EDAF30 – Programming in C++

5. Functions. The standard library.

Sven Gestegård Robertz

Computer Science, LTH

2022

Outline

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

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`)

Algorithms
Exempel: find

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

Exempel:

```
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
```

Algorithms
Example: find_if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last,
                       UnaryPredicate pred);
```

Exempel:

```
bool is_odd(int i) { return (i % 2) == 1; }

void test_find_if()
{
    vector<int> v{2,4,6,5,3};

    auto it = std::find_if(v.begin(), v.end(), is_odd);

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

```
    Found 5
```

Function pointer

Algorithms
`count` och `count_if`

Count elements, in a data structure, that satisfy some predicate

- `std::count(first, last, value)`
  - elements equal to value
- `std::count_if(first, last, predicate)`
  - elements for which predicate is true

Algorithms
Example: copy and copy_if

```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last,
                     OutputIterator result);
```

Example:

```
vector<int> a(8,1);

print_seq(a);          length = 8: [1][1][1][1][1][1][1][1]

vector<int> b{5,4,3,2};

std::copy(b.begin(), b.end(), a.begin()+2);
print_seq(a);          length = 8: [1][1][5][4][3][2][1][1]
```

copy_if *with predicate, as previous slide*

Algorithms
remove / remove_if

Remove elements equal to a value or matching a predicate.

- std:remove et al. do not actually remove anything. They
  - move the "retained" elements to the front
  - return an iterator to the first "removed" element
- To actually remove from a container, use the erase member function, e.g std::vector::erase()

The erase-remove idiom

```
auto new_end = std::remove_if(c.begin(), c.end(), pred);
c.erase(new_end, c.end());
```

or

```
c.erase(std::remove_if(c.begin(), c.end(), pred),c.end());
```

Algorithms
Insert iterators (in <iterator>)

Example:

```
vector<int> v{1, 2, 3, 4};

vector<int> e;
std::copy(v.begin(), v.end(), std::back_inserter(e));
print_seq(e);
                    length = 4: [1][2][3][4]

deque<int> e2;
std::copy(v.begin(), v.end(), std::front_inserter(e2));
print_seq(e2);
                    length = 4: [4][3][2][1]

std::copy(v.begin(), v.end(), std::inserter(e2, e2.end()));
print_seq(e2);      length = 8: [4][3][2][1][1][2][3][4]
```

Requirements on iterators

The standard library algorithms put requirements on iterators.
For instance, `std::find` requires its arguments to be

CopyConstructible (and Destructible) as it is passed by value
EqualityComparable to have **operator**!=
Dereferencable to have **operator**∗ (for reading)
Incrementable to have **operator**++

The requirements are often specified using iterator concepts.

Iterator concepts

- Input Iterator (`++ == != `) (dereference as *rvalue*: *a, a->)
- Output Iterator (++) (dereference as *lvalue*: *a=t)
- Forward Iterator (Input- and Output Iterator, reusable)
- Bidirectional Iterator (as Forward Iterator with `--`)
- Random-access Iterator (`+=`, `-=`, `a[n]`, `<`, `<=`, `>`, `>=`)

Different iterators for a container type (`con` is one of the containers `vektor`, `deque`, or `list` with the element type `T`)

con<T>::iterator	runs forward
con<T>::const_iterator	runs forward, only for reading
con<T>::reverse_iterator	runs backwards
con<T>::const_reverse_iterator	runs backwards, only for reading

Iteratorer validity

In general, if the structure an iterator is referring to is changed *the iterator is invalidated*. Example:

- insertion
  - sequences
    - `vector`, `deque`* : all iterators are invalidated
    - `list` : interators are unaffected
  - associative containers (`set`, `map`)
    - iterators are unaffected
- removal
  - sequences
    - `vector` : iterators *after* the removed elements are invalidated
    - `deque` : all iterators invalidated (in principle*)
    - `list` : iterators to the removed elements are invalidated
  - associative containers (`set`, `map`)
    - iterators are unaffected
- `resize`: as insertion/removal

istream_iterator<T>

istream_iterator<T> : constructors

```
istream_iterator();  // gives an end-of-stream istream iterator
istream_iterator (istream_type& s);
```

```
#include <iterator>

stringstream ss{"1 2 12 123 1234\n17\n\t42"};

istream_iterator<int> iit(ss);
istream_iterator<int> iit_end;

while(iit != iit_end) {
    cout << *iit++ << endl;
}
```

```
1
2
12
123
1234
17
42
```

istream_iterator<T>

Example: use to initialize a vector<**int**>:

```
stringstream ss{"1 2 12 123 1234\n17\n\r42"};

istream_iterator<int> iit(ss);
istream_iterator<int> iit_end;

vector<int> v(iit, iit_end);

for(auto a : v) {
    cout << a << " ";
}
cout << endl;
```

```
1 2 12 123 1234 17 42
```

istream_iterator<T>

Example: counting words in a string s:

Straight-forward counting

```
istringstream ss{s};
int words{0};
string tmp;
while(ss >> tmp) ++words;
```

Using the standard library

```
istringstream ss{s};
int words = distance(istream_iterator<string>{ss},
                     istream_iterator<string>{});
```

std::distance gives the distance (in number of elements) between two iterators. (UB if the second argument cannot be reached by incrementing the first.)

istream_iterator
Handling errors

```
stringstream ss2{"1 17 kalle 2 nisse 3 pelle\n"};
istream_iterator<int> iit2{ss2};
istream_iterator<int> iit_end;
while(!ss2.eof()) {
  while(iit2 != iit_end) { cout << *iit2++ << endl; }
  if(ss2.fail()){
    ss2.clear();
    string s;
    ss2 >> s;
    cout << "ss2: not an int: " << s << endl;
    iit2 = istream_iterator<int>(ss2); // create new iterator
  }
}
cout << boolalpha << "ss2.eof(): " << ss2.eof() << endl;
```

```
1
17
ss2: not an int: kalle
2
ss2: not an int: nisse
3
ss2: not an int: pelle
ss2.eof(): true
```

- on failure, the `fail`-bit is set in the stream
- the iterator is set to `end`
- if the stream is changed, a new iterator must be created

ostream_iterator and the algorithm copy

ostream_iterator

```
ostream_iterator (ostream_type& s);
ostream_iterator (ostream_type& s, const char_type* delimiter);
```

```
std::vector<int> v{1,2,12,1234,17,42};
cout << fixed << setprecision(2);
ostream_iterator<double> oit{cout, " <-> "};

std::copy(begin(v), end(v), oit);
```

```
1.00 <-> 2.00 <-> 12.00 <-> 1234.00 <-> 17.00 <-> 42.00 <->
```

transform and function objects

Iterate over a sequence, apply a function to each element and write the result to a sequence (*cf. "map" in functional programming languages*)

```
template < class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first, InputIt last, OutputIt d_first,
                    UnaryOperation unary_op );

template < class InputIt1, class InputIt2, class OutputIt,
           class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,
                    OutputIt d_first, BinaryOperation binary_op );
```

A function object is an object that can be called as a function.,

- function pointers
- function objects (*"functor"*)

The algorithm `transform` can handle both function pointers and functors.

Function objects and transform

Example with function pointer

```
int square(int x) {
    return x*x;
}

vector<int> v{1, 2, 3, 5, 8};
vector<int> w; // w is empty!

transform(v.begin(), v.end(), back_inserter(w), square);

// w = {1, 4, 9, 25, 64}
```

Function objects

A function object is an object that has **operator**()

Previous example with a function object

```
struct {
    int operator() (int x) const {
        return x*x;
    }
} sq;

vector<int> v{1, 2, 3, 5, 8};
vector<int> ww;  // ww empty!

transform(v.begin(), v.end(), back_inserter(ww)), sq);

// ww = {1, 4, 9, 25, 64}
```

Anonymous struct – *the type* has no name, only *the object*.

Function objects

The value of a lambda expression is a function object

Previous function object

```
struct {
    int operator() (int x) const {
        return x*x;
    }
} sq;
transform(v.begin(), v.end(), back_inserter(ww)), sq);
```

Previous example with a lambda

```
auto sq = [](int x){return x*x;};
transform(v.begin(), v.end(), back_inserter(ww)), sq);
```

Function objects
functions with state

Function objects can be used to create functions with state (more flexible than static local variables).

Example

```
struct {
    int operator()(int x) {return val+=x;}
    int get_sum() const {return val;}
    void reset() {val=0;}
    int val=0;
} accum;

std::vector<int> v{1,2,3,4,5};

for(auto x : v) {
    cout << "sum is " << accum(x) << endl;
}
cout << "Total sum is " << accum.get_sum() << endl;
```

Random numbers

<cstdlib>

Example: dice with the C standard lib

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using std::cout;
using std::endl;

int main( )
{
  unsigned int seed = time(0);
  srand(seed);
  int n{20};
  for (int i=0; i<n; i++) {
    cout << rand()%6+1 << " ";
  }
  cout << endl;
}
```

Random numbers
Better C++: encapsulate in an object – "function with state"

Assume that we have a class `Rand_int` giving random numbers in the interval $[min, max]$.

with RandInt object

```
int main()
{
  unsigned long seed = time(0);
  Rand_int dice{1,6, seed};
  int n{20};
  for(int i = 0; i != n; ++i) {
    cout << dice() << " ";
  }
  cout << endl;
}
```

The C version

```
int main( )
{
  unsigned int seed = time(0);
  srand(seed);
  int n{20};
  for (int i=0; i<n; i++) {
    cout << rand()%6+1 << " ";
  }
  cout << endl;
}
```

Random numbers
Example of a random integer class

Example: Rand_int

```
#include <random>

class Rand_int {
public:
    Rand_int(int low, int high) :dist{low,high} {}
    Rand_int(int low, int high, unsigned long seed)
            :re{seed}, dist{low,high} {}
    int operator()() {return dist(re);}
private:
    std::default_random_engine re;
    std::uniform_int_distribution<> dist;
};
```

Function objects
lambda expressions

syntax:

```
[capture] (parameters) -> return type {statements}
```

where

- **capture** specifies by value (`[=]`) or by reference (`[&]`), default or for each named variable (e.g., `[&x, y]`)
- **parameters** are like normal function parameter declaration
- **return type** can be inferred from **return** statements if unambiguous

Example

```
auto plus = [](int a, int b) {return a + b;}

int x = 10;
auto plus_x = [=](int a) {return a + x;}  // x is captured
```

Function objects
Predefined function objects: <functional>

Functions:

```
plus, minus, multiplies, divides, modulus, negate,
equal_to, not_equal_to, greater, less, greater_equal,
less_equal, logical_and, logical_or, logical_not
```

Predefined function object creation

```
operation<type>()
```

E.g.,

```
auto f = std::plus<int>();
```

Function objects
Example: std::plus from <functional>

transform with binary function

```
vector<int> v1{1,2,3,4,5};
vector<int> v2(10,10);

vector<int> res2;
auto it = std::back_inserter(res2);
auto f = std::plus<int>();
std::transform(v1.begin(), v1.end(), v2.begin(), it, f);

print_seq(res2);
```

```
length = 5: [11][12][13][14][15]
```

Example with accumulate <numeric>

```
auto mul = std::multiplies<int>();
int prod = std::accumulate(v1.begin(), v1.end(), 1, mul);

cout << "product(v1) = " << prod << endl;
```

```
product(v1) = 120
```

Function objects
Example: a function object class template

```
template<typename T>
class Less_than {
    const T val;
public:
    Less_than(const T& v) :val{v} {}
    bool operator()(const T& x) {return x < val;}
};

void use_less_than()
    auto lt5 = Less_than<int>(5);      // or Less_than<int> lt5{5}
    std::vector<int> v{1, 7, 6, 2, 8, 4, 9, 3};
    std::vector<int> small;

    std::copy_if(begin(v), end(v), std::back_inserter(small), lt5);

    for (auto x : small) {
        cout << x << " ";
    }
    cout << endl;
}
```

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);
```

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&)`

Suggested reading

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

Next lecture

Resource management

References to sections in Lippman

Dynamic memory and smart pointers 12.1

Dynamically allocated arrays 12.2.1

Classes, resource management 13.1, 13.2