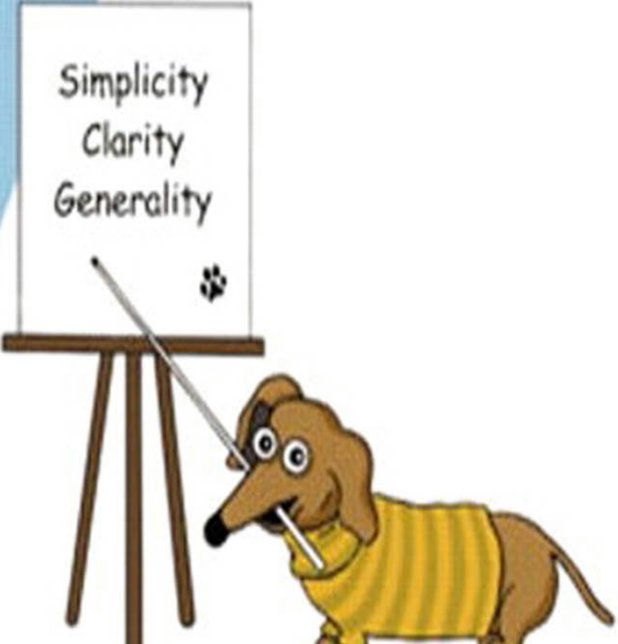
The Practice of Programming

Brian W. Kernighan
Rob Pike



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Practice of Programming

class, with which one can store and retrieve values of one type using objects of another type as keys.

In our application, Vectors of strings are the natural choice to hold prefixes and suffixes. We can use a Hashtable whose keys are prefix vectors and whose values are suffix vectors. The terminology for this type of construction is a *map* from prefixes to suffixes; in Java, we need no explicit State type because Hashtable implicitly connects (maps) prefixes to suffixes. This design is different from the C version, in which we installed State structures that held both prefix and suffix list, and hashed on the prefix to recover the full State.

A Hashtable provides a put method to store a key-value pair, and a get method to retrieve the value for a key:

```
Hashtable h = new Hashtable();
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

Our implementation has three classes. The first class, Prefix, holds the words of the prefix:

```
class Prefix {
    public Vector pref; // NPREF adjacent words from input
    ...
```

The second class, Chain, reads the input, builds the hash table, and generates the output; here are its class variables:

```
class Chain {
    static final int NPREF = 2; // size of prefix
    static final String NONWORD = "\n";
                    // "word" that can't appear
    Hashtable statetab = new Hashtable();
                    // key = Prefix, value = suffix Vector
    Prefix prefix = new Prefix(NPREF, NONWORD);
                    // initial prefix
    Random rand = new Random();
    ...
```

The third class is the public interface; it holds main and instantiates a Chain:

```
class Markov {
    static final int MAXGEN = 10000; // maximum words generated
    public static void main(String[] args) throws IOException
    {
        Chain chain = new Chain();
        int nwords = MAXGEN;

        chain.build(System.in);
        chain.generate(nwords);
    }
}
```

When an instance of class `Chain` is created, it in turn creates a hash table and sets up the initial prefix of `NPREF NONWORD`s. The `build` function uses the library function `StreamTokenizer` to parse the input into words separated by white space characters. The three calls before the loop set the tokenizer into the proper state for our definition of ‘‘word.’’

```
// Chain build: build State table from input stream
void build(InputStream in) throws IOException
{
    StreamTokenizer st = new StreamTokenizer(in);

    st.resetSyntax();                        // remove default rules
    st.wordChars(0, Character.MAX_VALUE); // turn on all chars
    st.whitespaceChars(0, ' ');          // except up to blank
    while (st.nextToken() != st.TT_EOF)
        add(st.sval);
    add(NONWORD);
}
```

The add function retrieves the vector of suffixes for the current prefix from the hash table; if there are none (the vector is null), add creates a new vector and a new prefix to store in the hash table. In either case, it adds the new word to the suffix vector and advances the prefix by dropping the first word and adding the new word at the end.

```
// Chain add: add word to suffix list, update prefix
void add(String word)
{
    Vector suf = (Vector) statetab.get(prefix);
    if (suf == null) {
        suf = new Vector();
        statetab.put(new Prefix(prefix), suf);
    }
    suf.addElement(word);
    prefix.pref.removeElementAt(0);
    prefix.pref.addElement(word);
}
```

Notice that if `suf` is null, `add` installs a new `Prefix` in the hash table, rather than `prefix` itself. This is because the `Hashtable` class stores items by reference, and if we don't make a copy, we could overwrite data in the table. This is the same issue that we had to deal with in the C program.

The generation function is similar to the C version, but slightly more compact because it can index a random vector element directly instead of looping through a list.

```
// Chain generate: generate output words
void generate(int nwords)
{
    prefix = new Prefix(NPREF, NONWORD);
    for (int i = 0; i < nwords; i++) {
        Vector s = (Vector) statetab.get(prefix);
        int r = Math.abs(rand.nextInt()) % s.size();
        String suf = (String) s.elementAt(r);
        if (suf.equals(NONWORD))
            break;
        System.out.println(suf);
        prefix.pref.removeElementAt(0);
        prefix.pref.addElement(suf);
    }
}
```

The two constructors of `Prefix` create new instances from supplied data. The first copies an existing `Prefix`, and the second creates a prefix from `n` copies of a string; we use it to make `NPREF` copies of `NONWORD` when initializing:

```
// Prefix constructor: duplicate existing prefix
Prefix(Prefix p)
{
    pref = (Vector) p.pref.clone();
}

// Prefix constructor: n copies of str
Prefix(int n, String str)
{
    pref = new Vector();
    for (int i = 0; i < n; i++)
        pref.addElement(str);
}
```

`Prefix` also has two methods, `hashCode` and `equals`, that are called implicitly by the implementation of `Hashtable` to index and search the table. It is the need to have an explicit class for these two methods for `Hashtable` that forced us to make `Prefix` a full-fledged class, rather than just a `Vector` like the suffix.

The `hashCode` method builds a single hash value by combining the set of `hashCodes` for the elements of the vector:

```
static final int MULTIPLIER = 31;   // for hashCode()

// Prefix hashCode: generate hash from all prefix words
public int hashCode()
{
    int h = 0;

    for (int i = 0; i < pref.size(); i++)
        h = MULTIPLIER * h + pref.elementAt(i).hashCode();
    return h;
}
```

and equals does an elementwise comparison of the words in two prefixes:

```
// Prefix equals: compare two prefixes for equal words
public boolean equals(Object o)
{
    Prefix p = (Prefix) o;

    for (int i = 0; i < pref.size(); i++)
        if (!pref.elementAt(i).equals(p.pref.elementAt(i)))
            return false;
    return true;
}
```

The Java program is significantly smaller than the C program and takes care of more details; Vectors and the Hashtable are the obvious examples. In general, storage management is easy since vectors grow as needed and garbage collection takes care of reclaiming memory that is no longer referenced. But to use the Hashtable class, we still need to write functions hashCode and equals, so Java isn't taking care of all the details.

Comparing the way the C and Java programs represent and operate on the same basic data structure, we see that the Java version has better separation of functionality. For example, to switch from Vectors to arrays would be easy. In the C version, everything knows what everything else is doing: the hash table operates on arrays that are maintained in various places, lookup knows the layout of the State and Suffix structures, and everyone knows the size of the prefix array.

```
% java Markov <jr_chemistry.txt | fmt
Wash the blackboard.  Watch it dry.  The water goes
into the air.  When water goes into the air it
evaporates.  Tie a damp cloth to one end of a solid or
liquid.  Look around.  What are the solid things?
Chemical changes take place when something burns.  If
the burning material has liquids, they are stable and
the sponge rise.  It looked like dough, but it is
burning.  Break up the lump of sugar into small pieces
and put them together again in the bottom of a liquid.
```

Exercise 3-4. Revise the Java version of markov to use an array instead of a Vector for the prefix in the State class. □

3.6 C++

Our third implementation is in C++. Since C++ is almost a superset of C, it can be used as if it were C with a few notational conveniences, and our original C version of `markov` is also a legal C++ program. A more appropriate use of C++, however, would be to define classes for the objects in the program, more or less as we did in Java; this would let us hide implementation details. We decided to go even further by using the Standard Template Library or STL, since the STL has built-in mechanisms that will do much of what we need. The ISO standard for C++ includes the STL as part of the language definition.

The STL provides containers such as vectors, lists, and sets, and a family of fundamental algorithms for searching, sorting, inserting, and deleting. Using the template features of C++, every STL algorithm works on a variety of containers, including both user-defined types and built-in types like integers. Containers are expressed as C++ templates that are instantiated for specific data types; for example, there is a `vector` container that can be used to make particular types like `vector<int>` or `vector<string>`. All `vector` operations, including standard algorithms for sorting, can be used on such data types.

In addition to a `vector` container that is similar to Java's `Vector`, the STL provides a `deque` container. A deque (pronounced ``deck'') is a double-ended queue that matches what we do with prefixes: it holds `NPREF` elements, and lets us pop the first element and add a new one to the end, in $O(1)$ time for both. The STL deque is more general than we need, since it permits push and pop at either end, but the performance guarantees make it an obvious choice.

The STL also provides an explicit `map` container, based on balanced trees, that stores key-value pairs and provides $O(\log n)$ retrieval of the value associated with any key. Maps might not be as efficient as $O(1)$ hash tables, but it's nice not to have to write any code whatsoever to use them. (Some non-standard C++ libraries include a `hash` or `hash_map` container whose performance may be better.)

We also use the built-in comparison functions, which in this case will do string comparisons using the individual strings in the prefix.

With these components in hand, the code goes together smoothly. Here are the declarations:

```
    typedef deque<string> Prefix;
    map<Prefix, vector<string> > statetab; // prefix -> suffixes
```

The STL provides a template for deques; the notation `deque<string>` specializes it to a deque whose elements are strings. Since this type appears several times in the program, we used a `typedef` to give it the name `Prefix`. The map type that stores prefixes and suffixes occurs only once, however, so we did not give it a separate name; the `map` declaration declares a variable `statetab` that is a map from prefixes to vectors of strings. This is more convenient than either C or Java, because we don't need to provide a hash function or `equals` method.

The main routine initializes the prefix, reads the input (from standard input, called `cin` in the C++ `iostream` library), adds a tail, and generates the output, exactly as in the earlier versions:

```
// markov main: markov-chain random text generation
int main(void)
{
    int nwords = MAXGEN;
    Prefix prefix;                  // current input prefix

    for (int i = 0; i < NPREF; i++) // set up initial prefix
        add(prefix, NONWORD);
    build(prefix, cin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

The function `build` uses the `iostream` library to read the input one word at a time:

```
// build: read input words, build state table
void build(Prefix& prefix, istream& in)
{
    string buf;

    while (in >> buf)
        add(prefix, buf);
}
```

The string `buf` will grow as necessary to handle input words of arbitrary length.

The `add` function shows more of the advantages of using the STL:

```
// add: add word to suffix list, update prefix
void add(Prefix& prefix, const string& s)
{
    if (prefix.size() == NPREF) {
        statetab[prefix].push_back(s);
        prefix.pop_front();
    }
    prefix.push_back(s);
}
```

Quite a bit is going on under these apparently simple statements. The `map` container overloads subscripting (the `[]` operator) to behave as a lookup operation. The expression `statetab[prefix]` does a lookup in `statetab` with `prefix` as key and returns a reference to the desired entry; the vector is created if it does not exist already. The `push_back` member functions of `vector` and `deque` push a new string onto the back end of the vector or deque; `pop_front` pops the first element off the deque.

Generation is similar to the previous versions: