# EDAF30 – Programming in C++

## *8. More about resource management, classes and the standard library.*

Sven Gestegård Robertz

*Computer Science, LTH*

2019

# Outline

# Custom comparator for std::set

A `std::set<T>` is sorted

- using **operator**< by default
- or with a custom binary predicate

The predicate should be of the form `bool (const T&, const T&)`, and can be given as

- a function pointer
- a functor
  - an object of a class with `operator()(const T&, const T&)`
  - the value of a lambda expression

# std::set
## details

### The type parameters

```
template< class Key,
          class Compare   = std::less<Key>,
          class Allocator = std::allocator<Key>
> class set;
```

### And some constructors:

```
set();

explicit set( const Compare& comp,
              const Allocator& alloc = Allocator() );

set( std::initializer_list<value_type> init,
     const Compare& comp = Compare(),
     const Allocator& alloc = Allocator() )
```

# Custom comparator for std::set

```
struct Longest_Str{
  bool operator()(const string& l, const string &r) const {
    return l.size() > r.size();}
};

bool short_str(const string& l, const string& r)
{
  return l.size() < r.size();
}
```

```
#include<set>

std::set<string, Longest_Str> s{"a", "bb"};
std::set<string, Longest_Str> s2;

using StrComp = bool (*)(const string&, const string&);
std::set<string, StrComp> t({"x", "yy"}, short_str);
std::set<string, StrComp> t2(short_str);
```

# Lvalues and rvalues

- Applies to *expressions*
- An *lvalue* is a named object (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*
- 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 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)

# 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>`)
- 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`

*rule-of-thumb: "if it has a name, it is an lvalue"*

# "Rule of ~~three~~ five"
## Canonical construction idiom, in 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 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* ⇒ copy construction
    - called with *rvalue* ⇒ move construction
- No code duplication
- Less error-prone
- Needs 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);
  }
};
```

# Argument Dependent Lookup (ADL)

## Name lookup is done in *enclosing scopes*, but...

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

# Argument Dependent Lookup (ADL)

```
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 functions `test::operator<<()` and `test::print(const Foo&)` are found through ADL.
- The function `test::print(int)` is not found.
- unless `using test::print`.

# Swapping – std::swap_ranges (from <algorithm>)

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt2 swap_ranges( ForwardIt1 first1, ForwardIt1 last1,
                        ForwardIt2 first2 );
```

Returns an iterator one past the last element swapped in the range beginning with `first2`

# 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 (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);
```

### 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

# 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

### 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 = t.first;
    string s = t.second;

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

# 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.

# Suggested reading

References to sections in Lippman

| | |
|---|---|
| swap | 13.3 |
| Copying and moving objects | 13.4, 13.6 |
| (allocators) | 12.2.2 |
| (Classes, dynamic memory allocation) | 13.5 |
| Container Adapters | 9.6 |
| Pairs | 11.2.3 |
| Tuples | 17.1 |