Coding workshop: the Wallet class

In this worksheet, we will develop a new class to represent the Wallet of the person operating the simulation. The Wallet can store different currencies; it provides a toString function to convert itself into a string, and it provides functions to check if it contains sufficient cash to make good on a given order.

The class interface

Let's start by laying out the interface we'd like and writing the minimal implementation. Add a new class to your project called Wallet. In Wallet.h (or hpp for XCode users), put the following code:

```
#include <string>
#include <map>

class Wallet
{
    public:
        Wallet();
        /** insert currency to the wallet */
        void insertCurrency(std::string type, double amount);
        /** remove currency from the wallet */
        bool removeCurrency(std::string type, double amount);
        /** check if the wallet contains this much currency or more */
        bool containsCurrency(std::string type, double amount);
        /** generate a string representation of the wallet */
        std::string toString();

    private:
        std::map<std::string,double> currencies;

};
```

Now go ahead and write the minimal code in Wallet.cpp to make it compile. For example, here is a minimal insertCurrency function:

```
void Wallet::insertCurrency(std::string type, double amount)
{

}
```

Now, for testing, comment out the code in the main function and change it to this:

```
#include "Wallet.h"
```

```
int main()
{
    Wallet wallet{};
}
```

Verify that everything is compiling and running. If it is, we are ready to start filling out the functions.

insertCurrency

The first function we will implement is the insertCurrency function. This will add the sent currency into the currencies map data member on the Wallet class. The simplest version of this is as follows:

```
void Wallet::insertCurrency(std::string type, double amount)
{
    currencies[type] += balance;
}
```

There are some problems with this, which it is your job to solve.

Value is less than zero

If the incoming amount is less than zero, the function will remove money from the Wallet. Can you adapt the code, so it throws an exception if the amount is less than zero? (See textbook Chapter 15 p573 for details on throwing exceptions)

Type is not currently in the map

If the incoming type is not presently in the map, you need to initialise the value to the incoming amount. You can check if a map has a key with its count function member, but it has interesting behaviour. As soon as you attempt to access a key in a map, the value of that key is initialised to zero. Try running the following:

```
std::map<std::string,double> things;
things["test"] = 10;
std::cout << "Count of things test: " << things.count("test") << " value " << things["test"]
std::cout << "Count of things test2: " << things.count("test2") << " value " << things["test
std::cout << "Count of things test2: " << things.count("test2") << " value " << things["test
```

You should see something like this:

```
Count of things test: 1 value 10
Count of things test2: 0 value 0
Count of things test2: 1 value 0
```

The interesting part is the final line - once we access things[“test2”], it is initialised to zero. I think the most explicit way to specify the behaviour we want is something like this:

```
if (currencies.count(type) == 0)
{
    currencies[type] = 0;
}
```

Can you include that code into the insertCurrency function?

toString

Now we can put money into our Wallet, it would be useful to be able to print out the contents. We can add a toString function to the Wallet class, as follows:

```
std::string s;
for (std::pair<std::string,double> pair : currencies)
{
    std::string currency = pair.first;
    double amount = pair.second;
    s += currency + " : " + std::to_string(amount) + "\n";
}
return s;
```

This code iterates over the currencies map, pulls each item off there as a std::string,double pair. While it does this, it constructs a string describing the contents of the Wallet.

Note that the ‘\n’ character will actually be converted into the appropriate characters needed by your platform to end a line.

We can now test out the wallet:

```
#include "Wallet.h"
#include <iostream>

int main()
{
    Wallet wallet{};
    wallet.insertCurrency("BTC", 1.5);
    std::cout << wallet.toString() << std::endl;
}
```

alternative solution: operator overloading

We can improve the syntax for printing the Wallet slightly using operator overloading. We saw operator overloading before - it allowed us to compare

strings as std::string defines what to do when it is operated upon by the > and < operators. For the std::string class, we did not implement the operator overloading ourselves.

How about if we could use operator overloading to allow us to feed a wallet directly to a stream, like this:

```
std::cout << wallet << std::endl;
```

Right now, you cannot do that as the wallet class has no way of responding to the << operator (unlike the std::string class, for example). How can we tell the Wallet class how to respond to << ?

Put this code into your Wallet.h file, in the public section:

```
friend std::ostream& operator<<(std::ostream& os, Wallet& wallet);
```

Put this code into Wallet.cpp:

```
std::ostream& operator<<(std::ostream& os,  Wallet& wallet)
{
    os << wallet.toString();
    return os;
}
```

You will notice that the function is defined as a friend of the Wallet class. The reason is that the function is actually global (not a function member of Wallet like the toString function and the others), and we want the << function to have special access to the wallet class.

With the operator overloading code in place, you can do this:

```
#include "Wallet.h"
#include <iostream>

int main()
{
    Wallet wallet{};
    wallet.insertCurrency("BTC", 1.5);
    std::cout << wallet << std::endl;
}
```

you should see this output:

```
BTC : 1.500000
```

I would argue that in fact, the toString function is a public function on the Wallet class, and therefore, the operator overloading function does *not* need special access (friend access) to the Wallet class. Still, the compiler disagrees and will not compile without the friend keyword. Try removing the friend keyword, and you will see what I mean.

You can read more about friend functions in textbook chapter 11 p411. You can read more about operator overloading in Chapter 12 p449.

containsCurrency

Hopefully, you are not too overloaded to continue working on the Wallet class. The next function to implement is the containsCurrency function. This function needs to check if the Wallet contains at least the specified amount of the specified currency. Here is the function signature for a reminder:

```
bool containsCurrency(std::string type, double amount);
```

Using the knowledge you now have of maps, can you write some code to check if the Wallet contains at least this amount of currency? You should check if the amount is positive in your function. It is up to you if you want to throw an exception on bad input. Return true if it contains at least this much currency, false otherwise.

removeCurrency

Next up, we need a function to remove currency from the Wallet. The function has the following signature:

```
bool removeCurrency(std::string type, double amount);
```

Here we can make use of the containsCurrency function to check if the Wallet contains at least this much currency. If it does, we can then update the currencies map accordingly by removing that much currency from the map. Go ahead and implement this. It should return true if the operation is successful, otherwise, return false.

Test the Wallet

Now you should be able to test your Wallet with code a bit like this:

```
int main()
{
    Wallet wallet{};
    wallet.insertCurrency("BTC", 1.5);
    std::cout << "Wallet should contain 1.5 BTC now" << std::endl;
    std::cout << wallet << std::endl;
    bool result = wallet.containsCurrency("BTC", 1.5);
    std::cout << "Result should be true " << result << std::endl;
    result = wallet.removeCurrency("BTC", 2.0);
    std::cout << "Result should be false " << result << std::endl;
```

```
    result = wallet.removeCurrency("BTC", 1.0);
    std::cout << "Result should be true " << result << std::endl;
}
```

The output of the above code is as follows:

```
Wallet should contain 1.5 BTC now
BTC : 1.500000

Result should be true 1
Result should be false 0
Result should be true 1
```

Can you think of and write a set of different tests for the code?

Conclusion

In this worksheet, we have developed the Wallet class, which allows us to insert, check and remove money.