A path to link the concepts around objects

This document is a response to the question below.

Question:
Can you perhaps provide a simple and easy to follow "route map" that links the creation of classes using private and public definitions of member variables and member functions through to the creation of constructors, default constructors, implementation and interface steps.

Answer:

I first provide some background info that you need to keep in mind when working with classes before the route map appears. You first have to understand the concept of object-oriented programming. Then some explanations are given, followed by the (high-level) route map for creating and using a class, an example of an ADT (class Temperature) and its application; and finally the path COS1512 follows through the prescribed book to put everything we cover into perspective.

Object-Oriented Programming

In object–oriented programming applications are organised around *objects* rather than processes. In the object-oriented paradigm, a system is seen as a collection of interacting objects that models the interaction between objects necessary to achieve the desired effect. In object-oriented programming, program components are envisioned as objects that belong to classes and are similar to real-world objects; the programmer then manipulate the objects and have them interrelate to achieve a desired result. With object-oriented analysis, design and programming, the focus is on determining the objects you want to manipulate rather than the processes or logic required to manipulate the data. Object-oriented programming involves three fundamental concepts: *encapsulation*, *inheritance* and *polymorphism*.

Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications, i.e. stand-alone executable programs, that use those objects. If they are designed properly, classes can be reused over and over again to develop new programs.

An object generally represents a concrete entity in the real world such as a person, place or thing. Such an object has certain characteristics or *attributes* and certain *behaviours* or actions. An object's characteristics represent data or facts that we know about the object. This is sometimes called the state of the object and represented by the *member variables*. The object's behaviours are actions that the object can take. These actions are expressed in *member functions* or methods that execute algorithms. Each object plays a specific role in the program design. The object interacts with other objects through messages or function calls. Attributes and behaviours, i.e. an object's data and algorithms, can be *encapsulated*, or combined together to form a class.

A class is like a template or blueprint that can be used to construct or *instantiate* many similar objects. A class can also be seen as a sort of user-defined data type, which we call an abstract data type (ADT). An object is an instance of a particular class. A class hides information by restricting access to it. In effect the data and methods within an object are hidden from users to prevent inadvertent changes. Users of the class simply need to understand how to use the interface or interaction between the object and its methods.

Through *inheritance* it is possible to define subclasses that share some or all of the parent class's attributes and methods, but with more specific features. This enables reuse of code.

Polymorphism describes the feature in languages that allows us to interpret the same word differently in different situations based on the context in which it is used. In object-oriented programs this means that similar objects can respond to the same function call or message in different ways.

Explanations for creating and using a class

A class is like a template or blueprint that can be used to construct or *instantiate* many similar objects. A class can also be seen as a sort of user-defined data type, which we call an abstract data type (ADT). An ADT is a user-defined data type, in comparison to a predefined data type such as `int`. A class is used to *instantiate* an object (similar to declaring a variable of a predefined type). An object is an instance of a particular class. You can think of the class as a cookie cutter and of the object as a cookie that you can eat or use for some purpose. A class hides information by restricting access to it. In effect the data (the member variables in the class) and methods (the member functions) within an object are hidden from users to prevent inadvertent changes. Users of the class simply need to understand how to use the interface or interaction between the object and its methods.

To create a class we need two files, a class declaration (interface) file (the `.h` file or header file) and a class implementation file (the `.cpp` file). The prefix for the .h and the.cpp files must be the same, because this is the way the compiler links the declaration and the implementation. The terms 'class definition', 'class specification', 'class declaration' and 'interface' all refer to the same thing and are used interchangeably in the literature.

The **class declaration** (the `.h` file) or **interface** contains the template or blueprint (cookie cutter) that can be used to construct or *instantiate* many similar objects. This is the part where we specify the member variables and member functions (typically only the headers of the member functions) for the class. This is the *only* part other programmers who want to use the class need to see – they only need to know what member variables are available in the class and how the headers for the member functions look.

We make the ***member variables*** **private** to restrict access to them to protect them from inadvertent changes by users. Therefore users of the class can only access the values of the member variables by means of accessor functions (also called *get* functions because they are used to get the values of the member variables); and only change the values of the member variables with mutator functions (sometimes called *set* functions because they are used to set the member variables to new values).

Member functions which may be used in application programs are declared **public**. Accessor and mutator functions are usually public functions. A class may also have private member functions, which may then only be called by other member functions of the class.

A **constructor** is a member function that is called automatically when an object of the class is declared. A constructor is used to initialise the values of the data members and to do any sort of initialisation that may be required. The constructor has the *same name* as the class and *does not return a value*. The ***default constructor*** has no parameters. Every class should have a default constructor. Classes often also have *overloaded constructors*, which can be used to specify specific initial values for the member variables when the object is instantiated. The overloaded constructor will be called when initialisation values are specified when the object is instantiated. Both the default and overloaded constructor are also public, but are *never* called explicitly in application files.

A *destructor* is a member function that is called automatically when the scope where the object has been instantiated is exited, e.g. when the end of the program is reached. The destructor for a class also has the same name as the class, but with a tilde (~) character in front of the class name. The destructor is also never called explicitly.

Pre-defined operators such as `+, -, * , /, >, >=, <, <=, ==, +=, --, -+, >>` and << may be overloaded for the class. This means that we would like to use these operators as normal, e.g. use the overloaded stream extraction operator >> to read in values for all the member variables for an object of the class in one statement. We provide the function header for the overloaded >> in the class declaration or interface and the code itself in the implementation file.

Friend functions are functions that are not member functions, but have access to the member variables of the class. Because pre-defined operators have already been defined, they cannot be member functions of a used-defined class. Therefore we declare them as friend functions of a class when we overload them for a class. This will allow the overloaded friend operators to access the member variables of the class directly (i.e. without using accessor or mutator functions).

The **class implementation** (the `.cpp` file with the same prefix as the `.h` file) contains the actual code for the member functions. Other programmers who use a class you created does not need this file – in fact, you should be able to change the code in this file without users of the class realising that the code has changed (this is how we hide the information about the class). In the class implementation, the code that should be executed when a member function is called, is provided for each member function. This is where the actual code (i.e. the body of the functions) for the default and overloaded constructors, the destructor, accessors, mutators and other member functions are provided. The code for friend functions are also provided here.

In an **application program** (typically the `.main` file in a project) the class is used to instantiate (or 'declare') objects of the class. The class is just a template and no memory is reserved for a class. Only when an object is instantiated is memory reserved for the member variables. When an object is instantiated, the constructor for the class is called automatically. If no initialisation values are specified, the default constructor is used. If initialisation values are specified, the overloaded constructor is called automatically.

The application program's only access to the values of member variables of an object is through accessor functions (get functions) to retrieve their values and mutator functions (set functions) to change their values. Other member functions may be used to manipulate the member variables to achieve what the program has to do.

When the program ends, the destructor is called automatically, and the memory used by object is released.

The `#ifndef, #define` and `#endif` macros are used to prevent including code for the interface more than once. The `#include` macro are used to include the class definition in the interface (.h or header file) in the implementation and the application files. The `#include` macro pulls in all the code in the class definition/specification

Route map steps

1. Create a project

2. Create the class definition/specification or interface (.h file or header file). This file does not need to be part of the project.

   2.1 Use the `#ifndef, #define` (at the start of the file) and `#endif` (at the end of the file) macros to prevent including code for the interface more than once in the implementation and application files.

   2.2 Member variables are private and represent the 'state' of the object.

   2.3 Member functions are public and represent the actions the object can execute.

   2.4 Include the following member functions:

   - a default and/or overloaded constructor
   - a destructor
   - accessors (get functions)
   - mutators (set functions)
   - other member functions as required by the object
   - friend functions (if necessary)
   - overloaded operators (if necessary)

   2.5 Remember the ; after the class declaration but before the `#endif` in the .h file!

3. Create the implementation file (.cpp file with the same name as the .h file). This file must be part of the project.

   3.1 Use the `#include` macro to include the class definition (.h file).

   3.2 Include the statement:

   ```
   using namespace std;
   ```

   3.3 Now you provide the code that should be executed for each of the member functions when the function is called, i.e the bodies of all the functions whose headers appear in the class declaration.

3.4 Scope resolution (name of the class followed by `::` operator) should precede each member function body. Note that the return type for the function appears before the scope resolution, e.g.

```
double Temperature::getFdegrees() const
```

3.5 Note that no scope resolution is required for the implementation of friend functions since they are not member functions of the class.

3.6 Use the const keyword when an individual member variable should not be changed by a member function, or when a function should not change the member variables, e.g.

```
double Temperature::getFdegrees() const
```

4. Create the application file (the `.main` file in the project)

4.1 Use the `#include` macro to include the class definition (`.h` file). Note that the `.cpp` file for the class must be part of the project.

4.2 Instantiate the objects required by the application by 'declaring' them. This will call the default or overloaded constructor automatically.

4.3 Manipulate the object(s) by calling member functions to execute actions. In a call to a member function, the object's name appears before the dot operator, followed by the member function name with appropriate parameters. Note the return type of the member function – that determines how the result from executing the function should be handled.

4.4 When the application program ends, the destructor will be called automatically.

An example of an ADT (class Temperature) and its application

Below the header file (**`Temperature.h`**) and implementation file (**`Temperature.cpp`**) for a class **`Temperature`** implemented as an ADT, as well as the application file (**`main.cpp`**) and the output is shown.

Class `Temperature` has two member variables, one to represent a temperature in degrees Celsius (`cdegrees`) and the other to represent the corresponding temperature in degrees Fahrenheit (`fdegrees`). Note what the output is after we have changed only the degrees Fahrenheit in the main program – the degrees in Celsius no longer corresponds.

Colours are used to link the declaration, implementation, application and corresponding output for the following concepts:

- Constructors in yellow
- Destructor in light grey (Note the output)
- Accessors in green (Note the main application program and the effect in the output)
- Mutators in turquoise (Note the main application program and the effect in the output)

- Two ordinary member functions in purple and red (also mutators as they change the values of the member variables)
- An overloaded member function implemented as a friend function for the stream insertion operator << in dark grey (Note the main application program and the effect in the output)

Temperature.h

```
//header file with class declaration
#ifndef TEMPERATURE_H
#define TEMPERATURE_H
#include <iostream>

using namespace std;

class Temperature{
public:
    Temperature();                          //default constructor
    Temperature(double F, double C);        //overloaded constructor
    ~Temperature();                         //destructor
    double getFdegrees() const;             //accessor return Fahrenheit degrees
    double getCdegrees() const;             //accessor return Celsius degrees
    void setFdegrees(double F);            //mutator to change Fahrenheit degrees
    void setCdegrees(double C);             //mutator to change Celsius degrees
    void convertTtoF (double C);            //convert Celsius to Fahrenheit
    void convertTtoC (double F);            //convert Fahrenheit to Celsius
    friend ostream& operator << (ostream& outs, const Temperature& T);
      //overloaded operator << as a friend function
private:
    double fdegrees;                        //holds Fahrenheit degrees
    double cdegrees;                        //holds Celsius degrees
};

#endif
```

Temperature.cpp

```
//Implementation file for class Temperature
#include "Temperature.h"

using namespace std;

Temperature::Temperature() : fdegrees(0), cdegrees(0)
//default constructor
{

}

Temperature::Temperature(double F, double C) : fdegrees(F), cdegrees(C)
//overloaded ocnstructor
{

}

Temperature::~Temperature()                   //destructor
{
      cout << "\nGoodbye!\n";
}
```

```
double Temperature::getFdegrees() const     //accesor to return Fahrenheit
degrees
{
    return fdegrees;
}


double Temperature::getCdegrees() const     //accesor to return Celsius
degrees
{
    return cdegrees;
}

void Temperature::setFdegrees(double F)       //mutator to change
Fahrenheit degrees
{
    fdegrees = F;
}

void Temperature::setCdegrees(double C)       //mutator to change Celsius
degrees
{
    cdegrees = C;
}

void Temperature::convertTtoF (double C)    //convert Celsius to Fahrenheit
{
    cdegrees = C;
    fdegrees = 9.0 / 5.0 * C + 32;
}

void Temperature::convertTtoC (double F)    //convert Fahrenheit to Celsius
{
    fdegrees = F;
    cdegrees = (F - 32) * 5.0 / 9.0;
}

//overloading stream insertion operator <<
//note no scope resolution necessary
ostream& operator << (ostream& outs,const Temperature& t)
{
    outs << t.cdegrees << " degrees Celsius is equal to "
         << t.fdegrees << " Fahrenheit \n";
    return outs;
}
```

Main.cpp

```
#include <iostream>
#include "Temperature.h"

using namespace std;

int main()
{
    double fahrenheit, celsius;
    Temperature temp (32,0); //overloaded constructor called automatically
    cout << "The current temperature is " << temp.getFdegrees()
         << " degrees Fahrenheit, which is "
         << temp.getCdegrees() << " degrees Celsius";//use accessors
    cout << "\nPlease enter degrees in Fahrenheit\n";
    cin >> fahrenheit;
    temp.convertTtoC(fahrenheit);
    cout << temp;       //use overloaded stream insertion operator <<
    cout << "\nPlease enter degrees in Celsius\n";
    cin >> celsius;
    temp.convertTtoF(celsius);
    cout << temp;
    cout << "\nNow we change the temperature ourselves:"
         << "\nPlease enter degrees in Fahrenheit\n";
    cin >> fahrenheit;
    temp.setFdegrees(fahrenheit); //use mutator
    cout << temp;
    return 0; // destructor called automatically
}
```

Output

```
The current temperature is 32 degrees Fahrenheit, which is 0 degrees
Celsius
Please enter degrees in Fahrenheit
50
10 degrees Celsius is equal to 50 Fahrenheit

Please enter degrees in Celsius
40
40 degrees Celsius is equal to 104 Fahrenheit

Now we change the temperature ourselves:
Please enter degrees in Fahrenheit
50
40 degrees Celsius is equal to 50 Fahrenheit

Goodbye!
Process returned 0 (0x0)   execution time : 16.543 s
Press any key to continue.
```

The path that COS1512 follows through the prescribed book

The path that COS1512 follows through the prescribed book can be outlined broadly as follows:

- Section 1.2 in chapter 1 provides a general overview over programming and problem-solving with a brief **introduction to object-oriented programming**.
- Though most of chapters 4 and 5 have been covered in COS1511, section 4.6 (**overloading functions**) and section 5.5 (the **`assert` macro**) are included to ensure that students have the necessary background knowledge to understand and implement object-oriented programming.
- Chapter 6 uses **file I/O streams as an introduction to objects and classes**, and teaches students how to **use pre-defined classes**.
- Chapter 8 builds on using pre-defined classes by introducing the **standard class `string`**. Chapter 8 also covers **C strings** and provides a **preview of the Standard Template Library** (STL) with the **`vector` class**.

- Chapter 9 further adds to the background knowledge necessary to understand and implement classes and objects by discussing **pointers and dynamic arrays**.
- In Chapter 10 students learn how to **define** their own **classes** in order to create an **abstract data type (ADT)**. Chapter 10 also covers **inheritance** briefly in order to make students aware of this concept.
- Chapter 11 continues to teach more techniques for **defining functions and operators for classes**.
- Chapter 12 covers **separate compilation**, to allow placing the i**nterface and implementation of an ADT** in files separate from each other and separate from the programs that use the ADT.
- In Chapter 14 **recursion** is introduced.
- In Chapter 15 **single inheritance**, i.e. deriving one class from another is covered.
- In Chapter 17 **function and class templates** are covered, which will allow students to understand and use the STL.