

**Instructions:**

- Print your name in the space provided below.
- Answer each question in the space provided. Legibility and organization count. If I can't read your answer, it's wrong.
- If you want partial credit, justify your answers briefly and concisely, even when justification is not explicitly required.
- There are 13 questions, priced as marked. The maximum score is 100.
- The code examples given in the test have been compiled and executed to verify correctness. Unless a question explicitly concerns itself with whether syntax is correct, you should assume that it is.
- This is a closed-book, closed-notes examination. No calculators or other electronic devices may be used during this examination.
- You may not discuss (in any form: written, verbal or electronic) the content of this examination with any student who has not taken it. You must return this test form when you complete the examination. Failure to adhere to any of these restrictions is an Honor Code violation.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.

**Do not start the test until instructed to do so!**

Name           **Solution**            
printed

**Pledge:** On my honor, I have neither given nor received unauthorized aid on this examination.

\_\_\_\_\_  
*signed*

For questions 1 through 5, consider the following class declaration and implementation:

```
class Point {
    friend ostream& operator<<(ostream& Out, const Point& P);
private:
    int    x, y;
public:
    Point(int ix = 0, int iy = 0);
    Point Shift(int deltaX, int deltaY);
    Point xReflect();
    Point yReflect();
};

Point::Point(int ix, int iy) {

    x = ix;  y = iy;
}

Point Point::Shift(int deltaX, int deltaY) {

    x += deltaX;  y+= deltaY;
    return (*this);
}

Point Point::xReflect() {

    y = -y;
    return (*this);
}

Point Point::yReflect() {

    x = -x;
    return (*this);
}

ostream& operator<<(ostream& Out, const Point& P) {

    Out << "(" << P.x << ", " << P.y << ")";
    return Out;
}
```

1. [8 points] The given class does not implement `operator=`. Should an implementation of `Point::operator=` be added to the class in order to make it more useful to client code? Justify your conclusion carefully.

**No. The default (automatic) assignment operator is adequate since the `Point` object does not allocate any memory dynamically.**

**Note that whether a `Point` stores any pointers is largely irrelevant. The mere presence of a pointer does not imply that there is a deep copy problem. The issue is purely whether the object allocates memory dynamically, and whether that memory needs to be copied when an object is copied. Consider the `Phrase` objects in the first project.**

2. [5 points] What would be output by the following code?

```
Point A(14, 32);
cout << A.Shift(1, 1).xReflect() << endl;
```

**The call to `Shift()` modifies the object `A` to represent the point (15, 33). The call to `xReflect()` operates on the copy of `A` which is returned by `Shift()`, modifying that copy to represent the point (15, -33). That object is returned by `xReflect()` and passed to the insertion operator, which prints the values (15, -33).**

3. [5 points] What would be output by the following code?

```
Point B(14, 32);
B.Shift(1, 1).xReflect();
cout << B << endl;
```

**This is identical to question 2 except that the object returned by `xReflect()` is not saved or passed to any other function or operator. So, only the changes made by the call to `Shift()` will be seen. The printed values are (15, 33).**

4. [8 points] In the given implementation, `operator<<` is **not** a member of the class `Point`. **Could** it be a member of `Point`? Explain.

**No. A binary operator can only be a member of the class to which its LEFT operand belongs. In the case of `operator<<`, the left operand will be an `ostream` object, as shown in the preceding two questions.**

5. [7 points] The class `Point` declares that `operator<<` is a friend. Is this necessary to make the given implementation of `operator<<` valid? Explain.

**Yes. In the given implementation, `operator<<` directly accesses private data members of the `Point` object it receives as a parameter. That is only possible if `operator<<` is either a member of `Point`, which is impossible, or a friend of `Point`.**

6. [5 points] In C++, when an object is used as an actual parameter (i.e., in a function call) and passed to a function by constant reference, the formal parameter (i.e., in the function implementation) is:

- 1) a copy of the actual parameter, made by the assignment operator.
- 2) a copy of the actual parameter, made by the copy constructor.
- 3) **logically the same object as the actual parameter.**
- 4) This is not allowed.
- 5) None of these

**Note that the formal parameter cannot be modified, but that is a restriction on the use of the object, and has nothing to do with whether it is a copy or the same object.**

For questions 7 through 9, consider the following class:

```
class Buffer {
private:
    char* B;
    int Size;
public:
    Buffer(int Sz = 100);
    bool Acquire(istream& In);
    void Display(ostream& Out);
    void Clear();
    ~Buffer();
};

Buffer::Buffer(int Sz) {
    Size = Sz;
    B = new char[Size];
    if ( B == NULL )
        Size = 0;
    else
        Clear();
}
...
Buffer::~Buffer() {
    delete [] B;
}
```

7. [8 points] Is the relationship between the class `Buffer` and the class (type) `char` an association or an aggregation? Justify your answer carefully.

**The `char` objects (in the dynamically allocated array) are created when a `Buffer` object is created and destroyed when that `Buffer` object is destroyed, so the `char` objects have no independent existence. Therefore, this is an aggregation.**

8. [8 points] The relationship between the class `Buffer` and the class `istream` is an association, **not** an aggregation. Explain why.

**A `Buffer` object does not contain an `istream` object as a member, does not create an `istream` member, etc.**

**The `istream` objects used by a `Buffer` object are passed to one of the `Buffer`'s member functions by the client code, which must create (and presumably destroy) those objects.**

9. [8 points] Is the relationship between `Buffer` and `istream` a **static** association or a **dynamic** association? Explain.

**An association is static if the objects that are associated is not fixed. In this case, the `Buffer` object has no data member it can use to maintain an association with an `istream` object, so the connection is established when `Acquire()` is called and broken when `Acquire()` terminates. So, the relationship itself is transitory.**

**In addition, the particular `istream` object passed to `Acquire()` may change on each call from the client, and the same `istream` object may be passed to multiple `Buffer` objects.**

**So this association is inherently dynamic.**

For questions 10 and 11, consider the following classes:

```

class BoxCar {
private:
    string Label;
    Crate** Cargo; //ptr to array of ptrs
    int Size;
    int numCars;
public:
    BoxCar(string L = "None",
            int Sz = 10);
    BoxCar(const BoxCar& RHS);
    bool addCrate(Crate*& newCrate);
    BoxCar& operator=(
        const BoxCar& RHS);
    ~Boxcar();
};

BoxCar::BoxCar(string L, int Sz) {

    Label = L;
    numCars = 0;
    if (Sz <= 0) {
        Size = 0;
        Cargo = NULL;
    }
    else {
        Size = Sz;
        Cargo = new Crate*[Size];
    }
}

BoxCar::BoxCar(const BoxCar& RHS) {
    // implementation not shown
}

bool BoxCar::addCrate(Crate*& newCrate) {
    if (numCars == Size)
        return false;
    Cargo[numCars] = newCrate;
    numCars++;
    newCrate = NULL;
    return true;
}

BoxCar& BoxCar::operator=(const BoxCar& RHS) {
    // implementation not shown
}

Boxcar::~~Boxcar() {
    delete [] Cargo;
}

class Crate {
private:
    string Label;
public:
    Crate(string L = "None");
    string getLabel() const;
};

Crate::Crate(string L) {
    Label = L;
}

string Crate::getLabel() const {
    return Label;
}

```

10. [8 points] Consider execution of the following code fragment:

```

for (int I = 0; I < 10; I++) {
    BoxCar* B = new BoxCar("foo", 100); // note: NOT an array declaration!!
    delete B;
}

```

Determine whether this code causes a memory leak. If yes, explain clearly how the leak occurs. If no, explain clearly what prevents a leak from occurring.

**On each pass through the loop body, a `BoxCar` object is allocated dynamically; that object in turn allocates an array of 100 `Crate` pointers (but NO `Crate` objects).**

**The next statement deallocates the `BoxCar` object by calling `delete`. Calling `delete` on a pointer to an object causes the destructor for that object to fire; in this case `~BoxCar()` deallocates the array of `Crate` pointers.**

**So there is not a memory leak.**

11. [8 points] Does the class `Crate` need a destructor in order to prevent memory leaks? If not, give a complete explanation why not. If yes, write an implementation of the needed destructor.

**Crate objects do not directly allocate dynamic memory, so no destructor is needed.**

**A `Crate` object does contain a `string` object, which does allocate memory. However, the `string` destructor will automatically fire whenever a `Crate` object is destroyed, and that will deallocate that memory.**

---

For questions 12 and 13, consider the following class declaration and implementation:

```
class Polynomial {
private:
    int* Coeff;
    int Degree;

public:
    Polynomial(int Deg, int C[]);
    Polynomial(const Polynomial& Source);
    Polynomial operator=(const Polynomial& RHS);
    . . .
    int evalAt(int X) const;
    ~Polynomial();
};

Polynomial::Polynomial(int Deg, int C[]) {
    Degree = Deg;
    Coeff = new int[Deg + 1];           // # of coeff's == degree + 1
    for (int Pos = 0; Pos <= Deg; Pos++)
        Coeff[Pos] = C[Pos];
}

. . .
int Polynomial::evalAt(int X) const {
    int Value;
    Value = Coeff[Degree];
    for (int Pow = Degree - 1; Pow >= 0; Pow--)
        Value = Value * X + Coeff[Pow];
    return Value;
}

Polynomial::~~Polynomial() {
    delete [] Coeff;
}
```

12. [10 points] Write the implementation of the copy constructor for the class `Polynomial`. (You may not assume that there is already a correct implementation of `operator=` for `Polynomial` objects.)

```
Polynomial::Polynomial(const Polynomial& Source) {
    Degree = Source.Degree;           // copy degree of source poly
    Coeff = new int[Degree];          // allocate array for coefficients
                                        //   of new poly
    int pos;
    for (pos = 0; pos < Degree; pos++) // copy coefficients of source poly
        Coeff[pos] = Source.Coeff[pos];
}
```

Note: there's no reason to test for self-copying because it is impossible. If the constructor is being called, then a NEW object is being created, and it cannot possibly be the same as the source object.

Also note: there's no reason to worry about "cleaning up" the target object's dynamic memory, because no dynamic memory has been allocated yet for that object. Basically, the point is the same as above, but there's an additional issue. When the constructor is called, the pointer `Coeff` of that object has been declared, but it has not been initialized. So, if you call `"delete [] Coeff"` inside the copy constructor, you will probably have a runtime error due to an attempt to deallocate memory that your program does not own.

13. [12 points] Complete the implementation of the following member operator:

```
Polynomial Polynomial::operator+(const Polynomial& RHS) const {
    int sumDegree;
    int *sumCoeff;

    if (Degree >= RHS.Degree) // LHS has largest degree
        sumDegree = Degree;
    else
        sumDegree = RHS.Degree; // RHS has largest degree

    sumCoeff = new int[sumDegree]; // create array for coeffs

    int pos;
    for (pos = 0; pos < sumDegree; pos++) // clear array
        sumCoeff[pos] = 0;

    for (pos = 0; pos < Degree; pos++) // add LHS coeffs
        sumCoeff[pos] += Coeff[pos];

    for (pos = 0; pos < RHS.Degree; pos++) // add RHS coeffs
        sumCoeff[pos] += RHS.Coeff[pos];

    return Polynomial(sumCoeff, sumDegree); // create and return new poly
}
```

Note: the use of `const` on the parameter and the operator make it impossible to modify either operand, which is the way it should be. Many solutions that students came up with would have changed one (or both!) operands.