Object-oriented scientific programming with C++

Matthias Möller, Jonas Thies, Cálin Georgescu, Jingya Li (Numerical Analysis, DIAM)
Lecture 4

Overview

Last lecture we started with **template meta programming**

- Implement type-independent functionality
    - Class templates/function templates
    - Generic attributes being able to hold arbitrary data type
    - Generic member function realizing the default behaviour
- Implement specialized variants of member functions to support special behaviour, e.g., dot product for complex types
- Instantiate class with concrete types (double, float, etc.)

Instantiate class with concrete types (double, float, etc.)

C++ allows you to **partially specialize** class templates

In [ ]:

```
template<typename S>
std::complex<S> Vector<std::complex<S> >::dot(const Vector<std::complex<S> > other) const
{
    std::complex<S> d=0;
    for (auto i=0; i<n; i++)
        d += data[i]*std::conj(other.data[i]);
    return d;
}
```

Note that this code will not compile. We will see why and learn remedies. Welcome to where template magic begins!

Overview

Today, **advanced template meta programming**

- Full template specialization of complete classes
- Full template specialization of individual member functions
- Partial template specialization of class templates
- Type traits
- SFINAE paradigm

Template specialization

Type-independent **default implementation** This implementation is used whenever there is no (partial) specialization of the `struct Demo` and/or its functions

In [ ]:

```
#include <iostream>
#include <complex>

template<typename T, typename I>
struct Demo
{
    static void info() {
        std::cout << "Generic info" << "\n";
    }

    static void test() {
        std::cout << "Generic test" << "\n";
    }
};

Demo<int, double>::info();               // Outputs: Generic info
Demo<std::complex<float>, char>::test(); // Outputs: Generic test
```

Go here for visualization: **https://www.online-ide.com/GWnUgMLJ3N**

Class template specialization

Task: implement a template specialization of the entire `struct Demo` for `T=float` and `I=long`
Note that template specialization does not imply class inheritance; that is, all attributes/functions that you want to have in a specialized class have to be implemented
Think of class specialization as implementing a new independent `struct Demo<float, long>` thatjusthasthe same name as the generic `struct Demo<T,I>`

Fully specialized implementation of the entire **structure**

In [ ]:

```
template<>
struct Demo<float, long>
{
    static void info() {
        std::cout << "Fully specialized info" << std::endl;
    }

    static void test() {
        std::cout << "Fully specialized test" << std::endl;
    }
};

Demo<float, long>::info(); // Outputs: Fully specialized info
Demo<float, long>::test(); // Outputs: Fully specialized test
```

Fully specialized implementation of the entire **structure** but without a member function test()
Note: run here **https://www.online-ide.com/1xpEOkqRAr**

In [2]:

```
template<typename T, typename I>
struct Demo
{
    static void info() {
        std::cout << "Generic info" << std::endl;
    }

    static void test() {
        std::cout << "Generic test" << std::endl;
    }
};

// Full specialization of the Demo class for float and long
template<>
struct Demo<float, long>
{
    static void info() {
        std::cout << "Fully specialized info" << std::endl;
    }

    // Optionally, provide a specialized version of test() if needed
    // static void test() {
    //     std::cout << "Fully specialized test" << std::endl;
    // }
};

Demo<float, long>::info(); // Calls the specialized info
//Demo<float, long>::test(); // Calls the specialized test (if defined)
```

Class-function template specialization

Task: implement a specialization of the member function `info()` for `T=float` and `I=long`

Since we only implement a specialization for the individual function `info()`, the implementation of function `test()` from the non-specialized `struct Demo` remains available

Think of member function specialization as superseding individual member functions by specialized variants

Fully specialized implementation of **function** `info()`

```
template<>
void Demo<double, long>::info() {
    std::cout << "Fully specialised info" << std::endl; }
```

This implementation provides the specialization of function `info()` and the generic implementation of function `test()`

```
Demo<double,long>::info(); // Calls the class-function specialization
Demo<double,long>::test(); // Calls the generic implementation
```

Class template partial specialization

Task: implement a specialization of the entire `struct Demo` for `T=float` and arbitrary template parameter value `I`

Partially specialized implementation of the **structure**
Note: click here **https://www.online-ide.com/bpLsXnCKG8**

In [27]:

```
template<typename T, typename I>
struct Demo
{
    static void info() {
        std::cout << "Generic info" << std::endl;
    }

    static void test() {
        std::cout << "Generic test" << std::endl;
    }
};

// Partial specialization of the Demo class for the first parameter being double
template<typename I>
struct Demo<double, I>
{
    static void info() {
        std::cout << "Partially specialized info" << std::endl;
    }

    static void test() {
        std::cout << "Partially specialized test" << std::endl;
    }
};

Demo<double, long>::info(); // Outputs: Partially specialized info
Demo<double, int>::test();  // Outputs: Partially specialized test
Demo<float, int>::info();   // Outputs: Generic info
```

Class-function template partial specialization

Task: implement a specialization of member function `info()` for `T=float` and arbitrary template parameter value `I`

Partial function template specialization is not **possible in C++**

```
template<typename I>
void Demo<float, I>::info() {...}
```

Stay tuned, there are tricks to solve this problem

Summary template specialization

Given a templated class with member functions

- Entire class can be fully or partially specialized
- Individual member functions can be fully specialized
- Individual member functions **cannot** be partially specialized

Full/partial class specialization is like implementing a new individual class that can be accessed by the same name
Full function specialization is like superseding individual member functions by specialized variants

Quiz

Remember the specialized dot product for complex-valued vectors from the previous session, will this work?

In [ ]:

```
template<typename T> class Vector {
    T dot(const Vector<T>& other) const {...}
};

template<typename S> std::complex<S>
Vector<std::complex<S> >::dot(const Vector<std::complex<S> > other) const
{
    std::complex<S> d=0;
    for (auto i=0; i<n; i++)
        d += data[i]*std::conj(other.data[i]);
    return d;
}
```

SFINAE paradigm

C++ allows us to write **overloaded functions with different** input parameter lists, e.g.,

```
static void info() {...}
static void info(int i) {...}
```

It is, however, **not** allowed to overload functions that only differ in the type of their return parameter, e.g.,

```
static void info() {...}
static int  info() {...}
```

C++11 standard states:
If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed if written using the substituted arguments. Only invalid types and expressions in the immediate context of the function type and its template parameter types can result in a deduction failure.
SFINAE: Substitution Failure Is Not An Error

C++11 standard rephrased for our purpose:
If a template substitution leads to invalid code then the compiler must not throw an error but look for another candidate (i.e. the second templated implementation of our function); an error is just thrown if no other candidate can be found so that the function call remains unresolved

SFINAE: Substitution Failure Is Not An Error

- Write multiple implementations of the same function with
    - the **same name** and
    - the **same input parameters**
- Ensure – via template meta programming – that **exactly one** at a time results in **valid code** upon substitution of the template parameters and **all other** candidates yield **invalid expressions**

Intermezzo: Traits

Consider the `is_double` **function** which returns `true`/`false` depending on the type of the parameter passed via explicit template specialization

In [26]:

```
#include <iostream>
#include <sstream>

template<typename T>
bool is_double(T a) { return false; }

template<>
bool is_double<double>(double a) { return true; }

std::cout << std::boolalpha; // Print 'true' or 'false' for bool values

std::cout << "is_int(42): "   << is_double(42)   << std::endl;   // Prints 'true'
std::cout << "is_int(42.0): " << is_double(42.0) << std::endl;   // Prints 'true'
std::cout << "is_int('A'): "  << is_double('A')  << std::endl;   // Prints 'false'
```

NOTE: Go to this link to run **https://www.online-ide.com/wkmgjVG8yH**

Consider the templated `is_double` **structure** with specialization

In [9]:

```
#include <iostream>

template<typename T>
struct is_double
{
    const static bool value = false;
};

template<>
struct is_double<double>
{
    const static bool value = true;
};

std::cout << "is_double<int>::value: "    << is_double<int>::value    << std::endl;   // Prints 'false'
std::cout << "is_double<double>::value: " << is_double<double>::value << std::endl;   // Prints 'true'
```

NOTE: Go to this link to run **https://www.online-ide.com/0RIfmNvHFK**

Detect if a given type is `double` without passing a parameter

```
std::cout << is_double<int>::value << std::endl;
std::cout << is_double<double>::value << std::endl;
```

The `is_double` **type trait** can be used in templated functions

```
template<typename T>
void test(T a)
{
  if (is_double<T>::value)
    std::cout << "Double :" << a << std::endl;
  else
    std::cout << "Non-Double :" << a << std::endl;
}
```

The `is_double` **type trait** is evaluated at compile time in contrast to the `is_double()` **function** which (theoretically) might trigger an extra function call at run time (slow!)
A smart compiler will eliminate the if-else clause

```
void test(double a)
  {
    if (is_double<T>::value)
        std::cout << "Double :" << a << std::endl;
    else
        std::cout << "Non-Double :" << a << std::endl;
}
```

C++ brings many **type traits** via `#include <type_traits>`

Function	Description
is_class<T>	Type T is of class type
is_const<T>	Type T has const qualifier
is_floating_point<T>	Type T is floating point (float, double, long)
is_fundamental<T>	Type T is of fundamental type (int, double, ...)
is_integral<T>	Type T is of integral type (int, long int, ...)
is_pointer<T>	Type T is of pointer type

For a complete list of standard type traits look at:**http://www.cplusplus.com/reference/type_traits/**

The aforementioned C++ **standard type traits** provide

- Member constants: `value (=true/false)`
- Member types: `value_type (=bool)` and `type (=true_type/false_type)`

Member constants/types can be directly accessed

```
is_fundamental<int>::value // true
is_fundamental<int>::value_type // bool
```

Intermezzo: Types Traits

C++ provides type traits that **operate on the type**

In [17]:

```
#include <type_traits>
#include <typeinfo>
#include <iostream>

typedef std::add_const<int>::type A;
typedef std::add_const<const int>::type B;
typedef std::add_pointer<int>::type C;
typedef std::add_pointer<const int>::type D;
typedef std::add_pointer<int&>::type E;
typedef std::add_pointer<int*>::type F;
typedef std::add_pointer<int(int)>::type G;

std::cout << "Type A: " << typeid(A).name() << " (expected: const int)"   << std::endl;
std::cout << "Type B: " << typeid(B).name() << " (expected: const int)"   << std::endl;
std::cout << "Type C: " << typeid(C).name() << " (expected: int*)"        << std::endl;
std::cout << "Type D: " << typeid(D).name() << " (expected: const int*)"  << std::endl;
std::cout << "Type E: " << typeid(E).name() << " (expected: int*)"        << std::endl;
std::cout << "Type F: " << typeid(F).name() << " (expected: int**)"       << std::endl;
std::cout << "Type G: " << typeid(G).name() << " (expected: int(*)(int))" << std::endl;
```

```
Type A: i (expected: const int)
Type B: i (expected: const int)
Type C: Pi (expected: int*)
Type D: PKi (expected: const int*)
Type E: Pi (expected: int*)
Type F: PPi (expected: int**)
Type G: PFiiE (expected: int(*)(int))
```

Out[17]:

```
@0x7efd5e99dde0
```

C++ provides type traits that **operate on the type**

```
typedef remove_const<      int> A;    // int (unchanged)
typedef remove_const<const int> B;    // int

typedef remove_pointer<int  > C;      // int
typedef remove_pointer<int* > D;      // int
typedef remove_pointer<int**> E;      // int*
typedef remove_pointer<const int > F; // const int
typedef remove_pointer<const int*> G; // const int
typedef remove_pointer<int* const> H; // int
```

C++ provides type traits that **operate on two types**

Check if two types are **exactly** the same (including qualifiers)

```
bool is_same<A, B>::value

bool is_same<int,       int>::value // true
bool is_same<int, const int>::value // false
bool is_same<remove_const<      int>,
             remove_const<const int> >::value // true
```

C++ provides type traits that **operate on two types**
Check if type B is derived from type A

In [19]:

```
#include <type_traits>
#include <iostream>

struct A {};
struct B : A {};

std::cout << std::boolalpha; // Print 'true' or 'false' for bool values

std::cout << "is_base_of<A, B>::value: " << std::is_base_of<A, B>::value << '\n'; // true
std::cout << "is_base_of<A, A>::value: " << std::is_base_of<A, A>::value << '\n'; // true
std::cout << "is_base_of<B, A>::value: " << std::is_base_of<B, A>::value << '\n'; // false
std::cout << "is_base_of<B, B>::value: " << std::is_base_of<B, B>::value << '\n'; // true
```

```
is_base_of<A, B>::value: true
is_base_of<A, A>::value: true
is_base_of<B, A>::value: false
is_base_of<B, B>::value: true
```

Out[19]:

```
@0x7efd5e99dde0
```

C++ provides type trait to **enable types conditionally**
If `is_odd` is called with an **integral type** (e.g., `int`) the compiler expands the following templated function as follows

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

NOTE: Go to this link to run **https://www.online-ide.com/R5hs4oQSAD**

In [ ]:

```
#include <iostream>
#include <type_traits>

template<typename T>
typename std::enable_if<std::is_integral<T>::value, bool>::type
is_odd(T i) {
    return bool(i % 2);
}


int i = 2;
std::cout << "i is odd: " << is_odd(i) << std::endl;
```

C++ provides type trait to **enable types conditionally**

```
template<typename T>
typename std::enable_if<std::is_integral<T>::value, bool>::type
is_odd(T i) {
    return bool(i%2);
}

float i=2;
std::cout << "i is odd :" << is_odd(i) << std::endl;
```

If `is_odd` is called with a **non-integral type** (e.g., `float`) the compiler expands the above templated function as follows

```
is_odd(float i) { return bool(i%2); } // compiler error
```

SFINAE revisited

SFINAE: Substitution Failure Is Not An Error

- Write multiple implementations of the same function with
    - the **same name** and
    - the **same input parameters**
- Ensure using the `enable_if` **type trait** that exactly one at a time results in **valid code** upon substitution of template parameters and **all other** candidates yield **invalid expressions**

Consider the `info()` member function

```
template<typename T, typename I>
struct Demo {
    static void info() { ... };
};
```

Enable return type `void` only in case `I=int` and let `info()` have no return type (=invalid code) if I is of any other type

```
bool v = std::is_same<I, int>::value // either true or false
std::enable_if<v, void>::type // either void or empty
```

SFINAE revisited

First attempt of partially specialized `info()` member function

```
template<typename T, typename I>
struct Demo
{
    // partial specialization for I=int
    typename std::enable_if< std::is_same<I, int>::value, void>::type
    static info() { ... };

    // partial specialization for I!=int
    typename std::enable_if<!std::is_same<I, int>::value, void>::type
    static info() { ... };
};
```

This code will **not compile**; we need to introduce an extra function template parameter for the `info()` function

Partially specialized `info()` member function (now working!)

```
template<typename T, typename I>
struct Demo
{
    template<typename J=I>
    typename std::enable_if< std::is_same<J, int>::value, void>::type
    static info() { ... };

    template<typename J=I>
    typename std::enable_if<!std::is_same<J, int>::value, void>::type
    static info() { ... };
};
```

In words...

- Introduce an **extra function template parameter** `J` that, by default, takes the value of the class template parameter `I`

```
template<typename J=I>
```

- Make type traits depend on extra template parameter `J`

```
typename std::enable_if< std::is_same<J, int>::value, void>::type
```

- Make sure that exactly one member function leads to valid code

```
typename std::enable_if<!std::is_same<J, int>::value, void>::type
```

Dot product revisited

Let us reconsider the dot product for complex-valued vectors
Use **SFINAE paradigm** to realise alternative implementations of the dot product for real- and complex-valued types
Strategy:

- Write **type trait** `is_complex<T>` that has `value=true` if `T` is of type `std::complex<U>` and `value=false` otherwise
- Use **type trait** `std::enable_if<...>` to distinguish between real- and complex-valued implementation of the dot product

Type trait is_complex

First implementation of type trait `is_complex` (will suffice for our purpose but is not really in line with standard traits)

```
template<typename T>
struct is_complex
    { static const bool value = false; };

template<>
struct is_complex<std::complex<float> >
    { static const bool value = true; };

template<>
struct is_complex<std::complex<double> >
    { static const bool value = true; };
```

C++ standard way to implement type traits is by deriving `is_complex` from structure `std::integral_constant<T, value>`

```
template<typename T>
struct is_complex
: std::integral_constant<bool,
    std::is_same<T, std::complex<float> >::value ||
    std::is_same<T, std::complex<double> >::value > {};
```

Logical combination (`&&`, `||`) of all `std::complex<S>` types that should be supported by the `is_complex` type trait

Implementation of dot product for **complex-valued types**

```
template<typename T>
class Vector {
...
    template<typename U=T>
    typename std::enable_if<is_complex<U>::value, U>::type
    dot(const Vector<T>& other) const {
      T d=0;
      for (auto i=0; i<n; i++)
        d += data[i]*std::conj(other.data[i]); return d;
    }
};
```

Implementation of dot-product for **real-valued types**

```
template<typename T>
class Vector {
...
    template<typename U=T>
    typename std::enable_if<!is_complex<U>::value, U>::type
    dot(const Vector<T>& other) const {
      T d=0;
      for (auto i=0; i<n; i++)
        d += data[i]*other.data[i];
      return d;
    }
};
```

Summary SFINAE paradigm

General approach to circumvent the limitations of C++ to not allow **partial specialization of member function templates**

- Use `std::enable_if` and `std::is_xyz` or self-written type trait to switch between different implementations of a function

Default template arguments for function templates (`template<typename J=I>`)are a new feature in C++11

For a complete list of standard type traits look at:
http://www.cplusplus.com/reference/type_traits/

SFINAE Quiz

What does this code do?

```
struct A {};
struct B : A {};
struct C {};

template<typename T>
typename auto get_base_type(T t) {
  typename std::conditional<std::is_base_of<A,T>::value, A, T>::type ReturnType;
  return ReturnType(t);
}
```

See the `get_base_type` function in action

```
A a; B b; C c;
typeid(a).name() // -> 1A
typeid(b).name() // -> 1B
typeid(c).name() // -> 1C

typeid(get_base_type(a)).name() // -> 1A
typeid(get_base_type(b)).name() // -> 1A
typeid(get_base_type(c)).name() // -> 1C
```

Final word on SFINAE

Recall that we started the SFINAE-journey since we needed partial specialization of the dot-product member function

It is also possible to **specialize the conj-function** instead

How would you implement the function `std::conj(...)`?

- What return type should we expect for real-valued data?
- What return type should we expect for complex-valued data?

A possible implementation of the function `std::conj(...)` that uses the self-written `is_complex` type trait

```
template<typename T>
typename std::enable_if<is_complex<T>::value, T>::type static conj(T t)
{ return T(t.real(), -t.imag()); }

template<typename T>
typename std::enable_if<!is_complex<T>::value, T>::type static conj(T t)
{ return T(t); }
```

In [ ]:

Loading [MathJax]/extensions/Safe.js