Assignment 7

Evolution in Action

`Answers.txt` Due:	10:00 PM,	Tuesday, March 25, 2003
`README` Due:	10:00 PM,	Tuesday, March 25, 2003
Runnable Code Due:	10:00 PM,	Wednesday, March 26, 2003
Final Code Due:	10:00 PM,	Thursday, March 27, 2003

This assignment gives you some experience writing C++ iterators, an understanding of some STL algorithms (and function objects), an opportunity to perform some benchmarking, and some experience in refactoring code. You will also develop a list class that can be used in place of the STL type *list<int>* in some (limited) circumstances.

Background — Genetic Algorithms

One of the more creative approaches to artificial intelligence is the genetic algorithm, invented by John Holland of the University of Michigan.

In brief, a genetic algorithm simulates the process of evolution by applying the usual rules of genetics to simulate natural selection. In real life, natural selection is the process by which members of a species that are well-suited to their environment tend to survive and breed and those that are ill-suited to that environment fair less well and are less likely to procreate. Future generations are made from the genes of those members of the species that are most fit for the environment in which that species lives. A genetic algorithm applies the mechanism of natural selection using a "fitness function" chosen by the programmer. For example, a simple fitness function might interpret the genes of an organism as the value of x in a complicated equation. The natural-selection process could then be tuned to prefer organisms that generate an output near zero, so that the survivors would eventually produce a solution to the equation.

Genetic algorithms were the first step in the current research area called "artificial life", and they have been used to successfully solve many problems that were otherwise intractable.

There are three basic processes in evolution: mutation, crossover, and selection. Mutation involves selecting a gene site and modifying it in some fashion, usually by replacing it with another gene. Mutation is very rare both in real life and in genetic algorithms. Crossover is the most important process in generating new organisms. It involves taking two gene strings (usually from two parent organisms), cutting them both at the same point, and splicing them together such that the head of the result comes from one parent and the tail from the other. Real genetic algorithms usually generate two children in this process, and may splice at more than one point, but we'll simplify things in our implementation.

The final step, selection, involves evaluating the organisms according to some criterion (the "fitness function") and choosing the ones that satisfy this criterion most successfully. In real life, selection is the harsh process of "survival of the fittest". In a genetic algorithm, the same method is used: the least fit organisms are discarded (i.e., "killed") without being allowed to reproduce. The fit organisms get a chance to reproduce before they die. As in real life, there is some randomness, so that a somewhat unfit organism has a chance of surviving to reproduce even though that may mean that a more fit organism is discarded. This randomness turns out to be important to the success of the method, because any two slightly unfit parents might (through crossover) generate an extremely fit child.

Scenario

To pay some bills over the summer, you've taken a job at *Lamarkian Enterprises*, whose eventual goal is to use powerful computers to evolve master mathematicians. Their lead programmer, Ginny F. O'Fenugreek, began exploring the problem space by writing a genetic algorithm to calculate square roots. Unfortunately, she left the company after writing the code claiming that she needed to spend more time building boats.

Management took a look at Ginny's code and discovered it had no `Makefile` and that almost all of Ginny's code resided in a single file. Worse, the code defines very few C++ classes, choosing instead to implement most of its functionality via top-level functions, even though there are some obvious ways in which the code could be broken into classes. In addition, one of the managers expressed concern at Ginny's extensive use of the C++ Standard Template Library (STL), claiming that the STL is inefficient, especially its *list* class, which he claims is wasteful because it is a template class and uses a doubly-linked list.

Your manager has handed you Ginny's code to clean up. You will have to

- Create a `Makefile` for the program
- Write an *IntList* class (storing a singly-linked list of integers) that can replace the uses of *list*<*int*> in the code
- Factor out logical components of the code into separate classes

Before You Begin

Download the source code for this assignment from the Homework area of the course website. The file archive to download is `cs70ass7.tgz`. On turing, you can download and unpack the file archive by executing

```
curl -O http://www.cs.hmc.edu/cs70/homework/cs70ass7.tgz
tar xzvf cs70ass7.tgz
```

This archive contains the following files and directories:

```
cs70ass7/natselsqrt.cpp
cs70ass7/natselsqrt-private.hpp
```

Current Data Structures

The program currently uses the following data structures:

Organism

An organism will be represented entirely by its gene sequence. Each element in the sequence will contain only a single integer from 0 to 9. In the current implementation of the code the gene sequence is represented using the type *list<int>*. The *Organism* type is simply a synonym for *list<int>* to make the code more readable.

Organism::iterator and *Organism::const_iterator*

At several places in the code, the program needs to cycle through the genes of the organism. At present this task is done using the above types (which, in the current code, are equivalent to *list<int>::iterator* and *list<int>::const_iterator*).

Colony

The *Colony* type is used to represent the population of organisms. It is simply a container type that holds *Organism* objects—in the current code, *Colony* is a synonym for *vector<Organism>*. Like the organism type, it currently has iterators.

Written Component

When answering the following questions, explain your answers clearly. Your answers should be placed in the file `Answers.txt` and submitted using the `cs70submit` system.

Several of the questions require you to make small changes to the code. In the coding component you will have to create a `Makefile`, but if you do not wish to do that yet, you can compile the code manually without a `Makefile` by executing: .

```
g++3 -g -Wall -W -pedantic -O2 -o natselsqrt natselsqrt.cpp
```

The `-O2` is "oh two", *not* "zero two"

(this command line includes the compiler option `-O2` to turn on optimization, which is useful for doing the performance comparison questions).

W1. Compile the code (with optimization) and then run the command

```
time ./natselsqrt -S 1 -p 3000 152399025 986902225
```

(a) What does this command line mean? (You can find out about the `time` command by typing `man time`, and about the arguments to the `natselsqrt` program by looking at the initial comments in `natselsqrt.cpp`.)

(b) What output does it produce? (If you run the command on a machine other than turing, indicate the processor and clock speed of the machine in question.)

W2. Comment out line 80 in `natselsqrt.cpp`, which reads

typedef *list<int> Organism*;

and add each of the following (in turn):

(a) **typedef** *vector<int> Organism*;

(b) **typedef** *vector<char> Organism*;

(c) **typedef** *string Organism*;

For each of the parts above, recompile the code (with optimization) and report the results from running

```
time ./natselsqrt -S 0 -p 3500 152399025 986902225
```

(When you are done, revert the definition of *Organism* to *list<int>*.)

W3. Lists and arrays have quite different representations. Why can we change *Organism* from being a synonym for *list<int>* to a synonym for *vector<char>* and *string* without causing compiler errors?

W4. The code uses a functor. What is a functor? Where is it created? (See Stroustrup, Section 18.4 for a discussion of functors.)

W5. Explain how the findBest works (i.e., find out what the STL algorithm min_element is used for and explain why it is the right tool for this problem).

W6. There is a comment at the top of the naturalSelection function explaining that early versions of the code used sort, later versions used partial_sort, and the current version uses nth_element.

(a) Highlight the similarities and differences between the STL algorithms sort, partial_sort, and nth_element. (You will almost certainly need to look these functions up, either in Chapter 18 of Stroustrup or on the web—see the resources page on the class website.)

(b) Explain why the code evolved from `sort` to `partial_sort` to `nth_element`, including why the code is still correct and whether any efficiency gains are likely.

(c) Modify the code to use each of the other methods in turn and run the code to determine whether each change improves performance in practice. Summarize your findings.

W7. Make sure you have written the `README` file for your code (it is due when the rest of the written work is due, not when the code is due, so you will have to have done a significant chunk of the coding component by the written work deadline).

Coding Component

These coding questions are designed so that you can submit your code after completing each question. You will, however, only be graded on your final submission.

C1. Create a `Makefile` for your code. (You will need to keep the `Makefile` up to date as you make changes.)

C2. Create the files `intlist.hpp` and `intlist.cpp`, implementing an *IntList* class. You should write your *IntList* class such that if the definition of the *Organism* type is changed to

```
#include "intlist.hpp"
typedef IntList Organism;
```

the program will still compile, run, and produce the same output as before.

The overall structure of your *IntList* class should be similar to the *StringStack* class seen during lectures (which you may use as a starting point). Thus, there will be three classes involved, *IntList*, *IntList::Node*, and *IntList::Iterator*. However, unlike the stack class, you will need both a head and a tail pointer in the header (in order to support `push_back` efficiently).

Your linked-list class must be named *IntList* and must support the following operations:

- A default constructor.
- A `swap` operation.
- A copy constructor.
- A destructor. The destructor must clean up properly; in other words, it must empty the list.
- An assignment operator.

- A push_back function that inserts a single integer at the tail of the list. This function should be declared as

  ```
  void push_back(int value);
  ```

  Your push_back function *must* operate in $O(1)$ time. You have already done a similar implementation in CS 60 when you examined queues.
- Two typedefs, defining the types *iterator* and *const_iterator* as synonyms for *Iterator*.[1]
- Two inner classes, *Iterator* and *Node*, where *Node* is private.
- A begin function that returns an iterator that refers to the start of the list. This function should be declared as

  ```
  iterator begin() const;
  ```

- An end function that returns an invalid/past-the-end iterator,

  ```
  iterator end() const;
  ```

Note: you may not change the function and type names given above. You may, however, write additional member functions such as push_front, front, back, pop_front, and reverse if you wish. (If you have written your list class correctly, it will not be not possible to write pop_back efficiently, so there is little point in supplying this function.)

The *IntList::Iterator* class must provide at least the following operations:

- A copy constructor
- An assignment operator
- A destructor
- An equality test (**operator**==) and an inequality test (**operator**!=)
- A preincrement operator (**operator**++)
- An **operator*** that returns an *int&* (so that the integer in the current position can be modified if necessary)

Compile and test your code thoroughly. Your *IntList* class will be tested separately, so it is important that it work correctly.[2]

After you have created and tested your *IntList* class, you can either continue to use it in the genetic-algorithm code or switch that code back to using the STL's *list<int>* type. Using your *IntList* may result in more comprehensible error messages while you are working on coding the next part, whereas using *list<int>* will insulate your later code from any bugs in your list class.

1. In an industrial-strength implementation, we would define separate types for *iterator* and *const_iterator*, but in this assignment doing so is more trouble than it is worth. As a result, people will be able modify objects even when they only have a *const_iterator*.

2. You may find that your *IntList* class seems slower than the STL's *list<int>* class, even if you have written clean, elegant, and efficient code. The reasons behind this slowdown are curious indeed—a small reward is available to any student who can discover why the slowdown occurs and whether it can be prevented.

C3. The code that has been provided to you uses many top-level functions and defines almost no classes. It could be argued that the code only provides the *NatSelEnv* class because it is very convenient to pass a function object to the STL algorithms.

Examine the code to discover the logical relationships between the functions and then break the code up into separate files and classes. You should create at least two new classes reflecting the logical structure of the program. (A solution involving four new classes is quite possible.)

In your final code, you should find that the overall code looks simpler and is easier to follow. If you have done things properly many of the functions will have fewer arguments.

Note that the execution behavior of your code must be *exactly the same* as the existing code—this requirement means that you must take care when making changes involving the random-number generator to make sure that the same numbers are generated. (The random-number functions used by the code are described in the Unix manual pages—`man random`).

C4. Clearly describe your design in your `README` file. Remember that the deadline for the `README` is earlier than the deadlines for code. You can make minor corrections to the `README` file when you submit your code.

You should submit your code using the electronic submission system described in the *Homework Policies* handout. The deadline for your `README` file is two days before the code deadline. You may resubmit your `README` when you submit your code, but you will lose points if there are substantial changes in the revised version.

Testing

Testing is your responsibility. If you leave all testing to the automated testing on Wednesday night, you may not have enough time or enough information to fix the bugs in your code. You should test your program a number of times under different conditions.

In its default condition, the program is nondeterministic (i.e., two successive runs may produce different results, especially if you use the `-d` switch to see the program's debugging output). To make testing easier, the program accepts a switch that makes it deterministic. If you use `-S` *n*, where *n* is an integer, the random seed will be set to that value. Specifying the random seed will allow you to control the program's behavior so that you can reproduce bugs.

You will also find it instructive to run the program with the `-d` switch, and to run it for many different values of the `-g`, `-m`, `-p`, `-r`, and `-s` switches. Judicious reading of the comments, together with experimentation, will reveal the purpose of these switches and how they interact.

Sample Runs

To make it clearer how the program is used, here are some sample runs from executing `natselsqrt` on turing (the same values are returned if the code is run on Mac OS X, but Linux-based systems will give different values). First, we can approximate the square root of 2, 000,000 (which is just 1000 times the square root of 2):

```
unix% ./natselsqrt -S 12345 2000000
0001414 * 0001414 = 1999396
```

If we start with a different random seed, we get a different result:

```
unix% ./natselsqrt -S 34567 2000000
0001413 * 0001413 = 1996569
```

A third attempt gives a pretty bad answer:

```
unix% ./natselsqrt -S 38 2000000
0001299 * 0001299 = 1687401
```

Finally, we can change the number of generations (`-g`), the mutation rate (`-m`), the population size (`-p`) the selection pool size (`-s`, which should be smaller than the population size), and the number of randomly-chosen survivors (`-r`, which should usually be pretty small), and run with debugging (`-d`):

```
unix% ./natselsqrt -S 0 -g 100 -m 0.2 -p 96 -s 36 -r 12 -d 2000000
Generation 0: 0039958
Generation 1: 0006414
Generation 3: 0002319
Generation 4: 0002063
Generation 5: 0000459
Generation 9: 0001404
Generation 12: 0001405
Generation 15: 0001415
Generation 18: 0001414
0001414 * 0001414 = 1999396
```

Note:

You can think of the running time of the programas being $O(population \times generations)$.[3] Don't use huge numbers or you' ll wait all day! If you don't specify the `-S` switch, you will get different results every time you run the program. That's a feature, not a bug.

3. You might want to try to analyse the complexity yourself and determine whether this is actually the correct bound.

Tricky Stuff

As usual, there are some tricky parts to this assignment. Some of them are

- Be sure to read the code in `natselsqrt.cpp` before you start, so that you understand the requirements placed on the *IntList* and *IntList::Iterator* classes.
- Before you try to write the iterator, it would be wise to debug the list code itself. To help with that task, you will probably want to write a special test driver program.
- Be sure your *IntList* destructor, copy constructor, and assignment operator are working before you try to run the main program. If you don't debug them in isolation, you will experience strange bugs that will be hard to find.
- Remember that the iterator access operator (**operator***) must return an integer by reference (*int&*). Otherwise you won't be able to get the mutation operator to work.
- Remember that `push_back` must run in $\mathrm{O}(1)$ time. You will be penalized if its complexity is $\mathrm{O}(n)$.
- Refactoring the code (breaking it up into separate files/classes) may seem daunting at first, but there are some fairly obvious lines you can draw between different parts of the code. Remember that you can plan things out on paper first—you don't have to try to keep all the details in your head.
- If you save a copy of the original executable, you'll be able to test whether your revised version of the program always behaves the same way.
- If you believe there may be bugs in your *IntList*, you can temporarily use the STL's *list<int>* and see whether the bug persists.
- The new classes you wrote need not exactly match the behavior of the types used in the current code. The operations your provide for each class need to be meaningful for that class. If, for example, you decided you needed a *Television* class, I would not be pleased to see it having a `push_front()` member function. Televisions don't "push front".
- The behavior of the program is extremely sensitive to the random numbers it gets from the `randomInt` function. These random numbers in turn depend on the number of times `randomInt` has been called. If you have trouble matching the sample output or the behavior of the original code, check to see whether you're creating scratch organisms that cause extra calls to `randomInt`.