STL and run-time speed

(C) Richèl Bilderbeek

March 23, 2015

Chapter 1

Introduction

1.1 Overview

- Introduction
- Containers
- Algorithms
- Conclusion

1.2 Goal

- Do some basic container use
- Encounter some trade-offs in container choice
- Use some algorithms
- Obtain some ideas about chosing the right algorithm
- The class extensions needed to put classes in containers or use these in algorithms

Chapter 2

Containers

2.1 Question

- Name some containers.
- Distinguish between STL or non-STL, and standard or non-standard

2.2 Answer

- STL sequence containers:
  - std::string, std::vector, std::list, std::deque, std::stack
- STL associative containers:
  - std::set, std::map, std::multi_set, std::multi_map
- Standard non-STL containers:
  - std::bitset, std::valarray, std::queue, std::priority_queue
- Nonstandard non-STL containers (for C++98):
  - std::tr1: slist, rope, hash_set, hash_map
  - Boost: array, circular_buffer, dynamic_bitset, graph, multi_array
  - More

2.3 Question

- Which 'container' is native to the (C and) C++ language?
- Which STL container uses this internally?

2.4 Answer

- Which 'container' is native to the (C and) C++ language?
  - array
  - arrays are evil[1] [2]
- Which STL container uses this internally?
  - std::vector
  - use std::vector by default[3]

[1] Marshall Cline, Greg Lomow and Mike Girou. C++ FAQs. ISBN: 0-201-3098301, FAQ 28.02: 'Are arrays good or evil?' (Answer: 'Arrays are evil'

[2] Bjarne Stroustrup. The C++ Programming Language (3rd edition). Chapter C.14.11 'Prefer vector over array'

[3] Herb Sutter and Andrei Alexandrescu . C++ coding standards: 101 rules, guidelines, and best practices. Chapter 76: 'Use vector by default. Otherwise choose an appropriate container'

2.5 Question

- Which fancy STL container is best at all of the following?
  - random access reading and writing
  - looking up elements
  - random access insertion and removal

2.6 Answer

- None: the characteristics are mutually exclusive
- There will be trade-offs
- Beware the illusion of container-independent code[4]

[4] Meyers, Effective STL, item 2

2.7 Question

- How to sort Persons?

```
struct Person
{
  std::string name;
  double money;
};

int main()
{
  //Sort Persons
}
```

2.8 Answer

- Define the global operator

```
bool operator<(const Person& lhs, const Person& rhs);
```

- Define a custom functor

```
struct SortOnMoney {
  bool operator<(const Person& lhs, const Person& rhs) const
};
```

2.9 Exercise

```
struct Person
{
  std::string name;
  double money;
};

int main()
{
  //Create some Persons
  std::vector<Person> v /* */;
  //Sort Persons on the amount of money they have
  std::sort( /* */ );
}
```

2.10 Question

- Describe the implemention of std::vector
- What are the consequences of this?

2.11 Answer: std::vector

- dynamically allocated array
- contant-time access to elements
- linear-time insertion-removal
- linear-time searching

2.12 Exercise: std::vector

- Create a class Gossip that prints when it is copied
- Create an empty std::vector<Gossip>
- Append 32 Gossips. How many copies are made?
- Insert a Gossip at the front. How many copies are made?
- Write a C-style function that works on an array of Gossips and swaps the first and last element

```
void SwapFirstAndLast(
  Gossip * const gossip_array,
  const int size);
```

- Call SwapFirstAndLast and check if it did what you expected

2.13 Conclusion

- Use std::vector<T>::push_back to append to std::vectors, as it is amortized constant-time
- Avoid inserting elements in the front or middle of a std::vector
- std::vector can be used to communicate with C style functions

2.14 Question

- Describe the implemention of std::list
- What are the consequences of this?

2.15 Answer: std::list

- Next to the data itself, each element has two pointers: to the next and previous element in a sequential list
- constant-time insertion and removal
- linear-time access to elements
- linear-time searching

2.16 Exercise: std::list

- Create a class Gossip that prints when it is copied
- Create an empty std::list<Gossip>
- Append 32 Gossips. How many copies are made?
- Insert a Gossip at the front. How many copies are made?
- Compare std::sort on a std::list and std::list<T>::sort. Does one call the other?

2.17 Conclusion

- When adding elements to a std::list, no additional copies need to be made
  - Not at the end, middle, not beginning
- A std::list is scattered through memory
  - Calculating the number of elements is an O(n) calculation

2.18 Question

- Describe the implemention of std::set
- What are the consequences of this?

2.19 Answer

- Next to the data itself, each element has three pointers: to the parent, left and right branch in a red-black tree
- contents always sorted
- logarithmic-time insertion and removal
- logarithmic-time access to elements
- logarithmic-time access searching

2.20 Exercise

- Create a class Person that prints when it is copied and has at least two member variables (why two?)
- Create an empty std::set<Person>
- Put in 4 different Persons. How many copies are made?
- Create a new unique person. Use std::set<T>::count to check he/she is not present yet. Insert the Person and check he/she is present

2.21 Conclusion

- A std::set keeps its elements ordered
  - need to define operator<

Chapter 3

Algorithms

3.1 Algorithms question

- What are algorithms?
- Why use algorithms?

3.2 Algorithms answers

- What are algorithms?
  - named operations on multiple elements
- Why use algorithms?
  - verbosity/readability
  - algorithms are written by experts
  - algorithms are standarized: common idiom
  - increase run-time speed: naive for-loops might result in higher Big-O

3.3 Algorithm example

```cpp
template<typename In, typename Out, typename Pred>
Out MysteryAlgorithm(In first, In last, Out res, Pred Pr)
{
  while (first != last)
  {
    if (Pr(*first)) *res++ = *first;
    ++first;
  }
  return res;
}
```

3.4 Question

- Which sorting algorithms exists?
- When to use which one?

3.5 Answer

- std::sort: when the whole range needs to be sorted
- std::partial_sort: when you only need the top-x values in a sorted order
- std::nth_element: when you only need the top-x values
- With 'stable_' added: when the order of equivalent items needs to be preserved
- If a container has a member function with the same name, always use that one

3.6 Exercise 1/4

- Create a big initialized randomly-shuffled std::vector<int>
- Write the following functions that obtain the three lowest values:
  - GetMinThreeUsingPartial_sort
  - GetMinThreeUsingNth_element
- Check if the two functions return the same top three
- Display the top three

3.7 Exercise 2/4

- Create a big initialized randomly-shuffled std::vector<int>
- Write the following functions that obtain the three highest values:
  - GetMaxThreeUsingPartial_sort
  - GetMaxThreeUsingNth_element
  - Hint: look up std::greater
- Check if the two functions return the same top three
- Display the top three

3.8 Exercise 3/4

- Create a Person class. Every Person has two member variables:
  - a std::string called 'name'
  - a double called 'money'
- Create a big initialized randomly-shuffled std::vector<Person>
- Write the functions that obtain the three persons with the least money:
  - GetMinThreeUsingPartial_sort
  - GetMinThreeUsingNth_element
  - Bonus: re-use the existing ones
- Check if the two functions return the same top three
- Display the top three

3.9 Exercise 4/4

- Create a Person class. Every Person has two member variables:
  - a std::string called 'name'
  - a double called 'money'.
- Create a big initialized randomly-shuffled std::vector<Person>
- Write the functions that obtain the three persons with the most money:
  - GetMaxThreeUsingPartial_sort
  - GetMaxThreeUsingNth_element
- Check if the two functions return the same top three
- Display the top three

3.10 Question

- Which searching algorithms exists?
- When to use which one?

3.11 Answer

- std::find: find an element in an unsorted container
- std::find_if: find the first element in an unsorted container for which a predicate is true
- std::search: find a sequence of elements
- std::search_n: find an n-times-repeated sequence of elements
- std::binary_search: find an element in an assumed-to-be-sorted container
- std::adjacent _find: find two adjacent equal values
- std::lower_bound, std::upper_bound: find first/last value in a sorted container
- std::min_element, std::max_element: find min or max element

- If a container has a member function with the same name, always use that one

3.12 Exercise: std::find

- Create a Person class. Every Person has two member variables:
  - a std::string called 'name'
  - a double called 'money'
- Put some Persons in a std::vector, std::list and std::set
- Create a new unique person with the name 'Mr X' and put him in each of these containers
- Shuffle these containers and try to retrieve the Person with name 'Mr X'
- Create a new unique person with 123.45$ and put him/her in each of these containers
- Shuffle these containers and try to retrieve the Person with 123.45$

- Find the person with the most money

3.13 Question

- Which algorithms is used to summarize a range?

3.14 Answer

- std::accumulate
- (found in numeric.h)

3.15 Exercise

- Create a Person class. Every Person has two member variables:
  - a std::string called ’name’
  - a double called ’money’
- Put some random Persons in a container
- Obtain the sum of their money

3.16 Question

- What is a predicate?
- Which algorithms use a predicate?

3.17 Answer

- A predicate is a functor returning true or false. Identical input should yield the same results
- Algorithms:
  - std::partition
  - every std::[something]_if

3.18 Exercise

- Create a Person class. Every Person has two member variables:
  - a std::string called 'name'
  - a double called 'money'
- Put some random Persons in a container, of which some are poor (money < 1000.0) and some are rich (money >= 1000.0)
- Find a way to seperate poor and rich persons in a container
- Display the poor persons
- Display the rich persons

Chapter 4

Conclusion

4.1 Conclusion

- Choose your containers with care
  - Use std::vector by default
- Choose your algorithms with care
  - Choose the simplest algorithm possible

Chapter 5

EOF