

# CS 211 Winter 2004

## Sample Final Exam (Solutions)

Instructor: Brian Dennis, Assistant Professor  
TAs: Tom Lechner, Rachel Goldsborough, Bin Lin

March 16, 2004

### Your Name:

There are 6 problems on this exam. Each one is worth 20 points. They are not all of equal difficulty. Please read all of them before starting and attempt the least difficult first.

You have 120 minutes to complete the exam. **When time is up, please stop writing and turn in your quiz. If you have to be asked to stop you will receive no credit for the question you are working on.**

Problem 1	
Problem 2	
Problem 3	
Problem 4	
Problem 5	
Problem 6	
Total	

### Problem 1

**10 Points.** Explain why a static member function of a class shouldn't try to access a `this` variable.

Static member functions are intended to be class wide and not specific to a particular instance. Thus, they actually never get a `this` pointer. Most compilers will catch this as a syntax error.

**10 points.** Briefly give and explain one positive aspect and one negative aspect of operator overloading in C++.

Overloaded operators are good in that we can use standard, terse, operator syntax for operations on datatypes that it makes sense. For example, manipulations on a matrix datatype might be done using overloaded operators as opposed to verbose member function calls.

The downside of overloaded operators is that they don't actually have to conform to the implied semantics of the operator. For example, we could have a `+` operator that implements subtraction. Overloaded operators provide an easy opportunity to write obfuscated code.

## Problem 2

```
-----  
// classes.h  
class Base {  
public:  
    Base();  
    virtual void howdy();  
    void doody();  
private:  
    // ...  
};  
  
class Derived : public Base {  
public:  
    Derived();  
    virtual void howdy();  
    void doody();  
private:  
    //...  
};  
-----  
  
// base.cpp  
#include <iostream>  
#include "classes.h"  
using namespace std;  
// ...  
void Base::howdy() {  
    cout << "Let's get ready to CS rumble!" << endl;  
};  
  
void Base::doody() {  
    cout << "I think I'm going to finish early." << endl;  
};  
-----  
  
// derived.cpp  
#include <iostream>  
#include "classes.h"  
using namespace std;  
// ...  
void Derived::howdy() {  
    cout << "I hope you were the groom!" << endl;  
}  
  
void Derived::doody() {  
    cout << "Ted Nugent called. He wants his shirt back." << endl;  
}
```

Assuming the code on the previous page, for each of the numbered lines below indicate what is printed out (2 points) and give a brief explanation (3 points).

```
// -----
```

```
int main(int argc, char **argv) {  
    Derived dr;  
    Base& br = dr;  
    Base* bp = new Derived;  
  
    dr.howdy(); // 1  
    br.doody(); // 2  
  
    bp->howdy(); // 3  
    bp->doody(); // 4  
  
}
```

```
I hope you were the groom! // 1  
I think I'm going to finish early. // 2  
I hope you were the groom! // 3  
I think I'm going to finish early. // 4
```

### Problem 3

On the next few pages is code for an integer `IntSet` class that is derived from `vector<int>`. Write 4 member functions for the class.

- A new constructor that takes an array of integers and an integer count, and initializes the `IntSet` to hold the integers in the array
- A union member function that takes another `IntSet` and returns a new `IntSet` which has the elements of both sets.
- An intersection member function that takes another `IntSet` and returns a new `IntSet` which has all elements which are in both sets.
- A difference member function that takes another `IntSet`  $s$  and returns a new `IntSet` which contains the elements that don't appear in  $s$

```
// Since arrays and pointers are interchangeable, the preexisting
// IntSet::IntSet(int* vals, int cnt) constructor is fine.
```

```
IntSet IntSet::set_union(IntSet is) {
    IntSet newSet(*this);
    // Can't use bracket operator since we overloaded it
    for (int i = 0; i < is.size(); i++) newSet.add(is.at(i));
    return newSet;
}
```

```
IntSet IntSet::set_intersection(IntSet is) {
    IntSet newSet;
    vector<int>::iterator ix = is.begin();
    while (ix < is.end()) {
        if (is_member(*ix)) newSet.add(*ix);
        ix++;
    }
    return newSet;
}
```

```
IntSet IntSet::set_difference(IntSet is) {
    IntSet newSet(*this);
    for (int i = 0; i < is.size(); i++) {
        if (is_member(is[i])) newSet.del(is[i]);
    }
    return newSet;
}
```

```

#ifndef _INT_SET_H_
#define _INT_SET_H_ 1

#include <iostream>
#include <vector>
using std::vector;
using namespace std;

// This class inherits from the template class
// vector<int> so it's an ordered collection of integers
// vectors are optimized for adding new elements to the end.

class IntSet : public vector<int> {
public:
    // Create an empty set
    IntSet();

    // Initialize a set from, an array of integers
    IntSet(int* ints, int cnt);

    // Test whether i is in the set
    bool is_member(int i);

    // Add an integer to the set
    void add(int i);

    // Delete an integer from the set
    void del(int i);

    // Test whether idx is in the set
    bool operator[](int idx);

    friend ostream& operator<<(ostream& os, IntSet& is);
private:
};

ostream& operator<<(ostream& os, IntSet& is);
#endif

```

```

#include <iostream>
using namespace std;

#include "IntSet.h"

// Nothing really exciting here, other than using
// submember initialization
IntSet::IntSet()
    : vector<int>()
{ }

// Since I'm a vector<int> and I know that the vector<int>
// part of an IntSet has already been initialized, it's safe
// to call the push_back member function
IntSet::IntSet(int* ints, int cnt) {
    for (int i = 0; i < cnt; i++) {
        if (!is_member(ints[i])) {
            push_back(ints[i]);
        }
    }
}

// Again, an example of using inherited member functions directly
// size, and at which are defined in vector<int>
bool IntSet::is_member(int i) {
    for (int j = 0; j < size(); j++) {
        if (i == at(j)) return true;
    }
    return false;
}

void IntSet::add(int i) {
    if (!is_member(i)) {
        push_back(i);
    }
}

// vectors provide iterators to iterate sequentially
// through their elements. You can also delete elements
// from the collection using an iterator
void IntSet::del(int i) {
    vector<int>::iterator ix = begin();
    while (ix < end()) {
        if (*ix == i) { erase(ix); return; }
        ix++;
    }
    return;
}

```

```
// This can't be used on the left hand side of an assignment
// since it doesn't return a reference type.
bool IntSet::operator[](int idx) {
    return is_member(idx);
}

// This doesn't actually have to be a friend of the
// IntSet class since it doesn't use any private members
ostream& operator<<(ostream& os, IntSet& is) {
    os << "BitSet[";
    for (int i = 0; i < is.size(); i++) {
        os << is.at(i) << " ";
    }
    os << "];";
    return os;
}
```

#### Problem 4

```
#ifndef _CONS_H_
#define _CONS_H_ 1
#include <string>
using namespace std;
class OurString {
public:
    // added
    friend ostream& operator<<(ostream& os, OurString& ostring);

    OurString(char* null_string);
    OurString(char c);

    int len();
    int set_char(int index, char c);
    OurString reverse(); // added

private:
    string s; // int buf_cnt;    // char* buf;
};
ostream& operator<<(ostream& os, OurString& ostring); // added
#endif
```

**10 points.** The above `OurString` class is intended to represent a sequence of characters that can hold any character including `'\0'`. `buf_cnt` represents the number of characters currently in the string. `buf` is completely private storage of an `OurString` instance's characters.

Rewrite the class so that it uses the builtin `string` class in a “has-a” relationship. Change the class declaration and write implementations for all of the declared member functions.

```
#include "OurString.h"

OurString::OurString(char* null_string)
    : s(null_string)
{ }

OurString::OurString(char c)
    : s()
{ s += c; }

int OurString::len() { return s.size(); }

int OurString::set_char(int index, char c) {
    char old_c = s[index];
    s[index] = c;
    return old_c;
}
```

**10 points.** Add two functions to the `OurString` class. A `reverse` function that returns a reversed `OurString` and an overloaded output operator for `OurStrings`.

```
OurString OurString::reverse() {
    OurString new_string("");
    for (int i = s.size() - 1; 0 <= i; i--) {
        new_string.s += s[i];
    }
    return new_string;
}

ostream& operator<<(ostream& os, OurString& ostring) {
    os << ostring.s;
    return os;
}
```

### Problem 5

**10 points** Give two reasons, with brief explanations, why polymorphic functions are better than using switch statements in classes.

One, programmers don't have to hunt down and change the source code for the switch statement if a new type of object is to be handled by the dynamic dispatch supported by polymorphic functions. In some cases, the programmer may not even have the source code to change.

Second, using switches tightly couples the derived and base classes, since the base class has to know about the derived classes. With polymorphic functions, the base class defines an interface, but doesn't have to know of any derived classes that implement it.

**10 points** Briefly explain what a pure virtual function is and why they're important. Also, give a brief code example of declaring one.

A pure virtual function is declared like so:

```
class Foo {  
    ...  
    void virtual foo_function() = 0;  
    ...  
}
```

This forces the class `Foo` to be an abstract base class. No instances of this class can be created. Derived classes can be declared and instances of these classes created if they override and implement `foo_function`. The pure virtual function forces derived classes to support a particular procedural interface.

### Problem 6

For each of the following statements indicate whether it is true or false (2 points) and give a brief explanation (2 points).

By default, C++ member functions are polymorphic.

**False.** We have to use the `virtual` keyword to get polymorphic behavior for member functions.

Only pointers allocated using `new` can be `deleted`

**True.** `delete` reclaims space dynamically allocated from the heap. Pointers to stack objects or global objects are automatically managed by the C++ runtime.

Template functions have to be member functions.

**False.** We could write a standalone templated version of `square` that operated on arithmetic types.

A compiler analyzes a program, builds an internal representation of the program, and immediately executes the program's actions.

**False.** An interpreter executes the actions immediately. A compiler takes the internal representation and generates binary representation to be executed later.

In C++ we can only overload binary operators

**False.** We can overload some unary operators such as `!`.