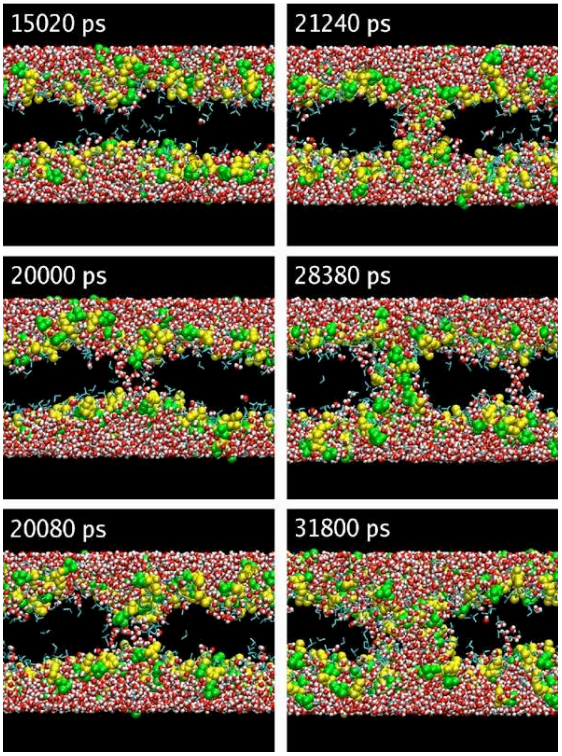
IOP | Institute of Physics

Computational Physics Group

Newsletter



C++ for Scientific Computing

Autumn 2007

MEMBERS OF THE COMMITTEE

Roger Barrett	R.Barrett@surrey.ac.uk
Peter Borcherds (chairman)	p.h.borcherds@birmingham.ac.uk
Alan DuSautoy	alan.dusautoy@npl.co.uk
Hans Fangohr (newsletter)	h.fangohr@soton.ac.uk
Andrew Horsfield	a.horsfield@ucl.ac.uk
Geraint Lewis (vice-chairman,travel bursaries)	dg.lewis@physics.org
Ian Morrison	i.morrison@salford.ac.uk
Matt Probert (honary secretary,treasurer)	mijp1@york.ac.uk
Jesus Rogel	j.rogel@imperial.ac.uk
Michael Sleigh	Michael.Sleigh@awe.co.uk

Our web page can be found here:
http://www.iop.org/activity/groups/subject/comp

Comments about the newsletter, letters and contributions for future editions are welcome and can be sent to Hans Fangohr.

Figure on cover (see also article on page 1):
Molecular dynamics trajectory of the formation of DMSO-induced water pores in lipid membranes. To aid clarity the alkyl chains of the bilayer are not shown. The water molecules are coloured red/white and DMSO is in yellow. The research aims to aid the design of molecules that facilitate the transport of drug molecules through skin and other membranes. (Reproduced from A. Gurtovenko and J. Anwar (2007) J. Phys. Chem. B, 111, 10453-10460. Copyright 1988 American Chemical Society.)

This Newsletter will change . . .

Dear newsletter readership,

due to budgetary constraints, this will be the last printed and (snail) mailed newsletter. In future, the newsletter will be distributed electronically.

You can find the current newsletter in electronic form at
`http://www.soton.ac.uk/~fangohr/randomnotes/iop_cpg_newsletter/index.html`.

The Computational Physics Group Committee.

Newsletter Contents

C++ Object Oriented Programming for Scientific Computing

Jamshed Anwar, Institute of Pharmaceutical Innovation, University of Bradford, Bradford, W. Yorkshire BD7 1DP U.K. Email: j.anwar@bradford.ac.uk

Introduction

Comparing the virtues of different computer languages and programming techniques in any rational way can be a daunting task. It is a bit like discussing religion. Both can evoke considerable emotion with little or no exchange of ideas taking place. However, I perceive that this problem may now be less of an issue as it might have been some 6-8 years back. Now, whenever I mention useful attributes of C++ and object orientated programming (OOP) to my Fortran-programming colleagues, the response typically is that 'you can do that in F2003'. So it appears, fortunately, that we are converging.

My research group committed itself to the OOP approach using C++ about 10 years ago. My intention here is to relate some of that experience to you. The article looks at limitations of structured programming, the essential features of the OOP paradigm, choice of programming languages for implementing scientific problems in OOP, and OOP features of C++. I illustrate the object oriented approach to modelling scientific problems with reference to molecule.h++, which is a framework of objects that have been developed by us for molecular simulations. While I advocate lucid code, there may be instances when such advice may be inappropriate, and I present some pointers for writing obscure code by design. Finally, I address the need for building in quality into the code and the discipline of testing code using a bottom up approach.

For those new to C++, I would recommend the text by Schildt (1) and the on line free book 'Thinking in C++' by Eckel (2). For advanced users the text by Meyer (3) would be invaluable. The C++ specification (4) is also likely to be useful.

Molecular simulation

The discussion on OOP using C++ makes reference to molecule.h++, which is a framework of objects that we have developed for carrying out molecular simulations. To make the discussion accessible, I briefly review the nature of these

calculations. The molecular forces between atoms and molecules are now sufficiently well characterized to simulate the molecular interactions and behaviour of large collections $O(10{,}000)$ of molecules including their molecular trajectories. The simulation of the molecular trajectory yields a molecular level view of the system, which may not be accessible by experiment. The molecular interactions may be considered at the quantum mechanical level where bond making and/or breaking may be involved or at a more approximate level (molecular mechanics) where the molecular integrity remains intact and electrons are not considered explicitly. molecule.h++ is restricted to the molecular mechanics domain. The link between molecular level quantities and the real world is the domain of statistical thermodynamics. Therefore, to calculate real world properties it is necessary to generate a large number of configurations of the system at the appropriate temperature (and pressure) and then to employ the machinery of statistical thermodynamics. The configurations may be generated using the Monte Carlo (MC) technique which is a stochastic method or using molecular dynamics (MD) in which the molecular trajectory is generated using Newtonian mechanics. The MD technique, being deterministic, has the advantage of yielding molecular pathways and dynamical properties such as diffusion constants. System sizes are of $O(10000)$ atoms. To access the nanosecond timescale for such systems the cpu resource requirement is about 4 weeks using about eight 2.6GHz processors. Clearly, computational efficiency is an important requirement for any code.

Data-driven programming is inappropriate for large projects

Traditional structured or data-driven programming (5) involves the use of subroutines or functions. As the code becomes large it becomes increasingly difficult to maintain a mental grasp of the interactions between the functions. Furthermore, even for codes where programmers have a good overview of the structure and flow of the program, it may be difficult to retain this overview over an extended period or after periods of inactivity. From personal experience, maintaining a scientific code based on traditional structured programming of about 20,000 lines can be hard work. It is particularly challenging when one has to make changes or amendments to the code after some inactivity. In such instances, on each occasion, there will be a significant gestation period as the programmer re-familiarises him/herself with the flow of the program and the interactions between the functions or subroutines before the amendments can be incorporated. With scientific codes in excess of 100,000 lines it becomes almost impossible to maintain the code in a research group environment, though it may be less of an issue for dedicated programmers. As it becomes difficult to carry a mental

map of the program flow, possible side effects of amendments or modifications in one part of the code on another part can no longer be foreseen, and one begins to rely on test functions to pick up errors that might be introduced by the amendments. A consequence is a loss of confidence in the code. The latter may not be a significant issue for codes whose main function is graphics orientated e.g. games, where any errors are likely to show up visually. Although I do not know of any systematic study of error (defect) rates in scientific codes, in the mainstream programming world defect rates have been quoted from 50-100 defects/1000 lines for new programs to 5-6 defects/1000 lines in production code. It would be instructive to get an estimate for defect rates in a molecular simulation code. The MD code DL_POLY (6), which is written in Fortran, is extensively used by the molecular simulation community throughout the world. The code is highly accessible and the molecular simulation community appear to test and examine the code in intimate detail. As a result of this interest at the code level, defects in the code continue to be picked up and are corrected by the maintenance team at Daresbury Laboratory. These defects are corrected in the primary source code but also brought to the notice of the community in a highly transparent way by means of 'bugmails'. The DL_POLY version 2.15 contained 203 files comprising a total of about 57,000 lines of code. For this code there have been about 25 bugmails each giving details of a number of bugs reported in the last period. Assuming for our purposes that each bugmail referred to say 4 defects, gives a defect rate of (25 x 4) bugs / 57,000 lines i.e. about two bugs/1000 lines, which in relative terms is excellent. However, on a personal level this could still mean having to issue an erratum to your published study! The upshot of these considerations is that for scientific codes we need to build in quality and to maintain it, and that traditional data-driven programming is unable to meet this objective for large projects.

The OOP paradigm

The OOP approach (7) promises the development of massive projects (>1,000,000 lines) that are robust, easier to maintain, and less prone (in principle) to introduction of defects. The basis of OOP is to model the system of interest as closely as possible to the *tangible* reality, which makes the interactions within the code (the interactions between the objects) easier to comprehend and maintain in an individual's memory. Coupled to this are enabling technical features such data locality and abstraction. With traditional programming the emphasis is on *doing* or *action* and the key components of the code are functions or subroutines e.g. `subroutine InvertMatrix()`. In contrast, in OOP the focus is on the object

which comprises data members (the attributes that define the object) and functions (referred to as methods in OOP) that specify what the object can do. For a `CMatrix` object the data members would be the elements of the matrix whilst the methods would include `Invert()`, `Transpose()`, `Multiply(CMatrix)`. Another illustrative example, a `CAtom` object, is given below.

```
CAtom
{       // variables defining characteristics
        int                         index;
        string                      label;
        double                      charge;
        double                      mass;
        CVector3d                   r;        // coordinates
        CVector3d                   v;        // velocity
        CVector3d                   f;        // force

        // behaviour/actions/functions/methods
        Get..                                 // get data member
        Set..                                 // set data member
        BondDistance(CAtom2);
        BondAngle(CAtom2,CAtom3);
        TranslateTo(r);
        TranslateBy(r);
        VerletStep(timestep);
        ...
}
```

Whilst functions in traditional programming can take any parameters as variables in their arguments, functions associated with objects only work in conjunction with those objects; neither the data members nor the functions can be accessed without the object. This *encapsulation* of the object minimizes the chance of changes in one part of the code causing inadvertent problems elsewhere. The relationship between the functions in traditional data-driven programming to objects in OOP is akin to that between a *verb* and a *noun* in language syntax. The shift of focus from isolated functions to objects in practice means that once an object has been specified, it would be natural to code all important methods for that object even though many such methods may not be required for the project at hand. This will result in some effort being expended in what might be considered to be peripheral activity. However, well specified and implemented objects can result in elegant libraries which should pay dividends in new projects

or in making enhancements to the original project.

C++, Java, Smalltalk?

OOP-based programs can be implemented in most modern programming languages, including F2003. However, certain languages such as smalltalk were designed specifically for OOP and therefore have built-in elegance for this approach. For implementing scientific problems in the OOP mode, the two main contenders are probably C++ and Java; smalltalk is not really designed for scientific applications, whilst F2003 (although it contains OOP features) has had to contend with some F77 baggage. C++ has numerous features that add unnecessary complexity and these result in a very steep learning curve. An important feature of C++, which is missing in Java, is operator overloading, where operators such as `+`, `-`, `/`, and `*` can serve as names of methods. Thus it is possible to write `tMatrixA = tMatrixB * (tMatrixC + tMatrixZ)`. Without operator overloading this instruction would take the form `tMatrixA = tMatrixB.Multiply(tMatrixC.Plus(tMatrixZ))`. Clearly, the operator overloaded form is much more lucid and expresses the operations in a form that we readily comprehend.

Java is essentially a subset of C++. Indeed it is possible to translate Java code automatically into C++. The reverse, of course, would not be possible. Java has some excellent design features, a particularly important one of which is to eliminate all unnecessary complexity and to retain only those features that minimize run-time errors. All memory is allocated dynamically but is deleted by an automatic garbage collector. In C++ the programmer is responsible for deleting any memory that has been allocated dynamically. Also, multi-threaduing is part of the Java language and graphics are built-in making the development of front ends easy. I understand that Java compiled code is almost as fast as C/C++ code. The major limitation is that there is no operator overloading.

OOP features of C++

C++ has all the OOP features that one may desire or expect. The language offers strong type-casting, separation of interface from implementation, encapsulation, polymorphism, templates, and inheritance. The strong type-casting helps to eliminate certain errors at compile-time. For example, function calls that do not match the arguments specified in the header file (.h) in terms of data types as well as their other characteristics such as 'constantness' will be picked up as errors during compilation. Separation of the interface (the specification of func-

tion calls) from the implementation (the actions within the function) enables changes to be made to the implementation without changes to the interface. For example, the call to a function associated with a `CMolecule` object that calculates the forces on the atoms could remain the same, whilst one could implement a more efficient method of calculating the forces. For libraries of code, this means that users do not have to change their existing code. Enscapsulation prevents data from being accessed without referring to an object, which minimizes the possibility of errors being introduced elsewhere whilst making local changes to the code. Indeed, in one's first encounter with C++, a novice is likely to find it difficult to get the objects to interact! Polymorphism allows function calls to have the same names but different arguments. This is more intuitive and results in a consistent interface. For instance, in the molecule.h++ framework we have a series of functions all called force but with different arguments:
`Force(CAtom1,CAtom2,Cutoff)`
`Force(tMolecule1,tMolecule2,Cutoff)` and
`Force(tMolecule1,tMoleculeSpecies2,Cutoff).`

Templates are a powerful feature that can significantly reduce the amount of coding. For scientific code it would be invaluable to have vector and matrix objects as a part of a library. Because of strong type-casting, we would need separate objects for each of the data types, `int`, `bool`, `double` (and `float` if required). Templates represent a single bit of general code that can work for different objects. This, of course, is fine for functions and behaviour that is common for all the desired objects. When specific functionality is required for a particular object, say a dot product for a `CVector<double>` object (which would be meaningless for `CVector<bool>`), then that function would need to be incorporated on an ad hoc basis.

Inheritance involves extending the functionality of a particular object by creating a new object that inherits all the features (data members and methods) of the parent object and adds to it the new required functionality. This approach facilitates reuse of previous code. Parent or base objects should be general with the specificity being enhanced by the derived objects. Novices to C++ often find it difficult to decide whether an extending object should inherit from the base object or include the base object in its definition. The decision, for most cases, is relatively easy. The appropriate question to ask is 'is the derived object a kind-of parent object?' If the answer is yes, then inherit. Let me illustrate this with a couple of simple examples. We take the base object to be `CAtom`. We want to extend the functionality to a molecule. Is a molecule a kind of `CAtom`? No! So inheritance is inappropriate and the relationship between that of `CAtom` and `CMolecule` objects would be of the *include* or *containment*-type. Clearly,

whenever the derived object consists of multiples of base objects, the relationship is that of containment. For the second example, the base object is `CMolecule`. We need to extend the behaviour of this object to rigid molecules. Is the rigid molecule a kind of molecule? Sure, so we inherit from `CMolecule`.

molecule.h++: Objects for molecular simulation

The major problem with OOP is not the issue of getting to grips with a particular programming language but rather one of design, more specifically the design of a model that describes our physical system of interest. It would not be unusual to spend up to two-thirds of the time on the design phase (this excludes the specification period) and just a third on coding. Novices to C++ are often very keen to actually begin coding and do not expend enough effort on designing the model. A consequence is that they regularly break up their objects and redesign, with much ensuing pain. The various objects defined in molecule.h++ as well as their interactions are shown in Figure 1, which serve as an illustrative example of a computational model for molecular simulations. The bread and butter objects are `CVector` and `CMatrix`, which are template based. As a significant part of the code deals with trajectories of particles, 3d vector objects are used extensively for handling coordinates, velocities and forces. These vector objects introduce a certain elegance into the code. The underlying mathematical relationships (e.g. Newtonian mechanics for molecular dynamics) are commonly expressed in vector notation. The 3d vector objects enable the code to be written effectively in vector notation, which makes the code terse, more accessible, and easier to read, leading to a reduction in coding errors. Let me illustrate this aspect using the implementation of the leapfrog integration scheme as an example. The objective here is to calculate the new coordinates $\mathbf{r}(t+\delta t)$ of a particle at $(t+\delta t)$, given the previous coordinates $\mathbf{r}(t)$ and velocity $\mathbf{v}(t-0.5\delta t)$, and the current force $\mathbf{f}$ that acts on the particle due to its environment.

The equations to be implemented take the form

$$\begin{aligned}
\mathbf{v}\left(t+\frac{1}{2}\delta t\right) &= \mathbf{v}\left(t-\frac{1}{2}\delta t\right)+\delta t\frac{\mathbf{f}(t)}{m} \\
\mathbf{r}(t+\delta t) &= \mathbf{r}(t)+\delta t\mathbf{v}\left(t+\frac{1}{2}\delta t\right) \\
\mathbf{v}(t) &= \frac{1}{2}\left[\mathbf{v}\left(t+\frac{1}{2}\delta t\right)+\mathbf{v}\left(t-\frac{1}{2}\delta t\right)\right]
\end{aligned}$$

For which the implementation is

```
// advance velocity to v(t+0.5dt)
v_fhdt = v_bhdt + dt * (f / m)
// advance coordinates using new velocity
r_fdt = r + (dt * v_fhdt)
// calculate velocity at t
v = half * (v_fhdt + v_bhdt)
```

where `r`, `r_fdt`, `v`, `v_fhdt`, `v_bhdt`, and `f` are all 3d vector objects.

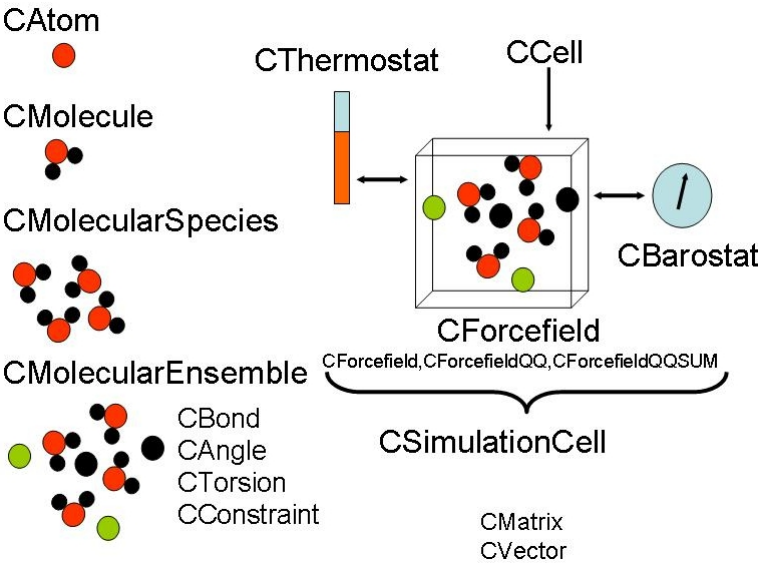


Figure 1: molecule.h++: objects for molecular simulation

The fundamental object in molecule.h++ is `CAtom`. A collection of `CAtom` objects make up a `CMolecule` object. A collection of `CMolecule` objects in turn comprise a `CMolecularSpecies` object. A number of distinct molecule types i.e. `CMolecularSpecies` objects, make up an assembly of species of molecules termed `CMolecularEnsemble`. This assembly of different molecular species is encapsulated in a `CSimulationCell` object, which is the highest level object and represents the overall system. The `CSimulationCell` also includes `CThermostat` and `CBarostat` objects to control the temperature and pressure respectively in the system, and a number of distinct forcefield objects whose purpose is to

calculate the interaction forces. `CSimulationCell` is derived from a generic `CCell` object. All the essential molecular simulation algorithms e.g. Monte Carlo, molecular dynamics, and thermodynamic integration for free energy calculations, are implemented as functions within `CSimulationCell`.

The design of the forcefield objects and their interaction with the rest of the system is not straight forward. Attempting to be faithful to the physical system substantially increases memory requirements and makes the code inefficient and cumbersome. The intuitive approach of associating forcefield parameters and methods to `CAtom`, `CMolecule` or `CMolecularSpecies` objects turns out to be inappropriate in practice. The van der Waals interaction between two atoms is commonly represented by the Lennard-Jones form $U = 4\epsilon\left[(\sigma/r)^{12} - (\sigma/r)^{6}\right]$, where the parameters ϵ and σ characterise the particular interaction between atoms i and j, and r is the distance between the two atoms. An interaction between any two distinct atom-types is therefore characterised, in general, by a different set of ϵ and σ values. The usual input to molecular simulation code consists of the homo-parameters i.e. the interaction between two like atoms (ii or jj) for each atom type. For a typical system the number of different atom types (C, N, O, H etc) rarely exceeds 10. The first step then is to calculate and store the various hetero-parameters. The hetero-parameters are obtained from the homo-parameters using the so called mixing rules: e.g. $\epsilon_{ij} = (\epsilon_{ii}\epsilon_{jj})^{0.5}$; $\sigma_{ij} = 0.5(\sigma_{ii} + \sigma_{jj})$.

How should one incorporate the van der Waals forcefield into the molecular simulation objects? We could make the two parameters, ϵ and σ, for a homo-interaction data members of a `CAtom` object. So, for a particular atom type, say a carbon atom, ϵ and σ would be assigned values that characterise a C..C interaction. There are at least two serious problems in pursuing this approach. Firstly, each atom object will carry two variables of the double type, which means a memory allocation of about 50 kb for a system comprising 20,000 atoms. This in itself is not much but it is important to keep data members of the CAtom object to a minimum to keep the overall memory footprint of the code small. The issue here is one of redundancy; all atoms of a particular type (e.g. all carbon atoms) have the same parameters. The second aspect is one of efficiency. Each time a pair interaction is calculated between two unlike atoms, say a carbon atom with a nitrogen atom, the first step is to calculate the hetero-parameters (C..N) from the homo-parameters (C..C and N..N), and then to employ these in the calculation. In calculating the hetero-parameters we note that one of the mixing rules involves a square root operation, which tends to be computationally very expensive. For a system of N atoms there will be N(N-1) pair-wise interactions at each MD timestep and a typical simulation may consist of 10^6 timesteps.

Wasting valuable cpu resources in carrying out these square root operations in a bid to keep the OOP model close to the physical reality would be unforgivable. An alterative approach, which circumvents the continual re-calculation of the hetero-parameters, is to make all the hetero-parameters, with respect to a single atom-type, data members of the `CAtom` object. There may be other variations on this theme. Such approaches again could significantly increase the memory requirement. In view of these considerations, we have opted for the forcefield components to be independent objects that contain the van der Waals parameters as data members and interact with the `CAtom` object and `CMolecule`, `CMolecularSpecies` and `CMolecularEnsemble`. The forcefield is comprised of three objects, `CForcefield`, a base class that contains a neighbourlist for speeding up pair interaction calculations, and two derived classes, `CForcefieldQQ` and `CForcefieldQQSUM`. `CForcefieldQQ` truncates the coulombic interaction between the atoms or employs a reaction field, while `CForcefieldQQSUM` incorporates Ewald summation.

The problem of minimizing storage of redundant data also arises with the `CMolecule` object. A particular molecule's *intra*molecular connectivity is defined in terms of bonds, bond angles, and dihedrals or torsions. Furthermore, in some instances, parts of the molecule may be kept rigid by means of constraints and there will be parameters associated with these. Therefore, ideally, each molecule should have all the atom indices and parameters that define the bonds, bond angles, torsions and constraints as its data members. This will increase the memory footprint of the `CMolecule` object and consequently the overall memory requirement of the code. These molecular parameters that define the molecule are in fact identical for all the molecules comprising a particular molecular species i.e. an instance of `CMolecularSpecies`. One might think that making these parameters `static` (i.e global to the molecule class), so that there is only one instance of the these parameters, would resolve the problem. This does not work as we may have two or more distinct molecular species in the system, and each requires a different set of molecular parameters. An efficient alternative, which we have adopted in molecule.h++, is to make the molecular parameters data members of the `CMolecularSpecies` object rather than the `CMolecule` object.

Finally, OOP, because of the nature of objects, tends to give rise to code with general functionality. General code invariably means slower code. It is still possible to write specific, fast code using appropriate functions associated with objects but this needs a certain discipline.

Dynamic memory allocation

Other than design issues, another challenging aspect in coding objects is implementing dynamic memory allocation. C++ requires that any objects created on the heap in a program must also be deleted (when no longer required) by the program. An infamous example is that of the hurriedly marketed SGI Irix 5.1 version of the unix operating system from Silicon Graphics. The base SGI workstation was Indy which came with 16 MB of memory. The memory allocation/deletion in this operating system was so bad that the workstation became completely unusable in 3-4 days as the machine inexorably increased its hard-disk pagefile requirement, effectively transforming the respectable R4000 processor to an Intel 386SX. SGI's short term solution was to give additional 16 MB of memory for free! When allocating memory dynamically for objects, one must write an explicit assignment operator and a copy constructor. An assignment operator enables one object to be assigned to another of the same type i.e `tMolecule1 = tMolecule2`. A copy constructor enables a new identical object to be constructed from another. Default versions of both functions are provided by the compiler but these will not work for dynamically allocated objects. The problem encountered is that on assigning one object to another, the pointer to the object in the assigned object points to the same memory as that being pointed to by the first object. Both objects thus point to the same bit of memory. Deletion of one of the objects causes no problems but deletion of the second object results in an attempt to delete the already deleted object again resulting in an error.

```
// create vector object for velocity
CVector *v = new CVector(3);
// Assignment operation
tAtom1 = tAtom2; {e.g.  tAtom1.v = tAtom2.v}
```

Since `v` is a pointer, then `tAtom1.v` → `tAtom2.v` i.e. pointer `v` of object `tAtom1` points to the same memory location as pointer `v` of object `tAtom2`.

```
// delete tAtom1
delete tAtom1.v; // this deletes tAtom1.v
// delete tAtom2
delete tAtom2.v; // this attempts to delete tAtom1.v again→ ERROR
```

To eliminate memory allocation errors one should use tools to check the heap before and after running components of the code, which should show no change. It is interesting to note that F2003 does not suffer from the above memory allocation problems outlined for C++.

Obscure programming by design

Whilst in general the emphasis of the article has been on writing lucid code, there is at least one legitimate reason for writing obscure code (other than simply being inclined towards such behaviour), that is, to ensure the longevity of one's code. For scientific problems it is not uncommon to see 'me too' programs emerging soon after the original program becomes established. How might one discourage the development and uptake of these 'me-too' competitive codes? I reproduce some advice (with minor amendments of my own) from ISI (8).

1. The program must be robust; it should 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).

2. Do not include any comment lines or employ any form of structured programming (OOP is completely out!). Any such attempt to make the code lucid will make it easier for others to 'improve' it and to re-issue it as their own. Refrain from using variable names that are in any of the main languages; English and Indian are out.

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

4. Make sure that the program contains one or two undocumented 'features' or even 'bugs'. This will make the users dependent on you and the expectation of getting the final/enhanced version will encourage users to cite you.

5. By definition, the final version is always six months from completion, so it can never be released.

In crystallography, there are two citation classics, the crystal structure determination and refinement codes SHELX76 (9) and SHELX-90 (10) by G.M. Sheldrick. SHELX76 had citations in excess of 4000 in 1989, whilst SHELX-90, which superseded SHELX76, currently has citations approaching 14,000! I had an opportunity to look at the source code of SHELX76 in the late 80s when I was a PhD student at Birkbeck College London. For a relatively small program ($\sim$ 1000 lines), I found SHELX76 to be brilliantly opaque.

The more conventional route to enhancing the longevity of one's code is to protect the copyright. This is best done by posting at least two copies of the code (at each significant stage of development) to yourself by special delivery. One must keep the certificates of postings and not open the packages until the lawyers need to. The certificates of postings serve as date stamps for the code. Also, it

is important to include a personal (or a group) *signature* throughout the code. This could take the form of 'do-nothing code' comprising important-looking do loops or if statements that appear to be an integral part of the code. Evidence of the signature appearing in a competitor's code would suggest copyright violation.

Correct behaviour above all else

The traditional approach to testing code was to include print or write statements within the code with the objective of determining the contents of key variables. Within the production code the print statements would be commented out or in some instances removed altogether. The sophisticated programmers would include the print statements within a compiler 'debug' directive, which enabled one to select either the production code or the debug version at compile time. For large codes, such an approach is wholly insufficient. I would advocate that the test code is part and parcel of the production code, and that only a bottom up approach where we test each and every function (which is often termed as unit testing) can assure the quality that the scientific community expects. The test code is typically about one-third of the source code. Furthermore, writing good quality test code often takes longer than the main source code, as one needs to devise appropriate input for each of the functions and ascertain the correct output. Our approach is to have a static test function for each object. The static keyword enables a call to the function from the class rather than having to create an instance of the object. The test function tests each of the functions of the object, be they big or small, trivial or complex, using appropriate input and pre-calculated output. For each function, both the ouput and the *expected* output are printed out. The expected output is determined using tools such as Mathcad or Mathematica. The test code becomes increasingly difficult to devise as the functions become of higher level, but is more than worth the effort. The test code is also invaluable when identifying inadvertent editing when the code is amended or its functionality extended. The discipline of testing code should never be neglected because the lack of confidence in the results (does the system really behave like this or is there a bug in the code?) at some later stage can be soul destroying.

Concluding remarks

The experience of the software industry is that OO code is significantly more accessible, easier to maintain and modify, and promotes higher quality, particularly for large projects. In developing or maintaining OOP-based scientific code the

rate limiting step is typically the design stage. It is not unusual to spend 5 days pondering and only half a day implementing the required amendments, which of course should be followed with testing of all the new functions incorporated. The OO approach is practically useless for small utility codes, for which the 5 days of pondering will simply become 5 days of wasted time. However, utility codes based on OO libraries can be developed extremely rapidly. The choice of the programming language is not so important provided all the essential constructs for intuitive OO design are available. For new programmers the first choice would probably be Java. Finally, for mainstream academics, it is important that they do not lose sight of their primary end goal, namely publications. Do not spend excessive time in adding 'all singing and dancing' functionality. Most codes rarely venture outside the originating laboratory.

Acknowledgement

I would like to thank Keith Refson, Rutherford Appleton Laboratory, for valuable discussions and a critical reading of the manuscript.

References

[1] H. Schildt (1997) Teach yourself C++, 3rd edition, McGraw-Hill Companies.

[2] B. Eckel (2007) Thinking in C++, 2nd edition, Free online resource at http://www.freeprogrammingresources.com/cppbooks.html.

[3] S. Meyers (2005) Effective C++: 55 specific ways to improve your programs and designs, 3rd edition, Addison-Wesley Professional.

[4] B. Stroustrup (2000) The C++ programming language, 3rd edition, Addison-Wesley Professional.

[5] B. W. Kernighan and P. J. Plaugher (1978) Elements of programming style, McGraw-Hill Education.

[6] W. Smith and T. Forester (1996) DL_POLY 2.0: A general-purpose parallel molecular dynamics simulation package J. Molec. Graphics, 14, 136-141.

[7] G. Booch (1993) Object-oriented analysis and design with applications, 2nd edition, Addison-Wesley Professional.

[8] Current Contents 41, Oct 9, 1989, ISI.

[9] G.M. Sheldrick (1976) SHELX76, program for crystal structure determination, Cambridge, England: University of Cambridge.

[10] G.M. Sheldrick (1990) SHELX-90, computer program for determining crystal structures, Acta Cryst. A 46, 467-73.

Jamshed Anwar holds a Chair in Computational Pharmaceutical Sciences at the Institute of Pharmaceutical Innovation (IPI), University of Bradford, and can be contacted by email at `j.anwar@bradford.ac.uk`.

Computational Physics Group News

The Computational Physics Thesis Prize 2007

The Committee of the Institute of Physics Computational Group has endowed an annual thesis prize for the author of the PhD thesis that, in the opinion of the Committee, contributes most strongly to the advancement of Computational Physics. A total prize fund of up to £250 will be divided between the prize-winner and the runners up. The number of awards is at the discretion of the Committee. The winner(s) would be expected to provide an article for the IoP Computational Physics Group Newsletter.

- ▷ The deadline for applications is February 29th, 2008.
- ▷ The submission format is a 4 page (A4) abstract together with a citation (max. 500 words) from the PhD supervisor and a confidential report from the external thesis examiner. Further details may be requested from short-listed candidates.
- ▷ The submission address is:
  Dr M Probert
  Department of Physics
  University of York
  York, YO10 5DD
  email: mijp1@york.ac.uk
- ▷ Please enclose full contact details, including an email address.

Applications are encouraged across the entire spectrum of Computational Physics. The competition is open to all students who have carried out their thesis work at a University in the United Kingdom or the Republic of Ireland, and whose PhD examination has taken place in 2007.

The Computational Physics Thesis Prize 2006

The computational physics thesis prize for 2006 has been awarded to Vera Hazelworth – Congratulations!

IoP Computational Physics Group - Research Student Conference Fund

The Institute of Physics provides financial support to research students to attend international meetings and major national meetings.

Eligibility

Bursaries are available only to research students who are members of the Institute and of an appropriate Institute group. For example, if an applicant is a member of the Women in Physics Group only then they could only seek support to attend a conference related to women in physics and not to low temperature physics. To be eligible for that meeting, the applicant would also need to be a member of the Low Temperature Group.

Financial support

Students may apply for up to £250 during the course of their PhD. Students may apply more than once, for example they may request the full amount or decide to request a smaller amount and then apply for funding again for another conference at a later stage. Note that grants will normally cover only part of the expenses incurred in attending a conference and are intended to supplement grants from other sources.

Application procedure

Details of how to apply and an application form are available at
`http://www.iop.org/activity/grants/Research_Student_Conference_Fund/page_26535.html`

Applications are considered on a quarterly basis and should reach the Institute by: 1 March, 1 June, 1 September or 1 December. A decision will be made within eight weeks of the closing date, so the deadline chosen should be at least three months before your event. We strongly recommend that you submit your application early.

Reporting from the meeting

All recipients are asked to produce a report on return from their conference before receiving payment. The report will be published in this Newsletter.

For further information please contact `supportandgrants@iop.org`.

International Union of Pure and Applied Physics: Young Scientist Prize in Computational Physics

Applications invited

The "International Union of Pure and Applied Physics Young Scientist Prize in Computational Physics" (IUPAP Young Scientist Prize) can be awarded to researchers who have a maximum of 8 years research experience following their PhD. See `http://c20.iupap.org/prizes.htm` for details.

Reports on meetings

Mainz Materials Simulation Days 2007

Report by: Mikhail Yakutovich, Sheffield Hallam University

The Mainz Materials Simulation Days are a series of discussion meetings focusing on method developments in computational materials science. In this, the second such meeting, the focus was on biopolymers, surfaces and interfaces, colloids and advanced sampling techniques. Thus, this workshop-style meeting attracted scientists from a number of different but related areas in computational physics and gave a good opportunity for exchange of ideas between specialists working in adjacent areas.

The meeting lasted for two days, each day comprising one main oral session and one poster session. The organizers aimed to limit participant numbers so as to promote close discussions, which in my opinion proved a great advantage over big conferences. For example, conference lunches and dinners were taken all together, so participants could continue their discussions informally and get to know each other better. The conference fee also included a dinner in a nice restaurant. As a result, the networking possibilities were excellent during this meeting.

During the oral and poster sessions, a great variety of simulation techniques were presented, ranging from atomistic to continuum models, in the context of a broad number of applications areas. These gave me a wider perspective on both my particular field of research and the relative advantages and shortcomings of other available techniques. The meeting commenced with an invited talk by Jean-Louis Barrat from Lyon. He spoke about importance of a proper interface description in nano structured systems. Continuum and atomistic levels of description were presented and the results from simulations compared. Another remarkable invited presentation was given by a Mike Allen from Warwick. He presented a talk on interfaces in liquid crystals and techniques for studying them. Colloidal particles adsorbed at liquid crystal isotropic fluid interfaces were also considered, which is currently a very active field of research.

I presented a poster entitled Smoothed dissipative particle nemato dynamics: from atomistic to application scale. It attracted good attention from a number of people the resulting discussions led me to both gain a different outlook on my project and learn how to communicate it effectively. The poster session was

spread over two days, so I also had enough time to thoroughly study the other contributed posters.

I found the conference both very interesting and educational and I gained a significant benefit from its attendance. I would recommend all research students to attend the future events. I am very thankful to the Institute of Physics Computational Physics Group for partially funding my attendance.

2007 APS March Meeting

Report by: Marco Pinna, (University of Central Lancashire, Preston), Conference took place in March 2007; Denver, Colorado (US)

The APS March Meeting at Denver was my first international conference with oral and poster presentations about computer simulation of block copolymers. I talked about sphere and gyroid morphologies of diblock copolymers under external fields using Cell Dynamics Simulation.

The conference hall was very huge, and for me it was unbelievable that so many people can present so many works in different fields. The duration of each talk was very short (only ten minutes) plus two minutes for the questions. Sometimes it was really hard to understand the aim or the final results of the work done. Although the time was really short, there was no delay during the oral sessions. Moreover, there was a focus session every day which anyone can attend (for example, my talk was in the session Phase transitions in polymeric systems). What was also very interesting is the poster session where a lot of different works were presented and where you can ask more questions compared to the oral session presentations. For me it was very useful to see the poster by D. Meng and Q. Wang Symmetric diblock copolymers under Nano-Confinement. The methods used in this work are a lattice Monte Carlo simulation and real-space self-consistent field calculation, which help to understand the formation of the structures and phase transitions of the various morphologies (for example cylinder) in pores of different diameters and different surface preferences. Another interesting poster was about mechanical properties of healthy and tumor tissue by Adriana Dickman. The subject was different from mine, but it was interesting to note that the equations used were similar to the dissipative model where a random force is used.

It is difficult to point a single oral presentation because there were so many very useful talks for continuing my PhD project like one by Venkat Ganesan on

Shear induced phase transition in ternary polymer blend. The system used is only polymer blends, however, the method can be used to study also three-block copolymers.

My talk was appreciated by an experimental group that can lead to a future collaboration on better understanding of polymeric systems under different conditions.

Not all talks on the APS March Meeting were only ten minutes long. Invited talks were longer, which made them easier to understand. Moreover, there were special talks during the special session Polymer Physics Prize. The length of these talks were about 30 minutes plus 6 minutes for questions. They were given by G. H. Fredrikson, F. Bates, E. J. Kramer and other very famous people in the field on polymer physics.

This conference was not only organized to present talks and poster but also to present books, software packages and experimental istruments. There was software to make plots or package to look up for references. The most interesting was the book exibition where anyone can find a book useful for own research field. In my field there were several books in computational and polymer physics. This conference was very useful to understand where the research (for me it is polymer research) is going and what are the possibilities in the future. Such a big conference gives possibility to interact with a lot of different people that work in the same research field and allows for exchange of ideas. This will help to improve the work I am doing in my PhD project.

Non-Adiabatic Molecular Dynamics - A Discussion

Report by Andrew Horsfield. Meeting 10 September 2007, Institute of Physics, London, UK

This one day meeting was held on 10 September 2007, at 76 Portland Place, and constituted a discussion on the subject of non-adiabatic molecular dynamics. Five experts (see below) representing various different methodologies each gave a talk explaining how their particular method works, and how it relates to the other methods discussed. The talks were invariably interrupted every few minutes as people wished to discuss the details. There was a small poster session so delegates could present specific results in some detail obtained using the various methods. About 20 people turned up in all (including a group from the Russian Academy of Sciences).

We were very lucky indeed to have such a select group of speakers for so modest a meeting. They were:

1. Giovanni Ciccotti [Quantum/classical propagators]
2. Todd Martinez [Multiple spawning Gaussian wavepackets]
3. Tchavdar Todorov [Correlated Electron-Ion Dynamics]
4. Nikos Doltsinis [Density Functional Theory surface hopping]
5. Claudio Verdozzi [Open boundary Ehrenfest dynamics]

This is a difficult field, and the talks were highly techical. But the meeting was enjoyed by many who turned up: including the organizer.

Upcoming Computational Physics Events

Multiphysics 2007

- Conference 12 December 2007 to 14 December in Manchester, UK
- webpage: `http://www.multiphysics.org/`

Theory, Modelling and Computational Methods for Semiconductor Materials and Nanostructures

- Workshop on 31 January 2008 and 1 February
- Abstract Deadline: 30th November 2007
- Further information: max.migliorato@manchester.ac.uk
- Webpage:
  `http://www.eee.manchester.ac.uk/research/groups/mandn/events/workshop/`

International Conference on Computational Science (ICCS) 2008

- ICCS 2008 : "Advancing Science and Society through Computation"
- Conference 23 June 2008 to June 25 in Krakow, Poland
- Paper submission deadline: 22 December 2007
- webpage: `http://www.iccs-meeting.org/`

Computational tools: Unit conversions

A re-occcuring problem in physics (and in computer simulations of physics) is the correct conversion of one unit (say metre) into another (say yard). Usually, this should just be a matter of working out the right conversation factor but – if this goes wrong or is forgotten – can be the cause for dramatic accidents.

A software is available (with the name `units`) which can be of great assistence in these situations. We describe the very fundamental way of using it but note for completeness that it is a framework that is much more powerful.

Suppose we need to know how many centimeters there are in one inch. After starting the unit program, it displays

```
You have:
```

and we complete this by typing `inch`

```
You have: inch
```

The unit program then displays the next line

```
You want:
```

and we enter `cm`

```
You want: cm
```

After pressing return, the unit program prints the following answer

```
        * 2.54
        / 0.39370079
```

which tells us that there are 2.54cm in one inch. The second number (0.39370079) can be used for the inverse conversion: there are approximately 0.4 inch within one centimetre.

We summarise this dialog for the following examples as follows (and it will look like this on the screen)

```
You have: inch
You want: cm
        * 2.54
        / 0.39370079
```

The unit program knows many units and constants, and these can be queried. For example, if we need to know what an erg is, we could type `erg`, and just press return when unit asks `You want:`

```
You have: erg
You want:
        Definition: cm dyne = 1e-07 kg m^2 / s^2
```

This may raise the question what a dyne is

```
You have: dyne
You want:
        Definition: cm gram / s^2 = 1e-05 kg m / s^2
```

Alternatively, if we are interested in how this erg energy relates to Joule, we can do this

```
You have: erg
You want: joule
        * 1e-07
        / 10000000
```

to find that 10^7 erg are equivalent to one Joule.

We can further provide a number in addition to the units for the conversion process. Here is an example to see what 32 psi are in bar

```
You have: 32 psi
You want: bar
        * 2.2063223
        / 0.45324293
```

In fact, much more complicated calculations can be carried out (see `http://www.gnu.org/software/units/#examples`).

Finally, we can also provide products and quotients of known units. For example, to convert 70 miles per hour into metre per second

```
You have: 70 miles/hour
You want: m/s
        * 31.2928
        / 0.031956233
```

Putting all this together, and exploiting one of the many known constants of the unit program, we computer what fraction of the velocity of light (abbreviated `c` within the units program) the velocity of 70 miles represents

```
You have: 70 miles / hour
You want: c
        * 1.0438155e-07
        / 9580237.6
```

It is – as expected – a somewhat small fraction.

It is worth noting that the program will complain if units are inconsistent. For example, magnetic fields and magnetic induction are oft used interchangably in some areas of research (and an implicit multiplication or division with the vacuum permeability is required for the conversion of one into the other). If we try to convert 1 milli Tesla into A/m we get the following message

```
You have: 1mT
You want: A/m
conformability error
        0.001 kg / A s^2
        1 A / m
```

Dividing the milli Tesla by μ_0, makes the units equivalent

```
You have: 1mT/mu0
You want: A/m
        * 795.77472
        / 0.0012566371
```

where

```
You have: mu0
You want:
        Definition: 4 pi 1e-7 H/m = 1.2566371e-06 kg m / A^2 s^2
```

There is much more to say about this useful tool, and further information can be found on the webpage `http://www.gnu.org/software/units/`.

The programme is available in all major Linux distributions as a standard package, via Fink for Mac OS X, Cygwin for MS Windows, or can be compiled from source.

(Hans Fangohr)