Object-oriented scientific programming with C++

Matthias Möller, Jonas Thies, Cálin Georgescu, Jingya Li (Numerical Analysis, DIAM)
Lecture 2

Task: Dot product

Write a $\mathrm{C++}$ code that computes the dot product

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

of two vectors $a = \left[a_1, a_2, \ldots, a_n\right]$ and $b = \left[b_1, b_2, \ldots, b_n\right]$ and terminates if the two vectors have different length.

Dot product function

The main functionality without any fail-safe checks

In [ ]:

```
double dot_product(const double* a, int n, const double* b, int m)
{
    double d=0.0;
    for (auto i=0; i<n; i++)
        d += a[i]*b[i];
    return d;
}
```

Dot product function - improved version

First version of the dot product with exception

In [ ]:

```
#include <exception>

double dot_product(const double* a, int n, const double* b, int m)
{
    if (a == nullptr || b == nullptr) {
        // Handle null pointers by throwing an exception
        throw std::invalid_argument("Null pointer argument");
    }

    if (n != m) {
        // Handle mismatched sizes by throwing an exception
        throw std::invalid_argument("Array sizes mismatch");
    }

    // Core functionality
    double d = 0.0;
    for (int i = 0; i < n; ++i) {
        d += a[i] * b[i];
    }
    return d;
}
```

In [ ]:

```
#include <iostream>

double x[5] = { 1, 2, 3, 4, 5 };
double y[4] = { 1, 2, 3, 4 };
try {
    double d = dot_product(x, 5, y, 4);
} catch (const std::exception &e) {
    std::cout << e.what() << std::endl;
}
```

It would be much better if `x` and `y` would "know" their length internally so that the calling function cannot provide inconsistent data

Cool! But what was the reason I enrolled in this course?

Flashback: Object Oriented Programming

Imagine each LEGO block as a piece of data (like an integer, string, etc.). A `struct` or a `class` is like a LEGO set that contains a variety of different blocks.

- OOP: bundle data (e.g. `array`) and functionality (e.g. `length`) into a `struct` or `class`
- Components of a struct are `public` (=can be accessed from outside the struct) by default
- Components of a class are `private` (=cannot be accessed from outside the class) by default
- Components of a struct/class are attributes and member functions (=methods)

Class vs. struct

In [ ]:

```
class Vector {
    private: //default
    public:
        double* array;
        int length;
};
```

In [ ]:

```
struct Vector {
    public: // default
        double* array;
        int length;
    private:
};
```

When to use `class` and when to use `struct`?

- `struct` is typically used when you want to group data together without needing to restrict access to it. It is straightforward and simple. Many type traits (later in this course) are implemented as `struct`s.
- `class` is typically used when you want more control over the data and the interface through which it is accessed and manipulated, promoting the principles of encapsulation and data hiding.

Dot product as a member function of Vector

Second version of the dot product using Vector class or struct

In [ ]:

```
#include <exception>

class Vector{
    public:
        double* array;
        int length;

    double dot_product(const Vector& a, const Vector& b)
    {
        if(a.length != b.length)
            throw std::invalid_argument("Vector lengths mismatch");

        double d=0.0;
        for (auto i=0; i<a.length; i++)
            d += a.array[i]*b.array[i];
        return d;
    }
};
```

Access of members of a `struct` or `class` by dot-notation („.“)

Is the above implementation really OOP? How would you invoke it from the main program?

In [ ]:

```
#include <iostream>

Vector x,y;
x.array = new double[5]; x.length = 5;
y.array = new double[5]; y.length = 5;
for (int i = 0; i < 5; ++i) {
        x.array[i] = i + 1;  // 1, 2, 3, 4, 5
        y.array[i] = i + 1;  // 1, 2, 3, 4, 5
}

try {
    double result = /** ??? **/;
    } catch (const std::exception &e) {
    std::cout << e.what() << std::endl;
}
```

The current implementation of the `dot_product` function comes from functional programming and is not OOP style because it takes two input arguments `x` and `y` and returns one output argument, the dot product. In other words, it is not attached to any object (`x` or `y`).

If we want to implement a function inside a `class` or `struct` that is not attached to an object we have to define it as `static`

In [ ]:

```
#include <exception>

class Vector{
    public:
        double* array;
        int length;

    double dot_product(const Vector& a, const Vector& b)
    {
        if(a.length != b.length)
            throw std::invalid_argument("Vector lengths mismatch");

        double d=0.0;
        for (auto i=0; i<a.length; i++)
            d += a.array[i]*b.array[i];
        return d;
    }
};
```

static member functions are invoked with the full classname (comparable to namespaces)

In [ ]:

```
#include <iostream>

Vector x,y;
x.array = new double[5]; x.length = 5;
y.array = new double[5]; y.length = 5;
for (int i = 0; i < 5; ++i) {
        x.array[i] = i + 1;  // 1, 2, 3, 4, 5
        y.array[i] = i + 1;  // 1, 2, 3, 4, 5
}

try {
    double result = Vector::dot_product(x, y);
    } catch (const std::exception &e) {
    std::cout << e.what() << std::endl;
}
```

- It is still possible to initialise `x.length` by the wrong value, e.g., `x.array = new double[5] {1, 2, 3, 4, 5}; x.length = 4;`
- The main function is not very readable due to the lengthy declaration, initialisation and deletion of data
- OOP solution:
  - **Constructor(s):** method to construct a new Vector object
  - **Destructor:** method to destruct an existing Vector object

Constructor

The constructor is called each time a new Vector object (=instance of the class Vector) is created

In [ ]:

```
class Vector
{
    public:
        double* array;
        int length;

    Vector() // Default constructor
    {
        array = nullptr;
        length = 0;
    }
};
```

A class can have multiple constructors if they have a different interface (=different parameters)

In [ ]:

```
class Vector
{
    public:
        double* array;
        int length;

    Vector() // Default constructor
    {
        array = nullptr;
        length = 0;
    }

    Vector(int len) // Another constructor
    {
        array = new double[len];
        length = len;
    }
};
```

What if a parameter has the same name as an attribute?

In [ ]:

```
class Vector
{
    public:
        double* array;
        int length;

    Vector(int length) // Another constructor
    {
        // this pointer refers to the object itself,
        // hence this->length is the attribute and length
        // is the parameter passed to the constructor

        array = new double[length];
        this->length = length;
    }
};
```

Destructor

The destructor is called implicitly at the end of the lifetime of a Vector object, e.g., at the end of its scope

In [ ]:

```
class Vector
{
    public:
        double* array;
        int length;

    ~Vector() // Destructor (and there can be only one!)
    {
        delete[] array;
        length = 0;
    }
};
```

Cleaning up the main program with constructors and (implicitly invoked) destructors

int main(){ Vector x; // Default constructor is called { Vector y(5); // Constructor is called // Destructor is called for Vector y } // Destructor is called for Vector x }

Without array = nullptr in the default constructor the destruction of x will lead to a run-time error.

Uniform initialisation constructors (C++11)

Remember this double x[5] = { 1, 2, 3, 4, 5 };

It would be cool to simply write Vector x = { 1, 2, 3, 4, 5 };

C++11 solution: initializer lists

Vector(const std::initializer_list&ltdouble&gt&amp list) { length = (int)list.size(); array = new double[length]; std::uninitialized_copy(list.begin(), list.end(), array); }

In [ ]:

```
#include <initializer_list>
#include <memory>

class Vector {
private:
    double* array;
    int length;

public:
    // Constructor with initializer list
    Vector(const std::initializer_list<double>& list)
    {
        length = (int)list.size();
        array = new double[length];
        std::uninitialized_copy(list.begin(), list.end(), array);
    }

    // Destructor
    ~Vector() {
        delete[] array;
    }
};
```

Dot product – close to perfection

Third version of the dot product using Vector class with uniform initialisation constructor (C++11) and exceptions

int main() { Vector x = { 1, 2, 3, 4, 5 }; Vector y = { 2, 4, 6, 8, 10}; try { double dot_product(x, y); } catch (const std::exception &e) { std::cout &lt&lt e.what() &lt&lt std::endl; } }

Delegating constructor (C++11)

Can we delegate some of the work

In [ ]:

```
#include <memory>
#include <initializer_list>

class Vector {
private:
    double* array;
    int length;

public:
    Vector(int length)
    {
        this->length = length;
        array = new double[length];
    }

    Vector(const std::initializer_list<double>& list)
    {
        length = (int)list.size();
        array = new double[length];
        std::uninitialized_copy(list.begin(), list.end(), array);
    }
}
```

Delegating constructors delegate part of the work to another constructor of the same or another class

```
Vector(int length) : length(length), array(new double[length]) { }
```

- Here, delegation is not really helpful but more a question of coding style, e.g., some programmers use delegation in all situation where this is technically possible
- It is no longer necessary to distinguish between the **attribute** (`this->length`) and the **argument** (`length`) if both have the same name. But be careful with the order in which delegated objects are constructed!

Quiz

- `Vector(int len)`
  `  : length(len),`
  `    array(new double[len])`
  `  {}`
- `Vector(int len)`
  `  : array(new double[len]),`
  `    length(len)`
  `  {}`
- `Vector(int len)`
  `  : length(len),`
  `    array(new double[lengh])`
  `  {}`
- `Vector(int len)`
  `  : array(new double[length]),`
  `    length(len)`
  `  {}`

If you have multiple constructors with increasing functionality, delegating constructors can be really helpful to remove duplicate code, e.g.

```
Vector(const std::initializer_list&ltdouble>& list) : Vector((int)list.size()) {
std::uninitialized_copy(list.begin(), list.end(), array); }
```

Function -> member function

Function that computes the sum of a Vector

```
static double sum(const Vector& a) { double s = 0; for (auto i=0; i<a.length; i++) s += a.array[i]; return s; }
```

This is not really OOP-style!

```
int main() { Vector x = { 1, 2, 3, 4, 5 }; std::cout &lt< sum(x) &lt< std::endl; }
```

Implementation of sum as an OOP-style member function

In [ ]:

```
#include <memory>
#include <initializer_list>

class Vector {
    private:
        double* array;
        int length;

    public:
        Vector(const std::initializer_list<double>& list) {
            length = static_cast<int>(list.size());
            array = new double[length];
            std::uninitialized_copy(list.begin(), list.end(), array);
        }

        ~Vector() {
            delete[] array;
        }

        double sum() {
            double s = 0;
            for (int i = 0; i < length; i++) {
                s += array[i];
            }
            return s;
        }
};
```

This is good OOP-style

```
Vector v = {1.0, 2.0, 3.0}; std::cout &lt< v.sum() &lt< std::endl;
```

Can we implement the `dot_product` function as a member function?

In [ ]:

```
#include <exception>

class Vector {
    private:
        double* array;
        int length;

    public:
        double dot_product(const Vector& other) {
            if (length != other.length)
                throw std::invalid_argument("Vector lengths mismatch");

            double d=0.0;
            for (auto i=0; i<length; i++)
                d += array[i]*other.array[i];
            return d;
        }
};
```

This is good OOP-style

```
int main() { Vector x = {1,2,3}; Vector y = {2,4,6}; std::cout &lt< x.dot_product(y)
&lt< std::endl; std::cout &lt< y.dot_product(x) &lt< std::endl; }
```

Formally, the dot product is an operation between two Vector objects and not a member function of one Vector object that needs another Vector object for calculation

Operator overloading

C++ allows to overload (=redefine) the standard operators

- Unary operators: `++a`, `a++`, `--a`, `a--`, `~a`, `!a`
- Binary operators: `a+b`, `a-b`, `a*b`, `a/b`
- Relational operators: `a==b`, `a!=b`, `a<b`, `a<=b`, `a>b`, `a>=b`

Interfaces:

> return_type operator() [const] return_type operator(const Vector& other) [const]

Complete list:https://en.cppreference.com/w/cpp/language/operators

Implementation of dot product as overloaded *-operator

double operator*(const Vector&amp other) const { if (length != other.length) throw std::invalid_argument("Vector lengths mismatch"); double d=0.0; for (auto i=0; i<length; i++) d += array[i]*other.array[i]; return d; }

Now, the dot product is implemented as *-operator that maps two Vector objects to a scalar value

```
int main() { Vector x = {1,2,3}; Vector y = {2,4,6}; std::cout &lt< x * y &lt< std::endl; std::cout &lt< y * x &lt< std::endl; }
```

The const specifier indicates that the Vector reference other must not be modified by the *-operator
The trailing const specifier indicates that the this pointer (aka, the object whose function is invoked) must not be modified by the *-operator

```
double operator*(const Vector&amp other) const { ... }
```

Assignment by operator overloading

Implementation of assignment as overloaded =-operator

```
Vector& operator=(const Vector& other) { if (this != &other) { length = other.length; delete[] array; array = new double[length]; for (auto i=0; i<length; ++i) array[i] = other.array[i]; } return *this; }
```

- Usage: `Vector x, y; x = y;`
- Note that the `this` pointer is modified so there must not be a trailing `const`

Implementation of incremental assignment as overloaded =-operator
`Vector&amp operator+=(const Vector&amp other) { if(length != other.length) throw std::invalid_argument("Vector lengths mismatch"); for (auto i=0; i<length; i++) array[i] += other.array[i]; return *this; }`

- Usage: `Vector x, y; x += y;`
- Note that the `this` pointer is modified so there must not be a trailing `const`

Container class

```
class Container {
private:
    double* data;
    int length;
public:
    Container(int length)
    : length(length), data(new double[length])
    { }
    Container(const std::initializer_list<double>& l)
    : Container( (int)l.size() )
    {
        std::uninitialized_copy(l.begin(), l.end(), data);
    }
};
```

Conversion constructors

Both constructors convert a single input argument into a Container object, hence, they are called conversion constructors. They can be called in two different ways

- Using the regular construction form

```
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} )
Container a = {4};       // which constructor is called?
```

Explicit specifier

The explicit specifier prevents the use of the constructor as conversion constructor
`explicit Container(int length) : length(length), data(new double[length]) { } }`
Now, copy-initialisation (`Container a = 4;`) is no longer possible but explicit constructor (`Container a( 4 );`) has to be used

Constructors summary

Constructor	Description	Usage
Default	Constructor with no parameters.	Used to create an object with default values.
Parameterized	Constructor with parameters to initialize an object with specific values.	Used to create an object with specified attributes.
Copy	A constructor that initializes an object using another object of the same class.	Used to create a copy of an object.
Explicit	Constructor with the explicit keyword to prevent implicit conversions or copy-initialization.	Used to enforce explicit object creation with constructor.

Task: Numerical integration

- Approximate a one-dimensional integral by numerical quadrature

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f\left(x_i\right)$$

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

Simple integration rules

- Midpoint rule

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

- Simpson rule

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

- Rectangle rule

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

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	w_i
1	0	2
2	-0.57735026919	2
	0.57735026919	1
3	-0.774596669241	5/9
	0.0	8/9
	0.774596669241	5/9
4	-0.861136311594053	0.347854845137
	-0.339981043584856	0.652145154862
	0.774596669241	0.652145154862
	0.861136311594053	0.347854845137

- Change of variable theorem

$$\int_a^b f(x)dx = \int_{-1}^1 f(\phi(t))\phi^i(t)dt$$

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

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

- Numerical quadrature rule

$$\int_a^b f(x)dx \approx \phi^{'} \sum_{n=1}^{n} w_i f(\phi(\xi_i))$$

Program design

We need...

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

- 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)

- A standard way to **pass user-definable function** f(x) from outside (=main routine) to the evaluation function
  - Function pointers (traditional approach)
  - Lambda expressions (recommended approach since C++11)

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 (*func)(double x), double a, double b) { ... }
```

- 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 = [&ltcaptures>] (&ltparameters>) {&ltbody>};
```

- Lambda expressions can be inlined (anonymous functions)

```
integrate([&ltcaptures>](&ltparameters>) {&ltbody>});
```

- Define function to be integrated

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

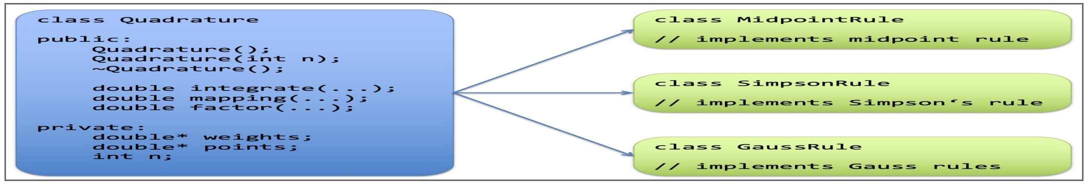
- Define interface of the integration function

```
double integrate(std::function&ltdouble(double)> func, double a, double b) const { ... }
```

- Usage:

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

Program design, revisited



Base class `Quadrature`

In [ ]:

```
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;
};
```

- *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) const = 0;

```
// pure (=0) virtual member function
virtual double integrate(std::function&ltdouble(double)> func, double a,
double b) const = 0;
```

- Keyword `virtual ... = 0;` declares the function to be **pure virtual**
- That is, each class that is derived from the **abstract class** `Quadrature` must(!!!) implement this function explicitly
- Otherwise, the compiler complains when the programmer forgets to implement a pure virtual function and tries to create an object of the derived but not fully implemented class

Base class `Quadrature`

- *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) const {...}
```

```
virtual double integrate(std::function&ltdouble(double)> func, double a, double b) const{...}
```

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

In [ ]:

```
class Quadrature {
    private:
        double* weights;
        double* points;
        int     n;

    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; }

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

    // ** virtual integration function (generic implementation) ** /
    virtual double integrate(double (*func)(double x), double a, double b) const {
        double integral(0.0);
        for (auto i=0; i<n; i++)
            integral += weights[i]*func(mapping(points[i],a,b));
        return factor(a,b)*integral;
    }
};
```

- 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

In [ ]:

```
class MidpointRule : public Quadrature
{
    // Implement pure virtual functions (not used but need to be implemented!)
    virtual double mapping(double xi, double a, double b) const  { return 0; }
    virtual double factor(double a, double b) const { return 1; }

    // 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) const
    {
        return (b-a) * func( 0.5 * (a+b) );
    }
};
```

Class `SimpsonRule`

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

In [ ]:

```
class SimpsonRule : public Quadrature
{
    // Implement pure virtual functions (not used but need to be implemented!)
    virtual double mapping(double xi, double a, double b) const  { return 0; }
    virtual double factor(double a, double b) const { return 1; }

    // 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) const
    {
        return (b-a)/6.0 * ( func(a) + 4.0 * func( 0.5*(a+b) ) + func(b) );
    }
};
```

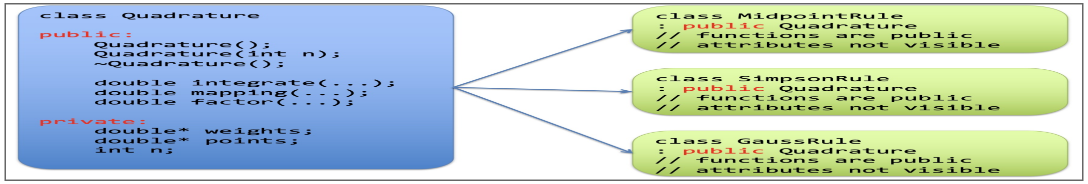
Class `GaussRule`

Derive class GaussRule from base class Quadrature

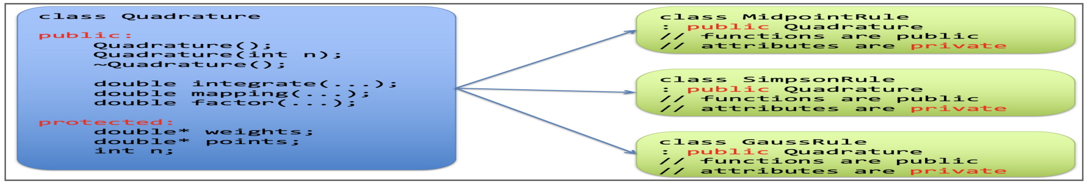
```
class GaussRule : public Quadrature
{ GaussRule(int n) : Quadrature(n) {
    switch(n) {
      case 1: weights[0] = { 2.0 };
              points[0]  = { 0.0 };
              break;
      case 2: ...
      default: throw “Invalid argument“;
    }
  }
};
```

Do we have access to these attributes?

Program design, revisited



Program design, revisited



Class `GaussRule`

- Attributes from base class are now visible in derived class
- Class GaussRule implements functions factor and mapping
- Class GaussRule inherits the virtual function integrate from class Quadrature

In [ ]:

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

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

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

In [ ]:

```
class GaussRule : public Quadrature
{
    virtual double factor(double a, double b) const 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

In [ ]:

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

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

In [ ]:

```

```

In [3]:

```
import asyncio
import os
import tempfile

from subprocess import PIPE, Popen
from pyppeteer import launch

import concurrent.futures

async def html_to_pdf(html_file, pdf_file, pyppeteer_args=None):
    """Convert a HTML file to a PDF"""
    browser = await launch(
```

```python
        handleSIGINT=False,
        handleSIGTERM=False,
        handleSIGHUP=False,
        headless=True,
        args=["--no-sandbox"],
    )

    page = await browser.newPage()
    await page.setViewport(dict(width=994, height=768))
    await page.emulateMedia("screen")

    await page.goto(f"file://{html_file}", {"waitUntil": ["networkidle2"]})

    page_margins = {
        "left": "20px",
        "right": "20px",
        "top": "30px",
        "bottom": "30px",
    }

    dimensions = await page.evaluate(
        """() => {
        return {
            width: document.body.scrollWidth,
            height: document.body.scrollHeight,
            offsetWidth: document.body.offsetWidth,
            offsetHeight: document.body.offsetHeight,
            deviceScaleFactor: window.devicePixelRatio,
        }
    }"""
    )
    width = dimensions["width"]
    height = dimensions["height"]

    await page.pdf(
        {
            "path": pdf_file,
            "format": "A4",
            "printBackground": True,
            "margin": page_margins,
        }
    )

    await browser.close()
```

```python
if __name__ == "__main__":

    html_input_file = "http://localhost:8888/notebooks/Desktop/Cpp-slides/OospCpp/notebooks/lecture2.ipynb#
/slide-0-0?print-pdf"
    pdf_output_file = "slides.pdf"

    pool = concurrent.futures.ThreadPoolExecutor()
    pool.submit(
        asyncio.run,
        html_to_pdf(
            html_input_file,
            pdf_output_file
        ),
    ).result()
```

```
---------------------------------------------------------------------------
NetworkError                              Traceback (most recent call last)
Cell In[3], line 72
     63 pdf_output_file = "slides.pdf"
     65 pool = concurrent.futures.ThreadPoolExecutor()
     66 pool.submit(
     67     asyncio.run,
     68     html_to_pdf(
     69         html_input_file,
     70         pdf_output_file
     71     ),
---> 72 ).result()

File ~/anaconda3/lib/python3.11/concurrent/futures/_base.py:456, in Future.result(self, timeout
)
    454     raise CancelledError()
    455 elif self._state == FINISHED:
--> 456     return self.__get_result()
    457 else:
    458     raise TimeoutError()

File ~/anaconda3/lib/python3.11/concurrent/futures/_base.py:401, in Future.__get_result(self)
    399 if self._exception:
    400     try:
--> 401         raise self._exception
    402     finally:
    403         # Break a reference cycle with the exception in self._exception
```

```
    404         self = None

File ~/anaconda3/lib/python3.11/concurrent/futures/thread.py:58, in _WorkItem.run(self)
     55     return
     57 try:
---> 58     result = self.fn(*self.args, **self.kwargs)
     59 except BaseException as exc:
     60     self.future.set_exception(exc)

File ~/anaconda3/lib/python3.11/asyncio/runners.py:190, in run(main, debug)
    186     raise RuntimeError(
    187         "asyncio.run() cannot be called from a running event loop")
    189 with Runner(debug=debug) as runner:
--> 190     return runner.run(main)

File ~/anaconda3/lib/python3.11/asyncio/runners.py:118, in Runner.run(self, coro, context)
    116 self._interrupt_count = 0
    117 try:
--> 118     return self._loop.run_until_complete(task)
    119 except exceptions.CancelledError:
    120     if self._interrupt_count > 0:

File ~/anaconda3/lib/python3.11/asyncio/base_events.py:650, in BaseEventLoop.run_until_complete
(self, future)
    647 if not future.done():
    648     raise RuntimeError('Event loop stopped before Future completed.')
--> 650 return future.result()

Cell In[3], line 24, in html_to_pdf(html_file, pdf_file, pyppeteer_args)
     21 await page.setViewport(dict(width=994, height=768))
     22 await page.emulateMedia("screen")
---> 24 await page.goto(f"file://{html_file}", {"waitUntil": ["networkidle2"]})
     26 page_margins = {
     27     "left": "20px",
     28     "right": "20px",
     29     "top": "30px",
     30     "bottom": "30px",
     31 }
     33 dimensions = await page.evaluate(
     34     """() => {
     35     return {
   (...)
     42 }"""
     43 )
```

```
File ~/anaconda3/lib/python3.11/site-packages/pyppeteer/page.py:829, in Page.goto(self, url, op
tions, **kwargs)
    826 timeout = options.get('timeout', self._defaultNavigationTimeout)
    827 watcher = NavigatorWatcher(self._frameManager, mainFrame, timeout, options)
--> 829 result = await self._navigate(url, referrer)
    830 if result is not None:
    831     raise PageError(result)

File ~/anaconda3/lib/python3.11/site-packages/pyppeteer/page.py:843, in Page._navigate(self, ur
l, referrer)
    842 async def _navigate(self, url: str, referrer: str) -> Optional[str]:
--> 843     response = await self._client.send('Page.navigate', {'url': url, 'referrer': referr
er})
    844     if response.get('errorText'):
    845         return f'{response["errorText"]} at {url}'

NetworkError: Protocol error (Page.navigate): Cannot navigate to invalid URL
```

```
In [ ]:
```