

COMPUTER SCIENCE 51

Midterm II (C++)

80 minutes/70 points

Fill in your name and logname below.

Name

FAS Username

TF

There are five sets of questions on this test. We have left a great deal of extra space that you may use in any way you choose. You need not feel compelled to fill in the entire space—be concise. Additionally, please make sure that the answer you wish us to grade is clearly indicated.

The problems are weighted according to how much time we think they should take (1 point per minute); budget your time accordingly. If all goes well, you should have 10 minutes at the end to check your answers.

Problem	Question	Total	Actual
1	Short Answer	20	
2	Lexing, Parsing, and Evaluating	10	
3	Fixing a Flawed Design	20	
4	Implementing a Stack	10	
5	Reference Counting Pointers	10	
Total		70	

Problem 1: Short Answer (20 points)

A. (2 points)

What is the advantage of passing by const reference rather than passing by value?

B. (2 points)

In the guest lecture, Walter Fontana discussed how biological facts could be captured by grammars. What does a “reachable state” mean, given some grammar?

C. (2 points)

What does it mean to “overload an operator”? Why is it useful?

D. (2 points)

Explain with a line of code how you would overload an operator.

E. (2 points)

What is a pure virtual function (explain in one sentence)?

F. (2 points)

Show how to declare a pure virtual function.

G. (4 points)

Consider the following code:

```
#include <iostream>
using namespace std;

class C {
public:
    C() { cout << "c"; }
    C(const C &other) { cout << "p"; }
    ~C() { cout << "d"; }
};

void f1(const C &c) {
    // do nothing ;)
}
void f2(C c) {
    // do nothing :-D
}
```

Suppose you have an instance of the class, called `c`. Will this code have a different output when `f1(c)` is called, as opposed to `f2(c)`? If so, explain the difference in output. If not, explain why they are the same.

Suppose you had the following main function:

```
int main(int argc, char *argv[]) {
    C *p = new C();
    C c;
    return 0;
}
```

What would this code print?

H. (4 points)

Consider the following code:

```
#include <iostream>
using namespace std;

class Counter {
private:
    static int count;
public:
    Counter() { ++count; }
    ~Counter() { --count; }
    static int getCount() { return count; }
};

int Counter::count = 0;

class MyCounter1 : public Counter { };
class MyCounter2 : public Counter { };

int main()
{
    Counter a;
    cout << Counter::getCount() << endl;

    MyCounter1 b;
    cout << MyCounter1::getCount() << endl;

    MyCounter2 c;
    cout << MyCounter2::getCount() << endl;

    return 0;
}
```

What does this code output?

Explain your answer in at most one or two sentences.

Problem 2: Lexing, Parsing, and Evaluating (10 points)

A. (6 points)

You may not know this, but Google has a calculator built-in to its search functionality. For instance, if you search for:

$$1 + 2$$

Google will come back with its calculator and give you

$$3$$

However, if you search for things like:

$$\text{CS51 C++} \\ 1 + 2 +$$

Google will figure out that these are not math equations and will instead return the results of a web-search with these phrases.

For these two questions, consider the problem of using a lexer, parser, and evaluator to determine if a given phrase is a mathematical equation or not, and if it is, evaluating it. Assume that you can only accept equations with numbers, and the basic operators $+$, $-$, $*$, and $/$.¹ You would like to determine if a phrase is an equation **as soon as possible**, that is, you would rather determine that a phrase is not a valid equation in the lexer rather than in the parser, and in the parser rather than the evaluator.

(a) Consider the search query: CS51 C++. Should you use the lexer, the parser, or the evaluator to determine that this is an invalid calculation? Explain in one sentence.

(b) Consider the search query: 1+2+. Should you use the lexer, the parser, or the evaluator to determine that this is an invalid calculation? Explain in one sentence.

-

¹In fact, Google calculator will do fancier things like unit conversion and checking for constants. For instance you can search “c * 10s” to get the speed of light times 10 seconds, or “the answer to life, the universe, and everything + pi” for $42 + 3.14$. Don’t worry about this for this problem.

(c) Write an example of a phrase that is an invalid calculation but which must be caught in the evaluator. That is, neither the lexer nor the parser would be able to tell that this is an invalid calculation.

B. (4 points)

Recall that the Scheme special form “if” works as follows:

```
(if <cond> <true branch> <false branch>)
```

Say you want to reject Scheme values of the form (if #t 2 3 4) or (if #t 2) [i.e., too many or too few arguments to the if statement].

(a) Explain briefly how you could catch these errors in the parser. What would you need to do to be able to do this?

(b) You could also catch this error in the evaluator by noticing that `if` is a procedure that needs to take exactly 3 arguments (and then throwing an error when you try to apply the `if` procedure to the arguments). Discuss one pro and one con of catching the error in the parser instead of the evaluator.

Problem 3: Fixing a Flawed Design (20 points)

Your friend John is working on an idea for a brilliant new video game. He calls it *Pokomon*, and he's sure that it's going to catch on like wildfire. But before that can happen, he needs to get his code, well... *working*. Knowing what a skilled coder you are, he's asked you to give him a little nudge in the right direction.

The game revolves around creatures known as Pokomon. Pokomon are animals that are capable of fighting one another using magical elemental-based attacks.

Each Pokomon is associated with one or more elemental "types". Much like a game of *Rock-Paper-Scissors*, the type of a Pokomon helps determine how it will fare in battle against another Pokomon. For example, a water-type attack would be super-effective against a fire-type Pokomon. Likewise, a ground-type attack would be quite weak against a flying Pokomon. (Note that the "effectiveness relationship" is not necessarily symmetrical: a flying-type attack might not be super-effective against a ground-type Pokomon.)

Currently John has thought of seven different types: fire, water, grass, earth, flying, electric, and normal. He has implemented the following class called `ETable` ("effectiveness table") that stores all the information about different types' relative effectiveness:

```
class ETable
{
private:
    // Here are the types I've come up with. I've thought long and
    // hard about them, so I probably won't want to add any more to the
    // list.
    enum type_t {FIRE, WATER, GRASS, EARTH, FLYING, ELECTRIC, NORMAL};

    // The first array index represents the attacker's type, the second
    // one represents the defender. The value is true if the attack is
    // effective, and false if it is not. It's const because I don't
    // ever want the values of the table to change.
    const bool table[7][7];

public:
    ETable() { }
    // Obviously, my destructor needs to delete the memory I've allocated
    // for my table.
    ~ETable() { delete table; }

    // This function loops through table to fill in the effectiveness
    // relationships I've painstakingly developed by hand. It returns
    // a pointer to the table I've just populated.
    ETable *populateTable();

    // Is an attack of type a effective against a defender of type b?
    bool isEffective(type_t a, type_t b);
}
```

A. (8 points)

Unsurprisingly, he's having a lot of trouble just getting his code to compile. Can you help John out? Explain four (4) errors or poor design decisions in this class implementation, and briefly (no more than a sentence) describe how you would fix them.

Now that you've worked on ETable, John wants to show you how he's implemented the Pokomon themselves. He explains:

"I know my design is good because I've used this fancy inheritance stuff I keep hearing about. But I'll run it by you just so you can marvel at my genius."

```
class Pokomon
{
private:
    int health;
    string name;    // allowed to give nicknames to Pokomon
    vector<attack_t> attackList;    // attack_t is a struct type defined elsewhere

    /* more private data fields [not shown] */

public:
    // the two virtual methods that follow have dummy implementations that will be
    // overwritten by inherited classes

    // return all types that the current Pokomon is weak against when defending
    virtual vector<type_t> getWeaknesses() { vector<type_t> dummy; return dummy; }

    // take in an opponent to attack; return the amount of damage dealt
    virtual int attack(string attackName, const Pokomon *opponent) { return -1; }

    void takeDamage(int d) { health -= d; }
}
```

```
class FireType : public Pokomon
{
public:
    getWeaknesses(); // implemented in .cc file
}
/* Other classes called FlyingType, NormalType, WaterType, etc. also extend
 * the Pokomon base class */

// This class has static stuff because Snarlox does not move around very much.
class Snarlox : public NormalType
{
private:
    static bool isAsleep;

public:
    static void wakeMeUp() { isAsleep = false; }
}

// I was very clever and used this thing I found on the internet called 'multiple
// inheritance'. It allows me to have a child class inherit from two different
// parent classes! This makes sense, because Charisaur is both a fire-type AND
// a flying-type Pokemon.
class Charisaur : public FireType, public FlyingType
{
public:
    // do a fire attack against the enemy
    int breatheFire(Pokomon *opponent);
}
```

B. (2 points)

Well, it turns out that John is correct—you *can* have a class inherit from two different parent classes. Unfortunately, something's not quite right with what he's done here. Explain (briefly!) how John's inheritance model introduces ambiguity into the design.

C. (4 points)

How might you make John's inheritance scheme simpler? Consider a new inheritance structure and potentially any new public or private data members. Your answer need not be terribly long—a diagram with a sentence or two of explanation should suffice.

D. (6 points)

Aside from the inheritance structure problem, explain three (3) other errors or poor design decisions in the code presented (specifically, the classes `Pokomon`, `FireType`, `Snarlox`, and `Charisaur`) and briefly (no more than a sentence) explain how you would fix them. Assume that the code seen here is only part of John's implementation. We're looking for problems in the existing code—not simply missing functionality. Also, there may be several places that have the same similar type of error—you should only count this as one error (we'd like to see three different types of errors described here!).

Problem 4: Implementing a Stack (10 points)

This question's task is to implement the `.h` file for a stack data structure and then answer some questions about it. For our purposes, a stack is a templated data structure with `typename T` which provides three functions:

```
// takes obj and puts it on "top" of the stack
void push(T obj);

// removes the "top" element from the stack, and then returns that element
T pop();

// returns the number of things in the stack
int size();
```

Your stack might be used as follows:

```
void foo()
{
    MyStack<int> s;
    s.push(1);
    s.push(5);
    s.push(10);

    s.size(); // returns 3

    s.pop(); // returns 10
    s.pop(); // returns 5
    s.pop(); // returns 1

    s.size(); // returns 0
}
```

In your implementation, you may use any STL container underneath your stack (except for the STL `stack` itself!). It should be reasonably clear to the staff how you plan to implement your `push()`, your `pop()`, and your `size()` functions, so if you think it is non-obvious given your data structures, please write brief comments explaining how those functions will work. (Note that you **DO NOT** need to fully implement the functions—a brief comment is plenty.)

As explained in the interface specification, your class should be templated to allow for different types of data to go on the stack. If you do not remember how to do this, you may write an integer-only version of `MyStack` for partial credit.

A. (6 points)

Write the `.h` file. Do not worry about `#include`ing things or writing using `namespace std;` at the top. Be sure to use good style/design principles.

B. (2 points)

What function might you want to throw an exception from? In what circumstance would you throw that exception?

C. (2 points)

Write the tests cases you'd want to run to convince yourself that `MyStack` works.

Problem 5: Reference Counting Pointers (10 points)

This problem will teach you the basics of reference counting. The ultimate goal for this question is to create a `SmartPointer` class that will automatically free its own memory as soon as the memory is no longer needed.

Before we present any of `SmartPointer`'s code, here is a brief example of how a client would use this `SmartPointer` class. Suppose the existence of a class `Dog`, which has public method, `bark()`, and a public data member, `age`.

```
// here is correct usage of a SmartPointer
void foo()
{
    // here, we create the Dog; notice that we do not keep a handle to
    // the raw pointer around--we are going to entrust the raw pointer
    // to the SmartPointer class, and we will do all accesses to the
    // pointer through our instance of SmartPointer (the variable p)
    SmartPointer<Dog> p(new Dog());

    // now, we can simply use p as if it were a regular pointer:
    cout << p->age << endl;
    p->bark();

    // Use SmartPointer's operator* function to assign a value for *p
    Dog d;
    *p = d;

    // p will fall out of scope here, and its refcount will fall to 0
    // the destructor will thus clean up the pointer, deleting the
    // memory allocated for Dog
}

// here is incorrect usage of a SmartPointer
// while the code would compile, it would cause problems when you
// ran it because it would double-free the p pointer
void incorrect()
{
    Dog *p = new Dog();
    SmartPointer<Dog> sp(p);

    // use p

    delete p; // wrong because SmartPointer is going to delete p!
}

```

`SmartPointer` is basically just a wrapper class that will automatically destruct the `Dog` object, as soon as the `SmartPointer` instance goes out of scope.

Now, let's look at some of the code for such a `SmartPointer`. Bear in mind while reading the code that this code has a subtle bug (addressed in part a) and an unimplemented function (addressed in part b).

```
// A simple counter abstraction that will be useful for counting references.
class RefCount {
public:
    RefCount() {
        count = 0;
    }
    void inc() {
        count++;
    }
    int dec() {
        return --count;
    }

private:
    int count;
};

// Here is the SmartPointer. Notice that it is templated, so it can
// support any pointer type. Pay special attention to where it
// increments and decrements the reference count to make sure you
// understand what is going on in the class.
template<typename T>
class SmartPointer
{
public:
    SmartPointer() {
        p_data = NULL;
    }

    SmartPointer(T *p) {
        p_data = p;
        ref_ctr.inc();
    }

    SmartPointer(const SmartPointer &other) {
        p_data = other.p_data;
        ref_ctr = other.ref_ctr;
        ref_ctr.inc();
    }

    SmartPointer<T>& operator=(const SmartPointer &other) {
        // todo!
    }
}
```

```
T& operator*() const {
    return *p_data;
}

// return the pointer address here (this provides the expected
// behavior for the -> operator)
T* operator->() const {
    return p_data;
}

~SmartPointer() {
    delete_helper();
}
private:
void delete_helper() {
    if(ref_ctr.dec() <= 0) {
        delete p_data;
    }
}

T *p_data;
RefCount ref_ctr;

// disallow assignment to a raw pointer (force them to use a constructor)
SmartPointer<T>& operator=(T *);
};
```

A. (3 points)

The code will not work as it is written. What is the problem with the current implementation? How might you fix it? Explain the general approach of your fix—you don't need to discuss every last detail.

Generous hint: look at the way the `ref_ctr` variable is declared in `SmartPointer`—why is this bad? You may also want to consider this test case:

```
SmartPointer<Dog> p1(new Dog());  
SmartPointer<Dog> p2(p1);
```

The count in `p1`'s `RefCount` variable will not be set properly, because of our bug.

B. (5 points)

This code is missing `operator=` for the `SmartPointer` class. Right now, when you have two pointers:

```
SmartPointer<Dog> p1(new Dog());
SmartPointer<Dog> p2(new Dog());
```

and you say:

```
p2 = p1;
```

the reference count will not be updated properly, since we have not provided an implementation for `operator=`. Your task for this part is to write a working `operator=` function. You will want to keep in mind the following while doing this:

1. You need to properly handle the case where you are trying to assign a smart pointer to itself (eg., `p1 = p1`). You may want to remember that the keyword `this` is a pointer to the current instance of the class. This may be useful for detecting when `other` is the same variable as the current class you're in.
2. Your function should return `*this` (an instance of the current object)—this is a standard idiom for an assignment operator such as `operator=`.
3. In the normal case (`p1 = p2`), you will want to delete your current state (and adjust the reference count appropriately), and then copy over `other`'s fields. When you copy over `other`'s fields, you should increment the reference counter appropriately. Note that you are able to access `other`'s private data members because you are within the class.

```
SmartPointer<T>& operator=(const SmartPointer &other)
{
```

```
}
```

C. (2 points)

The benefits of using `SmartPointer` are obvious. Name one downside to using a `SmartPointer`.

Scratch Paper—turn in with exam.