FINM 326: Computing for Finance in C++

Lecture 4

Chanaka Liyanarachchi

January 27, 2023

Currency Converter: More OO Designs

Introduction to Templates

Currency Converter: More OO Designs

Objectives

1. *Improve* the Currency Converter program design.
2. Illustrate/learn the use of:
   - Special class member functions (constructor, destructor, copy constructor and assignment operator) are used (today)
   - Pointers (today)
   - Containers (today, week 6)
   - Error handling (week 6)
   - C++ Standard Library and, more (week 6)

Currency Converter: Change #1

- Change #1: introduce a class to represent a currency.
- Attributes (data members):
  - symbol
  - exchange rate (with respect to the US Dollar)
- Operations (member functions):
  1. a constructor/constructors
  2. get/set functions for data member
  3. `ConvertFromUSD()`: to convert a given amount from USD
  4. `ConvertToUSD()`: to convert a given amount to USD

Currency Class Definition

- Here's what we wrote last week:

```
class Currency
{
public:
   Currency(string symbol, double rate);

   double GetExchangeRate();
   void SetExchangeRate(double rate);

   double ConvertFromUSD(double value);
   double ConvertToUSD(double value);

private:
   string symbol_;
   double exchange_rate_;
};
```

- We may add more members later.

Currency Class Implementation

- We saw the constructor and get/set member functions.
- `ConvertFromUSD()` can be implemented as:

```
double Currency::ConvertFromUSD(double amount)
{
   return  amount * exchange_rate_;
}
```

- `ConvertToUSD()` can be implemented as:

```
double Currency::ConvertToUSD(double amount)
{
   return  amount * 1/exchange_rate_;
}
```

Using the Currency Class

- Converting from one currency to another currency can be done using two steps:
  1. convert the base value to USD
  2. convert from USD to foreign currency
- E.g. to convert from CAD to AUD:
  1. Instantiate a CAD currency object:
     ```
     Currency c1("CAD", 1.3235);
     ```
  2. Use it to convert the CAD amount to USD:
     ```
     double cadValue = 100;

     double usdValue = c1.ConvertToUSD(cadValue);
     ```
  3. Instantiate a AUD currency object:
     ```
     Currency c2("AUD", 1.1);
     ```
  4. Use it to convert the USD amount to AUD:
     ```
     double audValue = c2.ConvertFromUSD(usdValue);
     ```

OO Currency Converter: Version 1

What can we say about this version:

- Maintainability?
- Efficiency and performance?

The CurrencyFactory class

- Change #2: relocate `Currency` object creation to a class (`CurrencyFactory`) .
- Use a `CurrentyFactory` object to create `Currency` objects.

The CurrencyFactory class

- The `CurrencyFactory`:
  - Here we use the compiler generated default constructor.[1]
  - Supports function to create a `Currency` object.

```
class CurrencyFactory
{
public:
   Currency GetCurrency(int currencyType);
};
```

[1] compiler may generate other special member functions, more on this later (week 6)

- We can explicitly ask the compiler to generate default constructor:

```
class CurrencyFactory
{
public:

   CurrencyFactory() = default;

   Currency GetCurrency(int currencyType);

};
```

- `GetCurrency()` creates and returns a new currency object:

```
Currency CurrencyFactory::GetCurrency(int currencyType)
{
   switch(currencyType)
   {
      case EUR:
         return Currency("EUR", 0.9494);
      case GBP:
         return Currency("GBP", 0.8381);
      .....
      .....
      default:
         return Currency("USD", 1.0);
   }
}
```

Using the CurrencyFactory

- Now, we use a `CurrencyFactory` object to create currency objects:

  ```
  int main()
  {
     CurrencyFactory factory;

     Currency cad =
                factory.GetCurrency(CAD);

     double amount = 100;

     cad.ConvertFromUSD(amount);
  }
  ```

- What are the pros and cons fo this solution?

- New design has some pros and cons.
- Pros:
  - Adding/changing a currency is easy – improves maintainability
  - We can use this class in other projects – promotes code reuse
  - Allows us to make further improvements
- Cons: Creates a new `Currency` object every time we call `GetCurrency()`.

Change #3

- Creating an object can be expensive.
- There's no need to create the same `Currency` object more than once.
- We can store `Currency` objects in a container (e.g. array) and reuse them.
- When should we create them?
- We have two options:
  - Create currencies ahead of time (eager initialization)
  - Create one when we need it for the first time (lazy initialization)
- What are the pros and cons of each (eager vs. lazy) method?

- Suppose we decide to use eager initialization.
- Let's create the currencies in the `CurrencyFactory` constructor.
- A constructor is usually used to do similar initializations.
- We cannot use the compiler generated default constructor anymore.

```
class CurrencyFactory
{
public:
   CurrencyFactory();
   Currency GetCurrency(int currencyType);

private:
   Currency currencies_[5];
};
```

- Note: We create an array of currencies using the default constructor; Currency class needs a default constructor now.

New CurrencyFactory Constructor

- Let's create the currency objects and store them in the array:

```
CurrencyFactory::CurrencyFactory()
{
    currencies_[USD] = Currency("USD", 1.0);

    currencies_[EUR] = Currency( "EUR", 0.9494);

    currencies_[GBP] = Currency("GBP", 0.8163);

    ......
    ......
}
```

- Now, `GetCurrency()` returns an existing `Currency` object:

```
Currency CurrencyFactory::GetCurrency(int currencyType)
{
   return currencies_[currencyType];
}
```

- Note how we use the `currencyType` (int) as the index.
- What do you think about this solution?

Exercise

- Write constructors, copy constructor and assignment operator for Currency class.
- Add a debug statement (cout) in each to indicate when each method is used.
- Observe object creation/copying/assignment/destruction.
- Are these operations necessary?

Automatic Objects and Free-Store Objects

Automatic Objects

- Objects we've looked at so far:
  - created at declaration
  - destroyed when they go out of scope (i.e. exit the block of code)
  - lifetime directly tied to the scope
- They are called *automatic* objects.
- Easy to use: Created and destroyed automatically.

Free Store Objects

- What if we need to manage the lifetime of the objects?
    1. destroy an object when we no longer need it
    2. need an object to live beyond its scope
- We can use free-store (dynamic) objects.

Creating Free-Store Objects

- We use the operator `new` to create an object on the free-store.
- The operator `new`:
  - uses a constructor to create an object
  - returns the address of the object

  ```
  Currency* c = new Currency("USD", 1.0);
  ```

- This object is not destroyed until we *delete* it, using the operator `delete`.

  ```
  delete c;
  ```

- If we don't/forget to delete a free-store object, we have a *memory leak*.
- Programmer has the responsibility to create and properly destroy free-store objects.
- It is not trivial to manage the lifetimes of free-store objects properly in large, complex programs.

Accessing Free-Store Members

- We use the dereference operator (*) or the `->` operator to access free-store members:

  ```
  Currency* c = new Currency("EUR", 0.9494 );

  (*c).GetSymbol();
  (*c).SetExchangeRate(0.95);
  ```

- Alternatively, we could write the same code above as:

  ```
  c->GetSymbol();
  c->SetExchangeRate(0.95);
  ```

- Common practice is to use `->` with free-store objects.

CurrencyConverter: Change #4

- Let's use free store objects:

```
CurrencyFactory::CurrencyFactory()
{
   currencies_[USD] = new Currency("USD", 1.0);

   currencies_[EUR] = new Currency( "EUR", 0.9494);

   currencies_[GBP] = new Currency("GBP", 0.8163);

    ...........
    ...........
}
```

- Now, `GetCurrency()` just returns the address of an already created `Currency` object:

```
Currency* CurrencyFactory::GetCurrency(int currencyType)
{
   return currencies_[currencyType];
}
```

- New `CurrencyFactory`:

```
class CurrencyFactory
{
public:
   CurrencyFactory();
   Currency* GetCurrency(int currencyType);

private:
   Currency* currencies_[5];

};
```

Using the New CurrencyFactory

- Now, we use a `CurrencyFactory` object to create currency objects:

```
int main()
{
   CurrencyFactory factory;

   Currency* cad =
               factory.GetCurrency(CAD);

   double amount = 100;

   cad->ConvertFromUSD(amount);
}
```

Exercise

- Observe object creation/copying/assignment/destruction.
- Now, we don't:
  - default construct `Currency` objects
  - assign `Currency` objects
  - copy construct `Currency` objects
- Eliminated *unnecessary* operations.
- It is very efficient to pass a pointer around than to pass an object around.

CurrencyConverter: Change #5

- We have another problem. We have a memory leak.
- We need to delete the `Currency` objects.
- We can delete them in the `CurrencyFactory` destructor.

- We use the destructor to delete the free store objects:

```
CurrencyFactory::~CurrencyFactory()
{
  for (int i=0; i<5; ++i)
  {
      delete currencies_[i];
  }
}
```

- We use the destructor of a class for similar resource cleanups.

CurrencyConverter: Changes So Far

- We have a better program now:
  - maintainability
  - efficiency and performance
- Concepts we have discussed and illustrated are more important.
- We will introduce some more new concepts and make more changes to this program later (week 6).

Introduction to Templates

Templates: Introduction

- Templates allow us to write functions and classes with types as parameters.
- They are known as *parameterized* classes/functions.
- Templates provide another way to write efficient, readable and reusable code.
- Most classes and functions in the Standard Library use templates.

Generic Programming using Templates

- When we use templates we use a generic type (or types).
- A generic type placeholder for an actual type.
- Templates are used as:
  1. function templates: to write functions that work with different types
  2. class templates: to write classes where the member variables can be different types

Function Templates

Function Templates: Example 1

- Let's look at the function, which add two `integers`, below:

```
int add(const int a, const int b)
{
  return a+b;
}
```

- Now, if we want to add two `std::string` objects, we need to write another function:

```
double add(const double a, const double b)
{
  return a+b;
}
```

- We have common code in both functions – only difference is the data types.

- Using function templates we can write only one function that works with *any* type:

```
template <typename T>
T add(const T a, const T b)
{
    return a+b;
}
```

- A parameterized function (e.g. `add()`) represents a family of functions.
- The compiler generates the appropriate function for the type/types that are used.

Using Function Templates

- The compiler can automatically deduce the types for the following two cases:
  ```
  int res1  = add(1, 2);

  double res2 = add(1.2, 2.3);
  ```

- We can explicitly tell the compiler what the types are (in this case it is not necessary):
  ```
  int res1  = add<int>(1, 2);

  double res2 = add<double>(1.2, 2.3);
  ```

- For the next case, we must tell the compiler types explicitly, as it cannot deduce the correct type:
  ```
  double res3 = add<double>(1, 1.2);
  ```

Templates: New Keywords

- We introduced a new keyword, `typename`: we use it to inform the compiler T is a generic type.
- We can use `class` instead of `typename`, both have the same meaning in this case.
  ```
  template <class T>
  ```
- Programmers tend to use T as the generic type letter, but you can use any other letter (or any word that's not a keyword in C++).

FunctionTemplates: Example 2

- We can provide two or more `typenames` to pass the parameters and a return `typename`.
- What is the return type here?

```
template<typename T1, typename T2, typename T3>
const T1 add(const T2& a, const T3& b)
{
   return a + b;
}

double value = add<double, int, double>(2, 3.1);
```

- T2 and T3 are typenames of the arguments
- T1 is the return type

Function Templates: Example 3

- Let's implement a function to return the max value:

```
template <typename T>
T max(const T& a, const T& b)
{
  if (a > b) return a;
  else return b;
}
```

- We can write the same code using a syntax known as *immediate-if*[2]:

```
template <typename T>
T max(const T& a, const T& b)
{
  return a > b ? a : b;
}
```

[2]this syntax is not related to templates

Exercises

Implement following functions using templates:

1. `min()`: returns the smaller of two values
2. `swap()`: swaps two values

Class Templates

Class Templates

- Suppose we want to a *vector* ($Nx1$ matrix) for various types:

```
class IntVector
{
   .......
private:
   int values[10];
};

class StringVector
{
   .......
private:
   string values[10];
};
```

Class Templates

- Class templates allow us to have data members of parametric types in a class.
- We can write one vector ($Nx1$ matrix) class using templates:

```
template <typename T>
class SimpleVector
{
   .......
private:
   T values[10];
};
```

- A parameterized class represents a family of classes.
- To use:

```
SimpleVector<int> v1;

SimpleVector<double> v2;
```

- The compiler generates classes for actual types we use.

Organizing Template Source Code

- We need to organize template code differently to enable compile time template instantiation.
- We don't have time to discuss this topic in detail.
- Instead, I will just outline 2 popular approaches:[3]
    1. *inclusion* technique: write the implementations in the header where a class/function is declared.
    2. explicit instantiation: requires instantiating the classes/functions for the types used.

[3] See class demo for details