General Instructions

1. Download `Practical07.zip` from the course website.

   Extract the file under the `[CppCourse]\[Practicals]` folder. Make sure that the file structure looks like:

   ```
   [CppCourse]
      -> [boostRoot]
      ...
      -> [Practicals]
         -> [Practical01]
         -> ...
         -> [Practical07]
            -> ...
            -> Practical07Exercises.hpp
            -> [Src]
               -> Practical07.cpp
      ...
   ```

2. Open the text file `[CppCourse]\CMakeLists.txt`, uncomment the following line by removing the `#`:

   ```
   #add_subdirectory(Practicals/Practical07)
   ```

   and save the file. This registers the project with `cmake`.

3. Run `cmake` in order ot generate the project.

4. The practical depends on the `MCLib` and `DOPLib` libraries. The documentation to these libraries is also available on the website.

5. The header file `Practical07Exercises.hpp` contains the declaration of certain classes and a function. The member functions of these classes and the function are to be implemented into a `cpp` file that is to be created by you under the `[Src]` folder.

6. After compiling and running your code - if the minimum requirements are met - an output text file is created:

   ```
   Practical07_output.txt
   ```

7. Hand in the output file and the `cpp` and `hpp` files you put your implementations into.

8. The files are to be submitted via Moodle.

Exercise 1a

The class `basket_payoff` is a particular derived class from

`public mc::payoff<mc::bvector>`

Given a trajectory of a solution path

$$Y_{t_1}, \ldots, Y_{t_m}, \text{ with } t_m = T, \text{ and } Y_t = (Y_t^1, \ldots, Y_t^N),$$

the basket payoff returns the max of the first n factors at time T:

$$\max_{1 \le i \le n} Y_T^i$$

for some n.
The data member `indMax_` defines n, however if $n > N$, only N components are to be taken into account.
Implement the members:

```
basket_payoff(unsigned int indMax=1);
mc::bvector & operator()(path_out & poArg,
            mc::bvector & bvOut);
unsigned int SizePayoff() const;
```

Exercise 1b

The class `geometric_average_payoff` is a particular derived class from

`public mc::payoff<mc::bvector>`

In nature, `geometric_average_payoff` is similar to

`asian_discretely_sampled_payoff`

declared in `particular_payoff_statistics.hpp` of `MClib`, and implemented in the corresponding `cpp` file.
Given a trajectory of a solution path

$$Y_{t_1}, \ldots, Y_{t_m}, \text{ with } t_m = T, \text{ and } Y_t = (Y_t^1, \ldots, Y_t^N)^T$$

this payoff returns the geometric average of the ith component over time:

$$(Y_{t_1}^i \cdots Y_{t_m}^i)^{1/m}$$

for some i and sampling partition $(t_1, \ldots, t_m)$.
The data member `indY_` defines the index i and `iSamplingAccuracy_` defines the sampling scale.
Implement the members:

```
geometric_average_payoff(unsigned int iSamplingAccuracy,
            unsigned int indY=0);
mc::bvector & operator()(path_out & poArg,
            mc::bvector & bvOut);
unsigned int SizePayoff() const;
```

Exercise 1c

The class `half_call_half_put` is a particular derived class from

`public mc::time_dependent_payoff<mc::bvector>`

Given a trajectory of a solution path

$$Y_{t_1}, \dots, Y_{t_m}, \text{ with } t_m = T, \text{ and } Y_t = (Y_t^1, \dots, Y_t^N)^T$$

and a dyadic interval $[s,t] = [Tk2^{-n}, T(k+1)2^{-n}]$ this time dependent payoff returns the

$$\begin{cases} \big(\max(Y_t^i - K_1, 0), \dots, \max(Y_t^i - K_l, 0)\big) & \text{if } s < T/2 \\ \big(\max(K_1 - Y_t^i, 0), \dots, \max(K_l - Y_t^i, 0)\big) & \text{otherwise} \end{cases}$$

for some vector $K = (K_1, \dots, K_l)$ of strike prices.
The data member `m_index` defines the index i and `m_bvStrikes` defines the vector K of strike prices.
Implement the members:

```
half_call_half_put(const mc::bvector & bvStrikes,
          mc::bvector::size_type index=0);
mc::bvector & operator()(path_out & pFactors,
             const mc::dyadic & dTimeStep,
             mc::bvector & bvValue);
unsigned int SizePayoff() const;
```

Exercise 1d

The class `UpRangeOut` is a function object that implements a knock out conditions similar to `UpAndOut`, `DoubleBarrier`, `UpDownAndOut` declared in

`particular_ko_conditions.hpp`

in `MClib`, and implemented in the corresponding `cpp` file.
The "Up-Range-Out" event occurs if the ith component of the underlying stock price goes above the upper barrier U for N (not necessarily consecutive) barrier times.
The data members

- `sUpperBarrier_` defines the upper barrier U
- `ind_` defines the index $i - 1$
- `iNumberOfeventsBarrier_` is the maximum number barrier events until knock out
- `iNumberOfeventsLeft_` is the remaining number of barrier events until knock out

Implement the members:

```
UpRangeOut(mc::scalar sUpperBarrier,
      unsigned int iNumberOfeventsBarrier,
      unsigned int ind=0);
bool operator()(const mc::bvector & bvArg);
```

Exercise 2

The aim of the exercise is to estimate the value of a derivative together with its the delta and gamma. The sensitivities are to be estimated using the finite difference ratio. In particular, the aim is to estimate:

$$V_0(s) = \mathbb{E}[f(S_T^s)] \tag{1}$$

$$\frac{\mathrm{d}V_0(s)}{\mathrm{d}s} \approx \frac{V_0(s+\varepsilon) - V_0(s-\varepsilon)}{2\varepsilon} = \mathbb{E}\left[\frac{f(S_T^{s+\varepsilon}) - f(S_T^{s-\varepsilon})}{2\varepsilon}\right] \tag{2}$$

$$\frac{\mathrm{d}^2V_0(s)}{\mathrm{d}s^2} \approx \frac{V_0(s+\varepsilon) - 2V_0(s) + V_0(s-\varepsilon)}{\varepsilon^2} = \mathbb{E}\left[\frac{f(S_T^{s+\varepsilon}) - 2f(S_T^s) + f(S_T^{s-\varepsilon})]}{\varepsilon^2}\right] \tag{3}$$

where S_t^x denotes the solution of some SDE, such that $S_0 = x$.

Note In order to reduce the variance of the Monte-Carlo estimates of (2) and (3), for each i, the ith sample of $S_T^{s+\varepsilon}(i)$, $S_T^{s-\varepsilon}(i)$ and $S_T^s(i)$ **are to be estimated using the same instance of input path** (that is driving noise).
The function to be implemented is declared as follows.

```
void Val_FDDelta_FDGamma(unsigned int iLocalAccuracy,
           unsigned int iGlobalAccuracy,
           unsigned int iNumberOfPaths,
           mc::scalar sT,
           mc::bvector & ppoInCond,
           mc::scalar eps,
           mc::mc_factory<mc::bvector,mc::bvector> & ParticularFactory,
           mc::payoff<mc::bvector> & ParticularPayoff,
           mc::statistics & ParticularStatisticsVal,
           mc::statistics & ParticularStatisticsDelta,
           mc::statistics & ParticularStatisticsGamma);
```

where

- `iLocalAccuracy` defines the scale of the numerical method,
- `iGlobalAccuracy` defines the scale of the solution path,
- `iNumberOfPaths` number of paths to be simulated,
- `sT` time horizon T,
- `ppoInCond` initial condition s,

- `eps` tweak size (or bump size) ε,
- `ParticularFactory` defines a set of rules (the underlying SDE, input noise generation, numerical approximation of the SDE etc.),
- `ParticularPayoff` defines the payoff function f,
- `ParticularStatisticsVal` statistics object to dump the payoff values into,
- `ParticularStatisticsDelta` statistics object to dump the finite diff that estimates delta into,
- `ParticularStatisticsGamma` statistics object to dump the finite diff that estimates gamma into.

The function does not return any value; the expectations, and other statistics are to be estimated via the last three input arguments.
Note This function is tested by the function call `testMCFDGreeks(10000);` in line 41 of `Practical07.cpp`. At this sample size you get decent accuracy, however it may take a long time to run. It is recommended to scale down the sample size by a factor or two while debugging your code. Only run it with the original sample size in release mode when the code is ready.