Maps with Expensive Keys

Andrei Alexandrescu

December 1, 2005

1 What's In a Symbol Name?

Well, I'll tell you what it is. A lot of embarrassment, that's what it is—particularly if the name happens to be a poorly chosen symbol name. If you're thinking of some foul word left in my source code and found by a scrutinizing colleague months later, you're mistaken. (I always use Romanian interjections when I need a "variable name that's impossible to occur in normal source code," so I'm covered there). The name I'm talking about is an innocuous `A`, and the opportunity for embarrassment showed itself right in front of some eigthty people anxiously waiting to explain myself. The `A` was staring at me from my own slides during my second talk at C++ Connections,[1] and the context was (paraphrased):

```
void* Alloc(Heap & h, size_t s) {
 return h.AllocImpl(
   (s + ((1 << A) - 1)) & ~A);
}
```

I'd felt good about myself while writing that code, good feeling that—just like Proust's madeleines [4]—came from back in time to make me smile jovially while I pulled the slide and asked the audience, "now, what does that expression do?" At that moment, someone asked, "what's `A`?" and the audience chuckled and looked at me waiting for a response.

Now, I believe all of us have had one moment when we entered a room and totally forgot why. (Judging by the way dogs enter rooms, they *always* forget.) Well, imagine how it's like to have such a moment in front of a large audience waiting for you to explain yourself in a foreign language. I don't know what you would do, and hopefully you'll never need to know. As of me, I stared at the expression for an eternity that lasted fifteen seconds, until a good soul told me "ah, doesn't matter, let's move on." Some highlights of the rest of the talk were red ears, a thickened accent, and a general sense of awkwardness. Sigh.

If only `A` had a better name, it would have revealed the bug that's lurking in the code above. You see, `A` stands for "Alignment," and the intent is to round up `s` to the closest alignment. The problem is, the use of `A` in the shift implies that it's given in bits (e.g., "align to clear the last `A` bits"), while `A`'s use in the logical operation suggests that `A` is given in bytes (e.g., "align to a multiple of `A`"). An appropriate name would have worked wonders towards clarifying the intent and fixing the bug:

```
const unsigned AlignBytes = 1 << AlignBits;
void* Alloc(Heap & h, size_t s) {
 return h.AllocImpl(
   (s + AlignBytes - 1) & ~(AlignBytes - 1));
}
```

That talk was my second worst ever, second only to my attempt at selling policy-based design to a crowd of Smalltalk aficionados. On the bright side, the conference was very enjoyable, with a strong C++ track, within which the prima donna was concurrent programming. In many people's opinion, concurrent programming will be the next dominating paradigm. But, let's leave that topic for later and turn our attention towards a flaw in `std::map`.

[1] `http://devconnections.com/shows/cppfall2005/default.asp?s=67`

2 Mailcontainer

But before that, let me bring up a letter that you might find interesting. Andrew Boothroyd writes about mandatory error codes [3]:

> We have successfully used this idea in our development. We do have a couple of minor questions, however:
>
> - Was there a deliberate reason that you didn't declare an assignment operator? Presumably, assigning from an `ErrorCode<T>` means reading it in the same way as copying it does, thus the assignment operator should mirror the behavior of the copy ctor?
> - In your opinion, would prefixing the explicit keyword to the template ctor make client code (a) more or less readable; (b) more or less robust?
>
> My personal view is that forcing the client to call the ctor explicitly makes it clear in the client code that an instance of `ErrorCode<T>` is not being constructed by another mechanism, e.g. type conversion, or a method of the type from which the conversion is being made.

Thanks for sharing. To answer the questions, it does make sense to implement `ErrorCode<T>::operator=(ErrorCode<T>)` with destructive semantics (also notice the unusual signature). About making `ErrorCode<T>`'s constructor **`explicit`**, I myself, after recently scrambling through a number of shell and scripting languages, have become increasingly inamored with the "make the default safe, allow unsafe expressiveness with extra syntax" mantra. I remain, however, ambivalent about making the constructor **`explicit`**—it risks of cluttering client code too much.

3 Maps with Expensive Keys

One nice thing about the STL containers is that they are extension-friendly—you can easily use them as a back-end for your own, more sophisticated, containers. Care for checked iterators? You don't need to start from scratch—you can use STL's original containers for storage and build on top of them. Want an always-sorted binary vector? You implement it as a thin shell over `std::vector` (that's exactly what `Loki::AssocVector` does). Dictionaries, factories, caches? `std::map` is there to help, either directly or as a time-saving implementation device.

I was, therefore, more than a bit surprised when stumbling upon a problem that has `std::map` written all over it, yet can't be solved with the help of an `std::map` in any reasonable way. This article discusses that situation and some possible remedies.

Let's start with two examples. The simplest would involve the often-used `map<string, something>`: user names and user IDs, stock tickers and stock prices, class names and pointers to functions, words in Proust's opera and their counts... You can easily access the "something" if you have a key in the desired format. The problem is, you might have the key in a different format that would require a conversion. What if, for example, in your high-speed, high-availability, power-hungry, bonus-bringing stock market program, you have securities data come down the wire in raw **`char*`** format, but you'd need to look it up in a `map<string, double>`?

```
void OnWirePacket(      // called a lot
    const char* sec,    // security name
    void* secInfo) {    // raw info
  // Creates an implicit temporary string
  const double price = map[sec];
  ...
}
```

You'd have to create a string from the **`char*`** (which could trigger a call all the way to the memory allocator), look it up in the map, and then likely throw it away. Employing a custom string and the small string optimization [1] would be an option, but in general securities come in a longer and more sophisticated format than the up-to-four-letters ticker symbols we all know (and love—as long, of course, as the stock market is bullish enough to let us mistake our naive elucubrations for insights of financial genius).

One other solution would be to store stock data as `map<const char*, double>`, but out the window are automatic memory management and other comforting amenities that true strings offer or at least strive chaotically to offer, as `std::string` seems to do. Ahem.

The second example is a real-world problem that inspired this article involves neural networks and memoizing. Simply put, a neural network is a function that takes vectors of number and returns vectors to numbers:

```
class NeuralNet {
  ...
  void Fun(             // NNs are fun
    const int * inputs,
    unsigned int inpCount,
    double * outputs
    unsigned int outCount);
};
```

(In the general case, the inputs could be floating numbers and/or the outputs could be integers.) Some neural networks learn as they go—are adaptive, in the sense that the output for the same input might vary across calls. But many usage scenarios of neural networks follow a "train once, use anywhere" mantra. You "train" a neural network to learn a specific function (one that would be prohibitively hard, or prohibitively expensive, to implement analytically) and then you use that neural network many times without ever changing its state. This setup has the consequence that the output of a trained neural network depends solely on its inputs. As far as efficiency goes, just computing the output pattern on an already-trained net involves quite expensive matrix multiplications and nonlinear math functions (such as exponentials) which aren't cheap.

Now let's assume that we have a problem in which input patterns tend to be quite repetitive, that is, the inputs of `NeuralFun` tend to not vary wildly within a time window. (That is the case for most signals and patterns. The real world is not jerky. Columnists sometimes are.) For such an input, instead of computing the output every time, it makes much sense to compute the output once and then save it. Then, if we detect the same input pattern occurs, we serve the stored response and we're done—no matrix multiplication, no exponentials, no nothing. This simple technique is known as memoization. ("To memoize" is also what managers inflict on programmers who didn't fill their TPS reports in time.)

Implementing a memoization scheme would naturally involve something like a `map<vector<unsigned>, vector<double>, comp>`. The comparison predicate would implement classic lexicographical comparison of two vectors:

```
struct MyLess {
  bool operator()(
      const vector<unsigned>& lhs,
      const vector<unsigned>& rhs) const {
    const unsigned
      *li = &*lhs.begin(),
      *ri = &*rhs.begin(),
      *const le = &*lhs.end(),
      *const re = &*rhs.end();
    for (; ; ++li, ++ri) {
      if (ri == re) return false;
      if (li == le) return true;
      if (*li != *ri) break;
    }
    return *li < *ri;
  }
};
```

The problem is, most of the time, input data doesn't come in the form of `vector`s, but instead as some pointer in a buffer that's been read from a file. Copying that buffer into yet another vector just for the sake of looking it up in the cache sounds much like selling apples just to buy pears. There's a more general problem lurking behind these examples.

4 Formalization

Let's formalize the problem a bit. Consider a class `K` (key), a class `V` (value), and a predicate `Less`. The predicate supports `operator()(const K&, const K&)`. With this troika we can build an `std::map`. Now, let's say we have some *alternate key types* `K1`, `K2`,... `Kn` that satisfy, for all `i` from `1` to `n`, the following two conditions:

- Constructing an object of type `K` from an object of type `Ki` is possible but not desirable for efficiency reasons; and
- `Ki` is directly comparable with `K`. That means you can implement a functor `Less_i` with the same semantics as `Less`, just without the temporary. To make things clear at the price of a yawn on your part: For every object `less` of type `Less` there is an object `less_i` of type `Less_i` such that the relationship:

  ```
  less_i(ki, k) == less(K(ki), k) &&
  less_i(k, ki) == less(k, K(ki))
  ```

  is true for all `ki` and `k`. Whew!

The charge is to implement a map that holds keys of type `K` (just like map), yet accepts for comparison alternate keys of type `Ki` without converting them to `K`.

Unfortunately, we need to dismiss `std::map` right off the bat. In spite of its considerable versatility, `std::map` is unable to serve as a back-end for our implementation. This is because all of `std::map`'s searching functions (such as `find`, `lower_bound`, and `operator[]`) require a `const K&`. By the rule of call-by-value, `std::map` needs an object of type `K` to even consider it for lookup. (We shall discuss later what changes to `std::map`'s interface might be useful for it to accept alternate key types.)

To give an example, consider `K` to be `std::string` and `Less` to be `std::less<std::string>`. Then, we can easily show that `const char*` (denoting zero-terminated strings) is an alternate key type. The proof is by construction—we implement `LessAsciiZString` as follows:

```
struct LessAsciiZString {
  bool operator()(
      const char* k1,
      const string& k2) const {
    return strcmp(k1, k2.c_str()) < 0;
  }
  bool operator()(
      const string& k1,
      const char* k2) const {
    return strcmp(k1.c_str(), k2) < 0;
  }
};
```

Have a `const char *` and a length instead of a null-terminated C-style string? We can readily define another alternate key type and comparitor:

```
typedef std::pair<const char*, const char*>
  MemRange;
struct LessMemRangeString {
  bool operator()(
      const MemRange k1,
      const string& k2) const {
    return std::lexicographical_compare(
        k1.first, k1.second,
        k2.begin(), k2.end());
  }
  bool operator()(
      const string& k1,
      const MemRange k2) const {
    return std::lexicographical_compare(
        k1.begin(), k1.end(),
        k2.first, k2.second);
  }
};
```

5 Design

Now, how to design such a map accepting alternate key types and implicitly alternate predicate functors? A number of design options spring to mind. One would be to have the map accept an unbounded number of comparitors in the form of a typelist [2]:

```
template <
  class K,
  class V,
  class TList = TYPELIST_1(std::less<K>)
>
class UberMap;
```

Such a design is viable and has certain advantages—it's easy to assemble a `Map` when you already have the comparitors lying around. For example, should you want to define a map that accepts not only `std::string`, but also zero-terminated strings and memory ranges, you'd write:

```
typedef UberMap<
  std::string,
  Something,
  TYPELIST_3(
    std::less<std::string>,
    LessAsciiZString,
    LessMemRangeString
  )
>
FastMap;
```

However, there's an even better possible design. How about collapsing all of the comparison predicates into one? Consider:

```
typedef std::pair<const char*, const char*>
  MemRange;
struct UberPred : std::less<string> {
  using std::less<string>::operator();
  bool operator()(
      const char* k1,
      const string& k2) const {
    ...
  }
  bool operator()(
      const string& k1,
      const char* k2) const {
    ...
  }
  bool operator()(
      const MemRange k1,
      const string& k2) const {
    ...
  }
  bool operator()(
      const string& k1,
      const MemRange k2) const {
    ...
  }
};
```

The `UberPred` class collects all of the predicates under one roof. `UberPred` also inherits the primary key type and injects its `operator()` through the `using` directive to give it a fighting chance. Then, we let overloading will easily take care of everything. To effect that, we implement `UberMap` as follows:

```
template <
  class K,
  class V,
  class Compare = std::less<K>
>
class UberMap {
  Compare pred_; // well, subject to EBO
  ...
public:
  template <class Kx>
  iterator find(const Kx& kx) {
    // Implement in terms of calls
    // to pred_(kx, something)
    // and pred_(something, kx)
  }
  ...
};
```

The design comprising all of the comparitors has simplicity on its side—you just plug the appropriate comparitor into a structure that otherwise is just like `std::map`, and you're done without any fuss, muss, or any other unpleasantry ending in "uss". The design also has the advantage of efficiency—the compiler generates one separate version of `find` for each type you call `find` with. On the downside, if you have a few predicates lying around, then... "some assembly required," as it reads on those impossible-to-put-together pieces of furniture. But fear not, with just a little handiwork you can assemble simple predicates into larger predicates quite easily:

```
struct UberPred
    : std::less<string>
    , LessAsciiZString
    , LessMemRangeString {
  using std::less<string>::operator();
  using LessAsciiZString::operator();
  using LessMemRangeString::operator();
};
```

6 Implementation

I bet you are starting this section hoping that I sat down and wrote a custom red-black tree implementation. Sorry, I didn't. But let's focus

our attention on implementing `UberMap` by hacking into `Loki::AssocVector` (which can be downloaded from `http://sf.net/projects/loki-lib`). `Loki::AssocVector` is a `std::map`-lookalike that uses a sorted vector for storage. Such a choice has the advantage of fast binary searches but slow insertions and removals. (Also, iterators are invalidated during insertions and removals, which doesn't happen with `std::map`'s node-based storage.)

You can download an implementation of `AssocVector` with secondary key types from `http://erdani.org/code`. The crux of the changes is in the implementation of `lower_bound`, which in turn helps in implementing `find`. Here it is:

```
// AssocVector With Alternate Keys
template<
  class K,
  class V,
  class C = std::less<K>,
  class A =
    std::allocator< std::pair<K, V> >
>
class AssocVectorWAK {
  ...
public:
  iterator lower_bound(const key_type& k) {
    MyCompare& me = *this;
    iterator left = begin(), right = end();
    while (left < right) {
      iterator i = left + (right - left) / 2;
      if (me(k, *i)) right = i;
      else if (me(*i, k)) left = i + 1;
    }
    return left;
  }
};
```

The `lower_bound` implementation is as unexciting an implementation of a binary search as it gets. The only reason for which we can't use `std::lower_bound` is that, at least in theory, that implementation might not work properly; a predicate with multiple overloads of `operator()` is not acceptable by the letter of the standard.

7 Conclusion

It is surprising that in spite of its versatility, `std::map` cannot efficiently accomodate keys of alternate type. One possible solution would be to extend `std::map`'s interface with two additional functions, `iterator left(iterator)` and `iterator right(iterator)`. These functions would return iterators pointing to the lesser and greater subtrees, respectively. You see, `std::map`'s iterators live in a two-dimensional world (the landscape of the tree they span), but only offer the unidimensional interface that conforms to bidirectional iterators. Uniformity is good, but then Procrustes was into uniformity, too. A map iterator should be different because it *is* different—it can instantly jump across the tree that's essential to implementing custom searches through the map. As things stand now, you'd have to reimplement your map from scratch, live with inefficient searches, or use the `Loki::AssocVector` paliativ. But whatever you do, please, please, always find informative names for your symbols.

References

[1] Andrei Alexandrescu. Generic⟨Programming⟩: A Policy-Based `basic_string` Implementation. *C++ Experts Online*, June 2001. Available at `http://erdani.org/publications/cuj-06-2001.html`.

[2] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley Longman, 2001.

[3] Andrei Alexandrescu. Generic⟨Programming⟩: Three μIdeas. *C++ Users Journal*, February 2005.

[4] Marcel Proust. *Remembrance of Things Past*. Penguin Classics, 1998. You can find the famous madeleines fragment at `http://www.haverford.edu/psych/ddavis/p109g/proust.html`.