EDAF50 – C++ Programming

7. Generic programming

Sven Gestegård Robertz
Computer Science, LTH

2018

Outline

Function templates

Example: compare

```
template<class T>
const T& compare(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < 1) return 1;
    return 0;
}
```

Can be instantiated for all types that have an **operator**<

Function templates
Requirements on types

Example: another compare template

```
template<class T>
int compare(T a, T b) {
    if (a < b) return -1;
    if (a == b) return 0;
    return 1;
}
```

More requirements on the type T:

- call-by-value: T must be copy constructible
- needs both **operator**< and **operator**==

Try to minimize the requirements on T

Templates
Concepts

- A concept is a named set of requirements (on a type)
- for template arguments
- Not yet part of the C++ language

Some example concepts

DefaultConstructible Objects can be constructed without explicit initialization

CopyConstructible, CopyAssignable Objects (of type X) can be copied and assigned.

LessThanComparable `a < b` is defined

EqualityComparable `a == b` and `a != b` is defined

Iterator and the more specific `InputIterator`, `OutputIterator`, `ForwardIterator`, `RandomAccessIterator`, etc.

Function templates
Example: type deduction

```
template <typename T>
int compare(const T& a, const T& b)
{
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}

void example()
{
    int x{4};
    int y{2};
    cout << "compare(x,y): " << compare(x,y) << endl;      T is int

    string s{"Hello"};
    string t{"World"};
    cout << "compare(s,t): " << compare(s,t) << endl;      T is string
}
```

The compiler can (often) infer the template parameters from the function arguments.

Function templates
Parameters must match

```
template <typename T>
int compare(const T& a, const T& b);
```

Example: compare

```
int i{5};
double d{5.5};

cout << compare(i,d) << endl;
```

`error: no matching function for call to 'compare(int&, double&)'`

- First argument gives: `T` is **int**
- Second argument gives: `T` is **double**
- Template is not instantiated (not an error)
- There is no function `compare(int,double)` (error)

Types must match exactly. No implicit conversions are performed.

Templates
Template instantiation: SFINAE

Substitution Failure Is Not An Error

If a template instantiation produces ill-formed code

- it is considered not viable
- and is silently discarded.

If *no viable instantiation* is found it is an error
("no such class/function")

Function templates
Explicit instantiation

```
template <typename T>
int compare(const T& a, const T& b);
```

Example: compare with explicit instantiation

```
int i{5};
double d{5.5};

cout << compare<double>(i,d) << endl;    // -1
cout << compare<int>(i,d) << endl;       // 0
```

An explicitly instantiated function template is just a function.
⇒ implicit type conversion of arguments

Function templates

Example: two template parameters

```
template <typename T, typename U>
int compare2(const T& a, const U& b)
{
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}

void example3()
{
    int i{5};
    double d{5.5};

    cout << compare2(i,d) << endl;  // -1
}
```

- First argument gives: `T` is **int**
- Second argument gives: `U` is **double** *OK!*

Function templates

Example: the minimum function

```
template<class T>
const T& minimum(const T& a, const T& b) {
    if (a < b)
        return a;
    else
        return b;
}
```

Can be instantiated for all types that have the operator <

Function templates
Overloading with a normal function

```
struct Name{
    string s;
    //...
};
```

Overload for `Name&`

```
const Name& minimum(const Name& a, const Name& b)
{
    if(a.s < b.s)
        return a;
    else
        return b;
}
```

Function templates
Trailing return type (c++11)

```
template <typename T, typename U>
T minimum(const T& a, const U& b);
```

Would not always work, as the return type is always that of the first argument.

```
template <typename T, typename U>
auto minimum(const T& a, const U& b) -> decltype(a+b)
{
    return (a < b) ? a : b;
}

void example()
{
    int a{3};
    int b{4};

    double x{3.14};

    cout << "minimum(x,a); " << minimum(x,a) << endl;    // 3
    cout << "minimum(x,b); " << minimum(x,b) << endl;    // 3.14
}
```

- **decltype** is an *unevaluated context*
- the expression a + b is not evaluated
- **decltype** gives the *type* of an expression
- NB! Return-by-value as argument may need to be converted

Function templates
min_element: minimum element in iterator range

```
template<typename FwdIterator>
FwdIterator min_element(FwdIterator first, FwdIterator last)
{
    if(first==last) return last;

    FwdIterator res=first;

    auto it = first;
    while(++it != last){
        if(*it < *res) res = it;
    }
    return res;
}
```

Use:

```
int a[] {3,5,7,6,8,5,2,4};
auto ma  = min_element(begin(a), end(a));
auto ma2 = min_element(a+2,a+4);

vector<int> v{3,5,7,6,8,5,2,4};
auto mv = min_element(v.begin(), v.end());
```


Function templates
std::min_element for types that don't have <

Overload with a second template parameter: Compare

```
template<class FwdIt, class Compare>
FwdIt min_element(FwdIt first, FwdIt last, Compare cmp)
{
    if(first==last) return last;

    FwdIt res=first;
    auto it = first;
    while(++it != last){
        if (cmp(*it, *res)) res = it;
    }
    return res;
}
```

Compare must have **operator**() and the types must match, e.g.,:

```
class Str_Less_Than {
public:
    bool operator () (const char *s1, const char *s2) {
        return strcmp(s1, s2) < 0;
    }
};
```


Function templates
std::min_element for types that don't have <

Example use on list of strings:

```
list<const char *> tl = { "hej", "du", "glade" };
Str_Less_Than lt; // functor

cout << *min_element(tl.begin(), tl.end(), lt);
```

The Str_Less_Than object can be created directly in the argument list:

```
cout << *min_element(tl.begin(), tl.end(), Str_Less_Than());
```

(C++11) lambda: anonymous functor

```
auto cf= [](const char* s, const char* t){return strcmp(s,t)<0;};

cout << *min_element(tl.begin(), tl.end(), cf);
```


Function objects
lambda expressions

syntax:

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

where

- capture: specifies by value ([=]) or by reference ([&]), default or for each named variable
- 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
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) {
    accum(x);
    cout << "sum is " << accum.get_sum() << endl;
}
```


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()
{
    Less_than<int> lt5{5};
    Vector<int> v{1,7,6,2,8};

    for(auto x : v) {
        cout << x << " < 5:" << boolalpha << lt5(x) << endl;
    }

}
```


Function objects
partial application: std::bind (in <functional>)

std::bind() : create a new function object by "partial application" of a function (object)

Example

```
std::vector<int> v = {1,3,2,4,3,5,4,6,5,7,6,8,3,9};
std::vector<int> w;

using std::placeholders::_1;
auto gt5 = std::bind(std::greater<int>(), _1, 5);

std::copy_if(v.begin(), v.end(), std::back_inserter(w), gt5);
```

or `using namespace std::placeholders;`

Function objects
the std::function type (in <functional>)

std::function is a type that can hold anything you can invoke with **operator()**.

Example

```
int call_f(std::function<int(int,int)> f, int x, int y){
  return f(x,y);
}

int add(int,int);
```

call_f can be called with anything callable $(int, int) \to int$:
a function pointer, functor, or lambda expression:

```
cout << call_f(add,10,20) << endl;
cout << call_f(std::multiplies<int>{},10,20) << endl;
cout << call_f([](int a, int b){return a+10*b;},10,20) << endl;
```


Class templates

- The container classes vector, deque and list are examples of *parameterized classes* or *class templates*
- The compiler uses the class template to *instantiate* a class with the given actual parameters
- No need to manually write a new class for every element type

- Classes can be parameterized
- Example: container classes in the standard library
  - std::vector
  - std::deque
  - std::list

"Container" is a generic concept, independent of the element type

Parameterized types

- Generalize Vector of doubles to Vector of anything.
- Class template with the element type as template parameter.

Example:

```
template <typename T>
class Vector{
private:
    T* elem;
    int sz;
public:
    explicit Vector(int s);
    ~Vector() {delete[] elem;}

    // copy and move ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const {return sz;}
};
```


The class template Vector
Member functions

- Invariant:
  - sz $>= 0$ (*NB! declared* **int** sz, *not* **unsigned** sz)
  - elem pointer to a T[sz];

```
template <typename T>
Vector<T>::Vector(int s){
    if(s < 0) throw invalid_argument("Negative size");
    sz = s;
    elem = new T[sz];
};
template <typename T>
const T& Vector<T>::operator[](int i) const
{
    if(i < 0 || size() <= i) throw range_error("Vector::operator[]");
    return elem[i];
}
template <typename T>
T& Vector<T>::operator[](int i)
{
    const auto& constme = *this;
    return const_cast<T&>(constme[i]);
}
```


Iterators

To use our Vector<T> with *range-for* and standard algorithms we need the functions begin() and end()

```
template <typename T>
const T* begin(const Vector<T> &v)

template <typename T>
T* begin(Vector<T> &v)

template <typename T>
const T* end(const Vector<T>& v)

template <typename T>
T* end(Vector<T>& v)
```


Iterators
the const versions

To use our Vector<T> with *range-for* and standard algorithms we need the functions begin() and end()

The const versions

```
template <typename T>
const T* begin(const Vector<T> &v)
{
  return v.size() ? &v[0] : nullptr;
}

template <typename T>
const T* end(const Vector<T>& v)
{
  return begin(v)+v.size();
}
```


Iterators
The non-const versions

- Avoid code duplication
  - Use the **const** versions
  - Example of (the?) use of **const_cast**

The non-const versions

```
template <typename T>
T* begin(Vector<T> &v)
{
  return const_cast<T*>(begin(static_cast<const Vector<T>&>(v)));
}

template <typename T>
T* end(Vector<T>& v)
{
  return const_cast<T*>(end(static_cast<const Vector<T>&>(v)));
}
```


Explanation
the non-const versions

```
template <typename T>
T* begin(Vector<T> &v)
{
    return const_cast<T*>(begin(static_cast<const Vector<T>&>(v)));
}
```

can also be written

```
template <typename T>
T* begin(Vector<T> &v)
{
    const Vector<T>& cv= v;      create const-reference
    const T* cbegin = begin(cv);  return value is const pointer
    return const_cast<T*>(cbegin);       make non-const pointer
}
```

NB! Call the **const** *version from the non-***const** *version.*
Never the other way.

Const overloading
member function example

operator[] const and non-const

```
template <typename T>
const T& Vector<T>::operator[](int i) const
{
  if(i<0 || size()<=i) throw range_error("Vector::operator[]");
  return elem[i];
}

template <typename T>
T& Vector<T>::operator[](int i)
{
  const auto& constme = *this;
  return const_cast<T&>(constme[i]);
}
```


The Vector class template
Constructor with std::initializer_list

We want to initialize vectors with values:

```
Vector<int> vs{1,3,5,7,9};

template <typename T>
Vector<T>::Vector(initializer_list<T> l)
                 :Vector<T>(static_cast<int>(l.size()))
{
    std::copy(l.begin(), l.end(), elem);
}
```

The pedantic **static_cast<int>** *is used as* std::initializer_list<T>::size() *returns an* **unsigned** *type*

Template parameters
Types or values

```
template <typename T, int N>
struct Buffer{
    using value_type = T;
    constexpr int size() {return N;}
    T buf[N];
};
```

- Buffer: like an array that knows its size
  - No overhead for heap allocation
  - Template parameters must be **constexpr**
    → cannot have variable size
  - cf. std::array
- The size as value parameter to the template
- An alias (value_type) and a **constexpr** function (size())
  - Users can access (read) template parameter values

Template parameters
and alias

All standard containers has an alias value_type

```
template <typename T>
class Container{
public:
    using value_type = T;
    ...
};

template <typename Cont>
typename Cont::value_type& get_first(Cont& t)
{
    return *t.begin();
}

void example()
{
    Vector<int> v{2,4,3,5,4,6};
    cout << "first element of v is " << get_first(v) << endl;
}
```

Here **typename** *is needed by the compiler to know that the name* value_type *is a type before the template is instantiated*

Alias

using can be used to create type aliases

```
using size_t = unsigned int;
```

including templates:

```
using IntVector = Vector<int>;
```


Iterator traits
Exempel: find

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

Better way: the compiler knows the actual value type.

With decltype and std::declval<T>

```
template <class InIt, class T=decltype(* declval<InIt>())>
InIt find (InIt first, InIt last, const T& val);
```

With std::iterator_traits<Iterator>

```
template <class InIt,
          class T=typename iterator_traits<InIt>::value_type>
InIt find (InIt first, InIt last, const T& val);
```

Class Templates
Definition of function members

```
template <typename T>
void Vector<T>::print() const
{
  for(size_t i = 0; i != sz; ++i)
    cout << p[i] << " ";
  cout << endl;
}
```

- Function members in a class template are function templates
- `print()` works for all types with an **operator**<<
- *"Duck typing":*
  if it walks like a duck and quacks like a duck, it is a duck

Class Templates
Definition of member functions

```
template <typename T>
void Vector<T>::print() const
{
  for(size_t i = 0; i != sz; ++i)
    cout << p[i] << " ";
  cout << endl;
}
```

- Works for all types with **operator**<<
- but not for elements of type

```
struct Foo{
   int x;

   Foo(int d=0) :x{d}{}
};
```

Template specialization for the type `Foo`:

```
template<>  full specialization: no template arguments
void Vector<Foo>::print() const
{
  for(size_t i = 0; i != sz; ++i)
    cout << "Foo("<<p[i].x << ") ";
  cout << endl;
}
```


Template specialization

- Class Templates can be specialized
  - fully
  - partially
- Function templates can be specialized
  - fully
  - *but overloading is always preferable*

Templates, comments

- Templates have parameters
  - type parameters: declared with **class** or **typename**
  - value parameters: declared as usual, e.g., **int** N
- The compiler needs the template definition to instantiate
  ⇒ it must be in the *header file* (if used by others)
- Overloading:
  - Functions can be overloaded ⇒ function templates can be overloaded
  - Classes cannot be overloaded ⇒ class templates cannot be overloaded
- Template specialization:
  - Class templates can be specializd *partially* or *fully*
  - Function templates can only be *fully* specialized, *but*
    - Specializations are not overloaded
    - Often better/clearer to overload with a normal function (not a template) than to specialize

Next lecture

Classes and polymorphism

References to sections in Lippman

Dynamic polymorphism and inheritance chapter 15 – 15.4
Accessibility and scope chapter 15.5 – 15.6
Type conversions and polymorphism chapter 15.2.3
Inheritance and resource management 15.7
Polymorph types and containers 15.8
Multiple inheritance 18.3
Virtual base classes 18.3.4 – 18.3.5

Läsanvisningar

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
Trailing return type 16.2.3
Templates and overloading 16.3