# Section Solutions #6

**Problem 1: The Philosophy of Objects and Classes**

**a)** In discussing objects and classes, it helps to be clear on our terminology. A class involves two different roles for programmers. The first role is the *implementer* of the object: the programmer who designs and writes the object. The second role is the *client*: a programmer who uses the object in writing code. Although we'll usually talk as if there is one client who is distinct from the one implementer of an object, there can be more than one person implementing an object and more than one client of that object. Also, it could be that the implementer and the client are the same person. In any of these cases, it is important to keep these two roles distinct. And these roles are both different from the *user,* the person using the program or application that the client and the implementer helped to create.

A C++ class has two parts, an interface and an implementation. The interface (the class declaration) is the list of functions and data types that the client can use in their code, and a concise explanation of how to use those functions and data types. The implementation provides the functionality. The client can use the data types to declare variables, but the client should never read or change the private data stored in the structure by directly accessing its fields. By writing code that reads or sets explicit fields or any way depends on the underlying implementation of the object, the client is "breaking the wall"— the separation between the interface and the implementation.

In C++, the `public` and `private` keywords are used to discriminate between an object's public data and methods, and its private data and methods. This allows the implementer of an object to enforce the wall of abstraction at compile time. Even if a client knows that the number of elements in the stack is stored in an integer field named "size", if they attempt to directly access a class's private field like this:

```
stack.size = 0;
```

the compiler will generate an error. Instead, the client should call the public method `stack.size()` to retrieve the number of elements in the stack.

If we didn't specify the fields in a class as private, a misbehaving client could change the values of the internal data. However, the implementer is relying on the data having certain values and being changed in a way that the implementer can control by providing methods that the implementer has carefully thought out. If the client changes data out from under the implementer, the client can mess up the implementation of the class. Also if the implementer would like to change the implementation, the client who breaks the wall could end up with code that doesn't even compile now or has bugs lurking all over it due to changes in the class implementation.

**b)** The use of classes gives the implementer great freedom: freedom to change the implementation, fix bugs, re-design the data structure, substitute more efficient algorithms, etc. all without disturbing the client. The client would not want to have to go back and rewrite their code just because the implementer felt like speeding up the implementation!

Another main advantage of an object is that it hides a lot of detail for the client. Clients can use the object without thinking much about how it is implemented. The class is written and debugged once, and it can be used again and again without reinventing the wheel. The client is free to solve bigger and more interesting problems. Also, many programmers can work together in a fairly organized way, without being involved with the details of each other's work.

**c)** a. Allowing for `width` and `height` public data members would essentially allow the client to break the wall of abstraction, as this exposes underlying implementation of the Rectangle class. In addition, if the implementer wanted to change the implementation (i.e. only store coordinates instead of having width/height variables), the client's code would immediately break.

b. While having `getWidth()/getHeight()` member functions protect the integrity of internal data, it forces the client to do the `(2 * getWidth() + 2 * getHeight())` perimeter calculation that the Rectangle class is supposed to provide.

c. The state of the object can be easily messed up as a public `perimeter` data member can be unexpectedly changed by anyone at any time, not just when the rectangle changes size. While this does not expose implementation, it does not enforce any constraints on the integrity of the data the class provides to the client, and thus defeats the purpose of data encapsulation.

d. A `getPerimeter()` member function allows for the calculation to be done on the implementation side, as well as enforcing one-way access of the perimeter value. This is the best option.

**Problem 2: A simple class**

```
class Rectangle {
  public:
      Rectangle(int x, int y, double width, double height);
      ~Rectangle();

      double getPerimeter();
      double getArea();

      void translate(int dx, int dy);
      void scale(double factor);

      void print();

  private:
```

```
        int x, y;
        double width, height;
        //perhaps some more functions or data…
    };
```

Note that since the constructor has picked the same variable names as our instance variables, we will have to use the **this** syntax to set up our initial state, like so:

```
    Rectangle::Rectangle(int x, int y, double width, double height) {
      this->x = x;
      this->y = y;
      this->width = width;
      this->height = height;
    }
```

**Problem 3: Function Pointers and Templates**

a)

```
template <typename ElemType>
  ElemType FindMax(Vector<ElemType> &v, int (cmpFn)(ElemType, ElemType)
                                                        = OperatorCmp)
  {
      ElemType max = v[0];
      for(int i = 1; i < v.size(); i++)
      {
            if(cmpFn(max, v[i]) < 0)
            {
                  max = v[i];
            }
      }
      return max;
  }
```

b)

```
struct Car {
    string name;
    int weight;
    int numAirbags;
};

int CarCompare(Car first, Car second)
{
      int diffAirbags = first.numAirbags - second.numAirbags;
      if(diffAirbags == 0)
            return first.weight - second.weight;
      else
            return diffAirbags;
}

int main()
{
      Vector<Car> cars;

      // assume variables have been initialized here

      Car safestCar = FindMax(cars, CarCompare);
```

```
      cout << safestCar.name << endl;

      return 0;
}
```

### Problem 4: Templatized Functions

```
template <typename ElemType>
  void Filter(Queue<ElemType> & q, bool (predFn)(ElemType))
  {
    int size = q.size();
    for(int i = 0; i < size; i++)
    {
      ElemType currValue = q.dequeue();
      if(!predFn(currValue))
      {
        q.enqueue(currValue);
      }
    }
  }
```

Because we go through the entire queue and when we keep elements we stick them on the back, by the time we're done the queue is in the original order (minus the removed elements).

### Problem 5: More Templatized Functions

**a)**

```
template <typename ElemType>
  void RemoveDuplicates (Vector<ElemType> & v, int (compFn)(ElemType,
ElemType))
  {
    for(int i = 0; i < v.size(); i++)
    {
      for(int k = v.size()-1; k > i; k--)
      {
        if(compFn(v[i], v[k]) == 0)
          v.removeAt(k);
      }
    }
  }
```

```
b)
int CompareIntAbsolute(int first, int second)
{
      first = abs(first);
      second = abs(second);
      if(first < second) return -1;
      else if(first > second) return 1;
      else return 0;
}

RemoveDuplicates(v, CompareIntAbsolute);
```