An introduction to C++

day 3

Sandro Andreotti, Chris Bielow

Bioinformatics Solution Center, FU Berlin

slides: Hannes Hauswedell *AG Algorithmische Bioinformatik* et al.

Day	Content	State
Monday	Variables, Constness, I/O, STL (vector, string, ...)	✓
Tuesday	Control flow, functions (overloading), function templates	✓
Wednesday	Lambdas, Enums, struct/class, class template, compilation	✓
Thursday	STL: tuples, more data containers, algorithms	←
Friday	Holiday	
Monday	Questions, Recap, Debugging, **Test**	

Standard library overview

Tuples

Sequence containers

Associative containers

Algorithms

Standard library overview

Standard library overview

- The standard library, also called *standard template library*[1] (STL) is a set of functions, types and templates provided with every implementation of the C++ standard.
- 99% of the STL is implemented as regular C++, most of it as plain header files.

[1] There is some controversy to whether STL refers to all of the standard library or just a subset, but I use these terms interchangeably.

Standard library overview

- The standard library, also called *standard template library*[1] (STL) is a set of functions, types and templates provided with every implementation of the C++ standard.
- 99% of the STL is implemented as regular C++, most of it as plain header files.

[1] There is some controversy to whether STL refers to all of the standard library or just a subset, but I use these terms interchangeably.

- In contrast to *built-in* types, names from the standard library are only available when the correct header is included.
- Standard library headers have no file-extension (e.g. `<iostream>`, not `<iostream.h>`); headers from the C-standard that are also part of the C++ standard are prefixed with c (e.g. `<cmath>`, not `<math.h>`).

Standard library overview

- Names from the standard library are in `namespace std`, i.e. you need to prefix them with `std::` (e.g. `std::vector`).
- You can omit the `std::` prefix if you write `using namespace std;` at the top your `.cpp` files.
- If you use `using namespace std;` make sure that you never do it in header files, only in the `.cpp`!

Standard library overview – type deduction guides

Many templates in the standard library can be used without specifying their template arguments:

```
std::vector vec{3.3, 5.7};      // deduced to std::vector<double>

std::tuple tup{7, 3.3};         // deduced to std::tuple<int, double>
```

Similar rules to `auto` apply:

- It only works when you initialise the variable with something.
- You should not do it if you can easily write the type (i.e. the above examples are bad!)

P.S: does not work for member variables, just inside function bodies…

Standard library overview – type deduction guides

Example

```
template <typename TLeft, typename TRight>
auto sum_and_diff(TLeft const & left, TRight const & right)
{
    return std::tuple{left+right, left-right};
}
```

It is helpful when the type of the arguments is also deduced, e.g. depends on the instantiation of a template.

Standard library overview – O-notation

O	verbatim	very informal description
`O(1)`	constant	no overhead, very good
`O(log(n))`	log	worse than const, better than linear; often relates to trees or binary searches
`O(n)`	linear	cost proportional to n, e.g. size of container

- You don't need to understand *why* certain operations have the respective complexity, **but** you do need remember how they differ to be able to pick the right data structure for a task!
- I do not differentiate between *amortized* and regular complexity.
- There are other factors beside asymptotic complexity that influence performance ☝

Standard library overview

Tuples

Sequence containers

Associative containers

Algorithms

Tuples

Tuples are a convenient way to wrap multiple variables (even different types) together:

```
std::tuple<std::string, std::string, uint16_t> cosmonaut{"Sigmund", "Jaehn", 1938};
```

Tuples

Tuples are a convenient way to wrap multiple variables (even different types) together:

```
std::tuple<std::string, std::string, uint16_t> cosmonaut{"Sigmund", "Jaehn", 1938};
```

This provides better encapsulation and makes interfaces more readable:

```
void match(std::tuple<std::string, std::string, uint16_t> const & person0,
           std::tuple<std::string, std::string, uint16_t> const & person1);

// is more readable than:
void match(std::string const & person0_first_name,
           std::string const & person0_last_name,
           uint16_t const person0_year,
           std::string const & person1_first_name,
           std::string const & person1_last_name,
           uint16_t const person1_year);
```

Tuples

And is the only way to return multiple values of different types from a function:

```
auto make_tuple(size_t const i, std::string const & s)
{
    return std::tuple{i, s}; // return type is std::tuple<size_t, std::string>
}
```

Tuples

And is the only way to return multiple values of different types from a function:

```
auto make_tuple(size_t const i, std::string const & s)
{
    return std::tuple{i, s}; // return type is std::tuple<size_t, std::string>
}
```

There is also `std::pair`, a tuple of size 2, but it only exists for historical reasons – always use `std::tuple` nowadays!

Tuples

Why not use a small `struct` instead?

- There are advantages and disadvantages to both approaches.
- A disadvantage of tuples is that the members are *unnamed*, i.e. you have to know that the first string refers to the first name and the second one to the last name.
- An advantage of tuples is that all the usual operators are already defined on a per-element basis, i.e. `std::tuple<float, int>{1.1, 3} == std::tuple<float, int>{2.2, 3}` compares first the first element, then the second…
- Another advantage is that a tuple clearly indicates that "it just does storage" and nothing else.

Tuples – access

```
std::tuple<std::string, std::string> cosmonaut{"Sigmund", "Jaehn"};

// via get
std::cout << "First name: " << std::get<0>(cosmonaut) << ' '
          << "Last name: "  << std::get<1>(cosmonaut) << '\n';

// via structured bindings
auto & [ f, l ] = cosmonaut;    // std::string & f = std::get<0>(cosmonaut)…
std::cout << f << ' ' << l << '\n';
```

You can access tuple elements via

1. `std::get<NUMBER>()`
2. `std::get<TYPE>()` (only if the types are unique!)
3. "structured bindings", i.e. declaring a set of variables of `auto` type (or `auto &` or `auto const &`…) and assigning the tuple.

Tuples -- creation

```
size_t const i = 7;
std::string s{"foo"};

std::tuple tup0{i, s};
// == std::tuple<size_t, std::string>
```

- Creating tuples from existing variables copies the values and discards `const`.

Tuples -- creation

```
size_t const i = 7;
std::string s{"foo"};

std::tuple tup0{i, s};
// == std::tuple<size_t, std::string>
```

- Creating tuples from existing variables copies the values and discards `const`.

```
std::tuple<size_t const &,
           std::string &> tup1{i, s};
```

- To make a *tuple of references* you would need to specify that.

Tuples -- creation

```cpp
size_t const i = 7;
std::string s{"foo"};

std::tuple tup0{i, s};
// == std::tuple<size_t, std::string>
```

- Creating tuples from existing variables copies the values and discards `const`.

```cpp
std::tuple<size_t const &,
           std::string &> tup1{i, s};
```

- To make a *tuple of references* you would need to specify that.

```cpp
auto tup2 = std::tie(i, s);
// == std::tuple<size_t const &,
//               std::string &>
```

- But there is a convenience function for this: `std::tie()`

Tuples

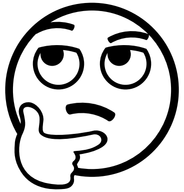
```
struct Person
{
    std::string first_n; std::string last_n;
    uint8_t age;
    bool operator<(Person const & r) const
    {
        if (first_n == r.first_n)
        {
            if (last_n == r.last_n)
                return age < r.age;
            else
                return last_n < r.last_n;
        } else
        {
            return first_n < r.first_n;
        }
    }
};
```

- Remember `struct Person`? What if we want to sort by all attributes?

Tuples

```
struct Person
{
    std::string first_n; std::string last_n;
    uint8_t age;
    bool operator<(Person const & r) const
    {
        if (first_n == r.first_n)
        {
            if (last_n == r.last_n)
                return age < r.age;
            else
                return last_n < r.last_n;
        } else
        {
            return first_n < r.first_n;
        }
    }
};
```

- Remember `struct Person`? What if we want to sort by all attributes?



Tuples

```
struct Person
{
    std::string first_n; std::string last_n;
    uint8_t age;
    bool operator<(Person const & r) const
    {
        return std::tie(first_n,
                        last_n,
                        age)
             < std::tie(r.first_n,
                        r.last_n,
                        r.age);
    }
};
```

- Since tuples have all the operators already defined, we can just use tuples to achieve this easily!
- BUT of course we need to make sure that comparing the values doesn't copy them, so we use `std::tie` to create a tuple of references.

Standard library overview

Tuples

Sequence containers

Associative containers

Algorithms

Sequence containers – Overview

Container	Informal summary
`std::array`	Fast access, but fixed number of elements
`std::vector`	Fast access, efficient insertion/deletion only at end
`std::basic_string`	Like `std::vector`, but optimised for character types
`std::deque`	Efficient insertion/deletion at beginning and end
`std::list`	Efficient insertion/deletion also in the middle, no `[]`
`std::forward_list`	Efficient insertion/deletion also in the middle, no `[]`

Customarily STL containers do not provide member functions for operations that would be slow, e.g. `std::vector` provides `.push_back()`, but not `.push_front()`, while `std::deque` and `std::list` provide both.

Sequence containers – Iterators

```
T my_cont{1, 2, 3, 4, 5};

for (auto const & elem : my_cont)                           // "range-based"
    std::cout << elem << ' ';

for (size_t i = 0; i < my_cont.size(); ++i)                 // via []
    std::cout << my_cont[i] << ' ';

for (auto it = my_cont.begin(); it != my_cont.end(); ++it)  // via iterator
    std::cout << *it << ' ';
```

- The second loop is much more flexible than the first, e.g. you could increment twice, to get every second element; or you could decrement the counter again, based on a condition.
- But `operator[]` it is not available for all containers, `std::forward_list` doesn't even have `.size()` 😱

Sequence containers – Iterators

```
for (auto it = my_cont.begin(); it != my_cont.end(); ++it)  // via iterator
    std::cout << *it << ' ';
```

- All STL containers return an iterator pointing to the first element when calling `.begin()`.
- This iterator can be incremented (`++`) to move to the next element, or *dereferenced* (`*`) to retrieve the actual element (both in `O(1)`!).
- It can also be compared against the special "end-iterator" retrieved by calling `.end()`; this points **behind the last element.**

Sequence containers – Iterators

```
for (auto it = my_cont.begin(); it != my_cont.end(); ++it)  // via iterator
    std::cout << *it << ' ';
```

- All STL containers return an iterator pointing to the first element when calling `.begin()`.
- This iterator can be incremented (`++`) to move to the next element, or *dereferenced* (`*`) to retrieve the actual element (both in `O(1)`!).
- It can also be compared against the special "end-iterator" retrieved by calling `.end()`; this points **behind the last element.**
- Most iterators offer more functions, like decrement (`--`) or `+=`.
- Iterators are *leight-weight objects*, they are cheap to copy!
- All STL containers can be looped over via iterators or "range-based", but not all via random access (`[]`).

Sequence containers

`std::array`

```
std::array<double, 2> df{3.1, 2.3};

std::cout << df[0]; // prints 3.1
df[1] = 32.0;       // assigns value
```

- size fixed at compile-time; specified via second template argument.
- provides *RandomAccessIterator*
- Use instead of built-in array in all serious projects, i.e. it has no drawbacks over built-in arrays.

Sequence containers

`std::array`

```
std::array<double, 2> df{3.1, 2.3};

std::cout << df[0]; // prints 3.1
df[1] = 32.0;       // assigns value
```

- size fixed at compile-time; specified via second template argument.
- provides *RandomAccessIterator*
- Use instead of built-in array in all serious projects, i.e. it has no drawbacks over built-in arrays.

`std::vector`

```
std::vector<double> df{3.1, 2.3};

std::cout << df[0]; // prints 3.1
df[1] = 32.0;       // assigns value
df.push_back(2.2);  // append value
df.resize(42);      // resize
```

- "Dynamic" array.
- append values in `O(1)`
- other inserts `O(n)`
- fast access and no size overhead
- If you are unsure, probably the right choice of container.

Sequence containers

`std::basic_string`

```
std::basic_string<char> str{"ABC"};
//== std::string

std::cout << str[0]; // prints 'A'
str[1] = 'B';        // assigns value
str[0] = 32;         // assigns value
```

- Like `std::vector<char>`, only supports character types.
- Slightly slower access.
- Optimisations for small strings.
- Convenience functions for input/output.

Sequence containers

`std::basic_string`

```
std::basic_string<char> str{"ABC"};
//== std::string

std::cout << str[0]; // prints 'A'
str[1] = 'B';        // assigns value
str[0] = 32;         // assigns value
```

- Like `std::vector<char>`, only supports character types.
- Slightly slower access.
- Optimisations for small strings.
- Convenience functions for input/output.

`std::deque`

```
std::deque<double> df{3.1, 2.3};

std::cout << df[0]; // prints 3.1
df[1] = 32.0;       // assigns value
df.push_back(2.2);  // append value
df.push_front(1.1); // prepend value
```

- Like vector, but:
- supports prepend in `O(1)`
- faster resizes
- high size overhead
- slightly slower access

Sequence containers

`std::list`

```cpp
std::list<double> df{3.1, 2.3};

std::cout << *df.begin();  // prints 3.1
df.insert(it, 2.2); // insert value
df.push_back(2.2);  // append value
```

- A doubly-linked list.
- Fast inserts/deletes anywhere.
- no random access! ~~`[]`~~
- 128bit size overhead per element
- provides *BidirectionalIterator*

Sequence containers

`std::list`

```
std::list<double> df{3.1, 2.3};

std::cout << *df.begin();  // prints 3.1
df.insert(it, 2.2); // insert value
df.push_back(2.2);  // append value
```

- A doubly-linked list.
- Fast inserts/deletes anywhere.
- no random access! []
- 128bit size overhead per element
- provides *BidirectionalIterator*

`std::forward_list`

```
std::forward_list<double> df{3.1, 2.3};

std::cout << *df.begin();  // prints 3.1
df.insert_after(it, 2.2);  // append
```

- A singly-linked list.
- Fast inserts/deletes anywhere.
- no random access, no `.size()`!
- 64bit size overhead per element
- provides *ForwardIterator*

Sequence containers

Container	++it	--it	`[]`	⇠	⇣	⇢	space overhead
`std::array`	✓	✓	✓				0
`std::vector`	✓	✓	✓			✓	3*8 byte per cont.
`std::basic_string`	✓	✓	✓			✓	small per cont.
`std::deque`	✓	✓	✓	✓		✓	large per cont.
`std::list`	✓	✓		✓	✓	✓	128bit per element
`std::forward_list`	✓			✓	✓	✓	64bit per element

- `++it` and `--it`: whether you can incr./decr. respective iterators
- ⇠, ⇣, ⇢: constant time insertions at begin, middle, end

Sequence containers

Quiz?

- What container do you choose if you know you will need to add an unknown amount of new elements periodically, but never delete any?

Sequence containers

Quiz?

- What container do you choose if you know you will need to add an unknown amount of new elements periodically, but never delete any?
- When would you prefer a `std::deque` over `std::vector`?

Sequence containers

Quiz?

- What container do you choose if you know you will need to add an unknown amount of new elements periodically, but never delete any?
- When would you prefer a `std::deque` over `std::vector`?
- When would you prefer `std::forward_list` over `std::list`?

Sequence containers

Quiz?

- What container do you choose if you know you will need to add an unknown amount of new elements periodically, but never delete any?
- When would you prefer a `std::deque` over `std::vector`?
- When would you prefer `std::forward_list` over `std::list`?
- Rule-of-thumb: It's seldom a good idea to make `std::deque`, `std::list` or `std::forward_list` of built-in types, because the overhead is just too high.

Standard library overview

Tuples

Sequence containers

Associative containers

Algorithms

Associative containers – Overview

Ordered	Unordered	Description
`std::set`	`std::unordered_set`	collection of unique keys
`std::multiset`	`std::unordered_multiset`	collection of keys
`std::map`	`std::unordered_map`	collection of key-value pairs; keys unique
`std::multimap`	`std::unordered_multimap`	collection of key-value pairs

- Ordered associative containers are sorted by key, operations are in `O(log(n))`.
- Unordered associative container are not sorted, operations are in `O(1)`.
- Maps are the most popular associative containers. ("dictionaries" in other languages)

Associative containers – `std::map`

```
//       key           value
std::map<std::string, size_t> name_salary{};

//              key         value
name_salary["Horst"]  = 7000;  // [] creates element if not found
name_salary["Angela"] = 5439;

name_salary["Horst"]  = 3999;  // overwrites previous value

for (auto const & [name, freq] : name_salary)
    std::cout << name << " is paid " << freq << " bitcoin!\n";
```

Associative containers – `std::map`

```
//       key            value
std::map<std::string, size_t> name_salary{};

//              key         value
name_salary["Horst"]  = 7000;  // [] creates element if not found
name_salary["Angela"] = 5439;

name_salary["Horst"]  = 3999;  // overwrites previous value

for (auto const & [name, freq] : name_salary)
    std::cout << name << " is paid " << freq << " bitcoin!\n";
```

- The element type of the map is a pair so we can use "structured bindings" to decompose the pair into individual variables.
- The first line printed will be "Angela...", because elements are sorted
- Each `[]` takes `O(log(n))`, ordered map usually implemented as tree.

Associative containers – `std::unordered_map`

```
//                   key           value
std::unordered_map<std::string, size_t> name_salary{};

//                key        value
name_salary["Horst"]  = 7000;  // [] creates element if not found
name_salary["Angela"] = 5439;

name_salary["Horst"]  = 3999;  // overwrites previous value

for (auto const & [name, freq] : name_salary)
    std::cout << name << " is paid " << freq << " bitcoin!\n";
```

Associative containers – `std::unordered_map`

```
//                    key           value
std::unordered_map<std::string, size_t> name_salary{};

//                 key         value
name_salary["Horst"]  = 7000;  // [] creates element if not found
name_salary["Angela"] = 5439;

name_salary["Horst"]  = 3999;  // overwrites previous value

for (auto const & [name, freq] : name_salary)
    std::cout << name << " is paid " << freq << " bitcoin!\n";
```

- The element type of the map is a pair so we can use "structured bindings" to decompose the pair into individual variables.
- It is undefined which element is printed first!
- Each `[]` takes `O(1)`; unordered map usually implemented as hash table.

Standard library overview

Tuples

Sequence containers

Associative containers

Algorithms

Algorithms

Many functions (and usually) function templates can be found in the `<algorithm>` header that work on sequences, e.g.

Ordered	Description
`std::sort`	Sort the range of items
`std::count`	Count the times an item apears in range
`std::find`	Find first item that matches
`std::reverse`	reverse the items in a range
`std::unique`	remove subsequent(!) duplicates

Algorithms

All of these take two iterators to denote the range of elements they work on, e.g.

```
template <typename TIterator>
std::sort(TIterator b, TIterator e);

template <typename TIterator, typename TLambda>
std::sort(TIterator b, TIterator e, TLambda const & l);
```

Algorithms

All of these take two iterators to denote the range of elements they work on, e.g.

```
template <typename TIterator>
std::sort(TIterator b, TIterator e);

template <typename TIterator, typename TLambda>
std::sort(TIterator b, TIterator e, TLambda const & l);
```

```
std::vector<size_t> vec{1, 5, 4, 9};
std::sort(vec.begin(), vec.end()); // vec == { 1, 4, 5, 9}

std::sort(vec.begin() + 1,
          vec.end(),
          [] (size_t const l, size_t const r) { return l > r; });
// vec == { 1, 9, 5, 4}
```

Tasks for the computer lab

Tasks for the computer lab I

- Adapt your program from yesterday's task2/3 to use a `std::tuple` instead of `struct Person`.
- Which things are now easier / better to understand? Which have become more difficult / obscure?

Tasks for the computer lab II

- A hospital asks you to implement a patient database and you have never heard of actual databases so you want to implement it in C++!
- You need to provide an interface that offers the following options:
  - Add new patient record [a]
  - Delete patient record by id [d]
  - Print record by id [p]
  - Quit [q]
- You expect the database to grow quickly so these operations should be as fast as possible.
- "Add…" produces a new entry unless the record (same ID) already exists - in this case update entry.
- A patient record consists of name, patient-id (0-1'000'000) and health-status ("healthy", "sick", "dead").

Tasks for the computer lab III

Sets are used less often than maps. One use-case are sparse matrices.

Take this example:

```
std::vector<bool> people_with_hat;      // whether person x has hat
people_with_hat.resize(1'000'000);      // lot's of people
people_with_hat[47] = true;
people_with_hat[120] = true;            // only few people have hat
```

1. How would you implement similar functionality with `std::set` or `std::unordered_set`?
2. How much space do you assume the above example uses? How much would it consume with the sets?
3. What if you want to take into account that people can have multiple hats?