Concepts of C++ Programming

Lecture 8: Containers and Iterators

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

std::optional[105]

- std::optional<T> (<optional>): value that might not exist
- Can be empty (no value) or non-empty (holding a value)
- Implicit conversion to bool, access contained value with * or ->

```
std::optional<std::string> mightFail(unsigned arg) {
  if (arg < 7) {
    return "lt␣7"; // equiv to: std::optional<std::string>("lt 7")
  } else {
    return std::nullopt; // alternatively: return {};
  }
}
void foo(unsigned n) {
  if (auto optStr = mightFail(n))
    std::println("{}", optStr->size()); // prints: 4
}
```

[105] https://en.cppreference.com/w/cpp/utility/optional

Optional Reference

Quiz: What is the most efficient way to return an optional reference?

A. `std::optional<Foo&>`
B. `std::optional<Foo*>`
C. `std::optional<std::reference_wrapper<Foo>>`
D. `Foo*`

std::pair[106]

- `std::pair<T, U>` (`<utility>`): pair of two values
- Members can be accessed with `first` and `second`
- Constructible with constructor or `std::make_pair`

```
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<int, double>
p1 < p2; // true
```

[106] https://en.cppreference.com/w/cpp/utility/pair

std::tuple[107]

- `std::tuple<...>` (`<utility>`): tuple of n values
- Members can be accessed with `std::get<i>()`
- Constructible with constructor or `std::make_tuple`

```
std::tuple<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
```

[107] https://en.cppreference.com/w/cpp/utility/tuple

Structured Bindings[108]

- `auto [a, b] = t;` initialized with `std::get<0>(t)` and `std::get<1>(t)`
- Also with `auto&` and `const auto&` for references to elements

```
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
```

[108] https://en.cppreference.com/w/cpp/language/structured_binding

Using Pair/Tuple

- Pair/tuple convey no information about semantics
- User-defined types often preferable, esp. in public interfaces

⇒ Use `std::pair`/`std::tuple` sparingly

```
struct Rational {
  long numerator;
  long denominator;
};
std::pair<long, long> canonicalize(long, long); // BAD
Rational canonicalize(const Rational&); // BETTER
```

std::variant[109]

- Type which holds exactly one of the alternative types
- Type-safe, alternative share same underlying storage $\rightsquigarrow$ smaller size
- Accessible with `std::get`, `std::holds_alternative`

```cpp
std::variant<int, double> v; // holds either an int or a double

v = 42; // now holds an int
assert(std::holds_alternative<int>(v));
assert(std::get<int>(v) == 42);

v = 1.0; // now holds a double
// get_if returns pointer to active value, or nullptr
assert(*std::get_if<double>(&v) == 1.0);
assert(std::get_if<int>(&v) == nullptr);
```

[109] https://en.cppreference.com/w/cpp/utility/variant

Iterators[110]

- Standard library provides various containers, code might define custom ones

- Problem: different containers can have different access methods

$\rightsquigarrow$ containers not easily exchangable

- Solution: abstract over element access with iterators
  - Same pointer-like interface for all containers

$\Rightarrow$ Allows for easy exchange of container type
  - Very useful in templates specialized on containers
- Containers define:
  - `begin()` – iterator pointing to first element
  - `end()` – iterator pointing to the first element *after* the container

[110] https://en.cppreference.com/w/cpp/iterator

Iterators: Usage Example

```cpp
#include <array>
#include <print>
int main() {
  std::array<int, 2> arr{1, 2};
  auto it = arr.begin();
  assert(*it == 1);
  ++it; // prefer pre-increment
  assert(*it == 2);
  ++it;
  assert(it == arr.end()); // end iterator not dereferencable (UB)

  for (auto it = arr.begin(); end = arr.end(); it != end; ++it)
    std::println("{}", *it);
}
```

Range-Based for Loop[111]

- for-range loop is syntactic sugar for:
  - Calling `begin()` and `end()` of the range
  - Looping until the iterator equals the end iterator
  - Defining variables inside the loop body from the iterator

```
#include <array>
#include <print>
int main() {
  std::array<int, 2> arr{1, 2};
  for (int& x : arr)
    x += 5;
  // ... is identical to:
  for (auto it = arr.begin(); end = arr.end(); it != end; ++it) {
    int& x = *it;
    x += 5;
  }
}
```

[111] https://en.cppreference.com/w/cpp/language/range-for

Input/Output Iterator

- Concepts: `std::input_iterator`/`std::output_iterator`
- Required features:
  - `it1 == it2` – whether iterators point to the same position
  - `*it`, `it->` – dereferencing
  - `++it`, `it++` – incrementing
  - Input iterator: dereferenced iterator can only be read
  - Output iterator: dereferenced iterator can only be written to
- Single-pass only: not decrementable, two iterators might yield different values

Forward/Bidirectional Iterator

- Concepts: `std::forward_iterator`/`std::bidirectional_iterator`
- Forward iterator – required features:
  - All features shared by input/output iterator
  - Multi-pass guarantee: `it1 == it2` implies `++it1 == ++it2`
- Bidirectional iterator – forward iterator with:
  - `--it`, `it--` – decrementing (walking backwards)

Random Access/Contiguous Iterator

- Concepts: `std::random_access_iterator`/`std::contiguous_iterator`

- Random access iterators – bidirectional iterator with:
  - `it[]` – random access
  - Relational operators, e.g. `it1 < it2`
  - Incrementable/decrementable by any amount, e.g. `it + 2`, `it -= 5`

- Contiguous iterator – random access iterator with:
  - Elements are stored contiguously in memory
  - `&*(it + n)` equivalent to `(&*it) + n`

Implementing Iterators for a Linked List

(see script)

Insertion and Removal

- Containers generally use iterators for removing elements
  - Already have some handle to the element $\rightsquigarrow$ use it
  - Especially important for data structures with non-$\mathcal{O}(1)$ access
  - Typically: `erase(iterator)`
- Likewise: insertion at a specific point
- **Important: might invalidate the used or some/all other iterators!**

How to remove elements from a singly-linked list?

No back pointers – how to update previous next pointer?

Containers in Standard Library: Overview

- Container: object that stores collection of other objects
- Types of elements specified as template parameter(s)

- Sequential: optimized for sequential access
  - E.g., `std::array`, `std::vector`, `std::list`
- Associative: sorted, optimized for search ($\mathcal{O}(\log n)$)
  - E.g., `std::set`, `std::map`
- Unordered associative: hashed, optimized for search ($\mathcal{O}(n)$, amortized $\mathcal{O}(1)$)
  - E.g., `std::unorderd_set`, `std::unorderd_map`

`std::vector`[112]

- Array that can dynamically grow size
- Elements stored contiguously in memory, access via `data()`
- Preallocates memory for a certain amount of elements (*capacity*)
  - Default: exponential growth; can `reserve()` to reduce reallocations

- Random access: $\mathcal{O}(1)$
- Insert/remove at end: $\mathcal{O}(1)$ (amortized)
- Insert/remove at other position: $\mathcal{O}(n)$

[112] `https://en.cppreference.com/w/cpp/container/vector`

std::vector Example

```
std::vector<int> fib = {1,1,2,3};
assert(fib[1] == 1);
int* fib_ptr = fib.data();
assert(fib_ptr[2] == 2);
fib[3] = 43;
fib.data()[1] = 41; // fib is now 1, 41, 2, 43

fib.push_back(5); // fib is now 1, 41, 2, 43, 5
assert(fib.size() == 5);
assert(fib.back() == 5);
fib.pop_back(); // fib is now 1, 41, 2, 43
auto it = fib.begin(); it += 2;
fib.insert(it, 99); // fib is now 1, 41, 99, 2, 43
it = fib.begin() + 2;
fib.erase(it); // fib is now 1, 41, 2, 43

fib.clear(); // remove all elements
assert(fib.empty());
```

`std::vector` Example

Quiz: What is problematic about this code?

```
#include <vector>
void func(std::vector<int>& v) {
  for (const int& i : v)
    if (i > 1)
      v.insert(v.begin(), -i);
}
```

A. Compile error: Cannot get `const` reference for element.

B. Compile error: `insert()` needs an index as first parameter.

C. Undefined behavior: after the `if` body, an invalidated iterator is used.

D. There is no problem.

std::vector Example

Quiz: How could this code be improved?

```
#include <array>
#include <cstddef>
#include <vector>
template <size_t N> void func(std::vector<std::array<int, N>>& v, int x) {
  std::array<int, N> a;
  for (size_t i = 1; i < N; i++) a[i] = a[i-1] * x + i;
  v.push_back(a);
}
```

A. Instead of copying the array, use `std::move` in `push_back`.

B. Construct the array in-place in the vector, then modify that.

C. Make `a` a reference to reduce stack memory usage.

D. There is nothing to improve.

std::vector: Emplacing Elements

- emplace(_back): construct element in place to avoid copying/moving
- Arguments forwarded to constructor, returns reference to object

```
struct ExpensiveToCopy { /* ... */ };

std::vector<ExpensiveToCopy> v;
ExpensiveToCopy e1;
e1.foo();
v.push_back(e1); // BAD: copy
v.push_back(std::move(e1)); // Better, but might still be expensive

// Best: element constructed in its final place in the vector
ExpensiveToCopy& e2 = v.emplace_back();
e2.foo();
```

std::vector: Reserving Memory

- reserve: size hint to avoid reallocations
- capacity: get currently allocated size

```
std::vector<int> v;

v.reserve(1'000'000); // allocate memory for 1M elements
assert(v.capacity() == 1'000'000);
assert(v.size() == 0); // the vector is still empty!

for (int i = 0; i < 1'000'000; ++i) {
  vec.push_back(i); // no reallocations in this loop
}
```

Quiz: What is problematic about this code?

```
std::vector<int> func(unsigned n) {
  std::vector<int> res;
  res.reserve(n);
  std::vector<int>::iterator it = res.end();
  for (size_t i = 0; i < n; i++) {
    res.push_back(i * i);
    if (i % 3 == 0) it = res.begin() + i;
  }
  res.push_back(*it);
  return res;
}
```

A. Returning a `vector` by value is very expensive.

B. The last `push_back` causes an out-of-bounds write.

C. `it` is invalidated immediately in the next loop iteration.

D. There is no problem.

`std::span`[113]

- Reference to contiguous array of objects; pair of pointer/length
- Supports iteration, subscript, `size()`, `data()`
- `subspan()`: sub-region, no elements copied

```
void printValues(std::span<const int> is) {
  for (auto i : is) std::print("{}␣", i);
}
std::vector<int> values{1, 2, 3, 4};
std::span<int> valuesRef = values;
valuesRef[2] = 4;
printValues(values); // prints "1 2 4 4 "
```

- Prefer `std::span` over reference to `std::array`, `std::vector`, ...
- Pass `std::span` by value (it is already a reference)
- Prefer `std::span<const T>` if possible

[113] https://en.cppreference.com/w/cpp/container/span

std::span Example

Quiz: What is problematic about this code?

```
void func(std::span<const int> cs, std::vector<int>& v) {
  for (int c : cs)
    if (c < 0)
      v.push_back(c);
}
int main() {
  std::vector<int> v{-1, 10, -100, 20};
  func(v, v);
}
```

A. Compile error: Must be `const int c : cs`

B. Passing a vector as `span` precludes passing it as reference at the same time.

C. The `push_back` invalidates the iterator of the loop.

D. There is no problem.

std::unordered_map[114]

- `std::unordered_map<KeyT, ValueT>` (`unordered_map`)
  - Accepts custom hash and comparison functions as extra template parameters
- Container that stores key–value pairs with unique key
- Internally a hash table, amortized $\mathcal{O}(1)$ search/insert/remove

```
std::unordered_map<unsigned, double> grades;
grades[12340001] = 1.3;
grades.insert({12340042, 2.7});
grades.emplace(12340123, 5.0); // emplace = construct in-place
assert(grades[12340042] == 2.7);

auto it = grades.find(12340001); // search
if (it != grades.end()) { // found
  assert(it->first == 12340001); // key
  assert(it->second == 1.3); // value
}
assert(grades.contains(12340001));
```

[114] https://en.cppreference.com/w/cpp/container/unordered_map

Unordered Map: Misleading Usage

Quiz: Which answer is NOT correct?

```
std::optional<double> lookup(std::unordered_map<unsigned, double>& map,
    unsigned key) {
  if (map[key])
    return map[key];
  return -1.0;
}
```

A. `key` is always inserted into the map.

B. If the stored value is zero, `-1` is returned.

C. `map` is not modified and therefore should be a const reference.

D. The map is searched twice, which is avoidable and inefficient.

Unordered Map: Modification

Insertion:

- `operator[]` – get reference to value, insert and default-construct if missing
- `insert` – insert if missing and copy/move construct
  - Returns `std::pair<iterator,bool>`; `second` true iff insertion happened
- `emplace` – construct in-place if missing
- Iterator invalidation: only on rehash

Removal:

- `erase(iterator)/erase(key)` – remove element
  - Iterator invalidation: only iterator for `key`
- `clear` – remove all elements
  - Iterator invalidation: all

`std::map`[115]

- `std::map<KeyT, ValueT>` (`<map>`) – map sorted by keys
- Interface largely similar to `unordered_map`
  - Also supported `upper_bound()`/`lower_bound()` – return iterator to first greater/not lower element
- Internally a tree (typically R/B-tree), $\mathcal{O}(\log n)$ search/insert/remove

- **Only use of *sorted* keys are required!**

[115] https://en.cppreference.com/w/cpp/container/map

`std::unordered_set` and `std::set`

- `std::unordered_set<KeyT>` (`<unordered_set>`) – hash set
- `std::set<KeyT>` (`<set>`) – set sorted by keys
- Largely similar to maps without values
  - Similar internal representation, methods, complexities
- Keys must not be modified

std::string[116]

- std::string (<string>) (alias for std::basic_string<char>)
- Class for (mutable) character sequences
- Manages memory and knows its length (unlike C strings)
- Access to underlying C-string: c_str()
- Prefer std::string over C-style strings (char*)!

```
std::string s; // default-constructs, empty string
assert(s.size() == 0);

std::string s_constructed("my literal");
std::string s_assigned = "hi";
s_assigned[0] = 'H';
std::println("{} {}", s_assigned, s_assigned[1]); // prints: "Hi i"
```

[116] https://en.cppreference.com/w/cpp/header/string

std::string: Null Bytes

Quiz: What is the output of the following program?

```
#include <print>
#include <string>
int main() {
  std::string s1 = "null\0byte";
  std::string s2("null\0byte", 9);
  std::println("{}/{}", s1, s2);
  return 0;
}
```

A. Compile error: String literals cannot include null-bytes

B. Undefined behavior: `std::string` cannot include null-bytes

C. null$_0$byte/null$_0$byte

D. null/null$_0$byte

E. null/null

`std::string`: Operations

- `==`, `<=>`: lexicographical comparison of full strings
- `size()`: number of characters in string
- `empty()`: whether string is empty
- `find()`: offset of first occurrence of substring, or `std::string::npos`

- `append()`, `+=`: append string/char, might cause memory allocation
- `+`: concatenate into new heap-allocated string
- `substr()`: new `std::string` containing substring
  - This is often *not* what you want!

`std::string_view`[117]

- Read-only view on existing string
- Similar to `span<const char>`: just a pointer and a length
- ⇝ Creation, substring, copying is constant time (linear for `std::string`)
- **Prefer** `std::string_view` **over** `std::string`/`std::string&`

```
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)

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
```

117 https://en.cppreference.com/w/cpp/string/basic_string_view

std::string: Implementation

- Different standard libraries have different implementations[118]
- Typically: pointer, size, capacity
  - Pointer (can) to heap memory, deleted on destruction
- Typically: small-buffer optimization
  - Most strings are small, heap allocations are expensive
  - ⇝ Store small buffer (e.g., 15 bytes) inline in `std::string`
  - Downside: more operations invalidate iterators
  - Permitted by C++ standard

[118] https://devblogs.microsoft.com/oldnewthing/20240510-00/?p=109742

Small Buffer Optimization

Quiz: Why does `std::vector` not implement small-buffer optimization?

A. Not very useful $\Rightarrow$ no one implemented it so far.
B. Insertion would no longer be amortized $\mathcal{O}(1)$.
C. Reduce memory usage by not having inline space.
D. Moving a vector must not invalidate iterators.

Containers and Iterators – Summary

- Standard library provides several utility and container templates
- Simple pairs/tuples; can be extracted with structured bindings
- Iterators provide unified pointer-like interface for container element access
- Modifications of containers typically invalidate iterators
- Vector: dynamically sized array, most popular container
- (Unordered) map/set: associative containers
  - Ordered containers typically less efficient
- String: character sequence with managed storage
- String view/span: view into array or string

- Containers good enough to not *immediately* write a custom implementation

Containers and Iterators – Questions

- When do iterators get invalidated? How does this vary for different containers and their operations?
- Why does iterator invalidation frequently cause problems in practice?
- How does a range-based for loop work?
- Why is are unordered maps/sets preferable over ordered maps/sets?
- What are the benefits of `std::string` over C-style strings?
- When to use `std::span`/`std::string_view` and pass them as parameters?
- Why is small-buffer optimization often beneficial/wanted?