COEN 79

Object-Oriented Programming and Advanced Data Structures

Assignment #2 - Due: October 9, 2018 – 11:59PM

Name: Jordan Murtiff Date: 10-9-18

- Number of questions: 10
- Points per question: 0.4
- Total: 4 points

1. What are the effects of using *inline member functions*?
   Write a class that uses at least two inline member functions.

The effects of using inline member functions is that it may make your code run more efficiently during runtime (since the program will no longer have to jump to the function, execute, and then return to where the program left off), however, it may be inefficient in terms of space (memory). Since many copies of the function will exist when you run your code it will take up more memory to run your program, but since the program doesn't have to keep jumping back to your function, the program will run faster in terms of time complexity. We usually only want to do inline member functions when the functions are only a single short statement and not when the implementation consists of more than one line of code.

Example:

```
class dog
{
 public:
  string get_name()
  {
   return name;
  }
  int get_age()
  {
   return age;
  }
  double get_weight()
  {
   return weight;
  }
 private:
  string name;
  int age;
  double weight;
};
```

2. What are the differences between *references* and *pointers*?

The differences between references and pointers is that pointers are variables that hold a memory address of a variable (where the address value stored in the pointer can be changed) while a reference holds a memory address of a variable that cannot be changed. Additionally, if we wish to get or change the value of the variable stored at a pointer's address, we have to explicitly dereference the pointer using the indirection operator *. If we use a reference, however, we do not need to reference or dereference the references to get or change the value that the reference is specifically pointing to. We can use references as formal parameters as functions so that we don't have to remember whether or not to use the indirection operator * when we want to assign or get the value of the reference. It makes code easier to read and easier to understand.

In essence, a reference does not have the same syntax as a pointer, but it acts almost the same as a pointer in terms of its use and function. A pointer can have its memory address changed, but once a reference is pointing to a specific variable, its content/memory address cannot be changed unless we either declare a new reference to reference a different variable.

3. What is the *value semantics* of a class?
Write a class for which you can *rely on the compiler* to provide you with valid value semantics.

The Value semantics of a class is how values are copied from one object to another. This is done with either the assignment operator (=) or the copy constructor. Assuming that our classes do not use pointers as private member variables, the compiler can create an assignment operator than simply copies each member variable from the object to the right of the assignment to the object on the left of the assignment.

Example:
In dog.h:

```
class dog
{
 Public:
 dog();
 dog(dog input);
 string get_name();

 Private:
 String name;
 int age;
double weight;
};
```

In main.cpp:

```
dog Joey;
dog Betty;
Joey = Betty;
```

In this case we take all the private member variables from the dog object “Betty” and copy them over to the dog object “Joey”.

The second way of copying over values from one object to another is through the copy constructor. This constructor creates a new object and then copies over all the characteristics from one object to the newly constructed object. The constructor can only have one parameter and it must be the same as the constructor’s class.
In main.cpp:

```
dog Joey;
dog Betty = Joey;
```

In this case we take all the private member variables from the dog object “Joey” and copy them over to the newly constructed dog object “Betty”.

4. Where should we include the *invariants* of a class? Why?

We should include invariants of a class in the implementation file or .cpp file of a program. Invariants are conditions that are implicit parts of every function’s pre and postconditions. They are rules that dictate how the member variables of a class represent a value. We put the invariants of a class inside the .cpp file because the programmer or person that wants to use a specific class does not need to know the invariants to use the class, so we don’t need to include the invariants in our header files (where documentation normally goes). The person that has implemented the program needs to know the invariants, but the person using the program doesn’t need to know about the invariants.

5. The header of the `point` class is as follows:

```
1.  class point
2.  {
3.  public:
4.      // CONSTRUCTOR
5.      point (double initial_x = 0.0, double initial_y = 0.0);
6.
7.      // MODIFICATION MEMBER FUNCTIONS
8.      void set_x (double& value);
9.      void set_y (double& value);
10.
11.     // CONST MEMBER FUNCTIONS
12.     point operator+ (double& in) const;
13.
14. private:
15.     double x; // x coordinate of this point
16.     double y; // y coordinate of this point
17.
18. };
```

- Which line of the following code results in error? Explain why.

```
1.  main() {
2.      point myPoint1, myPoint2, myPoint3;
3.      double shift = 8.5;
4.      myPoint1 = shift + myPoint2;
5.      myPoint3 = myPoint1.operator+ (shift);
6.      myPoint1 = myPoint1 + shift;
7.  }
```

Line 4 is incorrect. Line 4 is trying to use the overloaded addition operator but it not doing so correctly. Since point.h overloads the addition operator as a member function (since it is defined in the class) there are only a few ways in which you can use the overloaded addition operator.

- What is the solution?

So the easiest solution is to just switch the two statements in Line 4:

myPoint1 = myPoint2 + shift

The above statement would correct the code without having to write additional functions.

6. Why the following code does not compile? what is the solution?
Note: The notation "<>" is used for *template* classes. You will learn about this in future chapters.

```
1.  #include < iostream >
2.  #include < list >
3.
4.  namespace coen79 {
5.      template < typename T >
6.      class list   {
7.          private:
8.              int array[20];
9.      };
10. }
11.
12. using namespace std;
13. using namespace coen79;
14.
15. int main(int argc, const char * argv[])
16. {
17.     using namespace std;
18.     list < int > v1;
19.     list < int > v2;
20.     return 0;
21. }
```

The following code does not compile due to the fact that the compiler does not know which "list" class to use, the one defined in the coen79 namespace or the one defined the std namespace. Since the header file for this program redefines the class list (which is an already existing class in C++) the compiler does not know which "list" class to use, the user defined one in coen79 namespace or the one defined in the std namespace.

You can solve this in one of two ways. You can remove the namespaces in the main.cpp file (the two using namespace statements) and use the scope resolution operator for whichever "list" class you intend to use or include only one of the namespaces (either std or coen79) and use the "list" class from one of those namespaces. You could do std::list or coen79::list or use one namespace and then define the list objects normally as done above.

7. What is the output of this code? Discuss your answer.

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class CMyClass {
5.      public:
6.          static int m_i;
7.  };
8.
9.  int CMyClass::m_i = 0;
10.
11. CMyClass myObject1;
12. CMyClass myObject2;
13. CMyClass myObject3;
14.
15. int main() {
16.     CMyClass::m_i = 2;
```

```
17.     myObject1.m_i = 1;
18.
19.     cout << myObject1.m_i << endl;
20.     cout << myObject2.m_i << endl;
21.
22.     myObject2.m_i = 3;
23.     myObject3.m_i = 4;
24.
25.     cout << myObject1.m_i << endl;
26.     cout << myObject2.m_i << endl;
27. }
```

The static variable (m_i) defined in the CMyClass is shared between all objects of the class. This means that once the three objects named myObject1, myObject2, and myObject3 are all declared, they all can access the same static variable between all three of them. This means that once the function runs it should print out:
1
1
4
4
For the first two lines, the static variable is changed from 2 to 1 (and so 1 is then printed out twice) and for the second two lines, the static variable is changed from 3 to 4 (and so 4 is printed out twice).

8. I wrote the following code to print 11 “Ouch!”. Why it is not working as expected?

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  int main(int argc,  const char * argv[])
5.  {
6.      std::size_t i;
7.      for (i = 10; i >= 0; --i)
8.          cout << "Ouch!" << endl;
9.      return 0;
10. }
```

The code is not working as intended because size_t is an integer data type that can only hold non-negative numbers. In this case once the variiable "i" goes to 0 and then decrements it will become the highest possible value (bits change from 000000 to 111111) and so we get an infinite loop since the condition of the loop will never be falsified. This means that the loop will print out “Ouch!” indefinitely instead of only printing it out 11 times.

This is due to the fact that i is of type size_t, which is always non-negative (meaning that if it tries to become a negative number, it will instead become the highest possible positive number).

9. In the following code, indicate if the selected lines are legal or illegal:

```
#include <iostream>


class small
{
public:
    small( ) {size = 0;};
    void k() const;
    void h(int i);
    friend void f(small z);

private:
    int size;
};


void small::k() const
{
    small x, y;
    x = y; // LEGAL/ILLEGAL?
    x.size = y.size; // LEGAL/ILLEGAL?
    x.size = 3; // LEGAL/ILLEGAL?
};

void small::h(int i)
{

};

void f(small z)
{
    small x, y;
    x = y; // LEGAL/ILLEGAL?
    x.size = y.size; // LEGAL/ILLEGAL?
    x.size = 3; // LEGAL/ILLEGAL?
    x.h(42); // LEGAL/ILLEGAL?
};


int main() {

    small x, y;
    x = y; // LEGAL/ILLEGAL?
    x.size = y.size; // LEGAL/ILLEGAL?
    x.size = 3; // LEGAL/ILLEGAL?
    x.h(42); // LEGAL/ILLEGAL?

    return 0;
}
```

Legal
Legal
Legal
Legal
Legal
Legal
Legal
Legal
Illegal
Illegal
Legal

10. Modify the following code to generate the given output. Do not modify the `main` function.

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class box {
5.
6.  public:
7.      // Constructor definition
8.      box(double l = 2.0, double b = 2.0, double h = 2.0) {
9.              length  = l;
10.             breadth = b;
11.             height  = h;
12.             }
13.
14.     double volume() {
15.         return length * breadth * height;  }
16.
17. private:
18.     double length;   // Length of a box
19.     double breadth;  // Breadth of a box
20.     double height;   // Height of a box
21. };
22.
23. int main(void) {
24.     box Box1(3.3, 1.2, 1.5);   // Declare box1
25.
26.     box Box2(8.5, 6.0, 2.0);   // Declare box2
27.
28.     return 0;
29. }
```

```
Output:
Number of box objects created so far: 1
Number of box objects created so far: 2
```

Within the public section of the box class (but not in the constructor or volume function) add the line:

static int count;

Within the constructor of the box class add these two lines after line 11:
count++;
cout << "Number of box objects created so far: " << count << endl;

And in line 22 before the main function (but not in the box class) add the line:
int box::count = 0;

And after two box objects are created you will bring out:
Number of box objects created so far: 1
Number of box objects created so far: 2

So we create a static variable, set it to 0 before the main function runs, then update the value of the static variable each time a new box object is instantiated.