Containers and algorithms — session 5

Tristan Brindle

Feedback

- We'd love to hear from you!
- The easiest way is via the *cpplang* channel on Slack — we have our own chatroom, *#cpplondonuni*
- Go to https://cpplang.now.sh/ for an "invitation"

Last week

- Iterator categories

- An overview of standard containers

This week

- Finishing the container overview
- Algorithmic complexity
- Intro to STL algorithms

Standard containers overview

- The C++ standard library provides several container classes which can (and should!) be used when writing our own programs

- These can broadly be divided into four categories:
  - Sequence containers: elements are stored in the order they are added
  - Associative containers: keys are *sorted* for fast lookup
  - Unordered associative containers: keys are *hashed* for fast lookup
  - Container adaptors: provide a modified interface for specific tasks

Standard containers overview

- All standard containers are *class templates*

- This means that when you create an *instance* of a container to hold a particular type, it is specialised *just for that type*

- This means that a `std::vector<int>` is not the same type as `std::vector<double>`, and `std::list<std::string>` is not the same type as `std::list<bool>`

- Some containers have parameters which can be used to specialise the behaviour — for example the unordered containers can use a customised hash function

- All the standard containers have an `Allocator` template parameter which can be used to optimise memory allocations in advanced use cases

Standard containers overview

- The standard containers are designed to have a consistent programmer interface

- For example, `std::vector`, `std::list` and `std::deque` all have a `push_back()` member function which appends an element to the end of the sequence

- In particular, all the containers have `begin()` and `end()` member functions which return *iterators*

- They also have `cbegin()` and `cend()` functions which return *const iterators* — these provide read-only access to the container

- See https://en.cppreference.com/w/cpp/container for complete details

Sequence containers

- `std::array`: Fixed-size random-access array
- `std::vector`: Dynamically-sized random-access array
- `std::list`: Doubly-linked list
- `std::forward_list`: Singly-linked list
- `std::deque`: Double-ended queue
- `std::string`: Dynamically-sized random-access array of characters

`std::vector`

- `std::vector` is the standard's version of a dynamically-sized, random-access array
- Properties:
    - Element access is constant time
    - Adding elements to the end of a vector is *amortised constant time*
    - Adding elements anywhere else is *linear* in the number of elements in the vector
    - Iterators are random-access (guaranteed contiguous in C++17)
- `std::vector` should be your default, go-to container for most uses
- **When in doubt, use vector!**

`std::list`

- `std::list` is a doubly-linked list
- Properties:
    - Adding/removing an element anywhere in the list is constant time
    - Accessing a particular element is linear in the number of elements in the list
    - Accessing the `size()` of the list is constant time in C++11
    - Iterators are *bidirectional*
- Note that `std::list` is *node-based*, meaning adding an element requires a dynamic allocation
- Guideline: prefer `std::list` to `std::vector` only when frequently inserting and removing elements from the middle of a *large* sequence

`std::deque`

- `std::deque` is a *double-ended queue*
- Properties:
    - Element access is constant time
    - Insertion or removal at the end **or the beginning** of a deque is constant time
    - Insertion or removal elsewhere is linear in the size of the deque
    - Iterators are random access
- `std::deque` offers the same complexity guarantees as `std::vector`, plus an extra guarantee regarding insertion at the start
- But there's no such thing as a free lunch: `deque`'s operations have a larger constant factor, and there is likely to be extra memory overhead

Associative containers

- `std::map`: A collection of unique key-value pairs, sorted by keys
- `std::set`: A sorted collection of unique keys
- `std::multimap`: A collection of (possibly non-unique) key-value pairs, sorted by keys
- `std::multiset`: A sorted collection of (possibly non-unique) keys

`std::map`

- `std::map` is a sorted container of unique key-value pairs
- Properties:
    - Element access and insertion is `O(log n)`
    - Iterators are *bidirectional*
    - Iteration order is well-defined
- Typically implemented as a *red-black tree*
- By default, keys are sorted using `operator<`. This can be customised using `map`'s `Compare` template parameter
- `std::map` should be your default, go-to associative array type for most purposes

`std::set`

- A `std::set` is a sorted container of unique keys
- Think of it as a map without the values!
- `std::set` can be useful for ensuring that you have a collection of unique elements
- However, you can often achieve better performance by using a `std::vector` and keeping it sorted yourself

Unordered associative containers

- std::unordered_map
- std::unordered_set
- std::unordered_multimap
- std::unordered_multiset

std::unordered_map

- `std::unordered_map` is the standard library's version of a hash table
- Properties:
    - Element access and insertion: `O(1)` average, `O(n)` worst-case
    - Iterators are forward only
    - Iteration order is unspecified
- Compared with `std::map`, `unordered_map` offers constant-time lookup on average
- Keys are compared using `std::hash` by default. This is defined for all built-in types, but you need to specialise it for your own types
- As with all hash tables, performance is highly dependent on the quality of the hash function

Container adaptors

- The container adaptors in the STL take an underlying sequence container and wrap it in a new, more specialised, more restrictive API

- The adapted classes are not containers themselves (they have no `begin()` or `end()`), so in practise are rarely used

- The container adaptors are:

    - `std::queue`: Adapts a container (such as vector) into a FIFO queue

    - `std::stack`: Adapts a container into a LIFO stack

    - `std::priority_queue`: Adapts a container so as to provide constant-time access to the largest element

Exercise

- Clone the repository at
  https://github.com/CPPLondonUni/stl_week4_class_exercise

- Please complete exercise 1 in the README file

Solution

- https://github.com/CPPLondonUni/stl_week4_class_exercise/tree/ex1_solution

Online resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)