

EDAF50 – C++ Programming

6. *Generic programming. Algorithms.*

Sven Gestegård Robertz
Computer Science, LTH

2022



Outline

- 1 Generic programming
- 2 Standard library algorithms
 - Algorithms
 - Insert iterators
- 3 Iterators
 - Different kinds of iterators
 - stream iterators
- 4 Algorithms and function objects

Generic programming

Templates (mallar)

- ▶ 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&);
```

Templates

Template compilation

- ▶ 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
- ▶ 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
- ▶ Independent of class hierarchies
 - ▶ E.g., in Java: a class must implement Comparable
 - ▶ in C++, a < b must be well-formed

Generic programming

A class for a vector of doubles

```
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:

```
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

```
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;
}
```


Generic programming

Iterators

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;
}
```

Generic programming

A generic Vector class

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 user code

```
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 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 library algorithms

```
#include <algorithm>
```

Numeric algorithms:

```
#include <numeric>
```

Random number generation

```
#include <random>
```

Appendix A.2 in Lippman gives an overview

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

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

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

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`)

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

Iterator 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 : iterators 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> : 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
```

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

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

```
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 <->
```

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.

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(wv)), sq);
```

Previous example with a lambda

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

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

Function templates

References to sections in Lippman

Customizing algorithms 10.3.1

Lambda expressions 10.3.2 – 10.3.4

Binding arguments 10.3.4

Function objects 14.8

Class templates 16.1.2

Template arguments and deduction 16.2–16.2.2

Type aliases 2.5.1

Trailing return type 16.2.3

Templates and overloading 16.3