Overview of Standard Library containers

Tom Latham

THE UNIVERSITY OF WARWICK

The story so far...

- So far we have encountered a single example of each of the two main categories of containers:
  - `std::vector` is a **sequence** container
  - `std::map` is an **associative** container

- In these few slides I'll mention the other available containers within each of those two categories as well as those in the **unordered associative** container category

- I'll also try to give you an idea about the pros and cons of each container type, which will hopefully guide you as to when it is best to use each one

- I'll also provide guidance on how to most efficiently use vectors and maps

Sequence containers

- The sequence containers are perhaps the simplest to understand
- They hold a number of elements in a particular order
- The main differences between the various types lie in the efficiency (i.e. speed) of performing certain operations, in particular: accessing, inserting, removing and moving elements
- For example, the `std::vector`, which we've already met, has very efficient random access to its elements, and addition/removal of elements to/from the end of the container is also efficient, but addition/removal anywhere else is not very efficient and neither is changing the order of the elements

Sequence containers

- The different types available are:
  - `std::vector`
  - `std::array`
  - `std::deque`
  - `std::list`
  - `std::forward_list`

Dynamically sized, contiguous array.

i.e.
- Its size is defined at run time
- Its elements are stored consecutively in memory

Fast random access to elements, e.g. through `v[4]`, and fast addition to / removal from the end (`push_back` and `pop_back` functions).

Operations to insert/remove anywhere else are linear with distance from the end.

Sequence containers

- The different types available are:
  - `std::vector`
  - `std::array`
  - `std::deque`
  - `std::list`
  - `std::forward_list`

Statically sized, contiguous array.

i.e.
- Its size is defined at compile time
- Its elements are stored consecutively in memory

Essentially it is a fixed-size version of `std::vector` and therefore is very slightly more efficient for element access but adding/removing elements is not supported.

It has no memory overhead since it is statically sized.

Sequence containers

- The different types available are:
  - `std::vector`
  - `std::array`
  - `std::deque`
  - `std::list`
  - `std::forward_list`

A double ended queue.

Like an `std::vector`, its size is defined at run time and it provides fast random access to elements, e.g. through `q[4]`.
It also has fast addition to / removal from the end *and beginning* (`push_back`, `push_front`, `pop_back`, `pop_front` functions).
Operations to insert/remove anywhere else are linear with size.

Unlike `std::vector`, its elements are not all stored consecutively in memory.
Means that expansion is cheaper since no copying has to take place.
Comes at price of slightly larger memory overhead.

Sequence containers

- The different types available are:
  - `std::vector`
  - `std::array`
  - `std::deque`
  - `std::list`
  - `std::forward_list`

A doubly-linked list.

Fast insertion/removal/moving of elements anywhere in the container. As such it implements specialisations of several algorithms, such as `remove_if` and `sort`, as member functions.

But comes at price of having no random access to elements – you have to iterate through from either beginning or end. Results from fact that elements are not stored contiguously.

Sequence containers

- The different types available are:
  - `std::vector`
  - `std::array`
  - `std::deque`
  - `std::list`
  - `std::forward_list`

A singly-linked list.

As per `std::list` but can only iterate forwards through the elements.

Makes it slightly more memory-efficient than `std::list`.

Efficient use of vectors – how do they grow?

- The vector is very likely the container you'll use most often but it's very easy to use them in an inefficient manner

- They are designed to keep their memory use roughly to a minimum and so you might get a bit of surprise if you want to store a very large number of elements

- Checkout the **ContainerEfficiency** branch of the SpecialMemberFunctions repository, build and run the **efficient-vector** program – see how the capacity and used size change

- Each time the capacity changes there is a reallocation of memory and all the elements are copied or moved into the new memory

- Edit the efficient-vector.cpp file to uncomment the whatOperationsHappenDuringDefaultGrowth() function, build and run, to see these operations reported

Efficient use of vectors – reserving memory

- If you know how many elements you are likely to want in your vector you can specify this using the reserve() member function:

```
std::vector<int> myvec;
myvec.reserve(1000);
```

- This performs a single allocation of 1000 elements-worth of memory
- Until we push_back more than 1000 elements there will be no need for further allocation or copying/moving of elements
- Edit the efficient-vector.cpp file to uncomment the whatOperationsHappenDuringGrowthWithReserve() function, build and run, to see how the operations reported have changed
- If you're worried that you've reserved too much memory you can use shrink_to_fit() once you've finished putting elements into the vector – although this might perform one reallocation and copy/move operation, so you need to strike a balance between this and having a bit of wasted memory

Efficienct use of vectors – emplace_back

- While the number of operations has dropped considerably, we still have to construct a temporary object, copy it into the vector and then destroy the temporary

- Wouldn’t it be better if we could just construct the element in place inside the vector?

- This is where emplace_back(…) comes in – instead of providing an already-constructed object, you instead provide the arguments to be used to construct it and the construction is done in place

- Edit the efficient-vector.cpp file to uncomment the whatOperationsHappenDuringGrowthWithReserveAndEmplaceBack() function, build and run, to see how the operations reported have changed

Associative containers

- The associative containers associate a **key** with a **value**
- The elements are sorted by the key at time of insertion
- Elements are accessed by searching for the key
- Insertion, removal and searching operations all have O(log n) complexity

Associative containers

- The different types available are:
  - `std::set`
  - `std::map`
  - `std::multiset`
  - `std::multimap`

A sorted collection of unique keys.

If you attempt to insert an element that already exists in the container it will not do so and return a corresponding flag.

Useful for e.g. storing the letters already encountered when processing the Playfair cipher key.

Associative containers

- The different types available are:
  - `std::set`
  - `std::map`
  - `std::multiset`
  - `std::multimap`

A collection of key-value pairs, sorted by the unique keys.

The obvious example from the cipher code is in the Vigenère cipher, where the various Caesar cipher objects are stored associated with the particular letter in the key word.

Associative containers

- The different types available are:
  - `std::set`
  - `std::map`
  - `std::multiset`
  - `std::multimap`

A sorted collection of non-unique keys.

i.e. exactly like the `std::set` but each key can be stored more than once

Associative containers

- The different types available are:
  - `std::set`
  - `std::map`
  - `std::multiset`
  - `std::multimap`

A collection of key-value pairs, sorted by the non-unique keys.

i.e. exactly like the `std::map` but can have more than one entry with the same key

Unordered associative containers

- Essentially the same as the associative containers but they use hashing instead of sorted data structures to store the information

- Elements are accessed by searching for the key

- Insertion, removal and searching operations all have O(1) complexity, although the hashing does introduce some latency

- Available types are:
  - `std::unordered_set`
  - `std::unordered_map`
  - `std::unordered_multiset`
  - `std::unordered_multimap`

Tuples

- Not strictly a container but rather is part of the utilities library
- The `std::tuple` is a generalisation of `std::pair`, i.e. it is a collection of many (number fixed at compile time) heterogeneous values
- Helper functions exist to create tuples (`std::make_tuple`), to access the individual elements (`std::get`) and to unpack the whole tuple (`std::tie`)
- Example on right based on:

  http://en.cppreference.com/w/cpp/utility/tuple

tuple-example.cpp (~/Documents/Teaching/2015-16/Day6-Slides) - VIM

```cpp
#include <tuple>
#include <iostream>
#include <string>

std::tuple<double, char, std::string> get_student(const int id)
{
  if (id == 0) { return std::make_tuple(3.8, 'A', "Lisa Simpson"); }
  if (id == 1) { return std::make_tuple(2.9, 'C', "Milhouse Van Houten"); }
  if (id == 2) { return std::make_tuple(1.7, 'D', "Ralph Wiggum"); }
  // etc. (and should handle case of invalid ID)
}

int main()
{
  auto student0 = get_student(0);
  std::cout << "ID: 0, "
            << "GPA:   " << std::get<0>(student0) << ", "
            << "grade: " << std::get<1>(student0) << ", "
            << "name:  " << std::get<2>(student0) << '\n';

  double gpa1;
  char grade1;
  std::string name1;
  std::tie(gpa1, grade1, name1) = get_student(1);
  std::cout << "ID: 1, "
            << "GPA:   " << gpa1   << ", "
            << "grade: " << grade1 << ", "
            << "name:  " << name1  << '\n';
}

~
~
<hing/2015-16/Day6-Slides/tuple-example.cpp" 30L, 895C written 28,24        All
```

Efficient use of maps

- The map does not have quite the same potential pitfalls as vector - the elements are stored completely separately in memory and allocated one by one, so there is no need to reserve memory in advance
- However, there is still the issue of how to most efficiently insert elements into the container
- Build and run the **efficient-map** program – see which operations are performed when using the subscript operator (square brackets) to perform insertions
- The other functions show:
  - The behaviour when using insert() together with make_pair()
  - The behaviour when using emplace()
- The last of the emplace() examples, piecewise construction of the pair in-place by forwarding the arguments as a tuple of references, is apparently the most efficient – it simply calls the constructor once (the bear minimum that is needed!)
- However, it is always best to double check these things with a profiler!