COMP 151: Object Oriented Programming
Spring Semester 2005
Final Examination
Saturday, 28 May 2005
4:30PM – 7:30PM

K E Y

Name: _____

E-mail: _____

ID: _____

LAB: _____

Problem		Points	Score
1	FUNCTION POINTERS AND FUNCTION OBJECTS	10	
2	OPERATOR OVERLOADING	10	
3	ITERATORS AND STL ALGORITHMS	8	
4	POLYMORPHISM	14	
5	TEMPLATES AND INHERITANCE	18	
6	INHERITANCE AND MEMBER CLASSES	9	
7	MULTIPLE-CHOICE QUESTIONS	21	
Total		90	

1 Function Pointers and Function Objects (10 POINTS)

(a) *[5 pts]* Consider the following program:

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

class Dog {
public:
    int run(int i) const {
        cout << "run:" << i << endl;
        return i;
    }

    typedef int (Dog::*PMF)(int) const;

    class FunctionObject {
        Dog* ptr;
        PMF pmem;
    public:
        FunctionObject() {
            cout << "default constructor" << endl;
        }

        FunctionObject(Dog* wp, PMF pmf)
            : ptr(wp), pmem(pmf) {
            cout << "FunctionObject constructor" << endl;
        }

        int operator()(int i) const {
            cout << "FunctionObject::operator()" << endl;
            return (ptr->*pmem)(i); // Make the call
        }
    };

    FunctionObject operator->*(PMF pmf) {
        cout << "operator->*" << endl;
        return FunctionObject(this, pmf);
    }
};

int main() {
    Dog w;
    Dog::PMF pmf = &Dog::run;
    cout << (w->*pmf)(1) << endl;
    return 0;
}
```

Give the output of this program.
Answer:

```
operator->*
FunctionObject constructor
FunctionObject::operator()
run:1
1
```

(b) *[5 pts]* Consider the following definition:

```
int (*pfunc)(float, char, int);
```

(1) What does this definition mean?

Answer:
A type that is a pointer to a function with three parameters, `float`, `char`, `int` and returns an `int`.

(2) What does it mean if the first pair of parentheses is omitted, that is

```
int *pfunc(float, char, int);
```

Does it have the same meaning as (1)? If not, what is the new meaning?

Answer:
It is not the same as (1). It now declares a function with three parameters, `float`, `char`, `int` and returns a pointer to `int`.

2 Operator Overloading (10 POINTS)

Consider the following program:

```
class Date
{
private:
    int day;
    int month;
    int year;

public:
    Date(int = 20, int = 5, int = 2005);     // constructor
};

Date::Date(int dd, int mm, int yyyy)
{
    day = dd;
    month = mm;
    year = yyyy;
}
```

(a) *[4 pts]* Overload the operator `<<` to display `Date` objects so the following works.

```
Date today(28, 5, 2005);
cout << today;           // it prints "28-5-2005" on the screen
```

You have to declare the function inside the class but define it outside.

Answer:

```
class Date
{
    friend ostream& operator<<(ostream&, const Date&);
};

ostream& operator<<(ostream& out, const Date& d) {
    return out << d.day << "-" << d.month << "-" << d.year;
};
```

(b) *[6 pts]* Overload the subscript operator `[]` for the class `Date` so that it returns the *i*th day after the current date. For example,

```
Date today(28, 5, 2005);   // today is "28-5-2005"
Date tomorrow = today[1];  // the result is "29-5-2005"
Date future = today[100];  // the result is "8-9-2005"
```

We assume for simplicity that there are 30 days in each month.

Answer:

```
class Date
{
    Date operator[](int);   // overload the subscript operator
};

Date Date::operator[](int days)
{
    Date temp = new Date(); // a temporary Date to store the result

    temp.day = day + days;  // add the days
    temp.month = month;
    temp.year = year;

    while (temp.day > 30)   // adjust the day and month
    {
        temp.month++;
        temp.day -= 30;
    }

    while (temp.month > 12) // adjust the month and year
    {
        temp.year++;
        temp.month -= 12;
    }

    return temp;            // the values in temp are returned
}
```

3 Iterators and STL Algorithms (8 POINTS)

There are a lot of STL algorithms with an **if** in their names, such as **find_if** and **count_if**. Such algorithms take a predicate as one argument and operate only on the elements for which the predicate returns true.

You are given the prototype of the function **copy_if**. Implement the function according to the specification below.

```
template<typename InputIterator, typename OutputIterator, typename Predicate>
OutputIterator copy_if(InputIterator srcBegin, InputIterator srcEnd,
                       OutputIterator dstBegin, Predicate pred)

/*
copy_if copies those elements for which pred is true from the range
[srcBegin, srcEnd) to a range beginning at dstBegin. The return value
is the end of the resulting range.
*/
```

Ans:

```
InputIterator ii = srcBegin;
OutputIterator oi = dstBegin;
while (ii != srcEnd)
{
    if (pred(*ii))
    {
        *oi = *ii;
        oi++;
    }
    ii++;
}
return oi;
```

4 Polymorphism (14 POINTS)

(a) *[10 pts]* Trace the following program and write the program output on the next blank page.

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

class Polygon {
private:
    struct Point {
        int x, y;
        Point(int x = 0, int y = 0) : x(x), y(y) {};
        void print() {
            cout << "(" << x << "," << y <<")" << endl;
        }
    };
public:
    Polygon(int n_vertice = 0, Point* coordinate = NULL)
        : n_vertice(n_vertice), coordinate(coordinate) {
    };
    virtual ~Polygon() {
        cout << "Destructor of Polygon" << endl;
        if (coordinate) {
            delete [] coordinate;
            coordinate = NULL;
        }
    };
    void print() {
        cout << "This is Polygon" << endl;
    };
    virtual void printcoord() {
        int i;
        cout << "Coordinate of Polygon:" << endl;
        for (i = 0; i < n_vertice; i++) {
            coordinate[i].print();
        }
    };
protected:
    int n_vertice;
    Point* coordinate;
};
```

```
class Rectangle : public Polygon {
public:
    Rectangle(int x = 0, int y = 0, int width = 0, int height = 0)
        : n_vertice(0), coordinate(0), Polygon() {
        n_vertice = 4;
        coordinate = new Point[4];
        coordinate[0].x = x;
        coordinate[0].y = y;
        coordinate[1].x = x + width;
        coordinate[1].y = y;
        coordinate[2].x = x;
        coordinate[2].y = y + height;
        coordinate[3].x = x + width;
        coordinate[3].y = y + height;
    }
    ~Rectangle() {
        if (coordinate) {
            cout << "Destructor of Rectangle" << endl;
            delete [] coordinate;
            coordinate = NULL;
        }
    };
    virtual void print() {
        cout << "This is Rectangle" << endl;
    }
    void printcoord() {
        int i;
        cout << "Coordinate of Rectangle:" << endl;
        for (i = 0; i < n_vertice; i++) {
            coordinate[i].print();
        }
    }
protected:
    int n_vertice;
    Point* coordinate;
};
```

```
class Square : public Rectangle {
public:
    Square(int x = 0, int y = 0, int size = 0)
        : n_vertice(0), coordinate(NULL), Rectangle(x, y, size, size) {
    }
    ~Square() {
        if (coordinate) {
            cout << "Destructor of Square" << endl;
            delete [] coordinate;
            coordinate = NULL;
        }
    };
    void print() {
        cout << "This is Square" << endl;
    }
    void printcoord() {
        int i;
        cout << "Coordinate of Square:" << endl;
        for (i = 0; i < n_vertice; i++) {
            coordinate[i].print();
        }
    }
protected:
    int n_vertice;
    Point* coordinate;
};

void print(Polygon* P) {
    P->print();
    P->printcoord();
    cout << "--------" << endl;
};

void print(Polygon P) {
    P.print();
    P.printcoord();
    cout << "--------" << endl;
};
```

```
void print(Rectangle* P) {
    P->print();
    cout << "--------" << endl;
};

void print(Rectangle P) {
    P.print();
    cout << "--------" << endl;
};

int main() {
 1.  Polygon* polygonarray[3];
 2.  Rectangle* rectanglearray[3];

 3.  Polygon polygon;
 4.  Rectangle rectangle(1, 2, 3, 4);
 5.  Square square(5, 6, 7);

 6.  cout << "Part 1" << endl;
 7.  print(polygon);

 8.  polygonarray[0] = &polygon;
 9.  polygonarray[1] = &rectangle;
 10. polygonarray[2] = &square;

 11. rectanglearray[0] = &rectangle;
 12. rectanglearray[1] = &square;

 13. cout << "Part 2" << endl;

 14. print(polygonarray[0]);
 15. print(polygonarray[1]);
 16. print(polygonarray[2]);

 17. cout << "Part 3" << endl;

 18. print(rectanglearray[0]);
 19. print(rectanglearray[1]);

 20. return 0;
}
```

Write the output of the program on this page.

```
Part 1
This is Polygon (*)
Coordinate of Polygon: (*)
--------
Destructor of Polygon (*)
Part 2
This is Polygon (*)
Coordinate of Polygon: (*)
--------
This is Polygon (*)
Coordinate of Rectangle: (*)
(1,2) (*)
(4,2) (*)
(1,6) (*)
(4,6) (*)
--------
This is Polygon(*)
Coordinate of Square:(*)
--------
Part 3
This is Rectangle (*)
--------
This is Square (*)
--------
Destructor of Rectangle (*)
Destructor of Polygon (*)
Destructor of Rectangle (*)
Destructor of Polygon (*)
Destructor of Polygon (*)
```

(b) *[4 pts]* If line 3 of the main program is replaced by the following two lines of code, what will happen?

```
Point pt[2];
Polygon polygon(2, pt);
```

(i) Is there any compilation error?
(ii) Is there any segmentation fault?
(iii) If your answer to (i) or (ii) or both is "yes", state the reasons and the corresponding corrections so that the program will work properly producing no compilation error or segmentation fault. Note that you cannot modify the main program.

Answer:

(i) there is compilation error

(ii) there is segmentation fault even if the above compilation error is fixed.

Compilation error

Reason, 'Point' : undeclared identifier

Corrections:

(Marks will only be given to students who answer 'Compilation error')

1. Move the definition of Point to the global scope.

2a. Change the constructor of Polygon:

```
Polygon(int n_vertice = 0, Point*coordinate = NULL)
    : n_vertice(n_vertice), coordinate(coordinate)
```

from shadow copy to deep copy

OR

2b. Remove the following codes from the destructor of Polygon:

```
if (coordinate) {
   delete [] coordinate;
   coordinate = NULL;
}
```

5 Templates and Inheritance (18 POINTS)

Code up `single.h`, `couple.h`, `birthday.h`, and `meeting.h` so that the following `main.cpp` program can compile and run.

As the member functions are short, simply make them inline. The sample output is also shown. Write your solution in the space provided.

Sample output:

```
date: 2000/2/8  single name : chan tai man event: birthday of John
date: 1980/12/8 single name : chan tai man event: birthday of Peter
date: 2004/3/29 single name : chan tai man event: meeting at 12:03:45
date: 2004/3/30 single name : chan tai man event: meeting at 03:20:15
date: 2004/3/31 single name : chan tai man event: meeting at 19:20:02
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
date: 1948/2/8   couple's name: Mary, Michael   event: birthday of grandpa
date: 1952/4/28  couple's name: Mary, Michael   event: birthday of grandma
date: 2001/12/5  couple's name: Mary, Michael   event: birthday of son
date: 2003/9/11  couple's name: Mary, Michael   event: meeting at 09:12:40
date: 2003/9/18  couple's name: Mary, Michael   event: meeting at 14:23:20
date: 2003/9/25  couple's name: Mary, Michael   event: meeting at 18:30:00
```

The header file `date.h` is given below. To score full marks, whenever possible, you must

- use `Date::print`
- use base class initializer in the constructor functions.

```
//-------begin of date.h----------
#ifndef DATE_H
#define DATE_H

#include <iostream>

template <class T>
class Date {
protected:
  T data;
  int year, month, day;
public:
  Date () : year(0), month(0), day(0) {};
  Date (int iy, int im, int id, const T& idata) :
    year(iy), month(im), day(id) { data.set(idata); }
  Date (int iy, int im, int id) : year(iy), month(im), day(id) { }
  void set (const T& idata) { data.set(idata); }
  virtual void print () const {
    cout << "date: " << year << "/" << month << "/" << day << '\t';
  }
};
#endif
//-------end of date.h----------
```

```
//-------begin of main.cpp----------
#include "single.h"
#include "couple.h"
#include "date.h"
#include "birthday.h"
#include "meeting.h"

int main () {
  Single p("chan tai man",19);
  Birthday<Single> d1(p,2000,2,8,"John");
  Birthday<Single> d2(p,1980,12,8,"Peter");
  Meeting<Single>  d3(p,2004,3,29,"12:03:45");
  Meeting<Single>  d4(p,2004,3,30,"03:20:15");
  Meeting<Single>  d5(p,2004,3,31,"19:20:02");
  Date <Single>*d_arr[5];
  d_arr[0] = &d1;
  d_arr[1] = &d2;
  d_arr[2] = &d3;
  d_arr[3] = &d4;
  d_arr[4] = &d5;
  for (int i=0; i<5; i++) d_arr[i]->print();

  cout << "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\n";

  Couple c("Mary", "Michael", 35, 25);
  Birthday<Couple> D1(c,1948,2,8,"grandpa");
  Birthday<Couple> D2(c,1952,4,28,"grandma");
  Birthday<Couple> D3(c,2001,12,5,"son");
  Meeting<Couple>  D4(c,2003,9,11,"09:12:40");
  Meeting<Couple>  D5(c,2003,9,18,"14:23:20");
  Meeting<Couple>  D6(c,2003,9,25,"18:30:00");
  Date <Couple>*c_arr[6];
  c_arr[0] = &D1;
  c_arr[1] = &D2;
  c_arr[2] = &D3;
  c_arr[3] = &D4;
  c_arr[4] = &D5;
  c_arr[5] = &D6;
  for (int j=0; j<6; j++) c_arr[j]->print();

  return 0;
}
//-------endof main.cpp-------------
```

ans:

```
//---- single.h ------
#ifndef SINGLE_H
#define SINGLE_H

#include <iostream>
#include <string>

class Single {
  string name;
  int age;
  // ... other private data
public:
  Single () {}                // default constructor is needed?
  Single (const char *in, int ia) : age(ia) { name = in; }
  void set (const Single &in_s) { name = in_s.name; age = in_s.age; }
  void print() const {
    cout << "single name : " << name << " " ;
  }
};
#endif
//---- couple.h ------
#ifndef COUPLE_H
#define COUPLE_H

#include <iostream>
#include <string>

class Couple {
  string name_1, name_2;
  int age_1, age_2;
  // ... other private data
public:
  Couple () {}      // default constructor is needed
  Couple (const string in1, const string in2, int ia1, int ia2) {
    age_1 = ia1; age_2 = ia2;
    name_1 = in1; name_2 = in2;
  }
  void set (const Couple &in_s) {
    name_1 = in_s.name_1; name_2 = in_s.name_2;
    age_1 = in_s.age_1; age_2 = in_s.age_2;
  }
  void print() const {
    cout << " couple's name: " << name_1 << ", " << name_2 << '\t';
  }
};
#endif
//-------birthday.h----------
#ifndef BDATE_H
#define BDATE_H

#include <iostream>
#include <string>
#include "date.h"

template <class T>
```

```
class Birthday : public Date <T> {
  string name;
  // some other private data
public:
  Birthday (const T& idata, int iy, int im, int id, const string in) :
    Date<T>(iy,im,id,idata)
  { name = in; }
  virtual void print() const {
    Date<T>::print ();  // inherit from Date
    data.print();
    cout << "event: birthday of " << name << endl;
  }
};

#endif
//-------meeting.h----------
#ifndef MDATE_H
#define MDATE_H

#include <iostream>
#include <string>
#include "date.h"

template <class T>
class Meeting : public Date <T> {
  string time;
public:
  Meeting (const T& idata, int iy, int im, int id, const string in) :
    Date<T>(iy,im,id,idata)
  { time = in; }
  virtual void print() const {
    Date<T>::print ();   // inherit from date
    data.print();
    cout << "event: meeting at " << time << endl;
  }
};

#endif
```

6 Inheritance and Member Classes (9 POINTS)

The `ListNode` and `List` class templates together can be used to implement reusable linked list data structures. The class interface for each of these two classes is given below but its implementation is not shown here.

```
//-------begin of listnode.h----------
#ifndef LISTNODE_H
#define LISTNODE_H

template<class NODETYPE> class List;

template<class NODETYPE>
class ListNode {
    friend class List<NODETYPE>;
public:
    ListNode(const NODETYPE& info);
    NODETYPE getData() const;
private:
    NODETYPE data;
    ListNode<NODETYPE>* nextPtr;
};

#endif
//-------end of listnode.h----------

//-------begin of list.h----------
#ifndef LIST_H
#define LIST_H

#include "listnode.h"

template<class NODETYPE>
class List {
public:
    List();
    ~List();
    void insertAtFront(const NODETYPE& value);
    void insertAtBack(const NODETYPE& value);
    bool removeFromFront(NODETYPE& value);
    bool removeFromBack(NODETYPE& value);
    bool isEmpty() const;
    void print() const;
private:
    ListNode<NODETYPE>* firstPtr;
    ListNode<NODETYPE>* lastPtr;
    ListNode<NODETYPE>* getNewNode(const NODETYPE& value);
};

#endif
//-------end of list.h----------
```

The `Stack` class template below inherits the implementation of `List` but not its public interface. Instead, it provides its own public interface which includes member functions such as `push` and `pop`.

```
//-------begin of stack.h----------
#ifndef STACK_H
#define STACK_H
```

```
#include "list.h"

template<class STACKTYPE>
class Stack : private List<STACKTYPE> {
public:
    void push(const STACKTYPE& data) { insertAtFront(data); }
    bool pop(STACKTYPE& data) { return removeFromFront(data); }
    bool isStackEmpty() const { return isEmpty(); }
    void printStack() const { print(); }
};

#endif
//-------end of stack.h----------
```

Instead of inheriting from `List`, give a different definition of `Stack` which has a `List`-type data member without using inheritance. The full implementation of the class should be provided in `stack.h` by defining the member functions as inline functions as in the `stack.h` file above.

ans:

```
#ifndef STACK_H
#define STACK_H

#include "list.h"

template<class STACKTYPE>
class Stack {
public:
    void push(const STACKTYPE& data) { stackList.insertAtFront(data); }
    bool pop(STACKTYPE& data) { return stackList.removeFromFront(data); }
    bool isStackEmpty() const { return stackList.isEmpty(); }
    void printStack() const { stackList.print(); }
private:
    List<STACKTYPE> stackList;
};

#endif
```

7 Multiple-Choice Questions (21 POINTS)

Choose the best answer for each question by circling the corresponding letter. Each correct answer is worth one point. Choosing no answer, a wrong answer, or multiple answers will lead to zero point (but no extra point deduction).

1. Abstract classes

   (a) contain only one pure `virtual` function.
   (b) can have objects instantiated from them if the proper permissions are set.
   (c) cannot have abstract derived classes.
   (d) are defined, but the programmer never intends to instantiate any objects from them.

   Ans: (d)

2. Base class constructors and assignment operators

   (a) are not inherited by derived classes.
   (b) should not be called by derived class constructors and assignment operators.
   (c) can be inherited by derived classes, but generally are not.
   (d) can call derived class constructors and assignment operators.

   Ans: (a)

3. Returning references to non-`const` `private` data

   (a) allows `private` functions to be modified.
   (b) is only dangerous if the binary scope resolution operator (`::`) is used in the function prototype.
   (c) allows `private` member variables to be modified, thus breaking encapsulation.
   (d) results in a compiler error.

   Ans: (c)

4. When an object of a derived class is instantiated, the _____ constructor must be called for the _____ members.

   (a) base class, base class
   (b) derived class, base class
   (c) base class, derived class
   (d) derived class, `public`

   Ans: (a)

5. `Employee` is a base class and `HourlyWorker` is a derived class, with an overridden `print` function. Given the following statements, will the output of the two `print` function calls be identical?

```
HourlyWorker h; Employee* ePtr = &h;

ePtr->print(); ePtr->Employee::print();
```

   (a) Yes, if `print` is a `virtual` function.
   (b) No, if `print` is a non-`virtual` function.
   (c) Yes, if `print` is a non-`virtual` function.
   (d) Both (a) and (c)

   Ans: (c)

6. Inside a function definition, and for an object with data element `x`, which of the following is not equivalent to `this->x`?

(a) `*this.x`

(b) `(*this).x`

(c) `x`

(d) `(*(&(*this))).x`

Ans: (a)

7. Which of the following is false?

   (a) With a non-template class, one copy of a `static` data member is shared among all objects created from that class.

   (b) Each template class instantiated from a class template has its own copy of each `static` data member.

   (c) `static` data members of both template and non-template classes need to be initialized at file scope.

   (d) `static` member functions are shared between each class-template specialization in the class template.

   Ans: (d)

8. `static` member functions

   (a) can use the `this` pointer.

   (b) can only access other `static` member functions and `static` variables.

   (c) cannot be called until their class is instantiated.

   (d) can be declared `const` as well.

   Ans: (b)

9. Which overloaded operator is not required to be a member function of a class?

   (a) `()`

   (b) `[]`

   (c) `+=`

   (d) `==`

   Ans: (d)

10. Variables defined inside a member function of a class have

    (a) file scope.

    (b) class scope.

    (c) function scope.

    (d) class or function scope, depending on whether the binary scope resolution operator (`::`) is used.

    Ans: (c)

11. Assuming the following is the beginning of the constructor definition for class `Circle`:

```
Circle::Circle(double r, int a, int b)
    : Point(a, b)
```

    The second line

    (a) invokes the `Point` constructor with values `a` and `b`.

    (b) causes a compiler error.

    (c) is unnecessary because the `Point` constructor is called automatically.

    (d) indicates inheritance.

    Ans: (a)

12. A function template can be overloaded by

    (a) using other function templates with the same function name and parameters.
    (b) using non-template functions with the same function name but different parameters.
    (c) using non-template functions with the same function name and parameters.
    (d) using other function templates with a different function name but the same parameters.

    Ans: (b)

13. To prevent class objects from being copied,

    (a) make the overloaded assignment operator `private`.
    (b) make the copy constructor `private`.
    (c) both (a) and (b)
    (d) none of the above

    Ans: (c)

14. `protected` base class members cannot be accessed by

    (a) non-`friend` functions and non-derived class member functions.
    (b) `friend`s of the base class.
    (c) derived classes.
    (d) `friend`s of derived classes.

    Ans: (a)

15. `virtual` destructors must be used when

    (a) the constructor in the base class is `virtual`.
    (b) `delete` is used on a base class pointer to a derived class object.
    (c) `delete` is used on a derived class object.
    (d) a constructor in either the base class or derived class is `virtual`.

    Ans: (b)

16. The relationship between class templates and class-template specialization is most similar to the relationship between

    (a) classes and objects.
    (b) classes and functions.
    (c) functions and return types.
    (d) headers and source files.

    Ans: (a)

17. `namespace`s cannot contain

    (a) functions.
    (b) `main`.
    (c) classes.
    (d) other `namespace`s.

    Ans: (b)

18. A difference between function-template specializations and overloaded functions is that

    (a) function-template specializations are generated by the compiler, not the programmer.
    (b) function-template specializations cannot accept user-defined types.

(c) function-template specializations perform identical operations on each data type.

(d) overloaded functions usually do not perform similar operations on each data type.

Ans: (a)

19. Which of the following is a difference between vectors and arrays?

(a) access to any element using the `[]` operator

(b) contiguous blocks of memory

(c) the ability to change size dynamically

(d) efficient, direct access

Ans: (c)

20. Polymorphism and `virtual` functions are not appropriate for

(a) programs where classes may be added in the future.

(b) programs that have strict memory and processor requirements.

(c) programs that must be easily extensible.

(d) programs that use many inherited classes with similar functions.

Ans: (b)

21. _____ is/are used to solve the problem of variables with the same name and overlapping scopes.

(a) `namespace`s

(b) Classes

(c) Casts

(d) Dynamic memory allocation

Ans: (a)