EDAF30 – Programming in C++

9. Generic programming

Sven Gestegård Robertz
Computer Science, LTH

2022

Outline

Function templates

Example: compare

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

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.

$\Rightarrow$ *implicit type conversion of arguments*

Function templates
Prefer type casts to be clear about type converisons

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

Example: compare with explicit type conversion

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

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

For readability, it is often to better to be explicit about type conversions than to rely on 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
Use the standard library

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

adds the unnecessary requirement that there is an **operator**+.

Better:

```
#include <type_traits>

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

`std::common_type<T,U>::type` is
a type that T and U can be implicitly converted to *If no such a type exists, there is no member* `type`.

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:

```
std::vector<const char *> tl = { "strings", "in", "a", "vector" };

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);
```

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]);
}
```

Class templates
The Container classes

```
class Container {
public:
    virtual int size() const  =0;
    virtual int& operator[](int o) =0;
    virtual ~Container() {}
    virtual void print() const =0;
};

class Vector :public Container {
public:
    explicit Vector(int l);
    ~Vector();
    int size() const override;
    int& operator[](int i) override;
    virtual void print() const override;
private:
    int *p;
    int sz;
};
```

- generalize on element type

Class templates
Generic Container and Vector

```
template <typename T>
class Container {
public:
    using value_type = T;
    virtual size_t size() const =0;
    virtual T& operator[](size_t o) =0;
    virtual ~Container() {}
    virtual void print() const =0;
};

template <typename T>
class Vector :public Container<T> {
public:
    Vector(size_t l = 0) :elem{new T[l]},sz{l} {}
    ~Vector() {delete[] elem;}
    size_t size() const override {return sz;}
    T& operator[](size_t i) override {return elem[i];}
    virtual void print() const override;
private:
    T *elem;
    size_t sz;
};
```

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(std::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*

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
  $\Rightarrow$ it must be in the *header file* (if used by others)
- Overloading:
  - Functions can be overloaded $\Rightarrow$ function templates can be overloaded
  - Classes cannot be overloaded $\Rightarrow$ 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

Iterator traits
Exempel: `find`

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

Alternative: 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);
```

Variadic templates
A function template can take a variable number of arguments

```
void println() { base case: no argument
    cout << endl;
}

template <typename T, typename... Tail>
void println(const T& head, const Tail&... tail)
{
    cout << head << " ";     print the first element
    println(tail...);        recursion: print the rest
}

void test_variadic()
{
    string a{"Hej"};
    int b{10};
    double c{17.42};
    long d{100};

    println(a,b,c,d);

}
```

Example: A class hierarchy

```
class Animal{
public:
  void speak() const { cout << get_sound() << endl;}
  virtual string get_sound() const =0;
  virtual ~Animal() =default;
};

class Dog :public Animal{
public:
  string get_sound() const override {return "Woof!";}
};
class Cat :public Animal{
public:
  string get_sound() const override {return "Meow!";}
};
class Bird :public Animal{
public:
  string get_sound() const override {return "Tweet!";}
};
class Cow :public Animal{
public:
  string get_sound() const override {return "Moo!";}
```

Example
Use (not polymorphic)

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  d.speak();      Woof!
  c.speak();      Meow!
  b.speak();      Tweet!
  w.speak();      Moo!
}
```

Example
Call by reference

```
void test_polymorph(const Animal& a)
{
    a.speak();
}

int main()
{
    Dog d;
    Cat c;
    Bird b;
    Cow w;

    test_polymorph(d);          Woof!
    test_polymorph(c);          Meow!
    test_polymorph(b);          Tweet!
    test_polymorph(w);          Moo!
}
```

Example
Container with polymorph objects

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  std::vector<Animal> zoo{d,c,b,w};

  for(auto x : zoo){
    x.speak();
  };

}
```

```
error: cannot allocate an object of abstract type 'Animal'
```

Example
Must use container of pointers

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  std::vector<Animal*> zoo{&d,&c,&b,&w};

  for(auto x : zoo){
    x->speak();
  };
}
```

```
Woof!
Meow!
Tweet!
Moo!
```

The Curiously Recurring Template Pattern
Static polymorphism

- Polymorphism without the run-time overhead
  - Common functionality in base class
    - E.g., compute value
  - Specific functionality in derived classes
    - E.g., output to different devices (console, file, socket)
- Reuse of generic functionality in unrelated classes
  - Related to *Mixin classes*
  - E.g., counting allocations and instances

The Curiously Recurring Template Pattern
Dyanamic polymorphism

Normal abstract class

```
class Base{
public:
    virtual void method() =0;
};

class Derived1 :public Base{
public:
    void method() override{
        cout << "Derived1::method\n";
    }
};
```

The Curiously Recurring Template Pattern
Static polymorphism

The CRTP structure

```
template <typename T>
class Base {
public:
    void method() {
        static_cast<T*>(this)->method();
    }
};

class Derived : public Base<Derived> {
public:
    void method() {
        std::cout << "Derived method" << std::endl;
    }
};
```

The Curiously Recurring Template Pattern
Example: Animal sounds

```
class Animal {
public:
  Animal(const std::string& name) :name(name) {}
  void speak() const {cout << name << " says "<< get_sound() << "!\n";
  virtual std::string get_sound() const = 0;
  virtual ~Animal() =default;
private:
  std::string name;
};

class Dog : public Animal {
public:
  using Animal::Animal;
  virtual std::string get_sound() const override {return {"Woof"};}
};

class Cat : public Animal {
public:
  using Animal::Animal;
  virtual std::string get_sound() const override {return {"Meow"};}
};
```

The Curiously Recurring Template Pattern
Example: Animal sounds

If we don't need run-time polymorphism:

```
Dog d{"Fido"};
Cat c{"Caesar"};

d.speak();      Fido says Woof!
c.speak();      Caesar says Meow!
```

Base class template

```
template <typename Derived>
class Animal {
public:
    Animal(const std::string& name) :name(name) {}
    void speak() const {
        cout << name << " says "
        << static_cast<const Derived*>(this)->get_sound()
        << "!\n";
    }
private:
    std::string name;
};
```

The Curiously Recurring Template Pattern
Example: Animal sounds

Concrete derived classes

```
class Dog : public Animal<Dog> {
public:
    using Animal::Animal;
    std::string get_sound() const {
        return {"Woof"};
    }
};

class Cat : public Animal<Cat> {
public:
    using Animal::Animal;
    std::string get_sound() const {
        return {"Meow"};
    }
};
```

NB! No **override**

The Curiously Recurring Template Pattern
Example:

Base class template

```
template <typename Derived>
class Computer{
public:
    void print_answer(){
        auto ans = incredibly_complex_computation();
        static_cast<Derived*>(this)->do_print_answer(ans);
    }
private:
    int incredibly_complex_computation() {return 42;}
};
```

Behaves like it had a pure virtual function

```
    virtual void do_print_answer(int) =0;
```

The Curiously Recurring Template Pattern
Example:

Concrete classes

```
class Local_Computer :public Computer<Local_Computer>{
public:
    void do_print_answer(int ans) {
        cout << "Answer:" << ans << endl;
    }
};

class Networked_Computer :public Computer<Networked_Computer>{
public:
    Networked_Computer(ServerConnection c) :conn{c} {}
    void do_print_answer(int ans) {
        conn.upload(ans);
    }
private:
    ServerConnection conn;
};
```

```
    Local_Computer l{};
    l.print_answer();    Answer: 42
```

The Curiously Recurring Template Pattern
Static polymorphism

- Polymorphism without the run-time overhead
  - Common functionality in base class
    - E.g., compute value
  - Specific functionality in derived classes
    - E.g., output to different devices (console, file, socket)
- Reusing generic functionality in unrelated classes
  - E.g., counting allocations and instances

The Curiously Recurring Template Pattern
Example: counting instances

Base class template

```
template <typename Derived>
class Counted{
public:
    static int get_alive() {return alive;}
    static int get_created() {return created;}
protected:
    Counted() {++created; ++alive;}
    Counted(const Counted&) {++created; ++alive;}
    ~Counted() {--alive;}
private:
    static int created;
    static int alive;
};

template <typename Derived>
int Counted<Derived>::created {0};

template <typename Derived>
int Counted<Derived>::alive {0};
```

- The variables are **static**: one variable per *class* (not per object).
- This is a *class template*: a new Counted<T> *class* will be instantiated for each subclass
- Each subclass will have its own counters

The Curiously Recurring Template Pattern
Example: counting instances

Concrete subclass and helper function

```
class Foo  :public Counted<Foo>
{
public:
    Foo(int i) :x(i) {}
private:
    int x;

};

template <typename T>
void print_counts()
{
  cout << typeid(T).name() << " alive: " << T::get_alive()
       << ", created: " << T::get_created() << endl;
}
```

Suggested reading

References to sections in Lippman

Function templates 16.1.1

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

Next lecture

More about resource management, classes and the standard library.

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