
Sven Gestegård Robertz

Computer Science, LTH

2020
Outline

1. Classes, copy and move
   - Move semantics (C++11)
   - copy and swap

2. ADL

3. The standard library
   - Sequences
   - Insertion
   - Container adapters
   - Sets and maps
   - tuples and std::tie()
Resource management

**copy assignment:** \texttt{operator=}

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

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

**Definitionen (outside the class definition)**

```cpp
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 \texttt{delete} if copying succeeded.
Move semantics
Making value semantics more efficient

- Copying is unnecessary if the source will not be used again e.g. if
  - it is a temporary value, e.g.
    - (implicitly) converted function arguments
    - function return values
    - a + b
  - the programmer explicitly specifies it:
    `std::move()` is a type cast to `rvalue-reference` (`T&&`)
    (include `<utility>`)   

- Better to “steal” the contents
- Makes resource handles even more efficient
- Some objects may/can not be copied
  - e.g., `std::unique_ptr`
  - use `std::move`
Lvalues and rvalues

▶ Applies to *expressions*
▶ An *lvalue* is a named object (persists beyond an expression)
  ▶ x
  ▶ *p
  ▶ arr[4] (unless *operator[]* const returns a copy)
▶ An *rvalue* is a temporary value
  ▶ 123
  ▶ a+b (i.e., return values from functions)
▶ you can take the address of it ⇒ *lvalue*
▶ if it has a name, it is an lvalue
▶ Better rule than the old “Can it be the left hand side of an assignment?” (because of *const*)
An *lvalue reference* can only refer to a named object

An *const lvalue reference* can also refer to a temporary
  - Extends the lifetime of the temporary to the lifetime of the reference

An *rvalue reference* can only refer to a temporary

Syntax:
- *(lvalue) reference*: `T&`
- *rvalue reference*: `T&& (C++11)`
If a class owns a resource, it should implement

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. Move constructor
5. Move assignment operator
Move operations

- Copy values
- "Steal" pointers
- Make right hand operand an "empty hulk"
  - The only thing allowed is to run the destructor
  - Often: set pointers to nullptr
Move constructor
Example: Vector

Move constructor (C++-11)

Vector::Vector(Vector&& v) : elem{v.elem}, sz{v.sz}
{
    v.elem = nullptr;
    v.sz = 0; // v has no elements
}

Copy constructor

Vector::Vector(const Vector& v) : elem{new double[v.sz]}, sz{v.sz}
{
    for(int i=0; i < sz; ++i) {
        elem[i] = v[i];
    }
}
Copy control: (Move semantics – C++11)
Example: Vector

Move assignment

```cpp
Vector& Vector::operator=(Vector&& v) {
    if (this != &v) {
        delete[] elem; // delete current array
        elem = v.elem; // "move" the array from v
        v.elem = nullptr; // mark v as an "empty hulk"
        sz = v.sz;
        v.sz = 0;
    }
    return *this;
}
```
Copy and move special member functions
Can we 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

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

(C++11)

```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

```cpp
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)

```cpp
class Vector {
...

friend void swap(Vector& a, Vector& b) noexcept {

    using std::swap; // common idiom:
    swap(a.sz, b.sz);
    swap(a.elem, b.elem);
}
}
Name lookup is done in *enclosing scopes*, but...

```cpp	namespace test{
    struct Foo{
        Foo(int v) : x{v} {}
        int x;
    };
    std::ostream& operator<<(std::ostream& o, const Foo& f) {
        return o << "Foo(\n            \n        )\n    } // namespace test

    int main()
    {
        test::Foo f(17);
        cout << f << endl;
    }
```

- The function `operator<<(ostream&, const Foo&)` is not visible in `main()`.
- Through ADL it is found in the namespace of its argument (`test`).
namespace test{
struct Foo;
std::ostream& operator<<(std::ostream& o, const Foo& f);

void print(const Foo& f)
{
    cout << f << endl;
}
void print(int i)
{
    cout << i << endl;
}
} // namespace test

int main()
{
    test::Foo f(17);
    print(f);
    print(17);
    test::print(17);
}

▶ The function test::print(int) is not found.
▶ unless using test::print.
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.emplace_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()`, `d.emplace_front()`
Constructors and assign have three overloads:

- **fill**: n elements with the same value
  
  ```cpp
  void assign (size_type n, const value_type& val);
  ```

- **initializer list**
  
  ```cpp
  void assign (initializer_list<value_type> il);
  ```

- **range**: copies the elements in the interval \([first, last)\) (i.e., from first to last, excl. last)
  
  ```cpp
  template <class InputIterator>
  void assign (InputIterator first, InputIterator last);
  ```

*Use () for constructor arguments, and {} for list of elements.*
The classes `vector` and `deque` Constructor and `assign`, example

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

vector<int> v(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(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

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

**only in deque: *_front**

```cpp
void push_front (const value_type& val);
void pop_front();
```
pop_X(), front() and back()

NB! The return type of pop_back() is \texttt{void}.

\begin{verbatim}
auto val = v.back();
v.pop_back();
\end{verbatim}

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.
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

Avoid making unnecessary copies
  - Use `emplace_back` instead of `push_back` of a temporary
  - use `reserve` before doing a known number of `push_back`
  - reuse capacity when possible
  - call by `const &` as default
The classes `vector` and `deque` Insertion with `insert` and `emplace`

**insert: copying**

```cpp
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"**

```cpp
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)]
```
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:

```cpp
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

![Stack diagram]

The standard library: Container adapters

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

Compress 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

- `map<Key,Value>`: Unique keys
- `multimap<Key,Value>`: Can contain duplicate keys
- `set<Key>`: Unique keys
- `multiset<Key>`: 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());
    ...

};
```
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`

```cpp
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          NB! operator[] default constructs values for new keys
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);
```
Example: explicit element access

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

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

    cout << "i: " << i << ", s: " << s << endl;
}
```
void example1b()
{
    auto t = std::make_pair(10, "Hello");

    int i;
    string s;

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

    cout << "i: " << i << " , s: " << s << endl;
}
pairs and std::tie
Example: structured binding (C++17)

```cpp
void example1c()
{
    auto t = std::make_pair(10, "Hello");
    const auto& [i, s] = t;
    cout << "i: " << i << ", s: " << s << endl;
}

NB! cannot use std::ignore: warnings for unused variables.
```
tuples and std::tie
Example: using std::get(std::tuple)

Getting the elements of a tuple

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

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

```cpp
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

```cpp
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 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: warning for unused variables.
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
Moving objects 13.6