

## General Instructions

1. Download **Practical06.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]
-> ...
-> [Practical06]
-> Definitions.hpp
-> [Src]
-> Function.cpp
-> ...
```

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

```
#add_subdirectory(Practicals/Practical06)
```

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

3. Run **cmake** in order to generate the project.
4. The header file **Practical06Exercises.hpp** contains the declaration of eight functions. One is already implemented in **Payoffs.cpp**, the remaining seven are to be implemented by you. Write the implementation into .cpp files under the **[Src]** folder.
5. After compiling and running your code - if the minimum requirements are met - an output text file is created:

**Practical06\_output.txt**

6. Hand in the output file and the cpp file(s) you put your implementations into.
7. The files are to be submitted via Moodle.

### Exercise 1

Study the declaration and the implementation of the class `Function` (`Function.hpp`, `Function.cpp`). Study the basic relationship between `Function` and `IFunction`. Understanding the constructor of `Function` that takes a raw pointer to `IFunction` is crucial.

Note that `IFunction` is an abstract class. The raw pointer to `IFunction` used in the constructor of `Function` has to point to particular derived classes (derived from `IFunction`). `IFunction.hpp` already contains some examples of such derived classes, e.g. `IFunctionConst`, `IFunctionCoordinate`.

### Exercise 2

Study the implementation of `CallPayoff()` available in `Payoffs.cpp`.

The function `CallPayoff()` is declared as

```
1 Function CallPayoff(double dK,BVector::size_type index);
```

The function takes two arguments

- `dK` strike price
- `index` index of the vector of underlying that determines the relevant underlying

and returns a `Function` such that its `operator()` takes a `boost` vector (`BVector`) `bvArg` and returns `max(bvArg[index]-dK,0.0)`, that is the call payoff on the `index` component with strike `dK`.

To implement this functionality, a new class `ICallPayoff` derived from `IFunction` was created in `Payoffs.cpp`. This class stores the strike and index as data members. The `operator()` of `ICallPayoff` implements the call payoff.

Given the particular class `ICallPayoff`, the implementation of `CallPayoff()` is one line:

```
1 Function exercises::CallPayoff(double dK,BVector::size_type index)
2 {
3     return Function(new ICallPayoff(dK,index));
4 }
```

it calls the constructor of `Function` that takes a raw pointer to `ICallPayoff` that is a particular class derived from `IFunction`. This function creates a new instance of `Function` and plugs in the right `IFunPtr` (smart pointer to `IFunction`) defining the call payoff functionality. Note the use of the `new` command, and the use of the constructor of `ICallPayoff`.

Some of the following exercises below might require the implementation of new derived classes from `IFunction` similar to `ICallPayoff`. However, some of the exercises can be implemented in terms of the other ones without creating new classes.

### Exercise 3

The function PutPayoff() is declared as

```
1 | Function PutPayoff(double dK,BVector::size_type index);
```

The function takes two arguments

- **dK** strike price
- **index** index of the vector of underlying that determines the relevant underlying

and returns a **Function** such that its **operator()** takes a **boost** vector (**BVector**) **bvArg** and returns **max(dK-bvArg[index], 0.0)**, that is the put payoff on the **index** component with strike **dK**.

### Exercise 4

The function SpreadOption() is declared as

```
1 | Function SpreadOption(double dK1, double dW1, double dK2, double dW2, BVector::size_type index);
```

The function takes five arguments

- **dK1** strike price of the first call
- **dW1** weight of the first call
- **dK2** strike price of the second call
- **dW2** weight of the second call
- **index** index of the vector of underlying that determines the relevant underlying

and returns a **Function** such that its **operator()** takes a **boost** vector (**BVector**) **bvArg** and returns the spread payoff

$$dW1 * \max(bvArg[index] - dK1, 0.0) + dW2 * \max(bvArg[index] - dK2, 0.0)$$

### Exercise 5

The function StraddleOption() is declared as

```
1 | Function StraddleOption(double dK1, double dW1, double dK2, double dW2, BVector::size_type index);
```

The function takes five arguments

- **dK1** strike price of the call
- **dW1** weight of the call

- dK2 strike price of the put
- dW2 weight of the put
- index index of the vector of underlying that determines the relevant underlying

and returns a **Function** such that its **operator()** takes a **boost** vector (**BVector**) **bvArg** and returns the spread payoff

$$dW1 * \max(bvArg[index] - dK1, 0.0) + dW2 * \max(dK2 - bvArg[index], 0.0)$$

### Exercise 6

The function **BasketPayoff()** is declared as

1 | **Function BasketPayoff();**

The function takes no input arguments and returns a **Function** such that its **operator()** takes a **boost** vector (**BVector**) **bvArg** and returns the max of its entries.

You can write your own max-search algorithm, however you might consider using the **max\_element()** algorithm for this exercise. Reference

[http://www.cplusplus.com/reference/algorithm/max\\_element/](http://www.cplusplus.com/reference/algorithm/max_element/)

If you use this function, don't forget to **#include <algorithm>**

### Exercise 7

The function **AveragePayoff()** is declared as

1 | **Function AveragePayoff();**

The function takes no input arguments and returns a **Function** such that its **operator()** takes a **boost** vector (**BVector**) **bvArg** and returns the average of its entries.

You can write your own algorithm for computing average of entries in a vector, however you might consider using the **accumulate()** algorithm for this exercise. Reference

<http://www.cplusplus.com/reference/std/numeric/accumulate/>

If you use this function, don't forget to **#include <numeric>**.

### Exercise 8

The function **SumOfSquares()** is declared as

1 | **Function SumOfSquares();**

The function takes no input arguments and returns a `Function` such that its `operator()` takes a `boost` vector (`BVector`) `bvArg` and returns the sum of squares of its entries. You can implement your own algorithm for computing sum of squares in a range, however you might consider using the `inner_product` function for this exercise. Reference

[http://www.cplusplus.com/reference/std/numeric/inner\\_product/](http://www.cplusplus.com/reference/std/numeric/inner_product/)

If you use this function, don't forget to `#include <numeric>`.

### Exercise 9

The function `InnerProduct()` is declared as

```
1 Function InnerProduct(const BVector & bvBase);
```

The function takes one input arguments

- `bvBase` a `boost` vector

and returns a `Function` such that its `operator()` takes a `boost` vector (`BVector`) `bvArg` and returns the inner product of `bvBase` and `bvArg`.

You can write your own algorithm for computing inner product of two vectors, however you might consider using the `inner_product` function for this exercise. Reference

[http://www.cplusplus.com/reference/std/numeric/inner\\_product/](http://www.cplusplus.com/reference/std/numeric/inner_product/)

If you use this function, don't forget to `#include <numeric>`.