Problem Set 4

Due Friday, June 5, 3:15PM

In this problem set we'll explore some interesting applications of the language features we've covered over the past few weeks. As always, feel free to email me if you have any questions or comments.

This problem set is due on Friday, June 5 at 3:15PM, which according to finals scheduling is the latest possible time I'm allowed to have anything due. I understand that finals are approaching, so if you need any extra time send me an email and I'll be glad to work something out.

To submit the problem set, email any source files to htiek@cs.stanford.edu. Please make sure to include your name and SUNetID at the top of any files you submit. Then congratulate yourself – you've just completed the final problem set of the quarter!

Problem 0

Suppose that you are writing a program to play Hearts, as we did earlier in the quarter. You have a special struct `cardT`, which is defined as follows:

```
struct cardT
{
    suitT suit;
    valueT value;
};
```

Here, `suitT` and `valueT` are enumerated types for possible suits and values.

Using your newfound knowledge of operator overloading, implement an `operator <` function for `cardT` so that we can form an STL `set<cardT>`. You can compare two `cardT`s in any way you want, provided that the comparison follows all of the normal rules for < (i.e. if `one < two` then `!(two < one)` and any two nonequal cards are comparable by <).

Problem 1

In the next few problems, you'll get to see exactly how an STL algorithm is implemented behind the scenes. You may want to have Handout #27 on functors nearby before starting.

One of the STL algorithms we visited earlier in the quarter is `find_if`, which accepts three parameters – two iterators delineating a range and a predicate function – then returns an iterator to the first element in the range for which the predicate returns true. If the predicate returns false for all of the elements in the range, `find_if` returns the end of the iterator range as a sentinel. For example, if we want to get an iterator to the first even element in an STL `vector<int>`, we could do the following:

```
bool IsEven(int val)
{
    return val % 2 == 0;
}

vector<int>::iterator itr = find_if(myVector.begin(), myVector.end(), IsEven);
if(itr == myVector.end())
    cout << "No even elements found." << endl;
```

To see exactly how an STL algorithm like `find_if` is put together, we will rewrite this function from scratch. As a first step, let's consider writing a function which only works on iterators from an STL `vector<int>` and which only accepts genuine C++ functions as predicates. Here is one such implementation:

```
vector<int>::iterator FindIf(vector<int>::iterator start,
                             vector<int>::iterator end,
                             bool (predicate)(int))
{
    /* Continuously march start forward until we either find an element that
     * passes or hit the end of the range.
     */
    while(start != end)
    {
        /* If the current element passes the predicate, return it! */
        if(predicate(*start))
            return start;

        ++start;
    }

    /* No match.  Return end as a sentinel. */
    return end;
}
```

STL iterators use operator overloading for pretty much all of their core functionality. Identify three places in the code for `FindIf` that are calls to overloaded operators.

Problem 2

Using the techniques covered in Handout #27, templatize this function so that it can accept a functor as a predicate, not just a simple C++ function.

Problem 3

When writing functions which accepts functors as parameters, we templatize the function over the type of the functor. This technique works because C++ can infer the type of the object passed in as the final parameter and can deduce at compile-time what it means to "call" the object. Using similar reasoning, we can templatize the above `FindIf` function with respect to the type of iterator used to allow the caller to use `FindIf` on any type of iterator.

Modify the version of `FindIf` you just wrote so that it is templatized over both the type of predicate function and the type of iterator used. This shouldn't require that much of a code change – in fact, the only code that should change is the template declaration, the function return type, and the function parameter type.

Problem 4

Using the techniques you've just seen, write a function `ReplaceIf` which accepts as input a range of iterators, a predicate function (or functor), and a value, then updates the elements in the iterator range by replacing every element for which the predicate returns true with the specified value. Note that this is exactly the behavior of the STL `replace_if` algorithm. As an example of how `ReplaceIf` should work, consider the following code:

```
/* Create a vector of the first ten integers. */
const int NUM_INTS = 10;
vector<int> myVector;
for(int k = 0; k < NUM_INTS; ++k)
    myVector.push_back(k);

PrintVector(myVector); // Prints 0 1 2 3 4 5 6 7 8 9

/* Replace all even numbers with 0. */
ReplaceIf(myVector.begin(), myVector.end(), IsEven, 0);

PrintVector(myVector); // Prints 0 1 0 3 0 5 0 7 0 9
```

As a hint, your function should be templatized over three types – the type of the iterators, the type of the predicate, and the type of the value to replace elements with.

Problem 5

In Thursday's lecture we covered how C++ objects are laid out in memory, and in particular mentioned that virtual function calls are implemented by looking up which function to call in an object's vtable (for reference, the slides detailing object layout are available on the CS106L website as Lecture Code 16.0). If you'll notice, at the top of each vtable is a slot for metadata, which usually stores information about the runtime type of the object. For example, an object of type `BaseClass` would have metadata about the `BaseClass` type, while an object of type `DerivedClass` would have metadata about the `DerivedClass` type.

Knowing that this metadata is available at the top of every vtable, explain at a high level how the C++ `dynamic_cast` operator might be implemented. In particular, you should mention how `dynamic_cast` is able to determine whether the desired cast will fail.

Problem 6

The next few problems explores how virtual functions interact with destructors.

Compile and run the following code and report what the output is. What does this suggest about the order in which destructors are invoked in derived classes? How does this compare to the order in which constructors are invoked in derived classes?

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

class Base
{
public:
    Base() {};
    ~Base() { cout << "Destroying Base..." << endl; }
};

class Derived: public Base
{
public:
    Derived() {};
    ~Derived() { cout << "Destroying Derived..." << endl; }
};

int main()
{
    Derived d;
    return 0;
}
```

Problem 7

When calling a virtual function in a class constructor, the version of the function called is always the version for the class currently being constructed, rather than the overall object being constructed. For example, in Tuesday's lecture we noted how calling `vroom` in the `Engine` constructor always calls `Engine::vroom` rather than the `vroom` for the specific type of `Engine` being built (i.e. `CarEngine`, `GoldPlatedJetEngine`, etc.). This is also true of class destructors. Using your result from the previous problem about the order in which destructors are invoked, explain why this is the case.

Problem 8

How long did this problem set take you? How hard was it? Did the questions help you get a better understanding of the material? Any suggestions for next quarters' problem sets?