CSc 328, Spring 2004
Final Examination Solutions

Note: Statistics such as per-problem averages can be found here:
www.cs.arizona.edu/classes/cs328/ex1stats.xls

Problem 1: (1 point each, 10 points total)

For the following multiple choice questions, circle the best answer.

(1) In the statement `cout << "hello!" << endl;`, `endl` is a/an:
- a) extractor
- b) inserter
- c) manipulator
- d) terminator

(2) In the definition `int Rectangle::getArea() { ... }`, the '::' is called the
- a) class binder
- b) scope resolution operator
- c) member resolver
- d) method specifier

(3) Consider the following fragment of a Java class definition:

```
class Lamp {
   ...
        private Bulb itsBulb;
        }
```

Which of the following is the closest equivalent for `itsBulb` in C++? (assume all are in a private section):
- a) `Bulb  itsBulb;`
- b) `Bulb  *itsBulb;`
- c) `Bulb&  itsBulb;`
- d) `Bulb&  *itsBulb;`

[A popular wrong answer was (c) but (b) is better because it allows the possibility that `itsBulb` can be zero, just like `itsBulb` could be `null` in Java.]

(4) Consider this fragment of C++ code:

```
char c;
while (cin >> c) { ... }
```

What is that causes the while loop to be exited at end-of-file on standard input?

a) At end-of-file the character c will have the value 0.
b) The extraction fails and the failure propagates to the while.
c) At end-of-file the expression cin >> c evaluates to 0.
d) A break statement is executed in the body of the loop (not shown).

(5) Given no other knowledge of a class X, which of the following is the best indicator that X requires a copy constructor?

a) The class defines this assignment operator: X& operator=(const X&).
b) Memory allocation is performed in the constructor.
c) The class has a destructor.
d) There is a data member that is a pointer.

[None of these are absolute indicators that a copy constructor is required but in practice it is very rare to see a class that requires an assignment operator but doesn't need a copy constructor. It is less rare to see a class that does allocation in the constructor but doesn't require a copy constructor. (Such an allocation would likely be followed by a deallocation in the constructor.)]

(6) Default arguments can be used to

a) Specify missing arguments for overloaded operators.
b) Reduce the number of overloaded functions.
c) Control the order of evaluation in a member initialization list.
d) All of the above

[A popular wrong answer was (a) but that's a nonsensical alternative—you can't omit arguments for operators (only functions)!]

(7) For overloading, the prefix and postfix forms of the increment (++) and decrement operators (--) are distinguished by:

a) The postfix keyword.
b) Differing numbers of parameters.
c) Use of operator+++ and operator--- for the prefix forms.
d) There's no need to distinguish the prefix and postfix forms when overloading.

(8) Which of the following best describes the ordering of destructor calls for stack-resident objects in a routine:

a) The first object created is the first object destroyed; last created is last destroyed.
b) The first object destroyed is the last object created; last created is first destroyed.
c) Objects are destroyed in the order they appear in memory—the object with the lowest memory address is destroyed first.
d) The order is undefined and may vary from compiler to compiler.

(9) The statement `char *p = new char(100);` does which of the following?

a) Allocates sufficient memory in the heap to hold 100 `char` values.
b) Directs the memory allocator to allocate memory starting at address 100.
c) Allocates one byte of memory in the heap and initializes with the character `'d'`.
d) The statement won't compile.

[This was a pretty tricky question but I've seen this mistake—using parentheses instead of square brackets—made by students learning C++, and it's tough to find! The statement compiles just fine but instead of allocating memory for 100 characters, it allocates space for one character and initializes it to the ASCII value of 'd' (100). See slide 42.]

(10) If `a`, `b`, and `c` are of class type `V`, `V + V` is overloaded with a free-standing function, and `V * V` is overloaded with a member function, both returning a value of type `V`, then the expression `a + b * c` is equivalent to which of the following?

a) `c.operator*(operator+(a,b))`
b) `a.operator+(b.operator*(c))`
c) `operator*(a.operator+(b),c)`
d) `operator+(a, b.operator*(c))`

Problem 2: (1 point each, 5 points total)

Indicate whether each of the following statements is true (T) or false (F)

T *In C++ there is no direct analog to this Java class definition:* `abstract class A { }`

F The `string` class in the C++ standard library is essentially equivalent to `java.lang.String`.

F The semantics associated with the `bool` type in C++ are essentially the same as the `boolean` type in Java.

F A good first step for solving performance problems is to make more extensive use of inline functions.

F Multiple inheritance provides a good way to define a class that consists of a single instance of several other classes.

[As the statistics spreadsheet shows, this true/false section was the second-hardest problem on the test. The average was 3.08, a little better than flipping a coin!

The most commonly missed question was the one concerning `bool`/`boolean`. In retrospect I decided the question was a poor one, and to compensate for it, I added one point to all scores—that's the "Adjustment" on the table of your final exam scores.

Several students indicated that the last statement, regarding multiple inheritance was true. That's a classic misuse of multiple inheritance. The red flag in the statement is "consists of"—that's clearly indicative of a has-a relationship, which in turn implies aggregation, not inheritance.]

Problem 3: (2 points each, 10 points total)

a) *The routine* copyints(const int *p1, int *p2, int N) *copies* N ints *starting at one address to another address. In which direction is the data copied: from* p1 *to* p2 *or* p2 *to* p1*? Justify your answer.*

The const qualifier on p1 indicates an intention to not change the data referenced by p1. Therefore, it is reasonable to assume that the data is copied from p1 to p2.

b) *If a data member is declared as* mutable*, what does that mean?*

mutable data members can be modified in a const member function.

c) *Define a class* X *that makes the following free standing function compilable:*

```
X f() { return 1;}
```

This is sufficient: class X { public: X(int); };

[Quite a few students missed this problem, and I was surprised by that.]

d) *What does the* const *in* void X::f() const { ... } *indicate?*

The member function f() will not modify any data members.

e) *The instructor often said that in C++, "you only pay for what you use". What did he mean by that?*

A language feature that is unused should require no overhead in execution time or memory usage.

Problem 4: (8 points)

In Java, a method f() *that requires a parameter of type* X *and returns* void *is usually declared like this:*

```
void f(X value)
```

In C++, there are several choices for that same situation. Here are four of them:

```
void f(X& value)
void f(const X& value)
void f(X* p)
void f(X value)
```

Briefly describe the meaning of each alternative AND a situation where that alternative would be the best choice.

void f(X& value)

f() is passed a reference to X. The performance is equivalent to passing a pointer. Because the reference is non-const, value can be modified. This is commonly used to pass an instance of X that is possibly modified by f(). In iostream inserters the ostream is passed as a non-const reference.

void f(const X& value)

This is the same as f(X& value) with the exception that via the const qualifier, f() "promises" to not modify the value. This form combines (almost) the safety of call-by-value with the speed of passing only the address of an arbitrarily large object. It is the common way to pass a value of class type to a routine.

void f(X* p)

A pointer to a value of type X is passed to f(). Programmers with extensive C experience often prefer this to using a reference but a very practical reason to pass a pointer is to accommodate the possibility that for a given call, there is no X, clearly indicated by passing a null pointer.

void f(X value)

This is traditional call by value, just like in C. value is put on the stack using a copy constructor. One (thin) reason that this form might be preferred over f(const X&) is that it essentially guarantees that f() can't change the parameter value in the calling routine. That's the intention of the f(constX&) form, but it can be bypassed with casting.

Problem 5: (5 points)

The following paragraph is written using terms that a Java programmer might use. Strike out those Java-centric terms and write in the terms that a C++ programmer would most likely use.

A common example for teaching inheritance is a hierarchy of geometric shapes. A class named Shape has ~~subclasses~~ derived classes like Rectangle and Circle. ~~Methods~~ member functions such as getArea() and getPerimeter() provide good examples of ~~abstract~~ pure virtual methods. To see if students are thinking, an instructor will sometimes propose ~~fields~~ data members in Shape such as length, width, and radius. Some examples include a class Square, with Rectangle as its ~~superclass~~ base class, but by the Liskov Substitution Principle that is clearly a bad idea.

[A common error was omit "pure" in "pure virtual", equating "virtual" with "abstract".

Here's an interesting note about terminology: Stroustrup says that "A virtual member function is sometimes called a *method*."

If you're curious about the Liskov Substitution Principle, and why a square is-NOT-a rectangle, see www.objectmentor.com/resources/articles/lsp.pdf. They've got a lot of other good articles, too!]

Problem 6: (5 points)

Which element of C++ did you find the most difficult to understand? What did you ultimately find to be the key to understanding that element?

Almost any answer is acceptable as long as it is well written.

Problem 7: (4 points)

What is meant by the term "type extensibility"? Name two elements of C++ that support type extensibility.

Type extensibility is the ability to define new types that are as easy to use and as relatively efficient as built-in types such as int. Constructors, destructors, overloaded operators, and conversion operators are some of the elements of C++ that support type extensibility.

Problem 8: (5 points)

In the style of an e-mail message to a friend who knows C and Java, briefly describe one of the following features of C++: (1) destructors, and the guarantee of destruction, (2) operator overloading, (3) IO Streams, (4) templates, (5) multiple inheritance. **IMPORTANT: Your "message" should include some examples of code.**

Almost any answer is acceptable as long as it is well written and includes some good examples of code.

Problem 9: (3 points)

Write a class named Circle *such that this program*

```
int main()
{
   Circle c(7);

   cout << c.radius() << endl;
   c.radius() = 11;
   cout << c.radius() << endl;
}
```

produces this output:

```
7
11
```

```
class Circle {
   public:
      Circle(int r) : itsR(r) {}
      int& radius() { return itsR; }
   private:
      int itsR;
      };
```

Problem 10: (5 points)

Write a program that prints the integers from 1 through 1000, one per line. **RESTRICTION: Your solution may not use any control structures (if, while, for, etc.), goto statements, local variables, or employ recursion.**

```
class X {
   public:
      X() { cout << ++theCount << endl; }
   private:
      static int theCount;
      };

int X::theCount = 0;

int main()
{
   new X[1000];  // alternative: could be a global
}
```

[There wasn't a lot of middle ground on the scores for this problem—you either saw it or not. A common minor mistake was to create a local array, like `X a[1000];`. That resulted in a half-point deduction. I don't consider `new X[1000]` to be a leak because the operating system will reclaim the memory when the program exits (immediately after the allocation), but if you were worried about that you could use a global or `new (delete X[1000])`.

As a matter of practice I hesitate to pose exam questions like this, which leave you stuck if you just don't see it, but I feel that a small question or two of this sort is reasonable.]

Problem 11: (10 points)

(1) Overload `string & string` *to cause the value of the right-hand operand to be appended to the left-hand operand, changing the left-hand operand. (This operation is imperative, not applicative.)*

(2) Overload `~string` *to produce a copy of the string wrapped in angle-brackets.*

```
string& operator&(string& lhs, const string& rhs)
{
```

```
    lhs = lhs + rhs;
    return lhs;
}

string operator~(const string& s)
{
    return '<' + s + '>';
}
```

Problem 12: (10 points)

Create a templated class named Pair *whose instances are an aggregate of two values of possibly different types. For example,*

Pair<int, string> p1(10, "ten");

creates a Pair *named* p1 *that holds the* int *10 and the* string *"ten".*

```
template <typename A, typename B>
class Pair {
    public:
        Pair(const A& a, const B& b) : itsA(a), itsB(b) {}
        A getA() const { return itsA; }
        B getB() const { return itsB; }
    private:
        A itsA;
        B itsB;
    };

template<typename A, typename B>
ostream& operator<<(ostream& o, const Pair<A,B>& p)
{
    o << "<" << p.getA() << "," << p.getB() << ">";
    return o;
}
```

Problem 13: (15 points)

In this problem you are to define a three-class inheritance hierarchy...

```
class Container {
    public:
        Container(int serial) : itsSerial(serial) { }
        virtual int getSerial() const { return itsSerial; }
        virtual int getVolume() const = 0;
        virtual bool canHoldFluid() const = 0;
        static int fluidVolume(Container* containers[ ]);
    private:
        int itsSerial;
```

```
};

class Bucket: public Container {
   public:
      Bucket(int radius, int height, int serial)
      : Container(serial), itsR(radius), itsH(height) { }
      int getVolume() const { return (int)M_PI*itsR*itsR*itsH; } // truncates double ...
      bool canHoldFluid() const { return true; }
   private:
      int itsR, itsH;
      };

class Box: public Container {
   public:
      Box(int length, int width, int height, int serial, char type)
      : Container(serial), itsL(length), itsW(width), itsH(height),  itsType(type) {}
      int getVolume() const { return itsL * itsW * itsH; }
      bool canHoldFluid() const { return itsType == 'm'; }
   private:
      char itsType;
      int itsL, itsW, itsH;
      };

int Container::fluidVolume(Container* containers[ ])
{
   int total = 0;

   for (int i = 0; containers[i] != 0; i++) {
      Container *cp = containers[i];
      if (cp->canHoldFluid())
         total += cp->getVolume();
      }

   return total;
}
```

Problem 14: (5 points)

... At each of the points of execution that are marked with a series of dashes, enter the number of instances of X that exist at that moment. ...

```
int main()
{
    X x1;
    - - - - - - - - - -  Stack:  1, Heap:  0
    X *p = new X[3];
    - - - - - - - - - -  Stack:  1, Heap:  3
    f(p);
    - - - - - - - - - -  Stack:  1, Heap:  3
}
```

```
void f(X *arg)
{
    X p(*arg);
    - - - - - - - - - -  Stack:  2, Heap:  3
    delete arg;
    X *xp = 0;
    delete xp;
    - - - - - - - - - -  Stack:  2, Heap:  2
    X a[3];
    a[1] = a[2];
    - - - - - - - - - -  Stack:  5, Heap:  2
    X **a2 = new X*[5];
    - - - - - - - - - -  Stack:  5, Heap:  2
    for (int i = 1; i <= 3; i++) {
        X x1;
        a2[i] = new X;
        }
    - - - - - - - - - -  Stack:  5, Heap:  5
    delete a2[1];
    delete a2[2];
    - - - - - - - - - -  Stack:  5, Heap:  3
    delete [] a2;
    - - - - - - - - - -  Stack:  5, Heap:  3
}
```

[Despite checking repeatedly when writing the problem I still ended up with 18 blanks instead of the desired 20! Each answer was worth 1/4 point; 1/2 point was added to compensate for my miscount.]

Extra Credit Section (1 point each unless otherwise noted)

(1) What is "cfront"?

cfront, still available on lectura as CC, was the first widely used C++ compiler. It is written in C++ and generates C code, which is turn compiled to produce a machine code executable.

(2) Name a C++ reserved word that really isn't needed, and describe how a C++ programmer could work around its absence. (Ignore the problem of existing code that might use it.)

'class' is not really needed. See slide 19.

(3) A person telling a C++ programmer about Java says this: "Think of it this way: In Java, they are always explicit." What is the "they" that's being referred to? Answer: Constructors

(4) Offer a GOOD definition of the term "memory leak".

A memory leak occurs when allocated memory is not freed relatively soon after it is no longer of use. (But I can't say I'd call that a GOOD definition...)

[Despite asking for a "GOOD" definition, this problem was scored with great amount of latitude. The essential element required to earn credit was to indicate that a leak occurs when memory still allocated but no longer being used; saying it was memory that was "allocated but never freed" was insufficient.]

(5) What do the # and ## preprocessor operators do?

In a preprocessor macro the unary # operator produces a literal string corresponding to the value of its operand. See slide 155.

When writing the exam I failed to double check that ## was mentioned somewhere. It is not, so answering for only # is sufficient for full credit. ## concatenates tokens.

[A number of students produced an answer regarding #include, but # is not an operator in that context.]

(6) Why did Stroustrup choose the terms "base class" and "derived class" instead of "superclass" and "subclass"?

He found the terms superclass and subclass to be confusing.

(7) How many lectures were there?

15

(8) Answer any one of these three:

At what university did Stan Lippman earn a graduate degree in English? The University of Arizona

Who had the first Original Thought of the semester? Mr. Rooks

Who asked for coffee and doughnuts for the final exam? Mr. Somasundaram

[Just about everybody remembered that it was somu who put in for the coffee and doughnuts! I again apologize that things didn't work out on the coffee—I didn't imagine one should call ahead for a keg of coffee at Krispy Kreme!]

(9) (3 points) For problem 14, which asks you to calculate the number of instances existing at various points in time, the instructor wrote a program, docounts, that takes a C++ program as an input file and produces a listing essentially identical to that shown in problem 14 but with the counts filled in. The C++ code did contain an indication of each of the points where the counts were to be calculated.

(1) The constructors and destructor for the class at hand (X) need to determine where the object resides based on the value of this and maintain totals for the stack and heap counts. (2) At the points where a count is desired call X::showCounts(), which displays the totals. Write a little program in Icon, or something similar, like Python, that compiles the C++ source file and then runs it, collecting the output lines produced by showCounts(). Then read the source file and replace the calls to showCounts() with the collected output lines.

(10) (4 points) Consider the following header file, Rectangle.h ...

This is an exact recycle of problem six on assignment five, which only two students got right. If you took the time to understand the assignment five solutions you hopefully got the problem right on this second attempt. The point value was chosen to exactly balance the points lost on the assignment five problem, which, based on the poor results the first time around, wasn't a very good problem.

[My attempt to make amends didn't work out very well—on the exam, just like on the assignment, two students got it right, but it wasn't the same two students. Quite a few students responded with the same wrong answer they had put forth on the assignment.]