FINM 326: Computing for Finance

Lecture 3

Chanaka Liyanarachchi

January 20, 2023

Object Oriented Programming

User Defined Types

Example: OO Currency Converter

Special Member Functions

Object Oriented Programming (OOP)

Why OOP

Our civilization depends critically on software; it had better be quality software - Bjarne Stroustrup.

- We need better/more tools to write *good* software:
  - Correctness
  - Efficiency and Performance
  - Clarity and Readability:
    - write clear and easy to understand code
    - use proper language features/constructs
  - Reusability and Maintainability:
    - Use code written by others – enhances productivity
    - Use code known to work – reduces new bugs
    - Use common code – maintain less code
  - Extensibility:
    - Adding extra functionality to existing code for a specific use
- We introduce:
  1. OOP
  2. Generic programming, to give us more tools.

What is OOP

- Main Concepts - Roadmap:
  1. Classes and Objects
  2. Data Abstraction and Encapsulation
  3. Inheritance
  4. Polymorphism

Classes and Objects

Classes

- We've seen some fundamental data types.
- C++ defines types such as `int`, `short`, `char` and `float` etc.
- C++ also defines the operators for the fundamental types.
- Are the funtamental types and operators enough?

Why We Need Classes

- Suppose we want to represent a *student* in a program.
- We **may** use some of the attributes below (and more):
  - name (string)
  - age (int)
  - gender (char)
  - occupation (string)
  - email (string)
  - address (string)
  - citizenship (string)
- To represent complex types we use user defined types.
- A class is a user defined type.

Class Design

- Defining a type requires:
  1. The attributes (data members/member variables) to represent that type
  2. Appropriate operations (member functions)
- Attributes and operations depend on the needs of the program.

Example: Representing a Currency

- Suppose, we want to write a class to represent a Currency in the Currency-Converter program. What attributes should we use?
  1. symbol
  2. exchange-rate (e.g. relative to USD)
  3. country?
  4. anything else?
- What operations should we use?
  1. We may want to read the data members:
     - get exchange-rate
     - get symbol
     - get country?
  2. We may also want to change some data members:
     - set exchange-rate
     - anything else?
  3. Any other operations?

Representing a Currency in C++

Writing a class involves two steps:

1. Define/declare the class members (data and function)
2. Implement member functions

Currency Class Members

- Let's use the following data members:
  - symbol: use a `std::string`
  - exchange-rate: use a double
- We use following operations to read/get the member variables:
  - To read/get symbol: `string GetSymbol();`
  - To read/get rate: `double GetExchangeRate();`
- We use following operations to change/set the data members:
  - To set rate: `void SetExchangeRate(double rate);`

The Currency class

- Here's what we have so far:

```
class Currency
{
   string symbol;
   double exchange_rate;

   string GetSymbol();
   double GetExchangeRate();

   void SetExchangeRate(double rate);
};
```

- This defines a new type and a new scope.
- We are not done - this class is not complete.

Protection Levels

- Important OOP principle: Keep the data members internal to the class; allow outside access to data members using member functions.
- We use the *protection levels* to indicate what members (data and function) are internal to the class and what members are available to the outside world.
- There are three protection levels:
  1. `public`: anyone can access a `public` member (data/function) of a class.
  2. `private`: only the class and friend functions (week 6) can access private members.
  3. `protected`: we will discuss later (week 7)
- We keep the data members internal to the class by labeling them `private`.
- We allow the users to access the data members of the class via the `public` member functions.

Example: Protection Levels

- Let's add the protection levels. Now we have:

```
class Currency
{
private:
    string symbol;
    double exchange_rate;

public:
   string GetSymbol();
   double GetExchangeRate();

   void SetExchangeRate(double rate);
};
```

- We are not done yet.

Classes and Objects: Terminology

Class: code we write to introduce a new type.
Object: we create an instance of a class in a program:

- One Currency object for USD
- Another Currency object for CAD

The Constructor

- A constructor is used to initialize the data members (and do any other initializations) when an object is created.
- The constructor:
  - is a special class member function
  - has the name of the class
  - does not return any value
- We can overload:
  - we can have more than one constructor for a class
  - must take different argument types (same as overloading functions)
- A constructor that does not take any argument is called a **default constructor**.
- The compiler may generate a default constructor if any other constructor is not available.

Currency class: Constructor

- Let's write two constructors for illustration:
  1. This constructor does not take any arguments:
     `Currency();`
  2. This one takes 2 arguments:
     `Currency(string symbol, double rate);`

The Destructor

- There's another special member called a destructor.
- The destructor is called when an object is destroyed.
- We write the `Currency` class destructor as `~Currency()`
- A destructor is usually used to free up *resources* when an object is destroyed.

Currency Class Definition

- Now, we have:

```
class Currency
{
private:
   string symbol;
   double exchange_rate;

public:
   Currency();
   Currency(string symbol, double rate);

   ~Currency();

   string GetSymbol();
   double GetExchangeRate();

   void SetExchangeRate(double rate);
};
```

Detour: Coding Standard/Style

- Let's briefly talk about a *good practice* in software development known as *coding standard/style*.
- A coding standard establishes a coding style, including a project/firm specific way to:
  - Naming conventions
  - Good programming practices/tips
  - ...
- Good coding standards:
  - Improve clarity/readability.
  - Improve maintainability
  - Encourage/introduce good software practices.

- Coding styles are usually specific to organizations/projects.
- Google style is an example:
  `https://google.github.io/styleguide/cppguide.html`
- We will use a very simple coding standard in this course:
  1. To understand each others code easily.
  2. To introduce/learn an important practice.

Computing for Finance: Coding Style

1. We write each class definition in a separate header file.
2. Use include guards (discussed later).
3. We implement each class in a separate .cpp file.
4. First letter of the class name is uppercase, e.g. `Student`.
5. `public` member functions start with a upper case letter.
6. `private` members, see google standard.
7. Member variable names end with _ (underscore), e.g. `exchange_rate_`.
8. For anything else, you may use Google style.

Currency Class Definition

- Using our coding style, we have:

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

   string GetSymbol();
   double GetExchangeRate();

   void SetExchangeRate(double rate);

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

- Write this class definition in `Currency.h` file.

Currency Class Implementation

- The class implementation needs to see the definition of the class – we *include* the header file.

```
#include "Currency.h"
```

- The `GetExchangeRate()` function retuns a copy of the `exchange_rate_` member variable:

```
double Currency::GetExchangeRate()
{
  return exchange_rate_;
}
```

- The `SetExchangeRate()` function changes the `exchange_rate_` member variable:

```
void Currency::SetExchange(double exchange_rate)
{
   exchange_rate_ = exchange_rate;
}
```

- Member functions are defined in the class scope as indicated by `Currency::`
- We can implement other member functions in a similar way.

Currency Class: Constructors

- We initialize data members to empty strings/zero in the default constructor. Why?

```
Currency::Currency()
{
   symbol_ = "";
   exchange_rate_ = 0.0;
}
```

- Second constructor uses the arguments to initialize the data members:

```
Currency::Currency(string symbol,
   double exchange_rate)
{
    symbol_ = symbol;
    exchange_rate_ = exchange_rate;
}
```

- Two constructors above do not initialize the member variables efficiently.
- Constructors should initialize the member variables using **member initializer lists**.
- Initializer lists allow us to initialize variables at the time they are created.

```
Currency::Currency()
      :  symbol_(""),
         exchange_rate_(0.0)
{
}

Currency::Currency(string symbol,
   double exchange_rate)
    :  symbol_(symbol),
       exchange_rate_(exchange_rate)
{
}
```

Currency Class: Destructor

- This destructor doesn't do much right now.

```
Currency::~Currency()
{
}
```

- Later we will discuss how to free resources using a destructor.

Using the Currency Class

- We instantiate `Currency` class instances to represent different Currencies.
- We call them objects.
- We can create one `Currency` object for each currency:

```
#include ''Currency.h''

int main()
{
   Currency usd("USD", 1.0);

   Currency eur("EUR", 1.1);

   Currency gbp("GBP", 1.2);
}
```

Accessing Members

- We use *dot notation* to access a member function of a class.
- To get the rate of `eur`:
  `eur.GetExchangeRate();`

- To change the rate of `eur`:
  `eur.SetExchangeRate(1.2);`

- We can access `public` functions from any part of the program.
- We cannot access `private` data members from outside the class:
  `eur.exchange_rate_ = 1.2;` `error`

Classes - Exercises

Exercises

1. Write a class to represent a circle. Use it to find the area and circumference of a circle with radius of 7.
2. Write a class to represent a rectangle. Use it to find the area and circumference of a rectangle with sides of 5 and 4.
3. Write a `Student` class to represent a student.
4. Write an `Option` class to represent an option contract.

Recap: Classes

- A class defines a new type and a new scope.
- A class provides a natural way to treat data and functionality as a single entity:
  - Keeps related data together.
  - Defines operations to use on class data members in a *meaningful* fashion.
- We use classes as building blocks to write large, complex applications.

Encapsulation and Data Abstraction

- A class allows us to achieve *data abstraction* and *encapsulation*.
- Encapsulation refers to combining data and functions inside a class so that data is only accessed through the functions of the class.
- Data abstraction refers to the separation of interface (public functions of the class) and implementation:
  - The interface shows you how to use it.
  - A user of a class does not need to know how a member function in that class is implemented.

Example: OO Currency Converter

OO Currency Converter

- We're going to make several changes to the Currency Converter:
  1. Introduce a user defined type (class) – today
  2. Make design improvements – week 4 and week 6
  3. Introduce new concepts
- We will implement the initial OO version in two steps:
  1. Write a program to convert a value in USD to another currency (class discussion).
  2. Modify the program to convert a value from any currency to another (Assignment 2).

Currency Converter: Change #1

- Change #1: introduce a class to represent a currency.
- What attributes should we use to represent a currency?
  - symbol
  - exchange rate (relative to the US Dollar)
  - country?
- What functions should the `Currency` class support?
  1. a constructor/constructors
  2. get/set functions for data member
  3. `ConvertFromUSD()`: to convert a given amount in USD to that currency

Currency Class Definition

- Here's an attempt:

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

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

   double ConvertFromUSD(double value);

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

- We may add more members later.

Currency Class Implementation

- The constructor initializes data members.
- Get/Set members functions – easy.
- `ConvertFromUSD()` can be implemented as:

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

Using the Currency Class

- Now we can use a Currency object in the Currency Converter.
- Suppose, we want to convert an amount in USD to CAD:
  1. First, we instantiate a Currency object (e.g. CAD):
     ```
     Currency cad("CAD", 1.24);
     ```
  2. We can use it to convert a value from USD:
     ```
     double usdValue = 100;

     double cadValue = cad.ConvertFromUSD(usdValue);
     ```
- Now, we use Currency objects and member functions to do the conversions.

Assignment 2 (Graded)

- Write an OO version of the Currency Converter program using a `Currency` class.
- Other requirements same as Assignment 1.
  - 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.
  - Shutdown gracefully using user input.
  - You are not expected to test and validate the inputs.
- Individual Assignment.
- Due: Saturday, Jan 28 by midnight (Chicago time)

Struct and Class

struct and class

- A `struct` in C++ **is a class** with **one difference**.
- struct: Members have **public** protection level by default.
- class: Members have **private** protection level by default.

Include Guards

Include Guards

- In C++, a function, a class or a variable can be declared only once.
- This is known as the *One Definition Rule (ODR)*.
- We use header files to declare functions and classes.
- So, a program can read an include file only once.
- We use an *include guard* to make sure an include file is read only once.
- Visual C++ provides an easy mechanism (special preprocessor directive):

  ```
  #pragma once
  ```

- Visual Studio automatically puts this statement in every header file we create.
- However, this technique is non standard.

- The standard technique is to use preprocessor directives described next.
  - `define`: creates/defines a macro (simple identifier)
  - `ifndef`: checks if the macro is NOT defined
  - `endif`: ends the `ifndef`
- E.g. for `Currency` class (Currency.h) we can have:

```
#ifndef CURRENCY_H
#define CURRENCY_H

class Currency
{
    .....
    .....
};

#endif
```

- When a header file is read for the first time:
  1. Macro is not defined (`#ifndef` check passes).
  2. Macro is defined (`#define`)
  3. Header file is loaded/read
- Subsequent reads will be ignored since the macro is already defined (`#ifndef` check fails).
- For this technique to work, we need to create a unique macro in each header file.
- The name of the header file is usually used as the name of the unique macro.

Quiz

- What are the dfferences between a Struct and a Class in C++.
- What are include guards?

Special Member Functions

- We saw two special member functions:
  1. constructor
  2. destructor
- There are more:
  1. copy constructor
  2. assignment operator

Copy Constructing Objects

Copy Constructing Objects

- We can create a variable using an existing variable:

```
int i1 = 10;

int i2 = i1;
```

- User defined types (classes) may need to support similar operations:

```
Currency c1("CAD", 1.24);

Currency c2 = c1;

Currency c3(c1);
```

The Copy Constructor

- The copy constructor is used to construct a **new** object using an **already** constructed object of the **same** type:
- We define the copy constructor for the `Currency` class as:

```
Currency(const Currency&);
```

- We use the already constructed object's data members to initialize the new object's data members:

```
Currency::Currency(const Currency& other)
    : symbol_(other.symbol_),
      exchange_rate_(other.exchange_rate_)
{ }
```

Assignment Operator

The this Keyword

- Every non static[1] class member function has access to an implicit pointer; name of the pointer is *this*.
- The *this* pointer is initialized with the object's own address.

[1] All member functions we've seen so far are non static; we will discuss static members later

The Assignment Operator

- We can assign new values to existing variables:

```
int i1 = 10;
int i2 = 20;

i1 = i2;
```

- User defined types (classes) may need to support similar operations:

```
Currency c1("CAD", 1.24);

Currency c2;

c2 = c1;
```

- We use the assignment operator to assign an object to another object of the same type:

```
c2.operator=(c1);

same as,

c2 = c1;
```

- To define the assignment operator for a class, we *overload* the following operator:

  `operator=`

- It takes an already constructed object as an argument. We pass it by const reference:

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

- What should it return?

The Assignment Operator: Return Type

- Suppose it returns void.
- We can write an assignment operator (incorrectly) as:

```
void Currency::operator=(const Currency& other)
{
    symbol_             =  other.symbol_;
    exchange_rate_    =  other.exchange_rate_;
}
```

- This works for:

```
c2 = c1;
```

- It does not work for chained assignments:

```
c3 = c2 = c1;
```

- Remember: above statement is same as:

```
c3 = c2.operator=(c1);
```

- Why doesn't this work?

- For chainged assignments to work, the assignment operator should return a reference to itself (e.g. `Currency&`).
- How do we return itself from a member function?
- We use the `this` pointer.

```
Currency& Currency::operator=(const Currency& other)
{
    symbol_            =  other.symbol_;
    exchange_rate_   =  other.exchange_rate_;

    return *this;
}
```

The Assignment Operator: Self Assignment

- What happens if we assign an object to itself?
  ```
  c1 = c1;
  ```
- In this case there's no harm. But this is inefficient. Why?
- In some cases self assignment is dangerous (more on this later).
- We need to detect self assignment.

- If the two objects are the same, they should have the same memory address.

```
Currency& Currency::operator=(const Currency& other)
{
    if (this != &other)
    {
        symbol_             =  other.symbol_;
        exchange_rate_   =  other.exchange_rate_;
    }

    return *this;
}
```

Operator Overloading

- Operator overloading allows us to define operator symbols for classes. E.g. assignment operator.
- Overloading make it easier and more intuitive to use classes.

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

Matrix a, b, c;
...
...

c = a + b;
cout << c;
```

- Operator symbols we might want to overload depends on the class. E.g. Overloading $+$, - operators are meaningful for Matrix class but not meaningful for Student class.

- Most operators (symbols) can be overloaded, but some restrictions apply:
  - cannot overload operators that has the potential to change the meaning of the language
  - cannot add new operator symbols
  - `https://isocpp.org/wiki/faq/operator-overloading#overloadable-opers`
- Good rule of thumb is to pay attention to how the operators are overloaded for built in types:
  - users may expect similar behavior
  - e.g. chained assignments for operator=

Static Members

Non-Static Members

- Class members (data and function) we've seen so far are associated with an individual object of the class.
- E.g.: If we instantiate multiple Currency objects, each object has its own symbol and the exchange rate:

```
Currency c1("CAD", 1.3536);
Currency c2("EUR", 0.9494);
```

- A member function uses/changes data associated with that object:

```
c2.SetExchangeRate(1.1111);
```

- In the example above, we change the rate of `c2`; this operation has no effect on `c1`.
- This is nothing new, and this is how we use usually use classes.

Static Members

- We can also associate a member/members (data/function) with the class, not with individual objects.
- We use `static` keyword to associate a member with the class.
- Suppose, we want to write a *Counter* class to keep track of some value.
- Also, suppose, we want multiple counters to keep track of the same value.
  - we're trying to raise money for a charity
  - each one of us (one object) raises money
  - every dollar goes to one bank account
- We could use static data member (bank account) here.

Static Members: Example

- We can write the `Counter` class as (uses incorrect syntax):

```
class Counter
{
public:
   int GetCount();

   void Increment();

private:
   static int count_;
};
```

- The `count_` member belongs to the class, i.e. all objects of that type.

- `Counter` class implementation:

```
int Counter::GetCount()
{
    return count_;
}

void Counter::Increment()
{
   count_ ++;
}
```

- Suppose, now we create two counter objects:

```
Counter c1;
Counter c2;
```

- And, we increment one counter `c1`:

```
c1.Increment();
```

- This will increment the `count_` value seen by all objects, `c1` and `c2` in this case.

- A static data member cannot be accessed directly using a non-static member function.
- Two member functions have to be `static`.

```
class Counter
{
public:
   static int GetCount();

   static void Increment();

private:
   static int count_;
};
```

- Static member variables cannot be initialized through the class constructor:
  - this makes sense – if a data member does not belong to an object, we should not be able initialize it in the constructor
  - otherwise, different objects would initialize the static member to different value and the compiler won't know which one to use
- Static data members should be defined and initialized once outside the class body.
- Const static member variables can be initialized within the class body.

- A static data member must be initialized once before we use it (outside the class):

```
int Counter::count_ = 0;
```

- The rest of the `Counter` class implementation is the same:

```
int Counter::GetCount()
{
    return count_;
}

void Counter::Increment()
{
   count_ ++;
}
```

- We can create an object and access a static member (as before):

```
Counter c;
c.Increment();
```

- A static member (data/function) does not belog to an object.
- We do not need an object of a class to use a static member.
- We can also access the `static` members using the scope (::) operator without an object.

```
Counter::Increment();
```

- We can use the `Counter` with our without an object:

```
int main()
{
    Counter counter1;
    Counter counter2;

    counter1.Increment();
    cout << counter1.GetCount()  << endl;
    cout << counter2.GetCount()  << endl;

    Counter::Increment();
    cout << counter1.GetCount()  << endl;
    cout << counter2.GetCount()  << endl;
}
```

- This example illustrates the idea of a static data member.
- We could just have static member function/functions even if it doesn't use static data members.
- Usage is still the same.