Lab 3: Pseudo-Random Numbers and Strings

Random number generation in C/C++

Random number generation is critical to many simulations and games. C (and therefore C++) provides two different functions that can generate pseudo-random integers in the range from `0` to `RAND_MAX` inclusive. `RAND_MAX` is a large positive integer that is implementation dependent. It really does not matter what its actual value is for most applications. There are many pseudo-random number generators available and some are better than others. What is meant by the word "better"? The quality of a pseudo-random number generator is measured by various statistical measures, but intuitively, the better it is, the more its output will be like truly random numbers. Pseudo-random number generators are called ***weak*** or ***strong*** depending on how good they are.

If you want a very strong pseudo-random generator, you may have to write it yourself. The functions in the C/C++ library are pretty strong, but not suitable for all applications. The two functions of choice in that library are `rand()` and `random()`. Because the textbook for the course describes `rand()`, these notes will be limited to `rand()`. Almost everything to be said about `rand()` is true of `random()` as well.

The `rand()` Function

The `rand()` function is part of the C Standard Library and therefore you need to include the header `<cstdlib>` in your C++ program to use it. (In C, you would include `<stdlib.h>`.) The function generates a pseudo-random integer in the range `0` to `RAND_MAX`. `RAND_MAX` is a constant defined in `<cstdlib>`. Its default value may vary between implementations but it is granted to be at least `32767`. On the `cslab` hosts in the lab this header file has the following line:

```
#define RAND_MAX 2147483647
```

which gives `RAND_MAX` the value $2^{31} - 1$. (You can see this line by typing the command "`grep RAND_MAX /usr/include/stdlib.h`".)

What if you want to generate a random number between 1 and 100? How would you use this function to do this? One simple way is to use the modulus operator `%` to narrow the range. This is not the best way, but it works. To illustrate:

```
number = rand() % 50;
```

assigns to `number` a value between 0 and 49. If we add 1 to it we get a number from 1 to 50.

```
number = ( rand() % 50 ) + 1;
```

More generally, suppose we want to generate pseudo-random numbers in the range `[lower...upper]` inclusive, where `lower` and `upper` are any pair of integers such that `lower < upper`. There are (`upper - lower +1`) numbers in this range. Let

```
rangesize = upper-lower+1;
```

The expression

```
rand() % rangesize
```

generates a number between 0 and `rangesize-1`, or equivalently, between 0 and `upper-lower`. To get numbers that range between `lower` and `upper`, we just have to add `lower` to the result. This way, the numbers lie between `0+lower` and (`upper-lower+lower`), which is from `lower` to `upper`:

```
number = ( rand() % rangeSize ) + lower;
```

Example: To generate numbers in the range from 100 to 400, let *lower = 100* and *upper=400* in the above formula. Then the rangesize is $(400-100+1) = 301$. Using this, the assignment would be

```
number = (rand() % 301) + 100;
```

Note. I mentioned above that this is not a very good method of narrowing the range of pseudo-random numbers. This is because the `%` operator is throwing away many valuable bits of information. For greater strength, it is usually better to use a different approach in which you use division instead of modulus.

The `srand()` Function

The `rand()` function cannot be used reliably without first giving its internal generator a starting value. The process of giving it a starting value is called ***seeding the generator***. There is a function whose sole purpose is to seed `rand()`. Its name is easy to remember: `srand()`. This function seeds the pseudo-random number generator. The seed is passed as the argument to the function. The prototype (or declaration) of `srand()` is

```
void srand( unsigned int)
```

Therefore you use it like this:

```
srand(1000000);
```

In this case you gave it a seed of 1000000.

When it is given the same seed, `rand()` will always produce the same sequence. Therefore, if you give `srand()` the same seed each time a program is run, the sequence of numbers generated by `rand()` will be the same. This little program can be used to illustrate this:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    srand(100);
    for ( int i = 0; i < 10 ; i++ )
        cout << rand() << endl;

    return 0;
}
```

Compile it and run it repeatedly. You will see the same ten numbers each time.

To get a new sequence each time, the value given to `srand()` must be different in each run. The most common way of doing this is to seed `srand()` with the current time. UNIX has a function that returns the current time. The function `time()` (declared in the C header file `<ctime>`) returns the current time as an integer. You give it a 0 as an argument:

```
    current_time = time(0);
```

and it returns an integer value. To be more precise, `time(0)` returns the number of seconds that have elapsed since since 00:00 hours, Jan 1, 1970 UTC. (This is due to historical reasons, since it corresponds to a UNIX timestamp, but is widely implemented in C libraries across all platforms.) The important fact is that the value returned by time(0) is guaranteed to be different almost every time that the program runs, it is a perfect choice for the seed. Try running this program:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0));
    for ( int i = 0; i < 10 ; i++ )
        cout << rand() << endl;

    return 0;
}
```

and you will see the sequence vary from one run to another.

A Bit About Strings

C++ allows you to declare variables of type `string` in your code. In order to declare and use such variables, the program must include the `<string>` header file, which is placed into the `std` namespace, so you need the lines

```
#include <string>
using namespace std;
```

in the beginning of your program. The type `string` is different from elementary types such as `int`, `float`, and `char`. It is a ***class***. C++ has libraries that define complex things called ***classes***. You should think of a class, for now, as a collection of data values together with operations and functions that act on those values.

Declaring string objects

There are several ways to declare and initialize string objects. Two equivalent ones are

```
string filename = "input_file";
```

and

```
string filename("input_file");
```

Which one you use is a matter of taste.

Concatenating strings

The word ***concatenation*** means juxtaposing two strings one after the other. For example, the concatenation of "`butter`" and "`flies`" is "`butterflies`". The operator used for adding numbers, `+`, has a different meaning when it is placed between two string objects; it forms their concatenation[1].

If `username` and `hostname` are two string variables, and `username` contains the string "`stewart.weiss`" and `hostname` contains the string "`hunter.cuny.edu`", then the statement

```
cout << username + "@" + hostname;
```

will output "`stewart.weiss@hunter.cuny.edu`", and if `email` is a string variable

[1]The reuse of the + sign here is an example of an **overloaded operator**: it works on objects of the string class differently than it does on numbers.

```
email = username + "@" + hostname;
```

will assign "`stewart.weiss@hunter.cuny.edu`" to `email`. We could also have written the above with the @ symbol as a character, not a string:

```
email = username + '@' + hostname;
```

The concatenation operator lets you concatenate characters and string variables, string literals and string variables, and two string variables, but not string literals or characters with each other. Assume `username` and `hostname` are string variables that have values. The following illustrates what you can and cannot do:

```
username + hostname              // legal expression
username + '@'                   // legal expression
username + "@hunter.cuny.edu"    // legal expression
"hunter" + ".cuny.edu"           // illegal expression
"hunter" + '.'                   // illegal expression
```

Getting the size of a string object

Variables of the string class have a `size()` function that can return their size, which means the number of characters in them. To use this function, you have to put "`.size()`" after the variable name, just the way you use the `open()` function after an `ifstream` or `ofstream` variable. For example:

```
string input_line;
cin >> input_line;
cout << "You entered " << input_line.size() << " characters.\n";
```

In the third line, the value of `input_line.size()` is the number of characters presently stored in that string variable.

Accessing individual characters in a string object

There is a way to access each individual character in a string object using what is called the ***subscript operator***. The subscript operator is a pair of square brackets `[]` placed after the name of the string, with a number in between. The numbering starts at 0 and goes up to the size of the string minus 1. The number in brackets is called the ***index value***. Thus:

```
cout << "The first character you entered was " << input_line[0] << ".\n";
```

will display the character in the first position in the string `input_line`. Notice that the first position is at index 0, *not 1*! It will be hard getting used to this idea and you will likely forget, but try, try to remember that the positions in the string are numbered 0, 1, 2, 3, and so on up to `size()`-1:

```
int last_index = input_line.size() - 1;
cout << "The last character you entered was " << input_line[last_index] << ".\n";
```

will output the character in the last position of the string. You can modify individual characters using an assignment statement and the subscript operator:

```
input_line[0] = 'L';
```

will replace the first character in the string by the letter 'L'.

Warning: C++ compilers do not check if you are trying to access or modify a location outside of bounds of your `string` object. But if you do, your program may work incorrectly or even crash! Be careful!