Maven
Hackerrank - 2022 July31

Question 1

1. Zip And Multiply Vectors

A type level vector of integers can be defined like so:

```
template <int... Xs>
struct Vector;
```

It can be used like this:

```
Vector<1, 2, 3>
```

We want to write a function that takes multiple vectors, and zips them with *.

i.e. given input:

```
Vector<1, 2, 3>, Vector<4, 5, 6>, Vector<7, 8, 9>
```

produce:

```
Vector<28, 80, 162>
```

i.e.

```
Vector<1*4*7, 2*5*8, 3*6*9>
```

A common way to implement this kind of computation statically is with a metafunction:

```
template <typename... Vectors>
struct zip
{
    using type = XXXX;
};
```

where `XXXX` is the logic of the zip.

You could verify like so:

```
static_assert(
        std::is_same<
            zip<Vector<1, 2, 3>, Vector<4, 5, 6>, Vector<7, 8, 9>>::type,
            Vector<28, 80, 162>>::value);
```

My solution

```cpp
#include <iostream>
#include <type_traits>

// DEFINITION: Compile-time integer vector defined as:
template<int... I>
struct Vector;

// The code below will assume a 'zip' metafunction is used, but feel free to use a different approach.
// If you do, please adjust the static assert accordingly.

template<typename V0, typename V1>
struct dot_product
{
};

template<int...Ns0, int...Ns1>
struct dot_product<Vector<Ns0...>, Vector<Ns1...>>
{
    using type = Vector<Ns0*Ns1...>;
};

template<typename...Vectors>
struct zip
{
};

template<typename Vector>
struct zip<Vector>
{
    using type = Vector;
};

template<typename Vector0, typename Vector1, typename...Vectors>
struct zip<Vector0, Vector1, Vectors...>
{
    using type = typename zip<typename dot_product<Vector0, Vector1>::type, Vectors...>::type;
};
```

Question 2

2. Trading profit

You are processing a set of records representing a day of activity of trading shares in a market between two sides: Buyers and Sellers.

The record is in the format

```
Share action_1 size_1 price_1 [ action_2 size_2 price_2 [ ... ] ]
```

For example:

```
MAVEN OFFER 10 25 BID 20 45
```

where

- **Share** is a company name without spaces
- **action** is one of the strings *BUY, SELL, BID, OFFER* representing an order
- **size** is an integer number of shares
- **price** is an integer price per share

There may be multiple orders (action+size+price) on the same line for a share. Many different company names may be traded during a day.

where

- **Share** is a company name without spaces
- **action** is one of the strings *BUY, SELL, BID, OFFER* representing an order
- **size** is an integer number of shares
- **price** is an integer price per share

There may be multiple orders (action+size+price) on the same line for a share. Many different company names may be traded during a day.

Each time an order is read, an attempt must be made to match it against existing (previously read) orders. A match (and therefore a trade) is made when the price is equal to or *better than* existing orders. A price is *Better than* another price, if for a bid or buy price that is desired to be bought there exists another previous lower sell or offer price in the market, as the buyer profits from being able to get a lower price. Accordingly a sell or offer price is *better than* another price if there exists a higher bid or buy price, as the seller profits from being able to get a higher price. If an order cannot be matched it is kept for matching with future orders (left open in the market).

When matching orders the size is also considered. An order may be partially matched when it's size is greater than an order with a matching price. The unmatched size of an order can be matched against further orders or left remaining in it's (buy or sell) side of the market for future matches. Orders are matched in best price order (highest BID first, lowest SELL first). Orders with the same price are matched in first come, first serve (FIFO) order.

action is *BUY* or *SELL* when you place the order or *BID* or *OFFER* when someone else is buying or selling. In all cases when you or someone else is placing an order, a match is attempted.

You sell at prices you previously bought at and buy at prices you previously sold at. Based on your actions, profit is calculated as the difference in price between your order and someone else's order when they match. i.e $Profit_{BUY}$ = BUY - OFFER. $Profit_{SELL}$ = BID - SELL.
No profit is made when your orders match your own and you trade with yourself. Finally when all orders are processed, some may remain unmatched in the market. The value of your remaining orders (taken by summing the price times the size) is your *exposure*. Your *long exposure* is the value of the remaining size of all your unmatched buy orders, and your *short exposure* is the value of the remaining size of all your unmatched sell orders.

Your task is to calculate the profit, long exposure and short exposure remaining after processing all the records by completing the function

```
std::tuple<int, int, int> trade(std::vector<std::string> const& records)
```

where the argument represents the records and the return value contains profit, long exposure short exposure respectively.

My solution (part of it)

```
struct order_info
{
    bool is_my_order;
//  int timestep;
    int size;
    int price;
};

class order_book
{
public:
    order_book() : profit(0)
    {
    }

    void match_existing_buy_order(order_info info)
    {
        for(auto level_iter = sell_side.begin(); level_iter != sell_side.end(); ++level_iter)
        {
            if (info.size == 0) return;
            if (info.price >= level_iter->first)
            {
                for(auto order_iter  = level_iter->second.begin();
                         order_iter != level_iter->second.end(); ++order_iter)
                {
                    int filled_size = std::min(info.size, order_iter->size);

                    // *** update orderbook *** //
                    info.size -= filled_size;
                    order_iter->size -= filled_size;

                    // *** update profit *** //
                /*  if (info.timestep == order_iter->timestep)
                    { */
                        if ((info.is_my_order && !order_iter->is_my_order)||
                           (!info.is_my_order &&  order_iter->is_my_order))
                        {
                            profit += filled_size * (info.price - level_iter->first);
                        }
                /*  }
                    else
                    {
                        if (info.is_my_order && !order_iter->is_my_order)
                        {
                            profit += filled_size * (info.price - level_iter->first);
                        }
                    } */
                }
```

Maven – Round1

Hackerrank - 2022 Aug17

```cpp
// stub classes used in the tests in main
class Base
{
public:
    virtual ~Base() = default;
};

template<int aSize>
class ArrayHolder : public Base
{
public:
    std::array<int, aSize> _array;
};


namespace HandCoded {
using BasePtr = std::unique_ptr<Base>;

BasePtr Instantiate(int runTimeArg)
{
    switch(runTimeArg) {
        case 1:
            return BasePtr{new ArrayHolder<1>()};
        case 10:
            return BasePtr{new ArrayHolder<10>()};
        case 100:
            return BasePtr{new ArrayHolder<100>()};
        case 1000:
            return BasePtr{new ArrayHolder<1000>()};
        default:
            return nullptr;
    }
}
}

template<typename base_t,
         template<int> typename derived_t,  int... validValues>
struct InstantiatorOneInt<base_t, derived_t, N, validValues...>
{
    using ptr_t = std::unique_ptr<base_t>;

    static ptr_t Create(int runTimeArg)
    {
        // TODO
    }
};
```

```cpp
{
    std::unique_ptr<Base> zPointer{HandCoded::Instantiate(1)};
    assert(dynamic_cast<ArrayHolder<1>*>(zPointer.get()));
    zPointer = HandCoded::Instantiate(10);
    assert(dynamic_cast<ArrayHolder<10>*>(zPointer.get()));
    zPointer = HandCoded::Instantiate(100);
    assert(dynamic_cast<ArrayHolder<100>*>(zPointer.get()));
    zPointer = HandCoded::Instantiate(1000);
    assert(dynamic_cast<ArrayHolder<1000>*>(zPointer.get()));

    zPointer = HandCoded::Instantiate(2000);
    assert(!zPointer);
}

{
    using instantiator = InstantiatorOneInt<Base, ArrayHolder, 1, 10, 100, 1000>;

    std::unique_ptr<Base> zPointer{ instantiator::Create(1) };
    assert(dynamic_cast<ArrayHolder<1>*>(zPointer.get()));
    zPointer = instantiator::Create(2);
    assert(!zPointer);
    zPointer = instantiator::Create(10);
    assert(dynamic_cast<ArrayHolder<10>*>(zPointer.get()));
    zPointer = instantiator::Create(100);
    assert(dynamic_cast<ArrayHolder<100>*>(zPointer.get()));
    zPointer = instantiator::Create(1000);
    assert(dynamic_cast<ArrayHolder<1000>*>(zPointer.get()));
    zPointer = instantiator::Create(2000);
    assert(!zPointer);
}

{
    using instantiator = InstantiatorOneInt<Base, ArrayHolder, 1, 10, 100, 1000>;

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, 10000);
    std::unique_ptr<Base> zPointer = instantiator::Create(distrib(gen));
}
```

Objective is to create factory for family of `ArrayHolder`, the factory should be able to select the right type in runtime. There are two possible implementations : `HandCoded` and templated. Please implement the template version so as to pass the test. The challenge here is to construct a factory in compile time, which is used in runtime.

Possible solution

```cpp
template<typename base_t, template<int> typename derived_t,  int... validValues>
struct InstantiatorOneInt
{
};

template<typename base_t, template<int> typename derived_t, int N, int... validValues>
struct InstantiatorOneInt<base_t, derived_t, N, validValues...>
{
    using ptr_t = std::unique_ptr<base_t>;

    static ptr_t Create(int runTimeArg)
    {
        // TODO
        if (runTimeArg == N) return ptr_t{new ArrayHolder<N>};
        else return InstantiatorOneInt<base_t, derived_t, validValues...>::Create(runTimeArg);
    }

};

template<typename base_t, template<int> typename derived_t>
struct InstantiatorOneInt<base_t, derived_t>
{
    using ptr_t = std::unique_ptr<base_t>;
    static ptr_t Create(int runTimeArg)
    {
        //if (runTimeArg == N) return ptr_t{new ArrayHolder<N>};
        return nullptr;
    }
};
```

My implementation specialize the member function without specializing class, which is not right, we need to specialize the class as well.