ECE 449/590 – OOP and Machine Learning
Lecture 08 Containers and Algorithms

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 19, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 6 – 8
- Next lecture: The Builder Pattern

Outline

Access Elements in Vector

```
std::vector<int> integers;
... // populate the vector
for (size_t i = 0; i < integers.size(); ++i) {
    std::cout << integers[i] << std::endl;
}
```

- Though we access elements using `[]`, the elements are accessed sequentially.
  - The only operations on `i` are to initialize it to `0`, increment it by `1`, and to compare it with the size.
  - We do not access the elements randomly as allowed by `[]`.
- However, the library has no way to know it.
  - A sequence is expressed as a range $[\text{begin}, \text{end})$.
  - If we make that knowledge available to the library, then it is possible to reuse the pattern of asymmetric ranges and loops to visit elements in other containers.

Iterators

- A concept to allow traversing all the elements in a container.
  - Each kind of containers will define C++ types for its OWN iterators.
  - Iterators are generalization of C/C++ pointers.
- An iterator is an object that
  - Identify a container and a place in the container.
  - Allow to access the element at that place if the element is valid.
  - Provide operations for moving between elements in the container.
  - Restrict the available operations in ways that correspond to what the container can handle efficiently.

List Iterators

```
std::list<int> integers;
... // populate the list
for (std::list<int>::iterator iter = integers.begin();
    iter != integers.end(); ++iter) {
    ...
}
```

- The type of iterators for `std::list<T>` is `std::list<T>::iterator`.
  - We usually use `T` to refer to a value type
    - A value type is a type that is not a reference.
  - The iterator type is within the scope of `std::list<T>`.
- `begin()` and `end()`, as suggested by their names, return the either ends of the asymmetric range.
- Operators `==`, `!=`, `++`, `--` are overloaded on iterator types.
  - Comparisons like `<` and `<=` are not always supported.
- So the for loop pattern for the asymmetric range still works.

```
for (index = begin; index != end; ++index)
```

Access Elements using Iterators

```
for (std::list<int>::iterator iter = integers.begin();
    iter != integers.end(); ++iter) {
    std::cout << *iter << std::endl;
}
```

- For a container with n elements, an iterator should represent one of the $n+1$ places with the range $[\text{begin}, \text{end}]$.
- If it is within the asymmetric range $[\text{begin}, \text{end})$, there is a element at the corresponding place.
- The element can be accessed by the dereference operator `*`.
  - Unfortunately `*` is abused (it also stands for multiplication). Anyway, its meaning should be clear from the context.
  - The iterator must be within $[\text{begin}, \text{end})$ (cannot be end for this operation).

Vector Iterators

```
std::vector<int> integers;
... // populate the vector
for (std::vector<int>::iterator iter = integers.begin();
    iter != integers.end(); ++iter) {
    std::cout << *iter << std::endl;
}
```

- Why use iterators when it seems more easy to use indices?
  - (Compile-time) Polymorphism: using iterators allows to process elements in containers in a way independent of container types.
- Why not use iterators for vectors?
  - An previously stored iterator CANNOT be used if any element is inserted/erased from the container.

`auto` Type Deduction

```
for (auto iter = integers.begin(); iter != integers.end(); ++iter) {
    std::cout << *iter << std::endl;
}
```

- It is possible to ask the compiler to deduce the type of a variable for you when it is defined.
  - Use the `auto` keyword.
  - In the above case, the type of `iter` is deduced to be the same as the return type of `integer.begin()`.
- Quite convenient, though you still need to understand the C++ type system to reason with any compiling error.

Range-Based `for` Loops

```
for (int i: integers) {
    std::cout << i << std::endl;
}
```

- It is so common to iterate through a container using the range $[\text{begin}, \text{end})$ that C++ now allows range-based `for` loops.
- The `int i` says to make a copy of the elements in `integers`.
  - Similar to that in a function parameter list.
  - Use `int &i` if we need to modify the elements.
  - Use `const int &i` if we don't need to modify the elements but want to avoid the copy.

`auto` and Range-Based `for` Loops

```
for (auto i: integers) {
    std::cout << i << std::endl;
}
```

- `auto` type deduction works with range-based `for` loops.
- The `auto i` says to make a copy of the elements in `integers`.
  - `auto &i` or `const auto &i` are also valid here for their respective purposes.

Places in a Container

```
std::list<int> integers;          //                          empty list
integers.insert(begin(), 0);      // integers.push_front(0); 0
integers.insert(end(), 1);        // integers.push_back(1);  0,1
*integers.begin() = 2;           // integers.front() = 2;    2,1
*(--integers.end()) = 3;         // integers.back() = 3;     2,3
integers.erase(integers.begin()); // integers.pop_front();    3
integers.erase(--integers.end()); // integers.pop_back();     empty list
```

- The iterator `end()` refers to a place just beyond the last element.
  - Inserting to the place `end()` will append a new element.
  - Dereferencing `end()` would lead to undefined behavior and usually cause failures.
- Prefix `++` and `--` operators return the operand after increment/decrement.
  - `--end()` will return an iterator referring to the last element.
- `.` has higher precedence than `*`.

Invalidation of Iterators

```
std::list<int> integers;                          // empty list
integers.push_back(0); integers.push_back(1); // 0,1
std::list<int>::iterator beg = integers.begin();
integers.push_back(2);                            // 0,1,2
*beg = 3;                                         // 3,1,2
integers.push_front(4);                           // 4,3,1,2
*beg = 5;                                         // 4,5,1,2 instead of 5,3,1,2
integers.erase(beg);                              // 4,1,2   instead of 5,1,2
*beg = 3;                                         // undefined behavior
```

- How about store an iterator and use it later?
  - For `std::list`, the iterator can be used as long as the element remains in the list.
- The iterator will refer to the same element (instead of the same place) even if the element is moved due to insertion/deletion of other elements.
- Once the element is erased, dereferencing the iterator would lead to undefined behavior.
  - And you cannot use the iterator any more.

Outline

Iterators

Library Algorithms

Associative Containers

Library Algorithms

- Containers and iterators provide common interfaces to work with a set of elements.
- The C++ library exploits these common interfaces to provide a collection of standard algorithms.
  - Save your time to write and debug your own
  - Provide a higher abstraction level for your program
- Library algorithms also use consistent interface conventions so we can learn to use all of them by only inspecting a few.
- Most library algorithms are available as functions in the `std` namespace from the standard header `algorithm`.

Sequential Search

- Search a sequence of elements to find one that satisfy some condition.
  - Either return the first element satisfying the condition or indicate no such element exists.
  - How will you design the interface?
- A sequence of elements are represented by an asymmetric range.
  - Sequential search algorithms should take two iterators `begin` and `end` as parameters.
  - The return type should be the same as `begin` and `end`.
  - If the element exists, then the returned iterator is within the asymmetric range.
  - Otherwise, the returned iterator is `end`.
- How to model the condition?

Unary Predicates

- A predicate is a function that returns a boolean value.
- A unary predicate is a predicate that takes one argument.
- We can use a unary predicate which takes an element as the argument to model the condition.
- A unary predicate to decide if an integer is 0

```
bool is_zero(int x) {
    return x == 0;
}
```

  - A predicate should not change the element so the parameters should be either value types or `const` references.
  - Use `const` references to avoid making copies of elements

Use Unary Predicate for Sequential Search

```
bool has_zero(const std::vector<int> &integers) {
    std::vector<int>::const_iterator next_zero = std::find_if(
        integers.begin(), integers.end(), is_zero);
    return next_zero != integers.end();
}
```

- Use `std::find_if` to search for the first element satisfying the predicate.
- Compile-time polymorphism: the code still works if you replace all `vector` with `list`.

const Reference

```
bool has_zero(const std::vector<int> &integers) {
    std::vector<int>::const_iterator next_zero = std::find_if(
        integers.begin(), integers.end(), is_zero);
    return next_zero != integers.end();
}
```

- `integers` is a reference.
  - A reference is an alias of an object – no object is constructed as the function parameter.
  - No overhead for performance
- Use `const` to promise that the function won't change the object passing as the argument.
  - Any change to `integers` within the function body will result in compiling error.

Constness and Iterators

```
bool has_zero(const std::vector<int> &integers) {
    std::vector<int>::const_iterator next_zero = std::find_if(
        integers.begin(), integers.end(), is_zero);
    return next_zero != integers.end();
}
```

- You have promised not to change `integers` by using the `const` reference.
- The compiler can catch you if you try to change the container directly, e.g. by calling `insert()` or `erase()`.
- The compiler can catch you if you try to change the container indirectly by modifying an element through its iterator.
    - By restricting you to use `const_iterator`s only.

Lambda Function

```
bool has_zero(const std::vector<int> &integers) {
    auto next_zero = std::find_if(
        integers.begin(), integers.end(),
        [](int x) {
            return x == 0;
        });
    return next_zero != integers.end();
}
```

- A function can be defined in-place as a lambda function.
  - A lambda function starts with `[]` and has no name.
  - Parameter list and function body work as usual.
  - You don't have to specify the return type as the compiler will deduce it from `return`.
- Improve readability of your code.
  - Use `auto`.
  - Use lambda functions to replace short functions that are self-explanatory.
- Lambda functions are actually much more powerful than usual functions. Will discuss later in the semester if we have time.

Outline

Iterators

Library Algorithms

Associative Containers

Evaluation and Search

- When evaluating a DAG in the SSA form, we will need to maintain containers to store data and to search within.
  - A container to hold keyword arguments where we can search for `Input` values.
  - A container to hold intermediate variables where we can search for operands and store outputs.
  - Containers for more complex operators providing their parameters.
- In essence, we need to search by names and ids.
  - How much time does it take to search a vector or a list?

Sequential Search

- We may use `find_if` as discussed previously.
- On average you need to access $\frac{k}{2}$ elements for a total of k elements in the container.
- How to improve performance?

Sorting and Binary Search

- To search for things efficiently, you always need to sort them.
- If all elements are available before searching starts,
  - Sort the elements in the container according to their names and ids.
  - Perform binary search on the sorted container (must be a vector) to locate the value – on average you need to access $\log k$ elements.
- What if elements need to be updated and searched at the same time?
  - Is there a container supporting efficient search and update?
  - Also hiding details of library algorithms for sorting and binary search even if there is no more update?

Associative Containers

- Containers automatically arrange elements into a sequence depending on the contents of the elements themselves.
  - Instead of the sequence in which we inserted them, as for sequential containers.
- The ordering is further exploited to expedite searches for particular elements.
- Key: the value used for ordering and search
- Key-value pair: in addition to key, each element could contain additional information (value) that is not used for ordering or search
- `std::map`: associative containers that store key-value pairs
  - From the standard header `map`

The C++ Map

```
class evaluation {
    ...
    std::map<std::string, double> kwargs_;
}; // class evaluation
```

- `std::map<Key,Value>` is a template class that requires two types for instantiation.
- The type `Key` can be of any value type whose objects we can compare to keep them ordered.
  - E.g. built-in integral types and `std::string`
  - Never use `float` or `double` as `Key`
- The type `Value` can be of any value type.
  - Provide additional information about the keys
- `std::map` is usually implemented as red-black trees, though we don't need to understand its theory before we can use it.

Insert and Access Key-Value Pairs

```
void evaluation::add_kwargs_double(
    const char *key, double value) {
    kwargs_[key] = value;
}
```

- Similar to Python dictionary, a key-value pair may be inserted or accessed via key using `[]`.
- For our projects, we save the keyword arguments passed from Python for future evaluation.
- The member `kwargs_` would require a `std::string` as the key and the compiler will construct one from the C-style string `key` as needed.

Access Key-Value Pairs Sequentially

```
for (std::map<std::string, double>::iterator it = kwargs_.begin();
    it != kwargs_.end(); ++it) {
    out << "key " << it->first
        << ", value " << it->second << std::endl;
}
```

- The pattern of the `for` loop and the iterators is also applicable for `std::map`.
  - Note that the sequence follows the ordering of keys, e.g. the keyword names for our projects.
- The elements of `std::map` are key-value pairs.
  - The key is the member `first` of the pair.
  - The value is the member `second` of the pair.
  - They can be accessed using iterators. Also recall that `it->first` stands for `(*it).first`.

Access Key-Value Pairs Sequentially (Cont.)

```
for (auto &kv: kwargs_) {
    out << "key " << kv.first
        << ", value " << kv.second << std::endl;
}
```

- It is usually easier to use range-based `for` loops to access elements in containers sequentially.

Search for Elements

```
// try to evaluate an expression expr
if (expr.get_op_type() == "Input") {
    auto it = kwargs_.find(expr.get_op_name());
    if (it == kwargs_.end()) {
        ... // not found
    }
    else {
        double value = it->second;
        ... // found and make use of value
    }
}
```

- The member function `std::map<Key,Value>::find` is used to search for the element with a given key.
  - E.g when you need to find the output of the Input operator in `evaluation::execute`.
- It returns an iterator to the element.
  - Or `end()` if no such element exists.

Other Associative Containers

- Use `std::set<Key>` from the standard header `set` if you are only interested in keys instead of key-value pairs
- Hash tables instead of red-black trees.
  - To emphasize that the elements are not ordered in a way that makes sense to users. They are called `std::unordered_map` and `std::unordered_set`.
  - With the same interface as `std::map` and `std::set`
  - `insert`, `erase`, `find`, `[]` may use less time on average if there is enough memory.
- The idea to manage data as key-value pairs extends to most modern programming languages and is centric to many cloud computing and storage techniques.

Summary

- A sequential container stores a linear sequence of elements.
  - `std::vector` is a kind of sequential containers that is optimized for fast random access.
  - `std::list` is a kind of sequential containers that is optimized for fast insertion and deletion anywhere.
  - Use `iterator`s to access elements in containers sequentially.
- Use library algorithms to improve productivity and readability
  - Use `const_iterator`s for `const` containers.
  - Use `auto` and lambda functions if you are comfortable with them.
- Use associative containers if you need to search frequently.
  - Elements of `std::map` are key-value pairs and are ordered according to the keys.