Introduction to C++: MyVector and ODE_Solver

Knut–Andreas Lie

Dept. of Informatics, University of Oslo

Overview

1. Background: libraries and toolkits
2. Why object orientation / C++
3. Introduction to the C++ language: MyVector
4. More advanced topics: ODE_Solver

Tools for Building Simulators

- Standard programming tools:
  - programming languages (FORTRAN 77/90, C/C++, Java,..)
  - editors, debuggers and software analysis
  - version control systems
- Computer algebra programs
  - Maple, Mathematica, MathCad, Reduce, Axiom, ..
- Numerics:
  - Matlab, Mathematica, Maple, Scilab, Octave, ...
- Numerical libraries and toolkits:
  - a vast number available
- Visualisation
- Benchmark tests

Libraries and Repositories

- GAMS - guide to available mathematics software
  - `http://gams.nist.gov/`
  - catalogue and database of packages and libraries with tens of thousands of routines
  - the majority is numerical routines written in FORTRAN
- NETLIB - repository of free numerical software
  - `http://www.netlib.org/`
  - more than 160 libraries (BLAS, LAPACK, MPI, Atlas, ..)
- Numerical Recipes
  - `http://www.nr.com/`
  - book series (FORTRAN 77/90, C/C++,..)
  - algorithms and their (efficient) implementation
  - about 350 routines

Fundamental libraries: BLAS

BLAS - Basic Linear Algebra Subprograms:

- collection of portable, robust and efficient modules for elementary vector and matrix operations
- basis for many other routines, e.g., LAPACK (next slide)
- implemented in FORTRAN, but also used from C/C++/Java
- plug-and-play for numerical subroutines
- three levels:
  - Level 1: vector and vector-vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- see also ATLAS - Automatically Tuned Linear Algebra Software

Fundamental libraries: LAPACK

LAPACK - Linear Algebra PACKage

- popular collection of FORTRAN 77 subroutines for numerical linear algebra
  - linear systems, matrix decomposition, singular-value decomposition, ..
- dense and banded matrices (but not general sparse ones)
- fine-tuned for both single processors and supercomputers (memory access, block operations, etc)
- relies heavily on BLAS routines
- other variants
  - LAPACK90, JLAPACK, CLAPACK, LAPACK++
  - ScaLAPACK

Examples of toolkits

Diffpack:

- developed at SINTEF / Ifi, UiO
- object-oriented problem solving environment for PDEs

deal.II: A Finite Element Differential Equations Analysis Library

- developed at University of Heidelberg
- primarily targeted at adaptive finite elements and error estimation
- data structures and algorithms in C++

Clawpack - conservation law package

- developed by R.J. LeVeque, Univ. Washington
- finite volume routines for solving conservation laws
- written in FORTRAN

The choice of language

Traditional simulator software uses procedural programming

- subroutines
- data structures = variables, arrays
- data transferred between subroutines

typically coded in FORTRAN (or C).

New paradigm in late 1990s, object oriented languages (C++, Java, ..)

- data and operations on data grouped together in logical objects
- inheritance, hierarchies, polymorphism

The choice of language cont'd

Why stick to tradition?

- A lot of legacy codes!
- Very high efficiency
- A lot of procedural numerical libraries (Nag, Lapack, Eispack,..) available

Why not?

- large codes with too many visible details
- little correspondence between mathematical abstraction and computer code
- redesign and reimplementation tend to be expensive

My advice: use libraries whenever possible to avoid reinventing the wheel

Object-oriented numerics

Why is it suited for simulator codes (involving heavy number crunching)?

- High level of abstraction/modularity possible
  - hide irrelevant data/information
  - hide implementation details
  - stronger correspondence between math and code
- Extensive type-checking
- Localization
- Increased flexibility, code reuse, ...

Object-oriented numerics cont'd

However, one should be a bit careful to avoid

- low code efficiency
- the wrapper class explosion
- obfuscation of simple codes
- good designs with nothing inside

OO is a strong tool that should be utilized with great care!

It is the same as with fertilizer; without it, your crop grows slowly, but too much of it destroys your soil. It should be administred with great care

Introduction to OON in C++

As an example consider the implementation of a mathematical vector.

Surely, this must be available somewhere?

Oh yes! Vector classes are typically provided in (OO) numerical libraries and there will seldom be a need for writing new vector classes yourself. But this example can be used to give a hands-on feeling with many concepts:

- see how (numerical) C++ classes are programmed
- exemplify many different aspects of the C++ language
- exemplify efficiency considerations

Advice: get a good book on C++!

A mathematical vector

- What is a vector?

  A well defined mathematical quantity consisting of a set of elements $(v_1, ..., v_n)$ and a set of legal operations.

- Do standard arrays in any computer language give good enough support for matrices?

  No, for several reasons:

  - explicit transfer of length
  - explicit transfer of storage format
  - no software definition of legal operations
  - user implementented consistency checks

My first vector class

A class MyVector should do the following:

- Create vectors of length n: `MyVector v(n);`
- Create a vector with zero length: `MyVector v;`
- Redimension a vector to length n: `v.redim(n);`
- Create a vector as a copy of another vector w: `MyVector v(w);`
- Extract an entry: `double e = v(i);`
- Assign a number to an entry: `v(j) = e;`
- Extract the length of the vector: `int n = v.size();`
- Set two vectors equal to each other: `w = v;`
- Take the inner product of two vectors:
  `double a = w.inner(v);`
  or alternatively `a = inner(w,v);`
- Write a vector to the screen: `v.print(cout);`

We define the proposed syntax through functions in class MyVector

The MyVector class: <MyVector.h>

```
class MyVector
{
private:
  double* A;                          // vector entries (C-array)
  int     length;                     // length of vector
  void    allocate (int n);           // allocate memory, length=n
  void    deallocate ();              // free memory
public:
  MyVector ();                        // MyVector v;
  MyVector (int n);                   // MyVector v(n);
  MyVector (const MyVector& w);       // MyVector v(w);
 ~MyVector ();                        // clean up dynamic memory
```

Declaration of the class MyVector: internal stuff and constructors

- private: not accessible from outside MyVector
- public: accessible from outside MyVector

The MyVector class cont'd

```
int  redim (int  n);                              // v.redim(m);
int  size  ()  const { return length ; }          // n = v.size ();

void operator= (const MyVector& w);               // v = w;
double operator() (int  i ) const;                // a = v(i );
double& operator() (int i );                      // v(i ) = a;

void print  (ostream& o) const;                   // v. print (cout);
double inner (const MyVector& w) const; // a = v.inner(w);
};
```

Declaration of methods in MyVector

- member functions (print, inner, redim, size)
- operator overloading ($=$ and $()$)

Constructors

Constructors tell how we declare a variable of type `MyVector` and how this variable is initialized:

```
MyVector v;    // declare a vector of length 0
```

means calling the function

```
MyVector::MyVector ()
{ A = NULL; length = 0; }
```

Result: empty vector

Constructors II

```
MyVector v(n);   // declare a vector of length n
```

means calling the function

```
MyVector::MyVector (int n)
{ allocate (n ); }

void MyVector::allocate ( int n)
{
  length = n;
  // create n doubles in memory
  A = new double[n];
}
```

Destructor

A `MyVector` object is created (dynamically) at run time, but must also be destroyed when it is no longer in use. The destructor specifies how to destroy the object:

```
MyVector::~MyVector ()
{ deallocate (); }

// free dynamic memory:
void MyVector::deallocate ()
{
  delete [] A;
}
```

The assignment operator

To be able to use assignments of the form

```
v = w;          // v and w are MyVector objects
```

we *overload* the = operator:

```
void MyVector::operator= (const MyVector& w) // v = w;
{
  redim (w.size ());  // make v as long as w
  int i;
  for (i = 0; i < length; i++) // C arrays start at 0
    A[i] = w.A[i];
}
```

Redimensioning the length

```
v.redim(n);   // make v a vector from 1 to n

int MyVector::redim (int n)
{
   if (length == n)
     return 0;   // no need to allocate anything

   if (A != NULL) {
     // " this " object has already allocated memory
     deallocate ();
     }
   allocate (n);
   return 1;    // the length was changed
}
```

The copy constructor

We wish to create one vector as a copy of another

```
MyVector v(w);  // take a copy of w
```

This can be done as follows

```
MyVector::MyVector (const MyVector& w)
{
  allocate (w.size ());   // length equal w's length
  *this = w;              // call operator=
}
```

Here:

- `this` is a pointer to the current (“this”) object,
- `*this` is the object itself

The `const` concept

```
const int m=5; // not allowed to alter m

MyVector::MyVector (const MyVector& w)
// w cannot be altered inside this function
// & means passing w by _reference_

MyVector::MyVector (MyVector& w)
// w can be altered inside this function, the change
// is visible from the calling code

int MyVector::redim (int n)
// a local _copy_ of n is taken,
// changing n inside redim
// is invisible from the calling code


void MyVector::print (ostream& o) const
// member function does not alter members of class
```

Subscripting in vector

We use “( )” instead of “[ ]” for generality[1].

```
// a and v are MyVector objects; want to set
a(j) = v(i+1);

// the meaning of a(j) is defined by
inline double& MyVector::operator() (int i)
{
  return A[i-1];
  // base index is 1 (not 0 as in C/C++)
}
```

inline functions: function body is copied to calling code, no overhead of function call!

[1] In 2D: `a[i,j]` is a valid expression, but it calls `operator[]()` and evaluates the expression `i,j`.

Inlining

```
for (int i = 1; i <= n; i++)
  c(i) = a(i)*b(i);

// compiler inlining translates this to:
for (int i = 1; i <= n; i++)
  c.A[i-1] = a.A[i-1]*b.A[i-1];

// compiler should optimize this to
double* ap = &a.A[0];
double* bp = &b.A[0];
double* cp = &c.A[0];
for (int i = 0; i < n; i++)
  cp[i] = ap[i]*bp[i];          // pure C!
```

Inlining and complete control with the definition of `v(i)` allow

- safe indexing
- efficiency as in C or Fortran

Add safety checks

```
inline double& MyVector::operator() (int i)
{
  #ifdef SAFETY_CHECKS
   if (i < 1 || i > length)
  cout << "MyVector::operator(),␣ illegal ␣index,␣i=" << i ;
  #endif
  return A[i-1];
}
```

Compile with the `-DSAFETY_CHECKS` flag

Printing the vector

```
// MyVector v
cout << v;

inline ostream& operator<< (ostream& o, const MyVector& v)
{ v.print(o); return o; }


void MyVector::print (ostream& o) const
{
  int i;
  for (i = 1; i <= length; i++)
  o << "(" << i << ")=" << (*this)(i) << '\n';
}
```

Inner product[1]

```
double a = v.inner(w);

double MyVector::inner (const MyVector& w) const
{
  int i; double sum = 0;
  for (i = 0; i < length; i++)
  sum += A[i]*w.A[i];
  // alternative: sum += (*this)(i)*w(i);
  return sum;
}

double inner (const MyVector& v, const MyVector w)
{
  return v.inner(w);
}
```

[1] Without safety checks, of course... Add them yourself.

Norm

```
double a = v.norm();

double MyVector::norm () const
{
  int i ; double sum = 0;
  for ( i = 0; i < length ; i++)
  sum += A[i]*A[i ];
  return sqrt (sum);
}

double MyVector::norm() const { return sqrt(inner(this ));}
```

Operator overloading

We can redefine the multiplication operator to mean the inner product of two vectors:

```
double a = v*w;  // example on attractive syntax

class MyVector
{
   ...
  double operator* (const MyVector& w) const;
   ...
};

double MyVector::operator* (const MyVector& w) const
{
  return inner(w);
}
```

What about $+$, $-$, ... ?

Why have we not defined the following operations?

```
z = v + w;        z = va;
z = v - w;        z = av;
```

```
MyVector MyVector::operator+(const MyVector& w)
{
  int n=this->size();
  if (n != w.size ()) .....

  MyVector x(n);
  for (int i=1; i<=n; i++)
    x(i) = (*this)(i) + w(i);
  return x;
}
```

How would you implement $z = av$?

What about $+$, $-$, ... ?

What happens here in the `operator+(..)` call?

- allocate x
- x = v + w
- allocate tmp
- tmp = x
- z = tmp

Hmmm... this is hardly very efficient for large vectors!

What would you do?

Lessons learned

- Low-level C++/C programming involves a lot of intricate details (e.g. pointers, memory handling)
- Rely on ready-made tools (e.g. for arrays)
- Use libraries!
- Object-orientation hides low-level details that dominate in C and F77 programs

Evaluation of MyVector design

Some critical questions at the end:

- Have we made a flexible design?
- Can we reuse the code in other settings?
- What about vectors of integers, complex numbers, ...?
- What about arrays that are not vectors?

An answer to these questions may be the use of

- Parametrized types
- Layered design, inheritance

Parametrized types

We wish to have a family of vectors

```
MyVector(int)      n(245);
MyVector(float)    v(19);
MyVector(double) w(11);
MyVector(Complex) z(3);
```

In C++ this is achieved by class templates, i.e., a template for producing classes:

The MyVector<T> class template

```
template <class T>
class MyVector
{
private:
  T*      A;                                    // vector entries (C-array)
  :
public:
  :
  MyVector (const MyVector<T>& w);              // MyVector<T> v(w);
  MyVector ();                                  // clean up dynamic memory
  :
  void operator= (const MyVector<T>& w); // v = w;
  T  operator() (int  i ) const;                // a = v(i );
  T& operator() (int  i );                      // v(i ) = a;
  :
  T inner (const MyVector<T>& w) const; // a = v.inner(w);
  T norm () const ;                             // a = v.norm();
};
```

Layered design

Recall definition of a vector:

> A well defined mathematical quantity consisting of a set of elements $(v_1, ..., v_n)$ *and a set of legal operations.*

On the other hand, an array is defined as:

> A collection of elements, each an object of some fixed type, and an associated dimension.

In other words,

> a vector is a special array (in one dimension) with extra functionality.

This should be reflected in our design...

New array class template MyArray<T>

```
template <class T> class MyArray
{
protected:
  T*      A;                              // vector entries (C-array)
  :
public:
  MyArray ();                             // MyArray<T> v;
  MyArray (int n);                        // MyArray<T> v(n);
  MyArray (const MyArray<T>& w);          // MyArray<T> v(w);
 ~MyArray ();                             // clean up dynamic memory
  int redim (int n);                      // v.redim(m);
  int size () const { return length; }    // n = v.size ();
  void operator= (const MyArray<T>& w);   // v = w;
  T  operator() (int i) const;            // a = v(i);
  T& operator() (int i);                  // v(i) = a;
  void print (ostream& o) const;          // v.print(cout);
};
```

MyVector<T> derived from MyArray<T>

```
template <class T> class MyVector : public MyArray<T>
{
public:
   MyVector ();                              // MyVector<T> v;
   MyVector (int n );                        // MyVector<T> v(n);
   MyVector (const MyVector<T>& w);          // MyVector<T> v(w);
  ~MyVector ();                              // clean up dynamic memory

   void operator= (const MyVector<T>& w); // v = w;
   T inner (const MyVector<T>& w) const; // a = v.inner(w);
   T norm () const ;                         // a = v.norm();
};
```

Here `MyVector` inherits the data members and all the member functions from `MyArray` and provides extra functionality.

Constructors, destructors, and assignment

These are not inherited and receive special treatment. C++ automatically defines these functions unless we declare them. Constructors:

- First invoke constructor for each subobject
- Then construct members

In our case, invoke constructor for `MyArray` and do nothing else:

```
template<class T>
MyVector<T>::MyVector(int n) : MyArray<T>(n) {}
```

Questions from the audience

Q: Referring to operator overloading of $+$; wouldn't the following construction solve the problem?

```
&MyVector MyVector::operator+(const MyVector& w)
```

A: No! What would the & refer to? It is not possible to return reference to a local variable. Neither is the address of the temporary memory allocated by `return` available.

Q: Can I restrict the types of T in a template class?

A: It is possible to do using various tricks such as static assertions.

Questions from the audience cont'd

Q: Is it possibly to define my own operators to "overload"?

A: The only operators that may be overloaded are the following

+	-	*	/	%	
ˆ	&	|	˜	!	&&
||	++	−−	$\ll$	$\gg$	,
<	<=	==	!=	>	>=
=	+=	-=	*=	/0	%=
&=	|=	ˆ=	$\ll$=	$\gg$=	
[]	()	->	new	delete	

Book recommendations

- Timothy Budd: C++ for Java Programmers. Addison-Wesley. 1999.
- Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. (C++ In-Depth Series). Addison-Wesley.
- Essential C++. Stanley B. Lippman. Addison Wesley.

ODE Solver Environment

The purpose of this example:

- OO-design for a simple problem
- Continue the overview of C++ features
- Principles apply to advanced simulations

Mathematical problem:

$$\frac{dy_i}{dt} = f_i(y_1, \ldots, y_n, t), \quad y_i(0) = y_i^0, \quad i = 1, \ldots, n$$

- A system of ordinary differential equations (ODEs).
- A plenitude of numerical methods exist.
- Occurs frequently in numerics

Traditional procedural solution

Typical interface in FORTRAN77:

```
SUBROUTINE RK4(Y,T,F,WORK1,N,TSTEP,TOL1,TOL2,...
```

Here:

- `Y` is current y
- `T` is the value of t
- `F` is a function defining the f_i's
- `WORK1` is a work array (since runtime allocation is not allowed in F77)
- `TSTEP` is the current time step
- `TOL1, TOL2,..` are various parameters

FORTRAN 77 cont'd

For a specific ODE:

$$\ddot{y} + c_1(\dot{y} + c_2\dot{y}|\dot{y}|) + c_3(y + c_4 y^3) = \sin \omega t$$

Written as a system ($\dot{y}_1 = y_2$, $\dot{y}_2 = \ddot{y}$)

$$\begin{aligned} f_1 &= y_2, \\ f_2 &= -c_1(y_2 + c_2 y_2|y_2|) - c_3(y_1 + c_4 y_1^3) + \sin \omega t \end{aligned}$$

Possible FORTRAN function

```
      SUBROUTINE F(YDOT,Y,T,C1,C2,C3,C4,OMEGA)
```

Unfortunately this is problem dependent and cannot be used. Instead,

```
      SUBROUTINE F(YDOT,Y,T)
```

with `C1,C2,C3,C4,OMEGA` transferred in `COMMON` blocks

$\Longrightarrow$ dangerous side effects and possible hidden bugs

Object-oriented design

introduce two class hierarchies:

- `ODE_Solver` (e.g., forward Euler, Runge–Kutta , ..)
  Base class with:
  - initialisation of solver
  - virtual function `advance` for one time step
- `ODE_Problem` (e.g., Newton's 2.law, HIV-1 virus, ..)
  Base class with:
  - the generic system - virtual function `equation`
  - a driver function `timeLoop`
  - common data: Δt, T, name of solver,..

`ODE_Solver` must access `ODE_Problem` and vice versa

Implementation in C++

```
class ODE_Problem;     // tell C++ that this class name exists

class ODE_Solver
{
protected:                     // members only visible in subclasses
  ODE_Problem* eqdef;          // definition of the ODE in user's class

public:                        // members visible also outside the class
  ODE_Solver (ODE_Problem* eqdef_)
    { eqdef = eqdef_; }
  virtual ~ODE_Solver () {}    // always needed, does nothing here...
  virtual void init () {}      // initialize solver data structures
  virtual void advance (MyArray<double>& y,
                        double& t, double& dt);
};
```

Notice: `protected` means *public* access for derived classes and *private* access for others.

The ODE_Problem class

```
class ODE_Solver;       // tell C++ that this class name exists

class ODE_Problem
{
protected:
  ODE_Solver*        solver;      // some ODE solver
  MyArray<double> y, y0;          // solution (y) and initial cond. (y0)
  double             t, dt, T;    // time loop parameters
public:
  ODE_Problem () {}
  virtual ~ODE_Problem ();
  virtual void timeLoop ();
  virtual void equation (MyArray<double>& f,
                         const MyArray<double>& y, double t);
  virtual int  size ();    // no of equations in the ODE system
  virtual void scan ();
  virtual void print (ostream& os);
};
```

Virtual functions

C++ virtual function are a member functions, whose functionality may be overridden in derived classes

- Nonvirtual C++ member functions are resolved at compile time; this mechanism is called *static binding*.
- Virtual member functions are resolved during run-time; this mechanism is known as *dynamic binding*.

Here:

- `equation` and `size` only in subclasses
- `scan` and `print` extended in subclasses
- `timeLoop` can be written in base class

The `timeLoop` function

```
void ODE_Problem:: timeLoop ()
{
  ofstream outfile ("y.out");
  t = 0;   y = y0;
  outfile << t << "␣"; y.print ( outfile ); outfile << endl;
  while (t <= T) {
    solver->advance (y, t, dt);
    outfile << t << "␣"; y.print ( outfile ); outfile << endl;
  }
}
```

This function is *generic* and can be coded in the base class.

One specific problem — oscillator

$$\dot{y}_1 = y_2, \qquad \cot y_2 = -c_1(y_2 + c_2 y_2|y_2|) - c_3(y_1 + c_4 y_1^3) + \sin \omega t$$

```
class Oscillator : public ODE_Problem
{
protected:
  double c1, c2, c3, c4, omega;      // problem dependent paramters
public:
  Oscillator () {}
  virtual void equation (MyArray<double>& f,const MyArray<double>& y, double t);
  virtual int  size () { return 2; }
  virtual void scan ();
  virtual void print (ostream& os);
};
```

We inherit `y, y0, t, dt, T` and functions from `ODE_Problem`.

`timeLoop` can be reused and does not appear explicitly

Oscillator cont'd

```
void Oscillator::equation(MyArray<double>& f,
                          const MyArray<double>& y, double t)
{
  f(1) = y(2);
  f(2) = -c1*(y(2)+c2*y(2)*fabs(y(2))) - c3*y(1)*(1.0+c4*y(1)*y(1))
         + sin(omega*t);
}
```

A specific solver – ForwardEuler

$$y_i^{\ell+1} = y_i^\ell + \Delta t f(y_1^\ell, \ldots, y^\ell, n, t_m)$$

```
class ForwardEuler : public ODE_Solver
{
  MyArray<double> scratch1;  // needed in the algorithm
public:
  ForwardEuler (ODE_Problem* eqdef_);
  virtual void init ();   // for allocating scratch1
  virtual void advance (MyArray<double>& y, double& t, double& dt);
};

void ForwardEuler::advance(MyArray<double>& y, double& t, double& dt)
{
  eqdef->equation (scratch1, y, t);   // evaluate scratch1 (as f)
  const int n = y.size ();
  for (int i = 1; i <= n; i++)
    y(i) += dt * scratch1(i);
  t += dt;
}
```

Another solver — RungeKutta4

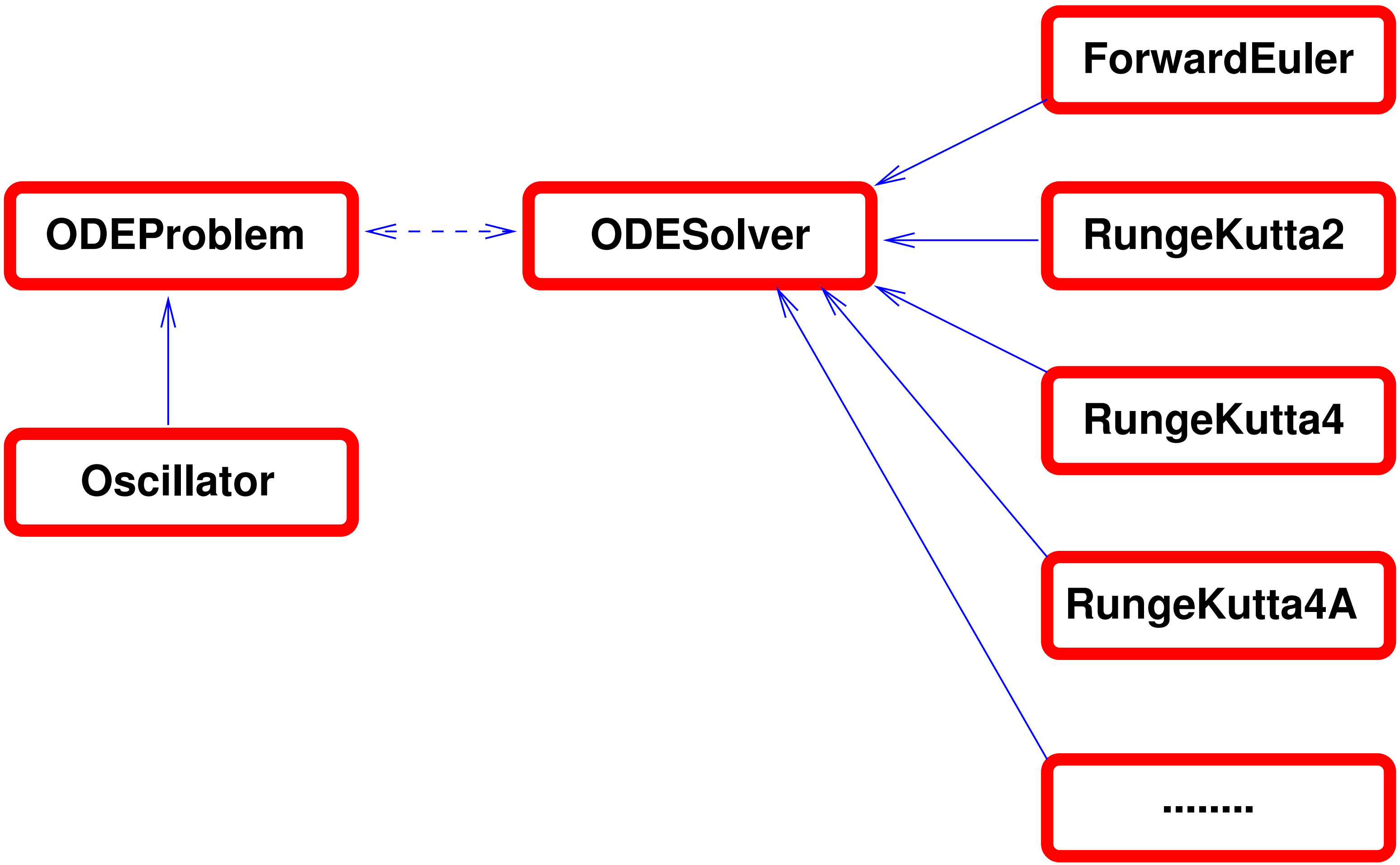
$$
\begin{aligned}
y_i^{\ell+1/4} &= y_i^{\ell} + \frac{\Delta t}{2} f(y^{\ell}, t_m) \\
y_i^{\ell+2/4} &= y_i^{\ell+1/4} + \frac{\Delta t}{2} f(y^{\ell+1/4}, t_m + \Delta t/2) \\
y_i^{\ell+3/4} &= y_i^{\ell} + \Delta t f(y^{\ell+2/4}, t_m + \frac{\Delta t}{2}) \\
y_i^{\ell+1} &= y_i^{\ell} + \frac{\Delta t}{6} \Big[f(y^{\ell}, t_m) + 2 f(y^{\ell+1/4}, t_m + \frac{\Delta t}{2}) \\
&\qquad + 2 f(y^{\ell+2/4}, t_m + \frac{\Delta t}{2}) + f(y^{\ell+3/4}, t_m + \Delta t) \Big]
\end{aligned}
$$

```
class RungeKutta4 : public ODE_Solver
{
  MyArray<double> y1, y2, y3; // needed in algorithm
public:
  RungeKutta4 (ODE_Problem* eqdef_);
  virtual  void  init   ();
  virtual  void advance (MyArray<double>& y, double& t, double& dt);
};
```

RungeKutta4 cont'd

```
void RungeKutta4::advance(MyArray<double>& y,double& t, double& dt) {
  const double dt2 = 0.5*dt;
  const double dt6 = dt/6.0;
  const int n = y.size (); int i;
  eqdef->equation (y1, y, t);
  for (i = 1; i <= n; i++)
    y2(i) = y(i) + dt2 * y1(i);
  eqdef->equation (y1, y2, t+dt2);
  for (i = 1; i <= n; i++)
      y2(i) = y(i) + dt2 * y1(i);
  eqdef->equation (y3, y2, t+dt2);
  for (i = 1; i <= n; i++) {
      y2(i) = y(i) + dt * y3(i);
      y3(i) = y1(i) + y3(i );}
  eqdef->equation (y1, y2, t+dt);
  eqdef->equation (y2, y, t);
  for (i = 1; i <= n; i++)
    y(i) = y(i) + dt6*(y1(i) + y2(i) + 2*y3(i ));
  t += dt;
}
```

The class hierarchy



How to connect and initialize?

```
class ODE_Solver_prm {
public:
  char          method[30];  // name of subclass in ODESolver hierarchy
  ODE_Problem* problem;      // pointer to user's problem class
  ODE_Solver* create ();     // create correct subclass of ODESolver
};

ODE_Solver* ODE_Solver_prm::create () {
  ODE_Solver* ptr = NULL;
  if      (strcmp(method, "ForwardEuler") == 0)
    ptr = new ForwardEuler (problem);
  else if (strcmp(method, "RungeKutta4") == 0)
    ptr = new RungeKutta4 (problem);
  else {
    cout << "\n\nODESolver_prm::create:\n\t"
         << "Method␣" << method << "␣is␣not␣available\n\\n";
    exit (1); }
  return ptr;
}
```

Initialisation of ODE_Problem

```
void ODE_Problem:: scan ()
{
  const int n = size ();    // call size in actual subclass
  y.redim(n);   y0.redim(n);
  cout << "Give␣" << n << "␣ initial ␣conditions:␣";
  // y0.scan(cin);
  int  i;
  for(i=1; (i<=n) && (cin>>y0(i)); i++);
  cout << "Read␣" << i-1 << "elements";
  y0.print(cout); cout<<endl;
  cout << "Give␣time␣step:␣";     cin >> dt;
  cout << "Give␣final␣time␣T:␣"; cin >> T;
  ODE_Solver_prm solver_prm;
  cout << "Give␣name␣of␣ODE␣solver:␣";
  cin >> solver_prm.method;
  solver_prm.problem = this;
  solver = solver_prm.create();
  solver->init();
  // more reading in user's subclass
}
```

Initialization of a specific problem

```
void Oscillator :: scan ()
{
  // first we need to do everything that ODE_Problem::scan does:
  ODE_Problem::scan();
  // additional reading here:
  cout << "Give␣c1,␣c2,␣c3,␣c4,␣and␣omega:␣";
  cin >> c1 >> c2 >> c3 >> c4 >> omega;
  print (cout ); // convenient check for the user
}
```

... and finally the main program

```
#include "Oscillator.h"

int main (int argc, const char* argv[])
{
   Oscillator problem;
  problem.scan();        // read input data and initialize
  problem.timeLoop();    // solve problem
}
```

Conclusion:

- a flexible ODE library
- introduction to general code design principles
- faster start for the next ODE we wish to solve....