Lab 9

17 October 2023

Today you'll start development on a project that provides a (small) library of classes to a potential user. Specifically, it will be a group of classes that store elements without duplication—a set.

Sets

What is a set? Its fundamental properties are that it

- contains elements,
- does not count or distinguish duplicates, and
- does not guarantee anything about their order.

That means that it can't, for instance, retrieve an element at a particular index, because indices imply order and sets don't (promise to) preserve order. Think about it, and in your notebook, write down the key methods that a `Set` class will have to have. There are three or four really important ones, plus a few that would be more optional. Make sure to mark which ones would be `const`.

Once you're pretty confident about your list, write a file `Set.h` that encodes this information in the form of valid C++ method headers. It will look a lot like the generic header file from the book that covers all lists (`List.h`); in particular, unlike an implementation (such as `AList.h`), it won't have instance variables (ie no `private` section) and the methods won't be defined. We would like to make our `Set`s able to hold any type of element; recall that we can use templates for that. To make that happen, you just need to precede the class header with

```
template <class Thing>
```

and then use `Thing` as the name of the type the `Set` would hold, whenever you add a value or search for a value or anything like that. (Feel free to use a different name than `Thing`—in class we've mostly used `T`. Up to you!)

Because our `Set` class is meant to define an interface, we want to mark its methods as "pure virtual": the implications of this we'll discuss in class, but the mechanics simply involve marking it `virtual` and setting the body to zero. That is, if you had written a method

```
int getSomeValue() const;
```

you would mark it pure virtual by writing

```
virtual int getSomeValue() const = 0;
```

Go ahead and do that (add `virtual` and `= 0` to each of your method declarations) in `Set.h`.

Then, write a simple test file called `test_VSet.u` that, for now, just `#include`s your `Set.h` file and has an empty test suite. Compile that file to confirm that your header has no errors.

Test cases

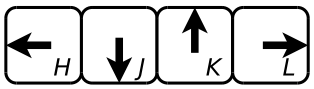
Now that we have a public interface, we can start planning our test cases. In your notebook (*not* yet in the `.u` file), describe a few useful examples (which will eventually become the test fixture). Then, write some sequences of method calls, using those examples, that collectively verify that a `Set` would correctly contain its elements, and does not count or distinguish duplicates.

At this point, run the handin script (with assignment `lab9`) on your directory for this lab, or just on the `Set.h` and `test_VSet.u` files, so I can get a sense of what your plan is, while you move on to the FOTD section. I'll try to check in relatively soon and give you feedback on it.

Vim FOTD: movement keys

Vim responds to the arrow keys and keys like PageUp and PageDown, but there are a number of additional keys that can be pressed in command mode to move around the file. Open one of the files you have lying around and try some of them out.

Key(s)	**Movement**
`h`	Left one character
`j`	Down one line
`k`	Up one line
`l`	Right one character
`^`	To beginning of current line
`$`	To end of current line
`Ctrl-F`	Forward one page (screen)
`Ctrl-B`	Back one page (screen)
`G`	To last line of file
`#G`	To line # (e.g. 1G to go to top of file or 23G to go to line 23)
`w`	To beginning of next punctuation-delimited "word"
`W`	To beginning of next whitespace-delimited "word"
`e`	To end of this punctuation-delimited "word"
`E`	To end of this whitespace-delimited "word"
`b`	To beginning of this punctuation-delimited "word"
`B`	To beginning of this whitespace-delimited "word"
`}`	To next (batch of) blank line(s)
`{`	To previous (batch of) blank line(s)
`%`	To matching paren/bracket
`[(`	To previous unmatched left paren
`[m`	To start of current function



Some of these are more mnemonic than others, of course. The first four are not mnemonic at all, but super-convenient once you've got them in muscle memory, because they're right in the home row, so your fingers don't have to go anywhere to type them.

So what, right? Well, all of the delete commands that you learned in earlier labs were special cases of a rule: `d` plus a movement command deletes from "here" to wherever that movement goes. So, `d1G` deletes to the top of the file. And `d%` deletes everything between this paren and the matching one. Since the `p` command only pastes the most recently-deleted thing, it's very helpful to be able to delete everything you want to "cut" all at once. Same goes for the `y` ("yank", i.e. copy) commands. The re-indent command (`=`) is another one that works with an arbitrary movement: `=%` reindents everything between "here" and the matching paren or bracket, while `=G` reindents everything from "here" to the end of the file, and so on.

There's no need to memorise all the movement commands right now, of

course. A couple might stick, but for the rest, even if you don't remember the command, you'll remember it exists, and you can always come back and refer to this sheet.

Starting an implementation

Eventually, we'll write `Set` implementations that run efficiently and mimic the standard implementations, but before we worry about efficiency we have to aim for correctness. Our first implementation will be `VSet`, and will use the `vector` built in to C++ to store the data.[1] Its main inefficiency will be that when the user requests to add an element, it will have to check to see whether it's already in the set, and only add it if it's not already there.

Edit a file `VSet.h` to start working on the class definition. The `VSet` will declare itself to be a subclass of `Set` by using the following class header:

```
template <typename Thing>
class VSet : public Set<Thing>
```

(again, feel free to use a word other than `Thing`, and it doesn't have to be the same placeholder name that you used in the `Set` definition). Inside the class, you'll start by making a private instance variable that is a `vector` to hold the data; and then for every pure virtual method in the `Set` definition, you'll write a stub method in the `VSet` definition (for now). Note: because it is a templated class, *all* the code for `VSet` will go in the `.h` file. Other than the "`: public Set<Thing>`", the `.h` file will be structurally quite similar to the `AList.h` and `LList.h` files we've been working on in lecture.

Testing it

Now that you have the bare bones of an implementation, go ahead and type the test cases you wrote out earlier into the file `test_VSet.u` you created earlier.

Because the main constructor for any set (or any collection) should take no parameters and construct an empty instance, your fixture will have to have this overall structure:

[1] Note that a `VSet` object "has-a" `vector`, but "is-a" `Set`.

```
fixture:
  VSet<int> example1 = VSet<int>{};
  // plus more like that, but all (initially) empty
  // also probably with better names

  setup
  {
    // code to add the contents to each of your fixture examples
  }
```

There should certainly be multiple examples, and some of them could be sets of string or whatever, and you should use names more descriptive than "`example1`". Make sure to have a couple different sizes of sets, and with different properties—e.g. same values added in different orders, or duplicate values.

Once you have your test file typed in, compile it and run it to confirm that everything compiles. If you run your test now, most if not all of the tests will still be failing—they're still just stubs!

Actually writing it

Now go back and start filling in the stub methods. At this point you can compile and test fairly frequently. The more frequently you do so, the easier it will be to find bugs that you inadvertently introduce.

Several of the methods will be quite short, and can simply call an existing method of `vector`! Don't write more than you have to.

Another implementation

Once you've finished `VSet`, write a different class called `LazyVSet`. From a user perspective, the results it gives should be exactly the same (but may take more or less time) as a `VSet`. The difference is that when the user requests to add an element, it *always* justs adds it (using `push_back`) to the internal `vector`, even if this creates duplicates—making this a cheap operation, which is why it's "lazy"—but then it has to do a bit more work when it removes something, and when it computes how many distinct elements are in the set.

Testing that one

The tests for `LazyVSet` should be identical to the ones for the other set, right? Copy your existing test file to one called `test_LazyVSet.u` and replace all occurrences of `VSet` (which should only be at the top of the fixture, not in the `setup` or in any of the `test` blocks) with `LazyVSet`, and compile the test suite and run it. Debug your `LazyVSet` and keep testing it until it passes as well. (Size/length is a bit tricky, and you can get full credit on the lab without getting it working *if* it's at least well-tested.)

Handing in

Hand your code in by 4pm Monday, as `lab9` .

RUBRIC

1 Present and engaged in lab with preview stuff done
Set
1 Method headers
½ pure virtual
½ compiles ♣
VSet
1 Class definition as subclass ♣
1 Test suite tests correct behaviour (fail ok) ♣
1 Either add or contains is defined and correct
1 Add, contains, remove, size are correct ♣
LazyVSet
1 Class definition, subclass, add is correct ♣
1 Remove and size are effectively tested (fail ok) ♣
1 Remove is correctly defined

♣ indicates point is only available if the code compiles, with at least a stub for the relevant method(s).

Extra

Produce a table of times and a group of graphs à la Lab 8/Hwk 3 to show the efficiency differences between `VSet` and `LazyVSet`.