LECTURE 9

Why work with objects/classes?

- Consider an environment with **huge, continuously evolving and expanding projects, where various new needs are arising all the time and great (if not extreme) flexibility is required.**
- Managing this is possible if the huge project is separated into small (as small as possible) independent **modules**.
- Proper separation makes it possible that each module can be **implemented**, **extended** and **tested** independently of other modules.
- There is no need to run tests on the whole project but only on the modified module.
- Developers of one module need not know about the tiny details of other modules; their actions have no impact on the other modules, etc.
- Within a module, units with very limited (single if possible) responsibilities are identified and turned into **classes** (bundle of data and behaviour).
- Each class communicates with the outside world through its **interface**.
- The interface of a class is all a user needs to know (information hiding, **encapsulation**), in other words: the user knows what a class can do, but there is no need to know how it does that. The "how"-part can change dynamically (**polymorphism**).
- A well written class can be **composed** with other classes and can be **re-used** in many applications; starting from scratch is rarely required.
- Different modules (built of classes) can be linked and used together.
- Object oriented approach yields the desired flexibility.

Object oriented design

- Read about **object oriented principles**, for exampele in *"Object-oriented Software Construction"* by Bertrand Meyer (Prentice Hall 2000)
- Read about **design patterns** (efficient ways to compose objects), for example *"Design patterns : elements of reusable object-oriented software"* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Addison Wesley 1995, new edition: 2000)

Ways of composing objects

The basic ways

- **simple "has-a" relationship**: class `A` has a class `B` as data member and can access all of `B`'s `public` members (or even more) (this relationship is also referred to as **"implemented in terms of"**)
- **"is-a" relationship**: class `A` is derived from class `B` inheriting all the members (**the interface**) of `B` although some of the members might be overridden yielding a polymorphic behaviour. (Recall that the inheritance hierarchy can be more general than this.)
- **polymorphic "has-a" relationship**: class `A` has a reference or pointer member to a base class `B` pointing to particular derived classes of `B`.
- Members of a class `A` can take instances of different classes as arguments communicating with and using them through their `public` interface (or even through their non-public interface).
- These relationships can be combined in many ways (see design patterns).

See the next slides for particular examples on how the above relationships can be implemented in C++.

a special "has-a" relationship - `private` inheritance

```
1  class MyTree : private std::list<MyTree>
2  {
3  public:
4     using std::list<MyTree>::empty;
5     using std::list<MyTree>::push_back;
6
7     MyTree() {}
8     MyTree(double dArg) : dMem_(dArg) {}
9     MyTree(const std::list<MyTree> & cnArg)
10       : std::list<MyTree>(cnArg) {}
11
12    std::ostream & print(std::ostream & os,
13                         int iLevel) const;
14 private:
15    double dMem_;
16 };
```

- in C++ the `private` inheritance realizes a special "has-a" relationship.
- using the keyword `using` the members of the base class can be brought into scope.
- members of the base class not listed in the derived class with a `using` keyword are not accessible from outside `MyTree`.
- this example is a tree of `double`, a particular instance of `MyTree` is a node of the tree. (see auxiliary notes for details of implementation)

"has-a" relationship - implementation matters

- Is it necessary to make the actual object a member? This might make copying the object too expensive, if many instances or copies are created, the memory is heavily used.
- If the data member is a pointer or reference, polymorphic behaviour can be implemented. (Recall the Bridge pattern.)
- With **reference data member**, many instances can share the same data member. However, the default copy assignment doesn't work, a user-defined one must be implemented.
- A **pointer data member** is also useful to share data members among many instances. However, decisions on copying must be made: "Should the copied instance point to the same member or to a cloned copy of it?"
- If the former one is chosen, copying will be easy, economic memory usage can be achieved. However a modification in the shared member has an impact on all the owner instances. Is that really intended?
- Using **smart pointers** is more convenient. With raw pointers one has to take care of many issues.

The purpose of the library

The main goal is to implement a **tool for pricing financial derivatives** based on Monte Carlo type simulations. In particular, we would like to use the library for

- **simulating general input noise** (e.g. increments of Brownian motion, increments of more general Lévy processes, or possibly more general *path-piece* objects - for example *local signature* in the sense of *Rough Paths Theory*)
- **solving/approximating solutions to stochastic differential equations** driven by some input noise using general **global strategies** (e.g. equal step-size method, variable step-size method, Multilevel Monte Carlo method, Cubatre on Wiener Space) in combination with general **local techniques** (Euler method, Milstein method, or even higher order Rough Paths Theory-based methods)
- evaluating general **payoff** functions (European type or even path-dependent ones)
- computing general **statistics** (mean, variance, etc)

The purpose of the library

Further requirements:

- At the points where the term *general* is used, the library must be designed to be **extendible** (i.e. by adding particular classes derived from the abstract ones).
- The different points must be **separated**, implemented without any dependence beyond the communication through the public interfaces. This makes the flexible combinations of components of different versions possible.
- Many different particular tools can be assembled using this library, some might not work properly. Therefore there is a need for **factory** classes assembling working tools for the user. The assembled tools can be used through a **unified user interface**, regardless of the chosen components. For example **one should be able to implement a pricing strategy and run it on different price models by using the same high level code but with a different factory switched on.**

A particular example

Input noise

- Modelled in terms of a d-dimensional Brownian increments: $(B_t - B_s)$ with a 0th component as time increment.

Numerical SDE solver

- For approximating the SDE

$$\mathrm{d}Y_t = \sum_{i=0}^{d} V_i(Y_t)\mathrm{d}B_t^i,\ Y_0 = y.$$

- An equal step-size (Δt) Euler method is implemented for approximating the solution to the

$$\begin{aligned} \widehat{Y}_0 &= y \\ \widehat{Y}_{(k+1)\Delta t} &= \widehat{Y}_{k\Delta t} + \sum_{i=0}^{d} V_i\left(\widehat{Y}_{k\Delta t}\right)\left(B^i_{(k+1)\Delta t} - B^i_{k\Delta t}\right) \end{aligned}$$

The solution paths

- are trajectories on a discrete time scale:

$$\left\{\widehat{Y}_0, \widehat{Y}_{\Delta t}, \ldots, \widehat{Y}_{k\Delta t}, \ldots, \widehat{Y}_T\right\}.$$

Path dependent **payoff functions** would take the whole set as input and produce value.

Basic modelling decisions

Modelling trajectories

We model a path $X : [0, T] \to \mathbb{R}^d$ on the dyadic scale, in particular

- For the non-negative integers k and n, we define the **simple dyadic interval** $[s, t] = [Tk2^{-n}, T(k + 1)2^{-n}]$.
- We model X by the function $f_X(\cdot)$ taking simple dyadic intervals and returning some information on the corresponding **path piece** (see the last two points)
- In MClib, a path object is implemented with the member function $f_X()$, called `find()`.
- In MClib, a path object is templated by the type of the path pieces `path<path_piece>`.
- **The particular input noise paths implemented so far in MClib are in terms of increments**, i.e. $f_X([s, t]) = X_t - X_s$. Generalized versions of increments (collections of iterated integrals) are to be implemented.
- **The particular solution paths implemented so far in MClib are in terms of values**, i.e. $f_X([s, t]) = X_t$. Generalized versions (e.g. *signature* of the path) are to be implemented.

The high level interface, an example

```
1  typedef mc::mc_factory<PPI,PPO> factory;
2  mc::bvector bvStocks;
3  for(unsigned int i=0; i<iNumberOfPaths; ++i)
4  {
5    path_out poPath(ParticularFactory.GetNewPathOut(iAccuracyUsed,
         iGlobalAccuracy,ppoInCond,sT));
6    ParticularPayoff(poPath,bvStocks);
7    ParticularStatistics.DumpOnePayoff(bvStocks);
8  }
```

- The factory `ParticularFactory` of type `myFactory` is initialized with a certain SDE
- the member `GetNewPathOut()` of `myFactor` **generates a new price trajectory** (on a scale of `iGlobalAccuracy` step, the numerical solution is based on input noise generated on a scale with $2^{\texttt{iGlobalAccuracy}}$ steps, the initial condition is set by `ppoInCond`, and the time horizon by `sT`)
- The **payoff** is computed by the function object `ParticularPayoff`, the object `ParticularStatistics` keeps track of the simulation **statistics**.

The high level interface, an example - continued

```
1   typedef mc::mc_factory<PPI,PPO> factory;
2   mc::bvector bvStocks;
3   for(unsigned int i=0; i<iNumberOfPaths; ++i)
4   {
5     path_out poPath(ParticularFactory.GetNewPathOut(iAccuracyUsed,
          iGlobalAccuracy,ppoInCond,sT));
6     ParticularPayoff(poPath,bvStocks);
7     ParticularStatistics.DumpOnePayoff(bvStocks);
8   }
```

- The main code, containing these lines, is based on: Brownian input noise (increments), equal step-size global numerical SDE strategy, Euler local steps, and the geometric Brownian motion SDE.
- To write high level code (e.g. the body of the `for` loop) does not require the detailed knowledge of the components described in the previous point. **For various combinations of components, the same high level code is applicable.**
- MC simulation based pricing algorithm of many financial products can be implemented only by writing a proper payoff function object, again independently from the components in the background.
- On the other hand, one can implement fancy **input noise models**, **input noise generators**, **SDE solvers** and other components together with a **factory** assembling a matching and working composition. The above high level code will run with this new factory too.

Next lecture

High-level interface in details:

- `mc_factory`
- `payoff`
- `statistics`

MClib in more details:

- `path`
- `path_generator`
- `solution_path_generator`
- `strategy`
- `numerical_step`