



## Outline

- 1 The standard library
  - Sequences
  - Insertion
  - Container adapters
  - Sets and maps
  - tuples and `std::tie()`
- 2 Class idioms
  - Classes, copy and move
  - CRTP

## Standard containers

### Sequences (homogeneous)

- ▶ `vector<T>`
- ▶ `deque<T>`
- ▶ `list<T>`

### Associative containers (also *unordered*)

- ▶ `map<K,V>`, `multimap<K,V>`
- ▶ `set<T>`, `multiset<T>`

### Heterogeneous sequences (not “containers”)

- ▶ `tuple<T1, T2, ...>`
- ▶ `pair<T1,T2>`

## The classes `vector` and `deque`

### Operations in the class `vector`

```
v.clear(), v.size(), v.empty()
v.push_back(), v.pop_back()
v.front(), v.back(), v.at(i), v[i]
v.assign(), v.insert(), v.emplace()
v.resize(), v.reserve()
```

### Additional operations in `deque`

```
d.push_front(), d.pop_front()
```

## The classes `vector` and `deque` Constructors and the function `assign`

Constructors and `assign` have three overloads:

- ▶ **fill**: *n* elements with the same value
 

```
void assign (size_type n, const value_type& val);
```
- ▶ **initializer list**

```
void assign (initializer_list<value_type> il);
```
- ▶ **range**: copies the elements in the interval [*first*, *last*) (i.e., from *first* to *last*, excl. *last*)
 

```
template <class InputIterator>
void assign (InputIterator first, InputIterator last);
```

Use `()` for sizes, and `{}` for list of elements.

## The classes `vector` and `deque` The member function `assign`, example

```
vector<int> v;
int a[]={0,1,2,3,4,5,6,7,8,9};

v.assign(3,3);
print_seq(v);           length = 3: [3][3][3]

v.assign({3,3});
print_seq(v);           length = 2: [3][3]

v.assign(a, a+5);
print_seq(v);           length = 5: [0][1][2][3][4]

std::deque<int> d;
d.assign(v.begin(), v.end());
print_seq(d);           length = 5: [0][1][2][3][4]
```

*Examples of iterators*

## The classes `vector` and `deque` Member functions `push` and `pop`

`push` adds an element, increasing size  
`pop` removes an element, decreasing size  
`front`, `back` get a reference to the first (last) element

`*_back` operates at the end, available in both

```
void push_back (const value_type& val); //copy
void pop_back();
reference front();
reference back();
```

only in `deque`: `*_front`

```
void push_front (const value_type& val); //copy
void pop_front();
```

## `pop_X()`, `front()` and `back()`

NB! The return type of `pop_back()` is `void`.

Why separate functions?

- ▶ Don't pay for what you don't need.
  - ▶ A non-void `pop()` has to return by value (copy).
  - ▶ `front()/back()` can return a reference.
  - ▶ Let the caller decide if it wants a copy.

## Container and resource management

- ▶ Containers have value semantics
- ▶ Elements are copied into the container

## The classes `vector` and `deque` Insertion with `insert` and `emplace`

`insert`: copying

```
iterator insert (const_iterator pos, const value_type& val);
iterator insert (const_iterator pos, size_type n,
                const value_type& val);
template <class InputIterator>
iterator insert (const_iterator pos, InputIterator first,
                InputIterator last);
iterator insert (const_iterator pos,
                initializer_list<value_type> il);
```

`emplace`: construction "in-place"

```
template <class... Args>
iterator emplace (const_iterator position, Args&&... args);

template <class... Args>
void emplace_back (Args&&... args);
```

## The classes `vector` and `deque` Example with `insert` and `emplace`

```
struct Foo {
    int x;
    int y;
    Foo(int a=0,int b=0) :x{a},y{b} {cout<<"this <<"\n";}
    Foo(const Foo& f) :x{f.x},y{f.y} {cout<<"**Copying Foo\n";}
};
std::ostream& operator<<(std::ostream& os, const Foo& f)
{
    return os << "Foo("<< f.x << ", "<< f.y<<")";
}
vector<Foo> v;
v.reserve(4);
v.insert(v.begin(), Foo(17,42)); Foo(17,42)
                               **Copying Foo
print_seq(v); length = 1: [Foo(17,42)]
v.insert(v.end(), Foo(7,2));   Foo(7,2)
                               **Copying Foo
print_seq(v); length = 2: [Foo(17,42)][Foo(7,2)]
v.emplace_back();             Foo(0,0)
print_seq(v); length = 3: [Foo(17,42)][Foo(7,2)][Foo(0,0)]
v.emplace_back(10);          Foo(10,0)
print_seq(v); length = 4: [Foo(17,42)][Foo(7,2)][Foo(0,0)][Foo(10,0)]
```

## Container and resource management

- ▶ Containers have value semantics
- ▶ Elements are copied into the container
- ▶ When an element is removed, it is destroyed
- ▶ The destructor of a container destroys all elements
- ▶ Usually a bad idea to store owning raw pointers in a container
  - ▶ Requires explicit destruction of the elements
  - ▶ Use smart pointers

## Queues and stacks

- ▶ *adapter classes*, providing a limited interface to one of the standard containers: `stack`, `queue`, `priority_queue`
  - ▶ fewer operations
  - ▶ do not have iterators

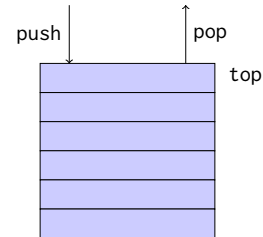
Has a default underlying container. E.g., for `stack`:

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

but `stack` can be instantiated with any class that has `push_back()`, `pop_back()` and `back()`.

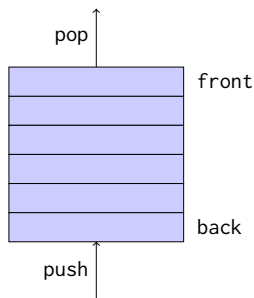
## Queues and stacks

- ▶ `Stack`: LIFO queue (Last In First Out)
- ▶ Operations: `push`, `pop`, `top`, `size` and `empty`



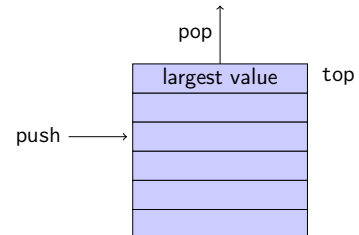
## Queues and stacks

- ▶ `Queue`: FIFO-queue (First In First Out)
- ▶ Operations: `push`, `pop`, `front`, `back`, `size` and `empty`



## Queues and stacks

- ▶ `Priority queue`: sorted queue. The element highest priority is first in the queue.
- ▶ Operations: `push`, `pop`, `top`, `size` and `empty`



*Compares elements with `std::less<T>` by default.  
A custom comparator can be used. E.g., using `std::greater<T>`  
would cause the smallest element to be first.*

## Sets and maps

### Associative containers

<code>map&lt;Key, Value&gt;</code>	Unique keys
<code>multimap&lt;Key, Value&gt;</code>	Can contain duplicate keys
<code>set&lt;Key&gt;</code>	Unique keys
<code>multiset&lt;Key&gt;</code>	Can contain duplicate keys

*set is in principle a map without values.*

- ▶ By default orders elements with `operator<`
- ▶ A custom comparator can be provided, e.g.

```
template<class Key, class Compare = std::less<Key>>
class set{
    explicit set( const Compare& comp = Compare());
    ...
};
```

## Sets and maps

`<set>`: `std::set`

```
void test_set()
{
    std::set<int> ints{1,3,7};

    ints.insert(5);
    for(auto x : ints) {
        cout << x << " ";
    }
    cout << endl;
    auto has_one = ints.find(1);

    if(has_one != ints.end()){
        cout << "one is in the set\n";
    } else {
        cout << "one is not in the set\n";
    }
}

1 3 5 7
one is in the set
```

Or

```
if(ints.count(1))
```

## Sets and maps

<map>: std::map

```
map<string, int> msi;
msi.insert(make_pair("Kalle", 1));
msi.emplace("Lisa", 2);
msi["Kim"] = 5;
for(const auto& a: msi) {
    cout << a.first << " : " << a.second << endl;
}
cout << "Lisa --> " << msi.at("Lisa") << endl;
cout << "Hasse --> " << msi["Hasse"] << endl;
auto nisse = msi.find("Nisse");
if(nisse != msi.end()) {
    cout << "Nisse : " << nisse->second << endl;
} else {
    cout << "Nisse not found\n";
}
Kalle : 1
Kim : 5
Lisa : 2
Lisa --> 2
Hasse --> 0
Nisse not found
```

## Sets and maps

*A std::set is in principle a std::map without values*

### Operations on std::map

insert, emplace, [], at, find, count,  
erase, clear, size, empty,  
lower\_bound, upper\_bound, equal\_range

### Operations on std::set

insert, emplace, find, count,  
erase, clear, size, empty,  
lower\_bound, upper\_bound, equal\_range

*Use the member functions, not algorithms like std::find()  
(It works, but is less efficient – linear time complexity instead of logarithmic.)*

## Sets and maps

The return value of insert

### insert() returns a pair

```
std::pair<iterator, bool> insert( const value_type& value );
```

The insert member function returns two things:

- ▶ An iterator to the inserted value
  - ▶ or to the element that prevented insertion
- ▶ A **bool**: **true** if the element was inserted

insert() in multiset and multimap just returns an iterator.

### Using std::tie to unpack a pair (or tuple)

```
bool inserted;
std::tie(std::ignore, inserted) = set.insert(value);
```

## pairs and std::tie

Example: explicit element access

### Getting the elements of a pair

```
void example1()
{
    auto t = std::make_pair(10, "Hello");

    int i;
    string s;

    i = t.first;
    s = t.second;

    cout << "i: " << i << ", s: " << s << endl;
}
```

## pairs and std::tie

Example: using std::tie

### Getting the elements of a pair

```
void example1b()
{
    auto t = std::make_pair(10, "Hello");

    int i;
    string s;

    std::tie(i, s) = t;

    cout << "i: " << i << ", s: " << s << endl;
}
```

## tuples and std::tie

Example: using std::get(std::tuple)

### Getting the elements of a tuple

```
void example2()
{
    auto t = std::make_tuple(10, "Hello", 4.2);

    int i;
    string s;
    double d;

    i = std::get<0>(t);
    s = std::get<1>(t);
    d = std::get<2>(t);

    cout << "i: " << i << ", s: " << s << ", d: " << d << endl;
}
```

**NB!** std::get(std::tuple) takes the index as a *template parameter*.

## tuples and std::tie

Example: using std::tie

### Getting the elements of a tuple

```
void example2b()
{
    auto t = std::make_tuple(10, "Hello", 4.2);

    int i;
    string s;
    double d;

    std::tie(i,s,d) = t;

    cout << "i: " << i << ", s: " << s << ", d: " << d << endl;
}
```

## std::tie

Example: ignoring values with std::ignore

### Getting the elements of a tuple

```
void example2c()
{
    auto t = std::make_tuple(10, "Hello", 4.2);

    int i;
    double d;

    std::tie(i, std::ignore, d) = t;

    cout << "i: " << i << ", d: " << d << endl;
}
```

std::ignore is *an object of unspecified type such that assigning any value to it has no effect.*

## std::tie

Example: implementation sketch

### tie for a pair<int, string>

```
std::pair<int&, string&> mytie(int& x, string& y)
{
    return std::pair<int&, string&>(x,y);
}
```

- ▶ returns a *temporary* pair of *lvalue references*
- ▶ the assignment operator of pair assigns each member
- ▶ the references are *aliases for the variables* passed as arguments
- ▶ assigning to the references is the same as assigning to the variables

```
int i;
string s;

mytie(i,s) = t;
```

## std::tie

Comments

### possible implementation

```
template <typename... Args>
std::tuple<Args&...> tie(Args&... args)
{
    return std::tuple<Args&...>(args...);
}
```

- ▶ std::tie can be used on both std::pair and std::tuple, as a tuple has an implicit conversion from pair.
- ▶ The variables used with std::tie must have been declared.
- ▶ C++17 introduces *structured bindings* that lets you write code like `const auto& [i,s,d] = some_tuple;`
  - ▶ No need to declare variables before
  - ▶ Cannot use std::ignore: compiler warning if you don't use all variables.

## Resource management

copy assignment: operator=

### Declaration (in the class definition of Vector)

```
const Vector& operator=(const Vector& v);
```

### Definitionen (outside the class definition)

```
Vector& Vector::operator=(const Vector& v)
{
    if (this != &v) {
        auto tmp = new int[sz];
        for (int i=0; i<sz; i++)
            tmp[i] = v.elem[i];
        sz = v.sz;
        delete[] elem;
        elem = tmp;
    }
    return *this;
}
```

① check "self assignment"  
② Allocate new resources  
③ Copy values  
④ Free old resources

For error handling, better to allocate and copy first and only delete if copying succeeded.

## We can do better

- ▶ Code complexity
  - ▶ Both copy and move assignment operators
  - ▶ Code duplication
  - ▶ Brittle, manual code
    - ▶ self-assignment check
    - ▶ copying
    - ▶ memory management
- ▶ exception safety: what if copy or move throws
  - ▶ Weak exception guarantee: don't leak memory
  - ▶ Strong exception guarantee: retain object state
  - ▶ Important for move, as it destroys the source

*alternative: The copy-and-swap idiom.*

## Swapping – std::swap

The standard library defines a function (template) for swapping the values of two variables:

### Example implementation (C++11)

```
template <typename T>
void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}

template <typename T>
void swap(T& a, T& b)
{
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
```

The generic version does unnecessary copying, for Vector we can simply swap the members.

### Overload for Vector (needs to be friend)

```
void swap(Vector& a, Vector& b) noexcept
{
    using std::swap;
    swap(a.sz, b.sz);
    swap(a.elem, b.elem);
}
```

## Copy assignment The copy and swap idiom

### Copy-assignment

```
Vector& Vector::operator=(Vector v) {
    swap(*this, v);
    return *this;
}
```

- ▶ Call by value
  - ▶ let the compiler do the copy
  - ▶ works for both copy assign and move assign
    - ▶ called with *lvalue* ⇒ copy construction
    - ▶ called with *rvalue* ⇒ move construction
- ▶ No code duplication
- ▶ Less error-prone
- ▶ Needs an overloaded swap()
  - ▶ In the same namespace, to be found through ADL
- ▶ Slightly less efficient (one additional assignment)

## Swapping – std::swap

- ▶ The swap function can be both declared as a friend and *defined inside the class definition*.
- ▶ Still a free function
- ▶ In the same namespace as the class
  - ▶ Good for ADL

### Overload for Vector ("inline" friend)

```
class Vector {
    ...

    friend void swap(Vector& a, Vector& b) noexcept
    {
        using std::swap;
        swap(a.sz, b.sz);
        swap(a.elem, b.elem);
    }
};
```

*common idiom:*

- ▶ use `using` to make `std::swap` visible
- ▶ call `swap` unqualified to allow ADL to find an overloaded `swap` for the argument type

## The Curiously Recurring Template Pattern Static polymorphism

- ▶ Polymorphism without the run-time overhead
  - ▶ Common functionality in base class
    - ▶ E.g., compute value
  - ▶ Specific functionality in derived classes
    - ▶ E.g., output to different devices (console, file, socket)
- ▶ Reuse of generic functionality in unrelated classes
  - ▶ Related to *Mixin classes*
  - ▶ E.g., counting allocations and instances

## The Curiously Recurring Template Pattern Dynamic polymorphism

### Normal abstract class

```
class Base{
public:
    virtual void method() =0;
};

class Derived1 :public Base{
public:
    void method() override{
        cout << "Derived1::method\n";
    }
};
```

## The Curiously Recurring Template Pattern Static polymorphism

### The CRTP structure

```
template <typename T>
class Base {
public:
    void method() {
        static_cast<T*>(this)->method();
    }
};

class Derived : public Base<Derived> {
public:
    void method() {
        std::cout << "Derived method" << std::endl;
    }
};
```

## The Curiously Recurring Template Pattern Example: Animal sounds

```
class Animal {
public:
    Animal(const std::string& name) :name(name) {}
    void speak() const {cout << name << " says " << get_sound() << "\n";}
    virtual std::string get_sound() const = 0;
    virtual ~Animal() =default;
private:
    std::string name;
};

class Dog : public Animal {
public:
    using Animal::Animal;
    virtual std::string get_sound() const override {return {"Woof"}};
};

class Cat : public Animal {
public:
    using Animal::Animal;
    virtual std::string get_sound() const override {return {"Meow"}};
};
```

## The Curiously Recurring Template Pattern Example: Animal sounds

If we don't need run-time polymorphism:

```
Dog d{"Fido"};
Cat c{"Caesar"};

d.speak();      Fido says Woof!
c.speak();      Caesar says Meow!
```

### Base class template

```
template <typename Derived>
class Animal {
public:
    Animal(const std::string& name) :name(name) {}
    void speak() const {
        cout << name << " says "
            << static_cast<const Derived*>(this)->get_sound()
            << "\n";
    }
private:
    std::string name;
};
```

## The Curiously Recurring Template Pattern Example: Animal sounds

### Concrete derived classes

```
class Dog : public Animal<Dog> {
public:
    using Animal::Animal;
    std::string get_sound() const {
        return {"Woof"};
    }
};

class Cat : public Animal<Cat> {
public:
    using Animal::Animal;
    std::string get_sound() const {
        return {"Meow"};
    }
};
```

*NB! No override*

## The Curiously Recurring Template Pattern Example:

### Base class template

```
template <typename Derived>
class Computer{
public:
    void print_answer(){
        auto ans = incredibly_complex_computation();
        static_cast<Derived*>(this)->do_print_answer(ans);
    }
private:
    int incredibly_complex_computation() {return 42;}
};
```

Behaves like it had a pure virtual function

```
virtual void do_print_answer(int) =0;
```

## The Curiously Recurring Template Pattern Example:

### Concrete classes

```
class Local_Computer :public Computer<Local_Computer>{
public:
    void do_print_answer(int ans) {
        cout << "Answer:" << ans << endl;
    }
};

class Networked_Computer :public Computer<Networked_Computer>{
public:
    Networked_Computer(ServerConnection c) :conn{c} {}
    void do_print_answer(int ans) {
        conn.upload(ans);
    }
private:
    ServerConnection conn;
};

Local_Computer l{};
l.print_answer();    Answer: 42
```

## The Curiously Recurring Template Pattern Static polymorphism

- ▶ Polymorphism without the run-time overhead
  - ▶ Common functionality in base class
    - ▶ E.g., compute value
  - ▶ Specific functionality in derived classes
    - ▶ E.g., output to different devices (console, file, socket)
- ▶ Reusing generic functionality in unrelated classes
  - ▶ E.g., counting allocations and instances

## The Curiously Recurring Template Pattern

Example: counting instances

### Base class template

```
template <typename Derived>
class Counted{
public:
    static int get_alive() {return alive;}
    static int get_created() {return created;}
protected:
    Counted() {++created; ++alive;}
    Counted(const Counted&) {++created; ++alive;}
    ~Counted() {--alive;}
private:
    static int created;
    static int alive;
};

template <typename Derived>
int Counted<Derived>::created {0};

template <typename Derived>
int Counted<Derived>::alive {0};
```

► The variables are **static**: one variable per class (not per object).

► This is a *class template*: a new Counted<T> class will be instantiated for each subclass

► Each subclass will have its own counters

## The Curiously Recurring Template Pattern

Example: counting instances

### Concrete subclass and helper function

```
class Foo :public Counted<Foo>
{
public:
    Foo(int i) :x(i) {}
private:
    int x;
};

template <typename T>
void print_counts()
{
    cout << typeid(T).name() << " alive: " << T::get_alive()
    << ", created: " << T::get_created() << endl;
}
```

## Suggested reading

References to sections in Lippman

[Sequential containers](#) 9.1 – 9.3

[Container Adapters](#) 9.6

[Associative containers](#) chapter 11

[Tuples](#) 17.1

[Swap](#) 13.3