Name: Zaid Qureshi Poly-Id: 0185649

1. [extra credit] Who created C?

   a. Gosling
   b. Kildall
   c. Ritchie
   d. Stroustrup
   e. Thompson
   f. van Rossum
   g. Wirth
   h. Wall

[Questions 2 - 10 are worth five points each]

2. Given a class called `Thing` and the code

```
Thing thingOne, thingTwo;
```

What function **call** is the following line equivalent to?

```
thingTwo = thingOne;
```

   a. `operator=(thingTwo, thingOne)`
   b. `thingTwo.operator=(thingOne)`
   c. `ostream& Thing::operator=(const Thing& rhs);`
   d. Neither (a) nor (b) because it is using the `Thing` copy constructor.
   e. Neither (a) nor (b) because the operator has to be overridden as a friend
   f. Either (a) or (b), depending on how the programmer chose to implement the operator.
   g. None of the above

3. Given a vector of ints called intVec, write a "ranged for" loop, also sometimes known as a "foreach" loop, to double the values of all the elements in the vector.

```
for(int& item : intVec)
{
    item *= 2;
}
```

4. Assume that the class Eeyore has been defined and that the + operator has been overloaded as a non-member function to add two Eeyores. *What other function is* needed in order to allow the lines below to compile.

```
Eeyore e;
e + 5;
```

- *Do not* overload the + operator again.
- *Do not* implement this new function. Just give its prototype.

```
Eeyore(int); //Constructor
```

Prototype: Eeyore(int)

8. Given:

```
class Member {
public:
  Member() {cout << 1;}
};

class Base {
public:
  Base( ) {cout << 2;}
  Member member;
};

class Derived : public Base {
public:
  Derived() {cout << 3;}
};

int main() {
  Derived der;
}
```

What is the output?

a. 123
b. 132
c. 213
d. 231
e. 312
f. 321
g. Fails to compile
h. Runtime error (or undefined behavior)

9. Given:

```
class Integer {
public:
   Integer(int n) { val = n; }
private:
    int val;
};
```

What has to be added to the Integer class, so that the following will correctly display "myInt is positive" when the value in myInt is positive:

```
int main() {
    int n;
    cin >> n;
    Integer myInt(n);
    if(myInt) cout << "myInt is positive\n";
}
```

bool operator==( )
{ if (val > 0) return true; }

Integer:: operator bool( ) const{
return val > 0; }

Name: Zaid Qureshi Poly-Id: 0485649

11. [8 pts] What is the result of the following?

```
class Base {
public:
    virtual void foo() { cout << " - Base::foo()\n"; }
};
class Derived : public Base {
public:
    void foo() { cout << " - Derived::foo()\n"; }
};

void func(Base& arg) {
    cout << "func(Base)";
    arg.foo();
}
void func(Derived& arg) {
    cout << "func(Derived)";
    arg.foo();
}

void otherFunc(Base& arg) {
    func(arg);
}

int main() {
    Derived d;
    otherFunc(d);
}
```

a. The program runs and prints:
`func(Base) - Base::foo()`

b. The program runs and prints:
`func(Derived) - Derived::foo()`

c. The program runs and prints:
`func(Base) - Derived::foo()`

d. The program runs and prints:
`func(Derived) - Base::foo()`

e. The program fails to compile

f. A runtime error (or undefined behavior)

g. None of the above

Name: Zaid Qureshi Poly-Id: 0485649

Ok, so **how will you represent** all of this? You should be able to work out a good design, but let me "help" you.

- Since the baker hands over the collection of treats that he has baked, he needs to have a *pointer* to the collection.
- No, don't ask me what sort of collection to use. Use whatever you like. But the baker better be free to bake and store as many treats in your container as needed. We don't know how many that will be.
- And the **collection itself needs to be on the heap.**
- When the baker bakes a treat, if he has no place to put it, either because he just came on the job or because he just delivered his collection to a company, he will first need to get / create another collection. Where? Again, obviously from the heap.
- The treats will later be repackaged and sold and eaten, finally being destroyed in the process.

No you don't have to represent all of those steps, but clearly the treats must each exist on the heap so they can have an arbitrarily long shelf-life and can be moved about as needed by all the code that will use your class.

Sample test code:

```
int main() {
    Baker fred("fred");
    cout << fred << endl;
    fred.bakes("Twinkie");
    fred.bakes("Cupcake");
    fred.bakes("Twinkie");
    fred.bakes("Twinkie");
    fred.bakes("Cupcake");
    fred.bakes("Wonderbread");
    cout << fred << endl;

    Baker joe("joe");
    cout << joe << endl;
    joe = fred;
    cout << fred << endl;
    cout << joe << endl;

    // Don't implement Bakery
    Bakery hostess("Hostess");
    hostess.receives(fred.delivers());
    cout << fred << endl;
    cout << joe << endl;
}
```

Resultant output:

```
Baker: fred; No treats :-(
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
Baker: joe; No treats :-(
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
Baker: fred; No treats :-(
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
```

12)

```
class Person {
public:
    void marry(Person& other) {
        if (spouse != NULL)          // if you had a spouse
            spouse->spouse = NULL;   // Now they are not married to you
        spouse = &other;             // and you are now married to them
        if (other.spouse != NULL)    // if your new spouse was married
            other.spouse->spouse = NULL;   // now she's not married to you
        other.spouse = this;         // and married to you
    }
    Person(const string& name) : name(name), spouse(NULL) {}
};

friend ostream& operator<<(ostream& os, const Person& p) {
    os << p.name << " married to: " << (p.spouse == NULL ? "No one" : p.spouse->name);
    return os;
}

private:
    string name;
    Person* spouse;
};
```

```
[13]  class Foo {
        private:
          Vector<Glob*> vec;
        public:
          Food operator= (const Food& rhs) {
            // First, check if the same
            if (&rhs == this)
                return *this;
            // clean up all because we don't want memory leak
            for (size_t i=0; i<vec.size; i++)
                delete vec[i];
            // copy over vector parameters
            vec = rhs.vec;      // copy over size essentially and pointers.
            for( size_t i=0; i< vec.size; i++)   // copy over glob
                vec[i] = new Glob (*rhs.vec[i]);
            return *this;
          } // end "="
          bool operator == ( const Food rhs) const {
            // check if same
            if ( this == &rhs)
                return true;
            if (vec.size != rhs.vec.size)   // make sure sizes are equal
                return false;
            for(size_t i=0; i<vec.size; i++)     // make sure all Globs are equal
              if (*(vec[i]) != *(rhs.vec[i]))
                return false;
            return true;                         // All steps good return true.
          } // end "=="
```