COMP 151H: Object Oriented Programming (Honor Track)
Spring Semester 2007
Final Exam
Instructor: Chi Keung Tang
Saturday, May 26, 2007
12:30 – 3:30pm
Room LG1027

This is a **CLOSED-BOOK-CLOSED-NOTES** exam consisting of eight (8) problems. Follow the instructions carefully. Please write legibly in the space provided. Keep the exam booklet stapled.

K E Y

Problem	Points	your score
1 MULTIPLE CHOICE	5	
2 OBJECT CONSTRUCTION AND DESTRUCTION	4	
3 INHERITANCE / ABSTRACT BASE CLASS	4	
4 STL ALGORITHMS	5	
5 FUNCTION OBJECTS	5	
6 TEMPLATE AND EXCEPTIONS	6	
7 STATIC CLASS	6	
8 CLASS TEMPLATES	5	
Total	40	

1 Multiple Choice

Select the answer by drawing a circle around the letter.

1. You can use private inheritance
   (a) to be able to inherit and thus access private variables in the bass class
   (b) **to provide a new interface, thus hiding the interface of the bass class**
   (c) to make the code more efficient as all references are local
   (d) all of the above
2. Template code can be precompiled like regular code can
   (a) True
   (b) **False**
3. A pure virtual function exists if and only if the pertinent class is an abstract base class
   (a) **True**
   (b) False
4. Inheritance usually represents
   (a) aggregation relationship
   (b) has-a relationship
   (c) **is-a relationship**
   (d) association relationship
5. Code reuse in object-oriented programming means
   (a) Copy/Paste code from a previous project/student.
   (b) Using code found on the web, with slight modifications.
   (c) Using code from a book, with slight modifications.
   (d) **Reusing full class definitions without any modification.**
6. Slicing in object-oriented programming is
   (a) what happens when you order pizza
   (b) **what happens when you assign an object of a derived type to a variable of the base class**
   (c) what happens when you assign an object of a base type to a variable of the derived class.
   (d) none of the above.
7. To get true polymorphism in C++ you need
   (a) object and virtual functions
   (b) **pointers to objects and virtual functions**
   (c) pointers to objects but not virtual functions
   (d) none of the above
8. A pure virtual function

(a) a method defined with body of $= 0$

(b) a way to get polymorphism in C++

(c) may make the code less efficient in run-time.

(d) **all of the above**

9. An abstract base class is a class defined

(a) as `class ClassName { ... } = 0;`

(b) **with at least one pure virtual function**

(c) with at least one member variable declared pure virtual

(d) all of the above

10. Protected member of a base class becomes _____ when inherited via public inheritance.

(a) Public

(b) **Protected**

(c) Private

(d) all of the above

2 Object construction and destruction

Write the output of the following programs on page 6 of the exam.

1.
```
#include <iostream>

class Base {
public:
  Base () { cout << "Base" << endl; }
  Base (int i) { cout << "Base" << i << endl; }
  ~Base () { cout << "Destruct Base" << endl; }
};

class Der : public Base {
public:
  Der () { cout << "Der" << endl; }
  Der (int i) : Base (i) { cout << "Der"  << i << endl; }
  ~Der () { cout << "Destruct Der" << endl; }
};

int main () {
  Base a;
  Der d(2);
  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class C {
public:
  C() : i(0) { cout << i << endl; }
  ~C() { cout << i << endl; }
  void iSet (int x) { i = x; }
private:
  int i;
};

int main() {
  C c1, c2;

  c1.iSet(5);

  {
    C c3;
    int x = 8;
    cout << x << endl;
  }
```

```
  return 0;

}
```

3.

```
#include <iostream>

class A {
public:
  int f() { return 1; }
  virtual int g() { return 2; }
};

class B : public A {
public:
  int f() { return 3; }
  virtual int g() { return 4; }
};

class C : public A {
public:
  virtual int g() { return 5; }
};

int main() {
  A *pa;
  A a;
  B b;
  C c;

  pa = &a; cout << pa->f() << endl; cout << pa->g() << endl;
  pa = &b; cout << pa->f() + pa->g() << endl;
  pa = &c; cout << pa->f() << endl; cout << pa->g() << endl;

  return 0;
}
```

4.

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

class A{
protected:
  int a;
public:
  A (int x=1) { a = x; }
  int f() { a += 2; return a; }
  virtual int g() { a += 1; return a; }
  int h() { f(); return a; }
  int j() { g(); return a; }
```

```
};

class B : public A {
private:
  int b;
public:
  B (int y=5) { b = y; }
  int f() { b += 10; return b; }
  int g() { a += 3; return a; }
};

int main()
{
  A obj1;
  B obj2;

  cout << obj1.h() << endl;
  cout << obj1.g() << endl;
  cout << obj2.h() << endl;
  cout << obj2.g() << endl;
}
```

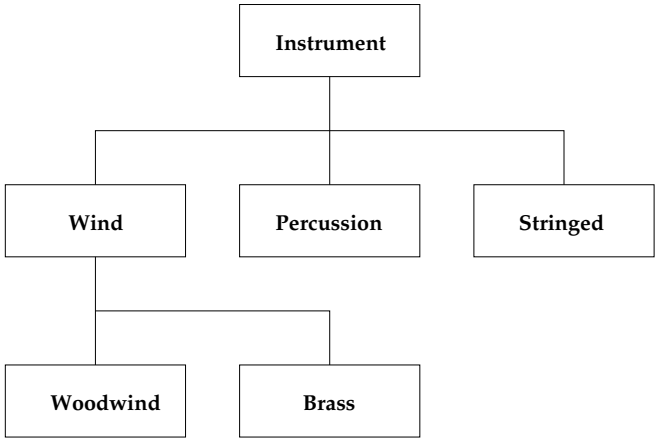
Key:

```
Base
Base2
Der2
Destruct Der
Destruct Base
Destruct Base

0
0
0
8
0
0
5

1
2
5
1
5

3
4
3
6
```

3 Inheritance / Abstract base class

Often in a design, you want the base class to present *only* an interface for its derived classes. That is, you do not want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by abstract base class. The following shows a class hierarchy where `Instrument` is an abstract base class.

In this particular example, there are no private data member and no concrete implementation for the member functions for `Instrument`.



Implement the necessary classes in `Instrument.h` so that the following code compiles and run. You should make the base class `Instrument` *abstract*. The sample output is also shown below. To make your coding easier and more readable, feel free to use the enumerated types `Note`, and two macros `note2text` and `int2note`, which are defined below.

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

enum Note { middleC, Csharp, Cflat };

#define note2text(note) \
((note == middleC) ? "middleC" : ((note == Csharp) ? "Csharp" : "Cflat"))

#define int2note(i) \
((i%3 == 0) ? middleC : ((i%3 == 1) ? Csharp : Cflat))

#include "Instrument.h"

int main() {
  Instrument* A[] = {
    new Wind,
    new Percussion,
    new Stringed,
    new Brass,
    new Woodwind
  };
  for (int i = 0; i < 5 ; i++) {
    A[i]->what();
    A[i]->play(int2note(i));
  }
  return 0;
}
```

Sample output:

```
Wind::play middleC
Percussion::play Csharp
Stringed::play Cflat
Brass::play middleC
Woodwind::play Csharp
```

Key:

```
class Instrument {
public:
  virtual void play (Note n) const = 0;
  virtual char* what () const = 0;
};

class Wind : public Instrument {
public:
  void play (Note n) const { cout << "Wind::play " << note2text(n) << endl; }
  char* what() const { return "Wind"; }
};

class Percussion : public Instrument {
public:
  void play (Note n) const { cout << "Percussion::play " << note2text(n) << endl; }
  char* what() const { return "Percussion"; }
};

class Stringed : public Instrument {
public:
  void play (Note n) const { cout << "Stringed::play " << note2text(n) << endl; }

class Brass : public Wind {
public:
  void play (Note n) const { cout << "Brass::play " << note2text(n) << endl; }
  char* what() const { return "Brass"; }
};

class Woodwind: public Wind {
public:
  void play (Note n) const { cout << "Woodwind::play " << note2text(n) << endl; }
  char* what() const { return "Woodwind"; }
};
```

4 STL algorithms

The following shows a standard C++ program that reads a list of integers, sort them and print them.

```
#include <stdlib.h>
#include <iostream.h>

// a and b point to integers.  cmp returns -1 if a is less than b,
// 0 if they are equal, and 1 if a is greater than b.
inline int cmp (const void *a, const void *b)
{
  int aa = *(int *)a;
  int bb = *(int *)b;
  return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

// Read a list of integers from stdin
// Sort (c library qsort)
// Print the list

int main (int argc, char *argv[])
{
  const int size = 1000;  // array of 1000 integers
  int array [size];
  int n = 0;
  // read an integer into the n+1 th element of array
  while (cin >> array[n++]);
  n--; // it got incremented once too many times

  qsort (array, n, sizeof(int), cmp);

  for (int i = 0; i < n; i++)
    cout << array[i] << "\n";

  return 0;
}
```

Re-implement the above program using *STL container, iterator, and algorithms*. Assume all operators have been appropriated overloaded and/or provided for, and all appropriate header files have been included. Although there are ways other than using STL to implement this simple function, full credits will only be given for solutions that directly employ STL features to solve the problem.
Some of the function prototypes and description that you *may* find them useful are as follows:

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

Usage: Sort sorts the elements in [first, last) into ascending order, meaning
that if i and j are any two valid iterators in [first, last) such that
i precedes j, then *j is not less than *i. This algorithm compares
objects using operator<.

template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last,
                     OutputIterator result);
```

```
Usage:  Copy copies elements from the range [first, last) to the range
[result, result + (last - first)). That is, it performs the assignments
*result = *first, *(result + 1) = *(first + 1), and so on.
```

Key 1:

```
#include <string.h>
#include <algo.h>
#include <vector.h>
#include <stdlib.h>
#include <iostream.h>

main ()
{
  vector<int> v;  // create an empty vector of integers
  int input;
  while (cin >> input)    // while not end of file
    v.push_back (input);  // append to vector

  sort(v.begin(), v.end());

  int n = v.size();
  for (int i = 0; i < n; i++)
    cout << v[i] << "\n";
}
```

Key 2:

```
#include <string.h>
#include <algo.h>
#include <vector.h>
#include <stdlib.h>
#include <iostream.h>

main ()
{
  vector<int> v;
  istream_iterator<int,ptrdiff_t> start (cin);
  istream_iterator<int,ptrdiff_t> end;
  back_insert_iterator<vector<int> > dest (v);

// note: for the above, if ptrdiff_t is skipped, it is still right for certain compilers.
// Iterators are sometimes parameterized on two types; the second is a
// distance type, which is really for compatibility on operating systems
// with multiple memory models.

  copy (start, end, dest);
  sort(v.begin(), v.end());
  copy (v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
}
```

5 Function objects

1. Fill a `vector` of 100 elements with random numbers. In your answer, you are required to use STL container class `vector`, STL algorithm `generate`, and the `rand` function. The prototypes of `generate` and `rand` are as follows:

```
int rand(void);

template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

   Key:

```
  vector<int> V(100);
  generate(V.begin(), V.end(), rand);
```

2. Define a class that allows for a function object to be instantiated, so that the following code, which sorts a vector of double by magnitude, i.e. ignoring the elements' signs, can compile and run. Assume all appropriate header files have been included.

```
    vector<double> V;
    ...
    sort(V.begin(), V.end(), less_mag());
```

   Key:

```
class less_mag {
public:
  bool operator()(double x, double y) { return fabs(x) < fabs(y); }
};
```

3. Define a class that allows for a function object to be instantiated, so that the following code can compile and run. Assume all appropriate header files have been included.

```
    vector<double> V;
    ...
    adder result = for_each(V.begin(), V.end(), adder()); [3]
    cout << "The sum is " << result.sum << endl;
```

   Key:

```
class adder {
public:
  adder() : sum(0) {}
  void operator()(double x) { sum += x; }
private:
  double sum;
};
```

6 Templates and Exception

Write a generic function, `Find`, that receives an array `arr`, size of the array `size`, a search key `key`, and an integer parameter passed by reference `location`.

The function should perform linear search on `arr` looking for `key`, and place the location of the item (if found) in `location`. The function should return a `bool` value to indicate whether the item was found or not. You can assume all comparison and subscript operators have been properly overloaded.

Here is some sample code on how this function would be used. The function should generate an exception object if the `size` parameter is less than or equal to zero. The program should recover after handling an exception.

You can merely output an error message if an exception is thrown.

```
int main() {
  int x, index, numbers[5] = { 10, 20, 30, 45, 66 };
  const char* names[5] = { "Bruce", "Clark", "Diana", "Dick", "Peter"};

  cout << "Enter a number " ;
  cin >> x;
  if (Find(numbers, -4, x, index))
    cout << "Found " << x << " at " << index << endl;
  else
    cout << "Item " << x << " not in array" << endl;

  if (Find(names, 5, "Diana", index))
    cout << "Found Diana at " << index << endl;
  else
    cout << "Item " << names << " not found" << endl;

  return 0;
}
```

Key:

```
class InvalidArraySize {
public:
   InvalidArraySize() : msg("error") { }
   const char* what() { return msg; }
private:
   const char* msg;
};
template <class T>
bool Find (T arr[], int size, T key, int& index)
{
  try { if (size <= 0) throw InvalidArraySize(); }
  catch (InvalidArraySize ex) {
    cerr << ex.what() << endl;
  }

  for (int i = 0; i < size; i++)
    if (arr[i] == key) {
       index = i;
       return true;
    }
  return false;
}
```

7 Static class

Define a *short* class called `Singleton` that cannot be instantiated more than once. That is, during run-time, a program that uses the `Singleton` class can have either zero or one instance of `Singleton`. If you try to create a second instance, you would get the first instance already created.

The content of the class is irrelevant for the problem. The key is that there could be no way to create more than on instance of the class.

Requirements (Hints): The solution involves three things 1) Make the default constructor private; 2) declare a friend function that is used to create the instance of the class, or return the instance already created; 3) declare something `static` to keep track of the unique instance itself.
Key:

```
class Singleton {
private:
  static Singleton* instance;
  Singleton() {}       // must have constructor private so no (default)
                       // public constructor function to construct instance
public:
  friend Singleton* NewSingleton();
};

Singleton* Singleton::instance = NULL;

Singleton* NewSingleton() {
  if (Singleton::instance == NULL)
    Singleton::instance = new Singleton();
  return Singleton::instance;              // return the unique instance
}
```

8 Class templates

Define a class to hold a pair of values. Use a template for the class (named `Pair`) so that the class can hold two values (a pair) of the same type, the type being the parameter in the template.

The class should have a constructor with two arguments of the same type (the parameter in the template) to initialize the member variables. Define a read function with an `istream&` as argument, that reads a pair of the format: (X,X) where X represents a datum of the template parameter. The class should also have a write function with an `ostream&` as an argument, that will print pairs in the format (X,X). Here is some sample code on how this class would be used.

```
int main() {
  Pair<double> q(100.1, 34.5);
  q.write(cout);
  cout << "Enter values " ;
  q.read(cin);
  q.write(cout);

  Pair<int> p(10, 20);
  p.write(cout);
  cout << "Enter values ";
  p.read(cin);
  p.write(cout);
  return 0;
}
```

Sample run:

```
(100.1, 34.5)
Enter values (3.4, 4.5)
(3.4, 4.5)
(10, 20)
Enter values (2, 3)
(2, 3)
```

Key:

```
template <class T> class Pair {
private:
  T first;
  T second;
public:
  Pair (T a1, T a2) : first(a1), second(a2) {}

  void read (istream& in) {
    char ch;
    in >> ch;         // read (
    in >> first;
    in >> ch;         // read ,
    in >> second;
    in >> ch;         // read )
  }
  void write (ostream& out) {
    cout << "(" << first << ", " << second << ")" << endl;
  }
};
```