COMP 2012 Midterm Exam - Spring Semester 2015 - HKUST

Date: March 28, 2015 (Saturday)
Time Allowed: 2 hours, 2:15 – 4:15 pm
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **6** questions on **15** pages (including this cover page).
3. Write your answers in the space provided in black/blue ink. *NO pencil please.*
4. All programming codes in your answers must be written in ANSI C++.
5. Use *only* the C++ language features and constructs covered in the course so far.
6. For programming questions, you are **NOT** allowed to define additional helper functions or structures, nor global variables unless otherwise stated. You also **cannot** use any library functions not mentioned in the questions.

Student Name	
Student ID	
Email Address	
Lecture & Lab Section	

Problem	**Score**
1. True-false Questions	/ 15
2. Class Basics	/ 12
3. Template and Operator Overloading	/ 10
4. Abstract Base Class	/ 24
5. Construction and Destruction	/ 23
6. Array Operator Overloading	/ 16
Total	/ 100

Problem 1 [15 points] True or false

Indicate whether the following statements are *true* or *false* by circling **T** or **F**. You get 1.5 point for each correct answer, −0.5 for each wrong answer, and 0.0 if you do not answer.

T **F** (a) Base-class constructors are not inherited by derived classes.

T **F** (b) A *has-a* relationship is implemented via inheritance.

T **F** (c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.

T **F** (d) Inheritance encourages the reuse of proven high-quality software.

T **F** (e) Only existing operators can be overloaded.

T **F** (f) A function template can be overloaded by another function template with the same function name.

T **F** (g) Template parameter names among template definitions must be unique.

T **F** (h) All `virtual` functions in an abstract base class must be declared as pure `virtual` functions.

T **F** (i) Referring to a derived-class object with a base-class handle (e.g. pointer, referece) is dangerous.

T **F** (j) A class is made abstract by declaring that class `virtual`.

Q1. Solution

(a) True.

(b) False. A *has-a* relationship is implemented via composition. An *is-a* relationship is implemented via inheritance.

(c) False. This is an example of a *has-a* relationship.

(d) True.

(e) True.

(f) True.

(g) False. Template parameter names among function templates need not be unique.

(h) False. An abstract base class can include virtual functions with implementations.

(i) False. Referring to a base-class object with a derived-class handle is dangerous.

(j) False. Classes are never declared `virtual`. Rather, a class is amade abstract by including at least one pure virtual function in the class.

Problem 2 [12 points] Classes

Find the error(s) in each of the following and explain briefly how to correct it (them). Your answers should be short and fit in the space provided.

(a) The following prototype is declared in the definition of class `Time`:

```
void ~Time(int);
```

Answer:

(b) The following is a partial definition of class `Time`:

```
class Time
{
  public:
  // function prototypes
  private:
    int hour = 0; int minute = 0; int second = 0;
};
```

Answer:

(c) The following prototype is declared in the definition of class `Employee`:

```
int Employee(string, string);
```

Answer:

Q2. Solution

- For each part: 2 points for pointing out the Errors, 2 points for the Correction.

(a) *Error*: Destructors are not allowed to return value (or even specify a return type) or take arguments.

Correction: Remove the return type `void` and the parameter `int` from the declaration.

(b) *Error*: Members cannot be explicitly initialized in class definition.

Correction: Remove the explicit initialization from the class definition and initialize data members in a constructor.

(c) *Error*: Constructors are not allowed to return values. *Correction*: Remove the return type `int` from the declaration.

Problem 3 [10 points] Template and Operator Overloading

(a) Turn the following C++ function definition that only works for `int` *values*

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

into a function template that can be used to work on *objects* of any type that supports `operator+`. Be sure that the template will work even for *objects* that overload the `operator+`, so that dynamic binding of objects will work successfully wherever possible.

Answer:

(b) Define a simple C++ template for the class `Container`, parameterized by a single type, such that the following code will work:

```
Container<int> s1, s2;
Container<int> s3 = sum(s1, s2, s2);
```

Make the `Container` class template as simple as possible. The following incomplete skeleton code is given as a hint.

```
class Container
{
  public:
    Container( ) { std::cin ≫ value; }

    // Other member functions

  private:
    T value;
};
```

Answer:

Q3. Solution

- Grade distribution: (a) 4 (b) 6
- Part (a)
  - 1 point for only correct arguments;
  - must accept (const) T reference as parameter to allow dynamic binding.
  - 3 more points for complete correctness
- Part (b)
  - no points for any additional functions
  - -2 points if any additional things will make it incorrect

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

/* Part (a) */
template<class T>
T sum (const T& a, const T& b, const T& c) { return a + b + c; }
/* Acceptable too:
T sum (T& a, T& b, T& c) { return a + b + c; }
*/

/* Part (b) */
template<class T>
class Container
{
  public:
    Container( ) { std::cin >> value; }

    // Only the operator+ member function is really needed
    Container operator+(Container& m) const                        // 2 points
    {
        Container r(*this);                                        // 2 points
        r.value += m.value;                                        // 1 points
        return r;                                                  // 1 points
    }

    // Added here for testing; no points
    void print( ) const { std::cout << value << endl; }

  private:
    T value;
};

// Added here for testing; no points
int main( )
{
    Container<int> s1, s2;
    Container<int> s3 = sum(s1, s2, s2);
    s3.print( );
}
```

Problem 4 [24 points] Abstract Base Class and Down Casting

Complete the class definitions of Bird, Seagull, Parrot, and Parrot_Cockatoo which are assumed to all reside on a file called "bird.h" so that the program ("bird.cpp") in the next page will compile and run successfully with no errors, and produces the following outputs:

```
Seagull is not a pet
Parrot is a pet
Parrot can talk
Cockatoo is a pet
Cockatoo talks better
```

Your solution must satisfy the following requirements:

- Implement only those member functions necessary to produce the above outputs.
- Implement all the member function as inline functions within the class definitions; the correct solution is very short.
- The `Bird` class must be an abstract base class.
- Only birds of the `Parrot` family can talk. Calling the `talk( )` function (as in line #26 which is commented out) by other kinds of birds will result in compilation errors.
- There should be **NO** data members in the definitions of the 4 classes. Thus, it also means that **NO** user-supplied constructors and destructors are required and they are **NOT** allowed.
- No "dummy functions" (e.g. { } or { `cout << "";` }) are allowed.

Hint: The object returned by the function typeid( ) has a member function name( ) to retrieve a character string that contains the type name of the object together with other possible characters depending on the compiler.

```
#include <iostream>                                             /* File: bird.cpp */
#include <string>
using namespace std;
#include ”bird.h”

// Check if s2 is a sub-string of s1. Return true if it is, otherwise false
bool is_substring(const string& s1, const char* s2)
{
    return (s1.find(s2) != string::npos);
}

int main( )
{
    Seagull s;
    Parrot p;
    Parrot_Cockatoo c;

    Bird *b[ ] = { &s, &p, &c };

    for (int i = 0; i < sizeof(b)/sizeof(b[0]); i++)
    {
        cout ≪ b[i]→name( ) ≪ " is "
             ≪ ((b[i]→is_pet( )) ? "" : "not ") ≪ "a pet\n";

        // The following line, if not commented out, will cause compilation
        // error because only birds of the Parrot family talk.
        // b[i]->talk( ); // Line #26

        string bird_name = typeid(*b[i]).name( );               // Gets class name
        if (is_substring(bird_name, "Parrot"))
            dynamic_cast<Parrot*>(b[i])→talk( );
    }

    return 0;
}
```

Answer: /* File: bird.h */

Q4. Solution

- grade distribution: 5, 5, 7, 7.
- name( ) and is_pet( ) must be pure virtual functions in class bird; -1 point for each of them if they are not.
- talk( ) must not be a member function in bird; otherwise, -1 point.
- talk( ) must be a virtual function in derived class parrot, but it should not be a pure virtual function; otherwise, -1 point.
- all member functions should be const, otherwise -0.5 point for each one.
- the keywod "virtual" is optional for virtual functions in a derived class.

```
/* File: bird.h */
class Bird                                                              // 1 point
{
  public:
    virtual string name( ) const = 0;                                  // 2 points
    virtual bool is_pet( ) const = 0;                                  // 2 points
};

class Seagull : public Bird                                             // 1 point
{
  public:
    virtual string name( ) const { return "Seagull"; }   // 2 points
    virtual bool is_pet( ) const { return false; }   // 2 points
};

class Parrot : public Bird                                              // 1 point
{
  public:
    virtual string name( ) const { return "Parrot"; }   // 2 points
    virtual bool is_pet( ) const { return true; }   // 2 points
    virtual void talk( ) const { cout ≪ "Parrot can talk" ≪ endl; }   // 2 points
};

class Parrot_Cockatoo: public Parrot                                    // 1 point
{
  public:
    virtual string name( ) const { return "Cockatoo"; }   // 3 points
    // virtual bool is_pet( ) const { return true; } // optional
    virtual void talk( ) const { cout ≪ "Cockatoo talks better" ≪ endl;
}  // 3 points
};
```

Problem 5 [23 points] Order of Construction and Destruction

You are given the following class definitions.

```
class Bulb                                                        /* File: order.h */
{
  public:
    Bulb( ) { cout << "B " << endl; }
    ~Bulb( ) { cout << "~B" << endl; }
};

class Lamp
{
    Bulb bulb;
  public:
    Lamp( ) { cout << "L" << endl; }
    ~Lamp( ) { cout << "~L" << endl; }
};

class Room
{
  public:
    Room( ) { cout << "R" << endl; }
    ~Room( ) { cout << "~R" << endl; }
  private:
    Lamp l;
};

class Dining_Room : public Room
{
  public:
    Dining_Room( ) { b = new Bulb; cout << "D" << endl; }
    ~Dining_Room( ) { delete b ; cout << "~D" << endl; }
  private:
    Bulb *b;
    Lamp l;
};
```

Write the outputs of the following program. Part of the outputs is given to you already; so only fill out the remaining outputs.

```
#include <iostream>
using namespace std;
#include ”order.h”

int main ( )
{
    cout ≪ "--- Part (a) --- " ≪ endl;
    Dining_Room diningroom;

    cout ≪ "--- Part (b) --- " ≪ endl;
    Room room;

    cout ≪ "--- Part (c) --- " ≪ endl;
    { Lamp lamp1; }

    cout ≪ "--- Part (d) --- " ≪ endl;
    Lamp *lamp2 = new Lamp [2];
    delete [ ] lamp2;

    cout ≪ "--- Part (e) --- " ≪ endl;
    return 0;
}
```

Answer:

```
--- Part (a) ---
```

--- Part (b) ---

--- Part (c) ---

--- Part (d) ---

--- Part (e) ---

Q5. Solution

- Sub-grade distribution: (a) 6 (b) 2 (c) 4 (d) 3 (e) 8
- grade by blocks as indicated.

```
--- Part (a) ---
B
L
R <--- 2 points
B
L <--- 2 pointsL
B
D <--- 2 points
--- Part (b) ---
B
L
R <--- 2 points
--- Part (c) ---
B
L <--- 2 points
~L
~B <--- 2 points
--- Part (d) ---
B
L <--- 1 points
B
L <--- 1 points
~L
~B <--- 0.5 pointsB
~L
~B <--- 0.5 points
--- Part (e) ---
~R
~L
~B <--- 2 points
~B
~D <--- 2 points
~L
~B <--- 2 points
~R
~L
~B <--- 2 points
```

Problem 6 [20 points] Operator Overloading for an Array Class

```
#include <iostream>                                            /∗ File: Array.h  ∗/

class Array
{
    friend std::ostream &operator≪(std::ostream&, const Array&);
  public:
    Array(int = 10);              // Default constructor creating an array of 10 int zeros
    ∼Array( );                                                          // Destructor
    bool operator==( const Array & ) const;                       // Equality operator

    // Inequality operator; opposite of operator==
    bool operator!=( const Array &right ) const;                             // Part (a)

    // Subscript operator[ ] for non-const objects returns modifiable lvalue
    // Part (b) Fill in the member function prototype here:



    // Subscript operator for const objects returns rvalue
    // Part (c) Fill in the member function prototype here:



  private:
    int size;                                                // Pointer-based array size
    int ∗ptr;                          // Pointer to the first element of pointer-based array
};
```

The above file “`Array.h`” shows a typical class definition for a dynamic integer array. Your task is to overload the inequality operator `operator!=`, and 2 versions of the subscript `operator[]`:

i. one for non-const objects returning a modifiable `int` lvalue, and

ii. one for const objects returning an `int` rvalue.

Here is a checklist of the requirements:

- The following is a testing program "`Array-test.cpp`" for the Array class.

```cpp
/* File: Array-test.cpp — Array class test program */
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
#include "Array.h"

int main( )
{
    Array integers1( 7 );                              // seven-element Array
    Array integers2;                           // 10-element Array by default

    // Use the overloaded inequality (!=) operator
    cout << "Evaluating:  integers1 != integers2" << endl;
    if ( integers1 != integers2 )
        cout << "integers1 and integers2 are not equal" << endl;

    // Use overloaded subscript operator to create rvalue
    cout << "\nintegers1[5] is " << integers1[ 5 ];

    // Use overloaded subscript operator to create lvalue
    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
    integers1[ 5 ] = 1000;

    cout << "integers1:\n" << integers1;

    // Attempt to use out-of-range subscript
    cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
    integers1[ 15 ] = 1000;                            // ERROR: out of range
    return 0;
}
```

- You may assume that *all* other member functions in the Array class definition except the 3 required operators have been correctly implemented or overloaded and their implementations are available when you compile the executable.
- Implement the 3 required overloaded operators in a file called "`Array.cpp`", that is different from the file "`Array.h`".

- For both subscript operator functions, you are required to check if the subscript is out of range. If so, print an appropriate error message and call the function `exit(1)` to terminate the program immediately.
- Your answers should compile and work with the testing program to produce the following outputs:

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

integers1[5] is 0

Assigning 1000 to integers1[5]
integers1:
0       0       0       0       0       1000    0

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range... exiting the program
```

Here are what you need to do:

(a) Write the simplest code to score full points for overloading the inequality `operator!=` by using the equality `operator==` function. Again, you may assume the implementation of the equality `operator==` is already given so you do not need to implement it.

Answer:

(b) Fill in the function prototype of the first version of the subscript `operator[]` in the above "`Array.h`". Then write its implementation below as if it is inside the file "`Array.cpp`".

Answer:

(c) Repeat part (b) for the second version of the subscript `operator[]`.

Answer:

(d) There is a suggestion to replace the above 2 versions of subscript `operator[]` by a single operator function of the following prototype:

```
int& operator[ ]( int subscript ) const;
```

Discuss whether this is a feasible solution. That is, will it ever work with the testing program above if the function is properly implemented? An answer of simple "yes" or "no" will get *no* marks. You must give reason(s) to support your answer.

Answer:

Q6. Solution

- grade distribution: 4, 4, 4, 4
- Part (a): if it doesn't make use of operator==, at most 3 points
- Part (b) and (c): 1 point for prototype; 1 point for return value; 1 point for checking the range; 1 point for cerr and exit.
- Part (d)
  Yes, it is feasible if we don't mind that the subscript operator always returns a modifiable array item for both const and non-const Array objects. It works because it doesn't modify any data members of Array object, but only the items pointed to by Array::ptr!

```
/* Array.cpp */
#include ”Array.h”
using std::cout;
using std::cerr;
using std::endl;


// Inequality operator; returns opposite of == operator
bool Array::operator!=( const Array &right ) const
{
   return !( *this == right );                    // Invoke Array::operator==

}

/*
// Overloaded subscript operator for non-const Arrays;
// Reference return creates a modifiable lvalue
int &Array::operator[ ]( int subscript )
{
   // Check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
   {
      cerr << ”\nError: Subscript ” << subscript
           << ” out of range... exiting the program” << endl;
      exit( 1 ); // Terminate program; subscript out of range
   }

   return ptr[ subscript ];
}


// Overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[ ]( int subscript ) const
{
   // Check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
   {
      cerr << ”\nError: Subscript ” << subscript
           << ” out of range... exiting the program” << endl;
```

```
        exit( 1 ); // Terminate program; subscript out of range
    }

    return ptr[ subscript ]; // returns copy of this element
}
*/
```

```
int& Array::operator[ ]( int subscript ) const
{
    // Check for subscript out-of-range error
    if ( subscript < 0 || subscript >= size )
    {
        cerr ≪ "\nError:  Subscript " ≪ subscript
               ≪ " out of range...  exiting the program" ≪ endl;
        exit( 1 );                              // Terminate program; subscript out of range
    }

    return ptr[ subscript ];                                  // returns copy of this element
}
```

```
/* The following functions are assumed to be given */
std::ostream &operator≪(std::ostream& os, const Array& x)
{
     for (int j = 0; j < x.size; j++)
          os ≪ x[j] ≪ "\t";
     os ≪ endl;
     return os;
}
```

```
// Default constructor creating an array of n int's
Array::Array(int n) : size(n), ptr(new int [n])
{
     for (int j = 0; j < size; j++)
          ptr[j] = 0;
}
```

```
// Destructor
Array::~Array( )
{
    delete [ ] ptr;
    size = 0;
}

// Equality operator
bool Array::operator==( const Array& x ) const
{
    if (this == &x)
        return true;
    else if (size == x.size)
    {
        for (int j = 0; j < size; j++)
            if (ptr[j] != x[j])
                return false;

        return true;
    }
    else
        return false;
}
```