Detektoren und Algorithmen zur Spurrekonstruktion geladener Teilchen

Sommersemester 2015

Vorlesung 3

Themen

- Grundlagen C++ Teil 1:
  - „Hallo Welt“
  - Variablen und Ablaufsteuerung
  - Umgang mit dem Compiler
- Grundlagen C++ Teil 2:
  - Funktionen und Parameter
  - Objektorientierte Programmierung
  - Klassen
- Grundlagen C++ Teil 3:
  - Vererbung und Polymorphie
- Detektorphysik
  - Detektortypen: Silizium, Vieldrahtproportionalkammer, TPC
  - Geometrien: Pixel und Streifen
- Monte-Carlo-Simulationen
  - Konzept
  - Erzeugung von „Zufall“
  - Detektorsimulation mit PandaRoot

Themen

- Spurrekonstruktion Teil 1:
  - Spurfindung
  - Hough-Transformation, Riemann-Oberflächen
  - Abweichungen von der „idealen“ Spur: Kleinwinkelstreuung und Energieverlust
- Spurrekonstruktion Teil 2:
  - Spuranpassung
  - Kalman-Filter
  - Spurpropagation in der MC-Simulation
- Spuranalyse Teil 1:
  - Betrachtung der Ereignistopologie
  - Exklusive und inklusive Rekonstruktion
  - Kinematische Fits, Vertexfits
- Spuranalyse Teil 2:
  - Kombination des Spurverlaufs mit weiteren Informationen
  - Impulsbestimmung, Teilchenidentifikation

Wiederholung

C++

Grundlagen Teil 3

Klassen - Elementfunktionen

Header File

```
1:  class komplexeZahl {
2:    public:
3:      komplexeZahl() {};
4:      komplexeZahl(double re, double im)
5:        : _re(re), _im(im)
6:        {};
7:      ~komplexeZahl() {};
8:      komplexeZahl(const komplexeZahl& rhs);
9:      komplexeZahl& operator=(const komplexeZahl& rhs);
10:     void Addiere(const komplexeZahl);
11:     double _re;
12:     double _im;
13: };
```

Implementation File

```
1: void komplexeZahl::Addiere(const komplexeZahl meineZahl){
2:   _re += meineZahl._re;
3:   _im += meineZahl._im;
4: }
```

(Part of) Main File

```
1: komplexeZahl a(1,2);
2: komplexeZahl b(2,3);
3: a.Addiere(b);
```

Operator Overloading

Every field has shorthand notation for frequently used concepts. E.g. x +y*z is much simpler than *"multiply y time z and add the result to x".* This is more conventional and convenient than the usual functional notation. For example:

```
class complex {  // very simple complex
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) { }
    complex operator+(complex);
    complex operator*(complex);
};
```

These operators are functions. If b and c are objects of type complex then b+c means b.operator+(c). We can also do things like:

```
complex c;
c = a*b + complex(1,2);
```

Mixed mode arithmetic: if we want to do *complex d = 2+b*; then we can overload the operators

```
class complex {
    double re, im;
public:
    complex & operator += (complex a) {    //member operators
        re  += a.re;
        im  += a.im;
        return  *this;
   }
    complex & operator += (double a) {
        re  += a.re;
        return  *this;
   }
   // …
};

                                              //non-member operators
complex operator+(complex a, complex b) {  // allow  c = a + b
    complex r = a;
    return  r  += b;  }  //calls complex::operator+=(complex)
complex operator+(complex a, double b) {    // allow   c = a  + 2.0
    complex r = a;
    return  r  += b;  }  //calls complex::operator+=(double)
complex operator+(double a, complex b) {     // allow   c = 2.0 + a
    complex r = b;
    return  r  += a;  }  //calls complex::operator+=(double)
```

Initialization: we can make the code more compact if we use default initialization:

```
class complex {
    double re, im;
public:
    complex() : re(0) , im(0)    { }
    complex(double r) : re(r) , im(0)    { }
    complex(double r, double i) : re(r) , im(i)    { }
    // ..
};
```

This becomes

```
class complex {
    double re, im;
public:
    complex(double r=0, double i=0) : re(r) , im(i)    { }
    // ..
};
```

This procedure becomes very tedious and error prone if we must make each variation explicitly. Thus, an alternative is to rely on conversion. For example our *complex* class has a constructor that converts a *double* to a *complex*. Thus we can do:

```
bool  operator == (complex,complex);

void f(complex x, complex y) {
    x == y;    // means operator==(x,y)
    x == 3;    // means operator==(x,complex(3))
    3 == x;    // means operator==(complex(3),x)
}
```

Freunde

Friends: an ordinary member function declaration specifies 3 things

1. The function can access the private parts of the class
2. The function is in the scope of the class
3. The function must be invoked on an object (has a *this* pointer)

By declaring a member function *static* it has only the first 2 properties. By declaring a *friend* it only has the first property.

For example: let us define an operator that multiplies a *Matrix* by a *Vector*. Each of these classes has a complete set of operations on objects of their type, but this operator belongs in both, but can not be a member of both. Also we do not want low-level access functions to allow everybody to read and write the data of both *Matrix* and *Vector*. To avoid this we declare the *operator** a friend to both

Friends Example

```
class Matrix;

class Vector {
        float v[4];
        // …
        friend Vector operator* (const Matrix&, const Vector&);
};
class Matrix {
        Vector v[4];
        // …
        friend Vector operator* (const Matrix&, const Vector&);
};

Vector operator* (const Matrix& m, const Vector& v )
{
        Vector r;
        for (int i=0; i<4; i++) {  // r[i] = m[i]*v;
            r.v[i] = 0;
            for (int j=0; j<4; j++)  r.v[i] += m.v[i].v[j] * v.v[j];
        }
        return r;
}
```

A friend declaration can be placed in private or public, it does not matter.
A member function of one class can be the friend of another:

```
class List_iterator {
    // …
    int*  next();
};

class List {
    friend int* List_iterator::next();
    // …
};
```

Often all of the member functions of one class should be friends to another. There is this shorthand notation:

```
class List {
    friend class  List_iterator;
    // …
};
```

Derived Classes

When we have many different types of objects they are often related in some sort of hierarchy.

For instance, if we have many different types of geometrical shapes, (point, circle, triangle, etc) they are logically all related, and in general there will be operations that we would like to perform on all kinds of shapes (for example draw).

We would lose essential information if we ignore the connections between objects.

Consider the classes *Employee* and *Manager*. A *Manager* is an *Employee*, but with additional information:

```
class Employee {
    string first_name, family_name;
    // …
public:
    void print() const;
    string full_name() const
       { return first_name + ‘ ‘ family_name; }
    // …
};

class Manager  :  public Employee  {
    // …
public:
    void print() const;
    // …
};
```

A member of a derived class can use the public (and protected: more on this later) members of its base class

```
void Manager::print() const
{
    cout << “name is “ << full_name() << endl;
    // …
}
```

However, a derived class cannot use a base class’ private members.

```
void Manager::print() const
{
    cout << “name is “ << family_name << endl;    //  ERROR !!!
    // …
}
```

Usually the cleanest solution is for the derived class to only use public members of its base class

```
void Manager::print() const
{
    Employee::print(); // Employee:: required since print() has been
                      // redefined in Manager
    cout << level; // print additional information
    // …
}
```

Default constructors are called implicitly. If all constructors of the base class require arguments, then it must be explicitly called:

```
class Employee  {
    string first_name, family_name;
    short department;
    // …
public:
    Employee (const string& n, int d);
    // …
};

class Manager : public Employee {
    short level;
    // …
public:
    Manager (const string& n, int d, int lvl);
    // …
};
```

Now when we create a Manager, we must pass n and d to the constructor of Employee:

In the class definition file:

```
Employee::Employee  ( const string& n, int d)
    :  family_name(n), department(d)     // initialize
members
{
    // …
}

Manager::Manager ( const string&n, int d, int lvl )
    : Employee(n,d) ,        // initialize base class
    level(lvl)               // initialize members
{
    // …
}
```

Class hierarchies:

A derived class can itself be a base class. For example:

```
class Employee  { /* …  */  };
class Manager :  public Employee  { /* …  */  };
class Director : public Manager  { /* …  */  };
```

This hierarchy is often a tree (as above), but can be more general

```
class Temporary  { /* …  */  };
class Secretary :  public Employee  { /* …  */  };
class Tsec : public Temporary, public Secretary  { /* …  */  };
class Consultant: public Temporary, public Manager { /* …  */  };
```

Virtual Functions allow functions that were defined in the base class to be redefined in a derived class

```
class Employee  {
    string first_name, family_name;
    short department;
    // …
public:
    Employee(const string& name, int dept);
    virtual void print() const;
};
```

A virtual function must be defined for the class in which it is first declared, unless it is pure virtual, e.g.: `virtual void print() const = 0;`

```
void Employee::print() const
   { cout << family_name << '\t' << department << endl;   }
```

A virtual function can be used even if no class is derived, and a derived version does not need its own version. If you want a different version provide it as usual.

Abstract classes: many classes can be used either by themselves or as a base class. Some classes such as *Shape* represent abstract concepts for which objects can not exist. A *Shape* makes only sense as a base class for a derived class.

```
class Shape {
public:
    virtual void rotate (int) = 0;      // pure virtual
    virtual void draw()    = 0;         // pure virtual
    virtual bool  is_closed() = 0;      // pure virtual
    // …
};
```

If one are all of the member functions are *pure virtual* then no object of this class can be created, and it is called an abstract class.

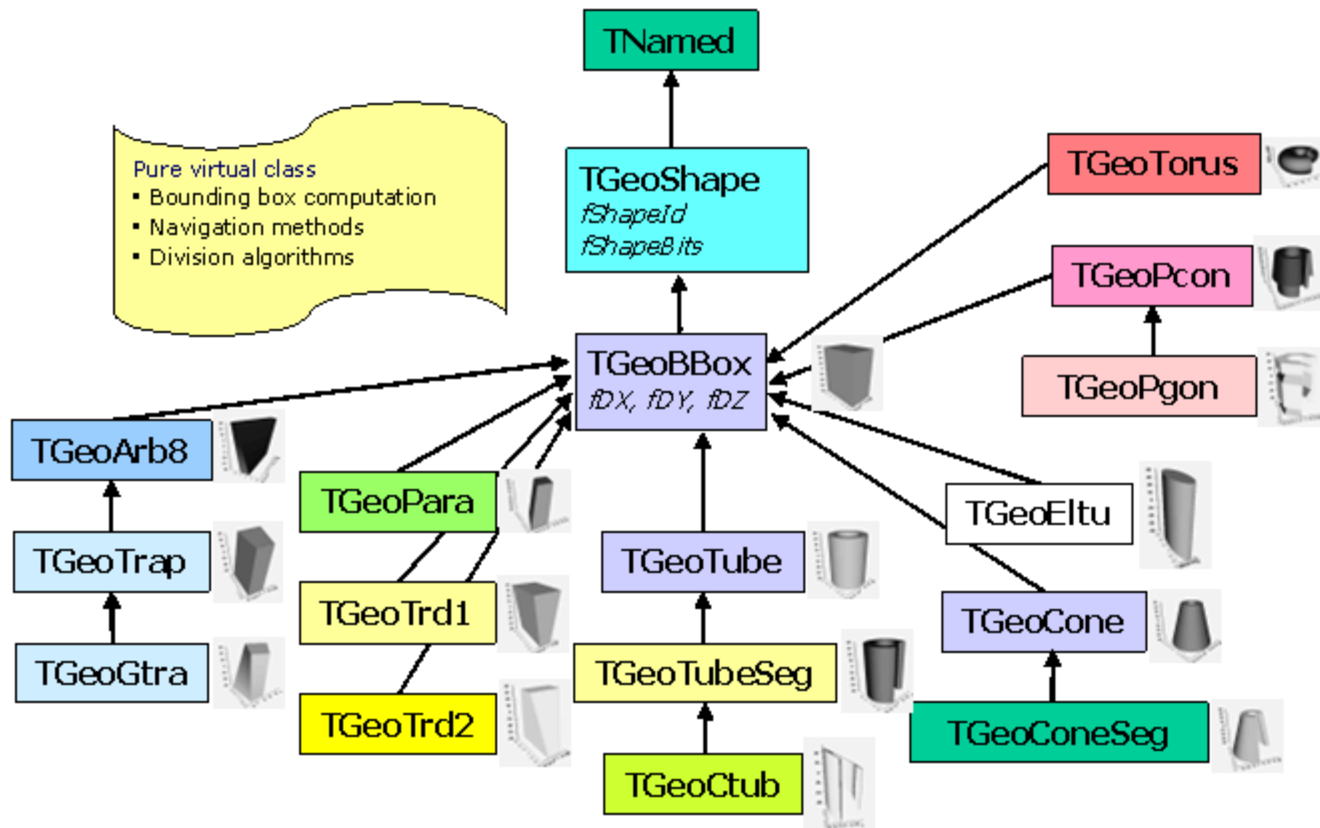
These can only be used as an interface and as a base for other classes.

```
Shape s; // ERROR, variable of abstract class Shape

class Point {  /* … */  };

class Circle: public Shape {
public:
    Circle (Point p, int r);  // constructor

    void rotate (int)  { }      // override Shape::rotate
    void draw ();                          // override Shape::draw
    bool is_closed()  {return true; } // override Shape::is_closed
private:
    Point center;
    int radius;
};
```

A pure virtual function that is not defined in a derived class remains pure virtual, so the derived class is also an abstract class. This allows us to build the implementation in steps.

```
class Polygon : public Shape {
public:
    bool is_closed() {return true;} // override Shape::is_closed
}; // still an abstract class because draw and rotate are not defined
class Irregular_polygon : public Polygon {
 // can override draw and rotate here, then no longer abstract
}
```

Shapes in ROOT



Simple ROOT Macro to draw a geometry

```
{
gSystem->Load(“libGeom”);
new TGeoManager(“world”, “the simplest geometry”);
TGeoMaterial *mat = new TGeoMaterial(“Vacuum”,0,0,0);
TGeoMedium *med = new TGeoMedium(“Vacuum”,1,mat);
TGeoVolume *top = gGeoManager->MakeBox(“Top”,med,10.,10.,10.);
TGeoVolume *tube = gGeoManager->MakeTube(“Tube”,med, 1.0, 1.5, 5.);
top->AddNode(tube, 1);
gGeoManager->SetTopVolume(top);
gGeoManager->CloseGeometry();
top->Draw();
}
```

Protected members of a class: in addition to *public:* and *private:,* there is the label *protected:* . Members of this type can be accessed within derived classes and friends.