Lab: Defining Semi-Realistic Input Stream

Learning Outcomes

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

Overview

Think about how we might count the number of times each distinctive word occurs in a text file. In class lectures, this problem was trivially solved using associative arrays:

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <map>

int main() {
  std::ifstream ifs {"haiku.txt"};
  std::map<std::string,size_t> histogram; // word and its associated count
  std::string s;
  // read input file stream, keeping track of each word and how often it occurs
  while (ifs >> s) {
    ++histogram[s];
  }

  std::cout << "Total words in histogram: " << histogram.size() << "\n";
  // write each word and its associated count to standard output
  for (std::pair<const std::string,size_t> const& p : histogram) {
    std::cout << "<" << p.first << ">: " << p.second << "\n";
  }
}
```

If file haiku.txt contains the text:

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

```

the output consists of 27 distinct words [note that individual words are delimited by `<` and `>` symbols to ensure there are no whitespace characters in the word]:

```
<a>: 2
...
<day->: 1
<day.>: 1
<day;>: 1
...
<today>: 1
<today.>: 1
<tomorrow>: 1
<tomorrow:>: 1
...
<works>: 1

```

Our trivial program has a glaring problem that must be rectified. And, adding two new features would enhance the program's usefulness.

1. As illustrated by the output, our trivial program doesn't know anything about punctuation. The program identifies day-, day., and day; as distinct words. Ditto with tomorrow and tomorrow; . When we read strings, words are by default separated by whitespace. Unfortunately, `istream` doesn’t offer a facility for us to define what characters make up whitespace or in some other way directly change how `>>` reads a string. So, if we read the following line from a file

   ```
   As planned, the guests arrived; then,

   ```

   we would get the "words"

   ```
   <As>: 1
   <planned,>: 1
   <the>: 1
   <guests>: 1
   <arrived;>: 1
   <then,>: 1

   ```

   This is not what we’d find in a dictionary: planned, and arrived; are not words. They are words plus distracting and irrelevant punctuation characters. For most purposes we must treat punctuation just like whitespace. How might we get rid of such punctuation?
2. For files containing large amounts of data, the output will be overwhelmed by commonplace and uninteresting words such as a, be, is, our, we, your, and so on. How might we prevent such words from being analyzed by the word counter?
3. Once we know how to count how often words occur in a text file, a logical next step is to write a program to generate a cross-reference table that indicates where each word occurs in the file. How can we generate such a cross-reference table?

Task 1: Dealing with Punctuation Symbols

Looking ahead to Task 3, instead of reading a word at a time, we'll need to read a line at a time, so that we can associate line numbers with words. We could then read characters, remove the punctuation characters - or turn them into whitespace - and then read the "cleaned-up" input again for words:

```cpp
std::ifstream ifs {"haiku.txt"};
std::map<std::string,size_t> histogram; // store each word and associate count
std::string line;
while (getline(ifs, line)) { // read each line
  for (char& ch : line) { // replace each punctuation character by a space
    switch(ch) {
      case ';': case '.': case ',':
      case '?': case '!': case '-': ch = ' ';
    }
  }

  // read each distinct word from line and add to histogram
  std::istringstream iss{line};
  std::string word;
  while (iss >> word) {
    ++histogram[word];
  }
}
```

Using that to read the line

```
As planned, the guests arrived; then,

```

we get the desired output by printing the elements of `histogram`:

```
<As>: 1
<arrived>: 1
<guests>: 1
<planned>: 1
<the>: 1
<then>: 1

```

Unfortunately, the code above is messy and rather special-purpose. What would we do if we had another definition of punctuation? Let’s provide a more general and useful way of removing unwanted characters from an input stream. What would that be? What would we like our user code to look like? How about:

```
std::ifstream ifs{"haiku.txt"};
hlp2::punc_stream ps{ifs}; // source of characters
ps.whitespace(";:,."); // treat semicolon, colon, comma, and dot as whitespace
ps.case_sensitive(true); // retain case-sensitivity in stream's text
std::map<std::string,size_t> histogram; // store each word and associate count

// ps would:
//  a) read each line from input stream
//  b) replace punctuation characters in that string
//  c) split line into words contained in std::vector<std::string>
for (std::vector<std::string> line_words; ps >> line_words; ) {
  // insert words from line_words into histogram ...
}
```

How would we define a stream that would work like `ps`? The basic idea is to read words from an ordinary input stream and then treat the user-specified "whitespace" characters as whitespace; that is, we do not give "whitespace" characters to the user, we just use them to separate words. For example,

```
as.not
```

should be the two words

```
<as>: 1
<not>: 1

```

We can define a class to do that for us. It must get characters from an `istream` and have a `>>` operator that works just like `istream`'s except that we can tell it which characters it should consider to be whitespace. For simplicity, we will not provide a way of treating existing whitespace characters [space, newline, etc.] as non-whitespace; we'll just allow a user to specify additional "whitespace" characters. Nor will we provide a way to completely remove the designated characters from the stream; as before, we will just turn them into whitespace. Let's call that class `hlp2::punc_stream`:

```
namespace hlp2 {
class punc_stream {
public:
  punc_stream(std::istream& is);         // attach to input stream
  void whitespace(std::string const& s); // make s the whitespace set
  void add_whitespace(char ch);      // add to whitespace set
  bool is_whitespace(char ch) const; // is ch in whitespace set?
  void case_sensitive(bool b);    // set stream's case-sensitivity
  bool is_case_sensitive() const; // return case sensitivity

  punc_stream& operator>>(std::vector<std::string>& s); // see below
  operator bool() const; // see below
private:
  std::istream& source; // source of characters
  std::string white;    // characters considered whitespace
  bool sensitive;       // is the stream case-sensitive?
};
} // end namespace hlp2
```

The basic idea is - just as in the example above - to read a line at a time from the `istream`, convert "whitespace" characters into spaces, and then split the converted line into words which are then inserted into a `vector<string>` container. In addition to dealing with user-defined whitespace, we have given `punc_stream` a related facility: if we ask it to, using `case_sensitive`, it can convert case-sensitive input into non-case-sensitive input. For example, by making `ps` case-insensitive:

```
ps.case_sensitive(false); // not case-sensitive
```

we can get a `punc_stream` to read

```
Man bites dog!

```

as

```
<bites>: 1
<dog>: 1
<man>: 1

```

`punc_stream`'s constructor takes the `istream` to be used as a character source and gives it the local name `source`. The constructor also defaults the stream to the usual case-sensitive behavior. We can make a `punc_stream` that reads from the input file stream `ifs` regard semicolon, colon, and dot as whitespace, and that turns all characters into lower case:

```
std::ifstream ifs{"some-text-file"}; // open an input file stream
hlp2::punc_stream ps{ifs}; // ps reads from ifs
ps.whitespace("";:."");     // semicolon, colon, dot are also whitespace
ps.case_sensitive(false);   // not case-sensitive
```

Obviously, the most interesting operation is the input operator `>>`. It is also by far the most difficult to define. Our general strategy is to read one line at a time from the `istream` into a `string`, say `line`. We then convert all of "our" whitespace characters in `line` to the space character. That done, we parse individual `string`s in `line` and store them a `vector<string>`. After all lines in the input [file] stream have been read, we return the words in these lines in the `vector<string>`. Each element of the `vector<string>` is then be inserted into a `map<std::string,size_t>` container:

```
std::map<std::string,size_t> histogram;
for (std::vector<std::string> line_words; ps >> line_words; ) {
  for (std::string const& word : line_words) {
    ++histogram[word];
  }
}
```

This leaves one mysterious function:

```
punc_stream::operator bool() const;
```

The conventional use of an `istream` is to test the result of `>>`. For example:

```
while (ps>>s) { /* . . . */ }
```

That means that we need a way of looking at the result of `ps>>s` as a Boolean value. The result of `ps>>s` is a `punc_stream`, so we need a way of implicitly turning a `punc_stream` into a `bool`. That's what `punc_stream`'s `operator bool()` does. A member function called `operator bool()` defines an implicit conversion to `bool`. In particular, it returns `true` if the operation on the `punc_stream` succeeded.

Now, we can write our program:

```
// given a text file, produce a frequency counter of distinct words
// in that file by ignoring punctuation and case differences
int main(int argc, char *argv[]) {
  std::ifstream ifs{argv[1]};
  hlp2::punc_stream ps{ifs};
  ps.whitespace(";:,.?!()\"{}<>/&$@#%^*|~"); // note \" means "
  ps.case_sensitive(false); // case-insensitive
  std::map<std::string,size_t> histogram;
  // insert words in file into histogram ...
  for (std::vector<std::string> words; ps>>words; ) {
    for (std::string const& word : words) { ++histogram[word]; }
  }
  // print frequency counter as shown in previous examples
}
```

Define class `punc_stream` in file puncstream.hpp and then define the member functions of the class in source file puncstream.cpp. In the header file, declare a function `print` to print the contents of `histogram` to standard output:

```
void print(???, std::string const& pre="", std::string const& post="\n");
```

Define function `hlp2::print` in the source file. In addition to printing the `pre` and `post` strings, the function must print the contents of `histogram` like this:

```
// write each word and its associated count to standard output
for (std::pair<const std::string,size_t> const& p : histogram) {
  std::cout << "<" << p.first << ">: " << p.second << "\n";
}
```

If `hlp2::print` requires other function(s), make sure to provide those definitions in puncstream.cpp. The output files of the autograder are generated by this call: `hlp2::print(words, "", "");`

Feed a text file containing this input to the program

```
There are only two kinds of languages: languages that people complain
about, and languages that people don't use.
and it

```

and it should generate this ***exact*** output [using your function `print`]:

```
<about>: 1
<and>: 2
<are>: 1
<complain>: 1
<don't>: 1
<it>: 1
<kinds>: 1
<languages>: 3
<of>: 1
<only>: 1
<people>: 2
<that>: 2
<there>: 1
<two>: 1
<use>: 1

```

Why did we get `don't` and not `dont`? We left the single quote out of the `whitespace()` call.

Caution: `hlp2::punc_stream` behaves like an `istream` in many important and useful ways, but it isn't really an `istream`. For example, we can't ask for its state using `rdstate()`, `eof()` isn't defined, and we didn't bother providing a `>>` that reads integers. Importantly, we cannot pass a `punc_stream` to a function expecting an `istream`. Could we define a `punc_istream` that really is an `istream`? We could, but we don't yet have the programming experience, the design concepts, and the language facilities required to pull off that stunt [if you - much later - want to return to this problem, you've to look up stream buffers in an expert-level guide or manual].

Tasks 2 & 3:

We'll reserve this for next week's lab. You will receive an advance copy of the specs for these tasks in the next few days.

Submission Details

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

Submission files

You will be submitting file `puncstream.hpp` and `puncstream.cpp`.

Compiling, executing, and testing

Compile your source file(s) with the full suite of $g++$ flags. To celebrate the [near] conclusion of this semester, there is no driver. Instead, begin testing your implementation of `hlp2::punc_stream` using the examples in this document. Test your output by turning on and off case-sensitivity. Are there any edge cases you should worry about? What about an empty input file?

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.