TW3720TU: Object Oriented Scientific Programming with C++ (11/14/17)

Matthias Möller

Numerical Analysis

TUDelft

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 specialised variants of member functions to support special behaviour, e.g., dot product for complex types
  - Instantiate class with concrete types (double, float, etc.)

Overview, cont'd

- Today, advanced template meta programming
  - Full template specialisation of complete classes
  - Full template specialisation of individual member functions
  - Partial template specialisation of class templates
  - Type traits
  - SFINAE paradigm

TUDelft

Template specialisation

- Type-independent implementation

```
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; }
};
```

- This implementation is used whenever there is no (partial) specialisation of the `struct Demo` and/or its functions

Class template specialisation

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

TU Delft

Class template specialisation, cont‘d

- Fully specialised implementation of the entire `struct Demo`

```
template<>
struct Demo<float,long>
{
  static void info() {
    std::cout << „Fully specialised info“ << std::endl; }
  static void test() {
    std::cout << „Fully specialised test“ << std::endl; }
};
```

- This implementation is used for the special case

```
Demo<float,long>::info() -> class specialisation
Demo<float,long>::test() -> class specialisation
```

TU Delft

Class template specialisation, cont‘d

- *Fully specialised* implementation of the entire `struct Demo` but without a member function `test()`

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

- This implementation yields a compiler error

```
Demo<float,long>::info() -> class specialisation
Demo<float,long>::test() // compiler error
```

Class-function template specialisation

- **Task:** implement a specialisation of the member function `info()` for `T=float` and `I=long`
- Since we only implement a specialisation for the individual function `info()`, the implementation of function `test()` from the non-specialised `struct Demo` remains available
- Think of member function specialisation as superseding individual member functions by specialised variants

TU Delft

Class-function template specialisation, cont‘d

- *Fully specialised* implementation of function `info()`

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

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

```
Demo<double,long>::info() -> class-function specialisation
Demo<double,long>::test() -> generic
```

TUDelft

Class template partial specialisation

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

Class template partial specialisation, cont‘d

- *Partially specialised* implementation of the `struct Demo`

```
template<typename I>
struct Demo<double, I>
{
  static void info() {
    std::cout << „Partially specialised info“ << std::endl; }
  static void test() {
    std::cout << „Partially specialised test“ << std::endl; }
};
```

- This implementation is used for the special case

```
Demo<double,int>::info() -> partial class specialisation
Demo<double,int>::test() -> partial class specialisation
```

Class-function template partial specialisation

- **Task:** implement a specialisation of member function `info()` for `T=float` and arbitrary template parameter value `I`
- Partial function template specialisation is **not possible**

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

- Stay tuned, there are tricks to solve this problem

Summary template specialisation

- Given a templated class with member functions
  - Entire class can be fully or partially specialised
  - Individual member functions can fully specialised
  - Individual member functions **cannot** be partially specialised
- Full/partial class specialisation is like implementing a new individual class that can be accessed by the same name
- Full function specialisation is like superseding individual member functions by specialised variants

Quiz

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

```
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;
    }
```

TUDelft

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() {...}
```

TUDelft

SFINAE paradigm, cont‘d

- 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

SFINAE paradigm, cont‘d

- 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 paradigm, cont‘d

- **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_int **function** from the previous assignment

```
template<typename T>
bool is_int(T a) { return false; }
template<>
bool is_int<int>(int a) { return true; }
```

- This function returns true/false depending on the type of the parameter passed via explicit template specialisation
- We look for an even more elegant solution without the need to call a function and pass a parameter at all

TUDelft

Intermezzo: Traits, cont‘d

- Consider templated **structure** with specialisation

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

template<>
struct is_int<int>
{
    const static bool value = true;
};
```

Intermezzo: Traits, cont‘d

- Detect if a given type is `int` without passing a parameter

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

- The `is_int` **trait** can be used, e.g., in templated functions

```
template<typename T>
void test(T a)
{
  if (is_int<T>::value)
    std::cout << „Integer :“ << a << std::endl;
  else
    std::cout << „Non-Int :“ << a << std::endl;
}
```

Intermezzo: Traits, cont‘d

- The `is_int` **trait** is evaluated at compile time in contrast to the `is_int()` **function** which (at least theoretically) might trigger a function call at run time
- A smart compiler will eliminate the if-else clause

```
void test(int a)
{
  if (is_int<T>::value)
    std::cout << „Integer :“ << a << std::endl;
  else
    std::cout << „Non-Int :“ << a << std::endl;
}
```

Intermezzo: Type traits

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

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/

Intermezzo: Type traits, cont'd

- The aforementioned C++ standard type traits provide
  - Member constants:
    ```
    value      (=true/false)
    ```
  - Member types:
    ```
    value_type (=bool)
    type       (=true_type/false_type)
    ```
- Member constants/types can be directly accessed

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

Intermezzo: Type traits, cont'd

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

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

typedef add_pointer<int>           C // int*
typedef add_pointer<const int>     D // const int*
typedef add_pointer<int&>          E // int*
typedef add_pointer<int*>          F // int**
typedef add_pointer<int(int)>      G // int(*)int
```

TUDelft

Intermezzo: Type traits, cont'd

- 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
```

Intermezzo: Type traits, cont‘d

- 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
```

TUDelft

Intermezzo: Type traits, cont‘d

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

```
struct A {};
struct B : A {};
bool is_base_of<A,B>::value

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

TUDelft

Intermezzo: Type traits, cont'd

- 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); }

int i=2;
cout << „i is odd :“ << is_odd(i) << endl;
```

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

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

TUDelft

Intermezzo: Type traits, cont‘d

- 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;
cout << „i is odd :“ << is_odd(i) << 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
```

TUDelft

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**

TUDelft

SFINAE revisited, cont'd

- 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, cont'd

- First attempt of partially specialised `info()` member function

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

SFINAE revisited, cont‘d

- Partially specialised `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() { ... };
};
```

SFINAE revisited, cont'd

- In words…
  - Introduce function template parameter that by default takes the value of the class template parameter (`template<typename J=I>`)
  - Make type traits depend on artificial template parameter

```
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
```

TUDelft

SFINAE revisited, cont'd

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

Type trait is_complex, cont‘d

- 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 or (||) combination of all `std::complex<S>` that should be supported by the `is_complex` type trait

Type trait is_complex, cont‘d

- 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;
  }
};
```

Type trait is_complex, cont‘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 specialisation of member function templates**
  - Use `std::enable_if` and `std::is_XXX` or self-written type trait to switch between different implementations of a function
- Code gets less readable due to dummy function template
- 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/

TU Delft

SFINAE Quiz

- What does this code do?

```
struct A {
  A() {}
  A(const A& a) {}
};
struct B : A {
  B() {}
  B(const B& b) {}
};
struct C {
  C() {}
  C(const C& c) {}
};
```

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

SFINAE Quiz, cont‘d

- See `get_base_type` 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
```

TUDelft

Final word on SFINAE

- Recall that we started the SFINAE-journey since we needed partial specialisation of the dot-product member function
- It is also possible to specialise 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?

TUDelft

Final word on SFINAE, cont'd

- Viable 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); }
```

TUDelft