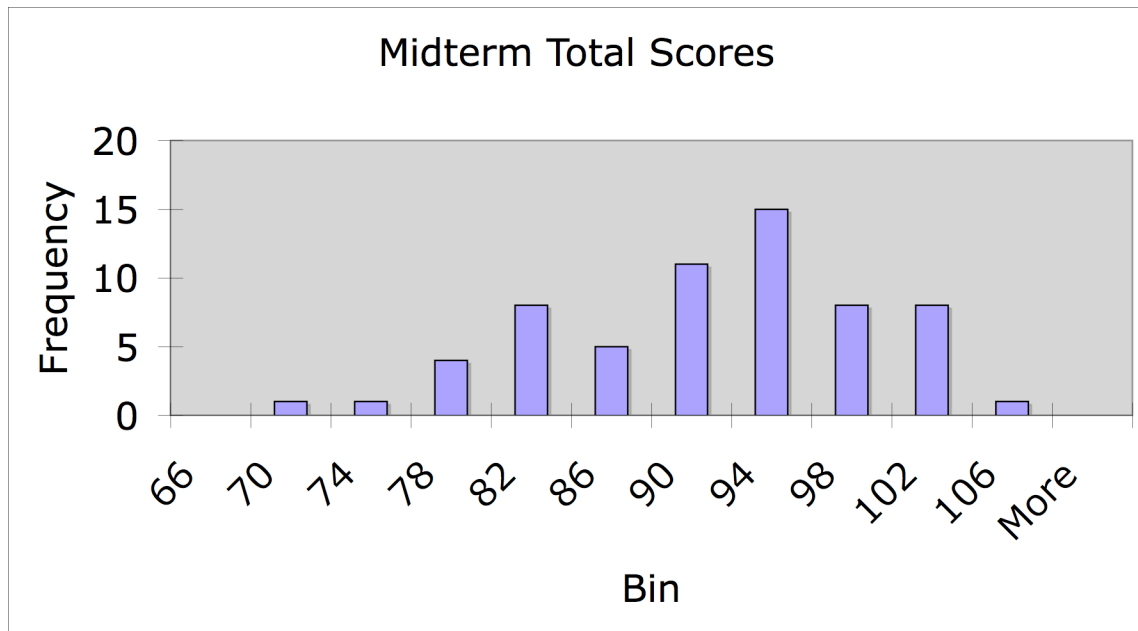


Midterm Solutions

Overall, the scores on the midterm were very good. The mean was 89.2 and the range was from the high 60's to over 100.

Here is a histogram showing the distribution of scores. If you squint enough, it's almost a perfect bell curve:



There are comments and solutions for each problem below. If you have questions about your exam, take a look at the solution first, then we'll be glad to answer any further questions.

Part 1: True or False (30 points)

1) Two primary reasons for subclassing are to achieve **code-reuse** or **polymorphism**.

Answer 1: **True.** There are other reasons, but usually you subclass because you've found a class that already does *almost* what you want to do (reuse) or you want to be able to treat all subclasses in terms of their parent (polymorphism).

2) Since references are really pointers, a reference can be re-bound to another object by taking that object's address. For example, if you start with:

```
int i, j;
int& intRef = i; // intRef is a reference to i
```

You can re-bind `intRef` to `j` by doing this:

```
intRef = &j; // intRef is now a reference to j
```

Answer 2: **False.** You cannot re-bind a reference. The code won't compile.

3) C++ supports both the Object-Oriented Paradigm and Generics.

Answer 3: **True.** The C++ feature for Generics is templates.

4) To avoid memory leaks, *every* class should have a default ctor, copy ctor, destructor, and operator=, even if the built-in versions would suffice.

Answer 4: **False.** Even if you are of the opinion that it's better style to always have these class features, it has nothing to do with memory leaks if you don't have any dynamic memory.

5) The *initializer list* is merely an alternative syntax to setting up data members in the body of the constructor. They are equivalent in functionality.

Answer 5: **False.** The initializer list is the only way to send arguments to the parent class constructor, initialize reference data members, etc.

6) Static methods and data members are only accessible from static methods. Static methods and data cannot be accessed from non-static methods.

Answer 6: **False.** A non-static method can call a static one. For example, you might have a static helper method to convert Fahrenheit to Celsius, which is called by the `setTemperature()` method.

7) Design Patterns are high-level ideas to solve object-oriented design issues.

Answer 7: **True.** Design Patterns don't necessarily tell you what code to write, but they give you ideas for how to approach a problem.

8) The compiler provides your class with a built-in no-argument constructor, but once you define *any* constructor, the built-in one is no longer generated.

Answer 8: **True.** Even if you only write a copy constructor, the no-argument constructor will go away.

9) If a function takes a parameter of type `const char*`, it can neither reassign nor delete the associated memory. For example, this won't work:

```
void foo(const char* inVal) {
    inVal = new const char[10]; // this won't compile
    delete[] inVal;           // this won't compile
}
```

Answer 9: **False.** Both lines will compile since the `const` applies to the individual elements of the array (line 1) and `const` doesn't protect against deletion (line 2).

10) Assume C is a subclass of B and B is a subclass of A. If you create an `A&` (reference to an A) that actually refers to a C object, you can call any public method of A on the object, even if class C has tried to hide it.

Answer 10: **True.** The methods of A are the methods declared public in the A definition. Any of those methods can be called on an `A&`, even if C makes those methods private.

Part 2: Short Answer (40 points)

1) Name one advantage of using Makefiles over simply typing "g++ *.cpp" to compile:

Some of the best answers include make's ability to let you avoid recompiling unchanged files, the ability to set up different targets, the ability to compile some files without compiling all the .cpp files in the directory, etc.

2) Assume class B is a subclass of class A. Which of the following lines of code will *slice* the object myB, effectively turning it into an A object and losing any of its uniqueness from class B? Check all (if any) that apply:

B myB;

NO_ A& myARef = myB; // slice?

NO_ A* myAPtr = &myB; // slice?

YES_ A myA = myB; // slice?

The last line is the only line that slices because it forces the compiler to store an object of type B in the "smaller" box of type A. The other two maintain the B object but just refer to it through a pointer or reference.

4) Find (circle and briefly describe) three separate stylistic problems with the following code. There are more than three, but **you should only indicate three!** **Note that we'll accept any three justifiable criticisms.** (answer on next page)

```
class ScottsAwesomeClass
{
    public:
        ScottsAwesomeClass(int i);

        DoStuff();

        int TheNumberOfDaysThatTheUserHasRequested;
};
```

We accepted all sort of criticisms here, even if you said something like, "Scott is not awesome and neither is ScottsAwesomeClass." Other criticisms include the poorly named methods, class, and variables, the fact that a data member is public, the fact that there are no comments, the fact that DoStuff() has no return type, and the fact that data members and method names are both capitalized.

5) We've said in class that the STL has a steep learning curve, doesn't behave well with references, and has several glaring omissions. Given these drawbacks, what are two reasons why you should still learn about and use the STL?

Because it's standardized and to avoid reinventing the wheel are the best answers. We also accepted the fact that it's well-tested (which isn't necessarily true depending on your implementation), the fact that everybody else uses it, the fact that it's cross-platform, etc.

6) Give examples of two *different* ways of using the `const` keyword. For each example, briefly describe what `const` does in that use case and why you'd use it in this way.

Here are some acceptable answers:

| Usage | What it Means | Why You'd Do It |
|--|--|---|
| <code>void foo(const char* in);</code> | The characters in the character array "in" cannot be modified. | To assure the caller that the function won't change the string. |
| <code>int getVal() const;</code> | The method <code>getVal()</code> cannot change the underlying object. | To allow the method to be called on a <code>const</code> instance of the class. |
| <code>const int kMaxVal = 12;</code> | <code>kMaxVal</code> cannot be changed. | To create a constant. |
| <code>const Foo& operator+(...)</code> | The caller cannot immediately call a non- <code>const</code> method on <code>retVal</code> | To prevent modifying a temporary, unnamed variable. |

7) In Assignment 1, we didn't specify how the `Blurb` class should behave if someone called `addTagID()` twice with the same id. We want to add some new functionality so that duplicate ID's won't be added. Instead, `addTagID()` should have no effect on the `Blurb` if the tag is already present.

Write a **simple unit test** that will only pass if this functionality is in place and working. The `Blurb.h` and `Database.h` files are attached to the back of the exam for reference. You can assume that `testutils.h` is also available if you want to use it.

There were two main things we looked for here. First, was your test simple and atomic? It should only test that one thing and should only be a few lines of code. Second, did you make too many assumptions? If you depended on certain functionality of `removeTag()` or other methods, you only got full credit if you *also* tested those methods explicitly. Note that it's okay if you called `testing_getNumTags()`, since it simply returns the length of a vector.

```
void testAddDuplicateID()
{
    Blurb b("test");
    b.addTagID(1);
    b.addTagID(1);

    int numTags = b.testing_getNumTags();
    if (numTags != 1) {
        failed();
    } else {
        passed();
    }
}
```

8) Assume that the `Blurb` class has been implemented using an STL `vector<int>`. The initial implementation of `addTagID()` looks like this:

```
void Blurb::addTagID(int inID)
{
    mTagIDs.push_back(inID);
}
```

The version of `addTagID()` shown above should cause the test you wrote in question 7 (above) to fail. Write a new version of `addTagID()` below that will check for duplicates before inserting, thus passing the test:

```
void Blurb::addTagID(int inID)
{
    if (hasTagID(inID)) return;

    mTagIDs.push_back(inID);
}
```

Part 3: Object Tracing (30 points)

What will this program print? Feel free to rip this page out of the exam. Your answer goes on the next page.

```
class A {
public:
    A() { cout << "A ctor" << endl; }
    A(const A& a) { cout << "A copy ctor" << endl; }
    virtual ~A() { cout << "A dtor" << endl; }

    virtual void foo() { cout << "A foo()" << endl; }
    virtual A& operator=(const A& rhs) { cout << "A op=" << endl; }
};

class B : public A {
public:
    B() { cout << "B ctor" << endl; }
    virtual ~B() { cout << "B dtor" << endl; }

    virtual void foo() { cout << "B foo()" << endl; }
protected:
    A mInstanceOfA; // don't forget about me!
};

A foo(A& input) {
    input.foo();
    return input;
}

int main() {
    B myB;
    B myOtherB;
    A myA;

    myOtherB = myB;
    myA = foo(myOtherB);
}
```

Very few people got this 100% correct. The most common mistake was forgetting that B's operator= (the built-in version) is going to call operator= for mInstanceOfA. Many people got it entirely correct except for one missing call to "A op=". Several people also forgot about the temporary object that gets created when foo() returns. We also saw some operator= versus copy constructor confusion.

We took off 1.5 points for each missing or incorrect statement. If you had the right output in the wrong order, it was 1 point off for each occurrence. Also, since we didn't explicitly discuss in class when the destruction of temporary objects occurs, we gave you full credit as long as you acknowledged that their destructors get called *somewhere*.

```

// instantiate myB
A ctor // for myB's superclass, A
A ctor // for myB's data member, mInstanceOfA
B ctor // for myB's ctor
// instantiate myOtherB
A ctor // for myOtherB's superclass, A
A ctor // for myOtherB's data member, mInstanceOfA
B ctor // for myB's ctor
// instantiate myA
A ctor // for myA's ctor
// myOtherB = myB;
A op= // for copying myB.mInstanceOfA
A op= // for copying the A part of myB
// myA = foo(myOtherB)
B foo() // foo() is a virtual method!
A copy ctor // make a copy for return value
A op= // assign return value to myA
// pop the stack values
A dtor // delete temporary return value
A dtor // delete myA
B dtor // delete myOtherB
A dtor // delete myOtherB.mInstanceOfA
A dtor // delete the A part of myOtherB
B dtor // delete myB
A dtor // delete myB.mInstanceOfA
A dtor // delete the A part of myB
```

Bonus Question (5 points)

In C++, you're not allowed to have references in an array. For example, the following code will not compile:

```
int& myIntRefArray[20]; // BUG! Won't compile!
```

Why doesn't the language allow this feature? What other language features would break if you were allowed to create arrays of references?

Many people pointed out the fact that you can't bind a reference after the fact. That's true, but since we're changing the rules to allow arrays of references, we could change the rules to allow re-binding too. And even if we didn't, we could still do this:

```
int& ref = i;  
int& myIntRefArray[1] = { ref };
```

So no rebinding would be necessary. What we're trying to get at is not why you can't declare the array, but what would break if you *did*. The answer is **pointer arithmetic**.

In C++ (and C), if you take the address of an array element (e.g. `&myArray[3]`), you not only get a pointer to that element, but you get a pointer to the location of the element in the array. That's what allows you to do pointer arithmetic. But if you somehow had an array of references, `&myArray[3]` would be a pointer to the array element, but it would *not* be a pointer to that element's location in the array. The reason is that `&` on a reference will give you the address of the underlying object, not the reference itself.

If you said anything about pointer arithmetic breaking, you got full credit. About 5 people got the extra credit problem correct – I'm very impressed!

Blurb Reference:

```
class Blurb {
public:
    Blurb(const std::string& inText);
    ~Blurb();

    void addTagID(int inID);
    bool removeTag(int inID);
    bool hasTagID(int inID);

    const std::string& getText() const;

    // for testing only!
    int testing_getNumTags();
};
```

STL vector Reference:

```
vector<int> myVectorOfInts;           // declare a vector of ints

myVectorOfInts.push_back(7);         // add a 7 to the end of the vector
int i = myVectorOfInts[0];           // look at element 0
int size = myVectorOfInts.size();    // returns number of elements

// Iterators
for (vector<int>::iterator myIterator = myVectorOfInts.begin();
     myIterator != myVectorOfInts.end(); ++myIterator)
{
    // process *myIterator
}
```