module08

Jump to bottom

qingqingqingli edited this page on Jan 28, 2021 · 3 revisions

Table of contents

Intro to STL containers

- A container is a `holder object` that stores a collection os other objects (its elements). They are implemented as `class templates`, which allows a great flexibility in the types supported as elements. The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

- The containers library is a collection of templates and algorithms that implement the common data structures that we work with as programmers. **A container is an object that stores a collection of elements (i.e. other objects)**. Each of these containers manages the storage space for their elements and provides access to each element through iterators and/or member functions.

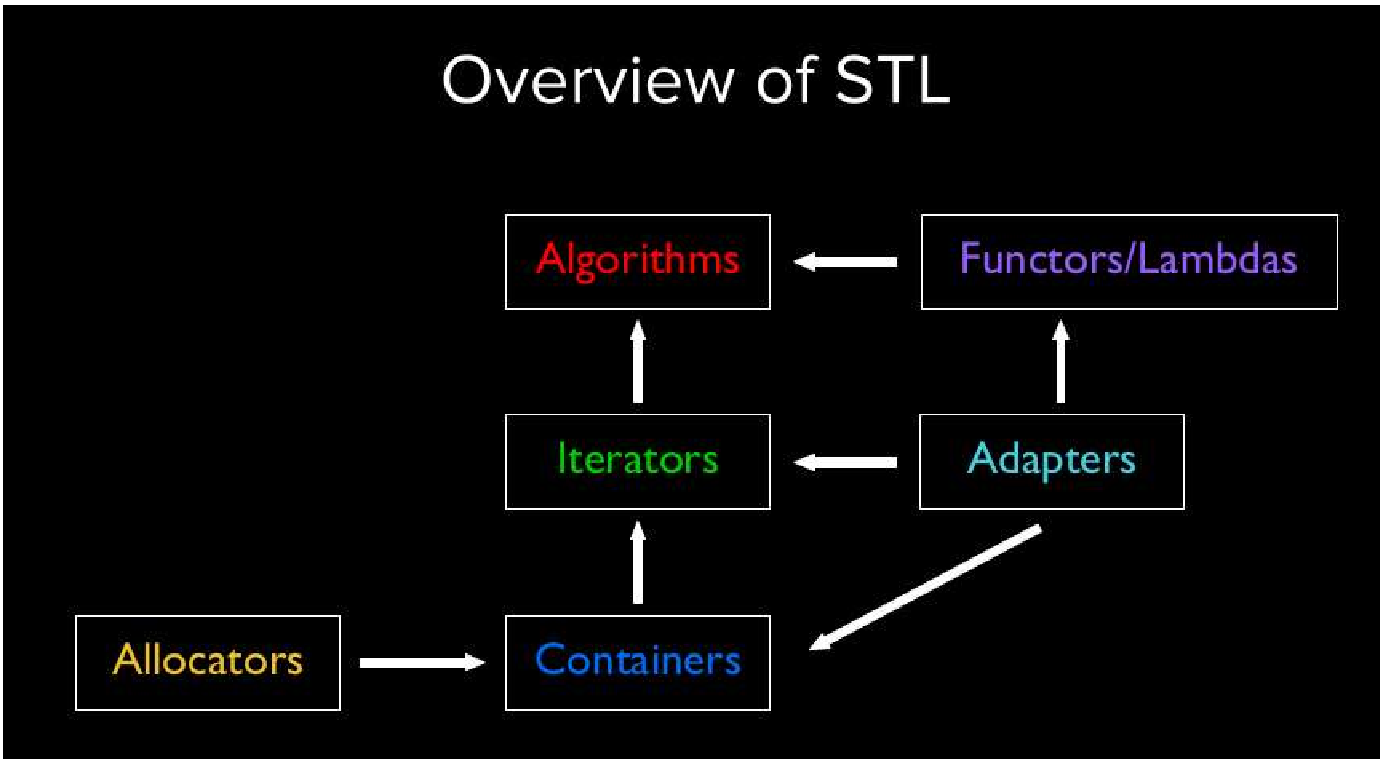
- Containers replicate structures very commonly used in programming: dynamic arrays (`vector`), queues (`queue`), stacks (`stack`), heaps (`priority_queue`), linked lists (`list`), trees (`set`), associative arrays (`map`)...

- **Many containers have several member functions in common, and share functionalities**. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on `the efficiency of some of its members (complexity)`. This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.

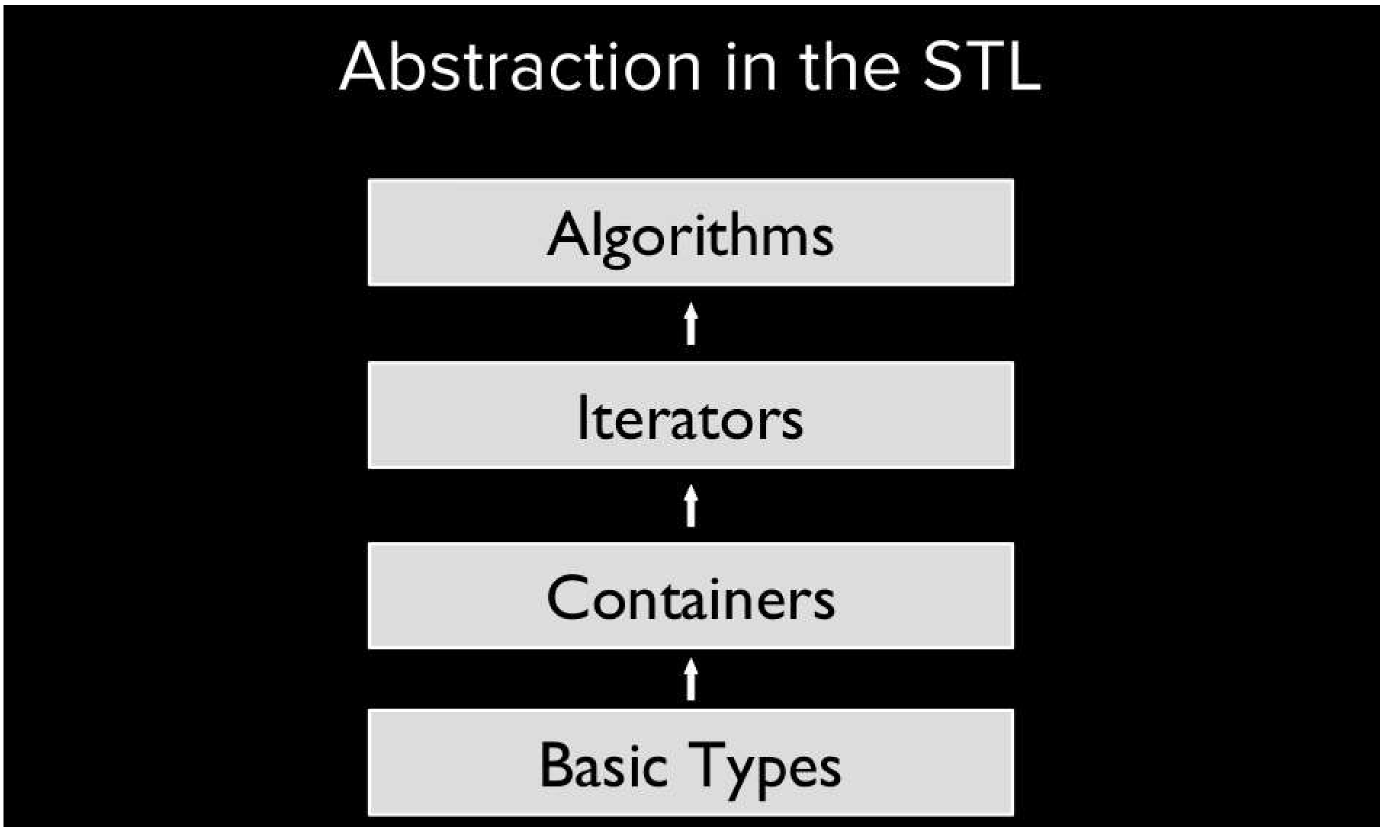
> example

```
#include <iostream>
#include <map>
#include <vector>
#include <list>

class IOperation;

int main()
{
        std::list<int>  lst1;
        std::map<std::string, IOperation*>  map1;
        std::vector<int>    v1; // array to contain what we need
        std::vector<int>    v2(42, 100);

        lst1.push_back(1);
        lst1.push_back(17);
        lst1.push_back(42);

        map1["opA"] = new OperationA;
        map1["opB"] = new OperationB;

        std::list<int>::const_iterator  it; // STL iterator works almost the same way as a
        std::list<int>::const_iterator  ite = lst1.end(); // lst1.end is not the last eleme

        for (it = lst1.begin(); it != ite; ++it)
    {
                std::cout << *it << std::endl;
    }
        return 0;
}
```

> overview of STL



> abstraction of STL



STL containers overview

- The standard containers
  - **Sequence containers** (used for data structures that store objects of the same type in a linear manner)
    - `array` : a static contiguous array
    - `vector` : a dynamic contiguous array

        - `forward_list` : a single-linked list
        - `list` : a doubly-linked list
        - `deque` : a double-ended queue, where elements can be added to the front or back of the queue
    - **Container adapters** (Not full container classes on their own, but wrappers around other container types. They encapsulate the underlying container type and limit the user interfaces accordingly.)
        - `stack` : provides an LIFO data structure
        - `queue` : provides a FILO data structure
        - `priority_queue` : provides a priority queue, which allows for constant-time lookup of the largest element (by default)
    - **Associative containers**
        - keys are unique
            - `set` : a collection of unique keys, sorted by keys
            - `map` : a collection of key-value pairs, sorted by keys
        - Multiple entries for the same key are permitted
            - `multiset` : a collection of unique keys, sorted by keys
            - `multimap` : a collection of key-value pairs, sorted by keys
    - **Unordered associative containers**
        - Keys are unique
            - `unordered set` : a collection of keys, hashed by keys
            - `unordered_map` : a collection of key-value pairs, hashed by keys
        - Multiple entries for the same key are permitted
            - `unordered_multiset` : a collection of keys, hashed by keys
            - `unordered_multimap` : a collection of key-value pairs, hashed by keys

std::vector

- Vectors are `sequence containers` representing arrays that can change in size.
- Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their `size can change dynamically` , with their storage being handled automatically by the container.
- Internally, vectors use a `dynamically allocated array` to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it.

- This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container. Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus `the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size)`.
- Reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity (see `push_back`).
- Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.
- Compared to the other dynamic sequence containers (`deques`, `lists` and `forward_lists`), **vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists**.

std::list

- Lists are `sequence containers` that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.
- List containers are implemented as `doubly-linked lists`; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.
- They are very similar to `forward_list`: The main difference being that forward_list objects are `single-linked lists`, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient.
- Compared to other base standard sequence containers (array, vector and deque), **lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms**.
- The main drawback of lists and forward_lists compared to these other sequence containers is that **they lack direct access to the elements by their position**; For example, to access the sixth element in a list, one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also **consume some extra memory** to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

std::map

- Maps are `associative containers` that store elements formed by a combination of a key value and a mapped value, following a specific order.
- In a map, the key values are generally used to **sort and uniquely identify the elements**, while the mapped values store the content associated to this key. The types of key and mapped value may differ, and are grouped together in member type value_type, which is a pair type combining both: `typedef pair<const Key, T> value_type`;
- Internally, the elements in a map are always sorted by its key following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).
- map containers are generally slower than unordered_map containers to access individual elements by their key, but **they allow the direct iteration on subsets based on their order**.
- **The mapped values in a map can be accessed directly by their corresponding key using the bracket operator (`operator[]`).**
- Maps are typically implemented as binary search trees.

std::stack

- The `std::stack` class is a container adapter that gives the programmer the functionality of a stack (a LIFO (last-in, first-out) data structure).
- The class template acts as a wrapper to the underlying container - **only a specific set of functions is provided**. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack
- `stacks` are implemented as container adaptors, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements
- The following operations are supported:
  - `empty` : check if the stack is empty or not
  - `size` : returns the number of elements present in the stack
  - `push` : insert the element at the top of the stack
  - `pop` : removes single topmost element from the stack. It does not return anything
  - `top` : returns the topmost element of the stack. It returns the element but not removes it.
  - `swap` : swaps the elements of the two stacks

iterators

- An iterator is an object designed to traverse through a container, providing access to each element along the way. A container may provide different kinds of iterators.

- Once the appropriate type of iterator is created, the programmer can then use the interface provided by the iterator to traverse and access elements without having to worry about what kind of traversal is being done or how the data is being stored in the container. And because C++ iterators typically use the same interface for traversal (`operator++ to move to the next element`) and access (`operator* to access the current element`), we can iterate through a wide variety of different container types using a consistent method.
- Iterators let us view a non-linear collection in a linear manner. The whole point of iterators was to **have a standard interface to iterate over data in any container**. But we still had to specify what type of data this iterator was pointing to.
- Any type that satisfy its implicit interface is valid to use with a templatized function.
- Every different collection comes equipped with its own type of iterator. We want to ultimately write generic functions to work with iterators over any sequence. With templates we can!

```
vector<int> v;
vector<int>::iterator itr = v.begin();

vector<double> v;
vector<double>::iterator itr = v.begin();

deque<int> d;
deque<int>::iterator itr = d.begin();
```

- There are 5 different types of iterators:
  - `Input` : For sequential, single pass input. Read only, can only be dereferenced on right side of expression. `int val = *itr;`
  - `Output` : For sequential, single pass output. Write only, can only be dereferenced on left side of expression. `*itr = 12;`
  - `Forward` : same as input/output iterators, except can make multiple passes. Can read from write to (if not const iterator). `int val1 = *itr; itr++; int val2 = *itr;`
  - `Bidirectional` : Same as forward iterators except can also go backwards with `--` . `int val1 = *itr; --itr; int val2 = *itr;`
  - `Random access` . Same as directional iterators except can be incremented or decremented by arbitrary amounts using `+` and `-` . `int val1 = *itr; itr = itr + 3; int val2 = *itr;`
- Common traits among all iterators:
  - Can be created from existing iterator
  - Can be advanced using `++`
  - Can be compared with `==` and `!=`
- **Better to be pre-increment if you don't need the value before it incremented** (source)

- Postincrement must return the value the iterator had BEFORE it was incrementing; so, that previous value needs to be copied somewhere before altering it with the increment proper, so it's available to return.
  - The extra work may be a little or a lot, but it certainly can't be less than zero, compared to a preincrement, which can simply perform the incrementing and then return the just-altered value -- no copying // saving // etc necessary.
  - So, **unless you specifically MUST have postincrement (because you're using the "value before increment" in some way), you should always use preincrement instead**.

iterator adapters

- Sometimes we need to form different types of iterators. They act like iterators (can be dereferenced with `*` and can be advanced with `++`). However, they don't actually point to elements of a container.

- `std::ostream_iterator` : whenever you dereference a `std::ostream_iterator` and assign a value to it, the value is printed to a specified `std::ostream` .

```
std::ostream_iterator<int> itr(cout, ", ")
*itr = 3; // prints 3 to console
++itr;
*itr = 1729; // prints 1729 to console
++itr;
*itr = 13; // prints 13 to console
```

- With this, you can treat streams like iterators and use algorithms with them.

```
std::vector<int> v{3, 1, 4, 1, 5};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(cout, ", "))
```

- The STL provides insert iterators (`std::inserter` , `std::back_inserter` , `std::front_inserter`). Writing to these iterators inserts the value into a container using one of the `insert` , `push_back` , or `push_front`

> example: insert value to a vector

```
std::vector<int> v; // empty vec
auto itr = std::back_inserter(v);

*itr = 1729; // does v.push_back(1729)
++itr;
*itr = 13; // does v.push_back(13)
++itr;
*itr = 3; // does v.push_back(3)
```

```
// v look like this: {1729, 13, 3}
```

> example: copy value one by one to a vector

```
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy; // start with an empty vector
std::copy(v.begin(), v.end(), std::back_inserter(vCopy));
```

Algorithms

- The STL containers pre-written algorithms that operate on `iterators` . Doing so lets them work on many types of containers. Uses are determined by `types of iterators` . Rely heavily on `templates`
- We won't always know how much space will be needed for the destination. We want to be able to copy into a collection by "inserting" into it, rather than making space for it first (eg. pushback).

> example

```
#include <iostream>
#include <algorithm>
#include <list>

void displayInt(int i)
{
        std::cout << i << std::endl;
}

int main()
{
        std::list<int>  lst;

        lst.pushback(10);
        lst.pushback(23);
        lst.pushback(3);
        lst.pushback(17);
        lst.pushback(20);

        for_each(lst.begin(), lst.end(), displayInt);

        return (0);
}
```

> result

```
10
23
3
17
20
```

resources

- containers cppreference
- An Overview of C++ STL Containers
- C++ Magicians STL Algorithm
- algorithm functions
- Standford class on template
- Standford class on algorithm

Clone this wiki locally

```
https://github.com/qingqingqingli/CPP.wiki.git
```