Project 3: Spyware

Exploration of algorithm runtime using a trojan horse comparison operator

Educational Objectives: After completing this assignment, the student should be able to accomplish the following:

- Define *function class*, *function object*, *predicate class*, and *predicate object*
- Define *generic algorithm*
- Design and implement function and predicate class templates
- Design and implement generic algorithms
- Use function objects (and predicate objects) in client programs
- Measure the number of calls made to an atomic operation in an algorithm
- Use the measured counts of atomic operations to empirically corroborate known theoretical asymptotic runtimes of algorithms
- Discuss the advantages and disadvantages among various implementations of algorithms

Operational Objectives: Create the predicate class template `fsu::LessThanSpy<T>` (in file `compare_spy.h`) and generic algorithms `seq::g_lower_bound` and `seq::g_upper_bound` (in file `gssearch.h`).

Deliverables: Three files `compare_spy.h`, `gssearch.h`, and `report.txt`.

Background: Order relations in C++

C++ uses the mathematical term *strict weak ordering* to state the assumption made about order relations that are used in:

A. Generic algorithms that use an order predicate, including sorts, searches, and set algorithms (union, intersection, difference, merge)
B. Ordered associative containers that use an order predicate to maintain their order structure (Set, Map, MultiSet, MultiMap)

In other words, everywhere. A predicate p is a *strict weak ordering* iff it satisfies these properties:

i. For all x, p(x,x) is false (p is irreflexive)
ii. For all x and y, if p(x,y) then !p(y,x) (p is anti-symmetric)
iii. For all x, y, and z, if p(x,y) and p(y,z) then p(x,z) (p is transitive)
iv. For all x, y, and z if x,y are incomparable and y,z are incomparable then x,z are incomparable (incomparability is transitive)

The first three of these are familier and are satisfied by our notion of order in numbers. The fourth, "transitivity of incomparability", requires some explaining. First we need to define the term: x and y are *incomparable* means that both p(x,y) and p(y,x) are false. In order terms, neither x < y nor y < x is true. Normally we would conclude that x and y are the same. But in a programming context, we often have two things that may have the same key but are not identical, so saying that !(x < y) and !(y < x) means only that x and y are equivalent in the sense of having the same key. Typically, this equivalence is acknowledged by the overload of the equality operator for the type, which would return true if x and y have the same key.

So, in a C++ context, we can replace the esoteric fourth axiom by an implicit assumptions about operator==, as follows:

i. For all x, p(x,x) is false (irreflexivity)
ii. For all x and y, if p(x,y) then !p(y,x) (anti-symmetry)
iii. For all x, y, and z, if p(x,y) and p(y,z) then p(x,z) (transitivity)
iv. For all x and y, if !p(x,y) and !p(y,x) then x == y (trichotomy)
v. Operator== is an equivalence relation - reflexive, symmetric, & transitive

("Trichotomous" meaning that exactly one of {p(x,y), p(y,x), x == y} must be true.)

In C++ when we use an order predicate for a type T in any of the situations in A or B above, it is assumed to be paired with a definition of operator== for type T so that axioms i ... v are satisfied. Mathematically, we could just define operator== to mean incomparability with respect to p, but in practice it is usually more convenient to define operator== directly and then verify axioms i .. v. In fact, it is typically clear "on its face" that the properties i .. v hold, and no formal proof is given.

We could contrive various contexts in which these axioms fail, and there would be really strange and unpredictable behavior from the normally reliable items in A and B above.

Procedural Requirements

1. The official development/testing/assessment environment is specified in the Course Organizer.
2. Copy all of the files in `LIB/proj3/` into your `proj3` directory, along with `LIB/scripts/submit.sh`. You should now have these files:

```
sort_spy.cpp      # client of LessThanSpy<T>
search_spy.cpp    # client of LessThanSpy<T> and the seq:: search algorithms
ranuint.cpp       # generates files of random numbers to use for sort data
hsort.cpp         # simple sort unitility if you need sorted data
submit.sh         # generic submit script
deliverables.sh   # submission configuration file
```

   The first two are clients of your deliverables. The other code files will help you generate test data.
3. Create the files `compare_spy.h` and `gssearch.h` and test them thoroughly to be sure they function correctly.
4. Use the supplied "_spy" clients (and any modifications/upgrades you may choose to create) to generate runtime data for the included sort and search algorithms
5. Write a brief report answering questions (given below) about these algorithms using the collected runtime data. Include your usual `log.txt` information as an appendix to `report.txt`.
6. Turn in the deliverables using the submit script.

   Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive two confirmations, the second with the contents of your project, there has been a malfunction.*

Code Requirements and Specifications: `class LessThanSpy<T>`

1. `LessThanSpy<T>` is a predicate class template.
2. `LessThanSpy<T>` objects maintain a count of the number of times operator() is called since the object was created or the last time Reset was called.
3. The `LessThanSpy<T>` API is as follows:

`bool operator()( const T& t1 , const T& t2 )`	returns true if and only if `t1 < t2`, where `T` is the template parameter.
`void Reset()`	Sets the internal counter to zero
`size_t Count() const`	Returns the number of times operator() is called since the object was created or the last time Reset was called.
Default constructor	Sets the internal counter to zero.

4. Class LessThanSpy<T> is defined and implemented in the file `compare_spy.h`.
5. Be sure to follow best practices defining and implementing the class.

Code Requirements and Specifications: `seq::g_lower_bound` and `seq::g_upper_bound`

1. `g_lower_bound` and `g_upper_bound` are generic algorithms.
2. `g_lower_bound` and `g_upper_bound` operate with forward iterators.
3. A pre-condition for successful operation of `g_lower_bound` and `g_upper_bound` is that the range to which they are applied is sorted (using the same predicate as used in the algorithm call).
4. `g_lower_bound(beg,end,t,p)` returns the lower bound of `t` in the range `[beg,end)` (using the predicate `p` to determine order).
5. `g_upper_bound(beg,end,t,p)` returns the upper bound of `t` in the range `[beg,end)` (using the predicate `p` to determine order).
6. `g_lower_bound` and `g_upper_bound` are in the namespace `seq`.
7. `g_lower_bound` and `g_upper_bound` are prototyped and implemented in the file `gssearch.h`.

Report

The report is a plain text file. Do not submit any file with special formatting in it, such as Word or rtf or pdf or html. The assessment

process will be able to read text files only.

The report file must be named "`report.txt`" for the submit script.

Begin your report before you even start coding, because some of the questions pertain to the files as distributed and before modification. Keep your log entries in an appendix to the report named "Work Log".

Start your report with file header info as follows:

```
COP 4530 Project 3
Using a Trojan Horse Comparison Operator to Analyze Algorithm Runtimes
<your name>
<your CS username>
<your FSU username>
```

Answer each of the following questions and/or supply evidence that you have performed the required tasks. Be sure to number the questions and repeat the question in the report prior to answering.

1. *Let f*(*n*) = *n* (*n* + 1) / 2. Of the following possibilities, state which are true about *f* and of those, which one best describes the asymptotic class of *f* ? (Be sure to argue your answers.)

a.	$\Theta(n^2)$	b.	$O(n^2)$	c.	$\Omega(n^2)$
d.	$\Theta(n\ (n+1)\ /\ 2)$	e.	$O(n\ (n+1)\ /\ 2)$	f.	$\Omega(n\ (n+1)\ /\ 2)$
g.	$\Theta(n^3)$	h.	$O(n^3)$	i.	$\Omega(n^3)$
j.	$\Theta(n \log n)$	k.	$O(n \log n)$	l.	$\Omega(n \log n)$
m.	$\Theta(n)$	n.	$O(n)$	o.	$\Omega(n)$

2. Which of these statements are best supported by data obtained using search_spy (argue your answer):
    a. fsu::g_lower_bound has asymptotic runtime $\Theta(\log n)$.
    b. fsu::g_lower_bound has asymptotic runtime $O(\log n)$ but not $\Omega(\log n)$.
    c. seq::g_lower_bound has asymptotic runtime $\Theta(n)$.
    d. seq::g_lower_bound has asymptotic runtime $O(n)$ but not $\Omega(n)$.
3. State an asymptotic runtime for each sort algorithm that is best supported by data gathered with sort_spy. Argue your answer using collected data, and also discuss characteristics of the algorithm body that support your answer.
4. Describe two scenarios, one under which the namespace fsu search algorithms are appropriate and one under which the namespace seq versions are appropriate.

Hints

- All of the programs distributed with this assignment can be compiled with one command line (or the c4530 macro), so no makefile is necessary.
- The files `LIB/tcpp/compare.h` and `LIB/tcpp/gbsearch.h` are good models from which to start your coding.
- Be sure that both `LIB/tcpp/compare.h` and `./compare_spy.h` can be used by the same client.
- Under "best practices" - be sure you remember how to initialize class variables in a constructor.
- The names `g_lower_bound` and `g_upper_bound` are used in at least three namespaces. The algorithms in general accomplish the same things but are implemented in different ways.
    i. In namespace `fsu` they are implemented iteratively using binary search, as in the lecture notes and the STL, and require random access iterators (file: `LIB/tcpp/gbsearch.h`).
    ii. In namespace `alt` they are implemented recursively as divide-and-conquer algorithms, but still require random access iterators (file: `LIB/tcpp/rbsearch.h`).
    iii. In namespace `seq` they are implemented iteratively using sequential search and operate with the less restrictive forward iterators (file: `./gssearch.h`).
- Recall that generic algorithms are special function templates that operate on iterators rather than specific containers.
- Both `sort_spy.cpp` and `search_spy.cpp` can be run in batch mode with a command file as an argument (not redirected). This may be handy when you get to collecting data.
- When running your spy programs, you may want to limit data sizes to 100,000 unless you "comment out" the calls to things with quadratic runtime.
- Example executables are available in `LIB/area51/sort_spy_i.x` and `LIB/area51/search_spy_[type]_i.x`
- In creating your report, keep in mind that the asymptotic category of a function depends on all input sizes, ignoring the first few. Therefore, when using actual runtime data to corroborate a given statement about asytmptotic runtime of an algorithm, data on various sizes is necessary. What one is looking for is a trend, not specific values.

Acknowledgement: Thanks to former student and recent grad Debbie Roy for suggesting an assignment like this for COP4530.