Fall 2005 EE 380L Final: **NAME:**_____

1. (7 pts) Both Java and C++ enforce static binding on any member function called from inside a constructor (even when calling a virtual function). Explain why. Write a simple program (the simpler the better) that demonstrates what would go wrong if dynamic binding were used.

```
class Base {
  virtual doit() { }
  Base() { doit(); }
};

class Derived : public Base {
  int *p;
  virtual doit() { *p = 42; }
  Derived() { p = new int; }
};
```

Static Binding must be used because Sub classes may not be fully constructed (Base classes are constructed first)

2. (12pts) Each of the following are required in order to preserve type safety. For each, write a statement (or two) that demonstrates how type safety might be violated if the feature were missing. Be sure to circle or somehow clearly indicate which variable reference might be bound to an object of the incorrect type.
   a. Bounds checking on all array accesses

```
T[] x = new T[10];
T y = x[11];
```

wrong type.

   b. Exception handling

Same as (a). We need to throw an exception since y cannot have a value.

   c. Automatic memory management (i.e., there can be no *free* or *delete* function).

3.

```
T p = new T();
delete p;
T2 q = new T2()
print (p)
```

← wrong type!

(15 pts) Assume now that I want to have a type safe language that is statically typed. There are no run-time type checks. For each of the following indicate if the language feature can be permitted (write "OK", or "BAD", a language feature is "BAD" if it can result in a violation of type safety, or if it requires a runtime type check).

OK a. Abstract types

BAD b. Data structures that hold generic objects (e.g., the data structure can hold two objects of arbitrary types)

OK c. Data structures that store different subtypes of some common base type (e.g., a data structure could hold both a Rectangle and a Triangle as long as they are both subtypes of Shape)

OK d. Subclasses may define methods (i.e., member functions) that are not present in the super class.

OK e. Automatic type conversion (i.e., the compiler uses constructors and/or user-defined type-conversion operators to implicitly convert objects on one type into another).

4. (6 pts) The C++ library does not use virtual functions because of the runtime overhead associated with dynamic method dispatch. However, it is possible to speculatively inline functions. Consider the C++ and the Java standard libraries' sort functions. The C++ sort is a template. The Java sort uses object oriented programming, where the argument to the function is an abstract type *List*. Since the sort function itself is recursive, it cannot be inlined. However, the functions that *sort* calls to access the values in the data structure could be inlined (speculatively in the case of the Java sort). What do you think? Do you think it's practical to offset or eliminate the runtime overhead of dynamic method dispatch in this case? Explain with one sentence (OK, two sentences if you need both) ((three sentences is pushing it)).

method to specialize the [illegible] is not possible when the type is abstract (runtime compilation can change this)

5. (6 pts) C++ has a seemingly arbitrary requirement where virtual functions receive dynamic binding only if the function is invoked using a pointer or reference to the actual object. So, for example,

```
void doit(const Base& x) { … }
void doit(Base x) { … }
```

can behave differently, even if the functions themselves are line-for-line identical. Java has no such requirement. What is the reason for this requirement? (it's not arbitrary at all, really), and why does Java not have the same requirement?

The problem is the amount of space allocated to hold the argument. In the case of Base& x, a pointer (ref) is allocated. This reference can point to either a base type or a subtype. In the case of Base (copy) There's not enough room for a subtype

(java doesn't need this since all args & locals are refences)

(15 pts) The string class is part of the standard C++ library. Change the string class as little as possible, and develop expression templates for an operator+ that concatenates two strings. Do not write the entire string class, just write those methods that need to be implemented differently because of your expression templates.

Your expression templates should result in foo() displaying identical behavior to bar(). (note the behavior of bar() should not be affected by your expression templates). You do not need to provide interoperability with char* (e.g., your expression templates don't need to worry about expressions like s = s + "hello world"). Assume both operands to operator+ are type string.

```
string foo(void) {
 string ret_value;
 ret_value = string1 + string2 + string3;
 return ret_value;
}

string bar(void) {
 string ret_value;
 ret_value.reserve(string1.size() + string2.size() +
string3.size());
 ret_value += string1;
 ret_value += string2;
 ret_value += string3;
}
```

7.

```
template<typename L, typename R>
class Op {
  int size() { return lhs.size() + rhs.size(); }
  Op(L l, R r) : lhs(l), rhs(r) {}
  L lhs; R rhs;
};
class String {
  template<typename R, typename L>
  String& op=(Op<L,R> op) {
      reserve(op.size);
      append(op);
  }
```

(12 pts) There are four reserved words (keywords) associated with the Java exceptions, but only three for C++. The "extra" keyword/construct in Java is **finally**. Recall that **finally** is used to create a "handler" that will always be run after the **try** block is completed.

a. Why is **finally** not needed in C++?

We have destructors

b. Simulate finally in the following example. Ensure that the following program prints "Hello World" exactly one time. You may use only one "cout" statement. If *doit* runs without throwing an exception, then "Hello World" must be printed only after doit completed. Rewrite the exq function so that it provides the required behavior.

```
void exq(void) {
    try {
        doit();
    }
}
```

8.

class Finally {
~Finally() { cout << "Hello World"; }
};

void exq(void) {
Finally x;
try {
doit();
}
}

(15 pts) Assume we wish to simulate Java-style iterators. Without changing any of the classes/types from the C++ standard library, write the *make_iterator* function and the Iterator<T> template (and any other classes you need) so that the following program works. You do not need to support any functionality of Java-style iterators except what's used in the program below. For this problem, I do not care about memory leaks, so leave destructors out of your solution. I do want the following program to work exactly (e.g. don't change "." into "->"). The *make_iterator* template must work with any data structure that conforms to the C++ library standard (not just vectors).

```
int main {
  vector<int> x = some_vector(); // don't care what the contents are
  Iterator<int> p = make_iterator(x);
  while(p.hasNext()) {
    cout << p.next();
  }
}
```

9.

```
template <typename T> class _Iterator {
    virtual T next() = 0;
    virtual bool hasnext() = 0;
};
template <typename T> class Iterator {
    _Iterator<T>* p;
    T next() { return p->next(); }
    bool hasNext() { return p->hasnext(); }
    Iterator(_Iterator<T> p) { this->p = p; }
};
template <typename It>
class Adapt : public _Iterator<
      typename iterator_traits<It>::value_type> {
    It p; It e;
    virtual hasnext() { return p != e; }
```

(12 pts) Assume I want to model the current Java standard data structure library. My program is in C++. Assume I'm using a java-like programming style with the following caveats.

- My program will allocate all objects on the heap, and access the objects through pointers.
- The most generic form of object reference is a "void*" pointer.

Create a type ArrayList<T> that will allow me to insert and remove objects of type T*. Other types of objects should not be permitted in the data structure. While the implementation of ArrayList<T> may have type casts, I do not want to have to use type casts when I use the public methods of ArrayList<T>. Your solution should preserve type safety. Static type checking is preferable to dynamic checking. If you use dynamic checking, then any problems should result in throwing a *NotType*<T> exception.

```
class _ArrayList {
public:
   void push_back(void* p);
   void pop_back(void);
   void* get(int k);
   void set(int k, void* x);
};
struct TypeError {};
template <typename T> struct NotType : TypeError {};
```

```
template <typename T>
class ArrayList : public _ ArrayList {
   void push_back ( T* p) {
      _ArrayList :: push_back (p);
   }
   T* get (int k) {
      return (T*) _ArrayList :: get(k);
   }
   void set (int k, T* x) {
      _ArrayList :: set (k, x);
   }
};
```