Paul Nickerson
COP3530
1087
11/25/2014
Project 1, Parts 1-3

I hereby affirm that the following work is my own and that the Honor Code was neither bent nor broken:

_____

Part 1 Learning Experiences

Part 1 was the perhaps the most challenging aspect of this project since I was very new to C++, and, in fact, my first exposure to modern C++ was through the C++ Primer I read before the semester. I was still getting used to how references worked and the various ways which modern C++ improves the clunky and error-prone C syntax.

Of the four list types, CDAL was, unsurprisingly given its complexity, the most difficult list type to implement. However, I believe it is a very useful list that I will consider using in future projects, since it offers fast random access benefits of a simple array while not requiring contiguous memory, which is useful since I frequently encounter large data sets in my research which may not fit in memory contiguously.

It was during part 1 that I developed a fuzzer which runs random operations against each list type and checks consistency among each (see SSLL Testing Strategies for a description of the fuzzer). This tool proved *extremely* valuable in uncovering bugs. I used it extensively while implementing parts 2-3 to catch bugs early on. My workflow typically looked like 1) write some code, 2) compile it, 3) (optionally) add/update a fuzzer operation to ensure the code path gets hit, 4) run the fuzzer for a while, 5) fix/refactor the code from (1), 6) repeat.

While part 1 was challenging and time consuming, as a learning experience the process was very valuable. Since in the course of my research I do a lot of C++ development, I now incorporate lists into my work quite often. Previously I had never used a linked list for anything and the extent of my list toolset consisted of simple arrays. I now have a much better grasp on the underlying list data structures and how to go about choosing one which meets my needs.

Part 2 Learning Experiences

Part 2 was much easier than part 1 since I had the list classes in place and just needed to add/update things. After updating all the relevant int types to size_t, I have now developed the habit to use size_t almost exclusively when I need a positional unsigned integer, since it alleviates the extra mental overhead of keeping track of overflows. Size_t is almost always big enough to hold positional integer values that I need, and it has the added benefit of preventing hacky code practices like using -1 as a special case value.

I really enjoyed implementing the iterators. I now use iterators all the time and am trying to develop some functional programing habits. When working with large data sets, it is very useful to have a custom iterator class which processes each entry incrementally, rather than fitting the whole thing in memory. For example, in my research one issue I'm facing is external sorting; I have several large files stored on disk in order, and I need to merge them. I solved that by treating each input file as an iterator and using a priority queue to externally merge their contents into a new, sorted iterator, whose value is processed and passed to yet another iterator, etc.

Implementing the iterators presented a great opportunity to refactor a ton of code. Since traversing the list is a common pattern – for example printing each item, copying a list, checking for the existence of an arbitrary item in the list – I went through and converted as many disparate traversal code paths as I could into methods that relied on iterators. This meant lowered complexity and fewer bugs. In fact, it allowed certain methods, such as print() and contains(), to be implemented using the same exact code among the different lists.

Part 3 Learning Experiences

While part 3 wasn't particularly difficult, generating the informal documentation was a huge, time-consuming pain in the neck. I hate writing documentation, so try to write self-documenting code using things like longer, verbose variable/method names, so that reading the code is nearly as natural as reading actual documentation.

That being said, writing the informal documentation for each method was very helpful. While doing so, I found numerous opportunities to refactor code and add thorough exception handling, as well as to scrutinize assumptions I made when writing the methods. So the process of writing the documentation was also a practice in effective code-cleaning. After the documentation was written, knowing what to target with CATCH test cases was trivial.

From the informal documentation, I wrote several sets of CATCH testcases, which collectively should provide a very high degree of code coverage. As described in the fuzzer write-up (see SSLL Testing Strategies), developing the lists alongside testing with the fuzzer led to lists which all behave similarly despite their storage differences. As such it was only necessary to write a single testing suite to target each list type. Prior to zipping the deliverables, my directory structure contained symbolic links which effectively duplicated the testcase CPP files to the various list directories. Each of the testcase files –

random_access.cpp, contains.cpp, iterators.cpp, replace.cpp, insert.cpp, remove.cpp, and copying.cpp – contain a variety of testcases related to the category referred to by the file name. All public members are covered by the testcases. I found that the key to effective testing is to write testcases which treat the lists as pure ADTs (that is, without making any assumptions regarding their implementation behavior aside from attempting to hit generic code paths such as growing/shrinking the list), while the lists themselves rely heavily upon exception handling to validate their internal state. Incidentally I have found that treating classes I am writing as pure ADTs, with the client knowing zilch about the underlying implementation, naturally leads to much cleaner and maintainable code.

The subscript operator introduced another opportunity to refactor. For example, the item_at method is functionally equivalent to passing the subscript operator through the copy constructor. Also, the replace just method does a std::swap of the input element and the subscript operator. Treating these two methods as such allowed me to use the same code to implement them among each of the four list types.

1. Does the program compile without errors?
    1. Yes
2. Does the program compile without warnings?
    1. Yes
3. Does the program run without crashing?
    1. Yes
4. Describe how you tested the program.
    1. Using a list fuzzer described in SSLL Testing Strategies, by manually writing unit tests with the CATCH framework, and by implementing extensive exception handling within the list internals
5. Describe the ways in which the program does *not* meet assignment's specifications.
    1. None
6. Describe all known and suspected bugs.
    1. None
7. Does the program run correctly?
    1. Yes, all output is as expected