CWI

Centrum Wiskunde & Informatica

The C++ STL

WISM454 Laboratory Class Scientific Computing, Jan-Willem Buurlage

February 21, 2019

C++

Standard libraries

C++ library

- `<iostream>`
- `<array>`
- `<vector>`
- `<string>`
- `<algorithm>`
- ...

C library (ports)

- `<cstdint>`
- `<cmath>`
- ...

<iostream>

```cpp
#include <iostream>

std::cout << "Hello, world!\n";

int a;
std::cin >> a;
```

Arrays in C++

```
int[5] xs = {1, 2, 3, 4, 5};
```

- Size has to be a compile time constant
- C-style dynamic (i.e. at runtime) arrays by heap allocation

```
int size;
std::cin >> size;
int* xs = new int[size];
// ... use xs
delete[] xs;
```

std::array

```
#include <array>

auto xs = std::array<int, 5>();

for (auto& x : xs) {
    x = 3;
}
```

- Standard statically (compile-time and constant) sized container

std::vector

```
#include <vector>

auto ys = std::vector<int>(5);

for (auto& y : xs) {
    y = 3;
}
```

- Dynamically (runtime and non-constant) sized container

Usage

```
void histogram(const std::vector<uint32_t>& xs,
    int bin_count = 10) {
    auto bins = std::vector<uint32_t>(bin_count);
    // ... fill bins
    for (auto& bin : bins) {
        // ... output bins
    }
}

struct lcrng_with_state {
    std::array<uint32_t, 10> state;
};
```

std::vector

```
auto xs = std::vector<int>(5);
auto y = xs[3]; // ~> 0
xs.empty(); // ~> false
xs.size(); // ~> 5
xs.capacity(); // ~> ?

xs.push_back(5);
xs.size(); // ~> 6

xs.clear();
xs.size(); // ~> 0
```

std::pair, std::tuple

```cpp
#include <utility>

std::pair<int, int> xs = std::make_pair(3, 3);
std::tuple<int, int, int> ys = std::make_tuple(3, 3, 5);
// xs.first, xs.second
// std::get<0>(ys), std::get<1>(ys)
```

<numeric>: ‘iota, accumulate‘

- Many algorithms in <numeric> operate on containers at once
- Does not have to be a vector, can also be a stack, queue or C-style array

```
// make a vector with 100 elements
auto xs = std::vector<int>(100);

// fill with 0 .. 99
std::iota(xs.begin(), xs.end(), 0);

// sum all the elements up
auto sum = std::accumulate(xs.begin(), xs.end(),
                           0, std::plus<int>());
// -> sum = 4950
```

<numeric>: partial_sum, adjacent_difference, inner_produc

```
// partial sum (in-place)
std::partial_sum(xs.begin(), xs.end(),
                 xs.begin());

// -> xs = [0, 1, 3, 6, ...]
std::adjacent_difference(xs.begin(), xs.end(),
                         xs.begin());
// -> xs = [0, 1, 2, 3, ...]

auto alpha = std::inner_product(xs.begin(), xs.end(),
                   xs.begin(), 0);
// -> alpha = 0 * 0 + 1 * 1 + 2 * 2 + 3 * 3 + ...
// -> alpha = 328350
```

<algorithm>: transform

- In <algorithm>, many more operations on container are defined, e.g.

```
std::transform(xs.begin(), xs.end(), xs.begin(),
    [](auto i) { return i * i; });
// -> xs = [0, 1, 4, 9, ...]

auto alpha = std::accumulate(xs.begin(), xs.end(), 0);
// -> alpha = 328350
```

<algorithm>: fill, generate

```
std::fill(xs.begin(), xs.end(), -1);
// -> xs = [-1, -1, -1, ..., -1]

int n = 1;
std::generate(xs.begin(), xs.end(),
              [&n](){ return n++; });
// -> xs = [1, 2, 3, ..., 100]
```

<algorithm>: any_of

```
// -> xs = [1, 2, 4, 6, ..., 100]
auto any = std::any_of(xs.begin(), xs.end(),
                       [](auto x) { x % 2 == 1 });

auto any_shift = std::any_of(xs.begin() + 1, xs.end(),
                             [](auto x) { x % 2 == 1 });
// -> any = true, any_shift = false
```

<algorithm>: all_of

```
// -> xs = [1, 2, 4, 6, ..., 100]
auto all = std::all_of(xs.begin(), xs.end(),
                       [](auto x) { x % 2 == 0 });

auto all_shift = std::all_of(xs.begin() + 1, xs.end(),
                             [](auto x) { x % 2 == 0 });
// -> all = false, all_shift = true
```

<algorithm>: none_of

```
// -> xs = [1, 2, 4, 6, ..., 100]
auto none = std::none_of(xs.begin(), xs.end(),
                         [](auto x) { x % 2 == 1 });

auto none_shift = std::none_of(xs.begin() + 1, xs.end(),
                               [](auto x) { x % 2 == 1 });
// -> none = false, none_shift = true
```

<algorithm>: ‘reverse, sort, partial$_{\text{sort}}$‘

```
// -> xs = [1, 2, 4, 6, ..., 100]
std::reverse(xs.begin(), xs.end());

// -> xs = [100, ..., 6, 4, 2, 1]
std::sort(xs.begin(), xs.end());

// -> xs = [1, 2, 4, 6, ..., 100]
std::iota(xs.begin(), xs.end(), xs.begin(), 0);
std::reverse(xs.begin(), xs.end());
// -> xs = [50, 49, 48, ... 0],

std::partial_sort(xs.begin(), xs.begin() + 3, xs.end());
// -> xs = [0, 1, 2, ?, ..., ?]
```

Other containers

- `std::stack`: FIFO container
- `std::queue`: LIFO container
- `std::dequeue`: fast insertion at begin and end
- `std::list`: linked list
- `std::priority_queue`: retrieve largest element in $\mathcal{O}(1)$, insert in $\mathcal{O}(log(n))$
- `std::map`: key-value dictionary, usually red-black tree
- `std::unordered_map`: key-value dictionary using hash map. $\mathcal{O}(1)$ insert, delete, find – but not sorted

Exercises

- I have compiled all the exercises in a file `exercises.pdf`. See the GitHub page.
- Implement LCRNG, Xorshift engines
- Implement distributions:
  - Uniform
  - Gaussian (with rejection)
  - Something with inversion
- Write function to randomly permute an array
- Statistically test your generators