EDAF50 – C++ Programming

9. Move semantics. Standard library containers.

Sven Gestegård Robertz

Computer Science, LTH

2021

Outline

1. Classes, copy and move
   - Move semantics (C++11)
   - copy and swap
2. The standard library
   - Sequences
   - Insertion
   - Container adapters

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 succeded.*

Lvalues and rvalues
Object lifetimes

- Applies to *expressions*
- An *lvalue* is an expression identifying an object (that persists beyond an expression)
- Examples:
  - `x`
  - `*p`
  - `arr[4]`
- An *rvalue* is a temporary value
- Examples:
  - `123`
  - `a+b`
- you can take the address of it ⇒ *lvalue*
- it has a name ⇒ *lvalue*
- Better rule than the old "Can it be the left hand side of an assignment?" (because of **const**)

Lvalues and rvalues
references

- An *lvalue reference* can only refer to a modifiable 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)

Move semantics
Making value semantics 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>`)
- Some objects may/can not be copied
  - e.g., `std::unique_ptr`
  - use `std::move`
- Better to “steal” the contents
- Makes *resource handles* even more efficient

Move semantics
Making value semantics efficient

Move operations:

```
class Foo {
public:
    ...
    Foo(Foo&&);                // move constructor
    Foo& operator=(Foo&&); // move assignment
};
```

- look like copying, but
- "steals" owned resources instead of copying

"Rule of ~~three~~ five"
Canonical construction idiom, in C++11

If a class owns a resource, it should implement (or =**`default`** or =**`delete`**)

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

Move constructor
implicitly generated

An automatically generated move constructor is provided if

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there is no user-declared destructor.

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 control: (Move semantics – C++11)
Example: Vector

Move assignment

```
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;
}
```

Resource management

copy assignment: **operator=**

Declaration (in the class definition of Vector)

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

Definition (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 succeded.*

Copy/move assignment
We can (often) do better

- Code complexity
  - Both copy and move assignment operators
  - Code duplication
  - Brittle, manual code
    - self-assignment check
    - copying
    - memory management

alternative: The copy-and-swap idiom.

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* $\Rightarrow$ copy construction
    - called with *rvalue* $\Rightarrow$ move construction
- No code duplication
- Less error-prone
- May need an overloaded `swap()`
- Slightly less efficient (one additional assignment)

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);
}
```

common idiom:

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

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 {
  // declarations of members ...

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

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

The standard library has two main sequence data types

std::vector your default sequence type

- Contigous in memory
- Grows at the back

std::deque Double ended queue

- Piecewise contigous in memory
- Grows at front and back

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()
```

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.

```
vector<int> v{3,3};
print_seq(v);          length = 2: [3][3]

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


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

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**.

```
auto val = v.back();
v.pop_back();
```

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.

Growing a vector
Size and capacity

A container has a *size* and a *capacity*.

On a `push_back`, if `size == capacity` the vector grows

- New storage is allocated
- The elements are copied

If you know how many `push_back` calls you will make,

- first use `reserve()` to (at least) the expected final size.
- then do a series of `push_back`

Container and resource management

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

The classes vector and deque
Insertion with insert/push_back and emplace(back)

insert: copying (or moving)

```
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);
```

and push_back.

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
  - Prefer 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

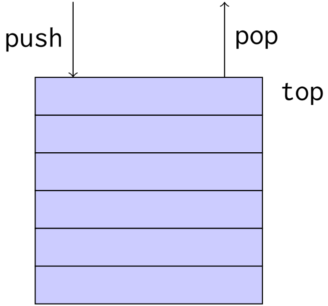
Has a default underlying container. E.g., for `stack`:

```
 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()`.

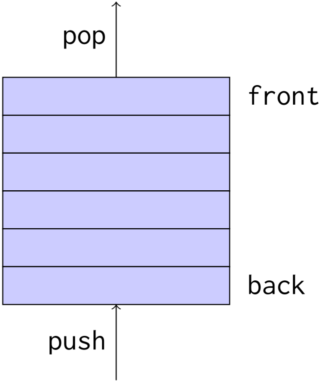
Queues and stacks

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



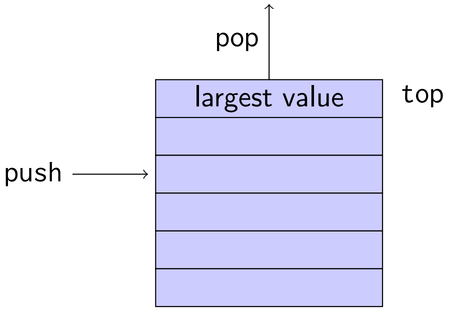
Queues and stacks

- 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

`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<`**

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

- A custom comparator can be provided

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       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.

Getting the result of an insert

```
auto result = set.insert(value);
bool inserted = result.second;
```

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

Moving objects 13.6