CMP2801M Workshop Week 9

VIRTUAL FUNCTIONS

Write and test your code, and add comments so that you can refer to it later (focus on *why* you made a particular decision to do something a certain way). This will help you to understand your work when you refer to it in future. Please refer to the lecture notes or ask for help if needed! There are two parts to this workshop: **I)** looking back to the last topic, for review and practice of Operator Overloading and Friends, and **II)** exploring the fundamentals of this week's topic, Virtual Functions. This week, also a bonus Christmassy themed task in part **III**.

PART I – Operator Overloading and Friends

Task 1:

Fix the following code so that it compiles and runs correctly (the desired output as shown in the `main` function). You need only change the code in the class declaration for A, but do not change the access modifier of value (it must remain `private`).

```
class A
{
private:
  int value;  // this must stay private
public:
  A(int v): value(v) {}
  void print() { cout << "My value is " << value << endl; }
};


int main()
{

  A a1 = A(2);
  A a2 = A(3);

  a1.print();
  a2.print();

  A a3 = add(a1, a2);
  a3.print();  // this should display "My value is 5"

  return 0;
}
```

```
A add(A a1, A a2)
{
  return A(a1.value + a2.value);
}
```

Task 2:

Overload the plus (+) operator so that it behaves in the same way as the `add(…)` function in Task 1. Add a line to the `main` function that adds the three objects (`a1, a2, a3`) together using your newly overloaded operator, and assign the result to a new object A `a4`. Check the value of `a4` using the `print()` function.

Task 3:

Overload the postfix increment operator (a++) so that it increases the `value` of A objects by one. Check its behaviour with the following code: `A a5 = a4++;` and check the values of `a4` and `a5` again by using `print()`.

Task 4:

Use the code snippet below (left) to implement a class called `Employee`. Add definitions for the constructor, and `print()` function. Implement a second class called `Manager` that is derived from `Employee`, using the code snippet below (right). The `Manager` class should have the following definitions:

- A constructor that calls the base constructor of `Employee`
- A member function called `assignEmployee(…)` that takes an employee reference as its sole argument, and stored it in the private memory pointer variable `assigned`. It should return nothing (`void`).
- A member function called `giveRaise(…)` that takes a double amount as its sole argument, and adds that amount to the assigned employee's salary (`assigned`). It should return nothing (`void`).

Compile and run the program and verify the following:

- Calling the `print()` function on `Employee`/`Manager` objects should display the name and salary
- The `Manager` class should have access to their employee's private member variables
- The salary of a new member of staff is increased to the correct value (not the `Manager`!)

```
#include <iostream>
using namespace std;

class Employee
{
private:
  string name;
  double salary;
public:
  Employee(string n, double s);
  void print();
};

Employee::Employee(string n, double s)
{
  // store the values in name and salary
}

void Employee::print()
{
  // print out the name and salary
}


int main()
{

  Employee cook = Employee("Jesse", 5000);
  cook.print();  // should print "Jesse earns $5000"

  Manager boss = Manager("Walter", 10000);
  boss.print();  // should print "Walter earns $10000"

  boss.assignEmployee(&cook);
  boss.giveRaise(5000);
  cook.print(); // should print "Jesse earns $10000"

  return 0;
}
```

```
class Manager : public Employee
{
private:
  Employee *assigned;
public:
  // declare constructor here
  // declare assignEmployee function here
  // declare giveRaise function here
};

// define the constructor/functions here
```

PART II – Virtual Functions

In this part of the workshop, we preview/review (depending on whether you have seen the lecture yet this week or not) the topic for week 9 (Virtual Functions). More in depth exercises on this topic will come in the next workshop (week 11), as in previous weeks.

Task 5:

Implement an inheritance hierarchy comprised of three layers: i.e. three classes, in a single chain of inheritance. Class A is the parent, class B inherits from A, and C inherits from B. Each of these classes should have the following properties:

- A public `print()` method, printing the class name to console. Each child class should override this;
- A private member variable, of some type and name that is unique for each class (i.e. each child class adds an additional private variable, as well as inheriting from its parent);
- An accessor function that returns the value of the defined private member variable – `getValue()`; and
- An overloaded constructor that initialises the value of the private member variable.

Given this arrangement, test the operation of the following `main` function. You will need to update the constructor calls according to your chosen private member variables. Does this work as you are expecting? What characteristics are being taken advantage of here?

```
int main () {
      A a = A(42);
      B b = B(20.21);
      C c = C(“test”);
      A *pB = &b;
      A *pC = &c;
      a.print();
      pB->print();
      pC->print();
      //--put code for task 6 main() here
      return 0;
}
```

Task 6:

Implement the following function (below, left), and test using additional code in the `main()` function from task 5 (below, right).

```
void doStuff (const A _in) {
    cout << “Returned value: “ <<
        _in.getValue() << endl;
}
```

```
doStuff(a);
doStuff(b);
doStuff(c);
```

Consider what you would expect to see, and compare with what actually happens.

PART III – Bonus Task: Making Music

This task will provide you with some practice of various of the topics we have covered in this module so far (and, if you're not careful, an annoying tune stuck in your head), including the Virtual Functions topic of this week 9 lecture.

Before you start, please note that this task involves the creation of sound, so please be considerate of those around as you engage with this exercise…

Let us start from the code provided in `music.cpp`. First, look through the code, and see what it does, and how it is organised. The remainder of the tasks are modifications and extensions to this base code to make it more usable and configurable, while practicing a number of the concepts covered in the module. Add/implement the following features:

1. Ensure that the classes you use are appropriately split into declaration/definitions, and separate files accordingly.

2. Implement a new base class (e.g. `Note`) that `CNote` inherits from. This base class could be an interface class. Create another new class (inheriting from `Note`) that handles pauses (between notes) in an appropriate way.

3. Modify the `CMelody` class to use base class (`Note`) pointers (i.e. take advantage of polymorphism) to handle both `CNotes` and `Pauses`.

4. Design and implement an appropriate representation of melodies that could be read in from an external file, stored in a `CMelody` instance, and played.

5. Implement a basic (command line) user interface that allows the user to load music from a file and/or define a melody.