Project 1, Part B
CS 303/503 Algorithms

Fall 2012

Due by: 8:00 a.m., Friday, October 19, 2012

This programming project is subject to the CSCI 303/503: Regulations for Programming Projects found at `www.cs.wm.edu/~va/CS303/project_regs.html`.

In this project, you are to extend the implementation of the `GroupOfNumbers` class to include two new constructors that allow more flexibility in the way that `GroupOfNumbers` class objects can be created and initialized, particularly those containing a large number of entries. In addition, to make our testing—and perhaps yours!—easier, you will implement one new accessor for the class:

```
writeGroupOfNumbersTo(const char* fname).
```

The revised header file group.h can be found on the Web page for this project.

We continue to restrict our attention to an array-based implementation of `GroupOfNumbers` since this will best suit our future uses of `GroupOfNumbers`. You also are to expand the error handling within the constructors now that there are four possible constructors, each of which can encounter difficulties under predictable circumstances.

Each of the two new constructors are discussed in detail below. Before we proceed, keep in mind that *all* your constructors for `GroupOfNumbers` *must* properly initialize *all* the private data members of any `GroupOfNumbers` class object, given the request of the user and the conditions outlined in the definition of `GroupOfNumbers`. A discussion of error handling within the constructors will then follow. Finally, a brief description of the new accessor will be given.

Initializing a group of numbers using a random number generator

On the Web page for this programming project, you will find links to the files `rng.c` and `rng.h`. The file `rng.c` contains an ANSI C library for random number generation that is a so-called "Lehman random number generator"; the file `rng.h` is the associated header file for this library.[1] Since C++ is "backward-compatible" with ANSI C, you can compile `rng.c` using `g++` just as you would any standard `.cc` file.

For your purposes, the fact of note is that this library contains a function `double Random(void)` that returns a pseudo-random *real* number uniformly distributed between 0.0 and 1.0. But the `GroupOfNumbers` class expects (long) *integer* entries. So the question becomes, how do you use the function `Random`, which returns a double precision floating point random variate in the range (0.0,1.0) to generate integer-valued random variates? The solution, which you are to implement, is straightforward.

Let a and b be integer-valued parameters with $a < b$. Our goal is to generate values of an integer-valued random variate x in such a way that all integers between a and b inclusive are equally likely. In this case, x is an *Equilikely(a,b)* random variate that can be generated by calling the following function:

[1] I am indebted to Professor Steve Park for providing me with copies of `rng.c` and `rng.h`, a reprint of the paper "Random Number Generators: Good Ones Are Hard to Find," Steve Park and Keith Miller, *Communications of the ACM*, October 1988, and a copy of the page proofs for the first three chapters of the manuscript *Discrete-Event Simulation: A First Course*, Steve Park and Lawrence Leemis—from which much of this discussion was drawn.

```
long Equilikely(long a, long b) { // use a < b
  return (a + (long) (( b - a + 1) * Random()) );
} // end Equilikely
```

Since `Equilikely` is a function that you need simply for the purposes of implementing this particular constructor, note that I have defined it to be a protected method of the class `GroupOfNumbers`; it is available to help you implement the constructor, but is not for general public use.

As an example of how you would use `Equilikely`, look at the file `example.cc`, available from the Web page for the project, which shows how to dynamically allocate and then initialize an array A of n long integers with random (long) integer variates. With this example as a guide, you should be able to figure out how to initialize a `GroupOfNumbers` class object with long-valued random variates as entries for the total number of entries requested by the user.

This raises the second condition you must remember when implementing this constructor. In general, the user probably will invoke this constructor only when interested in generating a large `GroupOfNumbers` class object that is completely initialized. However, we are committed to ensuring that the array allocated has a *minimum* of `default_LIMIT` positions, so your implementation of this constructor should ensure that this second condition is satisfied. If the total number of entries requested by the user is less than `default_LIMIT`, then, just as for the default constructor, the value used to allocate the `_group` array should be `default_LIMIT`. However, the number of *active* entries in the array initialized using the random number generator should be the total number requested by the user.

This is an important part of the specification for the project, so read it again and make sure you understand what is required.

For example, if `default_LIMIT = 10` and the user defines a new `GroupOfNumbers` object using this constructor with `total_requested` equivalent to 5, as in:

```
GroupOfNumbers G(-100,100,5);
```

then your implementation should *allocate* the array to have 10 (i.e., `default_LIMIT`) positions, but when the constructor is finished the number of active, *initialized* entries in the array (i.e., `_total`) should be 5 (i.e., `total_requested`). If you implement this correctly, then the above definition of `G`, followed by:

```
G.displayGroupOfNumbers();
```

should result in:

```
-90 -74 42 -39 72
```

If I then wish to add up to five more numbers to the `GroupOfNumbers` object `G`, this should be possible since all `GroupOfNumbers` objects are guaranteed to be able to contain at least ten numbers.

Initializing a group of numbers by reading in values from a file

The user also may want to create (once) a large file containing entries for a group of numbers and then read in these entries from the file created for this purpose. The list of tasks for the constructor are then as follows:

- allocate an array of sufficient size;
- open the file `fname`;
- determine the number of entries available in the file `fname` (which should be the first entry on the first line of the file);
- initialize the active entries in the array by reading them in from the file `fname`; and
- close the file `fname` *before* returning from the constructor.

Since we are depending on a file for the initialization, and that leaves us at the mercy of the user, there are now several obvious places where things can go wrong. A full description of what errors to handle, and how to respond, will wait for the next section.

It is expected that a user will invoke this constructor to create and initialize large `GroupOfNumbers` class objects, but once again your implementation of the constructor should check and make sure that the *minimum* number of positions we have guaranteed will be available is satisfied. If the total number of entries requested by the user is less than `default_LIMIT`, the allocation for `_group` should contain `default_LIMIT` positions, but the active number of entries initialized from the file should be the total number of entries requested by the user.

The format of the file should be as follows: the first entry on the first line gives the total number of entries available for the initialization. (Be sure that your constructor can handle the case `fname` is empty!) All subsequent entries in the file should be valid candidates to be added to a GroupOfNumbers object. For instance, the following would be one example of the contents of an admissible input file:

```
4
0 1
2 3
```

Note that the user may request less entries than are actually available in the file—which is fine. (For instance, the user may create a file with one million entries but on a particular test run only want the first one hundred entries in the file.)

Error handling within the constructors for `GroupOfNumbers`

Now that we have a richer suite of constructors from which to choose, and one of those constructors involves file input and output, we are going to expand the number of exit codes we use to handle errors. Of particular concern is that using input from a file opens up multiple possibilities for errors since now we must rely on the user to supply a file that is set up properly and contains sufficient data to handle the initialization request.

So let's outline the four error situations you now are required to handle and detail what your specific response should be. It's not difficult to imagine additional pathologies that could occur, particularly when a user-supplied file is involved, but you are required to handle the following four situations only. Please resist the urge to resort to further heroics in the project you submit (though you are more than welcome—even encouraged!—to experiment on your own).

`exit(1)`

The one possible error common to all four of the constructors for `GroupOfNumbers` is a failure to allocate an array of sufficient size. After Project 1, Part A, you should know how to detect such failures using a `try` block to `catch` the exception object `bad_alloc`. You also should know how to use `exit()` to terminate execution. Thus it continues to be the case that if the allocation fails in *any* of the four constructors (as wall as in the overloaded assignment operator), then the `catch` block should:

- send a message to standard error of the form specified below and
- exit with a return value of `1`.

The message to standard error (`cerr`) remains of the following form:

```
GroupOfNumbers: bad_alloc
GroupOfNumbers: allocation of size 500000000 failed
GroupOfNumbers: line: 10
GroupOfNumbers: file: group.cc
GroupOfNumbers: exit: 1
```

exit(2)

To ensure consistent initialization of a new `GroupOfNumbers` object using the pseudorandom number generator, we require that the values `lower` and `upper` satisfy the condition that `lower` $\leq$ `upper`. Your implementation of the constructor `GroupOfNumbers(long lower, long upper, long total_requested)` must check that the user's choice for `lower` and `upper` satisfy `lower` $\leq$ `upper`. If not, then you are required to:

- send a message to standard error of the form specified below and
- exit with a return value of `2`.

The message to standard error (`cerr`) must be of the following form if you detect that `lower` $>$ `upper`:

```
GroupOfNumbers: incorrect values for the arguments lower and upper: (42,-1)
GroupOfNumbers: lower is required to be less than or equal to upper
GroupOfNumbers: line: 52
GroupOfNumbers: file: group.cc
GroupOfNumbers: exit: 2
```

Here "`42`" is the value of `lower` that was passed to the constructor while "`-1`" is the value of `upper` that was passed to the constructor.

exit(3)

Since we may be required to open a file to initialize a `GroupOfNumbers` object, a common error for which you are required to check is that the file specified does not exist. In this case, use the file stream (`ifstream`) variable to check whether the opening was successful. If the file specified is unavailable, you are required to:

- send a message to standard error of the form specified below and
- exit with a return value of `3`.

The message to standard error (`cerr`) should be of the following form:

```
GroupOfNumbers: foo: No such file
GroupOfNumbers: line: 87
GroupOfNumbers: file: group.cc
GroupOfNumbers: exit: 3
```

Here, "`foo`" is the name of the file I specified when calling the constructor, but no file with that name was found in the current directory.

exit(4)

Another obvious problem is that the user may request a `GroupOfNumbers` object with, say, two million active entries and yet supply a file that has, say, only one million entries available for the initialization. This is another error that it is easy for you to check. So if the number of entries reportedly available in the file specified is less than the number of entries requested by the user, you are required to:

- send a message to standard error of the form specified below and
- exit with a return value of `4`.

The message to standard error (`cerr`) should be of the following form:

```
GroupOfNumbers: Insufficient number of entries in file: bar
GroupOfNumbers: Number requested from file: 100
GroupOfNumbers: Number reputedly available: 62
GroupOfNumbers: line: 106
GroupOfNumbers: file: group.cc
GroupOfNumbers: exit: 4
```

Here, "bar" is the name of the file I specified when calling the constructor; I asked for "100" entries, but the first entry on the first line of the file indicated that only "62" entries were available for the initialization.

exit(5)

Finally, the number of entries actually available in the file may fall short of the number reputed to be in the file if the file has not been set up properly. Again, the `ifstream` variable is your friend and can help detect such a situation. If, during the course of the initialization, you discover that there is an insufficient number of entries in the file, despite the initial promise that a sufficient number was available, you are required to:

- send a message to standard error of the form specified below and
- exit with a return value of 5.

The message to standard error (`cerr`) should be of the following form:

```
GroupOfNumbers: Insufficient number of entries in file: bar
GroupOfNumbers: Number requested from file: 24
GroupOfNumbers: Number reputedly available: 62
GroupOfNumbers: Number actually available:  4
GroupOfNumbers: line: 123
GroupOfNumbers: file: group.cc
GroupOfNumbers: exit: 5
```

Again, "bar" is the name of the file I specified when calling the constructor; I asked for "24" entries, the first entry on the first line of the file indicated that "62" entries were available for the initialization, but I could find only "4" entries in the file.

One more accessor

To make our testing—and perhaps yours!—easier, you are to implement one new accessor for the class:

```
void writeGroupOfNumbersTo(const char* fname) const;
```

Once you have finished reviewing how to work with files to implement the new constructor, implementing this accessor should be just a minor perturbation of the method `displayGroupOfNumbers()`. Just remember that since you are writing *out* to the file (rather than reading *in* from the file), you need an *output* stream variable of type `ofstream` (rather than an *input* stream variable of type `ifstream`). You also are required to close the file `fname` before returning from the method.

When done, an admissible output file must look the same same as an admissible input file.

Some practical matters

Timing execution

On the Web page for the project you will find a copy of the file `timing.c`, which contains an ANSI C library for timing any portion you designate in either a C or a C++ program running under a Unix/Linux operating system.[2] Again, because C++ is "backwards-compatible" with ANSI C, you can compile `timing.c` using

[2] My thanks to Professor Michael Lewis for providing me with a copy of `timing.c` and for a useful discussion on various ways to determine CPU time on a Unix/Linux system.

`g++`, just as you would any standard `.cc` file.

For your purposes, the important thing is that this library contains two functions: `double user_time()` and `double system_time()`. The function `user_time` returns the CPU ("Central Processing Unit") time used while executing instructions in the user space of the calling process. The function `system_time` returns the CPU time used by the system on behalf of the calling process. The sum of these two times gives you a reasonable estimate of the total CPU time spent executing the section of code you designate – provided that the execution time was long enough for the timers to capture!

So how do you use these two functions? Returning to the example in the file `example.cc`, you can see that I used these two functions to estimate the time spent initializing the array `A`, the time spent displaying the contents of `A`, and the time spent writing the contents of `A` to a file. (If you compile and run this program, you are likely to observe that the time it takes to initialize `A` is an order of magnitude less than the time it takes either to display the entries in `A` or to write the entries in `A` to a file – which is indicative of the relative cost of computing versus input/output.)

With this example as a guide, for sufficiently large group of numbers you should be able to start timing the execution of portions of your own test program (e.g., How long does it take to create a large `GroupOfNumbers` object using either of your two new constructors for the class?).

Keep in mind that the Department of Computer Science maintains five machines (the "`bg`'s": `bg1`, `bg2`, `bg3`, `bg4`, and `bg5`) for running computationally expensive programs. Courtesy dictates that you *not* use these machines to edit your files, read your email, or any other such activities since the purpose of these machines is to serve as "compute engines." But if you are going to run larger/longer experiments on the Computer Science system, you may want to log in to one of these machines.

Compiling

The only thing new here is that you also will need to include the file `rng.c` when compiling the list of files for your test program. If you elect to run some timing tests (and I suggest you start practicing now, in anticipation of the projects for which you will be required to turn in timing results), then you also will need to include the file `timing.c` when compiling. So at the command line in your directory, execute:

```
g++ -Wall test.cc group.cc rng.c timing.c
```

The compiler will orchestrate the coordination between all these files. Remember that you must *not* include *any* source files (e.g., any `.cc` or `.c` files) inside any of your other `.cc` files. The C preprocessor command `#include` should be used by you only to include header files, and then only as appropriate.

Final instructions

To submit your file for grading, go to the directory in which your file `group.cc` resides and at the prompt in that directory enter:

```
~gpeng/bin/submit cs303 project1b
```

In spelling file names and identifiers which appear in these specifications (or in attached files) you must use the exact case and spelling.

Note that `submit` will only accept your file `group.cc` for grading. We will use the posted copies of `group.h`, `rng.h`, `rng.c`, and `timing.c`, along with our own test program, to grade your implementation of the methods for `GroupOfNumbers`.

Point breakdown for the project

To give you some idea of how to allocate your time and effort on this project, note the following distribution of points:

28 points for implementing:
`GroupOfNumbers(const char* fname, long total_requested);`

16 points for implementing:
`GroupOfNumbers(long lower, long upper, long total_requested);`

10 points for "past sins" from Projects 1A

4 points for implementing:
`writeGroupOfNumbersTo(const char* fname) const;`

2 points for implementing:
`Equilikely(long lower, long upper);`

Total: 60 points

The graders are allowed 10 points for "past sins," which come in two flavors. The first type of "past sin" includes any errors that were detected during the grading of Project1, Part A but remain uncorrected when you turn in Project 1, Part B. The second type of "past sin" includes any errors that managed to sneak by undetected during the grading of Project 1, Part A, but should have led to a point deduction—and will, now that the error has been uncovered.