# Object Oriented Scientific Programming in C++ (WI4771TU)

Matthias Möller

Numerical Analysis

TUDelft

# 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
- 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), // using delegating constructor
        data(new double[length])
      { }
      Container(std::initializer_list<double> l)
      : Container( (int)l.size() ) // using delegation
      { std::uninitialized_copy(l.begin(), l.end(), data); }
  private:
      double* data;
      int length;
  }
  ```
  ```

# Container class, cont‘d

- Class has no default constructor

```
Container() { }
```

- Class has the (one and only) destructor

```
~Container() {
    delete[] data;
    length=0;
}
```

- All constructors/destructor have as last line

```
std::clog << „[Name of con-/destructor] called“ <<
std::endl;
```

# Container class, cont‘d

- Class has method to print information about object

```
void info()
{
    std::cout << „Length:  “ << length << std::endl;
    std::cout << „Pointer: “ << data   << std::endl;
    std::cout << „Address: “ << &data  << std::endl;
}
```

# 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})
```

# 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 types of their class

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

# 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

# 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).

# 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 declare one of the constructors or assignment operators explicitly, auto-generation of the others is disabled
- Our class provides only constructors `Container(int _length)` and `Container(std::initializer_list<double> l)`. Hence, copy and move constructurs are auto-generated by C++ compiler
- Main function with the following code compiles:

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

TUDelft

# Special member functions, cont'd

- Executable 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); // no log since copy constructor is auto-generated
  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 constructors 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
```

- Solutions
  - Disable copy constructor (you loose this handy functionality but, at least, your program does no longer crash)
  - Implement user-defined deep-copy constructor

TUDelft

# Copy constructor, cont‘d

- Explicit disabling of auto-generation of copy constructor
  `Container(const Container& c) = delete;`
- C++ 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;
     ^
```

TUDelft

# Copy constructor, cont'd

- User-implemented **deep-copy constructor**

```
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, not only fundamental type) of the content of object a when creating object b

# 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() ...
  }
  ```

TUDelft
  ```

# 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)` 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(a);
    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
    ```

TUDelft
    ```

# 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));
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
```

# 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;`
  ```

# 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);`

TUDelft

# Copy vs Move

- Move operations make sense if you want to prevent the allocation of temporal memoral in a nested operation
  `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

# 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 $\omega_i$ and points $x_i$ determines the concrete numerical integration rule

# 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 | $\xi_i$ | $\omega_i$ |
|---|---|---|
| 1 | 0 | 2 |
| 2 | -0.57735026919<br>0.57735026919 | 1<br>1 |
| 3 | -0.774596669241<br>0.0<br>0.774596669241 | 5/9<br>8/9<br>5/9 |
| 4 | -0.861136311594053<br>-0.339981043584856<br>0.339981043584856<br>0.861136311594053 | 0.347854845137454<br>0.652145154862546<br>0.652145154862546<br>0.347854845137454 |

# 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
    - Polymorphism: Base class Quadrature provides common attributes and function members (at least their inderface 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

// implement midpoint rule
```

```
class SimpsonRule

// implement Simpson rule
```

```
class GaussRule

// implement 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 Quadratue
  {
  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 specify the interface of the integrate function but we want to force the user to implement each integration rule individually

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

- Virtual … = 0; declares the function **pure virtual**. That is, each class that is derived from class Quadrature must implement this function explicitly. Compiler complains of the programmer forgets to implement a pure virtual function

# 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;
```

TUDelft

# Base class Quadrature, cont'd

- Scenario II: We provide a generic implementation but allow the user to overwrite it 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 overwritten in derived classes. If no overwriting takes place, the function implementation from the bas class is used

TUDelft

# Base class Quadrature, cont'd

- ```
  class Quadratue
  {
  // pure virtual mapping functions (must be implemented!)
  virtual double mapping(double xi, 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;
  }
  ```

TUDelft
  ```

# Base class Quadrature, cont'd

- Note that the virtual integrate function makes use of the pure virtual functions factor and mapping which are not implemented in the 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 simple unavaiable.

# Class MidpointRule

- Class MidpointRule is derived from the base class Quadrature

```
class MidpointRule : public Quadrature
{
  // Implementation of the pure virtual mapping function
  virtual double mapping(double xi, double a, double b)
  { return 0; }

  // Implementation of the pure virtual factor function
  virtual double factor(double a, double b)
  { return 1; }
}
```

# Class MidpointRule, cont'd

- Class MidpointRule is derived from the base class Quadrature

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

TUDelft

# Class SimpsonRule

- Class SimpsonRule is derived from the base class Quadrature

```
class SimpsonRule : public Quadrature
{
  ...
  // Overwriting implementation of virtual integrate function
  virtual double integrate(double (*func)(double x),
                                  double a, double b)
  { double m=0.5*(a+b);
    return (b-a)/6.0*(func(a)+4*func(m)+func(b));
  }
}
```

TUDelft

# Class GaussRule

- Class GaussRule is derived from the 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);
    }
  }
}
```

# 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.