Answer Key to Midterm Exam 1

Fall 2013
Time: 50 minutes.

Fri, Oct 04

Total points: 100 plus one extra credit question worth 5 points 7 pages

Please use the space provided for each question, and the back of the page if you need to. Please do not use any extra paper. The space given per question is **a lot** more than sufficient to answer the question. Please be brief. Longer answers do not get more points!

- **No electronic devices of any kind. You can open your textbook and notes**
- **Please leave your UB ID card on the table**
- **This booklet must not be torn or mutilated in any way and must not be taken from the exam room**
- **Please stop writing when you are told to do so. We will not accept your submission otherwise.**
- **If you wanted to, you can answer the extra credit question without answering all of the other questions**

Your name:	_____
Your UBIT Name:	_____

The rest of this page is for official use only. Do not write on the page beyond this point.

Problem Number	Score obtained
name and id (5 max)	
Problem 1 (30 max)	
Problem 2 (30 max)	
Problem 3 (20 max)	
Problem 4 (15 max)	
Extra credit problem (5 max)	
Total Score: (105 max)	

Problem 1 (30 points)**.** Mark the correct choice(s) or give a brief answer. Each question is worth 3 points. All codes are in C++.

1. Which of the following are declarations but **not** definitions? Check all that apply.

   ☐ `int i;`
   ☐ `int i=1;`
   ⊗ `int foo(int);`
   ⊗ `int foo(int i);`
   ⊗ `typedef string my_string;`
   ☐ None of the above

2. Consider the following definition `char var[] = "this";` What is `sizeof(var)`?

   ☐ 4 ⊗ 5 ☐ 6 ☐ 7 ☐ 8

3. Consider the following definition `char var[] = "this\0is";` What is `sizeof(var)`?

   ☐ 4 ☐ 5 ☐ 6 ☐ 7 ⊗ 8

4. Consider the following snippet of C++

```
void ubswap(int **a, int **b) {
    int* temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 1, y=9;
    int* u = &x; int* v = &y;
    int** a = &u; int** b = &v;
    ubswap(a, b);
    return 0;
}
```

   which pairs of variables in `main` are swapped? Check all that apply.

   ☐ `x and y` ⊗ `u and v` ☐ `a and b`

5. Continue with the above snippet, which of the following – when called *after* `ubswap(a, b);` line – will produce `9`? (Check all that apply)

   ⊗ `cout << **a;` ☐ `cout << **b;` ⊗ `cout << *u;` ☐ `cout << *v;`
   ☐ `cout << x;` ⊗ `cout << y;`

6. Suppose you wanted to make use of `Lexer` routines I gave and all your codes are put in `yourprog.cpp`. The interface for for the `Lexer` is declared in `Lexer.h` and the implementation is stored in `Lexer.cpp`, all in the same directory. The `Lexer.h` header is properly included. Which of the following compilation commands will produce an executable file? (Check all that apply.)

   ☐ `g++ -c yourprog.cpp`
   ⊗ `g++ yourprog.cpp Lexer.cpp`
   ⊗ `g++ Lexer.cpp yourprog.cpp`
   ☐ `g++ -c Lexer.cpp`
   ⊗ `g++ yourprog.cpp Lexer.cpp -o best`
   ☐ `g++ Lexer.cpp -o yourprog.cpp`

7. Write a C++ line that defines a new type named `mytype_t`. The type is a function pointer to a function that takes two `char`'s and returns an `int`.

```
typedef int (*mytype_t)(char, char);
```

8. Consider the following fragment

```
void foo(int&, int&); // foo()'s definition is somewhere else
int a=1, b=2;
foo(a, b);
// what's a & b now?
```

   The values of `a,b` after the fragment are

   ☐ `a==1` and `b==2` ☐ `a==2` and `b==1` ⊗ Can't tell, need to know what `foo()` does

9. Consider the following fragment

```
void foo(int, int); // foo()'s definition is somewhere else
int a=1, b=2;
foo(a, b);
// what's a & b now?
```

   The values of `a,b` after the fragment are

   ⊗ `a==1` and `b==2` ☐ `a==2` and `b==1` ☐ Can't tell, need to know what `foo()` does

10. Suppose we want to define a variable `var` so that later we can assign `var["abc"] = true;` and `var["xyz"] = false;`. How would we define `var`?

```
map<string, bool> var;
```

Problem 2 (30 points)**.** In writing the following functions, you can assume that `using namespace std;` is at the top of the file.

1. (**15**) Write a C++ function `foo()` that takes a vector `myvec` of integers and an additional integer `k` as arguments, and returns `true` if `k` appears at least twice in `myvec`, `false` otherwise.

```
bool foo(vector<int> myvec, int k) {
    int count=0;
    for (size_t i=0; i<myvec.size(); i++) {
        if (k == myvec[i]) count++;
        if (count == 2) return true; // optimization
    }
    return false;
}
```

2. (**15**) Write a C++ function `bar()` that takes a stack `st` of `int` as argument, and returns a stack `ret` of `int` which contains every alternate integer in `st`, starting from the top, in the same relative order. In other words, if we were to mark the positions of integers in `st` from the top with position 1 (the top), 2 (below the top), 3 (below the two top), etc. Then, `ret` contains all integers at the odd positions. For example,

```
     bottom --> top                bottom --> top
st = [ 1 3 -2 9 4 3],   then   ret = [ 3 9 3 ]
st = [ 1 3 -2 9 4 3 7], then   ret = [ 1 -2 4 7 ]
```

```
stack<int> bar(stack<int> st) {
    stack<int> tmp;
    int position=1;
    while (!st.empty()) {
        if (position++ % 2 == 1) tmp.push(st.top());
        st.pop();
    }
    stack<int> ret;         // this is to reverse the stack
    while (!tmp.empty()) { // it's OK if you missed this part
        ret.push(tmp.top());
        tmp.pop();
    }
    return ret;
}
```

Problem 3 (20 points)**.** You can assume that `using namespace std;` is at the top of the file. Write a function `foo()` that does the following. It takes two vectors `vec1` and `vec2` of `int` as input. A *sub-sum* of any integer vector `vec` is an integer of form $\sum_{i=j}^{k}$ `vec[i]` for some j, k such that $0 \leq j \leq k \leq$ `myvec.size()` $- 1$.

The function `foo()` returns `true` if there is some sub-sum in `vec1` that is equal to some sub-sum in `vec2`, and `false` otherwise.

For example,

```
if vec1 = [ 3 5 2 3 5 ], and vec2 = [ 4 11 0 ] then true is returned
if vec1 = [ 3 5 2 3 5 ], and vec2 = [ 4 -3 ] then false is returned
```

In the former case, $5 + 2 + 3 + 5 = 4 + 11 + 0$; in the latter, no sub-sum from `vec1` is equal to 4 or -3 or $4 - 3$.

```
bool foo(vector<int> vec1, vector<int> vec2)
{
    vector<int> ss1; // sub-sums from vec1
    vector<int> ss2; // sub-sums from vec2
    size_t j, k;
    int sum;

    for (j=0; j<vec1.size(); j++) {
        sum = 0;
        for (k=j; k<vec1.size(); k++) {
            sum += vec1[k];
            ss1.push_back(sum);
        }
    }
    // could have written a function for this; as it's done twice
    for (j=0; j<vec2.size(); j++) {
        sum = 0;
        for (k=j; k<vec2.size(); k++) {
            sum += vec2[k];
            ss2.push_back(sum);
        }
    }
    // now the real work
    for (i=0; i<ss1.size(); i++)
        for (j=0; j<ss2.size(); j++)
            if (ss1[i] == ss2[j]) return true;
    return false;
}
```

Problem 4 (15 points)**.** In writing the following function, you can assume that `using namespace std;` is at the top of the file. Consider a stripped-down version of `Token` type in the `Lexer.h` file in assignment 3:

```
enum token_types_t { OPERATOR, NUMBER };
struct Token { token_types_t type; std::string value; };
```

Write function `foo()` that takes in a vector `postfixvec` of tokens which contains an expression in *proper* postfix form, and returns a vector of tokens in infix form. **You can assume that there are only two types of operators: $+$ and $-$. In particular, you don't have to worry about precedence at all!** For example, suppose the input vector is

```
[(NUMBER, "123"), (NUMBER, "456"), (OPERATOR, "+"), (NUMBER, "7"), (OPERATOR, "-")]
```

then the output vector is

```
[(NUMBER, "123"), (OPERATOR, "+"), (NUMBER, "456"), (OPERATOR, "-"), (NUMBER, "7")]
```

```
vector<Token> foo(vector<Token> postfixvec)
{
    vector<Token> infixvec;
    stack<vector<Token> > st; // stack of infix expressions

    // IDEA: run postfix evaluation algorithm on the infix expressions
    for (size_t i=0; i<postfixvec.size(); i++) {
        if (postfixvec[i].type == NUMBER) {
            infixvec.clear();
            infixvec.push_back(postfixvec[i]);
            st.push(infixvec);
        } else { // i.e. type == OPERATOR
            vector<Token> a, b;
            b = st.top(); st.pop();
            a = st.top(); st.pop();
            st.push(combine(a, b, postfixvec[i]));
        }
    }
    return st.top(); // assumed input is proper
}

// return an infix expression for (a op b)
vector<Token> combine(vector<Token> a, vector<Token> b, Token op)
{
    if (op.value == "-") { // flip signs of all ops in b
        for (size_t i=0; i<b.size(); i++) {
            if (b[i].type == OPERATOR)
                b[i].value = (b[i].value == "+")? "-" : "+";
        }
    }
    a.push_back(op);
    for (size_t i=0; i<b.size(); i++)
        a.push_back(b[i]);
    return a;
}
```

Problem 5 (5 points extra credit problem)**.** Suppose we relax the assumption from Problem 4, we allow for the input vector to contain $*$ and $/$ operators. Do you need more types of tokens than just `OPERATOR` and `NUMBER`. Why or why not?

Solution. Strictly speaking, no; We can always use the distributive law to "expand" the expressions so that no parentheses are necessary.

If we decided to go that way, the `combine` routine has to be changed slightly.

But I would also accept a YES answer, as long as you articulate roughly the fact that for something like $3 * (4 + 5)$ you will need the `DELIM` type. □