CPS235 Object Oriented Programming in C++

LAB3 BESE-15A *15th March 2010*

Composition in C++

Objectives

By the end of this lab, you should be able to

- Create a multi-file program with class declaration in a file separate from the class definition file
- Use enumerations
- Use the concept of Composition (objects of one class used in another class)

Instructions

- If you have any problems, you are encouraged to consult with me.
- Complete the lab within the lab hours and submit it to the folder which will be specified during the lab.

  \\csdept\data\Assignments\Lec Aisha Khalid\OOP\15A\Lab3

- Make sure that you show your programs to me and answer any questions that I may have in order to get full credit for the lab.

Question 1: Separating the Interface and Implementation of a class

Interface of a Class

Interfaces define and standardize the ways in which things such as people and systems interact with one another. For example, a radio's controls serve as an interface between the radio's users and its internal components. The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differently, some provide push buttons, some provide dials and some support voice commands. The interface specifies **what** operations a radio permits users to perform but does not specify **how** the operations are implemented inside the radio.

Similarly, the **interface of a class** describes what services a class's clients can use and how to request those services, but not how the class carries out the services. A class's interface consists of the class prototype (the declaration only) since that defines what functions the class contains, what are their argument and return types. However, the implementation details (bodies of member functions) can be separated from the interface.

It is the usual C++ practice to separate the interface from the implementation by placing them in separate files. If client code does know how a class is implemented, the client code programmer might

write client code based on the class's implementation details. Ideally, if that implementation changes, the class's clients should not have to change. Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully eliminating, changes to client code.

In this example, we'll show how to break up the Temperature class into two files so that

1. The class is reusable
2. The clients of the class know what member functions the class provides, how to call them and what return types to expect
3. The clients do not know how the class's member functions are implemented.

Header Files

Each of the examples that we have studied till now consists of a single .cpp file, also known as a source-code file, that contains a class definition and a main function. When building an object-oriented C++ program, it is customary to define reusable source code (such as a class) in a file that by convention has a **.h** filename extension known as a **header file**. Programs use **#include** preprocessor directives to include header files and take advantage of reusable software components, such as type string provided in the C++ Standard Library and user-defined types like class Temperature that we used at the very beginning of our lectures.

Recall the Temperature class that we defined and used in the lectures. You may have to look at the lecture **2.Classes(I).ppt** in order to refresh it. All of the code was written in a single file with a .cpp extension.

We are now going to separate the code from that file to two files Temperature.h and Temperature.cpp. As you look at the header file Temperature.h, notice that it contains only the Temperature class declaration.

```
//saved as Temperaure.h
#ifndef TEMPERATURE_H
#define TEMPERATURE_H        /*to prevent multiple
                           inclusions of header file*/
class Temperature
{
 public:
  Temperature();
  Temperature(double mag, char sc);
  void display();
 private:
  double magnitude;
  char scale;
};
#endif
```

The Header file `Temperature.h` contains another version of `Temperature` class definition. This version is similar to the one we have seen before but the function definitions are replaced here with

function prototypes that describe the class's `public` interface without revealing the class's member function implementations. A function prototype is a declaration of a function that tells the compiler the function's name, its return type and the types of its parameters. Note that the header file still specifies the class's `private` data member as well. Again, the compiler must know the data members of the class to determine how much memory to reserve for each object of the class.

Implementation Files

Source-code file `Temperature.cpp` given below defines class `Temperature`'s member functions, which were declared in Temperature.h

```
//file  saved as Temperature.cpp
#include <iostream>
#include "Temperature.h"

Temperature::Temperature(): magnitude (0.0), scale('C')
    {}
Temperature::Temperature(double mag, char sc):
magnitude(mag),scale(sc)
{assert(sc =='F' || sc == 'C'}

void Temperature::display() {
  cout << "Magnitude:" <<magnitude ;
   cout <<"  ,Scale:" << scale;
}
```

Notice that each member function name is preceded by the class name and `::`, which is known as the **binary scope resolution operator**. This "ties" each member function to the (now separate) `Temperature` class definition, which declares the class's member functions and data members. Without "`Temperature::`" preceding each function name, these functions would not be recognized by the compiler as member functions of class Temperature, the compiler would consider them "free" or "loose" functions, like `main`. Such functions cannot access Temperature's `private` data or call the class's member functions, without specifying an object. So, the compiler would not be able to compile these functions.

To indicate that the member functions in `Temperature.cpp` are part of class Temperature, we must first include the `Temperature.h` header file

How Header Files Are Located

Notice that the name of the `Temperature.h` header file is enclosed in quotes (" ") rather than angle brackets (< >). Normally, a program's source-code files and user-defined header files are placed in the same directory. When the preprocessor encounters a header file name in quotes (e.g., "`Temperature.h`"), the preprocessor attempts to locate the header file in the same directory as the file in which the `#include` directive appears. If the preprocessor cannot find the header file in that directory, it searches for it in the same location(s) as the C++ Standard Library header files. When the preprocessor encounters a header file name in angle brackets (e.g., <`iostream`>), it assumes that the

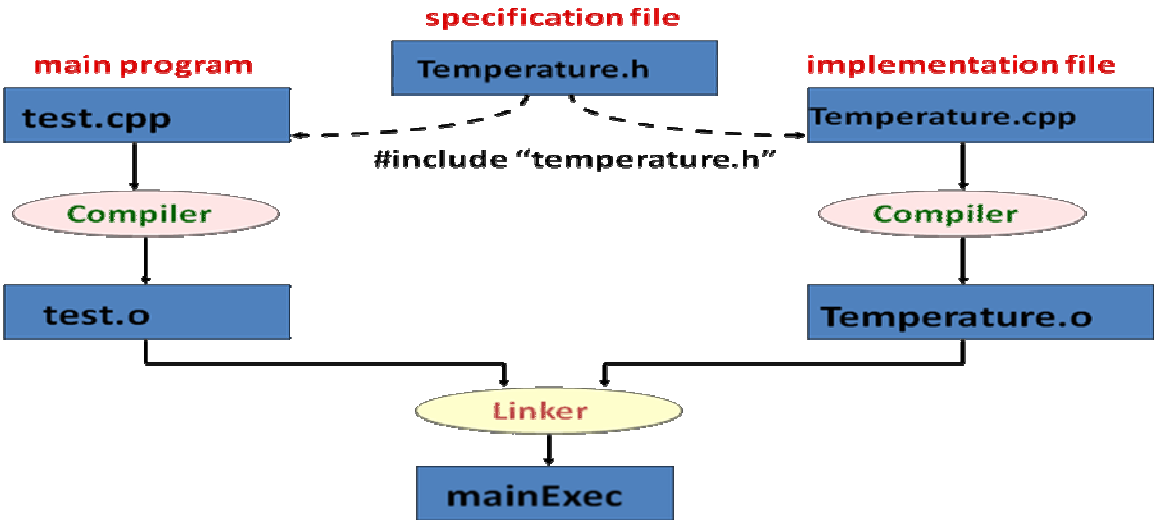
header is part of the C++ Standard Library and does not look in the directory of the program that is being preprocessed.

Main program that uses the Temperature class

```
//saved as test.cpp
#include <iostream.h>
#include <conio.h>
#include "Temperature.h"
int main()
{
   Temperature t1;
   Temperature t2(100.0);
  Temperature t3(100.0,'F');
  t1.display();
   t2.display();
   t3.display();
   getch();
   return 0;
}
```

Separate Compilation and Linking of Files

The following figure shows how the classes are linked together



Task to Do

Take the time class in Q2 of Lab2 and separate the interface and implementation of the class into two files, time.h and time.cpp. Then write a test program (saved as test.cpp) to create objects of the time class in main(). First save the class declaration (only the prototype) in time.h. Then put the implementation of all time class functions in time.cpp. The procedure to run a multi-file program in Borland C++ is given below:

Compiling multi-file programs in Borland C++ version 5.02

1. Go to File -> New -> Project
2. Name the project **test.ide**
3. In the **target type** field, select **EasyWin[.exe]**
4. Uncheck Class Library, BWCC and No Exceptions, then click on OK
5. If there are any nodes created in addition to test.exe, delete them by right clicking on each of them and selecting delete node
6. Right click on **test.exe** and select add node. From the options select **time.h**
7. Similarly add the file **time.cpp** to your project
8. Create a node named **test.cpp** and write your main program in that file to test the time class.
9. Go to Project -> Build all. If there are no errors, click on Run and that should execute your project.

Question 2: Stack Example

Let's define a `Stack` of characters. Think about the data members (state variables) and member functions (behaviors) of `Stack` class that stores characters.

```
#include <iostream.h>
#define MAX_LEN 100

class Stack {
/*  private part can be at the beginning of the class
 *  without the 'private' keyword. Class contents are
 *  by default private.                                */
    char s[MAX_LEN];
    int top; // points to the element on the top of stack

public:
    void reset() {
        top = 0;
    }
    void push(char c) {
        s[++top] = c;
    }
    char pop() {
        return s[top--];
    }
    char topOf() {
        return s[top];
    }
    bool empty() {
        return ( top == 0 );
    }
    bool full() {
        return ( top == MAX_LEN - 1 );
    }
};
```

Examine the Stack class. Is the first element of the stack used? How can you correct this design? What are the benefits of dividing the class design into private and public areas?

Trace the following main and show the output:

```
void main() {
    Stack s;
    char str[40] = {"This is a sample"};
    int i = 0;

    cout << str << '\n';
    s.reset();
    while ( str[i] )
        if ( !s.full() )
            s.push(str[i++]);
    while ( !s.empty() )
        cout << s.pop();
    cout << '\n';
}
```

Look at the modified version of the same Stack class:

```
class Stack {
    enum {EMPTY = -1, FULL = MAX_LEN - 1};
    char s[MAX_LEN];
    int top; // points to the element on the top of stack

public:
    void reset() {
        top = EMPTY;
    }
    void push(char c) {
        s[++top] = c;
    }
    char pop() {
        return s[top--];
    }
    char topOf() {
        return s[top];
    }
    bool empty() {
        return ( top == EMPTY );
    }
    bool full() {
        return ( top == FULL );
    }
};
```

Question 3: Composition

1. For each of the following classes, define all member functions outside the class declaration, and use initialize lists where possible. Also use the **`const`** keyword where applicable.
2. Create a class **`airtime`** with private data members **`hours`** and **`minutes`**, both of type **`int`**. Provide a default constructor for the class which initializes **`hours`** and **`minutes`** to 0 and a 2-argument constructor to initialize these variables to specified values. Write a **`display()`** function to show the time in **`hour:minute`** format. Add a **`get()` `function`** to your class which takes an **`airtime`** value as input from the user.
3. Create another class **`flight`** which has a **`long`** variable to hold the **`flight_number`** and a **`long`** variable to hold the **`cost`** of the flight. It also contains two objects of the **`airtime`** class, one to hold the **`departure`** time and the other to hold the **`arrival`** time. Provide a **`get()`** member function to take as input flight data from the user. This function should prompt the user to enter flight number, departure time, arrival time and the cost. Also provide a **`display()`** function to show the flight data.
4. Take the **`Stack`** class from Question 2 and modify it so that it holds a stack of **`flight`** objects rather than characters.
5. In **`main()`** , create a **`Stack`** object, get flight data from the user, store it on the stack and then display the contents of the stack. Add some more flight objects to the stack and then remove some of them. Each time a flight object is removed from the stack, its **`cost`** is added to a **`counter`** variable. After executing a few add and remove operations, report the value of the cost counter.
6. If you were to implement this program