80

Spring 2014 EE380L Midterm: NAME: Cheng Fu (cf24258)

You get 10 points for correctly attaching your name to this exam. Questions 1, 4, and 5 are worth 20 points each. Questions 2 and 3 are similar and are worth 30 points if you do both of them (15 points each), or worth 20 points if you choose to do only one.

20

1. The C++ standard library includes a template metafunction called is_same. In the C++ standard, this metafunction takes two type arguments T1 and T2 and exports a constant boolean called "value" that will be either true (when T1 is the same as T2) or false (when T1 is not the same as T2). I want you to improve upon this library function by writing a variadic template metafunction called all_same. The all_same metafunction should accept an arbitrary number of type arguments and should export a constant Boolean called "value" which is true if and only if all the types are the same. So, for example:

    all_same<int, int, int, int, int>::value should be true
    all_same<int, double, int>::value should be false

You can ignore the possibility that all_same is used with fewer than two arguments, so, all_same<char>::value is undefined. You can use std::is_same<T1, T2> in your solution, but you cannot use any other functions or metafunctions from the standard library. If you use a function/metafunction (other than is_same) you must write that function/metafunction.

```
template <typename T...> ... >
struct all_same;

template <typename T, typenam R>
struct all_same<T,R> {
static const bool value = std::is_same<T, R>::value;
};

template <typename first, typename second, typename Tail...>
struct all_same<first, second, Tail...> {
static const bool value = std::is_same<first, second>::value &&
                          all_same<second, Tail...>::value;
};
```

15/15

2. I have a generic algorithm doit (declared below) that takes two function arguments of type T. The type T will typically be a type that is supported by std::iterator_traits<>. However, it is possible that T is a type that is not supported by std::iterator_traits. **Please declare a local variable named x** such that the type of x is:

iterator_traits<T>::value_type if this type is defined (i.e., if iterator traits works)
decltype(*b) otherwise

You cannot assume that decltype(*b) is the same as iterator_traits<T>::value_type. You can only declare x to be decltype(*b) if iterator_traits<T>::value_type does not exist (i.e., attempting to use iterator_traits<T>::value type is a type error). You may not use any functions from the standard library other than iterator_traits. If you need other classes, functions or metafunctions in your solution, you must write those classes, functions or metafunctions.

```
template <typename T>
void doit(T b, T e) {
      // declare x following instructions above
using xtype = decltype( gettype<T>(b) );
}   xtype x;
```

```
template <typename T,
typename ret = std::iterator_traits<T>::value_type>
ret gettype(T);

template <typename T, typename ret =
decltype(decltype(val<T>().operator*())>>
ret gettype(...);
```

Close enough.

I wish this syntax existed (decltype typ aliases inside of the function declaration)

Oh, I see....

13/15

3. The C++ I/O library can be extended to permit the output of any type. However, by default, new types are not printable. For example, the struct Foo {} is not printable since there is not operator defined for cout << Foo{}. Write a protected ostream wrapper (called pstream) that works as described in the example below. Please provide just the constructors and operators needed for the example. You will need a constructor that takes an ostream& reference, for example, but no copy constructors, destructors, etc. When an object of type T is printed with a pstream, the behavior should be:

Print the object if T is a type that can be printed using operator<<

Print the text "<unprintable>" otherwise

```
struct Foo {};
int main(void) {
    pstream pout{cout};

    pout << "Hello World"; // prints Hello World
    pout << 42; // prints 42
    pout << Foo{}; // prints <unprintable>
}
```

```
class pstream {
public:
  template (typename T, typename unused = decltype( declval<ostream>()<<
      declval <T>() > >
  ostream & operator << ( T & item ) {
      out << item;
      return out; }
  ~~template (typename T>~~
  ostream & operator<< ( ... ) {
      out << "<unprintable>";
      return out.
  }
protected:
  ostream & out;
public:
  pstream ( ostream & _out ) : out (_out) { }
}
```

no need to make this a template

-2

12/20

4. Strings in C++ can be concatenated using operator+. However, cascading more than two + operators in the same expression results in unnecessary memory allocations – there's one temporary created for each operator+ in the expression. I want you to create a FastString type and an operator+ defined for it that eliminates all but one allocation. Please limit yourself to supporting the main function defined below. Note that there's at least two constructors defined, but no assignment operator (one constructor takes a const char* argument). Please don't write an assignment operator or any other functions. In the code below there should be exactly one allocation. As a hint on how to solve this problem, I've provided the append function. I would not expect append to be used in your solution, but you'll probably want to use reserve and push_back in the same manner. Finally, you probably noticed that this question has a significant resemblance to Project 2, and that's intentional. I have to warn you, however, that there are some subtle differences with the way that the proxies need to store their subtrees, and the technique used in Project2 will not work.

```
string append(string s1, string s2) {
      string result{};
      result.reserve(s1.size() + s2.size());
      for (char c : s1) { result.push_back(c); }
      for (char c : s2) { result.push_back(c); }
      return result;
}

int main(void) {
      FastString str{FastString{"Hello"}
            + FastString{" "}
            +FastString{"World"}}; // creates str with one allocation

      cout << str; // uses inherited operator<< for std::string
}
```

assume operator + goes from left to right with no ( ) operator

no "(a + b + c)" case.

```
class FastString {
public:
      string *mystr;

      FastString () { mystr = nullptr; };

      FastString ( string &s) {mystr = &s};

      template <typename L, typename R>

      FastString (Proxy <L, R> prx ) {
```

Yuck

ADDITIONAL SPACE FOR THIS QUESTION ON NEXT PAGE

```
            mystr = & getstring (prx) ;
      }
```

ostream & operator << (ostream & out , (Fast String & fs1)

out << (*(fs1.mystr));

return out

}

operator << (ostream & out , Proxy (L, R) & prx)

out << getstring (prx);

}

template <typename T1, T2)

T2 is always Fast String

Proxy(T1,T2) & operator + (T1 & a, T2 & b) {

return Proxy (T1,T2) (a, b);

}

template <typename T1>

← *Proxy calalation/evaluation*

String ~~&~~ getstring (Proxy <T1, Fast String > prx) {

return append ((getstring (prx.left), *(prx.right.myser));

}

↑ but, this performs an allocation for each pair, i.e. one allocation for each + in the expression

String ~~&~~ getString (Proxy <Fast String , Fast String > prx) {

return append (*(prx.left.myser), *(prx.right.myser));

}

THIS PAGE LEFT BLANK (SPACE FOR QUESTION 4)

(cont. from page left)

```
template<typename L, typename R>
class Proxy {
    L left; int leftType;
    R right; int rightType;
public:
    Proxy(FastString &fs1, FastString &fs2): left(fs1), right(fs2)
    { leftType=0; rightType=0; }
    template<typename T1, T2>
    Proxy(Proxy<T1,T2>&prx, FastString &fs2): left(*(new L(prx))),
    right(fs2) { leftType=1, rightType=0; }
    template<typename T1, T2, T3, T4>
    Proxy(Proxy<T1,T2>&prx1, Proxy<T3,T4>&prx2):
        left(*(new L(prx1))), right(*(new R(prx2))) {
        leftType=1; rightType=1;
    }
    ~Proxy() { if(leftType) delete left;
               if(rightType) delete right;
    }
};
```

what's with leftType & rightType?

```
template (typename T, typename R)

void reverse_helper(T b, T e, R) {
    using val_type = decltype(*b);
    vector<val_type> v;
    T cur = b;
    while (cur != e) {
        v.push_back(*cur);
        ++cur
    }
    while (b != e) {
        *b = v.pop_back()
        ++b
    }
}
```

10

5. For this problem, I want you to write a generic algorithm following the conventions of the C++ standard library (STL). A large part of the question is demonstrating that you know the conventions of the C++ standard library, so I have to be vague about how the function is declared and how the parameters are passed to it. I want you to write a generic function that reverses the sequence that elements appear inside a container. The question is in two parts.

   a. There are fairly straightforward methods to reverse the elements in an array or any other container. In the first method, we allocate extra storage (e.g., a vector or a stack) and we push the elements onto the extra storage and then pop them off in the opposite order, writing the elements back into the original container as they are popped. I call this function reverse1. The other method involves swapping the first and last element in the container, then swapping the 2nd and second-to-last element and so forth, swapping elements from the ends until you reach the middle. I call this function reverse2. For part (a) write a function called reverse that uses the conventions of the C++ standard library. The function takes a container. If the container supports method 2, then you must call reverse2. Otherwise, you must call reverse1.

```
template (typename T)
void reverse ( T b, T e ) {
    using iter_cate = std::iterator_traits<T>::iterator_category;
    reverse_helper ( b, e, cate );
}

template (typename T)
void reverse_helper (T b, T e, std::random_access_tag ) {
    --e;
    while (b<e) {
        auto temp= *b;
        *b = *e;
        *e = temp
        ++b; --e
    }
}
```

Overloading
2 selections

category

0

b. I wrote reverse1 as shown below. However, my implementation is needlessly inefficient. Rewrite the implementation so that it is more efficient. HINT: each element from the original container should be in either the original container or in the temporary vector, but not both at the same time.

```
/* NOTE: this function does not fully comply with C++ standards */
template <typename T>
void reverse2(T beg, T end) {
      using value_type = decltype(*beg);
      std::vector<value_type> storage;
      for (T k = beg; k != end; ++k) {
            storage.push_back(*k);
      }
      for (T k = beg; k != end; ++k) {
            *k = storage.back(); // last element
            storage.pop_back();
      }
}
```

Idea is use half storage to store the s

No.