CSE 250 Spring 2011
Homework 1
Due Date: Feb 14, Monday, by 2:05pm
Total Points: 24

1. (3 points) Trace the following program segments:

```
double x = 3.1416;
double y =2.71828;
double z;
double* px = &x;
double& ry=y;
*px = 6.28;
z = ry;
px = &z;
ry = *px;
cout << "x= " << x << endl;
cout << "z= " << z << endl;
cout << "y= " << y << endl;
cout << "ry= " << ry << endl;
cout << "*px= " << *px << endl;
```

What will be the output? (You may run the program to see the output. But make sure you understand why.)

Solution.

```
x= 6.28
z= 2.71828
y= 2.71828
ry= 2.71828
*px= 2.71828
```

x is originally equal to 3.1416, but the pointer px is assigned to the address of x, and when $*px$ (the thing px points to) is set to 6.28, this changes the value at the address of x. The reference variable ry is set to refer to y which holds the value of 2.71828, so ry also equals 2.71828. Then z is set equal to the variable that ry refers to (i.e. $z = y$, so $z = 2.71828$ now). The pointer px is then set to point to the address of z. The next line then sets the variable refered to by ry to be equal to the variable pointed to by px (i.e. $y = z$). Since y is already equal to 2.71828, its value (and that of its reference variable ry) does not change.

2. (2 points) Consider the following program segment:

```
void swap (int a, int& b)
{  int temp = a;
   a=b;
   b=temp;
}

int main()
{ int x=-5, y=5;
```

```
  swap (x,y);
  cout << "x = " << x << " y = " << y << endl;
}
```

What will be the output? (You may run the program to see the output. But make sure you understand why.)

Solution.

```
x = -5 y = -5
```

When swap is called, a is a local variable which is set equal to the first argument, and b is created as a reference to the second argument. *temp* is assigned to the value that a has at first, and then a is assigned to the value of b. b is then assigned to the original value of a, which was stored in *temp*. So the two values are swapped within the function. However, because the return type is void, the local value of a never gets returned to main, since x is passed by value. But y is passed by reference, so changes within the function to the value that b refers to (which is y) will exist beyond the scope of the function.

3. (2 points) Consider the following program segment:

```
int * p = new int();
int * q = new int();
(*q) = 4;
p = q;
(*p) += 5;
cout << *p << endl;
```

What value will be printed out?

Solution.

```
9
```

At first, the value of the integer pointed to by q is set to 4. Then pointer p is assigned to point to the same location that is pointed to by q, so q now points to a spot in memory holding the number 4. The next line increments the thing pointed to by p by 5 and assigns that value back to the thing pointed to by p (the location in memory that had previously held the number 4). The printed output is the number pointed to by p, so 9.

4. (3 points) Page 16, Programming Problem 1.

Solution. The following block of code will produce the same result as the `for` loop from Example P.4:

```
if(MAX_VAL > 0) {
    int next_int = 1;
    do
    {
        if (next_int % 2 == 0) {
            sum += next_int;
        }
```

```
        else {
            prod *= next_int;
        }
        next_int++;
    } while(next_int <= MAX_VAL);
}
```

5. (3 points) Page 29, Self-Check Problem 1.
   welcome holds the string "hello"
   greeting holds the string "ciao"
   p is a "point" object with values 1.5 and 3.7 for its coordinates
   $x = 5.5$
   $y = 10.2$
   $z = 0$
   $*px = 5.5$ ($*px$ now points to x)
   $ry = 10.2$
   $*px = 3.14$ and $x = 3.14$ (since we just changed the value in memory where px points)
   $z = 10.2$
   $r = 3.14$ (initialized reference variable r to be what px is pointing at)
   $r = 99.44$, $*px = 99.44$, and $x = 99.44$ (we changed r, but r is a reference to the spot in memory pointed to by px, which is where the variable x is stored, so by changing r, we change that spot in memory, and thus we change the value of all three)
   $*px = 10.2$ (since it now points to z's location in memory)
   $ry = 10.2$ (ry is now a reference to the spot in memory pointed to by px, which is the variable z. The value of ry doesn't change, because it was already 10.2, but now it refers to a different 10.2 in memory)

6. (4 points) Page 32, Self-Check Problem 2.

   (a) Call-by-value

   (b) Call-by-reference

   (c) Reference object

   (d) Constant reference object

7. (4 points) Page 38, Programming Problem 1. The following function will produce the desired result. Note that the *sort* function called here can be accessed by including the `<algorithm>` header in your code.

```
bool same_elements(int a[], int b[], int n)
{
    sort(a, a+n);
    sort(b, b+n);

    for (int index=0; index < n; index++) {
        if(a[index] != b[index]) {
            return false;
        }
    }
    return true;
}
```

8. (3 points) Page 44, Self Check, Problem 1.

Solution. Note: It's unclear whether the textbook authors intend for some of these statements to contain typos or not. You may have interpreted them literally as written, or perhaps you interpreted them in the spirit of what it seems like they're asking (i.e. in a way which would compile without the typos). Either way, you will not be penalized so long as you've justified your answer.

```
happy is now holding the string "Happy birthday to you"
false (the space at the end of the other string makes them unequal)
true
syntax error: un_happy needs to be assigned using = before it can be compared using ==
false *
true *
5 , 14 (first space at index 5, first after that at index 14)
what now holds the substring of happy from the 6th to the 13th indices
birthday
```

* - For these two, strings are compared lexicographically, meaning they are compared alphabetically in the natural way that one might sort a list of names: compare the first letters, then if they are the same, compare the second letters, and so on. Here, capital letters are considered $<$ lower-case letters based on their ASCII code values.