CSE 425 Studio: Object-Oriented Programming Studio II

These studio exercises are intended to continue to expand your experience with object-oriented language features in C++ as well as to continue to explore logic programming features and abstractions.

In this studio you will again work in self-selected groups of ~3 people, and will report the results of your work on the studio exercises in an e-mail to the course account. Students who are more familiar with the material are encouraged to partner with those who are less familiar with it, and asking questions of your classmates and professor during the studio sessions is highly encouraged as well.

Please record your answers as you work through the following exercises. After you have finished please send e-mail with your answers to the required exercises, and to any enrichment exercises you completed, to the course account (cse425@seas.wustl.edu), with "Object-Oriented Programming Studio II" in the subject line. The enrichment exercise is optional but provides a good way to dig into the material a little deeper, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people of your choice (preferably a mix of people familiar and unfamiliar with C++) and write down the names of the team members as the answer to this first exercise.

2. Extend your code from the previous studio (or optionally, copy it over into a new Microsoft Visual Studio 2013 project and work on that version) as follows. Declare and define a **`ConjunctiveClause`** class that has a single member variable – a vector of pointers to **`Predicate`** objects (i.e., a member variable of type **`vector<Predicate *>`**). In the **`ConjunctiveClause`** class, declare and define an **`add`** method with a **`void`** return type that takes a pointer to a **`Predicate`** object and calls its member variable's **`push_back`** method to add that pointer to the vector. In the **`ConjunctiveClause`** class, declare and define a **`print`** method with a **`void`** return type that takes no arguments, which iterates through the pointers stored in its vector member variable and calls the print method of the object pointed to by each of them.

In your program's main function, declare instances of different concrete subclasses of the **`Predicate`** base class and an instance of the **`ConjunctiveClause`** class. Call the **`ConjunctiveClause`** object's **`add`** method with each **`Predicate`** object's address (given by the address-of operator **`&`**), and then call the **`ConjunctiveClause`** object's **`print`** method. Build and run your program, and as the answer to this exercise, please show your implementation of the **`ConjunctiveClause`** class, and the output that was produced when you ran your program.

3. Add a pure virtual **is_subtype** method to your **Predicate** base class, which takes a pointer to a **Predicate** object and returns a **bool** result. In each of your concrete subclasses of **Predicate**, override that method so that it returns true if and only if a dynamic case to that specific type returns a non-zero pointer. For example, in the **ZeroIntegerPredicate** class the method might be implemented along the lines of:

```
return 0 != dynamic_cast<ZeroIntegerPredicate *> (ptr_argument);
```

In your program's main function, declare additional instances of the different concrete subclasses of the **Predicate** base class and on an object of each type call its **is_subtype** method with (1) its own address, (2) the address of another object of the same type, and (3) the address of another object of an unrelated type. Build and run your program, and as the answer to this exercise, please describe which combinations you tried and what the result was in each case.

4. Add an **is_compatible** method to your **ConjunctiveClause** class, which takes a reference to a **ConjunctiveClause** object and returns a **bool** value. This method should return **true** if and only if each **Predicate** object in the **ConjunctiveClause** object on which the **is_compatible** method was invoked can be uniquely matched to a **Predicate** object of the same type in the **ConjunctiveClause** object that was passed into the call to the **is_compatible** method. To determine whether or not two **Predicate** objects are of the same type, you can simply pass each into a call to the other's **is_subtype** method: if both calls return true they are of the same type (and otherwise are not).

If the **ConjunctiveClause** objects contain different numbers of pointers to Predicate objects, the **is_compatible** method can immediately return **false**. If they contain the same number of pointers, one way to check for unique matching is to declare a local vector of **bool** values that has the same number of elements as the **ConjunctiveClause** objects' vector member variables and whose entries are all initialized to be **false**. Then, whenever a **Predicate** object in a given **ConjunctiveClause** object (which one you'd designate for this depends on your implementation, though my tendency would be to make it the one that was passed into the call to the **is_compatible** method) is used to make a match, the entry at the same position in the vector of **bool** values can be set to true to indicate that it already has been used to make a match (and should not be considered for a subsequent one).

In your program's main function, declare a number of **ConjunctiveClause** objects and add the addresses of different **Predicate** objects to them so that: (1) some pairs **ConjunctiveClause** objects have exactly the same sequence of types of Predicate objects (and so are compatible); (2) some pairs of **ConjunctiveClause** objects have the same (multi-)set of types of **Predicate** objects but in different sequences (and so still should be compatible); and (3) some pairs **ConjunctiveClause** objects have different (multi-)sets of types of **Predicate** objects (and so are not compatible). Build and run your program, and as the answer to this exercise, please describe the results of calling the **is_compatible** method with different combinations of those **ConjunctiveClause** objects (including passing in the same object on which the call was made).

5. Add a pure virtual **clone** method to your **Predicate** base class, which takes no arguments and returns a pointer to a **Predicate** object. In each of your concrete subclasses of **Predicate**, override that method so that it returns the result of allocating and initializing (via copy construction) a new heap object of the same type. For example, in the **ZeroIntegerPredicate** class the method might be implemented along the lines of:

```
return new ZeroIntegerPredicate(*this);
```

Modify the **add** method of the **ConjunctiveClause** class so that instead of simply storing the pointer to the object that it was passed, it instead stores the result of calling the clone method via the passed object pointer (the arrow operator in C++ lets you call object methods via a pointer). Re-run some of the examples from the previous exercises to confirm that (other than a memory leak that we'll look at next) the code is showing the same behavior. As the answer to this exercise, show your implementation of one of the concrete Predicate subclass' clone methods, and also show your new implementation of the **ConjunctiveClause** add method.

6. Add a **static** (so there is one instance for the entire class) unsigned integer member variable to the **Predicate** base class, which is initialized to 0. Also in your **Predicate** base class, declare and define a default constructor that increments the value of that static member variable and then prints out its value to the standard output stream (using **cout**). Also in your **Predicate** base class, declare and define a destructor that decrements the value of that static member variable and then prints out its value to the standard output stream (**cout**). If you had defined other constructors or destructors, please also extend their behaviors to increment or decrement and then report the value of the static member variable. This in effect provides a running log of how many **Predicate** objects are in existence at any time.

Re-run your code from the previous exercise with those changes, and record the output that is produced. Then, add a destructor to the **ConjunctiveClause** class, which iterates through the vector of pointers to **Predicate** objects and calls **delete** on each one. Re-run your code again, and compare the output to the output from the previous run. As the answer to this exercise, please summarize what was different between the two runs, and explain briefly why those differences occurred.

PART II: ENRICHMENT EXERCISE (optional, feel free to skip or to try variations that interest you)

7. Modify your **ConjunctiveClause** class so that instead of storing a vector of pointers to objects it instead stores reference counted smart pointers (e.g., of type **shared_ptr<Predicate>**). Comment out the declaration and the definition of the **ConjunctiveClause** class destructor, and re-run the code from the previous exercise with those changes. As the answer to this exercise please describe how the output from this run compared to the output from the previous runs, and explain briefly why you saw what you saw.