The abstract class Battalion

We need to create the abstract class **Battalion**. Having an abstract class means that the class must have at least one abstract method (or what is the same, a pure virtual method). The abstract class **Battalion** should look something like this:

```
class Battalion {
 public:
  Battalion(const char* name) : name(name) {} // constructor

  const char* getName() const { return name; } // getter

  virtual int combatValue() const = 0; // pure virtual method

 protected:
  const char* name; // battalion name
};
```

Note that the attribute **name** is in the **protected** area and cannot be accessed from outside the class definition. On the other hand, all the methods (the **constructor**, **getName()** and **combatValue()**) are **public** members that can be accessed from outside the class.

The **constructor** here just initializes the **protected** name attribute with the parameter and does nothing more.

The **getName()** method provides public access to the **name** attribute, which cannot be directly accessed because it is a protected attribute. Note also that this method is **const**, meaning that **getName()** is a method that cannot modify any attribute of the class.

The **combatValue()** method is an abstract method, or what is the same, a pure virtual method. A pure virtual method is a method with no implementation in the base class (**Battalion** in our case) and is intended to be implemented in subclasses. The way in C++ to declare abstract (or pure virtual) methods is by placing the keyword **virtual** at the beginning of the method declaration, and also by adding the final part, **= 0**, meaning that this method is not intended to have implementation in this class. Having an abstract method in a class makes the whole class abstract, and abstract classes cannot be instantiated. For that reason, it is mandatory to have subclasses that override abstract methods of their base classes (that is, provide the implementation of those methods). In this case, subclasses of **Battalion** will provide concrete implementations of this abstract **combatValue()** method. Subclasses that do this are not abstract anymore and can be instantiated.

Note also that **combatValue()** has the **const** keyword at the end, also meaning that implementations of this method in subclasses cannot modify any attributes in the class.

The subclass Infantry

The statement asks you to implement the class **Infantry** as a subclass of the base class **Battalion**. It needs to have two private attributes **bayonetSoldiers** and **musketSoldiers** that tell the number of soldiers of each type in an **Infantry** object. Also, it needs to have a **constructor** that specifies the name of the **Infantry** (which is a **Battalion** itself, actually) and the number of bayonet and musket soldiers. And last, it must provide the implementation of the pure virtual method **combatValue()** declared in the base class **Battalion**. The **Infantry** class should look similar to this:

```
// class Infantry inherits from Battalion (is a subclass of Battalion)
class Infantry : public Battalion {
 public:
  Infantry(const char* name, int bayonetSoldiers, int musketSoldiers) // constructor
   : Battalion(name), // invoke the base class constructor
    bayonetSoldiers(bayonetSoldiers), // attribute initialization
    musketSoldiers(musketSoldiers) // attribute initialization
  {}

  // Infantry implementation of the pure virtual method
  // combatValue declared in the base class Battalion
  int combatValue() const { return musketSoldiers * 2 + bayonetSoldiers; }

 private:
  int musketSoldiers;
  int bayonetSoldiers;
};
```

The first line (**class Infantry : public Battalion**) declares Infantry as a subclass of **Battalion**. This means that **Infantry** is a **Battalion** itself, and as a **Battalion**, it also has the **name** attribute, and the **getName()** method.

The **constructor** has three parameters, the name of the infantry and the number of bayonet and musket soldiers. It invokes the base class constructor with the line **Battalion(name)** defined in the **Battalion** class, and then it also initializes the **bayonetSoldiers** and **musketSoldiers** attributes with the constructor parameters.

Last, the **Infantry** class must provide an implementation for the pure virtual method **combatValue()** declared in the base class. Note that the method has the same signature here and in the base class (both return an **int** and bot are **const**). However, the use of the **virtual** keyword here in the subclass is optional (it is actually not present in this case). The **virtual** keyword is only needed in the base class, where the abstract method is declared.

The implementation of **combatValue()** for the **Infantry** class is as described in the statement: it returns the total attack power of the infantry, which is the combat value of each type of unit multiplied by the number of units of each type respectively.

The subclasses Chivalry and Artillery

The implementations of both **Chivalry** and **Artillery** are almost identical to **Infantry**, only changing a few names (the class name and the attribute names) and providing a slightly different implementation of the virtual method **combatValue()**. Pay attention to the statement to assign the proper combat value to each type of combat unit in the implementation of this method!!!

The function victoryInBattlefield()

As the statement says, the function **victoryInBattlefield()**, given two pointers to **Battalion** instances, must return the name of the battalion with a higher combat value (for that purpose we must use the **combatValue()** abstract function). In the case the two battalions have the same combat value, it has to return the string **"Same combat value"**. A possible implementation of this function could be this:

```
const char* victoryInBattlefield(Battalion* bat1, Battalion* bat2)
{
 if (bat1->combatValue() > bat2->combatValue())
   return bat1->getName();
 else if (bat1->combatValue() < bat2->combatValue())
   return bat2->getName();
 else
   return "Same combat value";
}
```

The global function **victoryInBattlefield()** receives two parameters of the type **Battalion*** (pointer to **Battalion**). Any of the parameters passed could be an instance of **Infantry**, **Chivaltry**, or **Artillery**, because remember, **Infantry**, **Chivaltry**, and **Artillery** objects are **Battalions** also (that is, subclasses of **Battalion**), and pointers to the subclasses are compatibles with pointers to the base class. That is why we can use pointers of the type **Battalion** to point to any object that is a subclass of **Battalion**.

This function calls the **combatValue()** method of the **Battalion** class. Remember that the **Battalion** class does not actually give an implementation for this method, because it is an abstract method (pure virtual). But it is expected that the two parameters are not simple battalions (**Battalion** is an abstract class and cannot be implemented alone), but any of the subclasses that override the **combatValue()** method (**Infantry**, **Chivaltry**, or **Artillery**).

So, whenever the **combatValue()** method is invoked, despite the fact that **bat1** and **bat2** are pointers to **Battalion**, the implementations in the subclasses are invoked, because **combatValue()** is a virtual method.