Final Exam for Programming Methodologies (Fall of 2007)
Instructor: Kyuseok Shim

1. What is the definition of abstraction? What are the forms of abstraction? [5 points]

   SOLUTION: Abstraction of a process or object consists of its high-level and essential properties that are exposed the remaining low-level details that are hidden. The forms of abstraction in programming languages are procedural abstraction, data abstraction (type abstraction) and. object abstraction

2. Answer the following:

   - What is a virtual member function in C++? [5 pts.]

     SOLUTION: It is a member function whose calls are dynamically dispatched. That is, given the declaration of the form

     ```
     class C {
        ... virtual .. f(..) .. ...
     }
     ```

     and a call of the form

     ```
     ... x.f(...) ...
     ```

     where the variable x is declared to be of type C, the version of f that is executed depends on the actual type of the object that x refers to, not necessarily the declared type of x.

   - What happens if you leave the `virtual` keyword off the declaration of a member function? [5 pts.]

     SOLUTION: If you leave the keyword virtual off, then no dynamic dispatching of the member function occurs. In the example above, no matter what the actual type of the object that x refers to is, the version of f that is called is the one defined in the class C.

3. What is the output of the following code? [10 points]

```
> (define complex (lambda (r i)
      (define + (lambda (a)
         (complex (+ r (a ' real_part)) (+ i (a ' imaginary_part)))))
```

```
        (define - (lambda (a)   ...   ))
           ...
        (define this_object (lambda (func)
           (cond ((eq? func ' real_part) r)
               ((eq? func ' imaginary_part) i)
               ((eq? func ' + +))
                 ...      )))))
     this_object))
 > (define object1 (complex 7.6 3))
 > (define object2 (complex 5 1.1))
 > (object1 ' real_part)
```

SOLUTION See the lecture slides.

4. Turn the C++ definition

```
int sum(int a, int b, int c)
{ return a+b+c;
}
```

into a function template that can be used to work on any type that supports +, instead of just int. Be sure that the template will work for objects that implement the + operator, so that dynamic dispatching of + will occur where possible. [5 points]

SOLUTION

```
template<class T>
T sum(T &a, T &b, T &c)
{ return a+b+c;
}
```

5. What is the output of the C++ code below? [5 points]

```
class A { public: int f(int x) { cout << x << " "; }};
class B: public A { public: int f(int y) { A::f(y+1); }};
void g(A a, B b) { a.f(3); b.f(3); }
int main() { B p; B q; g(p,q); }
```

SOLUTION: 3 4

6. Given the C++ code

```
class A {
public:
  A() { a = new int[3]; for(int i=0; i<3; i++) a[i] = i;}
  ~A() { delete[] a; }
private:
  int *a;
};

void init(A &x) //sets elements of x.a to to 0,1,2
{ A y; //fresh object
  x = y;
}

int main()
{ A p;
  ...
  init(p);
  ...
}
```

you might notice that as the program runs following the call `init(p)`, the values stored in p.a start changing unpredictably. What might be happening? Fix this problem. [10 points]

SOLUTION: To fix it, the copy assignment operator should redefined in A so that the elements of the array are copied, not the array pointer.

```
class A {
public:
        ...
        A &operator=(const A &other)
        { for(i=0; i<3;i++) a[i] = other.a[i];
        }
        ...
}
```

7. What is the output of the following program: [10 points]

```
 include <iostream.h>
 class blah {
   public:
      static int a;
      int b;

      blah(int x) {
         b=x;
         a=b+1;
      }
};

int blah::a = 0;

int main(void) {
  blah b1(5);
  blah b2(12);
  cout << "b1.a is " << b1.a << endl;
  cout << "b1.b is " << b1.b << endl;
  cout << "b2.a is " << b2.a << endl;
  cout << "b2.b is " << b2.b << endl;
}
```

SOLUTION: b1.a is 13. b1.b is 5. b2.a is 13. b2.b is 12.

8. Consider the following declaration

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

 class CMotor {
    public:
        CMotor ();
        CMotor ( const string &id );
        string get_ID () const {}
    protected:
        void set_ID (const string &s) {}
    private:
```

```
        string mID;
};

class CElectricMotor: private CMotor {
   public:
        CElectricMotor (const string &id, double volts): CMotor () {
            m_nVoltage = volts;                  //A
            set_ID(id);                          //B
            mID = id;                            //C
        }
   private:
        double m_nVoltage;
};

int main ()
{
   CMotor M;
   M.set_ID ("12345");                           //D
   CElectricMotor E ("test", 0.5);
   E.get_ID ();                                  //E
}
```

Which of the statements (A,B,C,D) will produce errors? And why? [10 points]

SOLUTION: C: Because mID of CMotor is private, CElectricMotor cannot access.

D: Since set_ID of CMotor is protected, we cannot call outside.

E: CElectricMotor inherits CMotor as private, the access to get_ID becomes private. Thus, we cannot get_ID.

9. Consider the following declarations

```
class Employee {
public:
  void getName () {}                      //A
  virtual void CalcPay () {}              //B
};
```

```
Class SalariedEmployee: public Employee {
public:
  void getName () {}                    //C
  virtual void CalcPay () {}            //D
};

void foo (Employee worker) {
  worker.CalcPay ();
}

int main ()
{
   Employee *p0 = new Employee;
   Employee *p1 = new SalariedEmployee;
   p0->getName ();  //a
   p1->getName ();  //b
   p0->CalcPay ();  //c
   p1->CalcPay ();  //d
   foo(*p0);  //e
   foo (*p1);  //f
}
```

Which functions are called by a,..., f? And explain the difference. [10 points]

SOLUTION: a: A is called.

b: A is called (getName is not virtual).

c: C is called.

d: D is called (CalcPay is declared as virtual).

e: C is called.

f: C is called (foo receives Employee as a value not as a pointer).

10. Consider the following declarations

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

class Complex
```

```
{
 public:
    Complex();
    ~Complex();
    Complex( float re, float im );
    /* (A) '+' operation overriding declaration (member function) */
    /* (B) '*' operation overriding declaration (friend function) */
    void print ();

 private:
    float real;
    float imag;
};

Complex::Complex() {
  real = 0.0f; imag = 0.0f;
}
Complex::~Complex() {}

void Complex::print() {
  cout << real << " + " << imag << "i" << endl;
}

/* (C) '+' operation overriding implementation (member function) */

/* (D) '*' operation overriding implementation (friend function) */

int main( int argv, char** argc )
{
  Complex a (4, 2);
  Complex b (5, 4);

  Complex ans;
  ans = a + b;
  ans.print ();
  ans = a * b;
  ans.print ();
```

```
   return 0;
 }
```

Complete operation overriding functions by member function for + and by friend function for *. Provide the expected result message. [10 points]

SOLUTION:

```
(A): Complex operator+ (Complex &a);
(B): friend Complex operator* (Complex &a, Complex &b);
(C):
Complex Complex::operator+ (Complex &a) {
   Complex b (real + a.real, imag + a.imag);
   return b;
}
(D):
Complex operator* (Complex &a, Complex &b) {
    Complex c (a.real + b.real, a.imag + b.imag);
    return c;
}
```

11. We want to implement the operator overloading with the following code.

```
       class IntList {
         private:
            int* array;
            int size;
         public:
            IntList() {}
            IntList(int n, int* list);
              {...} // copy the values of the list to array
(A)         IntList operator-(const IntList& l);
              {...} // Allow to compute IntList - IntList
(B)         int operator[](int n); // Return the n-th value of the array
            { return array[n]; }
(C)         friend IntList operator+(const IntList&, const IntList&);
              // Allow to compute IntList + IntList
(D)         friend void operator=(IntList& a, const IntList& b);
```

```
            // a.array[i]=b.array[i]
(i)         _______________; // prefix operator++
(ii)        _______________; // postfix operator++
      };
      // Implement operator overloading for + and =
      ...
      int main() {
         int list1[5] = {0,1,2,3,4};
         int list2[3] = {10,11,12};
         IntList a(5,list1);
         IntList b(3,list2);
         ...
         IntList c = b + (a++);    // c.array[i]=b.array[i]+(a.array[i]++)
         c[0] = b[1];  // Copy
         return 0;
      }
```

- Let us know whether the statements (A)–(D) have errors. If there is an error, explain. [4 points]

  SOLUTION:

  (A) : O

  (B) : X - 반환 형태가 int&가 되어야 한다. int형으로 반환할 경우 c[0] = b[1]과 같은 작업을 수행할 수 없다. 왜냐하면 주어진 형태로 b[1]의 값을 읽어오는 작업은 할 수 있으나, c[0]의 위치에 b[1]을 카피하는 작업은 할 수 없다. c[0]의 위치가 아닌 값만을 반환받았기 때문이다. int&는 alias로 c.array[0]에 직접 접근하게 해준다.

  `>> int& operator[](int); or int& operator[](int xxx);`

  (C) : O

  (D) : X - '='는 friend를 이용해서 정의할 수 없다. C++문법임

  `>>void operator=(const IntList&); or void operator=(IntList&); or void operator=(const IntList& xxx); or void operator=(IntList& xxx);`

- We want to implement ++ operators in (i) and (ii) without using `friend`. ++ operator should increase every value of the array by 1. Provide the declaration and fill the body. [6 points]

  SOLUTION:

```
IntList operator++();   // prefix
IntList operator++(int); // postfix

IntList IntList::operator++() {
    for ( int i = 0; i < size; i++ ) array[i]++;
    return *this;
}

IntList IntList::operator++(int) {
    IntList res;
    res = *this;
    for ( int i = 0; i < size ; i++ ) array[i]++;
    return res;
}
```