Lab: Generating a Cross-Reference Table

Learning Outcomes

- Gain experience in writing adaptors to input streams
- Gain experience in using associative containers
- Gain experience in solving practical problems

Overview

Containers that support efficient look-up

Instead of storing data in a sequential container, we can use an ***associative container***. Such containers automatically arrange their elements into a sequence that depends on the values of the elements themselves, rather than the sequence in which we inserted them. Moreover, associative containers exploit this ordering to let us locate particular elements much more quickly than do the sequential containers, without our having to keep the container ordered by ourselves.

Associative containers offer an efficient way to find an element that contains a particular value, and might contain additional information as well. The part of each container element that we can use for these efficient searches is called a ***key***. For example, if we were keeping track of information about students, we might use the student's name as the key, so that we could find students efficiently by name.

In the sequential containers, the closest that we've seen to a key is the integer index that accompanies every element of a `vector`. However, even these indices are not really keys, because every time we insert or delete a `vector` element, we implicitly change the index of every element *after* the one that we touched.

The most common kind of associative data structure is one that stores key-value pairs, associating a value with each key, and that lets us insert and retrieve elements quickly based on their keys. When we put a particular key-value pair into the data structure, that key will continue to be associated with the same value until we delete the pair. Such a data structure is called an associative array. The most common kind of associative array in C++ is called a `map`, and, analogous with other containers, it is defined in the `<map>` header.

In many ways, `map`s behave like `vector`s. One fundamental difference is that the index of a `map` need not be an integer; it can be a `string`, or any other type with values that we can compare so as to keep them ordered.

Another important difference between associative and sequential containers is that, because associative containers are self-ordering, our own programs must not do anything that changes the order of the elements. For that reason, algorithms that change the contents of containers often don't work for associative containers. In exchange for that restriction, associative containers offer a variety of useful operations that are impossible to implement efficiently for sequential containers.

Counting words

In the previous lab, we used class `punc_stream` to purge punctuation symbols from each line of input. As a simple example, think about how we might use `punc_stream` to count the number of times that each distinct word occurs in our input:

```
int main() {
  std::ifstream ifs{"test1-in.txt"};
  hlp2::punc_stream ps{ifs};
  ps.whitespace(";:,.?!()\"{}<>/&$@#%^*|~"); // note \“ means ”
  ps.case_sensitive(true);

  // store each word and an associated counter ...
  std::map<std::string,size_t> words;
  // purge whitespace from line and split into words ...
  for (std::vector<std::string> line_words; ps >> line_words; ) {
    // write the words and associated counts
    for (std::string const& word : line_words) {
      ++words[word];
    }
  }

  // print word counter like this ...
  using MAP_IT = std::map<std::string const,size_t>::iterator;
  for (MAP_IT it{std::begin(words)}; it != std::end(words); ++it) {
    std::cout << it->first << "\t" << it->second << "\n";
  }

  // or like this ...
  for (std::pair<std::string const,size_t> const& x : words) {
    std::cout << x.first << "\t" << x.second << "\n";
  }
}
```

As with other containers, we must specify the type of the objects that the `map` will hold. Because a `map` holds key-value pairs, we need to mention not only the type of the values, but also the type of the keys. So,

```
std::map<std::string,size_t> words;
```

defines `words` as a `map` that holds values of type `int` that are associated with keys of type `string`. We often speak of such a container as a "a `map` from `string` to `int`," because we can use the `map` by giving it a `string` as a key, and getting back the associated `int` data.

The way we define `words` captures our intent to associate each word that we read with an integer counter that records how many times we have seen that word. We use our `punc_stream` to read a line, purge the line of whitespace, and then split the line into words stored in a `vector<string>`. Then we iterate through the `vector<string>` one word at a time, into `word`. The interesting part is

```
++words[word];
```

What happens here is that we look in `words`, using the word that we just read as the key. The result of `words[word]` is the integer that is associated with the `string` stored in `word`. We then use `++` to increment that integer, which indicates that we've seen the word once more.

What happens when we encounter a word for the first time? In that case, `words` will not yet contain an element with that key. When we index a `map` with a key that has not yet been seen, the `map` automatically creates a new element with that key. That element is value-initialized, which, for simple types such as `int`, is equivalent to setting the value to zero. Thus, when we read a new word for the first time and execute `++words[word]` with that new word, we're guaranteed that the value of `words[word]` will be zero before we increment it. Incrementing `words[word]` will, therefore, correctly indicate that we've seen that word once so far.

Once we've read the entire input, we must write the counters and the associated words. We do so in much the same way as we would write the contents of a `list` or a `vector`: We iterate through the container in a `for` loop, which uses a variable of the iterator type defined by the `map` class. The only real difference is in how we write the data in the body of the `for` statement:

```
std::cout << cit->first << "\t" << cit->second << "\n";
```

Recall that an associative array stores a collection of key-value pairs. Using `[]` to access a `map` element conceals this fact, because we put the key inside the `[]` and get back the associated value. So, for example `words[word]` is an `int`. However, when we iterate over a `map`, we must have a way to get at both the key and the associated value. The `map` container lets us do so by using a companion library type called `pair`.

A `pair` is a simple data structure that holds two elements which are named `first` and `second`. Each element in a `map` is really a `pair`, with a `first` member that contains the key and a `second` member that contains the associated value. When we dereference a `map` iterator, we obtain a value that is of the `pair` type associated with the `map`.

The `pair` class can hold values of various types, so when we create a `pair`, we say what the types of the `first` and `second` data members should be. For a `map` that a key of type `K` and a value of type `V`, the associated `pair` type is `pair<const K, V>`.

Note that the `pair` associated with a `map` has a key type that is `const`. Because the `pair` key is `const`, we're prevented from changing the value of an element's key. If the key were not `const`, we might implicitly change the element's position within the `map`. Accordingly, the key is always `const`, so that if we dereference a `map<string,int>` iterator, we get a `pair<string const,int>`. Thus, `it->first` is the current element's key, and `it->second` is the associated value. Because `it` is an iterator, `*it` is an lvalue, and therefore `it->first` and `it->second` are also lvalues. However, the type of `it->first` includes `const`, which prevent us from changing it.

With this knowledge, we can see that the output statement writes each key [that is, each distinct word from the input], followed by a tab and the corresponding count.

The range-`for` statement provides a simpler and more convenient way of printing the contents of a `map`.

Task: Generating a cross-reference table

A logical next step is to write a program to generate a cross-reference table that indicates where each word occurs in the file. We would like to identify not only the line numbers at which a word occurs but also the position within each of these lines. A text editor could use such as a cross-reference table to implement fast search-and-replace functionality. Since we don't have a text editor to check the correctness of our table, we could instead generate the table for an input file containing the text:

```
Today is a good day Today.
Tomorrow will be a better day;
we can't say much about day- after tomorrow;
but lets hope it works out fine like today

```

and print the table to the standard output stream:

```
File test1-in.txt has 4 lines of text.

"Today" occurs 2 times and is located at:
  line 1, position 1
  line 1, position 6
"Tomorrow" occurs 1 time and is located at:
  line 2, position 1
"a" occurs 2 times and is located at:
  line 1, position 3
  line 2, position 4
"about" occurs 1 time and is located at:
  line 3, position 5
...
```

We'll need to read a line at a time, so that we can associate line numbers with words. Once we read a line, we'll need to first purge the line of whitespace characters and then split the line into words. It would be convenient if we could turn each input line into a `vector<string>`, from which we can extract each word. Nicely enough, class `hlp2::punc_stream` does exactly that for us.

As before, we'll use a `map` with keys that are the distinct words from the input. This time, however, we'll have to associate a more complicated value with each key. Instead of keeping track of how often the word occurs, we want to know all the line numbers on which the word occurred, and for each such line, we want to know all the positions at which the word occurred. A `pair<int,int>` is a convenient data structure to store a line number and the position within the line at which a word occurs. Because any given word may occur multiple times on a single line or on many lines, we'll need to store the line number and position pair in a container.

When we get a new line number, all we'll need to do is append that number to those that we already have for that word. Sequential access to the container elements will suffice, so we can use a `vector<pair<int,int>>` to keep track of multiple occurrences of a word on a certain line number and at a certain position within that line. Therefore, we'll need a map from `string` to `vector<pair<int,int>>`. Let's think about this carefully:

```
using VPII = std::vector<std::pair<int,int>>;
std::map<std::string,VPII> sm; // xref table
std::string word{"Today"};      // a word in input
int line_num{1}, word_pos{1}; // word occurs at line 1 and at position 1
sm[word].push_back(std::make_pair(line_num, word_pos));
```

The statement may be hard to read on first reading,

```
sm[word].push_back(std::make_pair(line_num, word_pos);
```

so we'll pick it apart a bit at a time. We use `word` to index our map. The expression `sm[word]` returns the value stored in the `map` at the position indexed by `word`. That value is a `std::vector<std::pair<int,int>>` or simply `VPII`, which holds the line number and the position within that line at which this word has appeared so far. We call that `vector`'s `push_back` member function to append the current line number and current position within the line to the `vector`.

With these preliminaries out of the way, it's quite straightforward to implement a function `hlp2::xref` in file xref.cpp with one additional feature. Here I'll assume you've reviewed the lecture presentation and sample source file on `std::set`. We'd like the table to cross-reference only interesting words. So, we provide a parameter of type `set<string>` that contains words to be excluded:

```
namespace hlp2 {
// find all the line and word position pairs that refer
// to each word in the input file
// return a map and an int as a pair ...
std::pair<std::map<std::string, std::vector<std::pair<int,int>>>,int>
  xref(std::string const& filename,
           std::string const& whitespace,
           bool case_sensitivity,
           std::set<std::string> const& exclude) {
  // define a map from string to vector<pair<int,int>>
  std::pair<std::map<std::string, std::vector<std::pair<int,int>>>,int> ret;

  // open file stream
  // use hlp2::punc_stream to purge whitespace from each line of input
  // and split the line into words
    // remember current line in file
    // go through each word in vector returned by punc_stream
      // remember current position in line
      // if word is not in exclude
        // use word to index map and use vector's push_back member
        // to append pair representing current line number and
```

```
        // current position in line to the vector

  // line_num represents number of lines in file ...
  return std::make_pair(ret, line_num);
} // end function xref
} // end namespace
```

The driver would make a call to `xref` like this:

```
std::string filename{"test1-in.txt"}; // input file
std::string whitespace{"-;:,.?!()\"{}<>/&$@#%^*|~"}; // punctuation to be purged
bool word_case{true}; // case sensitive
std::set<std::string> exclude = exclude_words(); // exclusion set ...

using Key = std::string;
using Pair = std::pair<int,int>;
using Value = std::vector<Pair>;
std::pair<std::map<Key,Value>, int> words_pair
  = hlp2::xref(filename, whitespace, word_case, exclude);
std::map<Key,Value> const& words = words_pair.first;
int line_cnt = words_pair.second;
std::cout << "File " << filename << " has " << line_cnt << " lines of
text.\n\n";
```

You must add a function `print_wordmap` to print the words and associated pairs of line numbers and word positions:

```
void print_wordmap(
  std::map<std::string, std::vector<std::pair<int,int>>> const& words) {
  // fill the details here ...
}
```

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Submission files

You will be submitting files xref.hpp, and xref.cpp.

Compiling, executing, and testing

Compile your source file(s) with the full suite of g++ flags using a driver xref-driver.cpp.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of `g++` options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output must exactly match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). There are only two grades possible: $A+$ grade if your output matches correct output of auto grader; otherwise F.
    - A maximum of D grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.
    - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.