CS 7A - Fall 2016 - Converting between base 2, base 10, base 26 for encryption. Due 10/6/16

Background Theory

To protect your identity when your grades are posted to the internet, I've added a level of encryption that takes your student id number (base 10) and converts it to ascii characters with numbers between 65 and 90 (the English alphabet in capital letters).

Suppose your student id number (in base 10, the way you get it) is 0123456 or, as a base 10 number, without the leading zero, $N = 123,456$. One way to find the base-10 digits is to repeatedly divide by 10, each time recording the remainder and reducing the dividend by replacing it with the quotient. For $N = 123,456$ the iterative (repeating) process produces these results:

Dividend	*Divisor*	*Quotient*	*Remainder*
123456	10	12345	6
12345	10	1234	5
1234	10	123	4
123	10	12	3
12	10	1	2
1	10	0	1

The coefficients of the powers of 10 that make up 123456 appear as the remainders with the least significant digit coming out first.

$$123456 = 10^5 + 2 \cdot 10^4 + 3 \cdot 10^3 + 4 \cdot 10^2 + 5 \cdot 10 + 6 = 10\,(10(10(10(1 \cdot 10 + 2) + 3) + 4) + 5) + 6$$

Note that the left hand form (123456) is a succinct representation that is computable by either the middle or the right formula, but the right formula is more efficient (involves many fewer multiplications).

This all seems fairly strait-forward and obvious, but part of the beauty is in the way this algorithm to extends to bases other than 10. For example, to convert from base 10 to base 2, simply record the remainders (least significant first) when the number is divided by 2.

Dividend	*Divisor*	*Quotient*	*Remainder*
123456	2	61728	0
61728	2	30864	0
30864	2	15432	0
15432	2	7716	0
7716	2	3858	0
3858	2	1929	0
1929	2	964	1
964	2	482	0
482	2	241	0
241	2	120	1
120	2	60	0
60	2	30	0
30	2	15	0
15	2	7	1
7	2	3	1
3	2	1	1
1	2	0	1

Thus $123456_{10} = 2^{16} + 2^{15} + 2^{14} + 2^{13} + 2^9 + 2^6 = 2^6(2^3(2^4(2(2(2(2+1)+1)+1)+1)+1)+1) = 1111000100100000_2$
Here is pseudocode for accomplishing the conversion from base 10 to base 2:

```
number = positive integer

while (number > 0 ) {
  bit    = number mod 2 ;
  number = number div 2 ;
  put bit to the left of any previous bits
}
```

In general, the binary form of a number can written in this form:

$$N = d_0 + d_1 \cdot 2 + d_2 \cdot 2^2 + \cdots + d_n \cdot 2^n$$

where the ith division by 2 and produces the remainder d_i in turn.
For instance, $237 = 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 + 0 \cdot 16 + 1 \cdot 32 + 1 \cdot 64 + 1 \cdot 128 = 11101101_2$
Here is one way we could start writing some C++ code for this algorithm:

```
/** Geoff Hagopian
Converting from base 10 to base 26 using A=0,B=1,...,Z=25 as place values. **/

#include "std_lib_facilities.h"

int main() {
        string str;
    int x, temp;
    cout << "\nEnter␣and␣integer␣to␣convert␣to␣base␣26\n";
    cin >> x;
    temp = x%26; //the remainder after division by 2
    str = str + char(65+temp); //convert to an alphabetic character and append to st
    x /= 26; // compute the quotient after division by 26
    temp = x%26;
    str = str + char(65+temp);
    x /= 26;
    temp = x%26;
    str = str + char(65+temp);
    x /= 26;
    //...keep repeating this until it's done...how do we know?
}
```

After working a number of these conversions by hand and taking note of the what happens to `x` when it's repeatedly divided by 26 (recording the remainder at each step) we observe that we are done when `x` is zero. This suggests using a `while` loop and condition the loop on `x` not equal to zero:

```
//beginning comments and preprocessor commands omitted for brevity

int main() {
    int x, temp;
    string str;
    cout << "\nEnter␣and␣integer␣to␣convert␣to␣base␣26\n";
    cin >> x;
    while(x!=0) { // note that "!" is the negation operator.
        str = str + char(65+x%26); // 65 + remainder after division by 26 cast as a
        x /= 26; // integer division discards remainder, so 23/26 is 0.
    }
}
//-------------------------------------------------------------------------
```

Here's some code we developed in class for constructing a random 64-bit number and converting it to a string using the method described above. You'll need to include <iostream>, <cstdlib>, <ctime>, and <bitset>.

```
int main() {
    srand(time(0)); /// seed the pseudo-random number generator
    uint64_t id;
    id = random64bit();
    string idstr = numberToString(id);
    //string idstr;
    cout << "\nHere's a string from the random number\n" << id << ": " << idstr;
    /// Enter code here to compute the square of you student number .
}

string numberToString(uint64_t x) {
    string idstr;
    while(x) {
        idstr = char(65+x%26)+idstr;
        x /= 26;
    }
    return idstr;
}

uint64_t random64bit() {
    uint64_t randNum = rand(); // two byte random number
    for(int i = 0; i < 3; ++i) {
        randNum = (randNum << 16) + rand();
        cout << endl << bitset<64>(randNum);
    }
    return randNum;
}
//-------------------------------------------------------------------------------
```

Questions:

1. Adapt this code to "encrypt" the square of your student number as a string. You should find this in the grades section of geofhagopian.net

2. Now write code to check your result by reversing it, so we can recover the student number in base 10 from the encrypted version of its square. (Not a very secure encryption, is it?)

   Submit your code as <your initials>_encryptSID.cpp by 10/6/16.