C++ object oriented programming for scientific computing

Jamshed Anwar
Institute of Pharmaceutical Innovation
University of Bradford
C++ object oriented programming for scientific computing

- Problems with traditional structured programming
- Object oriented programming (OOP)
- Objects: e.g. CAtom
- OOP languages for scientific computing & OOP features of C++
- Atomh++: Objects for molecular simulation
- Elegance & efficiency
- ‘Unbiased’ assessment of C++ OOP
- Testing of code
- Concluding remarks
Problems with traditional structured programming

Mental image of subroutine interaction
Difficult to grasp;
Difficult to retain over an extended period.
Problems modifying and maintaining code

- 20,000 lines Hardwork!
- 50,000 lines Nightmare
- 100,000 lines Impossible;
  lose confidence

Graphics, games etc less of a problem

Large number of variables (typically reals) being passed in arguments → scope for errors

Defects rates 5-6/1000 lines in production code
50-100 defects/1000 lines in new programs

Large (esp. number-crunching) codes → object-oriented
Object-oriented programming (OOP)

Model system close to ‘tangible’ reality
Easier to comprehend and retain in memory

Traditional programming
VERB-based
Emphasis on doing (action)
e.g. subroutine invert_matrix()

OOP
NOUN-based Emphasis on object
object Matrix
{
    function invert()
    function transpose()
    function multiply(Matrix2)
    ..
}

Noun: word referring to person, place or thing
Verb: word expressing idea of action or being

What comprises an object?
1. Attributes that define the object => data members
2. What can the object do? => functions
Object CAtom

// Variables defining characteristics
int Index
string Label
double Charge
double Mass
bool IsFrozen
CVectorDouble R
CVectorDouble V
CVectorDouble F

// Behaviour/action/functions/procedure
Get..
Set..
BondAngle(CAtom2&, CAtom3)
TranslateTo(R)
TranslateBy(R)
VerletStep(timestep)
WritePDB
Object-oriented programming: Languages

- **C++**
  Numerous features; unnecessary complexity; operator overloading

- **Java**
  Designed to minimise run-time errors
  ‘Subset’ of C++ (?)
  All memory allocated dynamically; garbage collection

  No operator overloading (Java Grande → no progress)
  cannot do MatrixC = MatrixA * MatrixB * VectorV
  instead MatrixC = MatrixA.multiply( MatrixB.multiply(VectorV) )

  Multi-threading is part of language
  Built-in graphics (front end easy to develop)

- **Smalltalk**
  Not for scientific codes

- **C# (?)**
Objects

Encapsulation & implementation hiding
Data and functions cannot be accessed without object; Less chance of changes in one part of code causing inadvertent problems elsewhere; Strong type-casting;
First encounter: difficult to get objects to talk!
Separate implementation from interface
Can change implementation without changes to interface

Polymorphism
Same name for functions → consistent interface
function force(Atom1, Atom2, cutoff)
function force(Atom1, Molecule2, cutoff)
function force(Molecule1, Molecule2, cutoff)
OOP features of C++ (2 of 3)

- **Inheritance**
  - Reuse code – extend code
  
  CMolecule
  
  CRigidMolecule  CFlexibleMolecule
  
  CPartiallyRigidMolecule

- **Operator overloading**
  - Reuse code – extend code
  
  +  -  *  /  MatrixC = VectorA * MatrixB

- **Dynamic memory allocation**
  - Complicated (messy!); No garbage collection: need to explicitly delete allocated memory
Templates

Same code, different object
Minimise code

CVector<int>
CVector<double>
CVector<bool>

V<> function + (V2<>)
{
    CVector<> VT
    for (i=0; i<size; i++) VT.E[i] = E[i] + V2.E[i];
    return VT;
}
Molecular dynamics simulation

Simulate time evolution of a system of atoms/molecules

\[ \mathbf{f}_i = -\sum_j \nabla_{\mathbf{r}_i} U(\mathbf{r}_{ij}) \]

\[ \mathbf{r}_i(t + \Delta t) = 2\mathbf{r}_i(t) - \mathbf{r}_i(t - \Delta t) + \frac{\mathbf{f}_i(t)}{m_i} \Delta t \]

Limitations

Periodic boundaries & minimum image convention

NVE, NVT, NPT & N\(\sigma\)T

Employ extended Lagrangian e.g. \(L(\mathbf{r}, \mathbf{p}, \mathbf{H}, s)\)

1. Initial configuration
2. Equilibration
3. Production (averages)

NPT ↓ G

Limited system size

Cpu time (100ps -> 10ns)

Accuracy of interaction potential
Interaction potential

\[ U = \sum_{i<j} \sum 4\varepsilon_{ij} \left[ \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] + \sum_{\text{bonds}} \frac{1}{2} k_b (r - r_0)^2 \]

\[ + \sum_{\text{angles}} \frac{1}{2} k_a (\theta - \theta_0)^2 \]

\[ + \sum_{\text{torsions}} k_\phi \left[ 1 + \cos(n\phi - \delta) \right] \]

Ball & spring

Parameters

empirical; from experiment
and optimised

Pair-interactions
N(N-1)/2

LJ: short ranged \( R_c \)

qq: long-ranged
(Ewald summation)
Atom.h++: objects for molecular simulation

- CAtom
- CMolecule
- CMolecularSpecies
- CMolecularEnsemble
- CThermostat
- CCell
- CForcefield
- CMatrix
- CVector
- CBarostat
- Bond
- Angle
- Torsion
- Constraint
CMolecule object

Data members
{
    Name
    Index
    CAtoms[] // array of atoms
    CentreOfMassR
    Mass
    DegreesOfFreedom
    CBonds[]
    CAngles[]
    CTorsions[]
    CBondConstraints[]
...
}

Functions
{
    Align()
    Force()
    Force(Molecule2)
    Energy()
    GetAtom(index)
    GetCoordinates()
    SetVelocity(V)
    IncrementVelocity(dV)
    TranslateTo(R)
    TranslateBy(dR)
    RotateTo(..)
    RotateBy(..)
    WritePDB(file)
    ....
}

High memory requirement
Million particle systems
CSimulationCell object

Data members
{
    CMolecularEnsemble
    CCell
    CForcefield
    CBarostat
    CThermostat
    CMonitor
    CController
    ....
}

Functions
{
    Energy
    Force()
    ConstraintForce()
    Verlet..()
    MCCycle()
    MC()
    MD()
    ChemicalPotWidom()
    ...
    MC_ThermodynamicInt.
    MD_ThermodynamicInt.
    WriteConfig()
    ....
}
Dynamic memory allocation

Memory leaks: allocation of memory but no deletion

Object Vector Double* E=new double [Size] Vectors A & B
Copy object operation A = B {A.E = B.E}
Since E is a pointer, A.E → B.E
Pointer E of object A is pointing to the same memory location as that pointer E of object B

The original memory location of A.E has been dereferenced.
Delete B → delete B.E
Delete A → delete B.E (again) ERROR
Solution: Explicit copy constructor

SGI Irix 5.1 (1993/94)
Indy with 16MB memory: completely unusable in 3-4 days. Transformed R4000 processor into an intel 386SX.
SGI solution: give away free additional memory

Tools/code to check heap before & after running code
// Advance velocity to V(t+0.5dt)
Velocity_fDT = Velocity_bDT + (Timestep * Force)/Mass;

// Advance position R(t+dt) using new velocity
R_fDT = R + Timestep * Velocity_fDT;

// Calculate Velocity(t)
Velocity = CConstant::HALF * (Velocity_fDT +Velocity_bDT);
Citation Classics

Sheldrick G M. *SHELX76, program for crystal structure determination*. Cambridge, England: University of Cambridge, 1976. (Computer program.)

→ 4260 citations! SCI 1989


→ cited 2,870 times  ISI
Obscure programming by design: Achieving immortality in SCI rankings

(a) Program be robust; produce sensible numbers even when used for purposes for which it was not intended by someone who has lost the instructions (if there were any).

(b) “Comments” in a program and “structured programming” are superfluous and make it easy for users to “improve” the program and re-issue it as their own.

(c) Never publish the original algorithms employed (if any), or you will encourage cheap imitations.

(d) Make sure that the program contains one or two undocumented “features” or even “bugs” → user dependency and expectation of getting final/enhanced version will encourage users to cite you.

(e) By definition, the final version is always six months from completion, and so it never can be released.

Current Contents, 41, ISI, Oct 9, 1989
Scatter ‘do-nothing code’ throughout the program. Define some important looking variables, alter their contents in if-statements and within loops.

→ Your signature

Post copies (x2) of the code (at significant stages of development) to yourself by special delivery. Keep certificates of posting and DO NOT open the packages until the lawyers need to,
Efficiency

Speed of execution
C and C++ should be identical – a design specification
Java ~80-90% of C/C++; in principle Java could be faster due lack of pointers and garbage collection.

Memory usage
To comply with the idea of objects, all variables defining the object’s characteristics need to be defined in the class. Every molecule has info (variables) re atoms involved in bonds, angles and torsions; some variables may be redundant for a particular application; No scope for using variables transiently. → High memory requirement
Object-oriented programming: ‘unbiased’ assessment

Code is significantly more accessible
Easy to maintain & modify; Confidence

Major design issues
Spend 5 days pondering;
Implement in ½ day!

Obsession with elegance →
  break design (again & again!)

Not good for small codes.
Superfluous code: functions coded for complete
description may never be used

Steep learning curve
> 1 year to write elegant (intuitive) code
Testing of code (1 of 2)

```fortran
do 10 I=1,10
   a(I) = b(I,10) * sqrt(..)
C    write(5,*) print a(I);
    ....
10 continue
```

At some later stage, may even remove commented line!

Correct behaviour before efficiency
Testing of code (2 of 2)

Philosophy: test code is part & parcel of production code

Follow bottom up approach

*Test each and every function*

```c
Function test()
{
    x1=10; y1=-6; z1=25;
    x2=7; y2=7; z2=7;
    d = distance(x1,x2,y1,y2,z1,z2)
    print d, {22.4054}
    ...
    //test angle()
    ...
    //test torsion()
    ...
}
```

Test code is typically 1/3 of real code; Takes longer!

Use tools e.g. Mathematica; Debugger to follow flow & values

If file has been modified or further functions incorporated, just run test code to check if inadvertent editing occurred
Further reading

Beginners
*Teach Yourself C++* by Herbert Schildt

Advanced
1. *Effective C++: 50 specific ways to improve your programs and design*, Scott Meyers
2. *The C++ programming language*, Bjarne Stroustrup (Specification)

Free online resources
http://www.freeprogrammingresources.com/cppbooks.html
Thinking in C++, Bruce Eckel for C programmers
Concluding remarks

1. Consider OOP for large projects
   Be prepared to spend time in the design phase
   Don’t forget the end goal: Simulation => results => publications

2. Test each and every function of your code using appropriate input data
   Test code is part & parcel of the production code
   Employ debuggers to follow flow and values of variable => confidence