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

Matthias Möller

Numerical Analysis

TU Delft

Overview

- Last lecture we started with object-oriented programming:
  - Classes, structures, and attributes
  - Constructors/Destructors, and member functions
- This lecture we investigate the many details C++ does automatically to simplify (and sometimes complicate) OOP
- The second part of the lecture deals with polymorphism, that is, inheritance of one class from another class

Container class

```
class Container {
public:
    Container(int length)
    : length(length), data(new double[length])
    { }

    Container(std::initializer_list<double> l)
    : Container( (int)l.size() )
    {
        std::uninitialized_copy(l.begin(), l.end(), data);
    }
    ...
};
```

TUDelft

Container class, cont‘d

```
class Container {
public:
    ...
    ~Container()
    {
        delete[] data;
        length=0;
    }

private:
    double* data;
    int length;
};
```

Container class, cont‘d

- Class has **no default constructor** (at least we think so)
```
Container() { }
```
- Class has member function for printing
```
void info()
{
    std::cout << „Length:  “ << length << std::endl;
    std::cout << „Pointer: “ << data   << std::endl;
    std::cout << „Address: “ << &data  << std::endl;
}
```

TUDelft

Converting constructor

- Both constructors can be called in two ways
  - In the **explicit constructor** declaration

```
Container a( 4 );
Container a( {1,2,3,4} );
```

  - Using **copy initialisation**

```
Container a = 4;            // -> Container a(4)
Container a = { 1,2,3,4}; // -> Container a({1,2,3,4})
```

TUDelft

Converting constructors

- Constructors with a single parameter are called **converting constructor** since they specify an implicit conversion
  - from the types of their arguments
    - `Container(int length) {...}`
    - `Container(std::initializer_list<double> l) {...}`
  - to the types of their class

```
class Container {
private:
    double* data;
    int length;
};
```

Explicit specifier

- The **explicit specifier** prevents the use of a single non-default parameter constructor as conversion constructor

```
class Container {
public:
    explicit Container(int length)
    : length(length), data(new double[length]);
    { }
};
```

- Copy-initialisation (`Container a = 3;`) is no longer possible but explicit constructor (`Container a(3);`) has to be used

TUDelft

Explicit specifier, cont‘d

- Use of copy-initialisation constructor yields (Clang):

```
src/copy-move.cxx:73:15: error: no viable conversion from
'int' to 'Container‘
    Container d = 3;
              ^   ~

...
make: *** [all] Error 2
```

- Consequent use of explicit specifier can help to get better control over unrecognized implicit creation of objects, e.g., in user-defined assignments (later in this lecture).

TU Delft

Special member functions

- In C++ the following special member functions are created **automatically** if they are used but not declared explicitly
  - Default constructor
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor

Special member functions, cont‘d

- Once you implement one of the constructors (default, copy, move) or assignment operators (copy, move) explicitly, auto-generation of all others is disabled
- Our class provides only the constructors
  - `Container(int length)`
  - `Container(std::initializer_list<double> l)`

  Copy and move constructors are therefore auto-generated

Special member functions, cont‘d

- The following code compiles:

```
Container a({1,2,3,4}); // implemented constructor
Container b(a);         // auto-generated copy constructor
a.info();
b.info();
```

- What will happen when we execute the program?
  - All implemented constructors/destructors provide debug output

    ```
    std::cout << “Constructor/destructor called\n“;
    ```

    when they are called (explicitly or implicitly!)

TU Delft

Special member functions, cont'd

- Execution of the program:

```
Container a({1,2,3,4,});
  >>> Constructor(int length) called
  >>> Constructor(std::initializer_list<double> list) called



Container b(a);
   // no log since copy constructor is auto-generated



  >>> Destructor called
```

Special member functions, cont‘d

- Execution fails when object b is deallocated:

```
Container a({1,2,3,4,});
  >>> Constructor(int length) called
  >>> Constructor(std::initializer_list<double> list) called
  Length of data pointer:   4
  Address of data pointer: 0x7fff5947a4c0
  Data pointer:            0x7fa5a94031a0
Container b(a);
  Length of data pointer:   4
  Address of data pointer: 0x7fff5947a4a0
  Data pointer:            0x7fa5a94031a0
  >>> Destructor called
  copy-move(77163,0x7fff73acf000) malloc:
    *** error for object 0x1: pointer being freed was not allocated
    *** set a breakpoint in malloc_error_break to debug. Abort trap: 6
```

Same pointer

Copy constructor

- Auto-generated („soft“) copy constructor just copies the pointer of a.data and, upon deallocation, tries to deallocate the same pointer for a.data and b.data twice

```
Data pointer: 0x7fa5a94031a0
Data pointer: 0x7fa5a94031a0
```

- Solution
  - Disable auto-generated soft-copy constructor
  - Implement explicit **deep-copy** constructor

TUDelft

Copy constructor, cont'd

- Disable auto-generation of soft-copy constructor
  `Container(const Container& c) = delete;`
- Compiler now fails to build the program

```
src/copy-move.cxx:99:15: error: call to deleted
constructor of 'Container'
     Container e(a);
               ^ ~
src/copy-move.cxx:46:5: note: 'Container' has been
explicitly marked deleted here
     Container(const Container& c) = delete;
     ^
```

Copy constructor, cont'd

- Implement **deep-copy constructor** explicitly

```
Container(const Container& c)
: Container(c.length)
{
    for (auto i=0; i<c.length; i++)
        data[i] = c.data[i];
}
```

- Now, the copy constructor (`Container b(a);`) does a **deep copy** (=duplication of the data array) of the content of object a when creating object b

TUDelft

Intermezzo

```
class Container {
    // no default constructor
    explicit Container(int length) ...

    // unified initialization constructor
    explicit Container(std::initializer_list<double l) ...

    // deep-copy constructor
    explicit Container(const Container& c) ...

    // destructor
    ~Container() ...
}
```

Intermezzo, cont‘d

```
>>> Constructor(int length) called
>>> Constructor(std::initializer_list<double> list) called
>>> Constructor(int length) called
>>> Copy constructor
Length of data pointer:   4
Address of data pointer: 0x7fff52d954c0
Data pointer:            0x7fae4b4031a0
Length of data pointer:   4
Address of data pointer: 0x7fff52d954a0
Data pointer:            0x7fae4b4031c0
>>> Destructor called
>>> Destructor called
```

Different pointers

Move constructor

- Sometimes, the argument passed to the constructor should be **fully consumed** (and not deep-copied) into new object:

```
Container(Container&& c)
: length(c.length), data(c.data)
{
    c.length = 0;
    c.data = nullptr;
}
```

- To use the **move constructor** it must be called as follows

```
Container b(std::move(a));
```

TUDelft

Move constructor, cont'd

- `std::move(a)` forces to move the content from object a, otherwise the **copy constructor** would be called
  - ```
    Container a({1,2,3,4});
    Container b(a);      <- No explicit move
    >>> Constructor(int length) called
    >>> Constructor(std::initializer_list<double> list) called
    >>> Constructor(int length) called
    >>> Copy constructor <- Copy constructor
    Length of data pointer:  4
    Address of data pointer: 0x7fff52d954c0
    Data pointer:            0x7fae4b4031a0
    Length of data pointer:  4
    Address of data pointer: 0x7fff52d954a0
    Data pointer:            0x7fae4b4031c0
    >>> Destructor called
    >>> Destructor called
    ```
    ```

Move constructor, cont'd

- `std::move(a)` explicitly tells C++ to move the content from object a, otherwise the copy constructor would be called
  - ```
    Container a({1,2,3,4});
    Container b(std::move(a));             <- Explicit move
    >>> Constructor(int length) called
    >>> Constructor(std::initializer_list<double> list) called
    >>> Constructor(int length) called
    >>> Move constructor                   <- Move constructor
    Length of data pointer:  0             <- object a has length 0
    Address of data pointer: 0x7fff52d954c0
    Data pointer:            0x0           <- object a has lost its pointer
    Length of data pointer:  4
    Address of data pointer: 0x7fff52d954a0
    Data pointer:            0x7fae4b4031c0
    >>> Destructor called
    >>> Destructor called
    ```

TUDelft
    ```

Assignment operators

- As with constructors there exist
  - Copy assignment operator (=deep copy)
  - Move assignment operator (=soft copy)

Copy assignment operator

- ```
  Container& operator=(const Container& other)
  {
      if (this != &other)
      {
          delete[] data;
          data   = new double[other.length];
          length = other.length;
          for (auto i=0; i<length; i++)
             data[i] = other.data[i];
      }
      return *this;
  }
  ```
- Usage: `Container b; b=a;`

TU Delft
  ```

Move assignment operator

- `Container& operator=(Container&& other)`

```
{
    if (this != &other)
    {
        delete[] data;
        data    = other.data;    other.data    = nullptr;
        legnth = other.length; other.length = 0;
    }
    return *this;
}
```

- Usage: `Container b; b=std::move(a);`

Copy vs. Move

- Move operations make sense if you want to prevent the allocation of temporal memory in nested operations
```
r = a + b + c;
```
- Task: Implement a container class with +operators (both copy and move variant) and count how many temporary container objects are created with the copy variant

TUDelft

Task: Numerical integration

- Compute a one-dimensional integral of the form

$$\int_a^b f(x)dx$$

- Numerical approximation by quadrature rule

$$\sum_{i=1}^{n} \omega_i f(x_i)$$

- Choice of quadrature weights ω_i and points x_i determines the concrete numerical integration rule

TUDelft

Simple integration rules

- Midpoint rule

$$\int_a^b f(x)dx \approx (b-a)\cdot f(\frac{a+b}{2})$$

- Simpson rule

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a)+4f(\frac{a+b}{2})+f(b) \right]$$

- Rectangle rule

$$\int_a^b f(x)dx \approx h\sum_{n=0}^{N-1} f(x_n), \quad h=\frac{b-a}{N}, \quad x_n = a+nh$$

TUDelft

Gauss integration rules

- Zoo of Gauss integration rules with quadrature weights and points tabulated for the reference interval [-1,1]
- Complete list of weights/points is available, e.g., at Wikipedia

n	ξ_i	ω_i
1	0	2
2	-0.57735026919 0.57735026919	1 1
3	-0.774596669241 0.0 0.774596669241	5/9 8/9 5/9
4	-0.861136311594053 -0.339981043584856 0.339981043584856 0.861136311594053	0.347854845137454 0.652145154862546 0.652145154862546 0.347854845137454

TU Delft

Gauss integration rules, cont'd

- Change of variable theorem

$$\int_a^b f(x)dx = \int_{-1}^{1} f(\varphi(t))\varphi'(t)dt$$

- Mapping from interval [a,b] to interval [-1,1]

$$\varphi(t) = \frac{b-a}{2}t + \frac{a+b}{2}, \quad \varphi'(t) = \frac{b-a}{2}$$

- Numerical quadrature rule

$$\int_a^b f(x)dx \approx \varphi' \sum_{i=1}^{n} \omega_i f(\varphi(\xi_i))$$

TU Delft

Program design

- We need ...
  - A strategy to ensure that all numerical quadrature rules (=classes) provide an **identical interface** for evaluating integrals
  - Standard way to **pass user-definable function** f(x) from outside (=main routine) to the evaluation function

Program design

- We need ...
  - A strategy to ensure that all numerical quadrature rules (=classes) provide an **identical interface** for evaluating integrals
    - Polymorphism: Base class `Quadrature` provides common attributes and member functions (at least their *interface declaration*); derived classes *implement* specific quadrature rule (reusing common functionality of the base class, where this is possible and makes sense)
  - Standard way to **pass user-definable function** f(x) from outside (=main routine) to the evaluation function
    - Function pointers
    - Lambda expressions (since C++11)

Program design, cont‘d

```
class Quadrature

public:
    Quadrature();
    Quadrature(int n);
    ~Quadrature();

    double integrate(...);
    double mapping(...);
    double factor(...);

private:
    double* weights;
    double* points;
    int n;
```

```
class MidpointRule

// implements midpoint rule
```

```
class SimpsonRule

// implements Simpson‘s rule
```

```
class GaussRule

// implements Gauss rules
```

Function pointers

- Define a function to be integrated

```
const double myfunc1(double x)
{
    return x;
}
```

- Define interface of the integrate function

```
double integrate(const double (*f)(double x),
                 double a, double b)
{
    // do the numerical integration
}
```

- Usage: `integrate(myfunc1, 0, 1);`

Lambda expressions

- Introduced in C++11, lambda expressions provide an elegant way to write user-defined callback functions
- General syntax

```
auto name = [<return>] (<parameters>) {<body>};
```

- Lambda expressions can be inlined (anonymous functions)

```
integrate([<return>](<parameters>) {<body>});
```

Lambda expressions, cont'd

- Return
  - [] Capture nothing
  - [&] Capture any referenced variable by reference
  - [=] Capture any referenced variable by making a copy
  - [=, &foo] Capture any referenced by making a copy, but capture variable foo by reference
  - [bar] Capture variable bar by making a copy; don't capture any other variable
  - [this] Capture the this pointer of the enclosing class

Lambda expressions, cont'd

- Define function to be integrated

```
auto myfunc2 = [](double x) { return x; };
```

- Define interface of the integration function

```
double integrate(std::function<double(double)> f,
                 double, double b)
{
    // do the integration
}
```

- Usage:

```
integrate(myfunc2, 0, 1);
integrate([](double x){ return x; }, 0, 1);
```

TUDelft

Base class Quadrature

```
class Quadrature
{
public:
    Quadrature()
    : n(0), weights(nullptr), points(nullptr) {};
    Quadrature(int n)
    : n(n), weights(new double[n]), points(new double[n]) {};
    ~Quadrature()
    { delete[] weights; delete[] points; n=0; }
private:
    double* weights;
    double* points;
    int     n;
};
```

Base class Quadrature, cont'd

- Scenario I: We want to **declare the interface** of the integrate function but we want to *force* the user to implement each integration rule individually

```
// pure (=0) virtual member function
virtual double integrate(double (*func)(double x),
                         double a, double b) = 0;
// pure (=0) virtual member function
virtual double integrate(std::function<double(double)> func,
                         double a, double b) = 0;
```

TUDelft

Base class Quadrature, cont'd

- Virtual ... = 0; declares the function **pure virtual**.
- That is, each class that is *derived* from **abstract class** `Quadrature` *must implement* this function explicitly.
- Otherwise, the compiler complains if the programmer forgets to implement a pure virtual function

TUDelft

Abstract classes

- A class with at least one pure virtual function is an **abstract class** and it is not possible to create an object thereof

```
src/integration.cxx:110:16: error: variable type
'Quadrature' is an abstract class
    Quadrature Q;
               ^
src/integration.cxx:51:20: note: unimplemented pure
virtual method 'integrate' in 'Quadrature'
    virtual double integrate(const double (*func)(const
double x), double a, double b) = 0;
```

Base class Quadrature, cont‘d

- Scenario II: We provide a *generic implementation* but allow the user to override it explicitly in a derived class

```
virtual double integrate(double (*func)(double x),
                         double a, double b) {...}
virtual double integrate(std::function<double(double)> func,
                         double a, double b) {...}
```

- Virtual declares the function **virtual**. Virtual functions can be overridden in derived classes. If no overriding takes place, the function implementation from the base class is used

TUDelft

Base class Quadrature, cont‘d

```
class Quadrature {
   // pure virtual functions (implemented in derived class)
   virtual double mapping(double xi, double a, double b) = 0;
   virtual double factor(double a, double b) = 0;

   // virtual integration function (generic implementation)
   virtual double integrate(double (*func)(double x),
                            double a, double b) {
      double integral(0);
      for (auto i=0; i<n; i++)
         integral += weights[i]*func(mapping(points[i],a,b));
      return factor(a,b)*integral;
   }
};
```

Base class Quadrature, cont‘d

- The virtual `integrate` function makes use of the pure virtual functions `factor` and `mapping`
- Both functions are **not** implemented in class `Quadrature`
- It is therefore obvious that class `Quadrature` must be an **abstract class** (and cannot be instantiated) since some of its functions (here: `integrate`) are still unavailable
- Virtual functions make it is possible to call functions in the base class which will be implemented in the derived class

Class MidpointRule

- **Derive** class `MidpointRule` from base class `Quadrature`

```
class MidpointRule : public Quadrature
{
  // Implement pure virtual mapping function (not used!)
  virtual double mapping(double xi, double a, double b)
  { return 0; }

  // Implement pure virtual factor function (not used!)
  virtual double factor(double a, double b)
  { return 1; }
};
```

Class MidpointRule, cont'd

- **Derive** class `MidpointRule` from base class `Quadrature`

```
class MidpointRule : public Quadrature
{
  ...
  // Override the implementation of the virtual integrate
  // function from class Quadrature with own implementation
  virtual double integrate(double (*func)(double x),
                           double a, double b)
  {
    return (b-a)*func(0.5*(a+b));
  }
};
```

TU Delft

Class SimpsonRule

- **Derive** class `SimpsonRule` from base class `Quadrature`

```
class SimpsonRule : public Quadrature
{
    ...
    // Override implementation of virtual integrate function
    // function from class Quadrature with own implementation
    virtual double integrate(double (*func)(double x),
                             double a, double b)
    {
      return (b-a)/6.0*(func(a)+4.0*func(0.5*(a+b))+func(b));
    }
};
```

Class GaussRule

- **Derive** class `GaussRule` from base class `Quadrature`

```
class GaussRule : public Quadrature
{ GaussRule(int n) : Quadrature(n) {
    switch(n) {
      case 1: weights[0] = { 2.0 };
              points[0]  = { 0.0 };
              break;
      case 2: ...
      default: std::cout << „Invalid argument“ << std::endl;
               exit(1);
    }
  }
};
```

Do we have access to these attributes?

Program design, revisited

```
class Quadrature

public:
    Quadrature();
    Quadrature(int n);
    ~Quadrature();

    double integrate(...);
    double mapping(...);
    double factor(...);

private:
    double* weights;
    double* points;
    int n;
```

```
class MidpointRule
: public Quadrature
// functions are public
// attributes not visible
```

```
class SimpsonRule
: public Quadrature
// functions are public
// attributes not visible
```

```
class GaussRule
: public Quadrature
// functions are public
// attributes not visible
```

Program design, revisited

```
class Quadrature

public:
    Quadrature();
    Quadrature(int n);
    ~Quadrature();

    double integrate(...);
    double mapping(...);
    double factor(...);

protected:
    double* weights;
    double* points;
    int n;
```

```
class MidpointRule
: public Quadrature
// functions are public
// attributes are private
```

```
class SimpsonRule
: public Quadrature
// functions are public
// attributes are private
```

```
class GaussRule
: public Quadrature
// functions are public
// attributes are private
```

Class GaussRule, cont'd

- Attributes from base class are now visible in derived class
- Class `GaussRule` implements functions `factor` and `mapping`

```
class GaussRule : public Quadrature
{
  virtual factor(double a, double b)
  { return 0.5*(b-a); }

  virtual mapping(double xi, double a, double b)
  { return 0.5*(b-a)*xi+0.5*(a+b); }
};
```

TU Delft

Class GaussRule, cont‘d

- Class GaussRule implements concrete factor and mapping

```
class GaussRule : public Quadrature
{
  inline virtual factor(double a, double b)
  { return 0.5*(b-a); }

  inline virtual mapping(double xi, double a, double b)
  { return 0.5*(b-a)*xi+0.5*(a+b); }
}
```

- Inline specifier „suggests“ the compiler to substitute the function body instead of calling the function explicitly.

Keyword: override (C++11)

- With the `override` keyword you can force the compiler to explicitly check that the function in a derived class overrides a (pure) virtual function from the base class

```
class GaussRule : public Quadrature
{
  virtual factor(double a, double b) override
  { return 0.5*(b-a); }
};
```

- If the base class Quadrature does not specify a (pure) virtual function `factor` an error will be thrown.

Keyword: final (C++11)

- With the `final` keyword you can force the compiler to explicitly prevent further overriding of functions

```
class GaussRule : public Quadrature
{
  virtual factor(double a, double b) final
  { return 0.5*(b-a); }
};
```

- If a class `GaussRuleImproved` derived from `GaussRule` tries to override the function `factor` an error will be thrown.