Inheritance

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing

Outline

Introduction

- In the previous lecture we have developed a method for grid generation for PDEs on special ("four-sided") domains.
- The sides can be described by rather general (smooth) curves.
- The present lecture aims at:
  - Developing a general class for handling computational domains (structured grids)
  - Developing classes for handling sides (discrete curves)
  - Providing the necessary object-oriented tools available in C++
  - In particular: Inheritance

A Domain Class Skeleton

```
class Domain {
  public:
    Domain(Curvebase&, Curvebase&, Curvebase&,
           Curvebase&);
    void generate_grid (...);
    // more members
  private:
    Curvebase *sides[4];
    // more members
};
```

A Boundary Curve Skeleton

A first attempt:

```
class Curvebase {
  public:
    double x(double s); // Curve parametrization
    double y(double s);
    double xp(double p);   // Same in user coordinates
    double yp(double p);
    // more members
  private:
    double a, b; // Range for p
    // more members
};
```

Parametrized Curves

- Parametrized curves are given by

$$(x(p), y(p)), \quad p \in [a, b]$$

with the non-empty finite interval $[a, b]$.

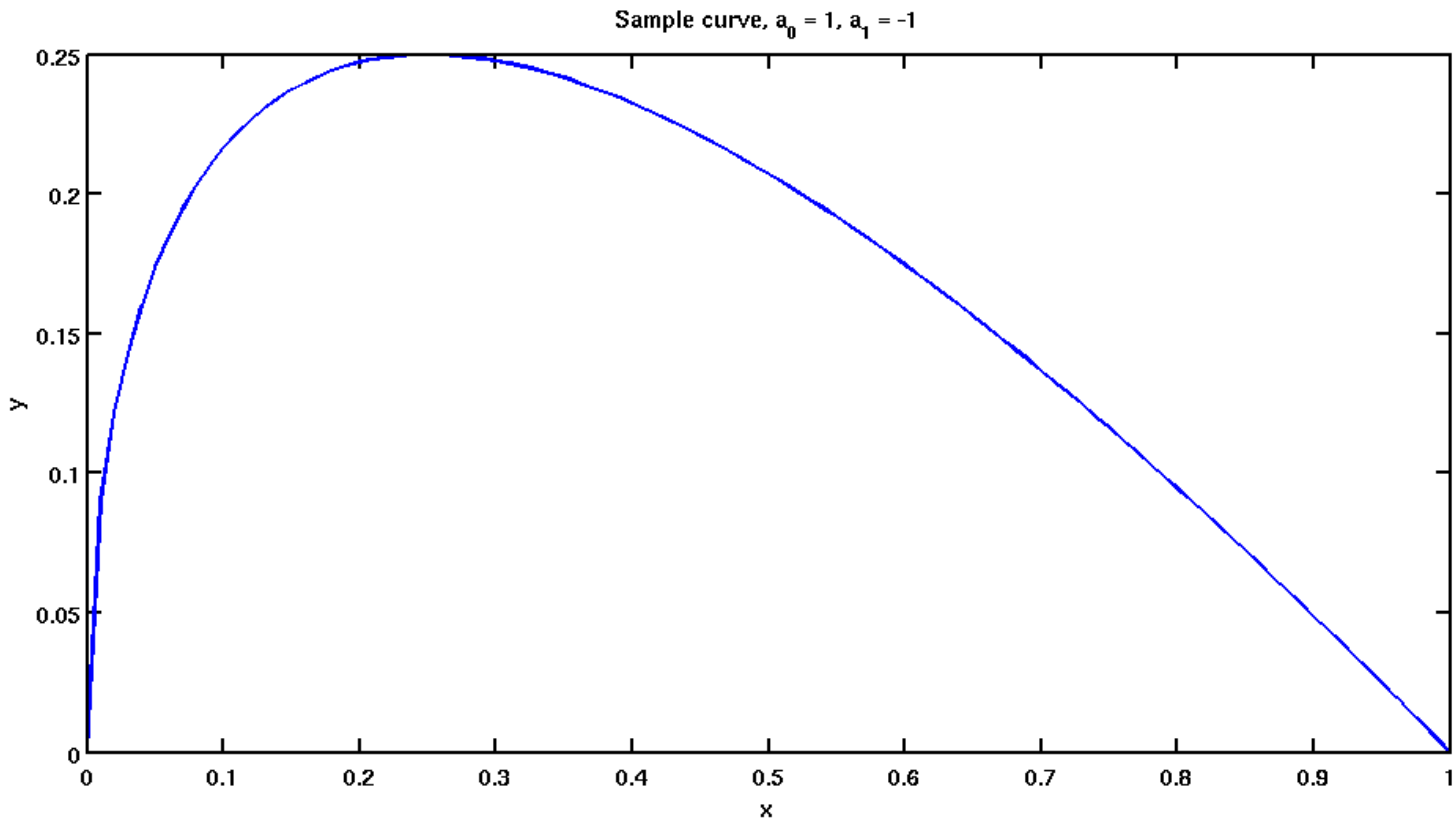
- Example:

$$(x(p), y(p)) = (p, a_0\sqrt{p} + a_1 p), \quad 0 \leq a < b.$$

The class should support

  - specification of parameters a, b, a_0, a_1
  - computation of $x(p)$ and $y(p)$

Example Curve



Implementation

```
class Bcurve {
  public:
    Bcurve(double a, double b, double a0, double a1) :
         a_(a), b_(b), a0_(a0), a1_(a1) {}
    double xp(double p) {return p;}
    double yp(double p) {return a0_*std::sqrt(p)+a1_*p;}
  private:
    double a_, b_, a0_, a1_;
};
```

In a real implementation it should be checked that

- in the constructor: $0 \leq a < b$,
- in `x` and `y`: $a \leq p \leq b$.

Boundary Representation

- The user should be allowed to provide the representation of the curve as it is most convenient for him/her.
- This parametrization by p may not be convenient for grid generation. Grid generation should be controlled by numerical aspects.
- In particular, equidistant grids should be easy to generate.

Solution:

- Specify the curve using arbitrary $\mathbf{X}(p)$, $p \in [a, b]$.
- Specify node distribution using $\mathbf{x}(s)$, $s \in [0, 1]$, the normalized arc length.

Transformation to Arc Length Coordinates

- Arc length $l(p)$ of $\{\mathbf{X}(q)|q \in [a,p]\}$:

$$l(p) = \int_a^p \sqrt{X'(q)^2 + Y'(q)^2}\,dq$$

- Given s, find p such that $\mathbf{X}(p) = \mathbf{x}(s)$ amounts to solving the nonlinear scalar equation

$$f(p) = l(p) - s \cdot l(b) = 0.$$

- Appropriate method: Newtond method: Given p_0, iterate until convergence

$$p_{i+1} = p_i - f(p_i)/f'(p_i)$$

Remarks

- Points along the curve can be computed for any $s \in [0, 1]$.
- Evaluating $\mathbf{x}(s)$ is (much) more expensive then evaluating $\mathbf{X}(p)$.
- Only "a few" points needed initially to generate grid on the boundaries.
- Useful, when the exact curve is not known, e.g., spline representation from a CAD model.
- Smoothnes of $\mathbf{X}$ is required if numerical integration/differentiation is used. *Why?*

Declaration

```
class Curvebase {
  public:
    Curvebase(double a = 0.0, double b = 1.0) :
                a_(a), b_(b);
    double x(double s); // Coordinates in arc length
    double y(double s);
    ~Curvebase();
    // more members
  protected:
    double a_, b_;
    double xp(double p);  // User parametrization
    double yp(double p);
    double dxp(double p); // derivatives
    double dyp(double q);
    double integrate(double p);
    // more members
};
```

Inheritance

- We can define classes for many different curves along the lines of `Curvebase`: Lines, circles, `Bcurve` etc
- This approach is possible but not very elegant:
  - several classes representing small variations of the same concept
  - additional functionality may significantly increase the size of the code
  - increased probability of errors when maintaining multiple copies
- Preferred solution:
  - The curves are conceptionally equivalent, let them inherit properties from a generic parametrized curve.
  - Make changes only when they are needed.

Derived Classes

A derived class is defined by

Derived class

```
class derived : label base, ... ;
```

- label is public, protected or private
- The derived class `derived` inherits members from its base class(es) `base`.
- Members of the base classes can be overwritten as usual in the derived class.

Rules For Visibility in Derived Classes

label	visibility in base		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Usually, the label public is what you want.

Reimplementation of Bcurve

```
class Curve1 : public Curvebase {
  public:
    Curve1(double a, double b, double a0, double a1) :
        Curvebase(a,b), a0_(a0), a1_(a1) {}
  private:
    double a0_, a1_;
    double xp(double p) {return p;}
    double yp(double p) {return a0_*std::sqrt(p)+a1_*p;}
    double dxp(double p) {return 1.0;}
    double dyp(double p) {
       if (p == 0.0) return HUGE_VAL;
       else return 0.5*a0_/std::sqrt(p)+a1_;
    }
};
```

Notes

- In order to get direct access to the protected members of the base class in the derived class, the explicit use of this is necessary!
- Example: Query for member a in Curvebase (do not forget to add the declaration in the class declaration)

```
Curve1::geta() {return this->a_; }
```

Remarks on Constructors

- In a derived class, a constructor of the base class can be directly invoked

  ```
  derived(...) : base(...) { ... }
  ```

  If this has not been done, the default constructor of the base class is called.
- The default constructor of the derived class invokes the default constructor of the base class.
- Order of initialization:
  - Base class constructor
  - Initializations of the derived classes data members
  - The statements of the function block `{...}`
- The complete process can become rather complex if the inheritance includes multiple inheritance (more than one base class) or inheritance over sequences of derivations!

Remarks on Destructors

- Destructors will not be inherited.
- Desctructors cannot be overwritten.
- Execution order of destructors:
  - The statements of the function block `{...}`
  - Base class destructors

Pointers and Derived Classes

- An instance of a derived class contains all information from the base class.
- Therefore, a type cast for pointers is meaningful:

```
class base {};
class derived : public base {};
base *p, *r;
derived *q;
p = new base;    // OK, creates an instance of base
q = new derived; // OK, creates an instance of derived
r = q; // OK, but only members of base are accessible (s
```

Pointers and Derived Classes (cont)

```
class base {};
class deriv1 : public base {};
class deriv2 : public base {
  public: void f() {}
};
int main() {
  base *p = new deriv2;
  deriv2 *r = new deriv2;
  p->f();  // Error base::f() not defined
  ((deriv1*) p)->f(); // Error deriv1::f() not defined
  ((deriv2*) p)->f(); // OK, explicit type cast
  r->f();  // OK, deriv2::f() defined
}
```

Hint: UML

Slicing: A Warning

```
class A {
private:
  int A_ = 0;
public:
  int getA() const { return A_; }
  A(const int a) : A_(a) {}
};
class B : public A {
private:
  int B_ = 0;
public:
 int getB() const { return B_; }
 B(const int a, const int b) : A(a), B_(b) {}
};
```

Slicing: A Warning (cont)

```
int main() {
  B b1(11,12);
  B b2(21,22);
  A& a_ref = b2;
  a_ref = b1;
  std::cout << "A = " << b2.getA() << ", B = "
            << b2.getB() << std::endl;
  return 0;
}
```

Output:

```
A = 11, B = 22
```

Oops! What is going on??

- *This effect may lead to an undefined state of an object!*

Dynamic Binding

- All what we have done so far had a nice property:
  - At each point in the code it was clear *at compile time* which version of an (overloaded) function to call.
  - This property is called *static binding*. (Not to be confused with statically linked programs!)
- This is not possible for our intended application. Consider an excerpt of our Domain class:

```
private: Curvebase *sides[4];
```

- The aim is to assign pointers to derived classes (for example `Curve1`, and others) to sides[i]. Since these objects will be created *dynamically during runtime*, its class is not known at compile time!
- What we will need is *dynamic binding*.

Dynamic Binding (cont)

- Each pointer has a static type.
- The dynamic type can vary:

```
class base {};
class derived : public base {};
base *p, *r;
p = new base;    // Dynamic type base*
r = new derived; // Dynamic type derived*
```

- Functions which are intended to be capable of dynamic binding are declared virtual:

```
virtual double Curvebase::xp(double p);
```

- A function declared virtual in a base class is virtual in derived classes even if the keyword `virtual` is not explicitely given.

Dynamic Binding: Example

```
class base {
public:
    void whoami() { cout << " base" << endl; }
};
class deriv1 : public base {
public:
    void whoami() { cout << " deriv 1" << endl; }
};
class deriv2 : public base {
public:
    void whoami() { cout << " deriv 2" << endl; }
};
```

Example (cont)

```
int main() {
    base *p, *all[3];
    deriv1 *q;
    deriv2 *r;
    p = new base;
    q = new deriv1;
    r = new deriv2;
    all[0] = p;
    all[1] = q;
    all[2] = r;
    p->whoami();
    q->whoami();
    r->whoami();
    all[0]->whoami();
    all[1]->whoami();
    all[2]->whoami();
}
```

Example: Output

Demo:

```
> ./a.out
base
deriv 1
deriv 2
base
base
base
```

Modified Example

```
class base {
public:
    virtual void whoami() { cout << " base" << endl; }
};
class deriv1 : public base {
public:
    void whoami() { cout << " deriv 1" << endl; }
};
class deriv2 : public base {
public:
    void whoami() { cout << " deriv 2" << endl; }
};
```

Modified Output

Demo:

```
> ./a.out
base
deriv 1
deriv 2
base
deriv 1
deriv 2
```

Note: Dynamic binding can only happen with pointer and reference variables.

Curvebase revisited

```
class Curvebase {
  public:
    Curvebase(double a = 0.0, double b = 1.0) :
                a_(a), b_(b);
    double x(double s); // Coordinates in arc length
    double y(double s);
    virtual ~Curvebase();
    // more members
  protected:
    double a_, b_;
    virtual double xp(double p);  // User parametrizati
    virtual double yp(double p);
    virtual double dxp(double p); // derivatives
    virtual double dyp(double q);
    double integrate(double p);
    // more members
};
```

Abstract Classes

- According to the language standard the Curvebase class as declared above must provide implementations of the virtual functions.
- This is, however, not what we want! These functions depend on the kind of curves and should, therefore, only be defined in the derived classes.
- In order to describe the interface which derived classes must implement without really defining the function in question, *pure virtual functions* are used in the base class:

```
virtual double xp(double p) = 0;
```

- A class with pure virtual functions is called *abstract*.
- An abstract class cannot be instantiated!

Hints

- The destructor of an abstract base class should always be virtual!
- The signature of virtual functions and the return type must be identical in the base class and all derived classes. (In fact, the return type may be slightly more general.)
- Debugging classes with dynamic binding can be extremly hard (simple typos can have far-reaching consequences).
- The C++11 standard contains means for a better control (`final`, `override`).
- The dynamic type of an object can be queried via `typeid(`*`expression`*`)`
- The counterpart of the static cast is the `dynamic_cast<`*`type`*`*>(pointer)`.

Overwrite Control

- A function declare `final` cannot be overwritten by a function in a derived class,

```
void integrate(double, double) const final;
```

- A function declared `override` shall overwrite a function of a base class,

```
double xp(double) override;
```

An Abstract Base Class For Boundary Curves

```
class Curvebase {
  public:
    Curvebase(double a = 0.0, double b = 1.0) :
                  a_(a), b_(b);
    double x(double s); // Coordinates in arc length
    double y(double s);
    virtual ~Curvebase();
    // more members
  protected:
    double a_, b_;
    bool rev;  // indication of curve orientation
    virtual double xp(double p) = 0;
    virtual double yp(double p) = 0;
    virtual double dxp(double p) = 0;
    virtual double dyp(double q) = 0;
    double integrate(double p);    // Need an implementation
                                   // in Curvebase!
                                    // Can be overwritten.
    // more members
};
```

Summary

- Derived classes and inheritance
- Dynamic binding
- Virtual functions and abstract classes

- What comes next:
  - Move constructors: Domains