EDAF30 – Programming in C++

4. The standard library. Algorithms and containers.

Sven Gestegård Robertz
Computer Science, LTH

2023

Outline

User-defined types
Concrete classes

A concrete type

- ▶ "behaves just like a built-in type"
- ▶ the representation is part if the definition,
  That allows us to
  - ▶ place objects
    - ▶ on the stack (i.e., local variables)
    - ▶ in other objects (i.e., member variables)
    - ▶ in statically allocated memory (e.g., global variables)
  - ▶ copy objects
    - ▶ assignment of a variable
    - ▶ copy-constructing an object
    - ▶ value parameter of a function
  - ▶ refer to objects directly (not just using pointers or references)
  - ▶ initialize objects directly and completely (with a *constructor*)

Constructors

Copy Constructor

- ▶ Is called when initializing an object
- ▶ Is *not called* on assignment
- ▶ Can be defined, otherwise a standard copy constructor is generated (=`default`, =`delete`)

```
Bar b1(10, false};

Bar b2{b1};          // the copy constructor is called
Bar b3(b2);          // the copy constructor is called
Bar b4 = b2;         // the copy constructor is called

void function(Bar); // by-value parameter

function(b2);        // the copy constructor is called
```

Copy Constructors
default

- ▶ Declaration:

```cpp
class C {
public:
    C(const C&) =default;
};
```

- ▶ default copy constructor
  - ▶ Is automatically generated if not defined in the code
    - ▶ exception: if there are members that cannot be copied
  - ▶ But do declare it =**default** when you want it.
  - ▶ *shallow copy* of each member
    - ▶ Works for member variables of built-in types,
    - ▶ or *classes that behave like built-in types* (RAII-types)
    - ▶ *Does not work* for classes which manage resources "manually" (More on this later)

```cpp
class Vector{
public:
    Vector(int s) :sz{s},elem{new double[s]} {}
    ~Vector() {delete[] elem;}
    double& operator[](int i) {return elem[i];}
    int size() {return sz;}
private:
    int sz;
    double* elem;
};
```
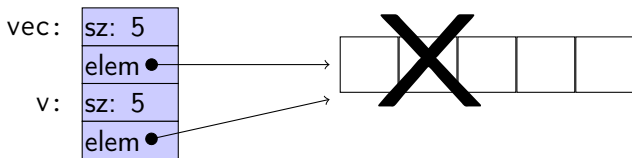

Vector vec: sz: 5, elem

No copy constructor defined ⇒ default generated.

＃

Classes
Default copy construction: shallow copy

```
void f(Vector v);

void test()
{
    Vector vec(5);
    f(vec); // call by value -> copy
    // ... other uses of vec
}
```

vec: sz: 5
      elem●
  v: sz: 5
      elem●

▶ The parameter v is default copy constructed: the value of each
  member variable is copied
▶ When f() returns, the destructor of v is executed:
  delete[] elem;
▶ The array pointed to *by both copies* is deleted. Disaster!

Constructors
Special cases: zero or one argument

Copy Constructor

▶ Has the same class as parameter: `Bar::Bar(const Bar& b);`

Converting constructor

▶ A constructor that can be called with one argument defines an *implicit type conversion* from the type of the parameter

```cpp
class ComplexNumber {
public:
    ComplexNumber():re{0},im{0} {}
    ComplexNumber(const ComplexNumber& k) :re{k.re},im{k.im} {}
    ComplexNumber(double x, double i=0):re{x},im{i} {}
    //...
private:
    double re;
    double im;
};
```

default constructor   copy constructor   converting constructor

Converting constructor
Warning - implicit conversion

```cpp
class Vector{
public:
    Vector(int s);      // create Vector with size s
    ...
    int size() const; // return size of Vector
    ...
};

void example_vector()
{
    Vector v = 7;

    std::cout << "v.size(): " << v.size() << std::endl;

}
```
    v.size(): 7

In std::vector the corresponding constructor is declared

    explicit vector(size_type count);

Converting constructor and `explicit`

explicit specifies that a constructor does not allow implicit type conversion.

```
struct A                    struct B
{                           {
  A(int);                     explicit B(int);
  // ...                      // ...
};                          };


A a1(2);      // OK         B b1(2);      // OK
A a2 = 1;     // OK         B b2 = 1;     // Error! [2]
A a3 = (A)1;  // OK         B b3 = (B)1;  // OK: explicit cast

a3 = 17;      // OK [1]     b3 = 17;      // Error! [3]
```

    [1]: construct an A(17), **and** then copy
    [2]: conversion from 'int' to non-scalar type 'B' requested
    [3]: no match for 'operator=' (operand types are 'B' and 'int')

```
Bar b1(10, false};

Bar b2{b1};         // the copy constructor is called
Bar b3(b2);         // the copy constructor is called
Bar b4 = b2;        // the copy constructor is called

void function(Bar); // by-value parameter

function(b2);       // the copy constructor is called

b4 = b3;            // the copy constructor is not called
```

copy assignment – not construction

The *copy assignment operator* is implicitly defined

- with the type `T& T::operator=(const T&)`
- if no copy assignment operator is declared for the type
- if all member variables can be copied
    - i.e., define a copy-assignment operator

- If all members are of built-in (and RAII) types the default variant works (same problems as with copy ctor).
- More on copy control when we discuss resource management

Preventing copying

- Declaration:
  ```cpp
  class C {
  public:
      C(const C&) =delete;
      C& operator=(const C&) =delete;
  };
  ```

- A class without copy constructor and copy assignment operator cannot be copied.
  - C++-98: declare private and don't define

- ▶ Uses *type parameters* to write more generic classes and functions
- ▶ No need to manually write a new class/function for each data type to be handled
- ▶ static polymorphism
- ▶ A template is *instantiated* by the compiler for the type(s) it is used for
  - ▶ each instance is a separate class/function
    - ▶ *different from java*: a java.util.ArrayList<T> holds java.lang.Object *references*
  - ▶ at compile-time: no runtime overhead
  - ▶ increases code size

Generic programming
Function templates

Example:
instead of

```
void print(int);
void print(double);
void print(const std::string&);

template <typename T> print(const T&);
```

- ► The compiler *instantiates* the template at the call site
- ► The entire *definition* of the template is needed
  - ► place template definitions in header files
- ► *Duck typing: if it walks like a duck, and quacks like a duck, it **is** a duck.*
  - ► cf. dynamically typed languages like python
- ► Independent of class hierarchies
  - ► E.g., in Java: a class must implement `Comparable`
  - ► in C++, a < b must be well-formed
- ► Requirements on the *use* of an object rather than its *type*
  - ► instead of "**class** T must have a function foo(U)"
  - ► "for objects t and u, the expression t.foo(u) is well-formed."
  - ► operator overloading: a+b or a < b is well-formed
  - ► a template can work for both built-in and user-defined types

```cpp
class Vector{
public:
    explicit Vector(int s);
    ~Vector() {delete[] elem;}
    double& operator[](int i) {return elem[i];}
    int size() const {return sz;}
private:
    int sz;
    double* elem;
};
```

can be generalized to hold any type:

```cpp
template <typename T>
class Vector{
public:
    ...
    T& operator[](int i) const {return elem[i];}
private:
    int sz;
    T* elem;
};
```

Generic programming
example: find an element in a Vector

```cpp
template <typename T>
T& find(const Vector<T>& v, const T& val)
{
    if(v.size() == 0) throw std::invalid_argument("empty vector");
    for(int i=0; i < v.size(); ++i){
        if(v[i] == val) return v[i];
    }
    throw std::runtime_error("not found");
}
```

- ▶ specific to Vector
- ▶ returning a reference is problematic: cannot return null
  - ▶ special handling of empty vector
  - ▶ special handling of element not found
- ▶ returning by value not a solution:
  - ▶ most types do not have a "not found" value
  - ▶ sometimes we need a reference to the actual found element, and not a copy.

Generic programming
example: find an element in an int array

```
int* find(int* first, int* last, int val)
{
    while(first != last && *first != val) ++first;
    return first;
}
```

Generalize to any array (pointer to ~~int~~ type parameter T).

```
template <typename T>
T* find(T* first, T* last, const T& val)
{
    while(first != last && *first != val) ++first;
    return first;
}
```

The standard library uses an abstraction for an element of a collection – *iterator*

- ▶ "points to" an element
- ▶ can be dereferenced
- ▶ can be incremented (to point to the following element)
- ▶ can be compared to another iterator

and two functions

begin() get an iterator to the first element of a collection

end() get an one-past-end iterator

Generic programming
example: find an element in a collection

find using pair of pointers

```
template <typename T>
T* find(T* first, T* last, const T& val)
{
    while(first != last && *first != val) ++first;
    return first;
}
```

Pointers are iterators for built-in arrays.

Find for any iterator range

```
template <typename Iter, typename T>
Iter find(Iter first, Iter last, const T& val)
{
    while(first != last && *first != val) ++first;
    return first;
}
```

Example implementation of `begin()` and `end()`:

```
template <typename T>
class Vector{
public:
    ...
    T* begin() {return sz > 0 ? elem : nullptr;}
    T* end() {return begin()+sz;}
    const T* begin() const {return sz > 0 ? elem : nullptr;}
    const T* end() const {return begin()+sz;}
private:
    int sz;
    T* elem;
};
```

The standard function template `std::begin()` has an overload for
classes with `begin()` and `end()` member functions.

Generic programming

Generic user code

```cpp
using std::begin;
using std::end;
void example1()
{
    int a[] {1,2,3,4,5,6,7};

    auto f5= find(begin(a), end(a), 5);
    if(f5 != end(a)) *f5 = 10;
}

void example2()
{
    Vector<int> a{1,2,3,4,5,6,7};

    auto f5= find(begin(a), end(a), 5);
    if(f5 != end(a)) *f5 = 10;
}
```

Generic programming

Generic user code

```
template <typename Iter>
void change_five_to_ten(Iter first, Iter last)
{
    auto f5= find(first, last, 5);
    if(f5 != last) *f5 = 10;
}

using std::begin;
using std::end;
void example1()
{
    int a[] {1,2,3,4,5,6,7};
    change_five_to_ten(begin(a), end(a));
}

void example2()
{
    Vector<int> a{1,2,3,4,5,6,7};
    change_five_to_ten(begin(a), end(a));
}
```

Algorithms

Standard libray algorithms

```
#include <algorithm>
```

Numeric algorithms:

```
#include <numeric>
```

Random number generation

```
#include <random>
```

Appendix A.2 in Lippman gives an overview

Standard algorithms

Main categories of algorithms

1. Search, count
2. Compare, iterate
3. Generate new data
4. Copying and moving elements
5. Changing and reordering elements
6. Sorting
7. Operations on sorted sequences
8. Operations on sets
9. Numeric algorithms

Standard algorithms

Algorithms operate on iterators.

Algorithm limitations

- Algorithms may *modify container elements*. E.g.,
    - `std::sort`
    - `std::replace`
    - `std::copy`
    - `std::remove` (sic!)
- No algorithm *inserts or removes container elements*.
    - That requires operating on the actual container object
    - or using an *insert iterator* that knows about the container (cf. `std::back_inserter`)

```cpp
template <class InputIterator, class T>
InputIterator
find (InputIterator first, InputIterator last, const T& val);
```

Example:

```cpp
vector<std::string> s{"Kalle", "Pelle", "Lisa", "Kim"};

auto it = std::find(s.begin(), s.end(), "Pelle");

if(it != s.end())
    cout << "Found " << *it << endl;
else
    cout << "Not found"<< endl;
Found Pelle
```

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()
```

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 ctor arguments (e.g., sizes), and {} for list of elements.

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

```
vector<int> v{3,4};
print_seq(v);          length = 2: [3][4]

v.assign(3,4);
print_seq(v);          length = 3: [4][4][4]


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

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

only in `deque`: `*_front`

```cpp
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.

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

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

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

Operations on `std::map`

```
insert, emplace, [], at, erase, clear,
size, empty,
find, count, lower_bound, upper_bound, equal_range
```

Operations on `std::set`

```
insert, emplace, erase, clear,
size, empty,
find, count, lower_bound, upper_bound, equal_range
```

Use the member functions, not algorithms like `std::find()`
(It may work, but is more complicated and less efficient – linear time complexity instead of logarithmic.)

Sets and maps

```cpp
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";
    }
}
```

Or

```
1 3 5 7
one is in the set            if(ints.count(1))
```

Sets and maps

```
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
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;
```

pair and tuple

- ▶ fixed-size heterogenous container
- ▶ can be used to return multiple values

std::pair is defined in <utility>
std::tuple is defined in <tuple>

pairs
Example: explicit element access

Getting the elements of a pair

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

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

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

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

Getting the elements of a tuple

```
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*.

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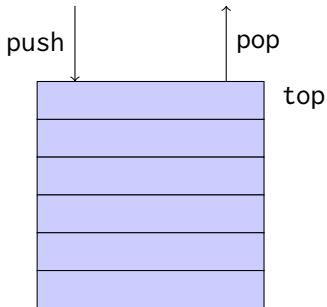
Has a default underlying container. E.g., for `stack`:

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

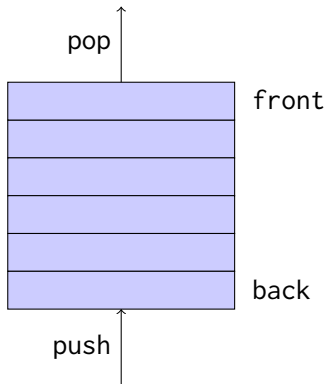but `stack` can be instantiated with any class that has `push_back()`, `pop_back()` and `back()`.

Queues and stacks

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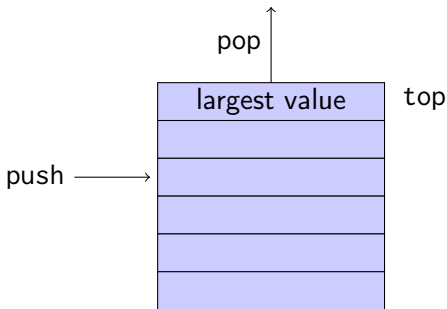- Operations: `push`, `pop`, `top`, `size` and `empty`

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.

Suggested reading

References to sections in Lippman

Next lecture

References to sections in Lippman

Function templates 16.1.1

Algorithms   10 – 10.3.1, 10.5

Iterators   10.4

Function objects 14.8

Random numbers 17.4.1