CMPT 225

Lecture 2 – Abstract Data Type (ADT)

Learning Outcomes

- At the end of this lecture, a student will be able to:
  - Define abstraction, information hiding and "abstract data type“ (ADT)
  - Write C++ code
  - Encapsulate methods and variables for an ADT into a C++ class
  - Differentiate between a class that has been designed as an ADT and a class that has not
  - Compare and contrast them

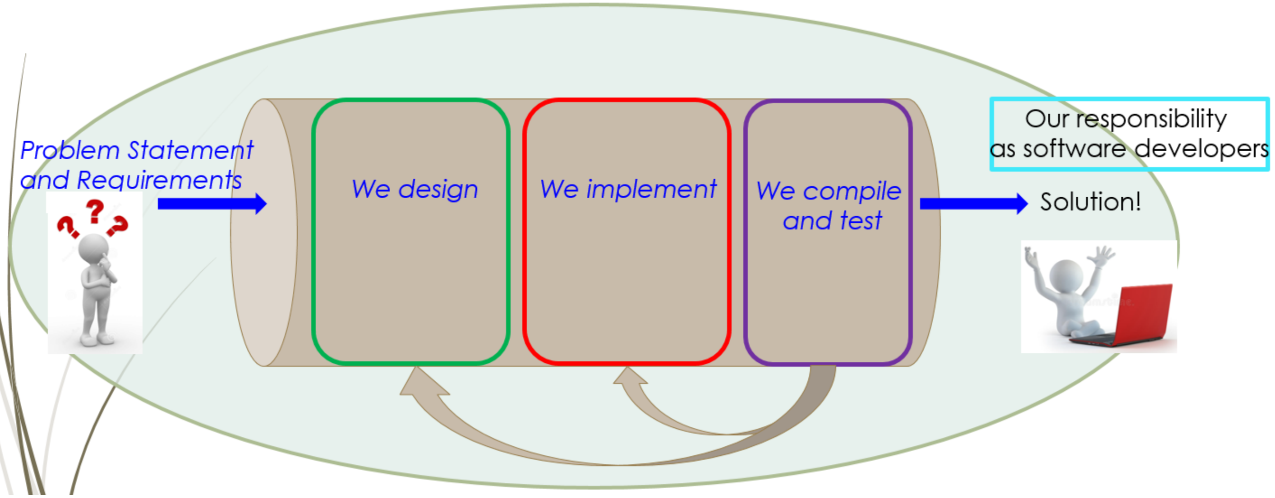
Last Lecture

- ✓ What are we learning in CMPT 225?
  - ✓ And what must we already know?
- ✓ Which *resources* do we have to help us learn all this?
- ✓ Activity - Thank you!
  - ✓ We know: Getter method, array, data members, class description, constructor parameters
  - ✓ We may not yet know: stack and heap, class invariant, precondition, postcondition
- ✓ Questions?
- ✓ What are we doing *next lecture* and how to get ready for it!

Today’s menu

- Introducing the concept of Abstract Data Type (ADT)
  - Definition + “Wall” metaphor
  - How to design an ADT
  - How to implement an ADT in C++
  - How to test an ADT
- Example: Temperature class
  - Implemented as an ADT
  - Implemented as a non-ADT
- Compare both implementations

Let's start with a problem



- Step 1 – Problem Statement and Requirements
  - Create a temperature conversion application

  => *What do we do in this step?*
- Step 2 – Design
- Step 3 – Implementation
- Step 4 – Compile and Test

Why is Step 1 so important?

- The quote below illustrates one of the reasons why we make sure the problem statement and the requirements we received from the client are clear to us (i.e., detailed and understood) such that there is no room for misinterpretation:

> **Feet or miles?**[a]
>
> During a laser experiment, a laser beam was directed at a mirror on the Space Shuttle Discovery. The test called for the laser beam to be reflected back toward a mountain top. The user entered the elevation of the mountain as "10,023," assuming the units of the input were in feet. The computer interpreted the number in miles and the laser beam was reflected away from Earth, toward a hypothetical mountain 10,023 miles high.

Source: page 122, Object-Oriented Software Engineering – Using UML, Patterns, and Java, 3rd edition, by Bernd Bruegge and Allen H. Dutoit

Step 2 – Design
=> *What do we do in this step?*

Step 2 – Design Data

UML Class Diagram

Class Name
Attributes
Operations

Exclude constructors, destructors, getters/setters.
Why?
Because these methods are quite standard. So we can assume that, if they are needed, they will be designed and implemented.

Temperature
degrees scale
inFahrenheit( ) inCelsius( ) raise(toWhat)

Application-related methods

Step 2 – Design Solution

- Describe the behaviour of our temperature conversion application by listing the steps it will perform when it will execute:

```
Display menu
1. Convert Celsius temperature to Fahrenheit
2. Convert Fahrenheit temperature to Celsius
Read user choice
Ask user for and read temperature degrees
Create a temperature object of required scale
Convert this temperature object to
   temperature object of other scale
Display resulted conversion
Repeat the above until user quits
```

This is an algorithm expressed in **pseudocode**

Step 3 – Implement solution

- Let's have a look at the code posted on our course web site!

Header file: Temperature.h

Step 3 – Implement solution

Preprocessor directives

```
/* Header Comment Block */

#ifndef Temperature_H
#define Temperature_H

/* Class Definition */
class Temperature {


private:
   double myDegrees = 0.0; // >= ABSOLUTE_ZERO for myScale
   char   myScale ='C';    // 'F' or 'C'
   bool   isValidTemperature( const double degrees,
                              const char scale );
public: … (expanded in the next slide)
};
```

Step 3 – Implement solution – cont'd

Constants

```
public:

    constexpr static double ABSOLUTE_ZERO_FAHRENHEIT = -459.67;
    constexpr static double ABSOLUTE_ZERO_CELSIUS = -273.15;

    // Constructors
    Temperature();
    Temperature(double degrees, char scale );
    // Getters
    double getDegrees( ) const;
    char getScale( ) const;
    // Setters
    void raise( const double amount );
    // Application-related methods
    Temperature inFahrenheit( ) const;
    Temperature inCelsius( ) const;
};
#endif
```

Implementation file: Temperature.cpp

Step 3 – Implement solution

Preprocessor and *using* directives

```
/* Header Comment Block */

#include <iostream>
#include <cctype>
#include "Temperature.h"
...
Temperature Temperature::inFahrenheit( ) const {
   Temperature result;
   if ( myScale == 'F' )
       result = Temperature( myDegrees, 'F' );
   else if ( myScale == 'C' )
       result = Temperature(myDegrees * 1.8 + 32.0, 'F');
   return result;
} ...
```

Step 3 – Implement solution

/* Header Comment Block */

```
/*
 * Temperature.h
 *
 * Class Description: Class modeling a valid temperature and offering converters.
 *
 * Class Invariant: myScale == 'C' && myDegrees >= ABSOLUTE_ZERO_CELSIUS ||
 *                  myScale == 'F' && myDegrees >= ABSOLUTE_ZERO_FAHRENHEIT
 *
 * Author: AL
 * Modified on: Sept. 2
 */
```

A **class invariant** is " … *used for constraining objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object.*" Thank you Wiki!

A **class invariant** is something about a class that must always be true, for all objects of the class.

TemperatureConverter.cpp

Step 3 – Implement solution

```
/* Header Comment Block */


int main() {
   // Display menu
   // 1. Convert Celsius temperature to Fahrenheit
   // 2. Convert Fahrenheit temperature to Celsius
   // Read user choice
   // Ask user for and read temperature degrees
   // Create a temperature object of required scale
   // Convert this temperature object to
   //   temperature object of other scale
   // Display resulted conversion
   // Repeat the above until user quits
   return 0;
}
```

Preprocessor and *using* directives

Trick:

1. Copy your pseudocode into a method
2. Transform your pseudocode into comment
3. Translate your pseudocode into C++
4. Keep your pseudocode and your code will already be commented! ☺

TemperatureTestDriver.cpp

Step 4 – Testing Temperature class

```
/* Header Comment Block */

int main() {
    // Create a valid Celsius temperature
    // Create an invalid Celsius temperature
    // Create a valid Fahrenheit temperature
    // Create an invalid Fahrenheit temperature
    // Converting a valid Celsius temperature
    //  to a Fahrenheit temperature
    // Raising a valid Celsius temperature to
    //  an invalid amount of degrees
    ...
    return 0;
}
```

Preprocessor and *using* directives

Goals of Test Driver:
1. To test each method of Temperature class by calling it at least once.
2. To "break" the code, i.e., calling each method with valid and invalid parameters.

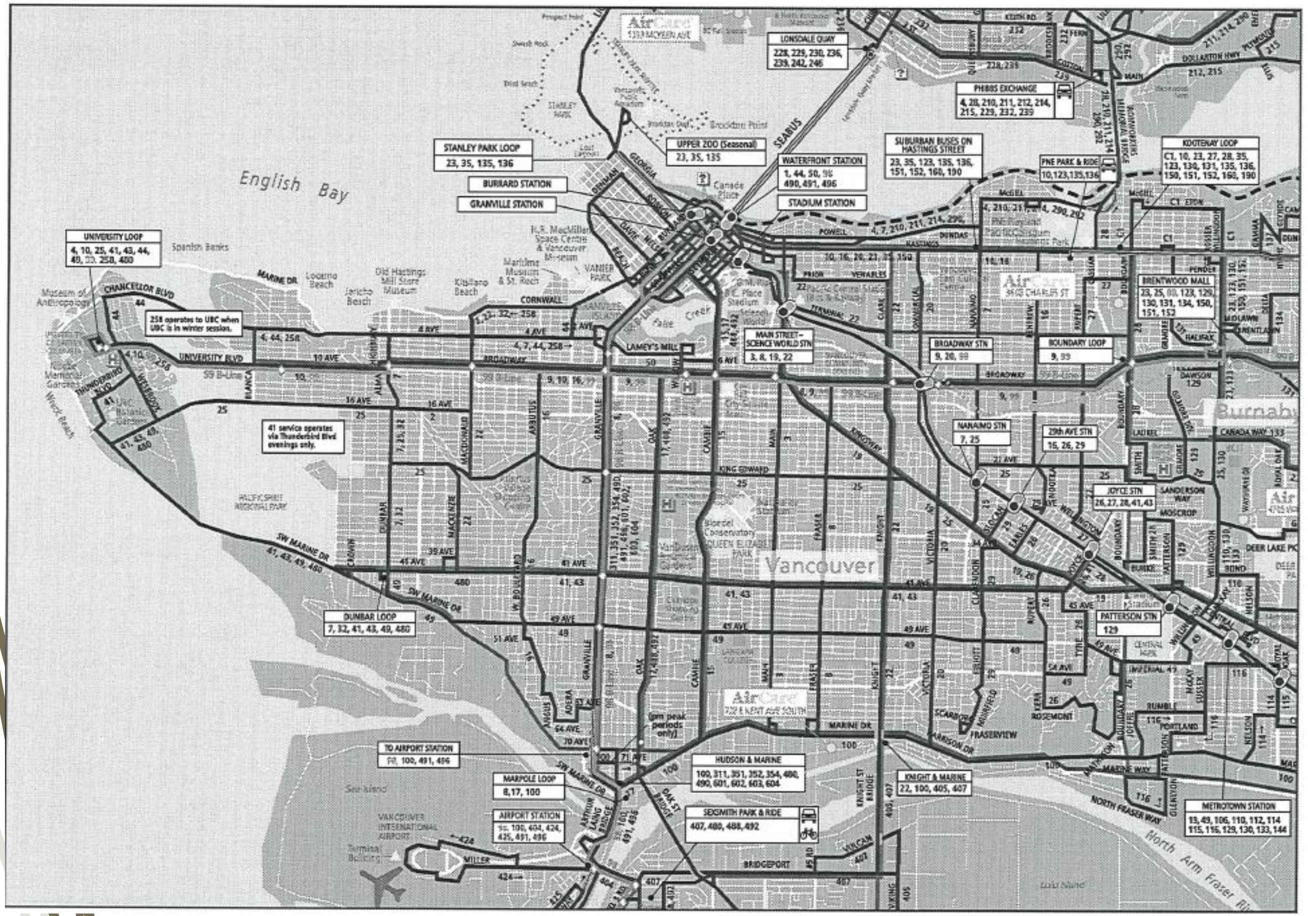
What makes this class an ADT?

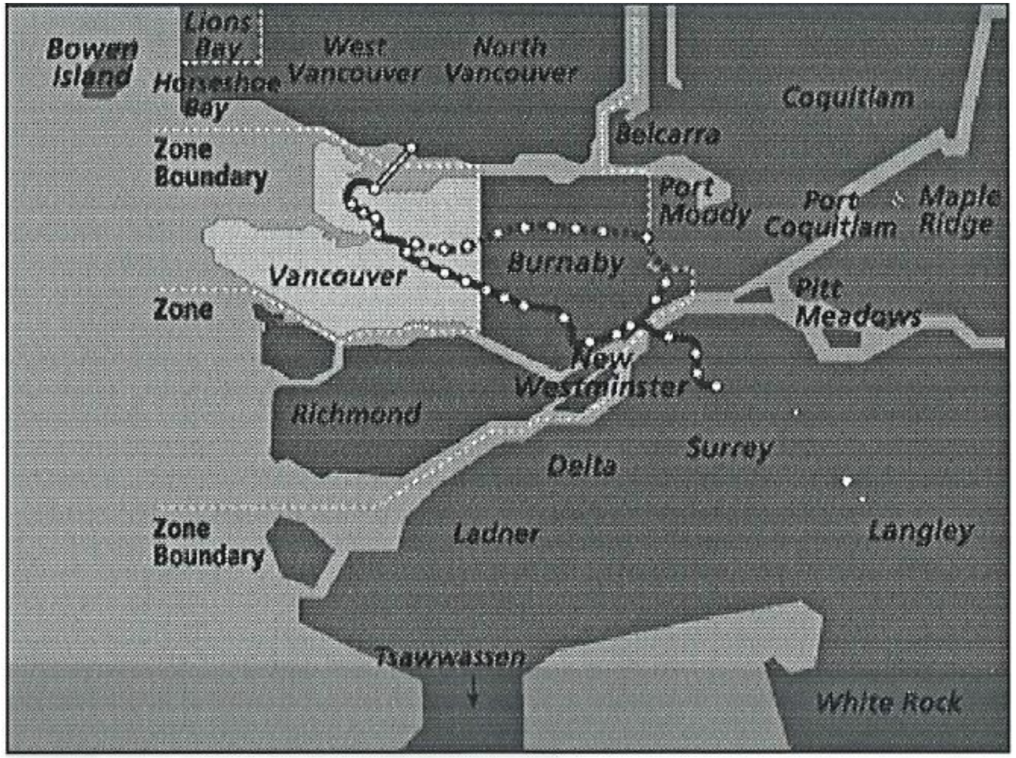
- So, what is an ADT (abstract data type)?

Abstraction – in the *real world*

- Abstraction
  - From the Latin *abs*, meaning *away from*
  - and *trahere*, meaning *to draw*
- "*Process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics*" Source: https://whatis.techtarget.com/definition/abstraction
- Examples:
  - Car
  - Map

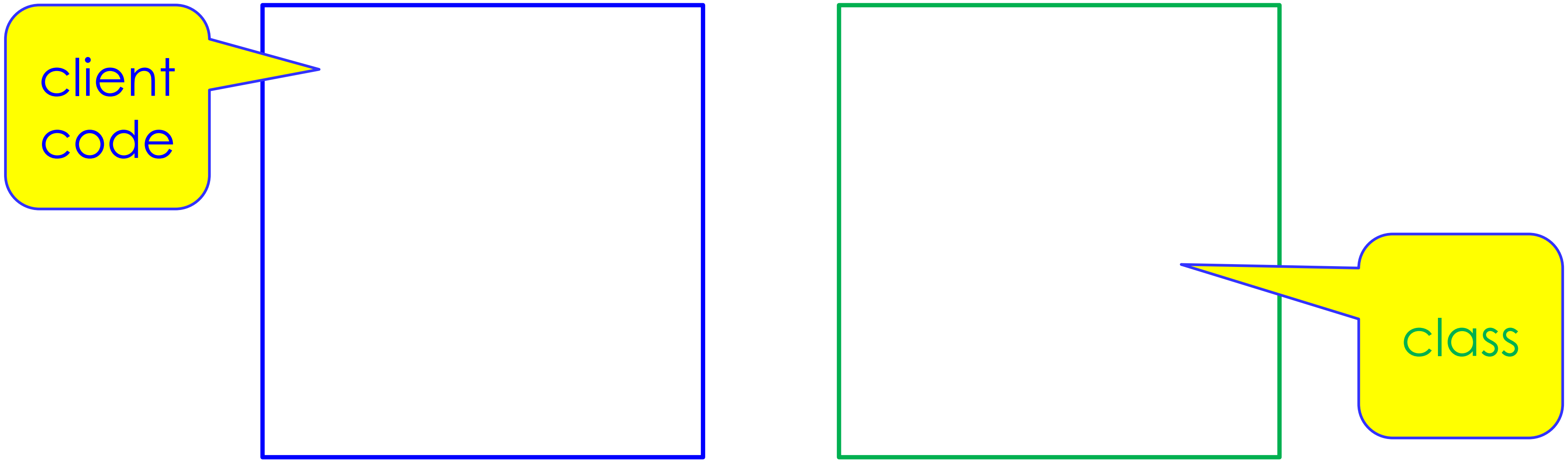
Example - Map


English Bay
Vancouver
STANLEY PARK LOOP
BURRARD STATION
GRANVILLE STATION
WATERFRONT STATION
STADIUM STATION
LONSDALE QUAY
PHIBBS EXCHANGE
KOOTENAY LOOP
UNIVERSITY LOOP
BROADWAY STN
BOUNDARY LOOP
NANAIMO STN
29th AVE STN
JOYCE STN
PATTERSON STN
METROTOWN STATION
DUNBAR LOOP
MARPOLE LOOP
AIRPORT STATION
HUDSON & MARINE
KNIGHT & MARINE
SEXSMITH PARK & RIDE


Bowen Island
Lions Bay
Horseshoe Bay
West Vancouver
North Vancouver
Coquitlam
Belcarra
Zone Boundary
Port Moody
Port Coquitlam
Maple Ridge
Vancouver
Burnaby
Zone
Pitt Meadows
New Westminster
Richmond
Surrey
Delta
Ladner
Langley
Zone Boundary
Tsawwassen
White Rock

Abstraction and **information hiding** – in the *software world*

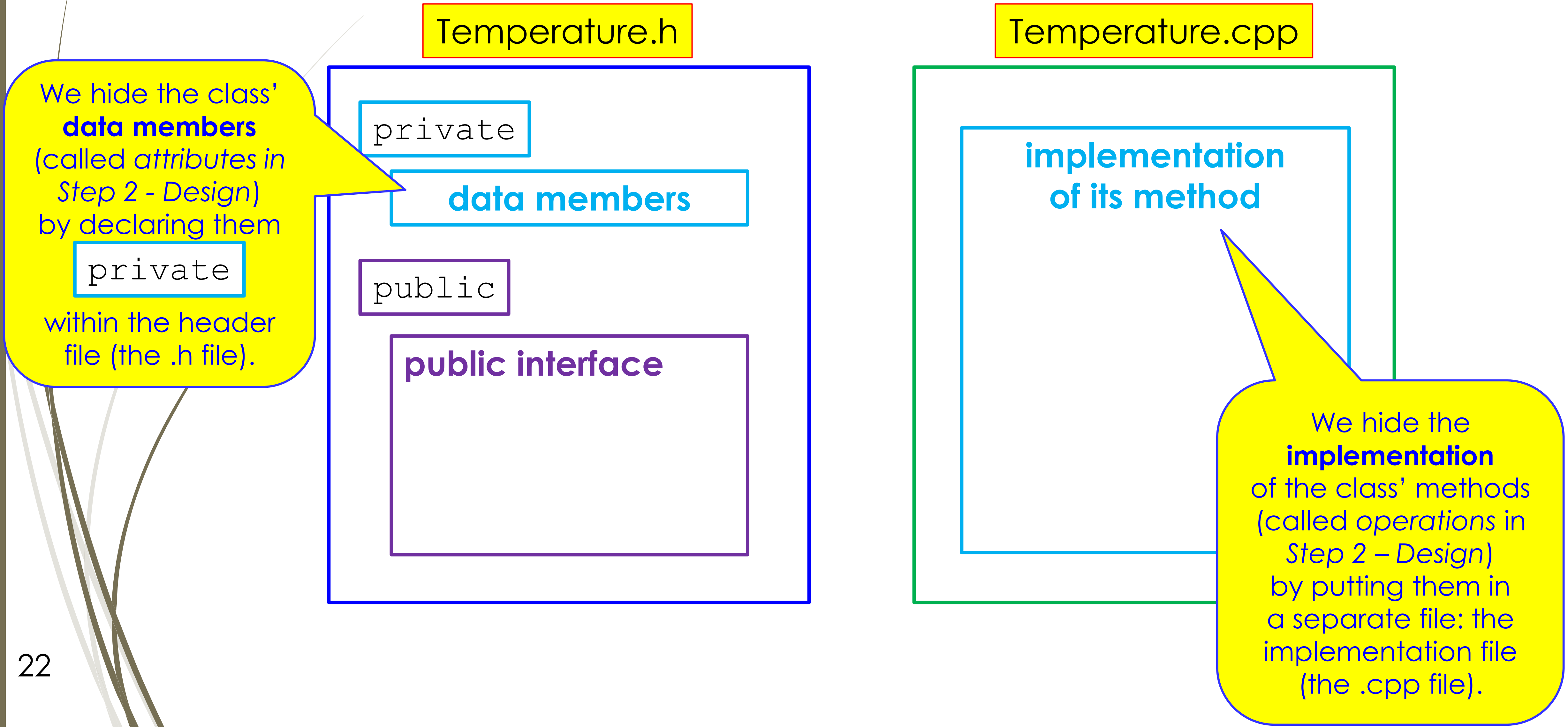
- We achieve **abstraction** by **hiding information** from ***public view***, *i.e.*, from other classes acting as **client code** such as application (e.g., converter) and test drivers



Abstraction and **information hiding** – in the *software world*

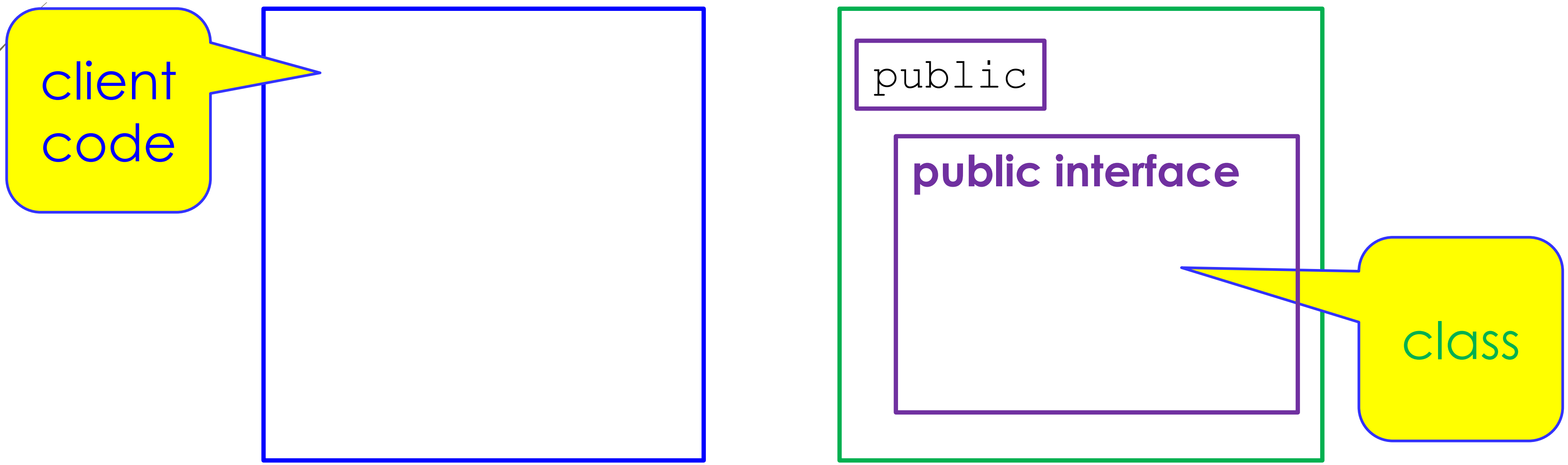
- We separate the **purpose** of a class (the ***what***), which is defined by its **public interface** from its **implementation** (the ***how***) by hiding the latter, which includes the class' **data members** and the **implementation** of its methods.

Abstraction and **information hiding** in **C++**



Abstraction and **information hiding** – in the *software world*

- This way, client code can use the class without knowing its **implementation** (**.cpp** file), it only needs to know the class' **public interface** (**.h** file).

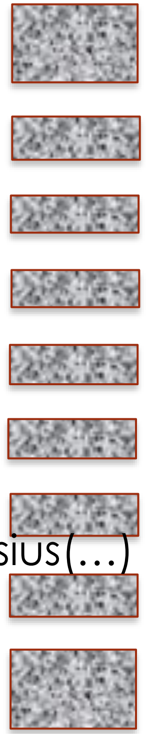


- This **reduces complexity** and allows for **easy modification**.
- And this is how we construct an **ADT**.

Wall metaphor for an **ADT**

Client Code

Here is the wall:



Temperature Converter.cpp accesses Temperature's **private** members by calling Temperature class' **public** methods

Temperature class' **private** members hide behind the wall

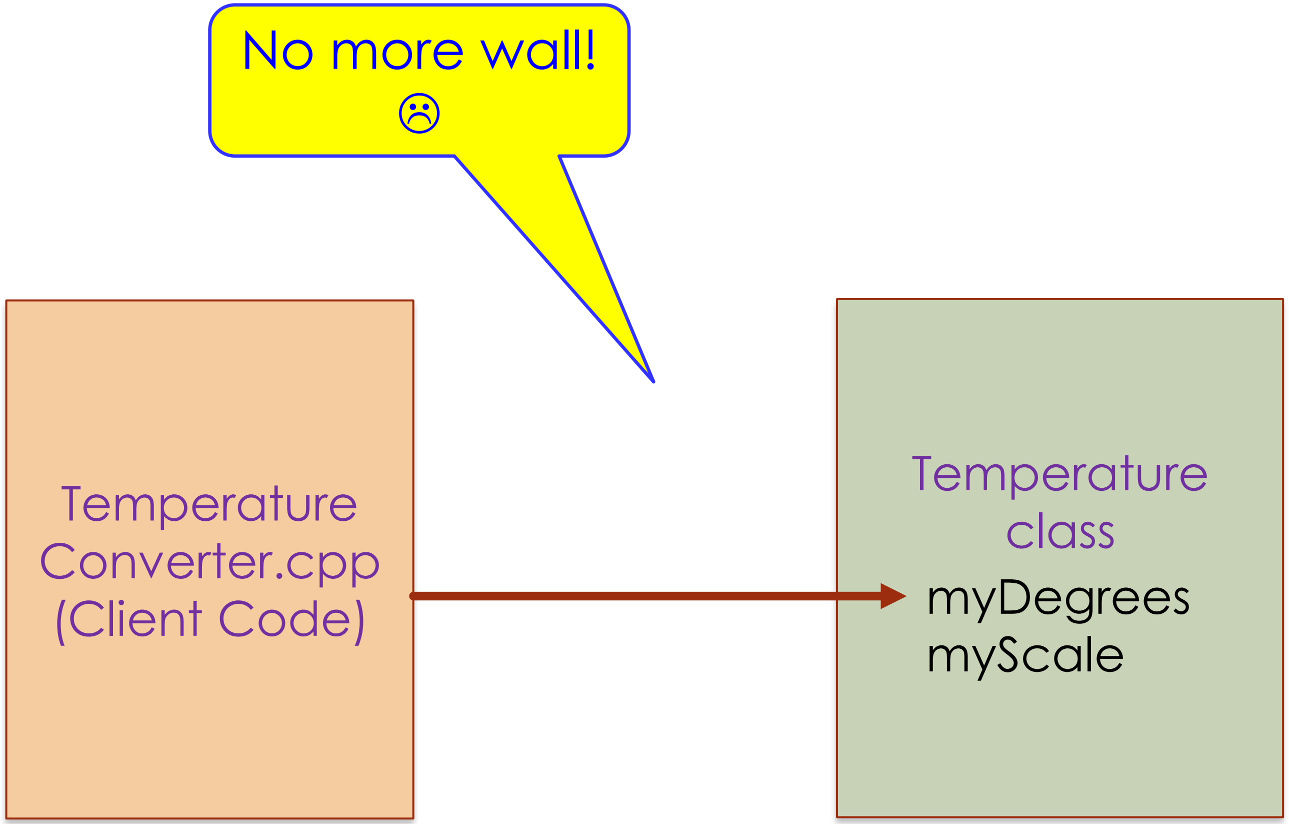
When Temperature class is implemented as an **ADT**

And **TemperatureConverter.cpp** knows how to call them because it has access to Temperature's public section, i.e., its **public interface**

Slits in the wall represent Temperature class' **public** methods like **inCelsius(…)** and **inFahrenheit(…)**

When Temperature class not implemented as an ADT

- So, **Client code** can tamper with the class' **private** members (data), hence it can break the **class invariant**



Step 4 – Compilation and Testing

- In our Temperature class example, client code can break Temperature class' invariant as follows:
- `Create a valid Fahrenheit temperature ->`
  `testing Temperature( 32.0, 'F' )`
  `Actual Result: 'tempFahr' -> 32 degree F`
- `Changing the amount of degrees of 'tempFahr' to -976.02F by setting 'myDegrees' to -976.02 directly.`
  `Actual Result: 'tempFahr' -> -976.02 degree F`

Advantages and disadvantages of ADT

- Advantages:
  - Preserve/control the integrity of a class' data (expressed as invariant of a class)
  -
- Disadvantages:
  - More code to write: must have getters/setters methods
  -

√ Learning Check

- We can now …
  - describe what happens in the 4 steps of the software development process:
    - Step 1 - Problem statement
    - Step 2 – Design
    - Step 3 – Implementation
    - Step 4 – Compilation and Testing
  - construct an ADT class in C++
  - define abstract data type (ADT), abstraction and information hiding
  - differentiate between a class that has been designed/implemented as an ADT and a class that has not
  - list some of the advantages and disadvantages of ADT

Next Lecture

- Introduce our first data collection: List
- Design a List class as an ADT
- Implement it using an array