FINM 326: Computing for Finance in C++

Lecture 6

Chanaka Liyanarachchi

February 10, 2023

The C++ Standard Library

STL Containers

Error Handling

Smart Pointers

I/O Streams

The C++ Standard Library

The C++ Standard Library

- The C++ Standard Library is a collection of classes and functions:
  - useful features
  - designed by experts
  - efficient implementations
- We've already seen some examples:
  - `string` class - defines a convenient type to work with a string of characters.
  - `iostream` class - helps us read inputs from the keyboard and write output to console.

- Features provided by the C++ Standard Library include:
  1. The Standard Template Library (STL), a collection of containers, iterators and algorithms.
  2. Support for input and output using stream classes.
  3. Support for numerical work, e.g. random number generation and distributions.
  4. Smart pointers.
  5. Error handling.
  6. Utility classes.
  7. and, more ...
- An in depth study of the C++ Standard Library is beyond the scope of this course.

The Standard Template Library (STL)

The Standard Template Library (STL)

- The STL is a core piece of the C++ Standard Library.
- Provides a collection of containers, iterators and algorithms.

STL Containers

- We use containers to organize data.
- We have seen the array already.
- The Standard Library supports several containers:
  - different features/usage
  - different performance
- We will look at 2 STL containers:
  - std::vector
  - std::map

std::vector

- It is like an array but *smarter*, and supports more features.
- Size is not fixed:
  - Not required to specify size at creation time.
  - Manages the size dynamically.
- `vector` is defined in `<vector>`.
- `http://www.cplusplus.com/reference/vector/vector/`

vector: Example 1

- Code below shows how to insert elements using `push_back()`.
- We use the default constructor (note that we're not specifying the size of the vector here).

```
#include <vector>

using std::vector;

int main()
{
  vector<int> v;

  for (int i=0; i<=10; ++i)
  {
      v.push_back(i);
  }
}
```

- Uses templates, i.e. takes the type as a parameter.
- Initially, the vector is empty.
- `push_back()` creates space (if necessary) and inserts an item.
- Creating space is an expensive operation (more on this later).

vector: Example 2

- Code below shows how to insert elements using an index.
- We can specify the initial size when the vector is created.

```
#include <vector>

using std::vector;

int main()
{
  vector<int> v(10);

  for (int i=0; i<10; ++i)
  {
      v[i] = i;
  }
}
```

Maps/Dictionaries

- An array/vector allow us to use an integer index to access an item.
- What if we need to use an index of a different type (e.g. `string`)?
- Why would we want to do that?
- Currency-Converter is an example: using an int/enum for currency type can be confusing and error prone.
- Why are we using an int currency type?
  1. to use switch (switch not used anymore)
  2. to index into the array of Currencies

std::map

- Stores a key (index) and a value.
- The value is associated with the key.
- A map maintains items in a certain (i.e. sorted) order.
- The key[1] is used to order the sequence.
- The key has to be unique, i.e. one key can have only one value.
- Defined in `<map>`.
- `http://www.cplusplus.com/reference/map/map/`

[1] should be comparable

map: Example 1

- Example shows using a map with different key/value types:

```
#include <map>
using std::map;

int main()
{
   map<unsigned int, string> zipcodes;

   zipcodes[60604] = "Chicago";
   zipcodes[60637] = "Hyde Park";

   map<string, unsigned int> population_in_cities;
   population_in_cities["Hyde Park"] = 10;
   population_in_cities["Chicago"]   = 12;

}
```

- Template arguments:
  - First type: type of key
  - Second: type of value

std::pair

- Another option is to use an std::pair to insert a key/value pair.
- A pair is defined in the Standard Library similar to:

```
template <typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
    ......
    ......
};
```

- We can create a pair and insert it:

```
std::pair<unsigned int, string> p(60606, ''Chicago'');
zipcodes.insert(p);
```

- Another way is to use `std::make_pair()`:

```
zipcodes.insert(std::make_pair(60606, ''Chicago''));
```

- Defined in `<utility>`.

Iterators

- We use an iterator to go over elements in a container.
- An iterator *looks like* a pointer:
  - Represents a certain position in a container
  - operator ***** : returns the element at the current position
  - if the element has members, use the -> operator to access the members

Navigating Containers using Iterators

- All container classes provide member functions to help navigate over the elements:
  - `begin()` returns an iterator pointing to the beginning (first element) of the elements in the container
  - `end()` returns an iterator pointing to the end of the elements in the container – the end is the position after the last element, known as the *past-the-end iterator*
  - and, more..
- This construction allows us to:
  - write simple loops – we can iterate over a collection until `end()` is not reached
  - simplifies handling empty *ranges*

Iterator: Example 1

- We can access each element in a `vector<int>` as:

```
std::vector<int> myvector;

//add some elements (code not shown)

for (vector<int>::iterator iter=myvector.begin();
              iter != myvector.end();
             ++iter)
{
    cout << *iter << endl;
}
```

- Fortunately, we don't have to type such long types in modern C++.

Detour: The auto Keyword

- The `auto` infers the type of a variable from an initialization expression:
  ```
  auto x = 1;   //int x = 1;
  ```
- auto is not a type; `x` above is an int
- Another example:
  ```
  auto y = 1.0; //double y = 1.0;
  ```
- We can use additional qualifiers with *auto*:
  ```
  auto& rx = x;

  const auto z = 3;
  ```
- Using *auto* has some advantages:
  1. write concise, flexible code
  2. reduce errors as all variables are initialized
  3. useful when the types of an expression is complex
- auto has other features but we won't discuss them in this course.

Iterator: Example 2

- An iterator points to an element in a container.
- We can use `auto` to access all elements in a `vector<int>` as:

```
for (auto iter=myvector.begin(); iter != myvector.end();
             ++iter)
{
    cout << *iter << endl;
}
```

- To iterate in reverse:

```
for (auto iter=myvector.rbegin(); iter != myvector.rend();
             ++iter)
{
    cout << *iter << endl;
}
```

Iterator: Example 2

- A element in a map is a pair, i.e. has two values, the key and the value, given by first and second members.
- The key is accessed using the first and the value is accessed using the second member as shown below:

```
for (auto iter=mymap.begin(); iter != mymap.end();
             ++iter)
{
   cout << ''key:''<< iter->first <<  '',value:''<< iter->second<< endl;
}
```

Change #6: Using a Map

- We can use a std::string for the currency-type now. We need to use a map to store currencies.
- We could define the map to store Currency objects as:

```
map<string, Currency*> currencies_;
```

- Use constructor to create free-store Currency objects and store them in the map:

```
currencies_["USD"] = new Currency( "USD", 1.0);
currencies_["EUR"] = new Currency( "EUR", 0.9494);
....
```

- Delete the free-store objects in the destructor:

```
for (auto iter=currencyMap_.begin();
   iter != currencyMap_.end(); ++iter)
{
   delete iter->second;
}
```

- We can use a string index (i.e. currency symbol) to find the currency:

  ```
  return currencies_[currencySymbol];
  ```

- Better approach is to use find method:

  ```
  auto iter = currencies_.find(currencySymbol);
  return iter->second;
  ```

- Suppose the currency we're looking for is not there.
- iter in this case points to `currencies_.end()`
- How should we handle that case in this program?

Error Handling

Error Handling

- Error handling is very important:
  - correct behavior
  - reliable and robust programs
- Error handling is complex – no single error handling mechanism that works for all problems.
- Different kinds of errors require different error handling techniques:
  - Invalid inputs: check to make sure if the inputs are valid
  - Resource management: RAII (Resource Acquisition Is Initialization) technique
  - Unexpected errors that cannot be handled at the point of detection: use exceptions

Exceptions

Error Handling Using Exceptions

- Very often the location (part of the program) where an error is detected may not know how to handle the error:
  - CurrencyFactory does not know how to handle a missing currency.
  - Input-file/database (later) not accessible to read currency rates.
  - Try to read exchange rates from yahoo/bloomberg but network down.
  - ...
- Exceptions allow us to propagate the error back to a place where it can be handled.

New Keywords

Using exceptions involve some of the following actions:

1. Raising the exception:
   - `throw`: raises/throws the error
2. Handling the exception:
   - `try`: encloses a block of code (one or more statements) that might throw an exception
   - `catch`:
     - one or more catch block should immediately follow a try block
     - each catch block specifies the type of exception it can handle
     - when an exception is thrown from a try block, the control reaches the appropriate catch block

Change #7: Using Exceptions

- We can modify `GetCurrency()` function to throw an exception (std::runtime_error) when a Currency is not found.

```
Currency* CurrencyFactory::GetCurrency(string currencySymbol)
{
   auto iter = currencyMap_.find(currencySymbol);

   if (iter == currencyMap_.end())
   {
      throw std::runtime_error("Currency not found");
   }

   return iter->second; // same as currencyMap_[currencySymbol]
}
```

- Now, we need to catch an exception it is thrown by `GetCurrency()`:

```
int main()
{
   try
   {
       CurrencyFactory factory;

       Currency* currency =
              factory.GetCurrency("BTC");

       double amount = 100;

       currency->ConvertFromUSD(amount);
   }
   catch(const std::runtime_error& e)
   {
        //handle error
        cout << e.what() << endl;
   }
}
```

- Here we're using `std::runtime_error` (defined in `<stdexcept>`) to throw the error.
- We will look at other exception types and their relationships later (week 9).

Handling Missing Currency

- We can use an exception to handle the missing currency case as shown above.
- We generally use exceptions for errors that occur rarely.
- We have another feature in C++ that we can use in this situation.
- The idea here is that the `GetCurrency()` may return a `Currency` if the currency-type is known, otherwise or may not return a valid `Currency` at all.

std::optional

- We use `std::optional` in situations where we may or may not have a value, as the case in `GetCurrency()`.
- A variable of this type can have a value or not.
- The absence of a value is indicated using `std::nullopt`.
- Available in C++20 (most recent C++ specification as of today).

```
std::optional<int> val1 = 2;
std::optional<int> val2 = std::nullopt;
```

- We can check if the variable is assigned a value before using it:

```
if (val1)
{
    cout << ''value: '' << *val1 << endl;
}
```

- Defined in `<optional>`.

Change #8: Using std::optional

- We can modify `GetCurrency()` function to return an std::optional.

```
std::optional<Currency*> CurrencyFactory::GetCurrency(string curre
{
   auto iter = currencyMap_.find(currencySymbol);

   if (iter == currencyMap_.end())
   {
      return std::nullopt;
   }

   return iter->second; // same as currencyMap_[currencySymbol]
}
```

- Now, we check to see if the return value from `GetCurrency()` contains a value before we use it:

```
int main()
{
   CurrencyFactory factory;

   std::optional<Currency*> currency =
          factory.GetCurrency("BTC");

   double amount = 100;

   if (currency)
   {
       (*currency)->ConvertFromUSD(amount);
   }
}
```

Smart Pointers

Pointers

- We often hear programmers say "pointers are hard".
- We have not seen anything in this course so far to say pointers are hard.
- However, using pointers in large programs is difficult:
  - Copying pointers is tricky (need to pay attention to shallow vs deep copying).
  - Several poiners may point to the same object. Figuring out when to delete a free-store object can be tricky:
    - if we delete too early: program will crash
    - if we delete too late: wasting memory
    - if we do not delete: memory leak
  - Exceptions complicate things – exceptions alter program flow.
- Fortunately, we don't have to use *raw pointers*. We have better tools in modern C++.

Using a Shared Pointer (Smart Pointer)

- A shared_ptr provide a way to:
  - obtain all the benefits of using pointers
  - copying is trivial
  - we do not delete them – they are automatically deleted
  - no *significant* performance overhead
- We use `std::make_shared()` to create a `std::shared_ptr`.
- Defined in `<memory>`.

Change #9: Using a Shared Pointer

- Modify the map to store a shared_ptr:

```
map<string, shared_ptr<Currency>> currencies_;
```

- Populate Currency objects in the constructor:

```
currencies_["USD"] = make_shared<Currency>( "USD", 1.0);
currencies_["EUR"] = make_shared<Currency>( "EUR", 0.9494);
....
```

- Now we don't delete any (free-store) objects in the destructor. The smart pointer takes care of cleaning up the free store object when it is no longer used.
- This is an example of RAII where an automatic object is used to represent a resource (in this case memory allocated on the free-store) so that the automatic object's destructor will release the resource when appropriate.[2]

[2] we will see another example of RAII when we discuss our next topic - file operations

I/O Streams

Stream Classes

- We use streams, i.e. sequences of bytes, to do I/O.
- Input: A sequence of byte flow coming into the computer memory from a device (e.g. keyboard, file).
- Output: A sequence of byte flow from the memory to an output device (e.g. console, file).
- To handle streams we have stream classes (in the Standard Library):
  - encapsulate a stream/sequence of data
  - define operations
- In this section we will look at:
  1. input/output using standard input/output ✓
  2. file input and output
  3. working with strings

Standard Input/Output

- The iostream section of the standard library defines objects of types istream and ostream which are used to read and write to standard I/O channels:
  - `cin`
  - `cout`
  - `cerr`
  - `clog`
- Ref:
  - istream: `http://www.cplusplus.com/reference/istream/istream/`
  - ostream: `http://www.cplusplus.com/reference/ostream/ostream/`

`cin`:

- a predefined object of instance of istream class
- attached to the standard input device, which usually is the keyboard
- uses the stream extraction operator, >>

```
int x;
cin >> x;
```

`cout`:

- a predefined object of instance of ostream class
- attached to the standard output device, which usually is the console
- uses the stream insertion operator, <<

```
cout << "hello";
```

`cerr`:

- a predefined object of instance of ostream class
- attached to the standard error device, which usually is the console
- not buffered
- uses the stream insertion operator, <<

```
cerr << "error";
```

`clog`:

- a predefined object of instance of ostream class
- attached to the standard log device, which usually is the console
- uses the stream insertion operator, <<

```
clog << "hello";
```

- In large programs, it is better to use different output devices to log different message types:
  - output messages to `cout`
  - log messages to `clog`
  - error messages to `cerr`
- Helps us identify errors/logs etc. clearly/fast.
- To avoid the overhead of responding to each call, some objects use buffers to accumulate the streams before writing them to the output device.
- Buffers can be flushed to the output device using `flush()`.

File I/O

- Following streams are used to access files:
  - `ofstream`: to write to files
  - `ifstream`: to read from files
  - `fstream`: to both read and write from/to files
- References:
  - `ofstream`: `http://www.cplusplus.com/reference/fstream/ofstream/`
  - `ifstream`: `http://www.cplusplus.com/reference/fstream/ifstream/`
  - `fstream`: `http://www.cplusplus.com/reference/fstream/fstream/`
- Defined in `<fstream>` .

Using Files

- We create the appropriate stream object and associate it with the filename.
- A file is automatically opened at stream construction time if the filename is passed as an argument.
- A file is automatically closed at stream destruction time.
- This is another example of RAII.

Example 1: Writing to a Text File

```
//name of the output file
string filename = "greetings.txt";

//open output file for writing
ofstream outfile(filename);

if (outfile)  //testing to see if the file is open
{
   outfile << "hello " << endl;
   outfile << "world" << endl;
}
else //we couldn't open open the file
{
   cerr << "Unable to open file " << filename << endl;
}
```

Example 2: Reading from a Text File

```
//name of the input file
string filename = "greetings.txt";

//open file for reading
ifstream infile(filename);

if (!infile)  //testing to see if the file is open
{
   cerr << "Unable to open file " << filename << endl;
   return;
}

//read the file one line at a time
string data;
while(infile)
{
   getline(infile, data);
   ...
   ...
}
```

Strings

- The `string` class is a special container, used to store a sequence of characters.
- `http://www.cplusplus.com/reference/string/string/`
- `std::string` is usually the better choice than C-style strings to handle a sequence of character:
    1. algorithm support
    2. stream class support

Streams for Strings

- Provides a way to manipulate strings easily
  - `istringstream`: input operations on strings
    http://www.cplusplus.com/reference/sstream/istringstream/
  - `ostringstream`: output operations on strings
    http://www.cplusplus.com/reference/sstream/ostringstream/
  - `stringstream`: input and output operations on strings
    http://www.cplusplus.com/reference/sstream/stringstream/
- Defined in `<sstream>` .

Example: Streams for Strings

- `ostringstream` example:

```
std::ostringstream oss;
oss << "USD ";
oss << 1.0 << ends;

cout << oss.str() << endl;
```

- `istringstream` example:

```
string s = "USD 1.0";
std::istringstream iss(s);

string symbol; double rate;
iss >> symbol >> rate;
```

Change #10: Using Streams

- Read the exchange rates from a file (use std::fstream or std::ifstream).
- Use std::istringstream to read the inputs.
- Throw an exception if the input file is not available.

Change #11: Using Default Implementations

- We don't need to implement some special member functions.
- Instead, we can use the compiler generated copy constructor, assignment operator and destruction for Currency class.

```
Currency(const Currency& other) = default;

Currency& operator=(const Currency& other) = default;
```

Currency class - Output Stream Operator

- We saw an example of operator overloading the assignment operator (operator=) last week.
- We can overload other operators.
- Let's overload the `operator<<` for the Currency class, so we can write:

```
Currency c(''EUR'', 1.2);
cout << c;
```

- We need to overload the following operator:

```
std::ostream& operator<<(std::ostream& os);
```

- Suppose we declare this as a class member function. We would have to use it as follows:

```
Currency c(''EUR'', 1.2);
c.operator<<(cout)
or, as:
c << cout;
```

- This usage is contrary to C++ convention and may confuse users. Remember, others use code we write.
- How do we address this problem?

- Solution is to make operator<< a non-member function:
  `std::ostream& operator<<(std::ostream& os, const Currency& c);`
- This function cannot access private members of the `Currency` class directly.
- One solution is to use public get functions of `Currency` in the overloaded function.
- Another solution is to allow access to non public members by making the overloaded function a friend of the `Currency` class.

Friend Functions

- A class can allow another class or a function to access its non public members by marking that class or the function a `friend`.
- We make functions such `operator>>` and `operator<<`, which are not members of the class but need to access the non public members of the class friend functions; these functions are not class members, they are *conceptually* part of the class interface.
- We should use friend functions only when it is necessary – to quote famous C++ author Scott Meyers: *Whenever you can avoid friend functions, you should, because, much as in real life, friends are often more trouble than they're worth*.

Change #12: Operator Overloading

- We can declare operator« as a friend function of Currency class:

```
class Currency
{
public:
....

friend std::ostream& operator<<(std::ostream& os, const Currency& c);

.....
.....
};
```

- We implement this as a non member function:

```
std::ostream& operator<<(std::ostream& os, const Currency& c)
{
     os << c.symbol_ << '':'' << c.rate_;
     return os;
}
```

- Now, we can use « operator on a Currency object:

```
Currency c(''EUR'', 1.2);

cout << c;
```

Currency-Converter: Wrap-up

- Used this example to illustrate many important topics in C++:
  - functions and classes
  - control structures
  - arrays, STL contrainers and iterators
  - pointers and smart-pointers
  - error handling (std::optional and exceptions)
  - file I/O
  - streams
  - good design: clarity, reusability, and maintainability
  - efficiency and performance
- We can use this example to introduce more new features.
- Due to time constraints, we stop this example here.

Assignment 3 (Graded)

- Write an OO version of the Currency Converter program.
- Use what you did for Assignment 2 as the starting point.
- Introduce at least 5 changes discussed above (changes #2–#11):
  - Pay attention to writing reusable and maintainable code
  - Pay attention to performance (i.e. don't use unnecessary operations)
- Use comments to clearly identify your changes.
- Your program should be able to convert a given value from one currency to any other currency.
- Your program should be able to convert more than one conversion in a single program run.
- Individual Assignment.
- Due: February 17 (Friday) by midnight (CST).

Week 1-6: Recap

- We discussed:
  - Fundamental types
  - Functions
  - Control structures
  - References and pointers
  - Arrays and (some) STL containers
  - Classes
  - Error handling
  - File I/O
  - Writing *good code: L3.pdf: slide #4*
    Illustrated every point using examples, except extensibility.

What's next?

- We will introduce new concepts, and discuss how to write extensible code.
- Discuss new applications.