Lab Class Scientific Computing 2022, WISM454

Adriaan Graas, Week 10

C++ programming

std::vector

Example of std::vector

```cpp
#include <vector>

// Data on the heap, so dynamic size. (C-style arrays are on the stack).
std::vector<int> v {7, 5, 16, 8};

v.push_back(25);  // insert at the end (constant time)
v.insert(v.begin() + 2, 25);  // insert at place 3 (slow, all data moves)

// loop example with `size_t` (a type guaranteed to handle the max. array size)
for (size_t i=0; i < v.size(); ++i) {
    std::cout << v[i] << " ";
}
```

A few example of std::vector **initialization**

```
// - Initialization from “brace-enclosed list”
std::vector<int> some_numbers = {7, 5, 16, 8};

// - Initialization with a fixed size
auto v = std::vector<int>(6);  // integers are zero-initialized

// - Initialization using 3 copies of a copyable object
//   (Suppose here that HitOrMiss is copyable)
std::vector<HitOrMiss> hello {3, HitOrMiss(...)};
```

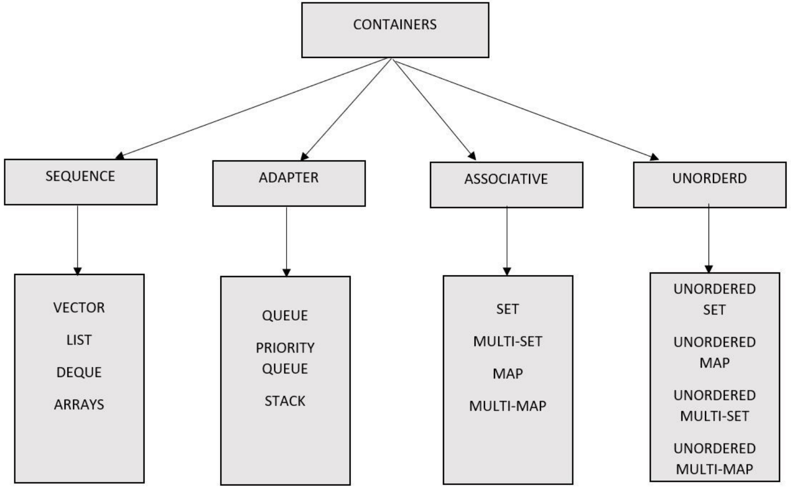
Range-based for loops

```cpp
// a vector of functions: int -> int: `std::function<int(int)>`
auto funcs = std::vector<std::function<int(int)>>(...);

// easy iteration with a range-based for loop
int sum = 0;
for (auto func : funcs) {
    sum += func(1);
}
```

Other containers

- std::vector is the most common “container type”
    - **Other containers have different qualities**
        - std::list has O(1) insertion
        - std::set is always ordered value-wise
        - std::stack is a LIFO stack
        - std::array wraps a C-style array
- Overview of functions:
    - **https://hackingcpp.com/cpp/std/vector.html**
    - **https://en.cppreference.com/w/cpp/container/vector**

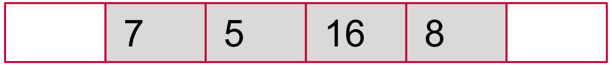


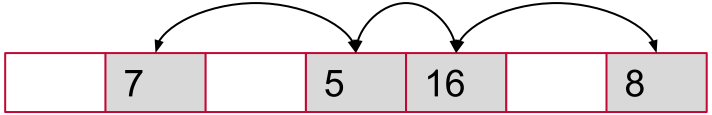
C++ programming

Iterators

Container types and memory

- std::vector
    - **An array that is dynamic (automatically expands).**
    - **Elements are stored contiguously in memory.**
        - *Access* to any element is fast (constant).
        - *Insertion/removal in the middle* is slow, as all the elements in memory need to be moved.
- std::list
    - **Every element in the list stores pointers to the previous and next element.**
    - **Elements may be noncontiguously stored.**
        - *Access* to arbitrary element is slow, as the list must be traversed.
        - *Insertion/removal* is always fast.

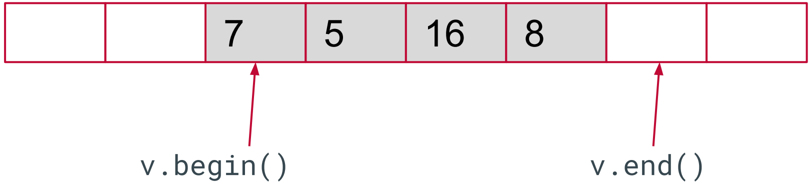




Iterators are a generic way of traversing containers

```
int main() {
    std::vector<int> v = {7, 5, 16, 8};

    // an "iterator" is a type to help traversing the vector
    auto it = v.begin();              // type std::vector<int>::iterator
    ...
}
```



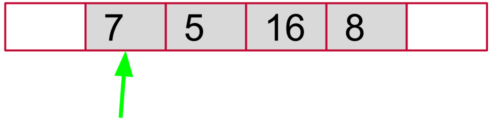
		7	5	16	8		

v.begin()

v.end()

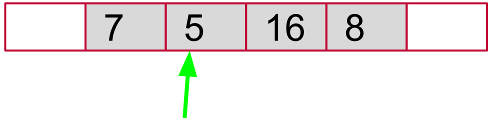
Iterators are a generic way of traversing containers

```
std::vector<int> v = {7, 5, 16, 8};
auto it = v.begin();
```

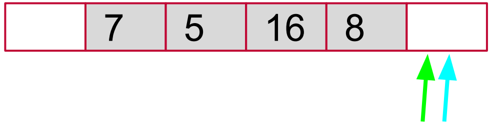


```
// iterators can be dereferenced
std::cout << *it; // prints "7"

// and incrementing moves the iterator
it++;
```



```
// and can be compared to other iterators
auto it2 = v.end();
while (it != it2) { it++; }
```



STL functions often expect iterators, rather than containers

```
std::vector<int> v = {7, 5, 16, 8};

std::sort(v.begin(), v.end(), std::greater<>());  // Sorting descending (>)
                                                     {16, 8, 7, 5}


auto it = std::next(v.begin(), 3);                // Shuffling first 3 elements
std::shuffle(v.begin(), it);                         {7, 16, 8, 5}

auto it2 = std::prev(v.end(), 2);                 // Filling last 2 elements
std::fill(it2, v.end(), 42);                         {7, 16, 42, 42}
```

Iterator taxonomy

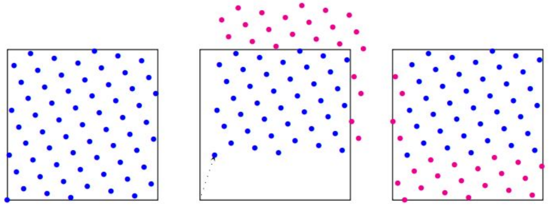
- Iterator taxonomy:

Forward iterator	it++
Bi-directional iterator	it++ and it--
Random access iterator	can jump, e.g. it += 7 to go 7 places at once
Input iterator	read-only
Output iterator	write-only

- Containers types expose iterators that are in line with their capabilities. For example:
    - **std::vector** returns a *random access iterator*.
        - Easy to grab a value from any spot in memory at once.
    - **std::list** returns a *bi-directional iterator.*
        - Traversing left or right in the array is easy, but jumping is not.

This week

- Today / this week:
  - **Multi-dimensional Monte Carlo methods**
    - Hit-or-miss and Simple-sampling Monte Carlo in *d* dimensions
    - Sampling from non-rectangular domains
    - Sampling from a low-discrepancy lattice rule



- Next week:
  - **Template metaprogramming**
    - Three examples where metaprogramming is useful