COMPUTER GENERATED POETRY USING THE C++ STANDARD TEMPLATE LIBRARY

Linda M. Wilkens
Math and Computer Science Department
Bridgewater State College
Bridgewater, MA 02325
(508) 697-1342
lwilkens@bridgew.edu

ABSTRACT

This paper describes using the C++ Standard Template Library (STL) to create a program that automatically generates naïve poems such as the following:

Next morning: the day
and a portrait are cousins.
Madam you, think he.

Although the poetry is nonsensical, the application is novel to junior or senior level students at small colleges. Furthermore, it affords an opportunity to develop a nontrivial C++ program for a problem that requires some domain analysis. Using predefined classes from the Standard Template Library allows the software process to concentrate on the problem domain, rather than the implementation detail. Computer-generated poetry provides the vehicle by which we attain our goal: to demonstrate an interesting application that benefits greatly from using object-oriented methods, in particular the code reuse that is made possible by the C++ Standard Template Library.

[KEYWORDS: Computer Science, Object-Oriented, C++, Standard Template Library, Computer Generated Literature]

INTRODUCTION

This paper describes a version of a haiku-generator based on C++ STL container classes. The basic data structure provided by the STL class is a directed

graph; the program first generates the graph and then takes a random walk through it to generate the haiku. This approach is based on a well-known technique/program borrowed from the Artificial Intelligence community, called "Henley". Output from Henley forms the preface to a popular Common Lisp text [5], and sounds eerily like some early English poetry that bedevils college freshmen in their literature classes:

Half lost on my firmness gains to more glad heart,
Or violent and from forage drives
A glimmering of all sun new begun
...

Haikus, and computer-generated poetry in general, occur in the domain of Computer Generated Literature (CGL). This paper begins with a brief review of Computer Generated Literature. We then present a Henley-style poetry-generator, and show how to implement it using predefined STL classes. We then discuss the constraints imposed by the haiku form of poetry, and how to respect these constraints in our poetry-generator. We conclude by considering strengths and weaknesses of the approach, and listing some of the poetry generated by our program.

Computer Generated Literature has been around since the early days of computers. As early as 1960, creative writing by computer had produced works such as small Western-style plays, illustrating the role of computer-as-playwright [1]. Currently, a cursory Web search for CGL using the phrase "computer generated literature" on a search engine turns up tens of matches, linking to sites as varied as one that explores the controversy surrounding Computer-Generated Literature (CGL) [2], a Cyberpunk reading list [3], and a site with novels and stories featuring machine intelligence characters [4].

Computer Generated Literature is easier to analyze than computer-generated visual or auditory art because literature deals with words, how to put words together, and how to derive meaning and beauty from the patterns of word compositions. Hence, it is an area in which both students and instructors already possess much domain knowledge. Investigating CGL is a nice starting point for research into non-traditional computer applications, and working with literature presents an opportunity for students whose expertise is language, rather than math, to excel. Furthermore, CGL provides a good opportunity to explore data structures and algorithms, and to utilize object-oriented technologies such as the C++ Standard Template Library.

Working with words is not unlike dealing with any sort of data: the program needs structures to hold the data, ways to traverse these structures, and algorithms to transform the input of the program into the desired output. However, we don't want to get bogged down in implementation detail; rather we want to concentrate on the particulars of a novel domain. Thus, we want to use ready-made data abstractions and pre-coded algorithms as much as possible. This is where the standard libraries such as the Standard Template Library for ANSI C++ come into play. We can store our data in container classes provided by the STL, and then access the data using the

corresponding iterators. The code re-use possibilities afforded by STL and other vendor-supplied classes relieve relatively inexperienced programmers from some of the tedium of low-level structure manipulations, allowing them to concentrate on domain-specific details.

THE BASIC HENLEY-STYLE GENERATOR

As Graham notes in his discussion of Henley [6], ordinarily people want to make sense out of a sequence of words strung together. That is, if the words hang together in the least way, the listener or the reader will fill in sometimes huge gaps to suppose meaning where there actually is none. Taking advantage of this natural psychological tendency, it is easy to write a fairly simple program that generates a very free-form pseudo poetry. The source of the Lisp version given in Grahams' text [7] consumes fewer than two pages. This is a lot of "bang-for-the-buck" if you are a line counter. In general, Lisp code tends to be much denser than code written in C or C++, mainly because it utilizes sophisticated data abstractions and convenient macros. The evolution of C++ and the inclusion in many C++ implementations of class and template libraries now makes it possible to work in C++ at a higher level of abstraction than was previously possible, and to adapt some of the coding paradigms of the AI community to object-oriented imperative solutions.

Lisp Henley has two phases: the first phase reads the input text and stores distinct words in a hash table. The hash table's keys are the words, and the values in the table are lists of words that follow the given word in the text, paired with a frequency. For example, the hash table entry for *bird* might be ((dog, 1) (house, 3)), meaning that the word *bird* occurred 4 times in the input text, and once it was followed by *dog*, and 3 times it was followed by *house*. Because Common Lisp uses manifest typing, the programmer doesn't have to worry about what type of element is used for the key and what type of element is used for the value in the hash table. The convenience of the hash table accessor functions supports storing whatever the application calls for. The second phase of Henley takes a random walk away from the current word, based on the relative frequencies of the following words. Thus, if we were at *bird*, we would move to *dog* with probability 1/4 and to *house* with probability 3/4. The simplicity of the random walk stands out, rather than data structure manipulation, since the manipulation is provided by the language itself.

Comparing Lisp and C++ is difficult; they represent two distinctly different language paradigms. However, both functional languages and object-oriented languages support programming at a level of abstraction that matches the problem by providing a variety of language and library features. Features that are useful for a C++ implementation of Henley are the *string* class which allows treating strings in a manner similar to primitive data types, and the sorted associative container classes along with their accompanying iterators.

Because the STL provides many template classes and many generic algorithms, choices for using these features abound. The STL provides 4 different sorted associative containers (sets, multisets, maps, and multimaps), all with O(log N) worst-case performance bounds for retrieval based on keys with some total order [8]. We chose to use maps for our basic containers, allowing that other choices are certainly viable. All of the sorted associative containers support the *find* member function, which searches for a given key and returns an iterator referring to the entry with that key. Our first try used vectors, but the linear insertion time quickly proved too severe a limitation on the input text file size. Maps are nice because, although they are in general implemented with balanced binary trees, they look to the programmer like arrays indexed using their keys. That is, given a string variable called *next_word* and a map called *all_words* keyed on the distinct words in the text, we can insert into *all_words* the value to be associated with *next_word* by using the indexing operator:

```
all_words[next_word] = some_value;
```

Because maps are based on ordered keys, an instantiation of the map template requires that we specify three template parameters:

1. the type of the keys,
2. the type of the values, and
3. the comparison function to used for the ordering.

Using the *string* class for the key makes it easy to instantiate the map template, since the *string* class defines operators for comparisons. Thus our type of map is defined by the following few lines of code, where *net_node* stores any information we need about a word, including the list of other words that follow it in the input text.

```
#include <cstring.h>      // For the C++ string class
class net_node;
typedef map<string, net_node*, less<string>> node_storage;
```

We found maps so easy to work with, that we stored the lists of word-followers as maps also, although this does end up using up quite a bit of space.

```
class net_link;
typedef map<string, net_link*, less<string>> link_storage;
```

Our main class has a member *all_words* of type *node_storage*, and each node has a *link_storage* member. Building our equivalent of the Lisp version's hash table is done by parsing though the input text to extract the individual words, and inserting them into our main *node_storage* map, using the map member function *find*.

```
// If the string is not in the map, insert it.
if ((wsi = all_words.find(buffer)) == all_words.end()){
          next = new net_node;
          assert(next != NULL);
          all_words[buffer] = next;
}
```

```
// Otherwise, retrieve it
else {
     next = (*wsi).second;
}

// Adjust links between it and its predecessor
if (prev) {
     prev->add_forward_link(buffer,next);
     next->add_backward_link(prev_buffer,prev);
}
 // Update what the previous word is
prev = next;
strcpy(prev_buffer,buffer);
```

Notice that variable *wsi*, which is returned by the map's *find* member function is of type *node_storage::iterator*, which acts like a pointer to a pair representing the key,value association stored in the map. Thus, the key may be retrieved using the syntax *(*wsi).first*, and the value may be retrieved using *(*wsi).second*.

Although the *string* class provides almost every operator and member function imaginable, you may want access to the underlying array of characters. Because this array is not meant to be modified, it is *private* but accessible via a member function *c_str()*. Since *c_str()* returns a *const char* *, the declarations and usage need to respect that *const*-ness.

LESSONS LEARNED USING STL

Code reuse is a mixed blessing, in that the learning curve can be quite steep for classes and templates with a lot of functionality. As with most learning endeavors, it is best to proceed with caution. To cite an old Buddhist saying: " We have little time, therefore it is important to proceed very slowly." Finding a good reference and a solid implementation is perhaps the most important first step. We suggest [8]. All of the code presented here was compiled with Borland C/C++ 5.0, and uses only the classes libraries that came with it. Our source code is available via anonymous ftp from *benny.bridgew.edu* in directory *pub/ESCCC98*. The STL itself is downloadable from a number of ftp sites, as is a separate *string* class include file, bstring.h [9].

As with most non-trivial programs, hindsight illuminates the need for careful planning and clean design, the advantages of simplicity, and the absolute necessity to pay attention to what the compiler tells you. Development of this code experienced problems either caused by or indicated by one or more of the following:

1. Circularity in class definitions, which may cause awkwardness that is difficult to work around. Overuse of the friend construct means it's time for a re-design.

2. Suspicious pointer conversion compiler warnings. Code that casts a pointer is definitely a red-flag item.

3. Complex for-header constructs. For-loops that do very much in the header and very little in the body need to be inspected carefully.

WRITING STRUCTURED POETRY

Of the various types of literature, computer-generated poetry is popular because it comes in many different forms and because poetic license allows bending the rules. For example, iambic pentameter is a form in which each line has five feet, and each foot roughly forms an iamb, i.e., an unaccented syllable followed by an accented one. However, the best poems don't follow the rules too closely; if they did, they would have the sing-song of a child's verse. In addition to rhythm, poetry often incorporates rhyming. For example, in "Little boy blue, come blow your horn. The sheep's in the meadow, the cow's in the corn", horn and corn rhyme. If we compare this structured child's verse, with both rhythm and rhyme, to the rambling free form of Henley, we see that they are very different in both form and style.

We chose to explore the haiku form of poetry, because it is a middle ground between very structured and completely free-form. The basic definition of a haiku is that it is a composed of three lines: the first and third have five syllables each; the middle has seven syllables. A haiku can be written on any topic in any language [10], but should try to follow some guidelines:

1. Attain the five/seven/five syllable ideal.
2. Keep each line self-contained; i.e. no sentence should spill to the next line.
3. Use words to reflect a theme, such seasonal words about fall or spring.
4. Use striking imagery.
5. Exhibit humor or clever wordplay.

Of these guidelines, the first two concern the structure of the poem, the third is arbitrary, and the last two are used to discriminate between good poetry and bad. A dictionary definition of poetry [11] stresses that a poem is a "characterized by intensity and beauty of language or thought." We reasoned that a poetry-generator could be programmed to obey at least the first rule, and perhaps the second as an option. We ignored the third rule, and hoped that randomness would produce results that satisfied the remaining artistic requirements. As it turned out, the artistry was harder to achieve than the required form.

COUNTING SYLLABLES

In order to satisfy the 5-7-5 rule, we need to know the number of syllables in a word. There are two choices: either look the word up in an electronic dictionary, or write a function to syllabify the words. We decided to derive a set of rules for counting the syllables in a word and to write a function to apply these rules. Unfortunately, the irregularity of the English language makes a rule-based approach difficult because there

are many exceptions to each rule. It may be reasonable to investigate the use of an electronic dictionary, at least for some difficult words.

On the bright side, the task of breaking a word into syllables provides an opportunity to use state-transition systems. We derived a basic syllable-counting algorithm based on a small state-transition system with only four states and a small amount of domain knowledge incorporated in the actions that are initiated when transitioning from one state to another. The states are

S: Start
C: processing a sequence of Consonants
V: processing a sequence of Vowels
E: End

An (i,j) entry in *Table 1* indicates the action to take when transitioning from state i = S, C, or V to state j = C, V, or E on a given character c. Transitioning between consonants and vowels, and vice versa is complicated slightly by *y*'s double nature, being a consonant in some cases and a vowel in others. When the current state is C and we see a vowel, we add 1 to the syllable count and change the state to V. If the current state is V and we see a consonant, we need to consider the pattern of contiguous vowels that we have seen. For example, the "oo" in *look* will not increment the syllable count, whereas the "eyi" in *journeying* will. It is hard to establish a set of rules for transitions and vowel combinations that will hold in all cases, and we have to rely on the forgiveness inherent in poetic license. Some words just don't fall into normal categories; for example, *Wednesday*, appears to have 3 syllables, but is almost always pronounced with two. Another difficulty derives from the original authors' use of non-standard words to indicate a regional accent, a fictional character's nonconformity, or other fancifulness. It is not uncommon to come across words such as beyoutiful, loook, or ataeus, pointing to the need for reasonable default actions for unexpected vowel sequences.

Finishing a word requires checking for a few special cases such as plural nouns and tenses on verbs. If a word ends in "es" or "ed", the standard actions may result in a syllable count that is too large. For example, *paved* has one syllable, whereas *pointed* has two. Similarly, for *fates* and *faces*. Again, there are other special rules that could be checked, but those listed are the most common.

	C	V	E
S	none	increment count	Count is 0
C	none	increment count	Finish
V	modify count	save character	Finish

Table 1: State Transition Action Table

PUTTING THE WORDS TOGETHER

Whereas a truly random walk would start at a random location, and travel randomly through the graph for a random number of steps, imposing the rules of the haiku means constraining some of that randomness. For example, the poem should start at a word that actually started some sentence in the input text. Similarly, it should end at a meaningful place. We need to do a little work while scanning the text, to find this information and to store it with the individual words. After the input text has been read and the graph has been created, we can apply the haiku-creation algorithm which is a randomized variation of a depth-first search through the space of random sentences:

Step 1: Do until first_line_ok is true:

- Start at a random starter word.
- Follow this starter word until the number of syllables accumulated, count_1, is > = 5.
- If count_1 is equal to 5, set first_line_ok to true.
- If we've tried too many times and failed, give up.

Step 2: Do until second_line_ok is true:

- Follow the last word in line 1 until the number of syllables accumulated for the second line, count_2, is > = 7.
- If count_2 is equal to 7, set second_line_ok to true.
- If we've tried too many times, go back and restart at step 1.

Step 3: Do until third_line_ok is true:

- Follow the last word in line 2 until the number of syllables accumulated for the third line, count _3, is > = 5.
- If count _3 = 5, and the last word is an ender, set third_line_ok to true.
- If we've tried too many times, go back and restart at step 2.

Stop.

This algorithm will succeed eventually, as long as there is enough variation in the input text, and enough connectivity in the graph. If the input text is too small, the output will have phrases repeated exactly from the input, yielding plagiarism rather than poetry. If the input text is too large, too much space will be consumed by the graph. Experience indicates that input texts in the range of 50 to several hundred K bytes yield interesting results.

Each poem created is based on some input text. The quality of this input text, its character, its tenor, and how varied it is, greatly impact the output of the program. The saying "garbage-in, garbage-out" takes on a whole new meaning in this context.

Fortunately, thanks to ambitious efforts such as Project Gutenberg [12], a large number of classical books are available in electronic form. Most of these are available either uncompressed or compressed in UNIX *g-zip* or DOS *zip* compression format, and provide wonderful input to this sort of program.

SOME POEMS

The final output of the haiku-generator is sometimes fanciful, sometimes sheer nonsense. We ran a short experiment, generating 30 haiku for each of 4 different input texts. About half sounded somewhat poetical, while the other half were more akin to the following, which falls into the jabberwocky category although it does satisfy the 5-7-5 rule.

Delicate, and drop
the world to the soft soap, and
thirty pounds and then.

Here are some of the best of each batch of haiku, with capitalization and punctuation added by hand. Notice that our algorithm fails to keep each line self-contained, the poems run over the lines and in some cases are really one long sentence.

The first is generated from the text of *Tanglewood Tales*, by Nathaniel Hawthorne (367Kb).

Nor had known such a
home to be sensible that
had several stone.

The next is based on *The Rose and the Ring*, by William Makepeace Thackeray (183Kb).

Revenge is a rhyme,
for whose means this is, this didn't.
Care to breakfast in?

It seemed from reading the original text that *Lorna Doone, A Romance of Exmoor* by R. D. Blackmore would be an excellent choice for a basis, but the results were somewhat disappointing. Because of the size of the complete text (1471Kb), only a portion of it was used.

According to make
them turned, and thighs, until they
have mercy, you mean.

Finally, a haiku based on *American Notes* by Rudyard Kipling (152Kb). Som might think it appalling, but we found it whimsical, and not that different from poetr found in modern collections.

Pity the power
of dollars, without being the
swift sentence, with all.

REFERENCES

1. Knuth, Donald. *The Art of Computer Programming Volume 2: Seminumerica Algorithms, 2nd Edition*, Addison-Wesley, Reading, MA, 1981, pp. 174-176.
2. http://www.ypn.com/topics/8151.html
3. http://www.cs.ubc.ca/spider/harrison/Cyberpunk/ cyberpunk.html
4. http://sflovers.rutgers.edu/archive/bibliographies/ machine-intelligence.txt
5. Graham, Paul. *ANSI Common Lisp*, Prentice Hall, Englewood Cliffs, NJ, 1996, pg. vi.
6. ibid, pg. 407.
7. ibid, pp 140-141.
8. Musser, David R., and Saini, Atul. *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.
9. ibid, pp. 383-385.
10. http://www.webcom.com/~erique/haiku/haiku.html
11. *Funk and Wagnalls Standard College Dictionary*, 1966, pg. 1042.
12. ftp://sunsite.unc.edu/pub/docs/books