Associative arrays in C++

Andrew Koenig

AT&T Bell Laboratories
184 Liberty Corner Road; Warren NJ 07060; ark@europa.att.com

ABSTRACT

An *associative array* is a one-dimensional array of unbounded size whose subscripts can be non-integral types such as character strings. Traditionally associated with dynamically typed languages such as AWK and SNOBOL4, associative arrays have applications ranging from compiler symbol tables to databases and commercial data processing.

This paper describes a family of C++ class definitions for associative arrays, including programming examples, application suggestions, and notes on performance. It concludes with a complete programming example which serves the same purpose as a well-known system command but is much smaller and simpler.

1. Introduction

An *associative array* is a data structure that acts like a one-dimensional array of unbounded size whose subscripts are not restricted to integral types: it associates values of one type with values of another type (the two types may be the same). Such data structures are common in AWK[1] and SNOBOL4[2], but unusual in languages such as C[3], which have traditionally concentrated on lower-level data structures.

Suppose that `m` is an associative array, or *map*, whose subscripts, or *keys*, are of type `String` and whose elements, or *values*, are integers. Suppose further that `s` and `t` are `String` variables. Then `m[s]` is an `int` called *the value associated with* `s`. In this example `m[s]` is an *lvalue*: it may be used on either side of an assignment. If `s` and `t` are equal, `m[s]` and `m[t]` are the same `int` object. If `s` and `t` are unequal, `m[s]` and `m[t]` are different objects.

This is exactly analogous to the way ordinary arrays work in most languages, except that it is much harder to pre-allocate enough memory in advance to contain all relevant elements. Thus some kind of dynamic memory scheme is necessary to allow a map to accommodate all its elements.

C data structures do not expand automatically, and no such expanding structures are built into C++ either. But C++[4] is extensible: although it does little to provide high-level data structures by itself, it makes it possible to implement such structures in ways that are easy to use.

1.1 An example

The following C++ program uses associative arrays to count how many times each distinct word occurs in its input:

```
#include <String.h>
#include <Map.h>

Mapdeclare(String,int)
Mapimplement(String,int)

main()
{
	Map(String,int) count;
	String word;

	while (cin >> word)
		count[word]++;

	Mapiter(String,int) p (count);
	while (++p)
		cout << dec(p.value(),4) << " " << p.key() << "\n";
}
```

The `#include` statements bring in declarations for a locally-written string package[5] and the author's associative array package, respectively. The string package requests `<stream.h>` so we do not have to name it explicitly.

`Mapdeclare(String,int)` defines two new types: `Map(String,int)` to represent an array of integers whose keys are of type `String` and `Mapiter(String,int)` to represent an element of such an array.

The first declaration in the main program thus defines `count` to be of type `Map(String,int)`.

The program does most of its work in the first loop. It reads each word from its input into `word` and increments the corresponding element of `count`. Trying to increment a non-existent element automatically creates the element first and gives it an initial value of `0`. This value is almost immediately incremented to `1`.

When the input is exhausted, the first loop finishes. All the information we need is now in `count`; we need merely print it. To do this, we use a *map iterator*.

A map iterator refers to a particular map (associative array) and usually to a particular element of that map. The declaration

```
Mapiter(String,int) p (count);
```

defines `p` to be an iterator that will refer to an element of `count`. Since we did not specify a particular element, `p` is initially *vacuous*: it does not refer to any element.

Incrementing a vacuous map iterator causes it to refer to the first element of the map; testing it in a `while` statement yields *true* as long as the iterator is not vacuous. Thus the final loop steps `p` through all the elements of `count`.

Every map element has two parts: a *key* and a *value*. In the final statement we print `p.value()` and `p.key()`, using the `dec` function to get the output to line up neatly.

Here is the result, collected into two columns, of giving a copy of this program as input to itself:

```
2 "                                1 "\n";
2 #include                         1 (++p)
1 (cin                             1 (count);
4 <<                               1 <Map.h>
1 <String.h>                       1 >>
1 Map(String,int)                  1 Mapdeclare(String,int)
1 Mapimplement(String,int)         1 Mapiter(String,int)
1 String                           1 count;
1 count[word]++;                   1 cout
1 dec(p.value(),4)                 1 main()
1 p                                1 p.key()
2 while                            1 word)
1 word;                            1 {
1 }
```

The program is obviously naive about the role of punctuation, but the output is otherwise about what one would expect. Notice especially that the output is sorted by key. The ordering, of course, depends on the particular C++ implementation's collating sequence for character strings. In ASCII, special characters come before letters and upper-case letters come before lower-case letters. Thus `while` appears before `word` and `String` comes before `count`.

2. Defining `Map` classes

Every file in every program that uses a map class must include `<Map.h>`. This header file defines four macros: `Mapdeclare`, `Mapimplement`, `Map`, and `Mapiter`. Each of these macros takes two type names as arguments representing the key and value types for a particular map class. Thus we use `Map(int,double)` as the class name of a map with `int` keys and `double` values, `Map(String,String)` as the class name of a map with `String` keys and `String` values, and so on.

2.1 Key and value types

There are several restrictions on the key and value types:

- Each type name must be expressed as a single identifier. Thus one cannot refer to `Map(char*,int)`: `char*` is not an identifier. Use a `typedef` declaration to circumvent this restriction.
- Each type must have *value semantics*. A map stores copies of the values given it, so it must be possible to copy such values. Moreover, a value stored in a map must persist, and be valid, as long as the map itself exists. Thus even if `Map(char*,int)` were syntactically possible, it would be a bad idea unless extreme care were taken to preserve the memory referenced by each pointer stored in the map. The `String` class does have value semantics and is therefore ideal for use in maps. Other types with value semantics include `int`, `char`, `long`, and `double`.

  There is no way for the map package to check the semantics of its underlying types. Hence trying to define a map on types of inappropriate semantics usually leads to disaster.
- Neither the key nor the value type may require arguments in its constructor, although either type may permit them. Equivalently, it must be possible to declare an object of each type without initializing it.
- The key type should not contain any underscores in its name. Failure to observe this restriction might cause two types that are conceptually different to generate the same internal name. This is consequence of implementation details that are beyond the scope of this paper.

In addition to these restrictions, it must be possible to compare keys for sorting purposes. In other words, if `k1` and `k2` are two keys, `k1<k2` must be well-defined. Moreover, when `<` is applied to two keys, it must be a *strong total order relation.* That is, it must have the following properties:

- `k1<k1` must never hold for any key `k1`.
- For any two keys `k1` and `k2`, either `k1<k2`, `k2<k1`, or `k1` and `k2` are equal. No more than one of these conditions may hold for any particular values of `k1` and `k2`.
- For any three keys `k1`, `k2`, and `k3`, if `k1<k2` and `k2<k3` both hold then `k1<k3` must also hold.

Notice that we do not require that `k1==k2` or `k1>k2` be defined because we can infer `k1>k2` from `k2<k1` and `k1==k2` from `!(k1<k2||k2<k1)`.

The integral types (`int`, `short`, `char`, and `long`) and the `String` class all have an appropriate definition for `<`. The floating-point types are a different matter: some machines support a special floating-point ''not-a-number'' value (NaN) that is neither less than, greater than, or equal to anything else. Using floating-point keys on such a machine invites disaster if a key is ever NaN. Floating-point *values* cause no trouble.

2.2 `Mapdeclare` and `Mapimplement`

If `K` is the key type and `V` is the value type, saying

```
Mapdeclare(K,V)
```

declares the classes `Map(K,V)` and `Mapiter(K,V)`. Thus the user must use `Mapdeclare` in every file in the program that uses either of these classes.

However, `Mapdeclare` includes only the declarations of these classes. It does not include the *definitions* of the classes' member functions: those appear in `Mapimplement`. Thus every program that uses `Map(K,V)` or `Mapiter(K,V)` *anywhere* must use `Mapimplement(K,V)` in *exactly one file* of the program. This file must also use `Mapdeclare(K,V)`.

Of course, if `K` or `V` is itself a class, the class definition must appear before the use of `Mapdeclare`.

For instance, we might split our sample program into two files. One file uses `Map(String,int)` and must therefore declare it:

```
#include <String.h>
#include <Map.h>

Mapdeclare(String,int)

main()
{
      Map(String,int) count;
      String word;

      while (cin >> word)
            count[word]++;

      Mapiter(String,int) p (count);
      while (++p)
            cout << dec(p.value(),4) << " " << p.key() << "\n";
}
```

The other file defines the map class member functions:

```
#include <String.h>
#include <Map.h>

Mapdeclare(String,int)
Mapimplement(String,int)
```

These two files can be separately compiled and linked together with the same effect as the original

sample program. Doing this is only of pedagogical interest for a program this small, but is important for larger programs.

In subsequent code fragments we will omit `#include` statements and uses of `Mapdeclare` and `Mapimplement`.

2.3 Using a map

Indexing is the fundamental map operation: if `m` is of type `Map(K,V)` and `x` is of type `K`, then `m[x]` is an lvalue of type `V`. Thus we store a value `v` in `m[x]` by writing

```
m[x] = v;
```

and retrieve it again by writing

```
v = m[x];
```

If we ask for the value of `m[x]` without ever having specified one, we get the *default value* of type `V`: the value of an otherwise uninitialized static variable of type `V`. It is very important that this is the value of a *static* variable in case `V` is a built-in type: the default value of an ordinary `int` variable is undefined but static `int` variables are initialized to zero. Thus after executing:

```
Map(String,int) m;

m["Morristown"]++;
```

`m` has one element with key `Morristown` and value `1`.

Because evaluating `m[k]` creates an element of `m` if an appropriate element does not already exist, we need a way to test whether an element exists without creating one. Thus every map has a member function called `element` which returns an appropriate value:

```
if (m.element(k))
      // m[k]  exists
```

Thus if we did not wish to rely on default values for map elements, we could write:

```
if (m.element(word))
      m[word]++;
else
      m[word] = 1;
```

The number of elements in a map `m` is `m.size()`.

We can explicitly nominate a default value for map elements by passing that value to the map constructor. For example:

```
Map(String,String) s ("(undefined)");
```

declares `s` to be a map in which keys and values are both of type `String` and in which element values are initially `(undefined)`.

2.4 Copying a map

If `m` is a map,

```
Map(K,V) n = m;
```

will create a new map `n` whose elements are logical copies of the elements of `m` (that is, the result of applying `operator=`). The default value for subsequently created elements of `n` will be the same as for `m`. The same effect can be obtained by writing:

```
Map(K,V) n (m);
```

If `m` and `n` are maps with the same key and value types, then

```
m = n;
```

effectively discards all the elements of `m` and creates new elements that are logical copies of all the elements of `n`. This does not change the default value for subsequently created elements of `m`; the value, if any, given when `m` was created remains.

One consequence of this is that passing a map as a function argument copies all the elements of the map. This preserves the normal call-by-value semantics of C++ but may have unexpected performance consequences. Consider writing functions whose arguments are references to maps instead of maps themselves.

3. Iterators

Every map class `Map(K,V)` has a companion *map iterator* class `Mapiter(K,V)` which is useful for extracting elements from a map. A map iterator refers to a particular map and optionally to a particular element of that map. For example:

```
Map(String,int) m;
Mapiter(String,int) mi (m);
```

These declarations create a map `m` and an iterator `mi` that can refer to an element of `m`. We did not specify any element in this example, so `mi` doesn't refer to one right now. Thus we call `mi` *vacuous*.

We can make `mi` identify the element of `m` with key `k` by saying

```
mi = m.element(k);
```

If `m[k]` exists, `mi` now identifies it; otherwise `mi` is vacuous.

Thus we see that `element` actually yields an iterator. We can tell whether an iterator is vacuous by using it as the subject of a test:

```
if (mi)
        // mi is not vacuous
```

Assigning a map to an iterator yields a vacuous iterator. Thus, after executing:

```
mi = m;
```

`mi` is a vacuous iterator referring to `m`.

Map elements are always kept sorted by key. Thus it makes sense to ask about the lowest and highest keys. For a map `m`, `m.first()` is an iterator that refers to the first element (the element with the lowest key) and `m.last()` is an iterator that refers to the last element (the element with the highest key). If `m` is empty, both `m.first()` and `m.last()` are vacuous.

If `mi` refers to an element, `++mi` makes `mi` refer to the element whose key immediately follows the current one. Similarly `--mi` makes `mi` refer to the immediately preceding key.

If `mi` is vacuous, `++mi` effectively assigns `m.first()` to `mi` and `--mi` effectively assigns `m.last()` to `mi`. If `mi` refers to `m.last()`, `++mi` makes `mi` vacuous, and if `mi` refers to `m.first()`, `--mi` makes `mi` vacuous. Thus we can think of all the keys in a map as being circularly connected, with a vacuous ''key'' marking a starting (and ending) point in the circle.

At this point we know at least two ways to refer to every element in a map:

```
for (Mapiter(K,V) mi = m.first(); mi; ++mi)
        // do something with mi
```

or

```
for (Mapiter(K,V) mi = m; ++mi; )
        // do something with mi
```

but we need a way to get at the key and value parts of the element referenced by `mi`. To this end, map

iterators have member functions called `key` and `value`. Thus we might write:

```
for (Mapiter(K,V) mi = m.first(); mi; ++mi) {
      K k = mi.key();
      V v = mi.value();
      // do something
}
```

If `mi` is vacuous, `mi.key()` and `mi.value()` are copies of static variables of type `K` and `V` respectively, to forestall core dumps and other disasters.

4. Performance

Associative arrays greatly simplify many programming tasks, but that is almost irrelevant if they are too expensive to use. Thus, while a map will never be as fast as an ordinary array, we have gone to some effort to ensure that the map classes perform reasonably well. In general, for a map with *n* elements,

- Searching for an element takes *O(log n)* time.
- Inserting a new element takes *O(log n)* time plus the time to call `malloc` once.[1]
- Incrementing or decrementing an iterator takes *O(log n)* time. Visiting every element of a map takes *O(n)* time (not *O(n log n)*).
- Copying an entire map takes *O(n log n)* time.

These are worst-case times: there is no pathological input that might cause the performance to degrade beyond these bounds.

A map takes a trivial amount of space for the map itself plus an additional overhead per element of three pointers, one character, and whatever additional overhead `malloc` imposes. Note that maps only require memory to be allocated for the elements actually stored.

What about absolute execution time?

As an example, we wrote a small program to generate 20,000 random integers and store them in a conventional array and a map. On our system (A DEC MicroVAX II running the Ninth Edition of the UNIX system), it takes 0.9 seconds to fill the conventional array and 8.8 seconds to fill the map, including the time spent in the `rand` function. So in this application maps are about ten times slower than conventional arrays.

But of course maps are not intended to replace conventional arrays where speed is all that matters – they do other things as well. For instance, storing values in a map has the side effect of sorting them by key. Thus using the random numbers as keys sorts them for free. Picking them out of the map again takes 0.8 seconds. In other words, using a map to sort 20,000 random integers takes 9.6 seconds: 8.8 seconds to put them in and 0.8 to take them out again.

In contrast, using the `qsort` routine to sort the conventional array takes 17 seconds.

5. Suggested applications

Although a map is not a good replacement for a conventional array when time is critical, time is often not critical. For example, we have seen many programs that use conventional arrays to build up an array of arguments to hand to some system command or other utility routine. The usual approach is to make the array some arbitrary size and then either not check for overflow (so that the program aborts

---

1. On some systems, `malloc` runs in time proportional to the number of blocks already allocated. We believe this could cause performance problems for some applications on such systems. The next version of C++ provides a clean way to define special-purpose `new` and `delete` operators for a particular class; we will consider that facility when it is widespread.

unpredictably when overflow occurs) or check for it and terminate the program explicitly.

A map can often be used in this context as if it were a conventional array of unbounded size. Even if the utility being called demands a conventional array as its input, it is possible to use the `size` function to allocate the right amount of memory at the last minute and then copy the elements. This approach may waste some machine resources, but the waste is often negligible compared to the execution time of the utility itself and carries a compensating gain in generality and robustness.

Maps are a good candidate for use in compiler symbol tables. In particular, it is unnecessary to sort such a symbol table in order to print a cross-reference listing.

The value of a map element can itself be a map (use `typedef` to get around the single-identifier restriction). If `m` is such a map, `m[k1][k2]` has the expected meaning: `m[k1]` is the map that is the value of the element of `m` with key `k1`, and `m[k1][k2]` is the value of the element of `m[k1]` with key `k2`.

6. Implementation notes and warnings

These classes are presently implemented as AVL trees. An AVL tree is a binary tree with the added requirement that the heights of the two subtrees of each node may differ by no more than one[6]. Although an algorithm exists for efficient deletion from an AVL tree, it is quite a nuisance and we have not implemented it in the current version.

We believe our implementation is reasonably robust. There are, however, a few places where a user might get into trouble:

- If `m` is a map, referring to `m[i]` creates it if it did not already exist. It is therefore a bad idea, though a tempting one, to check whether a map element exists by comparing its value to the default value. Use the `element` function instead.
- A map iterator is conceptually a pointer. Like any pointer, an iterator invites the possibility of a dangling reference if the lifetime of an iterator exceeds that of the associated map. For example:

  ```
  Map(K,V)* p;
  p = new Map(K,V);
  Mapiter(K,V) mi (*p);
  delete p;
  ```

  Because it is impossible to create an iterator without making it refer to some map, it is somewhat tricky to step into this particular trap. Nevertheless it is possible.
- Copying a map makes a logical copy of each element. If the map is large, this can take a long time. Many functions that deal with maps can deal just as profitably with pointers or references to maps instead.
- Data types used in maps should only be pointers if it is possible to ensure that those pointers refer to memory at least as persistent as the maps themselves. Otherwise some map element will ultimately contain a pointer to garbage, with chaos as the result.

7. Another example: topological sorting

We conclude with a non-trivial programming example that uses maps, strings, and lists. We will say enough about strings and lists in passing to enable the reader to follow the exposition; construction of appropriate class definitions is an interesting exercise for the reader.

7.1 The problem

Our input file consists of words separated by white space (spaces, tabs, newlines). Each word is considered to represent some abstract object; the words are taken in pairs to assert that the first object of each pair is ‘‘less than’’ the second. If both words in the pair are the same, that pair just says that the particular word exists without saying anything about its ordering. The problem is to find a single linear ordering that satisfies all the given constraints. This may be impossible because of a loop in the

ordering. In that case, the program should say so.

For example, the following input represents some constraints on clothing:

```
pants       belt
left_sock   left_shoe
right_sock  right_shoe
shirt       shirt
pants       left_shoe
pants       right_shoe
```

Here, we say that one article of clothing is ‘‘less than’’ another if it must be put on before the other. Thus each sock must be put on before the corresponding shoe, it is difficult to put on pants over shoes, one must put one's pants on before one's belt, and there is a shirt in there somewhere.[2]

One ordering that satisfies all these constraints is:

```
left_sock
pants
right_sock
shirt
belt
left_shoe
right_shoe
```

There are others, of course.

Topological sorting is often used to put modules into an appropriate order for load module archives in the UNIX® system; there is therefore a topological sort utility program called `tsort` available as a standard for comparison.

7.2 Implementation

We use an algorithm that is easy to understand and describe:[7] we read pairs of items, or *tokens*, keeping for each token a *count* of its predecessors and a *list* of its successors. Tokens with no predecessors can be written out immediately. When we write a token, we decrement the predecessor count of all its successors; decrementing a count to zero means that token can then be written. If we run out of tokens to write before we've dealt with every token, that means there is a loop somewhere.

Here are the details. First we bring in header files and say that we are going to want to use lists of strings:

```
#include <String.h>
#include <list.h>
#include <Map.h>

listdeclare(String)
listimplement(String)
```

The list package works similarly to the map package: the `listdeclare` and `listimplement` macros say that we are going to want to use lists of `String` objects.

Now we define a structure to represent the information we need to store about an object: a count of its predecessors and a list of its successors:

2. The author does not ordinarily wear a tie.

```
struct token {
	int predcnt;
	String_list succ;
};
```

A `String_list` is a list of `String` objects, as defined by the `listdeclare` macro.[3]

Having defined a structure with the information we need, we can now define a map whose keys are strings and whose values are tokens:

```
Mapdeclare(String,token)
Mapimplement(String,token)
```

We are ready to do some real work. The program first reads pairs of tokens. If both tokens in the pair are identical, we just record that the token exists. Otherwise, we increment the predecessor count of the second one and add the second to the successor list of the first. Here we use the `put` member function of a list, which appends its argument to the end of the list.

```
Map(String,token) m;
String p, s;

while (cin >> p >> s) {
	if (p == s)
		(void) m[p];
	else {
		m[s].predcnt++;
		m[p].succ.put(s);
	}
}
```

We look at each token and make `zeroes` into a list of the names of the tokens with no predecessors:

```
String_list zeroes;

for (Mapiter(String,token) i = m.first(); i; ++i) {
	if (i.value().predcnt == 0)
		zeroes.put(i.key());
}
```

Next we print the tokens on the list. Each time we print a token, we decrement the predecessor count of its successors. If a count reaches zero, we add that token to the list. Along the way, we count how many tokens we print.

To do this, we use the `getX` function, which removes an element from the beginning of a list, places the element in its argument (which is a `String&` in this case), and returns zero (false) or nonzero (true) to reflect the success of the operation.

---

3. As written, this example requires a version of the C++ translator that is still experimental; that version defines assignment and initialization of structures in terms of assignment and initialization of their elements, even if those elements have constructors.[8] For present versions of C++, we write it as follows:

```
struct token {
	int predcnt;
	String_list succ;
	void operator= (token& t) { predcnt = t.predcnt; succ = t.succ; }
	token (token& t): predcnt (t.predcnt), succ (t.succ) {}
	token() { }
};
```

```
int n = 0;

while (zeroes.getX (p)) {
      cout << p << "\n";
      n++;
      String_list& t = m[p].succ;
      while (t.getX (s)) {
            if (--m[s].predcnt == 0)
                  zeroes.put (s);
      }
}
```

Finally, we check whether we have printed every token:

```
if (n != m.size())
      cout << "the ordering contains a loop\n";
```

7.3 Performance

This straightforward program is about a quarter the size (in lines of source code) of `tsort`, so one should not be surprised to find that it runs substantially more slowly.

In practice, however, this is not the case. For very small inputs, `tsort` is faster because `String` input is quite slow in our present C++ implementation.[4] However, for larger inputs, the logarithmic behavior of maps wins over the quadratic behavior of the algorithm used in `tsort`.[5]

As an example, we used two sets of test data. Because topological sorting is often used on program dependencies, we generated our test data from two actual load module libraries: the C library on our system and the PORT library of mathematical software. Here are the results:

	C	PORT
total relations	742	6,513
distinct tokens	248	1,505
`tsort` execution time	7.0	406.5
C++ program execution time	9.3	101.8
input overhead	4.9	39.5

The first two lines in the table characterize the size of the problem in terms of the total number of relations and the number of distinct tokens in the input data. The next two lines give the execution time, in seconds, to process each set of data. Note that the C++ program is slightly slower on the smaller sample and much faster on the larger sample.

The last line shows how much of the execution time of the C++ program can be attributed to input overhead. This shows that if we could eliminate input overhead entirely, we could more than double the speed of the C++ program (remove 4.9 out of 9.3 seconds) on the small sample and significantly reduce it (remove 39.5 out of 101.8 seconds) on the large one. There are several ways to do this (aside from the obvious one of waiting for the library to improve); it is left as an exercise for the reader.

8. Conclusion

We have described a C++ implementation of a high-level data structure. The availability of this structure led us to demonstrate it by a simple solution to a well-known problem. Our solution turned out to be a

4. This problem is understood; current work aims to reduce the overhead.

5. To forestall the objection that it is unfair to compare a quadratic algorithm to one that is intrinsically faster, we note that (a) the present implementation of `tsort` was considered ‘‘not worth improving’’ for years, and (b) when David Gay finally improved it on the Ninth Edition system, his (very fast) improvement was about ten times the size of our example.

quarter the size of the ''standard'' one. Moreover, its performance on large input data was substantially better as well.

REFERENCES

1. Aho, Kernighan, and Weinberger: *The AWK Programming Language;* Addison-Wesley, 1988.
2. Griswold, Poage, and Polonsky: *The SNOBOL4 Programming Language;* Prentice-Hall, 1971.
3. Kernighan and Ritchie: *The C Programming Language;* Prentice-Hall, 1978
4. Stroustrup: *The C++ Programming Language;* Addison-Wesley, 1986
5. J. E. Shopiro: *Strings and Lists for C++*, unpublished internal memorandum
6. D. E. Knuth: *The Art of Computer Programming,* volume 3, section 6.2.3, page 451.
7. D. E. Knuth: *The Art of Computer Programming;* Volume 1, page 262
8. Stroustrup, *The Evolution of C++: 1985 to 1987;* Proceedings of the USENIX C++ Workshop, Santa Fe, New Mexico, 1987; pp. 1-21.