UMBC

AN HONORS UNIVERSITY IN MARYLAND

Student's Name ______________________________

Date ______________ Subject ______________________

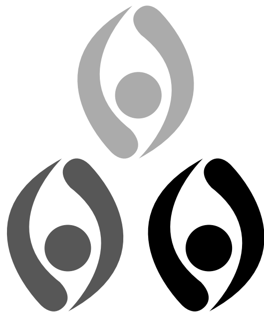
Professor's name: ______________________________

Comments: ______________________________

____

____

____



Integrity:
A Value That Endures

"By enrolling in this course, each student assumes the responsibilities of an active participant in UMBC's scholarly community in which everyone's academic work and behavior are held to the highest standards of honesty. Cheating, fabrication, plagiarism, and helping others to commit these acts are all forms of academic dishonesty, and they are wrong. Academic misconduct could result in disciplinary action that may include, but is not limited to, suspension or dismissal. To read the full Student Academic Conduct Policy, consult the UMBC Student Handbook, the Faculty Handbook, or the UMBC Policies section of the UMBC Directory."

UMBC Faculty Senate
February 13, 2001

THIS PAGE INTENTIONALLY BLANK

1. (15 points) There are *at least* five errors or omissions in the following program. Find five errors and write the the line number and correction for each in the space provided below.

```
 1 #include <iostream>
 2 using namespace standard;
 3
 4 class NoDonutEx {};
 5
 6 class Donut {
 7 public:
 8   Donut() : m_numDonuts(1) {}
 9   Donut(int numDonuts) : m_numDonuts(numDonuts) {}
10   void Eat() {
11     if (m_numDonuts = 0)
12       throw NoDonutEx;
13     else
14       cout << "Yummy donut!" << endl;
15       m_numDonuts--;
16     }
17   }
18 private:
19   int m_numDonuts;
20 }
21
22 int main() {
23   Donut d(2);
24
25   try {
26     d.Eat();
27     d.Eat();
28     d.Eat();
29   }
30   catch (DivByZeroEx) {
31     cerr << "No more donuts. I'm outta here." << endl;
32   }
33   return 0;
34 }
```

Line Number	Correction
2	**"standard" should be "std"**
11	**"=" should be "=="**
12	**"NoDonutEx" should be "NoDonutEx()"**
13	**Missing {**
20	**Missing ;**
30	**"DivByZeroEx" should be "NoDonutEx"**

2. (10 points) Complete the code:

a. I want to compute the *average* of the three integer variables *x, y*, and *z* and save it to the double variable *avg*.

```
avg = ( x + y + z ) / [ 3.0 ] ;
```

b. On a 24-hour clock, the hours are labeled 0, 1, 2, …, 23. A computationally intensive job will take 75 hours to complete; I want to determine the hour at which it will finish given the hour at which it will start (`startTime`):

```
int startTime = 11;

int endTime = (startTime + 75) [ % ] 24;
```

c. The program should only call the function `BadInput()` if the integer variable `numSelected` has a value less than 1 or greater than 42:

```
if ( [ numSelected < 1 || numSelected > 42 ] )
    BadInput(data, numSelected);
```

d. The user of a data analysis program can enter '`s`' to save their data or '`h`' to display a help message. The users selection is stored in the variable `selection`:

```
switch( selection ) {
  case 's':
    SaveData();
    break;
  case 'h':
    DisplayHelp();
    break;
  [ default ] :
    cerr << "Invalid selection" << endl;
}
```

e. I want to insert the value of the integer variable `aScore` into the integer set `scores`:

```
scores.[ insert(aScore) ] ;
```

3. (10 points) Complete the code:

a. I want to call the `Bride()` function of the `Princess` object pointed to by `pPtr`:

```
Princess *pPtr = new Princess;

pPtr [ -> ] Bride();
```

b. The variable `scores` is a pointer to an integer; it points to a dynamically allocated array of integers. That is,

```
int *scores = new int[arrayLen];
```

where `arrayLen` is a variable that depends on user input. I want to delete the array:

```
delete [ [] ] ratings;
```

c. I am overloading the assignment operator. I need to be sure I handle self-assignment (e.g. `x = x`) properly and that I return the appropriate value:

```
MyClass& MyClass::operator=(const MyClass& rhs) {

  if (this != &rhs) {

    /* execute only if NOT self-assignment */

  }

  return [ *this ] ;

}
```

d. I am writing the interface file for class `Derived`, which is derived from the class `Base`:

```
class Derived : [ public ] Base {

     /* class declarations go here */

};
```

e. I am writing the interface file for a base class from which other classes will be derived. The virtual function `ToString()` will *not* be implemented in the base class but *must* be implemented in any derived class:

```
virtual string ToString() [ =0 ] ;
```

4. (10 points) Complete the code:

a. The function `Dangerous()` may throw a `DarthVaderEx` exception. `DarthVaderEx` is an exception class; it has a `what()` function that returns a string description of the error. I want to catch the exception and print the error message by calling `what()`:

```
try {
   Dangerous();
}
catch ( [ DarthVaderEx& e  -or- DarthVaderEx e ] ){
   cerr << e.what() << endl;
}
```

b. I want to print the elements in the integer set `sizes`:

```
set<int>::iterator itr;
for (itr = sizes.[ begin() ] ; itr != sizes.end(); itr++)
    cout << [ *itr ] << endl;
```

c. I want to create a two-dimensional integer array named `data` with `NUMROWS` rows and `NUMCOLS` columns (`NUMROWS` and `NUMCOLS` are defined integer constants):

```
int [ **data ] ;
data = new int*[NUMROWS];
for (int i = 0; i < NUMROWS; i++)
    data[i] = new [ int[NUMCOLS] ] ;
```

5. (10 points) What output is produced by the following code?

```
1     int *p1, *p2;
2     int x = 3, y = 5;
3     int z[3] = {1, 2, 3};
4     p1 = &x;
5     p2 = &y;
6     cout << *p1 + *p2 << endl;
7     p2 = p1;
8     cout << *p1 * *p2 << endl;
9     p1 = z;
10    cout << *(p1 + 1) * *p2 << endl;
11    p1++;
12    cout << *p1 * *p2 << endl;
13    cout << (*p1 + 1) / *p2 << endl;
```

8

9

6

6

1

6. (30 points) True or False?

a. **F** There is one iterator type that is used with all STL containers.

b. **T** *Inheritance* implements the "is a" relationship.

c. **T** It is a good idea to throw an exception if an error occurs in a constructor.

d. **F** It is a good idea to throw an exception if an error occurs in a destructor.

e. **T** A `do-while` loop should be used if the body of the loop is to be executed one or more times.

f. **T** In Late Binding (also called Dynamic Binding), decisions as to which version of an overridden function should be called are made at run-time.

g. **T** When a derived class object is destroyed, the derived class destructor is called before the base class destructor.

h. **F** A class is *abstract* only if all of its functions are pure virtual.

i. **T** A set is a collection of unique elements.

j. **T** If `iset` is a set of integers, `iset.begin()` is the first integer in the set.

k. **F** *Encapsulation* implements the "wants a" relationship.

l. **T** Elements of arrays are stored in successive memory locations.

m. **T** A base class pointer can point to a derived class object.

n. **F** A derived class pointer can point to a base class object.

o. **F** The order of precedence is the same for multiplication (`*`) and addition (`+`).

7. (10 points) Consider the following program:

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Dinosaur {
6 public:
7   Dinosaur() {}
8   virtual string ToString() { return "I am a dinosaur."; }
9 };
10
11 class Pteranadon : public Dinosaur {
12 public:
13   Pteranadon() : Dinosaur(), m_name("Pterry") {}
14   Pteranadon(string name) : Dinosaur(), m_name(name) {}
15   void Fly() { cout << m_name << " is flying!" << endl; }
16   virtual string ToString() {
17     return m_name + " is a pteranadon.";
18   }
19 private:
20   string m_name;
21 };
22
23 int main() {
24   Dinosaur *dPtr = new Pteranadon("Barb");
25   Pteranadon *pPtr = new Pteranadon("Steve");
26
27   cout << dPtr->ToString() << endl;
28   dPtr->Fly();
29
30   cout << pPtr->ToString() << endl;
31   pPtr->Fly();
32
33   return 0;
34 }
```

a. Line 28 causes an error when the program is compiled. Why?

Dinosaur class does not have a Fly() function.

b. If line 28 is deleted and the program is compiled and run, what output will it produce?

Barb is a pteranadon.

Steve is a pteranadon.

Steve is a flying!

8. Write the required code:

a. (5 points) Write the implementation of a function to compute the sum of the elements of an integer array. The function should accept the array and the array size as parameters and return the integer sum of the elements. You do not need to comment the function.

```
int sum(int *array, int size) {
   int sum = 0;
   for (int i = 0; i < size; i++)
      sum += array[i];
   return sum;
}
```

b. (5 points) Write a function that allows the user to enter non-negative integer values on the keyboard and then returns a vector containing the values entered. The user enters a negative value to indicate that they are done (the terminating negative value should *not* be included in the vector). The function must provide appropriate instructions to the user. You do not have to comment the function or worry about `#include`s. You may assume the user will enter valid integer data.

```
vector<int> GetInput() {
   vector<int> data;
   int input;

   cout << "Enter a nonnegative integer (negative to end):";

   cin >> input;

   while ( input >= 0 ) {
      data.push_back( input );

      cout << "Enter another nonnegative integer (negative to end):";

      cin >> input;

   }

   return data;
}
```

c. (5 points) Consider the following class definition:

```
 1 class Kennel {
 2 public:
 3   Kennel(int numPens) : m_numPens(numPens) {
 4     m_occupied = new bool[m_numPens];
 5     for (int i = 0; i < m_numPens; i++)
 6       m_occupied[i] = false;
 7   }
 8   Kennel& operator=(Kennel& k) {
 9     if (this != &k) {
10       if (m_occupied != NULL)
11         delete [] m_occupied;
12       m_numPens = k.m_numPens;
13       m_occupied = new bool[k.m_numPens];
14       for (int i = 0; i < m_numPens; i++)
15         m_occupied[i] = k.m_occupied[i];
16     }
17     return *this;
18   }
19 private:
20   int m_numPens;
21   bool* m_occupied;
22 };
```

The overloaded assignment operator is not *exception safe*; that is, if an exception occurs during execution of the assignment operator, for instance if the attempt to allocate memory with `new` fails, then the object on the left-hand side may be corrupted. Rewrite the operator so that if `new` causes an exception, the left-hand side of the assignment is not changed.

See next page for details.

```
Kennel& operator=(Kennel &k) {

    if (this != &k) {

        bool* tmpArray = new bool[ k.m_numPens ];

        for ( int i = 0; i < k.m_numPens; i++ )

             tmpArray[i] = k.m_occupied[i];

        if ( m_occupied != NULL )

            delete [] m_occupied;

        m_numPens = k.m_numPens;

        m_occupied = tmpArray;

    }

    return *this;
}
```

Page	Points	Earned
1	15	
2	10	
3	10	
4	10	
5	10	
6	30	
7	10	
8	5	
9	5	
10	5	
Total	**110**	