COMP6771
Advanced C++ Programming

Week 2.1
STL Containers

Libraries

Most of us are quite familiar with libraries in software. For example, in COMP1511, we've used <stdio.h> and <stdlib.h>.

Being an effective programmer often consists of the effective use of libraries. In some ways, this becomes more important than being a genius at writing code from scratch (Don't reinvent the wheel!).

While there are many libraries that can be used with C++, the Standard Template Library is the one we will focus on.

STL: Standard Template Library

- STL is an architecture and design philosophy for managing generic and abstract collections of data with algorithms
- All components of the STL are templates
- Containers store data, but don't know about algorithms
- Iterators are an API to access items within a container in a particular order, agnostic of the container used
  - Each container has its own iterator types
- Algorithms manipulate values referenced by iterators, but don't know about containers

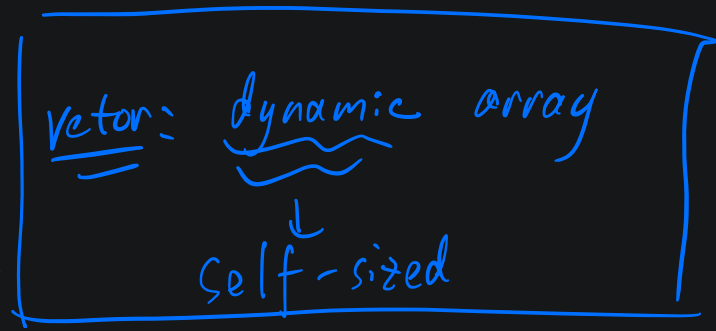
Iterating through a basic container

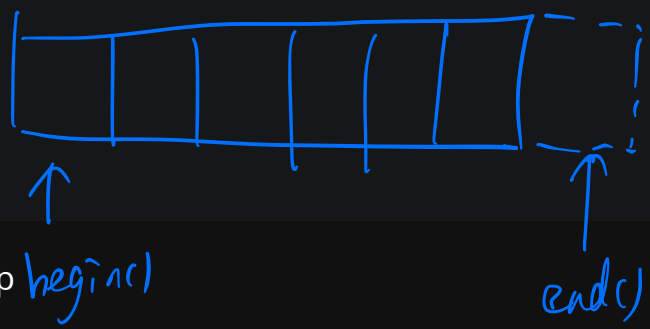
```cpp
#include <array>
#include <iostream>

int main() {
  // C-style. Don't do this
  // int ages[3] = { 18, 19, 20 };
  // for (int i = 0; i < 3; ++i) {
  //   std::cout << ages[i] << "\n";
  // }

  // C++ style. This can be used like any other C++ container.
  // It has iterators, safe accesses, and it doesn't act like a pointer.
  std::array<int, 3> ages{ 18, 19, 20 };

  for (unsigned int i = 0; i < ages.size(); ++i) {
    std::cout << ages[i] << "\n";
  }
  for (auto it = ages.begin(); it != ages.end(); ++it) {
    std::cout << *it << "\n";
  }
  for (const auto& age : ages) {
    std::cout << age << "\n";
  }
}
```





demo201-vec-iter.cpp

Sequential Containers

Organises a finite set of objects into a strict linear arrangement.

`std::vector`	Dynamically-sized array.
`std::array`	Fixed-sized array.
`std::deque`	Double-ended queue.
`std::forward_list`	Singly-linked list.
`std::list`	Doubly-linked list.

```
auto v = std::vector<int>{1, 2, 3, 4};
std::cout << v.size() << " " << v.capacity() << "\n";
v.push_back(5);
std::cout << v.size() << " " << v.capacity() << "\n";
v.pop_back()
v.pop_back()
v.pop_back()
v.pop_back()
std::cout << v.size() << " " << v.capacity() << "\n";
```

```
4 4
5 8
1 8
```

We will explore these in greater detail in Week 10.

It won't be necessary to use anything other than std::vector in COMP6771.

Another look at Vector

- Array-like container most used is <vector>
  - Abstract, dynamically resizable array
  - In later weeks we will learn about various ways to construct a vector

```cpp
#include <iostream>
#include <vector>

// Begin with numbers 1, 2, 3 in the list already
int main() {
  // In C++17 we can omit the int if the compiler can determine the type.
  std::vector<int> numbers {1, 2, 3};
  int input;
  while (std::cin >> input) {
    numbers.push_back(input);
  }
  std::cout << "1st element: " << numbers.at(0) << "\n"; // slower, safer
  std::cout << "2nd element: " << numbers[1] << "\n"; // faster, less safe
  std::cout << "Max size before realloc: " << numbers.capacity() << "\n";
  for (int n : numbers) {
    std::cout << n << "\n";
  }
}
```

checks if index out of range

demo202-vec-obj.cpp

Ordered Associative Containers

Provide fast retrieval of data based on keys. The keys are sorted. A **value** is accessed via a **key**.

`std::set`	A collection of unique keys.
`std::multiset`	A collection of keys.
`std::map`	Associative array that map a unique keys to values.
`std::multimap`	Associative array where one key may map to many values.

They are mostly interface-compatible with the unordered associative containers.

std::map example

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

int main() {
  std::map<std::string, double> m;
  // The insert function takes in a key-value pair.
  std::pair<std::string, double> p1{"bat", 14.75};
  m.insert(p1);
  // The compiler will automatically construct values as
  // required when it knows the required type.
  m.insert({"cat", 10.157});
  // This is the preferred way of using a map
  m.emplace("cat", 10.157);

  // This is very dangerous, and one of the most common causes of mistakes in C++.
  std::cout << m["bat"] << '\n';

  auto it = m.find("bat"); // Iterator to bat if present, otherwise m.end()

  // This is a great example of when to use auto, but we want to show you what type it is.
  for (const std::pair<const std::string, double>& kv : m) {
    std::cout << kv.first << ' ' << kv.second << '\n';
  }
}
```

auto m = std::map<std::string, double>{}; better

better, constructed in-place.

use find or at

ordered

demo203-map.cpp

Unordered Associative Containers

Provide fast retrieval of data based on keys. The keys are hashed.

`std::unordered_set`	A collection of unique keys.
`std::unordered_map`	Associative array that map unique keys to a values.

Container Performance

- Performance still matters
- STL containers are abstractions of common data structures
- cppreference has a summary of them here.
- Different containers have different time complexity of the same operation (see right)

Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

O(1)+ means amortised constant time

Feedback