CS250 Final Review Questions

The following is a list of review questions that you can use to study for the final. I would first make sure you review all previous exams and make sure you fully understand everything on each of those exams. Then review your lecture notes for the course, and finally work through as many of these topics as you feel is necessary. This is not a comprehensive list of terms and examples, just what I had time to put together for you to use as a study guide.

Important Terminology

- struct & class and the differences between the two
- class & object and the differences between the two
- public vs private vs protected
- pointers to static data and dynamic data
- array notation vs pointer notation especially pointers to string data
- object-oriented programming
- operator overloading
- friend
- implicit operand & explicit operand
- runtime stack, heap, activation record and how they are used
- constructor, destructor
- this, *this, this->
- static variables
- inheritance
- polymorphism
- abstract class
- derived class
- concrete class
- superclass and subclass
- virtual function vs a pure virtual function (how are they different? when would you use one over the other?)
- overloading vs overriding vs redefining

Arrays

a) Initializing

```
int values[] = {5, 4, 3, 2, 1};
int i = 2;
```

b) Accessing

```
cout << values[i]++;
cout << values[i++];
cout << values[++i];
```

c) As arguments

```
void swap (int &i, int &j)
{
  int t = i;
  i = j;
  j = t;
```

```
}
swap (values[2], values[3]);
swap (values[4], values[4]);
```

d) Array manipulation - finding largest value, smallest value, exchanging elements, inserting a value into an array in an arbitrary location. Array manipulation can be code segments or as functions and may require pointer notation.

e) Parallel arrays - copying one array to another, are two arrays the same, append one array on the end of another, storing related information into parallel arrays, searching and returning data in parallel arrays

f) Two-dimensional arrays – initializing, accessing, traversing a 2D array by rows or by columns, passing 2D arrays (or array elements) to functions, average values in a row or a column or down the diagonal

Structures

a) Declare

```
struct Point
{
  double x, y;
};
```

b) Define

```
Point sPt;
```

c) Access

```
sPt.x = 2;
```

d) As argument

```
void printPoint (const Point &rsPt)
{
  cout << rsPt.x << " " << rsPt.y;
}
```

e) Structure Variables - comparing, displaying, initializing

f) No unions

g) Arrays of structs - loading an array of structs from a file, searching, inserting into, deleting from

Classes

a) Declare

```
class Point
{
  private:
    double x, y;
  public:
    Point (double x, y); // constructor
    Point ();            // default constructor
```

```
    void setX (double);  // mutator
    void setY (double);       // mutator
    void printPt () const;    // accessor
};
```

b) Define

```
Point cPt, *pcPt = new Point;
```

c) Access

```
cPt.setX (4.5);
```

d) Defining Member function

```
void Point::setX (double xCoord)
{
  x = xCoord;
}
```

e) Class Variable - comparing, displaying, initializing, using a constructor with no arguments, using a constructor with arguments

f) What is object-oriented programming?

Inheritance

Show the output of the following program:

```
class Base
{
  public:
     Base (){cout << "Base" << endl;}
    Base (int i){std::cout << "Base" << i << std::endl;}
    ~Base(){std::cout << "Destruct Base" << std::endl;}
};

class Der: public Base
{
  public:
    Der (){std::cout << "Der" << std::endl;}
    Der (int i): Base(i) {std::cout << "Der" << i << std::endl;}
    ~Der (){std::cout << "Destruct Der" << std::endl;}
};

int main()
{
  Base a;
  Der d(2);
  return 0;
}
```

Polymorphism

Using the three classes defined next, answer parts a – c.

```
class Automobile
{
 public:
  Automobile();
  virtual string getName() const = 0;
  virtual string getType() const;
  string getColor() const;
...
};

class Car : public Automobile
{
 public:
  Car();
  virtual string getName() const;
  virtual string getType() const;
  virtual string getColor() const;
  virtual void driveMe();
...
};

class Sedan : public Car
{
 public:
  virtual string getType() const;
  virtual void driveMe();
...
};
```

a. What is the problem with the following statement?

```
Automobile anAuto;
```

b. Are there any problems with the code snippet below? Why or why not?

```
void drive (Car someCar)
{
  someCar.driveMe();
}

int main( )
{
  Sedan sedan;
  drive (sedan);
  return 0;
```

```
}
```

c. For the statements listed below (1-4), indicate the class from which the function will be called (i.e. Sedan, Car, or Automobile)

```
Sedan sedan;
Car car;

Automobile *p1 = &sedan;
Automobile *p2 = &car;

cout << sedan.getColor() << endl;  // (1)
cout << p1->getType() << endl;     // (2)
cout << p2->getColor() << endl;    // (3)
cout << car.getColor() << endl;    // (4)
```

Operator overloading

- Declare a class named Triple with three private data members (floats) x, y, and z. Provide public functions for setting and getting values of all the private data members. Define a constructor that initializes the values to user-specified values or, by default, sets the values all equal to 0. Also overload the following operators:
    - Addition so that corresponding elements are added together
    - Output so that it displays the Triple in the form "The triple is (x, y, z)."
    - Post-increment so that x and z are increased by one each.

Static variables

- Write a class that contains two class data members *numBorn* and *numLiving*. The value of *numBorn* should be equal to the number of objects of the class that have been instanced. The value of *numLiving* should be equal to the total number of objects in existance currently (ie, the objects that have been constructed but not yet destructed.)
- Explain why a static member function of a class shouldn't try to access a this variable.

Destructor

- What is a destructor and when is a destructor called?
- Why would you want to have a destructor?
- When is a virtual destructor necessary?

Pointers & dynamic memory allocation

- Consider the following C++ program:

```
class Base
{
 public:
 void show ()
 {
  std::cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show ()
 {
  std::cout << "Derived Class";
 }
}

int main ()
{
 Base* b;
 Derived d;
 b = &d;
 b->show ();
}
```

- Does the above example contain any static binding? If so, what?
- Does the above example contain any dynamic binding? If so, what?
- Is show overloaded, overridden, or redefined? Explain.
- Make sure you know how to make show overloaded, overridden, and redefined?

What is a memory leak? Create a memory leak.