EDAF30 – Programming in C++

10. More about resource management, classes and the standard library.

Sven Gestegård Robertz

Computer Science, LTH

2023

Outline

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

Function calls and results
Returning objects by value

- A function cannot return references to local variables
  - the object is destroyed at **`return`** – *dangling reference*
- How (in)efficient is it to return objects by value (a copy)?

return value optimization (RVO)

The compiler may optimize away copies of objects on **`return`** from functions

- *return by value* often efficient, also for larger objects
- RVO allowed *even if the copy-constructor or destructor has side effects*
- avoid such side effects to make code portable

Rules of thumb for function parameters

- Return by value more often
- Do not over-use call-by-value

"reasonable defaults"

	cheap to copy	moderately cheap to copy	expensive to copy
In	`f(X)`	`f(const X&)`	
In/Out	`f(X&)`		
Out	`X f()`		`f(X&)`

For results, if the cost of copying is

- small, or moderate ($< 1k$, contiguous): return by value (modern copilers do RVO: return value optimization)
- large : call by reference as *out parameter*
  - or maybe allocate with **new** and return pointer

Call by reference or by value?
Rules of thumb

For passing an object to a function when

- you may want *to change the value* of the object
  - reference: `void f(T&);` or
  - pointer: `void f(T*);`
- you *will not* change it, it is *large* (or impossible to copy)
  - constant reference: `void f(const T&);`
- otherwise, *call by value*
  - `void f(T);`

reference or pointer?

- required parameter: pass reference
- optional parameter: pass pointer (can be `nullptr`)

```
void f(widget& w)
{
    use(w); //required parameter
}

void g(widget* w)
{
    if(w) use(w); //optional parameter
}
```

Call by reference or by value?

- How big is "large"?
  - more than a few *words*
- When to use out parameters?
  - prefer code that is obvious

    Example: two functions:

    ```
    void incr1(int& x)
    {
        ++x;
    }

    int incr2(int x)
    {
      return x + 1;
    }
    ```

    Use:

    ```
    int v = 0;
    ...
    incr1(v);

    ...
    v = incr2(v);
    ```

    Here it is much clearer that v = incr2(v) changes v
  - For multiple output values, consider returning a **struct**, a std::pair or a std::tuple

Rules of thumb for function parameters

- Return by value more often
- Do not over-use call-by-value

"reasonable defaults"

	cheap to copy	moderately cheap to copy	expensive to copy
In	`f(X)`	`f(const X&)`	
In/Out	`f(X&)`		
Out	`X f()`		`f(X&)`

Function *inlining*:

- Replace a function call with the code in the function body
  - **`inline`** is a hint to the compiler
- Only suitable for (very) small functions
- Implicit if the function definition is in the class definition
- If the function is defined outside the class definition, use the keyword **`inline`**

Class definitions
Member functions and **inline**, example

Inline is implicit
in the class definition:

```
class Foo {
public:
    int scale(int x) {
        return value * x;
    }
    // ...
private:
    int value;
};
```

Inline definition outside the
class definition:

```
inline int Foo::scale(int x)
{
    return value * x;
}
```

Usage: With the code

```
Foo f;
//...
auto v = f.scale(17);
```

inlining means compiling to
code that behaves like

```
Foo f;
//...
auto v = f.value * 17;
```

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;
}
```

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);
```

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 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;
}
```

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 and move 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
- 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 may do unnecessary copying (especially pre move semantics, or if members cannot be moved), 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);
  }
};
```

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);
  }
};
```

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);
  }
};
```

Demo

Standard container iterators and swap

23.2.1 General container requirements includes

> *The expression a.swap(b), for containers a and b of a standard container type other than array, shall exchange the values of a and b without invoking any move, copy, or swap operations on the individual container elements.*

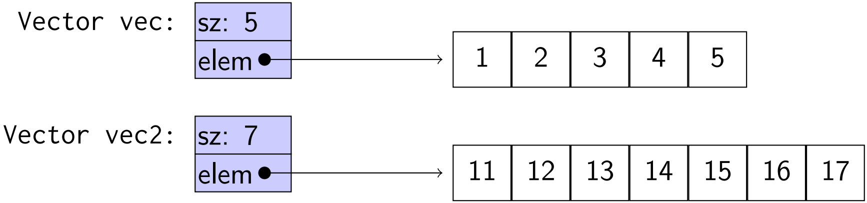
and

> *no swap() function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: The end() iterator does not refer to any element, so it may be invalidated. — end note]*

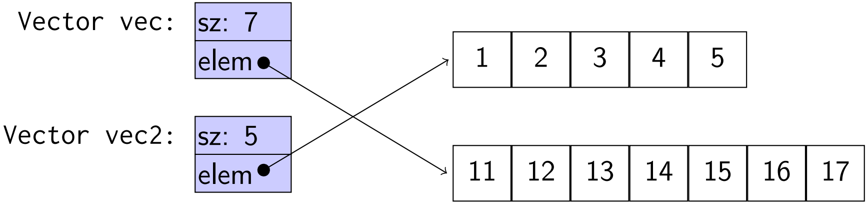
C++-14 clarifies:

> *Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap.*

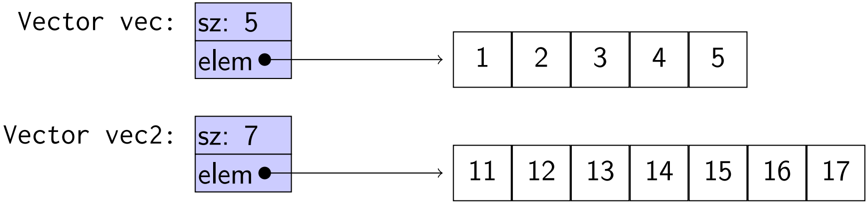
Swapping vectors vs. swapping elements
std::swap swaps the pointers



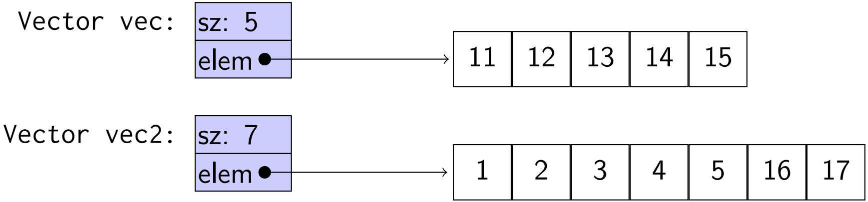

```
using std::swap;
swap(vec, vec2);
```



std::swap_ranges swaps elements




```
std::swap_ranges(vec.begin(), vec.end(), vec2.begin());
```



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`

References

References are similar to pointers, but

- A reference is *an alias to* a variable
  - cannot be changed (*reseated* to refer to another variable)
  - must be initialized
  - is not an object (has no address)

  - Dereferencing does not use the operator *
    - Using a reference *is* to use the referenced object.

Use a reference if you don't have (a good reason) to use a pointer.

Pointers

Similar to references in Java, but

- a pointer is the *memory address of an object*
- a pointer *is an object* (a C++ reference is not)
  - can be assigned and copied
  - has an address
  - can be declared without initialization, but then it gets an *undefined value* , as do other variables
- four possible states
  1. point to an object
  2. point to the address immediately past the end of an object
  3. point to nothing: `nullptr`. Before C++11: `NULL`
  4. invalid
- can be used as an iteger value
  - arithmetic, comparisons, etc.

References vs pointers

Use a reference if you don't have (a good reason) to use a pointer.

- E.g., if it may have the value `nullptr` (*"no object"*)
- or if you need to change("reseat") the pointer,
- for dynamically allocated objects: (**`new`** returns a pointer),
- for creating objects of polymorph types, and especially
- for storing polymorph types in a container like `std::vector`.

Pointers and references

Pointer and reference versions of swap

```
// References
void swap(int& a, int& b)
{

   int tmp = a;
   a = b;
   b = tmp;
}
```

```
// Pointers
void swap(int* pa, int* pb)
{
  if(pa != nullptr && pb != nullptr) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
  }
}
```

```
int m=3, n=4;
swap(m,n);   Reference version is called

swap(&m,&n); Pointer version is called
```

NB! Pointers are *called by value*: *the address* is copied

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
static members	7.6
inline	6.5.2, p 273