Math 211 Midterm
March 23, 2000
Professor L. Taylor

Name: ____

1. Consider the following bit of code

```
if( test_1 ) {code_1 }
else if( test_2 ) {code_2 }
else if( test_3 ) {code_3 }
else {code_4 }
code_after
```

If $test_1$ and $test_3$ are false, discuss the two ways the program can behave at this point. What happens if $test_1$ remains false, but $test_3$ is true?

The two possible cases are that $test_2$ is true or that it is false. If the **program** determines that $test_2$ is true, then the program does $code_2$, followed by $code_{after}$ **unless** $code_2$ contains a `return` or a `break` or other such statement, in which case you cannot really tell what happens without a more detailed examination of the code. If $test_2$ is false then we do $code_4$ followed by $code_{after}$ **unless** $code_4$ **....**

We have the same two cases for the second part of the question and if $test_2$ is true, we again do $code_2$, followed by $code_{after}$ **unless** $code_2$ **....** But if $test_2$ is false, then we do $code_3$, followed by $code_{after}$**unless** $code_3$ **....**

Notice that it is the running program that determines what happens and if we make several passes through this bit of code during the course of the program, the truth value of the test's may change. The ability to talk (or write) your way through bits of code depending on assumptions is an important skill. It comes up in writing the code and especially in debugging.

2. Define a function `S` with the following code. Assume `S` has been declared correctly.

```
short S(short x, short y) {
return(x-2*y); }
```

Discuss the possible values in the `x` and `i` boxes after the following two statements have executed:

```
i=3; x=S(++i,i--);
```

Several people brought up **precedence** with `x-2*y`. Precedence can be a source of errors, but the language specifies the precedence so it will work the same on all compilers. YOU (or I) may not remember the precise rules (which is why parentheses are a good idea) but the compiler does. For arithmetic expressions, all of us usually get the precedence right even if we'd rather not have to write out what the rules are exactly. Here precedence is a red herring.

There are also possible **scope** issues which are again irrelevant in this example. The `x` in `x=S(++i,i--);` has nothing to do with the `x` in the function `x` **because** we start the code for the function with `short S(short x, short y)` so inside the function `S`, `x` is the name for the first variable and `y` is the name for the second.

The real issue here is that the language does not specify the order in which the variables are evaluated. This is extremely insidious. As an example, nearly everyone in the class behaved as though evaluation **must** go from left to right, but it **need** not. C does however guarantee that once the program begins to evaluate a variable it will finish that evaluation before moving onto the evaluation of the next.

Once we see the problem, there are two cases, either we evaluate the first variable first (and then the second) or else we evaluate the second variable first (and then the first). C does not even guarantee left-to-right OR right-to-left, so when the function has three variables, there are 6 possibilities. Question: How many cases are there when the function has 5 variables?

Suppose we evaluate the first variable first. Then we first do `++i` which puts a `4` into the `i` box and sets the first variable of the function to `4`. Then we evaluate the second variable. There is a `4` in the `i` box, so the second variable is also `4` and then we put a `3` into the `i` box. Hence the `x` box contains `S(4,4)` which is `-4` and the `i` box contains `3`.

In the other case, we begin by evaluating the second variable. This sets the second variable to `3` and puts a `2` into the `i` box. Then we evaluate the first variable. This first puts a `3=2+1` into the `i` box and then sets the first variable to `3`. Hence the `x` box contains `S(3,3)` which is `-3` and the `i` box contains `3`.

This order issue is not limited to functions. With `i=3` still, think about what `x=(++i)+(i--);` might be.

3. Declare `xx` and `dd` as `char *xx, dd;`. After some code executes, `xx` points to the box with the `'A'` in it, with the next few memory locations filled as indicated.

'A'	'a'	'r'	'd'	'v'	'a'	'r'	'k'	'\0'

Discuss what is in the `dd` box after each of the statements

a. `dd=*(++xx);`
b. `dd=++(*xx);`
c. `dd=*(xx++);`

What is in the box pointed to by `xx` after each of these statements?

In addition to the nine boxes shown, there are two more relevant boxes. Somewhere there is a box called `xx` which holds the machine address of the box containing the `'A'`. This box is usually the size of four `char` boxes, but its size is irrelevant to our discussion. The second box is a `char` box named `dd`. (If I want `dd` to also be a `char` pointer, then I have to write `char *xx, *dd;`.) The question concerns the values in these two boxes after each of the 3 statements above executes.

One point you need to bear in mind is that every time you do a `++` or `--` the value in some box gets changed - you need to figure out which box and when the change is made. A second point deals with precedence and parentheses. Since we have parentheses, we work from the inside out.

Let's begin with a. Because of the parentheses, we begin by incrementing `xx`, so the value in the `xx` box is incremented by `1`. Since it is a `char` pointer, it now points to the next box in memory which is the first box containing the character `'a'`. Then we take that value and put it in the `dd` box so it now contains `'a'`. This answers both questions for case a.

For b, we first take the character in the box pointed to by `xx`, which is `'A'`, and then we increment the value and put it back in the box pointed to by `xx`. This time it is now the box pointed to by `xx` that has its value changed. Our bit of memory now reads

'B'	'a'	'r'	'd'	'v'	'a'	'r'	'k'	'\0'

The `xx` box points to the `'B'` and we put that value into the `dd` box, so `dd` is now `'B'`.

Finally, we examine c. First we take the value pointed to by `xx` and put that into the `dd` box. Hence `dd` is `'A'`. After we do this, we increment `xx` by `1` so `xx` points to the first box containing the character `'a'`.

4. Consider the following bit of code:

```
xx=2;
switch(cc){
case 'a': xx+=3;
case 'b': xx%=3; break;
case 'c': xx-=4;
default: xx=0;
}
```

What value will `xx` have if `cc='a'`? How about if `cc='c'`?

No real issues here, just being careful. If `cc` is `'a'` then we execute the code after the `case 'a':`. In other words, we increment the value in `xx` by `3`. Since the value in `xx` was `2` to begin with, it is now `5`. Then, because there is no `break;` we drop through and do the code after the `case 'b':`. This sets `xx` to the remainder of what is in there now, after division by `3`. Since there is a `5` now, after this statement, `xx=2`. Then we continue execution immediately after the `}` closing the `switch` since there is a `break;` after the `xx%=3;`.

If `cc='c'` then we begin with the code immediately after the `case 'c':` which does `xx-=4;` so `xx` now has the value `2-4` or `-2`. But again there is no `break;` so we fall through to the `default:` code and this just puts a `0` into the `xx` box.

5. Consider the sequence defined recursively by $a_n = a_{n-1} + a_{n-2} + a_{n-3}$; $a_1 = 1$, $a_2 = 2$ and $a_3 = 3$. Write the code for a function with declaration `short A(short n);` which returns the `n`th term in the sequence, $n \geq 1$, and returns `-1` otherwise.

There are lots of ways to do this. They all use **recursion** but there are lots of ways to write the non–recursive part. Here are some possibilities

1.

```
short A(short n){
if( n>=4) {return( A(n-1)+A(n-2)+A(n-3) ); }
else if( n==3) {return (3); }
else if( n==2) {return (2); }
else if( n==1) {return (1); }
else {return (-1); }
}
```

2.

```
short A(short n){
if( n==3) {return (3); }
else if( n==2) {return (2); }
else if( n==1) {return (1); }
else if( n>=4) {return( A(n-1)+A(n-2)+A(n-3) ); }
else {return (-1); }
}
```

3.

```
short A(short n){
if( n>=4) {return( A(n-1)+A(n-2)+A(n-3) ); }
else if( n>=1) {return (n); }
else {return (-1); }
}
```

4.

```
short A(short n){
if( n>=4) {return( A(n-1)+A(n-2)+A(n-3) ); }
else if( n==1 || n==2 || n==3 ) {return (n); }
else {return (-1); }
}
```

5.

```
short A(short n){
if( n>=4) {return( A(n-1)+A(n-2)+A(n-3) ); }
else {switch(n) {
      case '3': return 3;break;
      case '2': return 2;break;
      case '1': return 1;break;
      default: return -1;break;
      }
   }
}
```

6.

```
short A(short n){
if( n>=4) {return( A(n-1)+A(n-2)+A(n-3) ); }
else {switch(n) {
      case '3': case '2': case '1': return n;break;
      default: return -1;break;
      }
   }
}
```

Any of the `n>=4` tests can be replaced by `n> 3` with the same results. The `n>=4` test in example 2 can be replaced by `n>=3`, `n>=2` or `n>=1`, or even `n> 0`.

There are also versions that can be done without recursion.