A Short Introduction to Selected Classes of the Boost C++ Library

Dimitri Reiswich

December 2010

Boost has some useful and convenient macros which we will discuss first. The C++ code for the following discussion is shown below.

```
#include <vector>
#include <boost/current_function.hpp>
#include <boost/foreach.hpp>
#include <boost/static_assert.hpp>
#include <boost/detail/lightweight_test.hpp>

#define MY_NUM 1300

void testMacroa(){
        std::cout << "You have called:" << BOOST_CURRENT_FUNCTION << std::endl;
}

void testMacrob(){
        BOOST_STATIC_ASSERT(MY_NUM!=1400);
}


void testMacroc(){
        std::vector<double> myVec(10);

        BOOST_FOREACH(double& x,myVec) x=10.0;
        BOOST_FOREACH(double x,myVec) std::cout << x << std::endl;
}
void testMacrod(){
        BOOST_ERROR("Failure of this function");
        double x=0.0;
        BOOST_TEST(x!=0);
}
```

- `BOOST_CURRENT_FUNCTION` is a macro that returns the current functions name. Calling `void testMacroa()` yields

```
You have called:void __cdecl testMacroa(void)
```

  which shows the function name and its input/output variable types.
- `BOOST_STATIC_ASSERT` generates a compile time error message, if the input expression is false. The `void testMacrob()` version compiles without any problems, since the global variable `MY_NUM` is set to 1400. However, changing this variable to 1300 will result in an error in the compilation output as shown in the figure below.

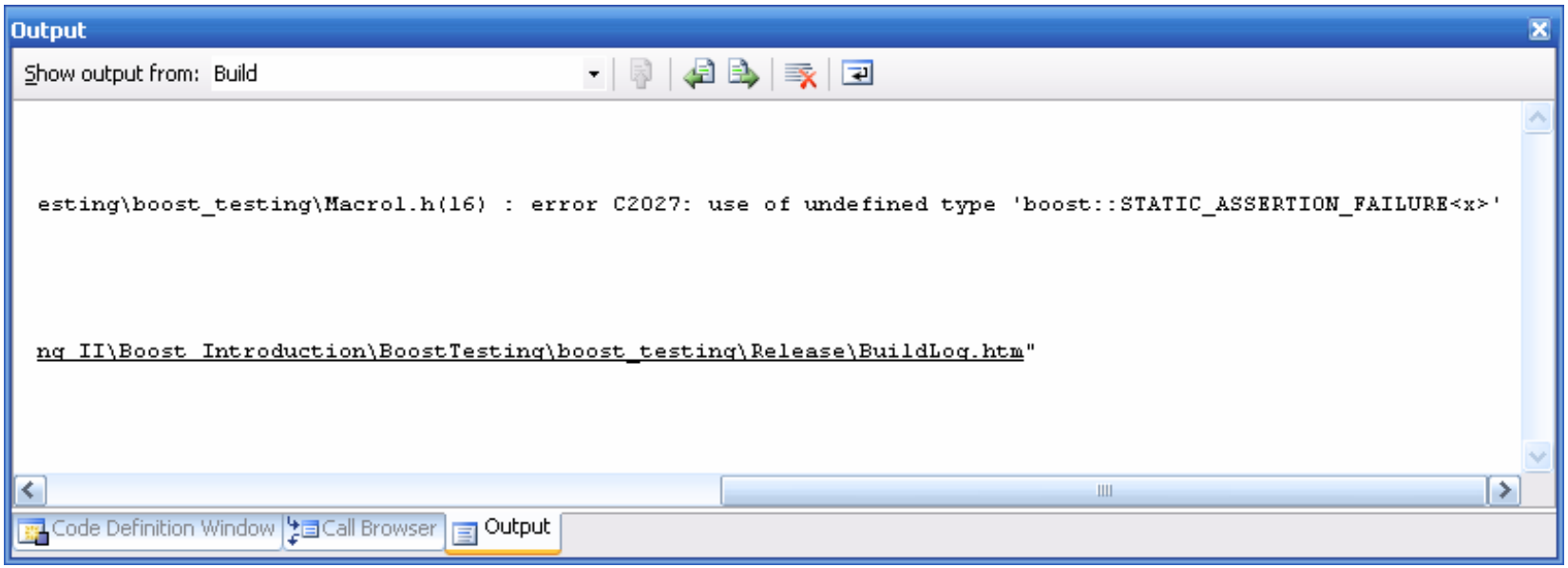

Figure: Static Assert Output

- `BOOST_FOREACH` replaces the often tedious iteration over containers. The usual iteration is either performed via a code similar to `for(int i=0;i<vec.size();i++)` or by using an iterator.

  `BOOST_FOREACH` uses a convenient syntax to do the same job. In the `void testMacroc()` function, the `BOOST_FOREACH(double& x,myVec)` writes the vector `myVec`, while the next line prints the components of the same vector. The output of the program is ten times the number 10. Note that there is a & in the first loop, such that the operation operates by reference on the variable x.

  A similar syntax allows to iterate through maps from the STL. Assume that you have a `std::map<int,double>` variable called `myMap`. The iteration through this map can be performed via

```
std::pair<int,double> x;
BOOST_FOREACH(x,myMap)std::cout << x.second << std::endl;
```

- Finally, we discuss the `BOOST_ERROR` and `BOOST_TEST` macros, which are part of the `lightweight_test` header file. These macros are useful in quick code testing. The first macro prints an error message with the name and location of the error. The output is

```
...boost_testing \Macro1.h(25):  Failure of this function in
function 'void __cdecl testMacrod(void)'
```

The `BOOST_TEST` macro tests if the supplied condition is true and prints an error otherwise. The output in this case is

```
...boost_testing \Macro1.h(26):  test 'x!=0' failed in function '
void __cdecl testMacrod(void)'
```

One of the problems of working with raw pointers which allocate memory on the heap is that is easy to forget to delete them. Another problem is that it is quite easy to reassign the pointer in the code to some other object and cause a memory leak since the reference to the original object is lost. Also, exceptions can cause a failure to reach the delete command, as will be shown later. Finally, it is quite hard to keep track of the object's life time, in particular if the object is passed to other objects and functions. It is hard to say, if the object is still referenced by some other object or if it can be deleted at a certain point.

Modern languages like *Java* have a garbage collector which takes care of all of the problems above. In `C++`, this can be handled by smart pointers, which are part of the `boost` library. The smart pointers are probably the most popular objects in `boost`. The next section will introduce the `boost::shared_ptr`, since it is the most often used smart pointer.

Consider the toy class `TestClassA.h` given below. Any time the constructor/destructor is called, a message will be printed.

```
#ifndef TEST_CLASS_A
#define TEST_CLASS_A

#include<iostream>

class A
{
private:
        double myValue_;
public:

        A(const double& myValue):myValue_(myValue){
                std::cout<<"Constructor of A"<<std::endl;
        }
        ~A(){
                std::cout<<"Destructor of A with value "<<myValue_<<std::endl;
        }
        double getValue() const{ return myValue_;}
};

#endif
```

The usual way to allocate the object `TestClassA` on the heap is to use the `new` operator and delete it at the end via `delete`. This is shown in the function `void testSharedPtra()` below.

```
#include "TestClassA.h"

void testSharedPtra(){
// (a)  Usual way to dynamically allocate memory space on the heap
        A* ptr_myA=new A(1.0);
        delete ptr_myA;
}
```

Calling `testSharedPtra()` yields

```
Constructor of A
Destructor of A with value 1
```

However, in the function `testSharedPtrb()` below we throw an exception before reaching the `delete` statement.

```
#include <boost/shared_ptr.hpp>
#include "TestClassA.h"


void testSharedPtrb(){
// (b) Behaviour of raw pointers for exceptions
        A* ptr_myA=new A(1.0);
        throw "Error ocurred for class A.";
        delete ptr_myA;
}
```

The output of calling the function is

```
Constructor of A.
Error ocurred for class A.
```

Obviously, the destructor has not been called and the object has not been deleted. This, and other problems will be solved by using the smart pointers.

To use boost's **shared_ptr**, include the following header in your project

```
<boost/shared_ptr.hpp>
```

The smart pointers are template classes and constructed via

```
shared_ptr<T> p(new T);
```

where `T` is some class. The constructor after the `new` keyword can be any of the implemented constructors of `T`.

We will test the pointer by allocating again an object of class `A` on the heap. This time, a smart pointer will be used. The code is given by the function `testSharedPtrc()` which is shown below.

```
#include <boost/shared_ptr.hpp>
#include "TestClassA.h"



void testSharedPtrc(){
// (c) Allocation via boost::shared_ptr
        boost::shared_ptr<A> bptr_myA(new A(1.0));
}
```

The output of calling this function is

```
Constructor of A
Destructor of A with value 1
```

Obviously, the object is deleted correctly without calling the `delete` statement in the code.

Is is also possible to use a raw pointer as an argument in the constructor of the **shared_ptr**. This is illustrated in the following function `testSharedPtrd()`.

```
#include <boost/shared_ptr.hpp>
#include "TestClassA.h"

void testSharedPtrd(){
// (d) Assign ownership of usual pointers to a shared_ptr
        A* ptr_myA=new A(1.0);
        boost::shared_ptr<A> bptr_myA(ptr_myA);
        std::cout <<  bptr_myA->getValue() << std::endl;
}
```

In this case, the ownership is assigned to the smart pointer which takes care of the memory management. The function illustrates, that we can call the functions of the object via the usual $\rightarrow$ operator known from raw pointer operations. Calling function `testSharedPtrd()` yields the following output:

```
Constructor of A
1
Destructor of A with value 1
```

The constructor introduced above is quite convenient if the task is to rewrite old **C++** code such that the new code uses smart pointers. The only operation that needs to be done is passing the old pointer to the smart pointer - which should take the old name- and erase the `delete` statement for the old pointer.

What about the exception problem we had before? What will happen, if we throw an exception after creating a smart pointer? An example of this case is given in the function `testSharedPtre()` below.

```
#include <boost/shared_ptr.hpp>
#include "TestClassA.h"

void testSharedPtre(){
// (e) Behaviour of boost::shared_ptr in exceptions
        boost::shared_ptr<A> bptr_myA(new A(1.0));
        throw "Error ocurred in testSharedPtr";
}
```

```
Constructor of A
Destructor of A with value 1
Error ocurred in testSharedPtr
```

Obviously, the object the smart pointer points to is deleted, even though an error is thrown. The next discussion shows that memory leaks can be avoided by using smart pointers.

Consider the function `testSharedPtrf()` given in the code below

```
#include <boost/shared_ptr.hpp>
#include "TestClassA.h"

void testSharedPtrf(){
// (f) Behaviour of raw pointers when reassigning
        A* ptr_myA      =new A(1.0);
        ptr_myA         =new A(2.0);

        delete ptr_myA;
}
```

The first function shows the typical memory leak problem: After assigning a new object to the pointer **ptr_myA**, we loose the reference to object A with the number 1. The first object is not deleted when calling the `delete` operation at the end of the program, as shown by the output

```
Constructor of A
Constructor of A
Destructor of A with value 2
```

The same operation is performed with a smart pointer by using the equivalent `reset` operation.

```
#include <boost/shared_ptr.hpp>
#include "TestClassA.h"

void testSharedPtrg(){
// (g) Behaviour of smart pointers when reassigning
        boost::shared_ptr<A> bptr_myA(new A(1.0));
        bptr_myA.reset(new A(2.0));
}
```

The output is

```
Constructor of A
Constructor of A
Destructor of A with value 1
Destructor of A with value 2
```

This doesn't work with the `reset` function only. It works even if we reassign `bptr_myA` to another boost pointer via `bptr_myA=bptr_myB` where `bptr_myB` points to some other object.

- Write a function which allocates a `std::vector<double>` vector of size 100 using the `shared_ptr` class. Fill the vector at index i with the number i, e.g.`vec[i]=i` by using a standard `for` loop. Calculate the sum of the vector by using the `BOOST_FOREACH` macro.

1. Bernoulli Distribution
2. Beta Distribution
3. Binomial Distribution
4. Cauchy-Lorentz Distribution
5. Chi Squared Distribution
6. Exponential Distribution
7. Extreme Value Distribution
8. F Distribution
9. Gamma (and Erlang) Distribution
10. Log Normal Distribution
11. Negative Binomial Distribution
12. Noncentral Beta Distribution
13. Noncentral Chi-Squared Distribution
14. Noncentral F Distribution
15. Noncentral T Distribution
16. Normal (Gaussian) Distribution
17. Pareto Distribution
18. Poisson Distribution
19. Rayleigh Distribution
20. Students t Distribution
21. Triangular Distribution
22. Weibull Distribution
23. Uniform Distribution

On each distribution, we can apply the usual operations, such as the calculation of the cumulative distribution value at a given point x. The following operations are available with the corresponding syntax

- Cumulative Distribution Function: `cdf(distribution,x)`
- Density: `pdf(distribution,x)`
- Inverse CDF: `quantile(distribution,x)`
- Complementary CDF: `cdf(complement(distribution,x))`
- Mean: `mode(distribution)`
- Variance: `variance(distribution)`
- Standard Deviation: `standard_deviation(distribution)`
- Skew: `skewness(distribution)`
- Kurtosis: `kurtosis(distribution)`
- Excess Kurtosis: `kurtosis_excess(distribution)`

To use the distributions, include the

<boost/math/distributions.hpp>

header in your code, the distribution constructors are straightforward since they simply accept the distribution parameters. Example C++ code is shown below:

```
#include <boost/math/distributions.hpp>

void distributionFunc1(){

        boost::math::normal_distribution<> d(0.5,1);

        std::cout << "CDF:"<< cdf(d,0.2)<< std::endl;
        std::cout << "PDF:"<< pdf(d,0.0)<< std::endl;
        std::cout << "Inverse:"<< quantile(d,0.1)<< std::endl;
        std::cout << "Mode:"<< mode(d)<< std::endl;
        std::cout << "Variance:"<< variance(d)<< std::endl;
        std::cout << "SD:"<< standard_deviation(d)<< std::endl;
        std::cout << "Skew:"<< skewness(d)<< std::endl;
        std::cout << "Kurtosis:"<< kurtosis(d)<< std::endl;
        std::cout << "Excess Kurt:" << kurtosis_excess(d)<< std::endl;

        std::pair<double,double> sup=support(d);
        std::cout << "Left Sup:"<< sup.first<< std::endl;
        std::cout << "Right Sup:"<< sup.second<< std::endl;

}
```

The output of function `void distributionFunc1()` is

```
CDF:0.382089
PDF:0.352065
Inverse:-0.781552
Mode:0.5
Variance:1
SD:1
Skew:0
Kurtosis:3
Excess Kurt:0
Left Sup:-1.79769e+308
Right Sup:1.79769e+308
```

The constructors for other distributions are straightforward, take a look at the boost documentation for details. Below, we give 10 constructor examples for various distributions.

```
#include <boost/math/distributions.hpp>

void distributionFunc2(){

        double leftBound=0.0,rightBound=2.0;
        boost::math::uniform_distribution<> d1(leftBound,rightBound);

        double numTrials=10,probTrial=0.2;
        boost::math::binomial_distribution<> d2(numTrials,probTrial);

        double degFreedom=20;
        boost::math::students_t_distribution<> d3(degFreedom);
        boost::math::chi_squared_distribution<> d4(degFreedom);

        double mean=0.0, var=0.20;
        boost::math::lognormal_distribution<> d5(mean,var);
        boost::math::cauchy_distribution<> d6(mean,var);

        double degFreedom1=20,degFreedom2=35;
        boost::math::fisher_f_distribution<> d7(degFreedom1,degFreedom2);

        double nonCentPar=0.2;
        boost::math::non_central_chi_squared_distribution<> d8(degFreedom1,nonCentPar);

        double arrivRate=0.2;
        boost::math::poisson_distribution<> d9(arrivRate);
        boost::math::exponential_distribution<> d10(arrivRate);

}
```

The general construction procedure for random number generators in boost is

1. set a seed such that you can reproduce the same numbers,
2. choose a random series generator such as Mersenne-Twister,
3. choose the distribution function,
4. connect the random series generator and distribution function via a variate generator,
5. generate random number.

The random number can be obtained by calling the `()` operator on the variate generator.

The following random series generators are available at the moment

- Linear Congruential
- Additive Combine
- Inverse Congruential
- Shuffle Output
- Mersenne Twister with a period length of 2^{19937}
- Various lagged Fibonacci generators

The following distributions have a random number generator

- `uniform_int`: Uniform on a set of integers
- `uniform_real`: Continuous uniform
- `uniform_on_spere`: Uniform on a unit sphere
- `triangle_distribution`
- `bernoulli_distribution`
- `binomial_distribution`
- `exponential_distribution`
- `poisson_distribution`
- `geometric_distribution`
- `cauchy_distribution`
- `normal_distribution`
- `lognormal_distribution`

In the first function below, we generate discrete uniform random numbers in the set $\{1, ..., 6\}$ with Mersenne-Twister. In the second function we generate a normal variable by using a Fibonacci Generator.

```
#include <boost/random.hpp>

void randomFunc1(){
        // create seed
        unsigned long seed=12411;
        // produces general pseudo random number series with the Mersenne Twister Algorithm
        boost::mt19937 rng(seed);
        // uniform distribution on 1...6
        boost::uniform_int<> six(1,6);
        // connects distribution with random series
        boost::variate_generator<boost::mt19937&, boost::uniform_int<>> unsix(rng,six);

        std::cout << unsix() << std::endl;
        std::cout << unsix() << std::endl;
        std::cout << unsix() << std::endl;
}
void randomFunc2(){
        // create seed
        unsigned long seed=89210;
        // produces general pseudo random number series with lagged fibonacci algorithm
        boost::lagged_fibonacci1279 rng(seed);
        // normal distribution with mean 10 and standard deviation 0.1
        boost::normal_distribution<> norm(10,0.1);
        // connects distribution with random series
        boost::variate_generator<boost::lagged_fibonacci1279&, boost::normal_distribution<>>
                unnorm(rng,norm);

        std::cout << unnorm() << std::endl;
        std::cout << unnorm() << std::endl;
        std::cout << unnorm() << std::endl;
}
```

The output of the first function is

```
2
2
5
```

The second function `randomFunc2` gives

```
9.95236
9.9209
9.88191
```

In the C++ code below, we show how to repeat a random number series by resetting its state. We generate random variables following the Cauchy distribution. After generating 3 random variables, we set the random number generator to the initial state by calling **rng.seed(seed)**.

```
#include <boost/random.hpp>

void randomFunc3(){
        // create seed
        unsigned long seed=12411;
        // produces general pseudo random number with MT
        boost::mt19937 rng(seed);
        // distribution, that maps to 1...6
        boost::cauchy_distribution<> cdist;
        // connects mapping with random series
         boost::variate_generator<boost::mt19937&,
                boost::cauchy_distribution<>> cauchy(rng,cdist);

        std::cout << cauchy() << std::endl;
        std::cout << cauchy() << std::endl;
        std::cout << cauchy() << std::endl;

        rng.seed(seed);
        std::cout << "-------"<< std::endl;
        std::cout << cauchy() << std::endl;
        std::cout << cauchy() << std::endl;
        std::cout << cauchy() << std::endl;

}
```

The output after calling the function is

```
1.89711
0.222357
-3.88349
-------
1.89711
0.222357
-3.88349
```

Finally, we show how to generate a random vector with a unit length with dimension 5. This is achieved via the unit sphere distribution.

```
#include <boost/random.hpp>
#include <boost/foreach.hpp>

void randomFunc4(){
        unsigned long seed=24061;
        boost::mt19937 rng(seed);
        boost::uniform_on_sphere<double,std::vector<double>> myUn(5);
        boost::variate_generator<boost::mt19937&,
                boost::uniform_on_sphere<double,std::vector<double>>>
                unSphere(rng,myUn);

        std::vector<double> res=unSphere();

        BOOST_FOREACH(double x,res) std::cout << x << std::endl;
        double sum=0.0;
        BOOST_FOREACH(double x,res) sum+=x*x;
        std::cout << "------------" << std::endl;
        std::cout << "Vector Length:" << std::sqrt(sum) << std::endl;
}
```

The output after calling the function is

```
0.897215
-0.312284
-0.311839
-0.0105574
0.0113303
------------
Vector Length:1
```

- Write a random number generator for the χ_2 distribution with any degree of freedom. This is currently not available in the library. Use the trick, that if $Y \sim U[0,1]$ is a uniform distributed random variable, then

$$X \sim F_X^{-1}(Y)$$

where F_X^{-1} is the inverse CDF of the random variable X.

The function class allows to instantiate a pointer to a function with a flexible and clean syntax. The underlying objects can be any function or function pointer. The class has two syntax forms: the preferred form and the portable form. The choice of the syntax depends on the compiler, we will use the preferred form throughout the following discussion (for Visual Studio 2008). The class requires the following header

```
<boost/function.hpp>
```

The basic structure of the constructor is

```
boost::function<return type (first variable type,..., last variable type)> f;
```

A boost function object which points to a function which returns a `double` and accepts two `double` variables can be initialized with

```
boost::function<double (double,double)> f;
```

A simple multiplication function

```
double myMult(const double& x, const double& y){return x*y;}
```

can be assigned to the boost function pointer via

```
f=myMult;
```

We can now call `f(x,y)` just as if this is the original function. The maximum number of input variables is currently 10.

As an example, we will show the passing of a function pointer to some other function, which does not know anything about the implementation of the passed object. The passed functions will be the payoffs of a call and a put respectively. Obviously, this is a toy example. However, this example can be extended easily to more complex examples. For example, one can think of a Monte Carlo engine, which accepts any payoff function.

```
#include <boost/function.hpp>

double myCall(const double& spot, const double& strike){
        return std::max(spot-strike,0.0);
}
double myPut(const double& spot, const double& strike){
        return std::max(strike-spot,0.0);
}
void printRes(boost::function<double(double,double)> f,
                        const double& x, const double& y){
        std::cout << "Result:" << f(x,y) << std::endl;
}

void testingFunction1(){
        double s=112.5,K=105.0;
        boost::function<double (double,double)> funcPtr;

        funcPtr=myCall;
        printRes(funcPtr,s,K);

        funcPtr=myPut;
        printRes(funcPtr,s,K);
}
```

The output of the function is

```
Result:7.5
Result:0
```

Calling a boost::function object which has not been assigned to another function will result in an exception. However, it is possible to check if the function object is a NULL pointer:

```
boost::function<double (double)> f;
 if(!f) //Error. Do something here
```

The same can be achieved by calling the `f.empty()` function. Assigning a function to **boost function** can be done in various ways. The following three versions for a given function **myFunc** are legal

```
boost::function<double (double)> f(myFunc);
```

```
boost::function<double (double)> f;
f=myFunc;
```

```
boost::function<double (double)> f;
f=&myFunc;
```

Note, that similar results can be achieved by a standard function pointer, such as

```
double (*myMultPtr)(double x, double y); //create function pointer
myMultPtr=&myMult;
```

However, `myMultPtr` needs exactly the same parameter types as the `myMult` function, no implicit conversions of variables are possible.

Something that can not be done by standard function pointers is the definition of a class member function pointer. For example, consider a class `X` with a function `double X::myFunc(double x)`. One is interested in "extracting" the member function by declaring a function pointer to it. This can be achieved with boost's function pointer by passing some class reference to the function (e.g. an object pointer or an object reference). A pointer example will be demonstrated below. First, define the function pointer as

```
boost::function<double(X*,double)> xMembFunc;
xMembFunc=& X::myFunc
```

Given an instance of `X` called `myX` the function pointer can be called via

```
xMembFunc(&myX,x)
```

This can be simplified further by using boost's `bind` library which will be introduced later. A complete example will be shown on the next slide.

```cpp
#include <boost/function.hpp>
#include <boost/bind.hpp>

class FunctionClass{
        private:
                double a_;
        public:

                FunctionClass(const double& a):a_(a){}

                double multWithA(const double& x) const{return a_*x;}
                double operator()(const double& x) const{return a_*x;}
};

void testingFunction2(){

        FunctionClass myClass(2.0);

        double x=12.0;
        // initialize function pointers to a class function
        boost::function<double(FunctionClass*,double)> funcPtr, funcPtr1;
        // assign the multWithA function and the operator
        funcPtr=&FunctionClass::multWithA;
        funcPtr1=&FunctionClass::operator();

        std::cout << myClass.multWithA(x)       << std::endl;
        std::cout << funcPtr(&myClass,x)        << std::endl;
        std::cout << funcPtr1(&myClass,x)       << std::endl;
        // bind the function with the class instance
        boost::function<double (double)> funcPtrNew;
        funcPtrNew=boost::bind(funcPtr,&myClass,_1);

        std::cout << funcPtrNew(x) << std::endl;

}
```

The output of the function is

```
24
24
24
24
```

In some cases it is expensive to have `boost function` clone a function object. In such cases, it is possible to request that Boost keeps a reference to the function object by calling the `boost::ref` function. Example code follows.

```
#include <boost/function.hpp>

double myMult(const double& x, const double& y){
        return x*y;
}

void testingFunction3(){
        boost::function<double (double,double)> myMultPtr;
        myMultPtr=boost::ref(myMult);
        std::cout << myMultPtr(3.0,3.0) << std::endl;
}
```

The bind class is able to bind any function object argument to a specific value or route input arguments into arbitrary positions. Bind does not need to know anything about the variable types the function accepts/returns, which allows a very convenient syntax. The class works for functions, function objects and function pointers. Example applications are:

- You have a function which accepts many arguments, and you would like to work with one variable only, keeping the others constant.
- You are given a function which you can not modify and you would like to call the function with the parameters being in a different, probably more intuitive order.

Many of the applications above arise in situations where a library is used which accepts a functor where the order or number of variables is different than in the own function interface. The function header is

```
<boost/bind.hpp>
```

Bind can be used very conveniently with `boost function` classes, which will be used throughout this introduction. Boost bind refers to input variables by their number with a `_` prefix in the order they are passed to the function. For example, `_1` denotes the first argument of the function.

As an example, we consider the indicator function called `indicatorFunc` on the interval $[a, b]$ declared as

```
double indicatorFunc(const double & x, const double & a, const double & b)
```

The indicator function $\mathbb{I}(x)_{[a,b]}$ is 1, if $x \in [a, b]$ and zero otherwise. Suppose, you need to work a lot with this function on the interval $[-1, 1]$. You certainly don't want to write `indicatorFunc(x,-1,1)` each time you need the function. And you might not be interested in declaring an extra function for each new interval. The new function you want will be a simple function accepting and returning a `double` variable. We will declare it as a boost function first

```
boost::function<double (double)> indPmOne;
```

The `bind` class allows you to bind specified parameters a, b to the declared `indPmOne` function object via

```
double a=-1.0, b=1.0;
indPmOne=boost::bind<indicatorFunc,_1,a,b)
```

The `_1` indicates that the first parameter should stay as it is, while the other parameters should be fixed. The $\mathbb{I}(x)_{[-1.0,1.0]}$ version can now be called easily via

```
indPmOne(x)
```

The corresponding code follows.

```
#include <boost/bind.hpp>
#include <boost/function.hpp>

double indicatorFunc(const double& x, const double& a, const double& b){
        if(x>=a && x<=b)        return 1.0;
        else                    return 0.0;
}

void testingBind1(){
        double a=-1.0, b=1.0;

        boost::function<double (double)> ind;
        ind=boost::bind(indicatorFunc,_1,a,b);

        std::cout << ind(2.0) << std::endl;
        std::cout << ind(0.5) << std::endl;

}
```

The output is

```
0
1
```

Now suppose you like the original function declaration

```
double indicatorFunc(const double & x, const double & a, const double & b)
```

but you prefer some other order of the variables, for example passing the interval bounds first

```
double indicatorFunc(const double & a, const double & b, const double & x)
```

To do this, define a corresponding function object

```
boost::function<double (double, double, double)> indReordered;
```

The `bind` class allows you to reorder the input parameters via

```
indReordered=boost::bind<indicatorFunc,_3,_1,_2)
```

This is somewhat tricky. If the function is called by using the command

```
indReordered(a,b,x)
```

it is read as follows: take third argument of the invoked function and put it in the first place of the original function. Take the first argument and put it in the second original place.... Example code follows.

```
#include <boost/bind.hpp>
#include <boost/function.hpp>

double indicatorFunc(const double& x, const double& a, const double& b){
        if(x>=a && x<=b)        return 1.0;
        else                    return 0.0;
}

void testingBind2(){

        double x=1.01,a=-1.0, b=1.0;

        std::cout << "Original Function:" <<indicatorFunc(x,a,b) << std::endl;
        boost::function<double (double, double, double)> indReordered;
        indReordered=boost::bind(indicatorFunc,_3,_1,_2); // (a,b,x)
        std::cout << "Reordered Arguments:" << indReordered(a,b,x)<< std::endl;
}
```

The output is

```
Original Function:0
Reordered Arguments:0
```

A very useful functionality is the binding of class member functions. Suppose you have a class `X` with a function f and you have an instance `x` of class `X`. The syntax to bind and call the function is

```
bind(&X::f, &x, _1)(y);
```

where we are passing the instance and function by reference. We will demonstrate this in a simple example. Suppose you have your own normal distribution class with a default constructor `NormalClass()` and a memeber function `normalPdf` which returns the density for a given mean and standard deviation as

```
double normalPdf(const double& x, const double& mean, const double& std)
```

You are unhappy about this architecture since you are convinced that passing the mean and standard deviation to the constructor would have been much more convenient. In particular, because you will use the standard normal setup most of the time. However, suppose you can't change the framework. You can setup the standard normal density by defining a boost function as

```
boost::function<double (double)> stdNd;
```

and bind the class member function to the boost function by calling

```
stdNd=boost::bind(&NormalClass::normalPdf,&nc,_1,0.0,1.0);
```

with `nc` being an instance of `NormalClass`.

Now, you can easily call the standard normal density with `stdNd(x)`. A complete example is shown below

```
#include <boost/math/distributions.hpp>
#include <boost/bind.hpp>
#include <boost/function.hpp>

class NormalClass{
public:
                NormalClass(){}
                double normalPdf(const double& x, const double& mean, const double& std){
                                boost::math::normal_distribution<> d(mean,std);
                                return pdf(d,x);
                }
                double normalCdf(const double& x, const double& mean, const double& std){
                                boost::math::normal_distribution<> d(mean,std);
                                return cdf(d,x);
                }
};
void testingBind3(){

        boost::function<double (double)> stdNd, stdNcumm;
        NormalClass nc;
        stdNd=boost::bind(&NormalClass::normalPdf,&nc,_1,0.0,1.0);
        stdNcumm=boost::bind(&NormalClass::normalCdf,&nc,_1,0.0,1.0);

        std::cout << stdNd(1.1) << std::endl;
        std::cout << stdNcumm(0.0) << std::endl;

}
```

The output is

```
0.217852
0.5
```

- Write a `SimpleGenericMonteCarloClass` with the following constructor

```
SimpleGenericMonteCarloClass(boost::function<double (double)>& pathGen,
                             unsigned long& seed, unsigned long& numSims)
```

The function `pathGen` accepts a standard normal variable and returns the asset at time T. The constructor should then invoke the generation of `numSims` standard normals saved in a `boost::shared_ptr<std::vector<double>>`. The normal variables can be generated using boost's random number generators with the provided variable `seed`. The class should have a function

```
void performSimulation(boost::function<double (double)> discPayoff)
```

which calculates the discounted expected Monte Carlo value. The function `discPayoff` accepts the asset at time T (generated by `pathGen`) and returns the discounted payoff. Implement a `getMean()` function which returns the mean. After setting up the class, calculate a plain vanilla put price in a Black-Scholes setup. All market parameters should be bind to the corresponding functions. This implies binding the discount factor and strike to `discPayoff` and binding all GBM parameters to `pathGen`.

The any class does exactly what the name says: it can take any value. This can be helpful in situations, where we receive a general object whose type is not known at this point. As shown in the code below, we can assign any variable to `myVar`, a `double`, a string or even a `std::vector<double>`. Also, we can pass any variable type to a function which accepts an any object. The code below compiles without any errors.

```
#include <boost/any.hpp>
#include <vector>

void callAny(boost::any x);

void AnyTesting1(){

        boost::any myVar;

        myVar=1.1;
        myVar=std::string("1.1");
        myVar=std::vector<double>(3);

        double x=1.1;

        callAny(x);
}

void callAny(boost::any x){
// do nothing
}
```

The type of the any variable can be checked with the `type()` function against the standard `typeid(T)` function, as shown in the code below. To retrieve the original variable back, use the `any_cast` function called by

```
T* ptrMyT=boost::any_cast<T>(&myAny);
```

for a general class `T`. The result is a `NULL` pointer, if the cast was not successful, otherwise it is a valid pointer.

```
#include <boost/any.hpp>


void AnyTesting2(){

        boost::any myAny;
        double myDbl(1.1);

        myAny=myDbl;

        bool isDbl=myAny.type()==typeid(double);
        std::cout << "Is Double:" << isDbl << std::endl;
        bool isString=myAny.type()==typeid(std::string);
        std::cout << "Is String:" << isString << std::endl;

        double* ptrMyDbl=boost::any_cast<double>(&myAny);
        if(ptrMyDbl!=NULL) std::cout << "My Double: "<< *ptrMyDbl << std::endl;

        int* ptrMyInt=boost::any_cast<int>(&myAny);
        if(ptrMyInt==NULL) std::cout << "Cast Failed" << std::endl;

}
```

The output of the function is

```
Is Double:1
Is String:0
My Double:1.1
Cast Failed
```

The **any** class can be very useful for setting up property sets of objects. An illustration for the property set of a barrier option is shown below.

```
#include <boost/any.hpp>
#include <string>
#include <map>

void AnyTesting3(){

        enum BarrierType{DownAndOut,UpAndIn,DownAndIn,UpAndOut};

        std::map<std::string,boost::any> myPropertySet;

        myPropertySet["domRate"]=0.003;
        myPropertySet["forRate"]=0.031;
        myPropertySet["Name"]=std::string("Barrier Option");
        myPropertySet["BarrType"]=BarrierType(DownAndOut);

}
```

Boost optional provides a framework to deal with objects whose initialization is optional. An example is a function which returns an object whose construction has either been successful or not. The library provides a framework to check if the object has been initialized or not. Another example is a class member variable, which is initialized by one of the constructors. Before using the variable, a check for initialization has to be performed. The library has to be included with the following header

```
<boost/optional.hpp>
```

The initialization of an optional object is performed with

- `boost::optional<T> opt(t);`

where `t` is an instance of the template class `T`. The variable `opt` now has a value of 1. If `opt` is not initialized, it has a value of 0. Consequently, we can check if `opt` has one of the values before proceeding. Alternatively, the variable can be checked against the `NULL` keyword.

The class object passed to `optional` can be dereferenced in various ways. Example functions which return objects or pointers to objects are

- `get()`: returns the object
- `operator *()`: returns the object
- `get_ptr()` returns a pointer to the object

Example code for an initialization of a `double` variable follows

```
#include <boost/optional.hpp>

void testingOptional1(){

        boost::optional<double> myOpt1;
        double b=1.1;
        boost::optional<double> myOpt2(b);

        std::cout << myOpt1 << std::endl;
        std::cout << myOpt2 << std::endl;

        if(myOpt1==NULL){
                std::cout << "Empty Object" << std::endl;
        }
        else{
                std::cout << *myOpt1 << std::endl;
        }

        if(myOpt2==NULL){
                std::cout << "Empty Object" << std::endl;
        }
        else{
                std::cout << myOpt2.get() << std::endl;
        }

}
```

The output of the function is

```
0
1
Empty Object
1.1
```

A nice feature of the class is the implementation of the == operator, which allows to compare the optional object with a base object. For example, the following code

- ```
  double a=10.0; boost::optional<double> optA(a);
  bool isA=(optA==a);
  ```

sets `isA` to true. To conclude the discussion, we give an example of an optional member variable initialization. We discuss a class called `SimpleSettlementClass` with two constructors. The first constructor takes the settlement date, while the second is created with a settlement date and the number of days which has to be added to the settlement date. A `settlement()` function returns the settlement while a `settlementDays()` function returns the days for the shift. The first constructor does not initialize the settlement days, so we set this variable to be optional. The class uses boost's date library which will not be introduced here. The syntax is straightforward. The example code follows.
  ```

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <boost/optional.hpp>

class SimpleSettlementClass{

private:
        boost::gregorian::date d_;
        boost::optional<int> settlementDays_;
public:
        SimpleSettlementClass(const boost::gregorian::date& d):d_(d){
                // default constructor with settlement date given
        };
        SimpleSettlementClass(const boost::gregorian::date& d, const int& settlementDays):d_(d),
                settlementDays_(settlementDays){
                // constructor with initial date + settlement days
        };

        boost::gregorian::date settlement() const{

                if(settlementDays_){
                        return d_+boost::gregorian::days(*settlementDays_);
                }
                else{
                        return d_;
                }
        }
        int settlementDays() const{
                if(settlementDays_){
                        return *settlementDays_;
                }
                else{
                        return 0;
                }
        }

};
```

The class functionality is tested with

```
#include "Optional2.h"

void testingOptional2(){
        boost::gregorian::date d1(2009,9,20);

        SimpleSettlementClass settlement1(d1);
        SimpleSettlementClass settlement2(d1,3); // advance settlement by 3 days

        std::cout << "Settlement 1: " << settlement1.settlement()<< std::endl;
        std::cout << "Settlement 2: " << settlement2.settlement()<< std::endl;
        std::cout << "Settlement 1 Days: " << settlement1.settlementDays()<< std::endl;
        std::cout << "Settlement 2 Days: " << settlement2.settlementDays()<< std::endl;

}
```

The output of the function is

```
Settlement 1:  2009-Sep-20
Settlement 2:  2009-Sep-23
Settlement 1 Days:  0
Settlement 2 Days:  3
```

Of course there is a much easier way to achieve the same result, by doing the appropriate calculations in the constructor. However, one can think of other cases where the optional framework is appropriate. We note that the optional class can provide the framework for a `Null` class which holds a not initialized object of any type.

The serialization library allows to store classes and objects in binary format. The original objects can later be reconstructed from the binary files. The binary format can be a text file, xml file or bin file. The library allows to incorporate serialization of existing classes and provides serialization for some standard STL objects. We will demonstrate the last case first. Let us assume that you want to generate 5.000.000 normal random variables once, store them in a `std::vector` and reuse it in other classes later. This will obviously save time as you just have to read in the vector without generating the numbers again. We will store the vector in a bin file which is in this case smaller than an equivalent text file. To do this, we initialize the standard interface to write data to files as output streams by creating a `std::ofstream` object with the `std::ios::binary` flag.

- `std::ofstream ostr("filepath", std::ios::binary)`

There are different headers for different serialization formats, in our case we include

`<boost/archive/binary_oarchive.hpp>`

since we are interested in a bin serialization. The "o" in "oarchive" indicates that this is an output. Next, we open a binary output archive, which needs an output stream in its constructor

- `boost::archive::binary_oarchive oa(ostr);`

Given a vector called `myVec` the vector can be serialized with

- `oa << myVec;`

That's it!

We will demonstrate the example by saving the resulting file in the folder `C:\Boost\Serialization` The file name will include the seed with which the numbers were created. The boost random number library will be used and we will print the time which is needed to run the code.

```
#include <boost/archive/binary_oarchive.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/foreach.hpp>
#include <boost/random.hpp>
#include <fstream>
#include <ostream>
#include <sstream>
#include <boost/timer.hpp>

void testingSerialization1(){
        // start timer
        boost::timer t;
        // ------------------------------------------------------------
        // Random Number generator setup
        unsigned long seed=89210;
        std::stringstream stream; stream << seed;
        // create and open an archive for output
        std::string filename("C:\\Boost\\Serialization\\");
        filename+= "normal_mt_"+stream.str()+".bin";
        //
        std::ofstream ostr(filename.c_str(),std::ios::binary);
        boost::archive::binary_oarchive oa(ostr);
        // setup random number generators
        boost::mt19937 rng(seed);
        boost::normal_distribution<> norm;
        boost::variate_generator<boost::mt19937&, boost::normal_distribution<>>  normGen(rng,norm);
        // ------------------------------------------------------------
        int numVars=5000000;
        std::vector<double> myVec(numVars);
        BOOST_FOREACH(double& x, myVec)         x=normGen();
        // serialize myVec
        oa << myVec;
        // close file
        ostr.close();
        std::cout << "Elapsed time:" << t.elapsed() << std::endl;
}
```

The output of the program is

```
Elapsed time:8.937
```

The program creates a file called `normal_mt_89210.bin` in the `C:\Boost\Serialization` folder. In the next step we will setup a different program which reconstructs the original std::vector from this file. To do this, we create an instream file with

- `std::ifstream istr(filename.c_str(), std::ios::binary);`

where we need again to pass the `std::ios::binary` flag. Then, the

```
<boost/archive/binary_iarchive.hpp>
```

header needs to be included. This time, it is an "in archive", which we initialize with

- `boost::archive::binary_iarchive ia(istr);`

Finally, we will read in the vector called `myVecLoaded` with

- `ia >> myVecLoaded;`

The vector is now ready for usage. The whole example is shown below, we print the first 10 components to check that the vector is not empty.

```
#include <fstream>
#include <ostream>
#include <sstream>
#include <boost/timer.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/archive/binary_iarchive.hpp>

void testingSerialization2(){

        boost::timer t;
   // create and open a character archive for input
        std::string filename("C:\\Boost\\Serialization\\");
        filename+= "normal_mt_89210.bin";

        std::ifstream istr(filename.c_str(), std::ios::binary);

                std::vector<double> myVecLoaded;
                // create and open an archive for input
                boost::archive::binary_iarchive ia(istr);
                ia >> myVecLoaded;

                istr.close();

        for(int i=0; i<10;i++) std::cout << myVecLoaded[i] << std::endl;
        std::cout << "--------------------"  << std::endl;
        std::cout << "Elapsed:" << t.elapsed() << std::endl;
}
```

The output of the function is

```
-0.0316076
0.585161
-0.00298983
1.83834
-1.12767
0.327939
0.531909
0.57683
1.39802
-0.0913574
--------------------
Elapsed:0.187
```

Obviously, the time to read in the vector is much smaller than the equivalent generating routine of new random numbers.

The next example shows how a whole class can be serialized. We will serialize again in a bin file, since the reconstruction in the example below works faster with the bin version. However, we will show how to serialize in a text file too, without reconstructing the class.

The example discusses the class `SimpleGenericMonteCarloClass`. The constructor accepts a seed variable called `seed` and a `numSim` variable which is the number of simulations. The constructor will then trigger a function called `constructNormalVec()` which constructs a `std::vector` with `numSim` standard normal random variables. The simulation is performed by calling the `performSimulation(...)` function. The function accepts the

- `boost::function<double (double)> discountedPayoff`
- `boost::function<double (double)> pathGen`

variables. The function pointer `discountedPayoff` accepts the value S_T and returns the discounted payoff. The `pathGen` variable accepts a standard normal variable and returns the asset at time T. The function `performSimulation(...)` will calculate the `mean_` variable which can be returned by calling the `getMean()` function.

The example shows that the library can serialize `boost::shared_ptr` objects, an important functionality since many classes will have such objects as members. The corresponding header has to be included with

```
<boost/serialization/shared_ptr.hpp>
```

The standard normal variables will be saved in a `boost::shared_ptr<std::vector<double>>` object, which will be serialized since it is a class member. The serialization is invoked by defining a template member function called

```
template<class Archive> void serialize(Archive & ar, const unsigned int vers)
```

The function accepts the archive (e.g. a text, xml, binary archive) and a version number. A member variable `myDbl` of type `double` can be serialized by calling

- `ar & myDbl;`

This needs to be performed on each member variable manually. The generic Monte Carlo class is an example of a possible serialization scenario. The Monte Carlo setup can be performed once and the class can be serialized, such that all important variables (i.e. mean) are available after reconstruction. Also, the vector with the normal variables is available after reconstruction such that we can call `performSimulation(...)` with some other payoff function which will not trigger any new random number generation. The code for the class is shown below.

```
#include <boost/function.hpp>
#include <boost/serialization/shared_ptr.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/foreach.hpp>
#include <boost/random.hpp>

class SimpleGenericMonteCarloClass{
private:
        boost::shared_ptr<std::vector<double>> normVec_;
        double mean_;
        unsigned long numSims_,seed_;
        // setup normal vec;
        void constructNormalVec(){
                boost::mt19937 rng(seed_);
                boost::normal_distribution<> norm;
                boost::variate_generator<boost::mt19937&, boost::normal_distribution<>>  normGen(rng,norm);
                BOOST_FOREACH(double& x, *normVec_) x=normGen();
        }
public:

        SimpleGenericMonteCarloClass(){};
        SimpleGenericMonteCarloClass(unsigned long numSims, unsigned long seed):mean_(0.0),
                numSims_(numSims),seed_(seed), normVec_(new std::vector<double>(numSims)){
                        constructNormalVec();
        };
        double getMean()const{return mean_;};
        double getNumberSimulations()const{return numSims_;};

        void performSimulation(boost::function<double (double)> pathGen,
                                                boost::function<double (double)> discountedPayoff){
                mean_=0.0;
                double pathVal;

                        BOOST_FOREACH(double x, *normVec_){
                                pathVal=pathGen(x);
                                mean_+=discountedPayoff(pathVal);
                        }
                        mean_=mean_/((double)numSims_);
        }

        template<class Archive> void serialize(Archive & ar, const unsigned int version){
        ar & mean_;
        ar & numSims_;
```

The following functions will assist in the function pointer setup

```
double gbmPath(double spot, double rd, double rf, double vol, double tau, double rn){
                double res=spot*std::exp((rd-rf-0.5*vol*vol)*tau+vol*std::sqrt(tau)*rn);
                return res;
}

double discountedCallPayoff(double assetValue, double strike, double rd, double tau){
        double res=std::max(assetValue-strike,0.0)*std::exp(-rd*tau);
        return res;
}

double discountedPutPayoff(double assetValue, double strike, double rd, double tau){
        double res=std::max(strike-assetValue,0.0)*std::exp(-rd*tau);
        return res;
}
```

The serialization can be performed with the same syntax which was used during the serialization of the `std::vector` class

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/binary_oarchive.hpp>
#include <boost/foreach.hpp>
#include <boost/function.hpp>
#include <boost/bind.hpp>
#include <boost/random.hpp>
#include <fstream>
#include <ostream>
#include <sstream>
#include "Serialization3.h" // Header with SimpleGenericMonteCarloClass
#include "Serialization4.h" // Header with payoffs and gbm functions

void testingSerialization3(){

        unsigned long numSims=1000000, seed=20424;
        double spot=100.0, strike=102.0, rd=0.02, rf=0.03, vol=0.124, tau=1.0;

        boost::function<double (double)> pathGen, discountedPayoff;
        pathGen=boost::bind(gbmPath,spot,rd,rf,vol,tau,_1);
        discountedPayoff=boost::bind(discountedCallPayoff,_1,strike,rd,tau);
        SimpleGenericMonteCarloClass mc(numSims, seed);

        mc.performSimulation(pathGen,discountedPayoff);

        std::string filenameTxt("C:\\Boost\\Serialization\\monteCarloTest.txt");
        std::string filenameBin("C:\\Boost\\Serialization\\monteCarloTest.bin");

        std::ofstream ostrTxt(filenameTxt.c_str());
        std::ofstream ostrBin(filenameBin.c_str(), std::ios::binary);

        boost::archive::text_oarchive oaTxt(ostrTxt);
        boost::archive::binary_oarchive oaBin(ostrBin);

        oaTxt << mc; oaBin << mc;

        ostrTxt.close();ostrBin.close();
}
```

We have serialized an instance of the Monte Carlo class in a textfile by including

- `<boost/archive/binary_oarchive.hpp>`

and using

- `boost::archive::text_oarchiv`

similar to the binary output archive case. After running the programm, a `monteCarloTest.txt` and `monteCarloTest.bin` file is created in the folder `C:\Boost\Serialization`. In the example we use the `bind` function to bind all relevant market variables to the payoff and path generation functions. In the next example, we will load the class and print the mean. This should return the Monte Carlo call value. Afterwards, we will call `performSimulation(...)` with a put payoff to check if the vector with the normal variables is recovered correctly. The mean (the put value) is printed again.

```
#include <boost/archive/text_iarchive.hpp>
#include <boost/archive/binary_iarchive.hpp>
#include <boost/function.hpp>
#include <boost/bind.hpp>
#include <fstream>
#include <ostream>
#include "Serialization3.h" // Header with SimpleGenericMonteCarloClass
#include "Serialization4.h" // Header with payoffs and gbm functions


void testingSerialization4(){

        std::string filenameBin("C:\\Boost\\Serialization\\monteCarloTest.bin");
        std::ifstream istrBin(filenameBin.c_str(), std::ios::binary);
        boost::archive::binary_iarchive iaBin(istrBin);

        SimpleGenericMonteCarloClass mc;

        iaBin >> mc;

        istrBin.close();

        std::cout << "Mean Old (Call):" << mc.getMean() << std::endl;

        double spot=100.0,strike=102.0, rd=0.02,tau=1.0, rf=0.03, vol=0.124;
        boost::function<double (double)> discountedPayoff,pathGen;

        discountedPayoff=boost::bind(discountedPutPayoff,_1,strike,rd,tau);
        pathGen=boost::bind(gbmPath,spot,rd,rf,vol,tau,_1);

        mc.performSimulation(pathGen,discountedPayoff);
        std::cout << "Mean New (Put):" << mc.getMean() << std::endl;
}
```

The output of the function is

```
Mean Old (Call):3.53694
Mean New (Put):6.4885
```

The analytical values for the given market parameters are 3.5421 (Call) and 6.4778 (Put). The discussed example is relatively simple. However, one can think of more complex examples of serializing a class with a numerically intensive procedure which can be performed once. We conclude the section by noting that inheritance can be incorporated by calling

- `boost::serialization::base_object<baseClassName>(*this);`

in the `serialize` function of the derived class. The details can be found in the library documentation.

The `filesystem` library provides functionalities to iterate through folders, checking if a folder exists or renaming a folder/file. A typical application is an installation procedure, where the installer has to check if a previous version of the program exists. Also, the existence of important files has to be checked before proceeding with the installation. The following discussion will analyze the previously discussed `C:\Boost\Serialization` path where the following three files are available from the previous examples

- `monteCarloTest.bin`
- `monteCarloTest.txt`
- `normal_mt_89210.bin`

Our first step is to write a function which lists the present files. This requires the definition of a path. An instance of a path can be created with

- `boost::filesystem::path myPath("C:/Boost/Serialization");`

We can check the existence of the path by calling the `exists(myPath)` function which returns a boolean.

Subpaths can be created by a convenient appending procedure. Assume that we have to create a subpath called `newPath` to the folder `TestFolder`. Instead of calling

- `boost::filesystem::path newPath("C:/Boost/Serialization/TestFolder");`

we can equivalently use the base path to append the subfolder with

- `boost::filesystem::path newPath=myPath /"TestFolder";`

To create a folder, the function `create_directory(path)` has to be called. A complete function which tests the introduced functionalities is shown below

```
#include <boost/filesystem.hpp>
namespace fs=boost::filesystem;

void testingFileSystem1(){

                fs::path myPath("C:/Boost/Serialization");
                bool pathExists=fs::exists(myPath);
                std::cout << pathExists << std::endl;

                fs::path newPath=myPath / "test.txt";
                pathExists=fs::exists(newPath);
                std::cout << pathExists << std::endl;
                fs::path myPathCreated=myPath/"TestFolder";
                fs::create_directory(myPathCreated);
                pathExists=fs::exists(myPathCreated);
                std::cout << pathExists << std::endl;

}
```

The output of the function is

```
1
0
1
```

There are various functions which return informations for the path, such as

- `root_path()` which returns the upper level root, i.e. `C:\`
- `parent_path()` which returns the next upper level, i.e. `C:\Boost`
- `filename()` returns the filename, i.e. `monteCarloTest.bin`

Other functions will be introduced below. To iterate through a folder, a `directory_iterator` has to be created. There are mainly two constructors for this iterator

- `basic_directory_iterator(const Path& dp)`: constructs an iterator pointing to the first entry in the directory dp.
- `basic_directory_iterator()`: constructs the end iterator, equivalent to the STL end iterators

The iterator can be increased via the usual `++` operator. In the next example we will iterate through the example folder and print the whole path as well as the filename. Furthermore, we will print the size of the file by calling

- `file_size(const Path& p)`

which returns the size in bytes. Additional functions allow to test if the current path is a directory or a file. The corresponding functions are

- `is_directory(path)`
- `is_regular_file(path)`

```
#include <boost/filesystem.hpp>
namespace fs=boost::filesystem;

void testingFileSystem2(){

        fs::path myPath("C:\\Boost\\Serialization");
        fs::directory_iterator itr(myPath);
        fs::directory_iterator end_itr;

        std::cout << myPath.root_path() << std::endl;
        std::cout << myPath.parent_path() << std::endl;


        while(itr!=end_itr &&  !fs::is_directory(itr->path())){
                std::cout << "-----------------------------------"<< std::endl;
                std::cout << "Path: "<< itr->path()<< std::endl;
                std::cout << "Filename: "<< itr->path().filename()<< std::endl;
                std::cout << "Is File: "<< fs::is_regular_file(itr->path()) << std::endl;
                std::cout << "File Size :"<< fs::file_size(itr->path())<< std::endl;
                std::cout << "-----------------------------------"<< std::endl;
                itr++;
        }


}
```

The output of the function is

```
C:/
C:/Boost
-----------------------------------
Path:  C:/Boost/Serialization/monteCarloTest.bin
Filename:  monteCarloTest.bin
Is Regular File:  1
File Size :8000069
-----------------------------------
-----------------------------------
Path:  C:/Boost/Serialization/monteCarloTest.txt
Filename:  monteCarloTest.txt
Is Regular File:  1
File Size :20161485
-----------------------------------
-----------------------------------
Path:  C:/Boost/Serialization/normal_mt_89210.bin
Filename:  normal_mt_89210.bin
Is Regular File:  1
File Size :40000043
-----------------------------------
```

The next steps will explain the copy, rename and remove functions:

- `fs::copy_file(toBeCopiedPath,copyPath)`
- `fs::rename(oldName,newName)`
- `fs::remove(fileName)`

To test the functions, we will copy the `monteCarloTest.bin` from `C:\Boost\Serialization` to `C:\Boost\Serialization\TestFolder`. The new file will be called `monteCarloTestCopied.bin`. We will then rename the file to `monteCarloTestRenamed.bin` and delete it afterwards. The directory contents at each stage will be printed to assess if the operations were successful. The code is shown below.

```
#include <boost/filesystem.hpp>
namespace fs=boost::filesystem;

void testingFileSystem3(){
        try{
                fs::path originalFile("C:/Boost/Serialization/monteCarloTest.bin");
                fs::path copiedFile("C:/Boost/Serialization/TestFolder/monteCarloTestCopied.bin");
                fs::path newFileName("C:/Boost/Serialization/TestFolder/monteCarloTestRenamed.bin");

                fs::directory_iterator itr(copiedFile.parent_path());
                fs::directory_iterator end_itr;

                while(itr!=end_itr ){
                   std::cout << "Directory files begin: "<< itr->path()<< std::endl;
                   std::cout << "-----------------------------------"<< std::endl;
                   itr++;
                }

                if(!fs::exists(copiedFile)){
                        fs::copy_file(originalFile,copiedFile);
                }

                fs::directory_iterator itr1(copiedFile.parent_path());
                while(itr1!=end_itr ){
                   std::cout << "Directory files ater copy: "<< itr1->path()<< std::endl;
                   std::cout << "-----------------------------------"<< std::endl;
                   itr1++;
                }

                fs::rename(copiedFile,newFileName);

                fs::directory_iterator itr2(copiedFile.parent_path());
                while(itr2!=end_itr ){
                   std::cout << "Directory files after rename: "<< itr2->path()<< std::endl;
                   std::cout << "-----------------------------------"<< std::endl;
                   itr2++;
                }

                fs::remove(newFileName);
                fs::directory_iterator itr3(copiedFile.parent_path());
                while(itr3!=end_itr ){
                   std::cout << "Directory files after remove: "<< itr2->path()<< std::endl;
                   std::cout << "-----------------------------------"<< std::endl;
```

The output of the function is

```
Directory files ater copy:
C:/Boost/Serialization/TestFolder/monteCarloTestCopied.bin
-----------------------------------
Directory files after rename:
C:/Boost/Serialization/TestFolder/monteCarloTestRenamed.bin
-----------------------------------
```

Note that the first and last iterations do not print anything since the directory is empty. Furthermore, the code shows that the class throws exceptions of type `fs::filesystem_error const & fe` which can be caught for error handling.

uBLAS is a `C++` version of the well known Fortran package `BLAS` with a STL conforming iterator interface. The library provides code for dense, unit and sparse vectors. Furthermore, dense, identity, triangular, banded, symmetric, hermitian and sparse matrices are part of the library.

The library covers the usual basic linear algebra operations on vectors and matrices: different norms, addition and subtraction of vectors and matrices, multiplication with a scalar, inner and outer products of vectors, matrix vector and matrix matrix products and a triangular solver.

It is quite intuitive to construct matrices and vectors in **uBLAS**. A vector containing a `double` variable is initialized via `vector<double> myVec(unsigned int dim, double x)` where **dim** is the dimension and **x** is a default parameter. The default parameter doesn't have to be initialized. Clearly, the class is a template such that `<double>` can be replaced by some other class. The vector components are accessed via `myVec(i)` or `myVec[i]`. Special vectors can be created too, such as the unit vector `unit_vector<double> myUnitVec(unsigned int dim)` or the zero vector `zero_vector<double> myZeroVec(unsigned int dim)`. We can then perform operations on the vector, such as summing the components or calculating some norm. This is summarized below.

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
using namespace boost::numeric::ublas;

void vecOperation1(){

        vector<double> myVec(3,2.2);
        myVec[2]=-5.1;

        std::cout << myVec                          << std::endl;
        std::cout << sum (myVec)                    << std::endl;
        std::cout << norm_1 (myVec)                 << std::endl;
        std::cout << norm_2 (myVec)                 << std::endl;
        std::cout << norm_inf (myVec)               << std::endl;
        std::cout << index_norm_inf(myVec)          << std::endl;
}
```

Here

$$\text{norm\_1(v)} := \sum_i |v[i]|$$

and

$$\text{norm\_2(v)} := \sqrt{\sum_i v[i]^2}.$$

Finally

$$\text{norm\_inf}[v] := \max(|v[i]|)$$

and `index_norm_inf` is the vector index where this is the case. The output of the previous function is

```
[3](2.2,2.2,-5.1)
-0.7
9.5
5.97411
5.1
2
```

The library provides the usual vector operations, such as a function that returns the size, the inner product, the sum of two vectors or multiplication of a vector with a scalar. Some examples are given below:

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
using namespace boost::numeric::ublas;

void vecOperation2(){

        vector<double> myVec1(3,2.2);myVec1[2]=-5.1;
        vector<double> myVec2(3,-1.2);myVec2[2]=1.1;

        double multiplier=2.0;

        std::cout << myVec1.size()<<std::endl;
        std::cout << myVec1<<std::endl;
        std::cout << myVec2<<std::endl;
        std::cout << inner_prod(myVec1,myVec2)<<std::endl;
        std::cout << myVec1+myVec2<<std::endl;
        std::cout << myVec1-myVec2<<std::endl;
        std::cout << myVec1*multiplier<<std::endl;
        std::cout << myVec1/multiplier<<std::endl;

}
```

A matrix can be created via `matrix<T> myMat(unsigned int rows, unsigned int cols)` where `T` is a template class. The elements are accessed via `myMat(row,col)` where round brackets are necessary. The matrix has a `size1()` and `size2()` function, which returns the rows and columns respectively. Furthermore, one can create various special matrices, such as the identity matrix via `identity_matrix<double>` or the zero matrix via `zero_matrix<double>`. Additional operations are the transposed of the matrix, the real part of the matrix or the conjugate. The **`resize`** function allows to resize the current matrix without loosing the current components. Example code is shown below.

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/matrix.hpp>
using namespace boost::numeric::ublas;

void matrixOperation1(){

        matrix<double> myMat(3,3,2.5);
        myMat(0,0)=myMat(2,2)=1.0;
        myMat(0,2)=-3.6;myMat(2,0)=5.9;

        std::cout << "My Mat:"<< myMat << std::endl;
        std::cout << "Num Rows:"<< myMat.size1() << std::endl;
        std::cout << "Num Cols:"<< myMat.size2() << std::endl;
        std::cout << "My Mat Transp:"<< trans(myMat) << std::endl;
        std::cout << "My Mat Real Part:"<< real(myMat) << std::endl;
        myMat.resize(4,4);
        std::cout << "My Resized Mat:"<< myMat << std::endl;

}
```

The output of the function is

```
[3](My Mat:[3,3]((1,2.5,-3.6),(2.5,2.5,2.5),(5.9,2.5,1))
Num Rows:3
Num Cols:3
My Mat Transp:[3,3]((1,2.5,5.9),(2.5,2.5,2.5),(-3.6,2.5,1))
My Mat Real Part:[3,3]((1,2.5,-3.6),(2.5,2.5,2.5),(5.9,2.5,1))
My Resized Mat:[4,4]((1,2.5,-3.6,0),(2.5,2.5,2.5,0),(5.9,2.5,1,0),(0,0,0,0))
```

The matrix provides iterators such as `const_iterator1` and `const_iterator2` to iterate through the matrix which reflects the behavior of containers in the STL.

The standard matrix operations, such as the sum and difference will be shown for the special case of a symmetric matrix. Other matrix classes such as banded or sparse matrices are part of the library too. The symmetric matrix type can be initialized by `symmetric_matrix<double, upper> myCorrMat(unsigned int dim)`. The remaining part is to write the upper triangular matrix only, the code takes care of filling the rest of the matrix. Matrix operations, such as adding two matrices and multiplying a matrix with a scalar are straightforward. The multiplication of a matrix with a matrix or a vector will be covered separately. Example code is shown below.

```
#include <boost/numeric/ublas/symmetric.hpp>

#include <boost/numeric/ublas/io.hpp>

using namespace boost::numeric::ublas;


void matrixOperation2(){
        // defining a symmetric correlation matrix
        symmetric_matrix<double, upper> myCorrMat(3);
        myCorrMat(0,0)=myCorrMat(1,1)=myCorrMat(2,2)=1.0;
        myCorrMat(0,1)=0.4;myCorrMat(0,2)=-0.6;
        myCorrMat(1,2)=0.1;
        std::cout << "Initial Mat:" << myCorrMat << std::endl;

        double multiplier=0.5;
        symmetric_matrix<double, upper> myCorrMat1=myCorrMat;
        std::cout << "Sum Mat:" << myCorrMat1+myCorrMat  << std::endl;
        std::cout << "Scalar Mult:" <<myCorrMat1*multiplier << std::endl;
        std::cout << "Scalar Dev:" <<myCorrMat1/multiplier << std::endl;
}
```

The output of the function is

```
Initial Mat:[3,3]((1,0.4,-0.6),(0.4,1,0.1),(-0.6,0.1,1))
Sum Mat:[3,3]((2,0.8,-1.2),(0.8,2,0.2),(-1.2,0.2,2))
Scalar Mult:[3,3]((0.5,0.2,-0.3),(0.2,0.5,0.05),(-0.3,0.05,0.5))
Scalar Dev:[3,3]((2,0.8,-1.2),(0.8,2,0.2),(-1.2,0.2,2))
```

It is possible to extract single rows and columns from a matrix. Also, ranges and submatrices can be extracted. The row and column case is shown in the function `matrixOperation3()` below. The product of a matrix with some other matrix or some other vector is calculated with `prod`. Example code is given in the function `matrixOperation4()`.

```
#include <boost/numeric/ublas/matrix_proxy.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>

using namespace boost::numeric::ublas;

void matrixOperation3(){

        matrix<double> myMat(3,3,2.5);
        myMat(0,0)=myMat(2,2)=1.0;
        myMat(0,2)=-3.6;myMat(2,0)=5.9;

        matrix_row<matrix<double>> mr(myMat,2);
        matrix_column<matrix<double>> mc(myMat,2);

        std::cout << "Mat:" << myMat << std::endl;
        std::cout << "Row:" << mr        << std::endl;
        std::cout << "Col:" << mc        << std::endl;
}

void matrixOperation4(){

        matrix<double> myMat(3,3,2.5);
        myMat(0,0)=myMat(2,2)=1.0;
        myMat(0,2)=-3.6;myMat(2,0)=5.9;

        vector<double> myVec(3,2.0);
        std::cout << prod(myMat,myMat) << std::endl;
        std::cout << prod(myMat,myVec) << std::endl;
}
```

The output of the first function is

```
Mat:[3,3]((1,2.5,-3.6),(2.5,2.5,2.5),(5.9,2.5,1))
Row:[3](5.9,2.5,1)
Col:[3](-3.6,2.5,1)
```

and the output of the second function is

```
[3,3]((-13.99,-0.25,-0.95),(23.5,18.75,-0.25),(18.05,23.5,-13.99))
(-0.2,15,18.8)
```

Currently, the `uBLAS` library does not seem to focus on common operations such as calculating the inverse of a matrix, its determinant or the eigenvalues. Also, decompositions such as the QR decomposition are not available. However, there is a LU decomposition header in the library, which is not well documented at the current stage. Also, there is a LUP decomposition class (LU decomposition with partial pivoting with P being a permutation matrix) which is in my opinion not very intuitive to use. However, we provide an example for solving the system $Ax = b$ with LUP on the next slide, where we have commented each step. The output of calling the function is presented below. The uBLAS library's current focus seems to be the efficient performance of very basic operations on special matrix classes. It is not a powerful Linear Algebra package in the current state.

```
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/lu.hpp>

using namespace boost::numeric::ublas;

void matrixOperation5(){
        // our goal is to solve: A*x=b for the variable x;
        matrix<double> A(3,3,-0.5);
        A(0,0)=A(2,2)=1.8;
        A(0,2)=-2.6;A(2,0)=1.9;

        vector<double> b(3,0.4); b(0)=-0.3;

        // define copies of A and b since the original
        // objects will be overwritten in the code !!!
        matrix<double> A1=A;
        vector<double> x=b;

        // define the permuation matrix, which is
        // actually a vector
        permutation_matrix<double> P1(A1.size1());

        // do the LUP factorization, overwrite A1
        // such that it summarizes L and U in A1,
        // also, P1 will be overwritten
        lu_factorize(A1,P1);
        // write x, our final solution with the
        // overwritten objects A1 and P1
        lu_substitute(A1,P1,x);

        std::cout << "x=" << x << std::endl;
        // check if we receive our original b back?
        std::cout << "A*x="     << prod(A,x) << std::endl;
        std::cout << "b="       << b<< std::endl;

}
```

The output of the function is

```
x=[3](-0.155857,-0.806776,0.162633)
A*x=[3](-0.3,0.4,0.4)
b=[3](-0.3,0.4,0.4)
```

which shows that the original vector `b` is recovered correctly.

Thank you!