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

Matthias Möller

Numerical Analysis

TUDelft

Overview

- Last lecture we started with polymorphism, that is, inheritance of one class from another class
  - Implement common functionality in base class (possibly realised as abstract class that cannot even be instantiated)
  - Derive specialised class(es) from the base class that
    - Implement the missing functionality (pure virtual functions)
    - Override generic functionality by specialised variants (virtual functions)
    - Reuse all other functionality from the base class

Overview, cont‘d

- Today, a more careful view on polymorphism
  - *Static* polymorphism: static binding/method overloading
  - *Dynamic* polymorphism: dynamic binding/method overriding
- C++11/14/17? auto functionality
- Template meta programming
  - A powerful variant of static polymorphism

Quiz: Guess what happens

- ```
  struct Base {
    virtual void hello() = 0;
  };
  ```
- ```
  struct Derived : public Base {
    void hallo()
    {
      std::cout << “Hallo\n”;
    }
  };
  ```

Quiz: Guess what happens

- ```
  struct Base {
    virtual void hello() = 0;
  };
  ```
- ```
  struct Derived : public Base {
    void hello()
    {
      std::cout << “Hallo\n”;
    }
  };
  ```

Quiz: Guess what happens

- ```
  struct Base {
    virtual void hello() {
      std::cout << “Hello\n”;
    }
  };
  ```
- ```
  struct Derived : public Base {
    void hello()
    {
      std::cout << “Hallo\n”;
    }
  };
  ```

Quiz: Guess what happens

- ```
  struct Base {
    virtual void hello() {
      std::cout << “Hello\n”;
    }
  };
  ```
- ```
  struct Derived : public Base {
    void hallo()
    {
      std::cout << “Hallo\n”;
    }
  };
  ```

The override keyword

- C++11 introduces the `override` keyword to explicitly state that a function from the base class shall be overridden

```
struct Base {
  virtual void hello() { std::cout << “Hello\n”; }
};
struct Derived : public Base {
  void hello() override { std::cout << “Hallo\n”; }
};
```

- If the function to be overridden does not exist in the base class an error is thrown by the compiler

TUDelft

The `override` keyword, cont'd

- Overriding function must exactly match the signature of the function from the base class

```
struct Base {
  virtual void hello() const { std::cout << "Hello\n"; }
};
struct Derived : public Base {
  void hello() const override { std::cout << "Hallo\n"; }
};
```

The `final` keyword

- C++11 introduces the `final` keyword to explicitly state that a function from the base class **must not** be overridden

```
struct Base {
  virtual void hello() final { std::cout << "Hello\n"; }
};
```

- If a derived class tries to override the function `hello` an error is thrown by the compiler

The `override` and `final` keywords

- The `override` and `final` keywords are often used together

```
struct Base {
  // Pure virtual function must be overridden in derived class
  virtual void hello() = 0;
};
struct Derived : public Base {
  // Override pure virtual function and make it final
  virtual void hello() override final
  { std::cout << “Hello\n”; }
};
struct Derived2 : public Derived {
  // Must not try to override function hello
};
```

Task: Calculator

- Write a class (or for demonstrating purposes a hierarchy of classes) that provide(s) a member function to calculate the **sum of two and three integer values**, respectively
  - Use *static* polymorphism: method overloading
  - Use *dynamic* polymorphism: method overriding

Static polymorphism

- **Method overloading** (at compile time)

```
struct Calc {
  int sum(int a, int b) { return a+b; }
  int sum(int a, int b, int c) { return sum(sum(a,b),c); }
};
```

- Class `Calc` has two member functions with identical names but different interface; it is decided **at compile time** which of the two functions should be called

```
std::cout << C.sum(1,2)   << std::endl;
std::cout << C.sum(1,2,3) << std::endl;
```

Static polymorphism, cont‘d

- **Method overloading** (not working!)

```
struct Calc {
  int sum(int a, int b) { return a+b; }
  void sum(int a, int b) { std::cout<< a+b <<std::endl; }
};
```

- Difference must be in the **interface of arguments passed** to functions since the compiler cannot distinguish between two functions if they only differ in the **return type**

Static polymorphism, cont'd

- **Method overloading**: decision about which method to call is made at compile time; hence the compiler can decide to inline code to improve performance (no overhead due to function calls/copy of data to and from the stack!)

```
std::cout << C.sum(1,2)   << std::endl;
std::cout << C.sum(1,2,3) << std::endl;
```

becomes

```
std::cout << (1+2)        << std::endl;
std::cout << ((1+2)+3)    << std::endl;
```

Static polymorphism, cont'd

- **Method overloading**: since static polymorphism takes place at compile time, the `inline` specifier can be used to explicitly 'suggest' to the compiler to inline the function

```
struct Calc {
  inline int sum(int a, int b)
    { return a+b; }
  inline void sum(int a, int b)
    { std::cout<< a+b <<std::endl; }
};
```

Dynamic polymorphism

- **Method overriding**: re-implement a function inherited from base class with new function body (same interface!)

```
struct BaseCalc {
  virtual int sum2(int a, int b) { return a+b; }
  int sum3(int a, int b, int c)
      { return sum2(sum2(a,b),c); }
};

struct DerivedCalc : public BaseCalc {
  int sum2(int a, int b) override { return a+b; }
};
```

Dynamic polymorphism, cont'd

```
DerivedCalc D;
std::cout << D.sum2(1,2) << std::endl;
->  DerivedCalc::sum2(a,b)
->  3
std::cout << D.sum3(1,2,3) << std::endl;
->  BaseCalc::sum3(a,b,c)
->  DerivedCalc::sum2(a,b)
->  DerivedCalc::sum2(a,b)
->  6
```

TUDelft

Dynamic polymorphism, cont‘d

- **Method overriding**: a common pitfall is to forget the `virtual` specifier to indicates that the `sum2` function from the base class might be overridden in a derived class

```
struct BaseCalc {
  int sum2(int a, int b) { return a+b; }
  int sum3(int a, int b, int c)
      { return sum2(sum2(a,b),c); }
};
struct DerivedCalc : public BaseCalc {
  int sum2(int a, int b) { return a+b; }
};
```

TU Delft

Dynamic polymorphism, cont‘d

- These bugs are hard to find (they often remain unrecognised) and can be prevented by using the `override` keyword in C++11

```
DerivedCalc D;
std::cout << D.sum2(1,2) << std::endl;
->  DerivedCalc::sum2(a,b)
->  3
std::cout << D.sum3(1,2,3) << std::endl;
->  BaseCalc::sum3(a,b,c)
->  BaseCalc::sum2(a,b)
->  BaseCalc::sum2(a,b)
->  6
```

Dynamic polymorphism, cont‘d

- **Method overriding**: decision about which virtual method to call is made at run time; hence inlining is not possible
- Common design pattern
  - Specify expected minimal functionality of a group of classes in abstract base class via pure virtual member functions
  - Implement generic common functionality of a group of classes abstract base class via virtual member functions
  - Implement expected functionality of a particular class by overriding the pure virtual member function

TUDelft

Example: inner product space

- In linear algebra, an inner product space is a vector space V that is equipped with a special mapping (inner product)

$$\langle \cdot , \cdot \rangle : V \times V \to \mathbb{R} \text{ or } \mathbb{C}$$

- Inner product spaces have a naturally induced norm

$$\| x \| = \sqrt{\langle x , x \rangle}$$

TUDelft

Example: inner product space, cont'd

- Class `InnerProductSpaceBase` declares inner product as pure virtual and implements the naturally induced norm

```
struct InnerProductBase
{
  pure virtual double inner_product(… x,… y) = 0;
  double norm(x) { return inner_product(x,x); }
};
```

- Derived `InnerProductSpace` class implements inner product

```
struct InnerProductSpace : public InnerProductSpaceBase
{
  double inner_product(… x, … y) = { return x*y; }
};
```

Task: Calculator2

- Extend the calculator class so that it can handle numbers of integer, float and double type at the same time
  - Prevent manual code duplication
  - Prevent explicit type casting
  - Make use of auto-functionality (C++11/14/17?)
  - Make use of template meta programming

TU Delft

Vanilla implementation in C++

- ```
  struct Calc2 {
    int sum(int a, int b)
        { return a+b; }
    int sum(int a, int b, int c)
        { return sum(sum(a,b),c); }
  };

  int main() {
    Calc2 C;
    std::cout << C.sum(1,2)   << std::endl;
    std::cout << C.sum(1,2,3) << std::endl;
  }
  ```
  ```

Automatic return type deduction (C++11)

- Explicit definition of the function return type

```
int sum(int a, int b)
    { return a+b; }
```

- Automatic function return type (since C++11)

```
auto sum(int a, int b) -> decltype(a+b)
    { return a+b; }
```

- Using `decltype`, the return type of the sum function is determined automatically as the type of `operator+(a,b)`

Automatic return type deduction, cont‘d

- decltype specifier (C++11) queries the type of an expression

```
struct Calc2 {
  auto sum(int a, int b)          -> decltype(a+b)
      { return a+b; }
  auto sum(int a, int b, int c)  -> decltype(sum(sum(a,b),c))
      { return sum(sum(a,b),c); }
};

int main() {
  Calc2 C;
  std::cout << C.sum(1,2)   << std::endl;
  std::cout << C.sum(1,2,3) << std::endl;
}
```

TUDelft

Automatic type deduction (C++14)

- C++14 deduces the type of parameters automatically

```
auto sum(int a, int b)          // no -> decltype(a+b)
    { return a+b; }
auto sum(int a, int b, int c) // no -> decltype(a+b+c)
    { return sum(sum(a,b),c); }
```

- Remark: Feature helps to improve readability and prevents deduction errors (due to forgotten/inconsistent deduction rule by the programmer) but it does not solve the problem to pass arguments of different type to the same function

TUDelft

Generic functions (C++17)

- C++17 standard will support generic functions

```
auto sum(auto a, auto b)
    { return a+b; }
auto sum(auto a, auto b, auto c)
    { return sum(sum(a,b),c); }
```

- Remark: code compiles with GNU g++ 5.x and better

Function templates

- Until C++17, template meta programming is the standard technique to deal with arbitrary function parameters
- **Function templates**: allow you to implement so-called parameterised functions for generic parameter types

```
template<typename R, typename A, typename B>
R sum(A a, B b)
{
    return a+b;
}
```

TUDelft

Function templates

- Types must be specified explicitly when function is called

```
int    s1 = sum<int, int, int>(1, 2);
double s2 = sum<double, double, int>(1.2, 2);
double s3 = sum<double, float, double>(1.4, 2.2);
```

- This can be slightly simplified using the auto specifier

```
auto s1 = sum<int, int, int>(1, 2);
auto s2 = sum<double, double, int>(1.2, 2);
auto s3 = sum<double, float, double>(1.4, 2.2);
```

TUDelft

Function templates, cont'd

- C++11: automatic return type deduction

```
template<typename A, typename B>
auto sum(A a, B b) -> decltype(a+b)
    { return a+b; }
```

- C++14: automatic type deduction

```
template<typename A, typename B>
auto sum(A a, B b)
    { return a+b; }
```

- Usage

```
auto s1 = sum<int, int>(1, 2);
```

Function templates, cont'd

- How to convert this function into a templated function

```
int sum(int a, int b, int c)
{
    return sum(sum(a,b), c);
}
```

Function templates, cont'd

- Use explicit return type parameter (ugly!)

```
template<typename R, typename A, typename B, typename C>
auto sum(A a, B b, C c)
{
    return sum<R,C>(sum<A,B>(a,b), c);
}
```

- Guess what this function call will return

```
auto s1 = sum<int,double,double,double>(1.1,2.2,3.3)
```

TUDelft

Function templates, cont'd

- Use a smart combination of templates and auto

```
template<typename A, typename B>
auto sum(A a, B b) -> decltype(a+b) // omit in C++14
{
    return a+b;

template<typename A, typename B, typename C>
auto sum(A a, B b, C c)
{
    return sum<decltype(sum<A,B>(a,b)),C>
                (sum<A,B>(a,b), c);
}
```

Function templates, cont'd

- Now, we can call sum functions as follows

```
auto s1 = sum<int, int>(1, 2);
auto s2 = sum<double, int>(1.2, 2);
auto s3 = sum<float, double>(1.4, 2.2);
```

- Since the compiler needs to duplicate code and substitute A,B,C for each combination of templated types both the compile time and the size of the executable will increase
- Template meta programming is simplest of the code resides in header files only; later we will see how to use template meta-programming together with pre-compiled libraries

TUDelft

Task: generic Vector class

- Write Vector class that can store real values (float/double) and complex values and supports the following operations:
  - Addition of two vectors of same length (possibly different type)
  - Multiplication of a vector with a scalar (possible different type)
  - Dot product of a vector with another one (possibly different type)

Vector class prototype

- Implementation of Vector-of-double class

```
class Vector {
private:
    double* data;
    int n;
public:
    Vector()      : n(0), data(nullptr)      {}
    Vector(int n) : n(n), data(new double[n]) {}
    ~Vector()     { n=0;  delete[] data; }
    ...
```

Vector class prototype, cont'd

```
    Vector& operator+=(const Vector& other) {
        for (auto i=0; i<n; i++)
            data[i] += other.data[i];
        return *this; }
    Vector& operator*=(double scalar) {
        for (auto i=0; i<n; i++)
            data[i] *= scalar;
        return *this; }
    double dot(const Vector& other) const {
        double d=0;
        for (auto i=0; i<n; i++)
            d += data[i]*other.data[i];
        return d; }
};
```

Brainstorming

- Function templates alone will not help since the type of a class attribute needs to be templated -> **class templates**
- Some member functions can be implemented generically, e.g., addition of two vectors and multiplication of a vector with a scalar value since they are the same for all types
- Some member functions must be implemented in different manners for real and complex values -> **specialisation**

$$x \cdot y = \sum_{i=1}^{n} x_i y_i, \quad x, y \in \mathbb{R}, \qquad x \cdot y = \sum_{i=1}^{n} x_i \bar{y}_i \quad x, y \in \mathbb{C}$$

Class template

- Implementation of Vector-of-anything class

```
template<typename T>
class Vector {
private:
    T* data;
    int n;
public:
    Vector()      : n(0), data(nullptr)  {}
    Vector(int n) : n(n), data(new T[n]) {}
    ~Vector()     { n=0;  delete[] data; }
    ...
};
```

Class template, cont'd

- Template parameter must be explicitly specified

```
Vector<int>    x(10); // Vector-of-int with length 10
Vector<double> y;     // Empty Vector-of-double
Vector<float>  z(5);  // Vector-of-float with length 5
```

- Remark: if you want to pass a Vector-of-anything to a function in the templated class Vector you have to write

```
Vector<T>  v
Vector<T>& v
```

instead of

```
Vector  v
Vector& v
```

Class template, cont‘d

```
template <typename T>
class Vector {
public:
    ...
    Vector<T>& operator+=(const Vector<T>& other)
    {
        (for auto i=0; i<n; i++)
            data[i] += other.data[i];
        return *this;
    }
    ...
};
```

Class template, cont‘d

```
template <typename T>
class Vector {
public:
    ...
    Vector<T>& operator*=(T scalar)
    {
        (for auto i=0; i<n; i++)
            data[i] *= scalar;
        return *this;
    }
    ...
};
```

Class template, cont‘d

```
template <typename T>
class Vector {
public:
    ...
    T dot(const Vector<T>& other) const
    {
        T d=0;
        for (auto i=0; i<n; i++)
            d += data[i]*other.data[i];
        return d;
    }
    ...
};
```

TUDelft

Class template, cont'd

- With the single class template parameter `T` we can do

```
Vector<int> x1(5), x2(5);
x1 += x2;
x1 *= (int)2;
int x1 = x1.dot(x2);
```

- How about?

```
Vector<int> x2(5);
x2 *= (double)1.2;
```

Intermezzo

- Class templates and function templates can be combined

```
template<typename T>
class Vector {
    ...
    template<typename S>
    Vector<T>& multiply<S scalar>
    {
        (auto i=0; i<n; i++)
            data[i] *= S;
        return *this;
    }
};
```

Intermezzo, cont‘d

- At first glance, this seems to be more flexible

```
Vector<double> x1(5);
x1.multiply<int>(5);
```

- But be really careful since strange things can happen

```
Vector<int> x1(5);
x1.multiply<double>(5.5);
```

- Rule of thumb: before using extensive templating like this think about all(!) implications; it can help to think if the planned functionality has a meaningful mathematical counterpart, e.g., dot product of $x \in \mathbb{R}, y \in \mathbb{C}$

TU Delft

Specialisation

- The dot product needs special treatment since

```
T dot(const Vector<T>& other) const
    {
        T d=0;
        for (auto i=0; i<n; i++)
            d += data[i]*other.data[i];
        return d;
    }
```

  - Lacks the complex conjugate of other and yields the wrong return type in case of complex-valued vectors

- Remedy: implement a specialised variant for this case

TU Delft

Specialisation, cont‘d

- Generic dot product implemented in Vector class

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

- This function is used whenever no specialised implementation for a concrete type is available

TUDelft

Specialisation, cont'd

- Specialised dot product for Vectors-of-complex-float

```
template<>
std::complex<float> Vector< std::complex<float> >::
    dot(const Vector< std::complex<float> >& other) const
    {
        std::complex<float> d=0;
        // special treatment of dot product
        for (auto i=0; i<n; i++)
            d += data[i]*std::conj(other.data[i]);
        return d;
    }
```

Specialisation, cont'd

- Current implementation yields

```
Vector<float> x1(5), x2(5);
auto x1.dot(x2); // calls generic implementation

Vector<std::complex<float> > y1(5), y2(5);
auto y1.dot(y2); // calls specialised implementation

Vector<std::complex<double> > z1(5), z2(5);
auto z1.dot(z2); // calls generic implementation

auto x1.dot(y1); // does not compile(!)
```

Outlook on next session

- C++ allows you to **partially specialise** class templates
  Note that this code will not compile (we will see why!)

```
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;
    }
```

- Welcome to where the magic begins!