COMP152H Object Oriented Programming and Data Structures
Spring Semseter 2010
Midterm Exam
March 18, 2010, 3:00-4:20pm in Room 3598
Instructor: Chi Keung Tang

This is a **CLOSED-BOOK-CLOSED-NOTES** exam consisting of five (5) questions. Write your answer in the answer booklet provided.

---

1. **OO concepts** (5 points)
   (a) What is *polymorphism* in object-oriented programming?
   (b) Express in *less than 30 words* how the following concepts are related: dynamic polymorphism, dynamic binding, virtual functions, overriding, static polymorphism, template.

2. **Order of Construction and Destruction** (10 points)

   What is the output of the following program?

```
#include <iostream>
using namespace std;
class Watch {
public:
   Watch() { cout << "W" << endl;}
   ~Watch() { cout << "~W" << endl;}
};

class Father {
public:
   Father() { cout << "F" << endl;}
   ~Father() { cout << "~F" << endl;}
private:
   Watch w;
};

class Son : public Father {
public:
   Son() { cout << "S" << endl;}
   Son(const Watch& iw) : w(iw) { cout << "Sw" << endl; };
   ~Son() { cout << "~S" << endl;}
private:
   Watch w;
};

int main ()
{
  Watch watch;
  cout << "son 1" << endl; Son s1;
  cout << "son 2" << endl; Son s2(watch);
  return 0;
}
```

3. **Dynamic Polymorphism** (10 points)

   Code up `square.h`, `circle.h`, and `polygon.h` in order the following program can compile and run. The `Polygon` class should be made into an abstract base class and contain no data members. The output is also shown.

```
/** file: area.c **/
#include <stdio.h>
#include "polygon.h"
#include "circle.h"
#include "square.h"

int main() {
  Polygon* p_array[2];
  Circle c(3.0);  // radius = 3
  Square s(4.0);  // width = 4
  p_array[0] = &c;
  p_array[1] = &s;

  double sum_area = 0;
  for (int i=0;i<2;i++) {
     p_array[i]->print() ;
     sum_area += p_array[i]->area();
  }
  printf("Total area = %f\n", sum_area);
  return 0;
}
```

   The output is shown below:

```
Circle(3.000000)
square(4.000000)
Total area = 44.274310
```

4. **Template Container Class** (10 points)

   Complete the following template class definition. Adopt the array as the main data structure in the implementation. The output of the `main` program is also shown below.

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

```
template <class T>
class Stack {
public:
  Stack(int max_size = 10);   // to be completed
  bool is_empty () const { return (size == 0); }
  bool is_full () const { return (size == MAX_SIZE); }
  bool push (const T& item);  // to be completed
  T pop ();                   // to be completed
private:
  const int MAX_SIZE;
  T *array;
  int size;
};

int main ()
{
  Stack<int> my_stack(3);
  cout << "pushing 1...\n"; my_stack.push(1);
  cout << "pushing 2...\n"; my_stack.push(2);
  cout << "pushing 3...\n"; my_stack.push(3);
  cout << "pushing 4...\n"; my_stack.push(4);

  cout << "popping " <<  my_stack.pop() << endl;
  cout << "popping " <<  my_stack.pop() << endl;
  cout << "popping " <<  my_stack.pop() << endl;
  cout << "popping " <<  my_stack.pop() << endl;
  cout << "value returned by pop() when stack is underflow is arbitrary!\n";
  return 0;
}
```

The output is:

```
pushing 1...
pushing 2...
pushing 3...
pushing 4...
stack overflow!
popping 3
popping 2
popping 1
popping stack underflow!
1
value returned by pop() when stack is underflow is arbitrary!
```

5. **Iterator** (5 points)

   The following uses `vector` and `list` containers. Complete the rest of the program according to the instruction in the comments.

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

class Student {
public:
   // ... various functions to perform the required operations
private:
   string name, ID;
   int mark;
};

int main()
{
    list<Student> students;
    // Read from data base
    while (more_students())
    {
       Student temp;
       temp.read();
       students.push_back (temp);
    }
    // Now print the students that failed - of course, the particular Student
    // object should provide a member-function (say, passed()) that will
    // determine that. Also, assume that class Student provides the overloaded
    // stream insertion operator <<

//---- beginning of your answer one ----
   list<Student>::iterator i;

   // YOUR CODE SEGMENT ONE HERE

//---- end of your answer one ----

   // Now remove the failed students from students list by making use of the
   // stl algorithm erase. The function prototype is
   // iterator erase (iterator loc)


//---- beginning of your answer two ----
   i = students.begin();

   while (i != students.end())
   {
     // YOUR CODE SEGMENT ONE HERE
   }
//---- end of your answer two ----
   // ...
   return 0;
}
```

SOLUTIONS TO MIDTERM

---

1. (a) Polymorphism in object-oriented programming means that we can work with objects without knowing their precise type at compile time; in C++, the same function can respond differently depending on the actual object being processed.

   (b) In dynamic polymorphism, dynamic binding is necessary, which is implemented using virtual functions in an inheritance setting to achieve overriding. To achieve static polymorphism, template is used in C++. (29 words)

2.
```
W
son 1
W
F
W
S
son 2
W
F
Sw
~S
~W
~F
~W
~S
~W
~F
~W
~W
```

3.
```
#ifndef POLYGON_H
#define POLYGON_H

class Polygon {
  public:
    virtual double area () = 0;
    virtual void print() = 0;
};
#endif
//------------------circle.h------------------------------------
#ifndef CIRCLE_H
#define CIRCLE_H
```

```
#include "polygon.h"

class Circle : public Polygon {
  double radius;
public:
  Circle (double in_radius) { radius = in_radius; }
  virtual double area () { return 3.14159*radius*radius; }
  virtual void print () { printf ("Circle(%f)\n",radius);}
};
#endif
//--------------------square.h---------------------------------
#ifndef SQUARE_H
#define SQUARE_H

#include "polygon.h"

#include <stdio.h>

class Square : public Polygon {
  double width;
public:
  Square (double in_width) { width = in_width;}
  virtual double area () { return width*width; }
  virtual void print() { printf ("square(%f)\n", width); }
};
#endif
```

4.

```
template<class T>
Stack<T>::Stack(int max_size) : MAX_SIZE (max_size), size(0)
{
  array = new T [MAX_SIZE];
}

template<class T>
bool Stack<T>::push (const T& item)
{
  if (!is_full()) { array[size] = item; size++; return true; }
  else {
    cerr << "stack overflow!" << endl;
    return false;
  }
}

template<class T>
T Stack<T>::pop()
{
  T tmp;
```

```
    if (!is_empty()) { size--; tmp = array[size]; }
    else {
      cerr << "stack underflow!" << endl;
    }
    return tmp;
  }
```

5.

```
  Solution 1:

      // Now print the students that failed (mark < 60%) - of
      // course, the particular Student object should provide a
      // member-function (say, passed()) that will determine that

     list<Student>::iterator i;
     for (i = students.begin(); i != students.end(); ++i)
     {
        if (! i->passed())    // iterators also provide operator ->
        {
           cout << "The student " << *i << " failed." << endl;
            // provided that class Student provides the overloaded
            // stream insertion operator <<
        }
     }

  Solution 2:

     // Now remove the failed students (of course, this could have
     // been done in the previous loop)

     i = students.begin()
     while (i != students.end())
     {
        if (! i->passed())
        {
           i = students.erase (i);
        }
        else
        {
           ++i;
        }
     }
```