Problem Set 0 Solutions

This handout contains sample solutions to the problems on Problem Set 0. There are many solutions to these problems, so don't worry if your answers don't precisely match the ones outlined here. Even if you think you're on top of the material, I recommend looking over these answers since they might illustrate an entirely different approach to solving the problems.

Problem 0

`sample.cpp` is a program that implements the classic "guess the number I'm thinking of" game. The program picks a random number between 0 and 99, inclusive, then invites you to guess what it is. At every step, the program will indicate whether your guess was higher or lower than the computer's number.

As an interesting exercise, you might want to see if you can create an "Evil Guessing" program that, like Evil Hangman, actively dodges the user's guesses. That is, instead of picking a number between 0 and 99, instead you maintain a range of values then gets progressively whittled down based on the user's guesses.

Problem 1

Here's a working version of `GetReal`:

```
double GetReal()
{
    while(true)
    {
        stringstream converter;
        converter << GetLine(); // Assume this is defined.

        double result;
        converter >> result;

        if(converter.fail())
            cout << "Please enter a real number." << endl;
        else
        {
            char leftover;
            converter >> leftover;

            if(converter.fail()) return result;
            else cout << "Unexpected character: " << leftover << endl;
        }
        cout << "Retry: ";
    }
}
```

There is very little code that needs to be changed from `GetInteger` to convert it into `GetReal`. We only need to change the return type, the definition of the `result` variable, and text "Please enter an integer" and we have a completely working input function. Isn't the streams library great?

Problem 2

There are many ways that we can encode all of the necessary information about directions. The following is, in my opinion, the simplest. First, we create a file called `directions.h` and encode it as follows:

```
/* directions.h: Represents a set of directions as calls to macros
 * DEFINE_DIRECTION where the arguments are
 *
 * DEFINE_DIRECTION(direction, opposite, dx, dy)
 */
DEFINE_DIRECTION(North, South,  0,  1)
DEFINE_DIRECTION(South, North,  0, -1)
DEFINE_DIRECTION(East,  West,   1,  0)
DEFINE_DIRECTION(West,  East,  -1,  0)
```

We can then automatically generate the `enum directionT` as follows:

```
enum directionT {
    #define DEFINE_DIRECTION(dir, opp, dx, dy) dir,
    #include "directions.h"
    #undef DEFINE_DIRECTION
};
```

`DirectionToString` can be implemented using an approach similar to that covered in lecture or in the course reader:

```
string DirectionToString(directionT dir)
{
    switch(dir)
    {
        #define DEFINE_DIRECTION(dir, opp, dx, dy) case dir: return #dir;
        #include "directions.h"
        #undef DEFINE_DIRECTION

        default: return "<unknown?>";
    }
}
```

`OppositeDirection` can be written similarly:

```
directionT OppositeDirection(directionT dir)
{
    switch(dir)
    {
        #define DEFINE_DIRECTION(dir, opp, dx, dy) case dir: return opp;
        #include "directions.h"
        #undef DEFINE_DIRECTION

        default: return dir;
    }
}
```

`MovePointInDirection` is perhaps the most interesting of the functions. Many of you implemented this function by writing a pair of functions `GetDX` and `GetDY` that return the Δx and Δy components of a

direction and then having `MovePointInDirection` glue the results together to move the point. This is perfectly fine. Alternatively, you can implement the function as follows:

```
pointT MovePointInDirection(pointT pt, directionT dir)
{
    switch(dir)
    {
        /* The following definition spans multiple lines, so we need to
         * terminate each line with \ so that the definition carries on to
         * the next line.
         */
        #define DEFINE_DIRECTION(dir, opp, dx, dy) \
        case dir: \
            pt.x += dx; \
            pt.y += dy; \
            return pt;
        #include "directions.h"
        #undef DEFINE_DIRECTION

        default: return pt;
}
```

One error I saw several times in this problem was to write the above code as

```
pointT MovePointInDirection(pointT pt, directionT dir)
{
    switch(dir)
    {
        /* The following definition spans multiple lines, so we need to
         * terminate each line with \ so that the definition carries on to
         *  the next line.
         */
        #define DEFINE_DIRECTION(dir, opp, x, y) \
        case dir: \
            pt.x += x; \
            pt.y += y; \
            return pt;
        #include "directions.h"
        #undef DEFINE_DIRECTION

        default: return pt;
}
```

The problem with this code is that we've now made the tokens `x` and `y` correspond to the arguments of the `DEFINE_DIRECTION` macro. This means that the code

```
pt.x += x;
pt.y += y;
```

Will get converted into something of the form

```
pt.0 += 0;
pt.-1 += -1;
```

This is clearly nonsense, but unfortunately the compiler is unlikely to give you a coherent error message. As a general rule of thumb, when defining macros, make sure that you don't assign a name to a macro parameter that already has a meaning somewhere else in the program.

Problem 3

There are many possible solutions to this problem, one of which is printed below. This code only handles the case where elements are inserted at the end of the container; it shouldn't be too difficult to modify this code so that it inserts at the beginning.

```
#include <iostream>
#include <vector>
#include <deque>
#include <ctime>
#include <string>
using namespace std;

const int NUM_STRINGS = 50000;
const string SOURCE_STRING = "Ce n'est pas une chaîne de caractères.";

void TimeVector(bool reserveFirst)
{
    vector<string> trialObject;
    if(reserveFirst) trialObject.reserve(NUM_STRINGS);

    clock_t startTime = clock();

    for(int k = 0; k < NUM_STRINGS; ++k)
        trialObject.push_back(SOURCE_STRING);

    startTime = clock() - startTime;
    cout << "Time  " << (reserveFirst ? "with" : "without") << " reserve: "
         << (static_cast<double>(startTime) / CLOCKS_PER_SEC) << endl;
}

void TimeDeque()
{
    deque<string> trialObject;

    clock_t startTime = clock();

    for(int k = 0; k < NUM_STRINGS; ++k)
        trialObject.push_back(SOURCE_STRING);

    startTime = clock() - startTime;
    cout << "Deque took: "
         << (static_cast<double>(startTime) / CLOCKS_PER_SEC) << endl;
}

int main()
{
    TimeVector(true);
    TimeVector(false);
    TimeDeque();
}
```

Problem 4

This is a reasonably straightforward exercise in `map` iterators. One solution is shown here:

```
map<string, string> ComposeMaps(map<string, string>& one,
                                map<string, string>& two)
{
    map<string, string> result;
    for(map<string, string>::iterator itr = one.begin(); itr != one.end();
        ++itr)
    {
        /* Get an iterator to the key/value pair in two that corresponds to
         * this key/value pair, if it exists.
         */
        map<string, string> match = two.find(itr->second);
        if(match != two.end())
            result.insert(make_pair(itr->first, match->second));
    }
    return result;
}
```

Problem 5

This problem is very open-ended and there are many ways that you can go about solving it. The main question is how you choose to represent the bag of chips. Many of you opted for a `vector` or `multiset`, both of which have their advantages and disadvantages. In particular, using a `vector` to represent the bag means that it is efficient (O(1)) to choose random elements but slow (O(N)) to remove them. The `multiset` has the opposite behavior – it takes O(N) time to pick a random element but O(lg N) time to remove it. An alternative solution is to represent the bag as three integers, each tracking the number of elements of each type in the bag. We can then add and remove elements from the bag efficiently (O(1)) by incrementing and decrementing the counts of each chip type. We can also pick elements out of the bag in O(1) time using the following trick: if we representing the bag of elements as three integers, then we can think of the bag as a range partitioned into three subranges – one for red chips, one for blue chips, and one for yellow chips. You can visualize this as follows:

Red Chips	Blue Chips	Yellow Chips

To pick a random element out of this bag, we pick a random number between 0 and the total number of chips in the bag. We can then check which color of chip we picked by looking at where in this range the number falls. There are three cases:

1. If the random number is less than the number of red chips, it is in the range of red chips and the random element is a red chip.
2. If the random number is greater than the number of red chips and less than the total number of red and blue chips, it is contained in the range of blue chips and the random element is a blue chip.
3. Otherwise, the chip must be in the range of yellow chips and the chip is yellow.

Using this technique, we can implement the simulation as follows:

```
#include <cstdlib>   // For rand, srand
#include <ctime>     // For time
#include <iostream>  // cout, etc.
using namespace std;

/* Type representing colors of chips. */
enum Color {Red, Yellow, Blue};

/* A type representing the bag of chips. */
struct Bag
{
      int red, blue, yellow;
};

/* Returns the total number of chips in a bag. */
int TotalChips(Bag& b)
{
      return b.red + b.blue + b.yellow;
}

/* Removes a random chip from the bag and returns its color.  See the
 * above discussion for why this works.
 */
Color RemoveRandomChip(Bag& bag)
{
      int index = rand() % TotalChips(bag);
      if(index < bag.red)
      {
            --bag.red;
            return Red;
      }
      else if(index >= bag.red && index < bag.red + bag.blue)
      {
            --bag.blue;
            return Blue;
      }
      else
      {
            --bag.yellow;
            return Yellow;
      }
}
```

```
/* Runs a simulation of the game and returns how steps it takes */
int NumStepsRequired()
{
      const int kMaxNumber = 10000; // Maximum number of chips
      int numSteps = 0; // Number of steps taken in the simulation

      /* Create an initialize the bag. */
      Bag bag;
      bag.red = bag.blue = bag.yellow = 5;

      /* Keep looping until we have too few or too many chips. */
      while(TotalChips(bag) < kMaxNumber && TotalChips(bag) >= 2)
      {
            ++numSteps;

            /* Pick two random chips out of the bag. */
            Color color1 = RemoveRandomChip(bag);
            Color color2 = RemoveRandomChip(bag);

            /* If both chips are yellow, add one yellow chip and two
             * blue chips to the bag.
             */
            if(color1 == Yellow && color2 == Yellow)
            {
                  bag.yellow += 1;
                  bag.blue   += 2;
            }
            /* If exactly one chip is yellow, add three blue chips. */
            else if(color1 == Yellow || color2 == Yellow)
                  bag.blue += 3;
            /* Otherwise, if one of the chips is blue and the other
             * isn't yellow, add five red chips.
             */
            else if(color1 == Blue || color2 == Blue)
                  bag.red += 5;
            /* Otherwise, both must be red, so we add one red chip. */
            else
                  bag.red += 1;
      }
      return numSteps;
}

int main()
{
      /* Seed the randomizer */
      srand(static_cast<unsigned>(time(NULL)));

      /* Run some number of trials (here, 100000) */
      const int kNumTrials = 100000;
      double total = 0;
      for(int i = 0; i < kNumTrials; ++i)
            total += NumStepsRequired();

      /* Output the average. */
      cout << "Average length: " << (total /kNumTrials) << endl;
      return 0;
}
```