `ForwardIterator` still apply

RandomAccessIterator and ContiguousIterator

`RandomAccessIterator` generalizes `BidirectionalIterator`

- Additionally allows random access with `operator[]`
- Supports relational operators, such as < or >=
- Can be incremented or decremented by any amount (i.e. `it + 2` *does* work)

`ContiguousIterator`

- Introduced with C++17
- Guarantees that elements are stored in memory contiguously
- Formally: For every iterator `it` and integral value n: if `it` + n is a valid iterator, then $*(\texttt{it} + n) \Leftrightarrow *(\texttt{std::addressof}(*\texttt{it}) + n)$
- Orthogonal to all other iterators (i.e. a `ContiguousIterator` is not necessarily a `RandomAccessIterator`)
- Code predating C++17 often treats `RandomAccessIterators` of `std::string`, `std::vector`, and `std::array` as if they were `ContiguousIterators`

Streams and I/O

The standard library has an entire library for I/O operations. The main concept of the I/O library is a *stream*.

- Streams are organized in a class hierarchy
- `std::istream` is the base class for input operations (e.g. `operator>>`)
- `std::ostream` is the base class for output operations (e.g. `operator<<`)
- `std::iostream` is a subclass of `std::istream` and `std::ostream`
- `std::cin` is an instance of `std::istream` that represents stdin
- `std::cout` is an instance of `std::ostream` that represent stdout

As for strings, streams are actually templates parametrized with a character type.

- `std::istream` is an alias for `std::basic_istream<char>`
- `std::ostream` is an alias for `std::basic_ostream<char>`
- `std::wistream` also exists and is an alias for `std::basic_istream<wchar_t>`
- `std::wcin` is an instance of `std::wistream` that also represent stdin
- Same for `std::wostream` and `std::wcout`

Common Operations on Streams

All streams are subclasses of `std::basic_ios` and have the following member functions:

- `good()`, `fail()`, `bad()`: Checks if the stream is in a specific error state
- `eof()`: Checks if the stream has reached end-of-file
- `operator bool()`: Returns true if stream has no errors

```
int value;
if (std::cin >> value) {
    std::cout << "value = " << value << std::endl;
} else {
    std::cout << "error" << std::endl;
}
```

Input Streams

Input streams (`std::istream`) support several input functions:

- `operator>>()`: Reads a value of a given type from the stream, skips leading whitespace
- `operator>>()` can be overloaded for own types as second argument to support being read from a stream
- `get()`: Reads single or multiple characters until a delimiter is found
- `read()`: Reads given number of characters

```
// Defined by the standard library:
std::istream& operator>>(std::istream&, int&);
int value;
std::cin >> value;

// Read (up to) 1024 chars from cin:
std::vector<char> buffer(1024);
std::cin.read(buffer.data(), 1024);
```

Output Streams

Output streams (`std::ostream`) support several output functions:

- `operator<<()`: Writes a value to the stream
- `operator<<()` can be overloaded for own types as second argument to support being written to a stream
- `put()`: Writes a single character
- `write()`: Writes multiple characters

```
// Defined by the standard library:
std::ostream& operator<<(std::ostream&, int);
std::cout << 123;

// Write 1024 chars to cout:
std::vector<char> buffer(1024);
std::cout.write(buffer.data(), 1024);
```

String Streams

std::stringstream can be used when input and output should be written and read from a std::string.

- Defined in the header <sstream>
- Is a subclass of std::istream and std::ostream
- Initial contents can be given in the constructor
- Contents can be extracted and set with str()

```
std::stringstream stream("1 2 3");
int value;
stream >> value; // value == 1
stream.str("4"); // Set stream contents
stream >> value; // value == 4
stream << "foo";
stream << 123;
stream.str(); // == "foo123"
```

File Streams

The standard library defines several streams for file I/O in the `<fstream>` header:

- `std::ifstream`: Input file stream to read to a file
- `std::ofstream`: Output file stream to write to a file
- `std::fstream`: File stream to read and write to a file

```
std::ifstream input("input_file");
if (!input) { std::cout << "couldn't open input_file\n"; }
std::ofstream output("output_file");
if (!output) { std::cout << "couldn't open output_file\n"; }
// Read an int from input_file and write it to output_file
int value = -1;
if (!(input >> value)) {
    std::cout << "couldn't read from file\n";
}
if (!(output << value)) {
    std::cout << "couldn't write to file\n";
}
```

Disadvantage of Streams

Even though streams are nice to use, they should be avoided in many cases:

- Streams make have use of virtual functions and virtual inheritance which by itself can sometimes be a significant performance overhead
- Streams respect the system's locale settings (e.g. whether to use a period or a comma for floating point numbers) which also makes them slow
- Especially parsing of integers is very inefficient

General rule: When input is typed in by a user, using streams is fine. When input is read from files or generated automatically, better use OS-specific functions.