Using Associative Containers

Lecture 18
Hartmut Kaiser
hkaiser@cct.lsu.edu
http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
**Programming Principle of the Day**

- Single Responsibility Principle - A component of code (e.g. class or function) should perform a single well defined task.

- Components should have exactly one *reason to change*. As an example, consider a module that compiles and prints a report.
  - Can be changed for two reasons:
    - The content of the report can change.
    - The format of the report can change.
  - These two things change for very different causes; one substantive, and one cosmetic.

- The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate components. It would be a bad design to couple two things that change for different reasons at different times.

Abstract

• Associative containers arrange their elements in a certain sequence based on an ordering criteria for the elements themselves

• They employ ordering to quicker find elements

• Often they store key/value pairs, ordering the values based on the key

• We will investigate these containers and use maps to write compact and efficient look-up-intensive programs.
Why Associative Containers?

• Finding an element in sequential containers entails sequential search
  • Potentially slow if container has many elements

• Alternative is to keep elements in sequential container in certain order
  • Devise a special search strategy, not easy
  • Potentially slow to insert as it might reorder elements

• Another alternative is to use associative containers
Associative Containers

• Associative containers automatically arrange their elements into a sequence that depends on the values of the elements themselves, rather than the sequence in which they were inserted.

• Allows to locate element with particular value quickly.

• The part which is used to locate an element is the key, which sometimes has an associated value.
Counting Words

• Almost trivial with associative containers:

```cpp
int main()
{
    string s;
    map<string, int> counters; // store each word and an
    // associated counter
    // read the input, keeping track of each word and
    // how often we see it
    while (cin >> s)
        ++counters[s];

    // write the words and associated counts
    for (map<string, int>::const_iterator it = counters.begin();
         it != counters.end(); ++it)
    {
        cout << it->first << "\t" << it->second << endl;
    }
    return 0;
}
```
Counting Words

• As `std::map` holds key/value pairs, we need to specify both: `std::map<string, int>`
  • Holds values of type `int` (the word counters) with a key of type `string` (the counted words)
  • We call this a ‘map from string to int’

• ‘Associative array’, we use a string as the index (the *key*)
  • Very much like vectors, except that index can be any type, not just integers
Counting Words

• Necessary: #include <map>

    while (cin >> s) ++counters[s];

• Indexing operator[]: invoked with string ‘s’

• Returns integer value associated with string ‘s’
  • We increment this integer: counting words
  • If no entry representing string ‘s’ exists, new entry is created and value initialized (integer is set to zero)

    cout << it->first << "\t" << it->second << endl;

• Iterator ‘it’ refers to both, key and value
  • std::pair: pair of arbitrary types, stored in map
  • The parts are named: first, second
Generating a Cross-Reference Table

• Write a program to generate a cross-reference table that indicates where each word occurs in the input
  • Read a line at a time, allowing to associate line numbers with words
  • Split line into words
  • Store more data in map: all lines a particular word occurred
    • map<string, vector<int>>
Generating a Cross-Reference Table

• Find all the lines that refer to each word in the input:

```cpp
map<string, vector<int>>
xref(istream& in, vector<string> find_words(string const&) = split)
{
    string line; // current line
    int line_number = 0; // current line number
    map<string, vector<int>> ret; // cross reference table

    // read the next line
    while (getline(in, line)) {
        // store current line number for each word
        // ...
    }
    return ret;
}
```
Generating a Cross-Reference Table

• Store current line number for each word:

```cpp
while (getline(in, line)) {
    // adjust current line number
    ++line_number;

    // break the input line into words
    vector<string> words = find_words(line);

    // remember that each word occurs on the current line
    for (auto const& s: words)
    {
        ret[s].push_back(line_number);
    }
}
```
Generating a Cross-Reference Table

• Default argument specification:

```cpp
map<string, vector<int>>
xref(istream& in,
    vector<string> find_words(const string&) = split);
```

• Allows to leave out this argument at invocation:

```cpp
// uses split() to find words in the input stream
... = xref(cin);
// uses the function named find_urls to find words
... = xref(cin, find_urls);
```
Generating a Cross-Reference Table

• What is this doing here:

    ret[s].push_back(line_number);

  • s: the current word
  • ret[s]: returns the value associated with the key (s) yielding the vector of line numbers
    • If this is the first occurrence, an empty vector is put into the map
  • ret[s].push_back(): adds the current line number to the end of the vector
Printing the Cross-Reference

• Print the generated map:

```cpp
int main()
{
    // call xref using split by default
    map<string, vector<int>> xrefmap = xref(cin);
    // write the results
    for (auto it = xrefmap.begin(); it != xrefmap.end(); ++it)
    {
        // write the word followed by one or more line numbers
        cout << it->first << " occurs on line(s): ";

        auto line_it = it->second.begin();
        cout << *line_it++;
            // write the first line number

        // write the rest of the line numbers, if any
        for_each(line_it, it->second.end(), [] (int line) { cout << ", " << line; });

        // write a new line to separate each word from the next
        cout << endl;
    }
    return 0;
}
```
### Generating Sentences

<table>
<thead>
<tr>
<th>Categories</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;noun&gt;</td>
<td>cat</td>
</tr>
<tr>
<td>&lt;noun&gt;</td>
<td>dog</td>
</tr>
<tr>
<td>&lt;noun&gt;</td>
<td>table</td>
</tr>
<tr>
<td>&lt;noun-phrase&gt;</td>
<td>&lt;noun&gt;</td>
</tr>
<tr>
<td>&lt;noun-phrase&gt;</td>
<td>&lt;adjective&gt; &lt;noun-phrase&gt;</td>
</tr>
<tr>
<td>&lt;adjective&gt;</td>
<td>large</td>
</tr>
<tr>
<td>&lt;adjective&gt;</td>
<td>brown</td>
</tr>
<tr>
<td>&lt;adjective&gt;</td>
<td>absurd</td>
</tr>
<tr>
<td>&lt;verb&gt;</td>
<td>jumps</td>
</tr>
<tr>
<td>&lt;verb&gt;</td>
<td>sits</td>
</tr>
<tr>
<td>&lt;location&gt;</td>
<td>on the stairs</td>
</tr>
<tr>
<td>&lt;location&gt;</td>
<td>under the sky</td>
</tr>
<tr>
<td>&lt;location&gt;</td>
<td>wherever it wants</td>
</tr>
<tr>
<td>&lt;sentence&gt;</td>
<td>the &lt;noun-phrase&gt; &lt;verb&gt; &lt;location&gt;</td>
</tr>
</tbody>
</table>

- Example: the table jumps wherever it wants
Representing the Rules

• Categories, rules and ‘normal’ words
  • Categories: enclosed in angle brackets
  • Right hand side is a rule consisting out of a sequence of categories and words

• How to represent/store categories?
  • Let’s use a std::map to associate the categories with the corresponding rules
  • Several rules for same category

```cpp
using category = string;
using rule = vector<string>;
using rule_collection = vector<rule>;
using grammar = map<category, rule_collection>;
```
Reading the Grammar

// read a grammar from a given input stream
grammar read_grammar(istream& in)
{
    grammar ret;
    string line;
    // read the input
    while (getline(in, line)) {
        // split the input into words
        vector<string> entry = split(line);
        if (!entry.empty()) {
            // use the category to store the associated rule
            ret[entry[0]].push_back(
                rule(entry.begin() + 1, entry.end()));
        }
    }
    return ret;
}
Generating the random Sentence

• Start off with category `<sentence>`
• Assemble the output in pieces from various rules
• Result is a vector<string> holding the words of the generated sentence

```cpp
vector<string> generate_sentence(grammar const& g)
{
    vector<string> ret;
    generate_aux(g, "<sentence>", ret);
    return ret;
}
```
Generating the random Sentence

• Our algorithm `generate_aux()` knows how to query the grammar and how to collect the words

• Needs to decide whether a string is a category:

```cpp
// return true if 's' represents a category
bool bracketed(string const& s)
{
    return s.size() > 1 && s[0] == '<' && s[s.size() - 1] == '>'; 
}
```

• If it’s a category, look up rule and expand it
• If it’s not a category, copy word to output
Generating the random Sentence

```cpp
void generate_aux(grammar const& g, string const& word, vector<string>& ret)
{
    if (!bracketed(word)) {
        ret.push_back(word);
    }
    else {
        // locate the rule that corresponds to word
        auto it = g.find(word);
        if (it == g.end())
            throw logic_error("empty rule");
        // fetch the set of possible rules
        rule_collection const& c = it->second;
        // from which we select one at random
        rule const& r = c[nrand(c.size())];
        // recursively expand the selected rule
        for (auto i = r.begin(); i != r.end(); ++i)
            generate_aux(g, *i, ret);
    }
}
```
Recursive Function Calls

• First looks like it can’t possibly work

• However, if the word is not a category (not bracketed), it obviously works

• Let’s assume the word is a category, but its rule has only non-bracketed words
  • Obviously works as this will return the words immediately

• Now, let’s assume the word is a category, it’s rule has bracketed words, but those refer to only non-bracketed right hand sides
  • Obviously works as well...
Recursive Function Calls

• We do not know any sure way to explain recursion. Our experience is that people stare at recursive programs for a long time without understanding how they work. Then, one day, they suddenly get it—and they don't understand why they ever thought it was difficult.

• Evidently, the key to understanding recursion is to begin by understanding recursion. The rest is easy 😊
Pulling everything together

```cpp
int main() {
    // generate the sentence
    vector<string> sentence = generate_sentence(read_grammar(cin));

    // write the first word, if any
    auto it = sentence.begin();
    if (!sentence.empty())
        cout << *it++;

    // write the rest of the words, each preceded by a space
    for_each(it, sentence.end(), [](string s) {
        cout << " " << s; });
    cout << endl;

    return 0;
}
```
Selecting a random Element

• Writing nrand(n) is tricky!

• Standard library has function rand() returning a (pseudo) random number in range \([0, \text{RAND\_MAX})\)

• We need to map that range to new range \([0, n)\)

• Simplest solution: \(\text{rand()} \% n\)

• However, this is not good:
  • For small \(n\), returned number tends to be not random
  • For large \(n\), returned numbers are not randomly distributed anymore

• Solution: create \(n\) equally sized buckets and find what bucket a given random number falls into
Selecting a Random Element

// return a random integer in the range [0, n)
int nrand(int n)
{
    if (n <= 0 || n > RAND_MAX)
        throw domain_error("Argument to nrand is out of range");
    int const bucket_size = RAND_MAX / n;
    int r;
    do {
        r = rand() / bucket_size;
    } while (r >= n);
    return r;
}
Selecting a Random Element

• Why does it print the same sentence whenever run?
  • Random number generators are not random
  • Generate a sequence of numbers with certain statistical properties
  • Each time they are used they generate the same sequence of numbers (by default)

• Use srand() to ‘seed’ the random number generator
• Use time(0) as a ‘random’ seed value
Performance Considerations

• Unlike to associative containers in other languages, map<> is not implemented as a hash table
  • For each key type those need a hash function
  • Performance is exquisitely sensitive to the details of this hash function.
  • There is usually no easy way to retrieve the elements of a hash table in a useful order.

• C++ associative containers are hard to implement in terms of hash tables:
  • The key type needs only the < operator or equivalent comparison function
  • Associative-container elements are always kept sorted by key

• C++11 introduces unordered_map<> , a hash table