Project 1, Part A
CS 303 Algorithms

Fall 2012

Due by: 8:00 a.m., Monday, October 1, 2012

In this project, which is the first part of a programming project that will continue throughout the semester, you are asked to implement a class that represents a group of numbers. Our ultimate goal will be to solve efficiently the *selection problem* introduced on the first page of your textbook. Over the remainder of the semester, the project will involve class implementation, random number generation, sorting, and a suite of algorithms, with varying time complexities, for solving the selection problem. The final purpose will be to demonstrate how good algorithm design can speed up problem solving.

Go to the Web page `www.cs.wm.edu/~va/CS303/projects_regs.html` – before starting this programming project – and review CSCI 303: Regulations for Programming Projects so that you understand what is expected of you.

C++

Those of you who are new to `C++` should first read §1.4-1.5 (pages 11–29) in your textbook. This gives a brief review of `C++` and introduces both the vocabulary – and the warnings – you will encounter in the discussion that follows.

Remember that my Introduction for the course suggests multiple suggested references for `C++`. Some of these are worth buying as a reference if you plan to be a computer scientist. Consider your options and make the arrangements that will work best for you.

Note that this semester we will use what can be informally referred to as `C++03`, rather than `C++11`, which is the current standard extending `C++` with new features. We'll do so to be consistent with the current edition of your textbook and to work with the current version of `GNU g++` installed on the Computer Science Linux systems, in an effort to minimize confusion since so many of you are new to `C++`. But do be advised that there are some attractive new features to `C++11`, as well as some changes to recommended practices and conventions, that you will want to familiarize yourself with should you go on to use `C++` in other settings.

`GroupOfNumbers` class

For this first project you are to begin the implementation of a `C++` class we will name `GroupOfNumbers`. The `C++` interface for the class `GroupOfNumbers` can be found in the header file `group.h`, which is available from the Web page for the project. You are *required* to include the file `group.h` in your file `group.cc`; this is how the compiler obtains the definition for the `GroupOfNumbers` class whose methods you are to implement. This means that at the top of your file `group.cc` you *must* include the line:

```
#include "group.h"
```

You are *not* allowed to modify the interface for `GroupOfNumbers` found in the header file `group.h`. As is standard in `C++`, your implementation of the methods of the class `GroupOfNumbers` are required to be in a separate file with the name `group.cc`. Section 1.4.3 in Weiss, *Separation of Interface and Implementation* discusses, and illustrates with a simple example, how the files should be set up.

When working with a group of numbers, the client programmer (otherwise referred to as the user) should be able to update the group by adding a new number. The user also should be able to modify a group by removing a number from the group. There should be a method for displaying the group of numbers. There should also be other simple accessor[1] methods: What is the size of a given group? Is a given group empty?

A conceptual feature worth noting is that inclusion in a group does *not* imply a *position* in the group. This is standard behavior for a group; all that matters is whether a given number is included in the group. You are required to use this fact to make sure that all your operations on the numbers in the group are as efficient as possible (i.e., enjoy the lowest possible time complexity without any additional increase in the space complexity).

We restrict our attention to an array-based implementation of `GroupOfNumbers` since this will best suit our future uses of `GroupOfNumbers`. As a consequence of our interest in using an array-based implementation, there are some details regarding the implementation of `GroupOfNumbers` that must be addressed. Specifically:

1. We allow the array for the group to be allocated *dynamically* so that the size of the array can be made to match the size we want, though we specify a default limit (represented using the constant `default_LIMIT`) on the *minimum* number of positions in the array allocated.

2. Since we are using dynamically allocated arrays, we need to make sure that we correctly implement the destructor, as well as the copy constructor and the overloaded assignment operator, both of which should make *deep* copies. These are the so-called "big three" in `C++`, which Weiss discusses in §1.5.5 on pages 23–26 of your textbook. I'll have more to say below.

3. Finally, since we are using the `C++ new()` function to dynamically allocate arrays within the default constructor, the copy constructor, and the overloaded assignment operator, we should verify that the allocation was successful before attempting to use the attached object and, if it was not, handle the error in a reasonable fashion.

The details regarding these new requirements are outlined in the sections that follow.

Dynamic allocation of the array for `GroupOfNumbers`

To allow ourselves to have arrays that are as big as we require – during the execution of any test program – we are going to dynamically allocate the array associated with each `GroupOfNumbers` class object. Thus you will notice that the private data member `_group`[2] is defined to be a pointer of type `long`. This means that you need to use the `C++ new()` function to dynamically allocate the array.

The following are two specific considerations to keep in mind as you implement the constructors[3] for the `GroupOfNumbers` class:

1. We would like to have a default limit on the number of positions available in the array. In fact, it would be useful to have a limit that specifies the *minimum* number of positions in the array that is allocated. So we define a constant `default_LIMIT` to specify this lower bound on the number of positions available in the `_group` array created for any `GroupOfNumbers` class object.

2. Since the array `_group` associated with each `GroupOfNumbers` object is allowed to contain any number of positions we desire, each instance of a `GroupOfNumbers` class object is going to have to keep track of `_LIMIT` for its data member `_group`. This is because the limit on the number of available positions in the array is now going to depend on how big we make the array for each individual `GroupOfNumbers` class object when we use `new()` to allocate the space for the array.

[1] Weiss discusses the difference between an *accessor* and a *mutator* under the subsection *Constant Member Function* on page 15.

[2] I'm making use of the classic `C` trick of prepending an underbar to a variable name so as to disambiguate what I mean when referring to the `GroupOfNumbers` member "group."

[3] You can find a review of constructor syntax in §1.4.2 on pages 12–15 of your textbook.

Initializing a `GroupOfNumbers` class object

We want to dynamically allocate the size of the array we think we will need so that we can work with a wide range of sizes for future testing. In the second part of this project, we will concern ourselves with efficient ways to initialize quickly a `GroupOfNumbers` class object with a large number of entries.

Keep in mind that 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`.

The default constructor takes the default argument `default_LIMIT` as an appropriate value to be used to dynamically allocate the `GroupOfNumbers` data member `_group`, but the user always is free to specify an alternate size, if so desired.

This raises an important issue you must consider. You are required to ensure that the array allocated has a *minimum* of `default_LIMIT` positions, so your implementation of this constructor should guarantee that this condition is satisfied if at all possible. If the limit requested by the user is less than `default_LIMIT`, the value used to allocate the `_group` array should be `default_LIMIT` – *not the user's request* – since your primary obligation is to satisfy the condition that the array must be able to hold a minimum of `default_LIMIT` numbers. However, if the limit requested by the user is greater than `default_LIMIT`, the user's choice should be honored on the assumption that the user has some idea of how large the `GroupOfNumbers` ultimately may be. Your implementation of the default constructor *must* satisfy this condition.

The big three

Since you are *dynamically* allocating the array used to represent the `GroupOfNumbers`, in `C++` you must deal with what are affectionately known as *the big three*: the destructor, the copy constructor, and the overloaded assignment operation (`operator=`). Weiss reviews the fundamental issues surrounding the big three in §1.5.5 on pages 23–26.

For instance, the programmer is in charge of explicit garbage collection in `C++`. Since we ultimately will work with extremely large groups of numbers, careful clean-up on your part is essential to ensure that you can test your implementations as required for the final project of the semester. If you do not start exercising discipline now and take care to deallocate memory when you no longer need it, then you will be unable to finish Project 4. Period. Don't let this happen to you.

The first place to start is with a proper implementation of the destructor. Since the only data member of the class `GroupOfNumbers` that is dynamically allocated using `new` is the array `_group`, this is the only data member that must be deallocated within the destructor using `delete`.

You also need to be sure that the copy constructor and the overloaded assignment operator work properly (i.e., make deep copies). As outlined in the specifications for `GroupOfNumbers`, the deep copies for the copy constructor and the overloaded assignment operator must lead to results that are identical in every respect, including the number of *available* – as opposed to *active* – positions in the array `_group`.

Notice that example from Weiss for overloading the assignment operator checks for assignment to self (what Weiss calls the "standard alias test"), which is crucial, as well as being standard. Weiss' implementation of the overloaded assignment operator includes the standard – and crucial – line `return *this;`. Also crucial is that you deallocate any memory that may have been dynamically allocated to the object on the left-hand side *before* commencing with the deep copy. Again, since in `C++` the programmer must handle garbage collection explicitly, and since memory *will* become an issue for this semester's series of projects, you will need to clean up carefully throughout your implementation of the class methods. Be warned that we will levy sizable point deductions if we detect that you have not.

Error handling within the methods for `GroupOfNumbers`

Because we are dynamically allocating space for the array, there is the possibility that an allocation will fail because there is insufficient memory available to satisfy the request. *You are required to check* all *memory* *allocations within your implementation of the methods for the class* `GroupOfNumbers`.

So now let's outline the error situation you are required to handle – failure to allocate an array of the size requested – and detail what your specific response should be. 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).

Verifying allocation using `try` blocks

The one possible error common to both of the constructors, as well as the overloaded assignment operator, for the class `GroupOfNumbers` is a failure to allocate an array of sufficient size. The question is: how to detect such a failure? The answer is to use a `C++` `try` block to `catch` the exception thrown by `new()` in the event of a failure to allocate memory. This is what you should be using to verify whether any allocation within a method of the `GroupOfNumbers` class is successful. In particular, if the `C++` `new()` expression cannot acquire memory from the program's free store, it throws a `bad_alloc` exception object. So every allocation you make within a `GroupOfNumbers` method should be made within a `try` block. Your method should `catch` the `bad_alloc` exception object (if one is thrown) and respond as outlined in the next section.

It should be acknowledged from the start that exception handling within constructors is a delicate business. An object is not considered constructed until its constructor has completed. Then and only then will stack unwinding call the destructor for the object to deallocate any memory that may be associated with the construction of the object. A constructor tries to ensure that its object is completely and correctly constructed. When that cannot be achieved, a well-written constructor restores – as far as possible – the state of the system to what it was before the attempted creation of the object since the destructor cannot do so.

Fortunately, our data members for the class `GroupOfNumbers` are limited in number and only one requires dynamic memory allocation, so we will simply exit, with an appropriate warning to the user, as outlined below.

All of the `C++` references I have given you contain chapters on exception handling. Parts of the above discussion were drawn from the seminal `C++` reference: *The C++ Programming Language*, Special Edition, by Bjarne Stroustrup.

Handling unsuccessful allocations using `exit(1)`

We have just discussed how to detect a failure to allocate an array of sufficient size by using a `try` block to catch the exception object `bad_alloc`. Now let's outline the required response required in the presence of such an error.

Since there is no graceful way to handle errors from within a constructor, we are going to rely on the good old-fashioned `C` standard library function `exit`, which terminates program execution when it is called.[4] Furthermore, the argument of `exit` is available to whatever process called the one that is being terminated, so the success or failure of the process can be tested by another process that is using the current process as a sub-process. Conventionally a return value of `0` signals that all is well; non-zero values usually signal abnormal situations. Thus, we can designate non-zero values (for this project, the value `1`) to signal any failure to allocate an array of the requested size.

To help put the use of `exit` in context, the use of

`return` *expression*

within `main` is equivalent to the use of

`exit(`*expression*`)`

[4]Most of my discussion of `exit` is drawn from the classic *The C Programming Language*, Second Edition, by Brian W. Kernighan and Dennis M. Ritchie, Prentice–Hall, Inc., 1988.

(e.g., `return 0` is equivalent to `exit(0)` and by convention signals that the termination was normal); `exit` has the advantage that it can be called from other functions (e.g., class methods).

For your implementation of the methods for `GroupOfNumbers`, if the allocation fails in *any* of the default constructor, the copy constructor, or 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`) should be of the following form:

```
GroupOfNumbers: bad_alloc
GroupOfNumbers: allocation of size 5000000000 failed
GroupOfNumbers: line: 9
GroupOfNumbers: file: group.cc
GroupOfNumbers: exit: 1
```

Here "`9`" in the line number in *my* file at which the failed allocation was attempted, "`group.cc`" is the name of the file in which the fail allocation was attempted, "`5000000000`" was the total number of entries requested by the user's program, and "`1`" is the exit code.

If the file `group.cc` is only one of a large number of files being used to create the final executable, then it helps to be specific about the file in which the error occurred. Since the allocation can fail in any one of three methods for `GroupOfNumbers`, it is helpful to isolate the particular method by including the line number at which the error occurred. Finally, it is useful to alert the user to the size of the allocation that failed; large numbers should not be a surprise, but small ones may indicate other problems within the user's implementation.

A useful aside

Two predefined names that `C++` gets from the `C` preprocessor are `__LINE__` and `__FILE__` (read as "underbar, underbar, LINE, underbar underbar" and "underbar, underbar, FILE, underbar, underbar").[5] You can use these two variables to indicate the file in which the error occurred (even if the name later changes) and the line number at which the error occurred (even if more lines are later to the beginning of the file). Be sure to adjust the line number to indicate the line at which the failure actually occurred – as opposed to the line at which the required output to `cerr` is being generated! Since most editors indicates line numbers, it should be easy to verify whether your adjustment is correct.

Some practical matters

Including files

It is considered bad form to include `.cc` files in other `.cc` files. When connections, in terms of the definition of a class, need to be made, they should be made using a `.h` file, which serves as the interface for the class. Thus, in any `.cc` file in which you declare a `GroupOfNumbers` object, you should include the file `group.h`, but you should *not* include the file `group.cc`. This means that your file `group.cc` is required to include `group.h`, as noted earlier. Furthermore, your test program (let's assume it is in a file called `test.cc`) needs to include `group.h` since you will be testing the correctness of your implementations of the methods for `GroupOfNumbers`. But your file `test.cc` should *not* include the file `group.cc`, even though it contains the implementations of the methods for the class `GroupOfNumbers`. Instead, the connection should be made at compilation time, as we'll discuss next.

Bottom line: do not include `.cc` files – only include header files.

[5] A useful bit of information drawn from Section 1.3 in *C++ Primer*, Third Edition, by Stanley B. Lippman and Josée Lajoie, Addison–Wesley, 1998.

Compiling

To compile a `.cc` file, without creating an executable, at the command line execute:

```
g++ -Wall -c group.cc
```

If all goes well, your prompt will return without any messages. But when you check the contents of the directory, you will find the object file `group.o`. The `-c` option is the signal to the compiler that you simply want to compile the file. Be warned that when you use this option, all the compiler will check is whether your file group.cc satisfies all the syntactical rules for `C++`. Passing this test is important, of course, but tells you absolutely nothing about the correctness of your implementation.

To ensure that your methods work as required, you will need to write your own test program (let's assume it is in a file called `test.cc`), which you will not submit for grading (we have our own test program). To create an executable that allows you to test your implementations of the methods for `GroupOfNumbers`, using your test program, at the command line execute:

```
g++ -Wall test.cc group.cc
```

Once again, if all goes well your prompt will return without any messages. But when you check the contents of the directory you will find the executable `a.out`.

To then run the executable `a.out`, at the command line execute:

```
./a.out
```

For more information on compiling and executing your `C++` test program on the CS system, review CSCI 303: Advisories for compiling and executing your `C++` program, which was linked to the Web page for Project 0, Part B and can be found at http://www.cs.wm.edu/ va/CS303/PR0/advisories.html.

Testing

A big mistake would be to assume that just because the file `group.cc` that you must submit compiles without warning that all is well. We will be testing – thoroughly – to be sure that the methods you were to implement work as required. If you do not test your implementation before submitting it, odds are quite good that you will earn few points for the project. To ensure a good grade, you must write your own test program and test *carefully* and *thoroughly* to be sure your implementations of the methods for the class `GroupOfNumbers` work as required.

Final instructions

To submit your file for grading, go to the directory in which your file `group.cc` (the only file that `submit` will accept) resides on the Computer Science system and at the prompt enter:

```
~gpeng/bin/submit cs303 project1a
```

In spelling the file name and identifiers that appear in these specifications you must use the exact case and spelling. Be sure to keep a copy of your file `group.cc` as you will need it for the second part of the project.

Before you submit your file for grading, be absolutely, positively sure that you have removed any extraneous output statements you may have inserted during the course of developing your implementation of the class methods. The *only* output that is specified for the class is as a result of the invocation of the method `displayGroupOfNumbers()`. If you leave other output statements in the implementation you submit for grading, you will lose points. The graders will write their own test program and run it "in batch" as a first pass toward grading your submissions. If you submit an implementation that generates extraneous (and thus, unexpected) output, it greatly impedes their progress on this first pass, so you will be penalized accordingly. Extra output may be helpful to you, but it is *not* helpful to the graders.

Point breakdown for the project

To give you some idea of how to allocate your time and effort on this project, note that the class methods you are to implement are worth the following points:

6 points (each):

```
GroupOfNumbers(const GroupOfNumbers & G);
~GroupOfNumbers();
GroupOfNumbers& operator=(const GroupOfNumbers & G);
```

5 points(each):

```
GroupOfNumbers(long defaultLimit = default_LIMIT);
bool add(const long& newNumber);
bool remove(const long& Number);
void displayGroupOfNumbers() const;
```

1 point (each):

```
bool isEmpty() const;
long total() const;
```

Total: 40 points