IB9N7 C++ for Quantitative Finance
Worksheet 11

Resources
Hints and solutions

28 January 2016
(Week 17)

Objectives for this lab session

By the end of this session, you should have completed the following:

- Work through some examples of pointers.

Exercises

Exercise 1: Writing addresses to a file

(a) Make a `vector<short>` called `a` which contains the numbers from 1 to 50. For each element write it and its address to a single line of the file `a.txt`. Note that pointers will appear in hexadecimal notation. Check you understand why the addresses of each element differ by the amount they do. What would you see with doubles?

(b) Make a vector `b` which contains the address of each element of `a`.

```
#include<fstream>
#include<vector>
int main(){
    std::ofstream ofs("a.txt");
    std::vector<short> a;
    for(short i=1; i<=50; ++i){
        a.push_back(i);
    }
    for(int& i : a){
        ofs<<i<<"␣"<<&i<<"\n";
    }

    std::vector<short*> b;
    for(int& i : a){
        b.push_back(&i);
    }
}
```

(c) Change your code so that you write to a file called `:.txt`. Note that your program will not write to this file. This is because in windows it is not permissible to have filenames containing the `:` character. However your program will run without error. This is because the `ofstream` object enters an error state where it does nothing when it cannot open the file (e.g. because the file is in use or has a bad name). You can check for this error state in several ways, for example by calling `is_open`. We saw similar things early in the course with `cin`.

Exercise 2: References simulated with pointers

Rewrite the following code using pointers in place of all references.

```
#include<iostream>
void assignDouble(int a, int& b){
    b = a+a;
}
int main(){
    int a=100, b=37, c=3;
    int& d = c;
    assignDouble(a,b);
    std::cout<<b<<"\n";
    assignDouble(d,a);
    std::cout<<d<<"␣"<<a<<"\n";
}
```

Answer:

```
#include<iostream>
void assignDouble(int a, int* b){
    *b = a+a;
}
int main(){
    int a=100, b=37, c=3;
    int* d = &c;
    assignDouble(a,&b);
    std::cout<<b<<"\n";
    assignDouble(*d,&a);
    std::cout<<*d<<"␣"<<a<<"\n";
}
```

Exercise 3: const pointers

Just like with references, pointers can be marked const to indicate that they cannot be used to change the pointee. In addition, pointers can be marked const after the * to indicate that they cannot be reassigned. For example:

```
int a = 4;
int aa = 4;
const int b = 9;

const int* c = & b;
c=&a;  //I can change c
//*c = 32; //This would be illegal

int* const d = &a;
*d = 54;  //I can change a through d
//d = &aa; //This would be illegal
c=d;  //This is fine
```

(a) Check all of this makes sense

(b) (Hard) Given

```
int ** a;
const int **b;
```

I cannot make the assignments b=a, *b=*a and *a=*b. Can you understand why all of these need to be banned? Apologies: There is a mistake in this question. *b=*a is allowed and is all right.

If I could make an `int*` point to a `const int` then I would be in trouble because I could modify the latter through the former.

```
int main(){
    int ** a;
    const int **b;

    int i = 4;
    int * pi = &i;
    int ** ppi = &pi;
    const int ci = 5;
    const int* pci = &ci;
    const int** ppci = &pci;

    a=&pi;
    b=&pci;

    //*a=*b; whoops — this would make pi point to ci

    //this is banned. if it wasn't, then the line after would make pi point to ci
    //b=a;
    *b=*ppci;

    //this is banned. if it wasn't, then the line after would make pi point to ci
    //a=b;
    *ppi=*a;
}
```

Now, if `*b=*a` was allowed, it would make `pi` point to `ci` which is bad.

`b=a` and also `a=b` are banned because they would indirectly allow the same problem to be caused. If `b` was declared as `const int * const * b;` then the assignment `b=a` is allowed.

Exercise 4: recursive structure

Consider an investment firm which divides its holdings into portfolios. Each portfolio may contain its own financial assets and other portfolios. We might represent the value of a portfolio with the following object structure:

```
#include<memory>
#include<vector>
#include<string>

using std::string;

class Portfolio{
    std::vector<std::unique_ptr<Portfolio>> m_subportfolios;
    double m_valueExcludingSubportfolios;
    string m_name;
public:
    double getValueExcludingSubportfolios() const
                {return m_valueExcludingSubportfolios;}
    string getName() const {return m_name;}
    Portfolio(string name, double valueExcludingSubportfolios);
};

Portfolio::Portfolio(string name, double valueExcludingSubportfolios)
    :m_name(name), m_valueExcludingSubportfolios(valueExcludingSubportfolios){}

int main(){
    Portfolio allAssets("total",0);
    //....
}
```

Note that I have made a small mistake in this code. There is an unnecessary copy of the name in the constructor. I should probably have taken the parameter `name` as `const string& name`.

(a) Add a member function to add a subportfolio to a portfolio. Be careful with ownership. The simplest thing to do here is to take the new name and value. The new member function might be this.

```
void addSubportfolio(const string& name, double value){
    m_subportfolios.push_back(make_unique<Portfolio>(name,value));
}
```

Alternatively, it could be left up to the caller to create the Portfolio object. In which case, because the new function gets a pointer to the object as a parameter and takes control of its ownership, it should take a `unique_ptr` (not by reference) as its parameter. Because we cannot copy one `unique_ptr` to another, we have to be a bit cunning.

```
void addSubportfolio(std::unique_ptr<Portfolio> p){
    m_subportfolios.push_back(std::unique_ptr<Portfolio>());
    //or, equivalently, m_subportfolios.push_back(nullptr);
    m_subportfolios.back().swap(p);
}
```

The effect of the `swap` is to make `p` become an empty `unique_ptr` object and the new element of `m_subportfolios` become what `p` was.

When we discuss `std::move` (for which we need `#include<utility>`), we will learn that it allows an object to be moved-from instead of copied from - i.e. it can be copied from in a destructive way. We typically use it on an object which we know we will not be using again. We could then write this. (When `std::move` is used on a `unique_ptr` we never need to write the `std::` due to argument-dependent lookup).

```
void addSubportfolio(std::unique_ptr<Portfolio> p){
    m_subportfolios.push_back(move(p));
}
```

Also, another way to implement the constructor without unnecessary copies would be to use `std::move` as follows.

```
Portfolio::Portfolio(string name, double valueExcludingSubportfolios)
    :m_valueExcludingSubportfolios(valueExcludingSubportfolios),
        m_name(move(name)){}
```

(b) Add a member function to get a reference to the portfolio's nth subportfolio.

The new member function might be this.

```
Portfolio& getSubportfolio(size_t n){
    return *m_subportfolios.at(n);
}
```

(c) Add a member function to get the total value of the portfolio.

The new member function might be this.

```
double getTotalValue(){
    double value = m_valueExcludingSubportfolios;
    for(auto& p : m_subportfolios){
        value += p->getTotalValue();
    }
    return value;
}
```

Note that `p` here has to be a reference (to a `unique_ptr<Portfolio>`).

(d) In fact, the company has portfolios A and B. Portfolio A has assets with value 4 and also contains a subportfolio called C. Portfolio B has subportfolios D and E. The value in portfolio C, D and E is 47, 2, and 343 respectively. Set all this in the main function.

See next answer.

(e) Write a function which prints for a portfolio the value of each of its subportfolios recursively.

This function can be a member or nonmember function. If it is a nonmember function, we will need to add a member function to `Portfolio` to return the total number of subportfolios. (This is probably a good idea in any case.) One way to make this pretty is to use indenting.

Having finished, the whole program might look like this. I have added the `iostream` header and the definition of `make_unique`. I have also adjusted the order of the initialisation list in the constructor to avoid a compiler warning.

```
#include<memory>
#include<vector>
#include<string>
#include<iostream>

template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(
               new T(std::forward<Args>(args)...));
}

using std::string;

class Portfolio{
    std::vector<std::unique_ptr<Portfolio>> m_subportfolios;
    double m_valueExcludingSubportfolios;
    string m_name;
public:
    double getValueExcludingSubportfolios() const
                {return m_valueExcludingSubportfolios;}
    string getName() const {return m_name;}
    Portfolio(string name, double valueExcludingSubportfolios);
    void addSubportfolio(const string& name, double value){
        m_subportfolios.push_back(make_unique<Portfolio>(name,value));
    }
    Portfolio& getSubportfolio(size_t n){
        return *m_subportfolios.at(n);
    }
    double getTotalValue(){
        double value = m_valueExcludingSubportfolios;
        for(auto& p : m_subportfolios){
            value += p->getTotalValue();
        }
        return value;
    }
    void print(size_t indentLevel){
        using std::cout;
        for(size_t i=0; i<indentLevel; ++i){
            cout<<"␣";
        }
        cout<<"Name:␣"<<m_name<<"␣Value:␣"<<m_valueExcludingSubportfolios
            <<"␣Total␣Value:␣"<<getTotalValue()<<"\n";
        for(auto& p : m_subportfolios){
            p->print(indentLevel+2);
        }
    }
};

Portfolio::Portfolio(string name, double valueExcludingSubportfolios)
    :m_valueExcludingSubportfolios(valueExcludingSubportfolios), m_name(name){}

```

```
53 int main(){
54     Portfolio allAssets("total",0);
55     allAssets.addSubportfolio("A",4);
56     allAssets.getSubportfolio(0).addSubportfolio("C",47);
57     allAssets.addSubportfolio("B",0);
58     Portfolio& b = allAssets.getSubportfolio(1);
59     b.addSubportfolio("D",2);
60     b.addSubportfolio("E",343);
61     allAssets.print(0);
62 }
```