`yasli::vector` Is On The Move

Andrei Alexandrescu

February 19, 2006

Last month I've been to the Software Development Conference in Santa Clara, CA. If anything I've seen there is illustrative for the general mood, C++ is in very good shape. The C++ track has been the strongest of the conference and the main culprit for the unexpectedly high attendance (in the 800s). Old and new projects that use C++ galore, and the programmer enthusiasm in advanced C++ techniques is on the rise. (Even *I* felt jaded in comparison!) Also, I got word that C++ programmers are the most sought after, that C++ consultancy requests have suddenly ramped up, and that C++ books are selling best. I'm not sure on why that is the case; proponents of other languages consider C++ and "efficient language" as opposed to other languages that are dubbed "productivity languages." Oh well.

Recently, there's been a surge of questions and discussions on the Usenet newsgroup `comp.lang.c++.moderated` about using `std::vector` effectively. It turns out that `std::vector`'s current interface and implementation need some enhancements if we are to use it as an effective replacement for the built-in array, and some more enhancements if we are to use it multidimensionally (vector of vectors). Such need for enhancements has also been motivated in past articles [2] and in Herb's book [8].

What `std::vector` cannot do well or at all, `yasli::vector` does like a champ. The legendary YASLI (Yet Another Standard Library Implementation) is by definition the best implementation of the C++ standard library around. YASLI uses advanced C++ implementation techniques, optimizations, and system and architectural assumptions, to achieve performance numbers that smoke any other implementation around, and in particular the one you might be currently using. Unfortunately—and here's where the "legendary" bit kicks in—YASLI, developed in Yasland, is not available to anyone, not even to yours truly; all we can do here on Earth is to write approximations of it. This column presents such an approximation of `yasli::vector`, explains the rationale and implementation of each artifact that distinguishes it from a state-of-the-art `std::vector` implementation.

1 Mailcontainer

Following my presentations at the aforementioned conference, there's been a lot of good feedback. First, many people liked `flex_string`[5] a whole lot. Given that the version published together with the original article had some bugs, I finally decided to publish it on my website (`http://moderncppdesign.com`). Click on "Code" and you can download the latest and greatest `flex_string` from there. I've also gotten favorable real-world usage data from Harmut Kaizer, who wrote me that in his C++ preprocessor [7], simply replacing `std::string` with `flex_string` boosted (no pun intended) speed by 5 to 10 percent, depending on input.

2 What *they* wouldn't want you to know about `std::vector`

Do you want to know what *they* don't want you to know about `std::vector`? Do you know why *they* are afraid that the truth will escape out one day for everybody to know?

I'll tell you. I'll tell you now, at the risk of being kidnapped and confined by *them* at a secure, top-secret location right as you are reading this.

The largest problem with `std::vector` is that it conflates two different concepts: data *moving* and data *copying*.

Here. I said it.

`std::vector` is overly generous with copying data around, when most of the time moving is the way to go. The untold assumption that the Standard spelling of `std::vector` makes is that copying vector elements around is not a major issue, and is not part of the asymptotic behavior (it's traditionally considered O(1)). That makes for nice numbers on paper, while the situation in the trenches might not be as bright as it could—and should. There are ways around that; they are not perfect, but they go a long way. Part of the solution is to define traits that give insights into `vector`'s element type, and also to use ideas from Mojo [6] which differentiates between copying and moving. This article sets out to discuss such issues.

Moving means, I have an object `obj` at a location `adr` and I want to move it to a different address `adr1`. Now, there are two variations, depending on the post-move situation: (1) the old location `adr` is considered junk and never read anymore, or (2) the old location `adr` is still considered to contain an "empty" object that might not hold the same state, but is still usable and will ultimately be destroyed elsewhere. Because yours truly is an equal-opportunity name-coiner, let's call the first version a "destructive" move, and the second one, well, just a "pure-and-simple" move.

First, let's motivate again the reasons (already mentioned in many places [4]) for which a move can often be less expensive than a copy. Oftentimes, a class just holds pointers or handles to expensive-to-duplicate resources, such as large chunks of memory or OS resources or whatnot. To C++, moving (reseating) is not a fundamental operation, and therefore `std::vector` typically implements it as a copy followed by destruction of the source. During the copying, resources might be duplicated; but that's not needed, because just in the next step, the just-duplicated resources will be destroyed. Obviously, it is more efficient to just pass the "baton" from the source to the destination.

So the central concept is to differentiate between copying and moving data. However, it's all in the details. What my previous articles [2, 3, 4] missed was that they were focusing on moving data using `memmove`. The key questions in implementing typed buffers was, what conditions does a type have to meet in order to be moveable by simply copying its bits by using `memmove(&obj, ptr1, sizeof(obj))`? In theory, nothing but Plain Old Data (POD) types can be moved with `memmove`. POD types are, roughly speaking, all the types that would make sense in C: fundamental types, arrays thereof, and simple `struct`s without embellishments such as virtuals or private data. In reality, it turns out that many types are moveable, but not all. Types that qualify don't have internal pointers (pointers that point to addresses inside the object—those would be totally messed if the object is simply `memmove`d). The most dangerous internal pointers are those that are also hidden—data added by the compiler (such as the pointers added by some compilers in the case of virtual inheritance). But then again, if we were to just follow the Standard, moving any fancy class via `memmove` is a no-no; the possibility of `memmove`-ing classes is based on arguments like "yeah, it's illegal, but the compiler would be really crazy to cause any trouble."

But focusing on whether or not a type can be moved with `memmove` was simply wrong. What the Yaslanders understood very well was that the real focus is not to decide on `memmove`ing, but rather to define and follow a *protocol* between `vector<T>` and its hosted type `T`. Then, YASLI defines a default conservative implementation of the protocol. Any type that wants to achieve efficiency with vector only needs to override the default protocol implementation with the one of choice. In particular, that implementation could use `memmove`. If `memmove` doesn't work with a particular class, there still are other efficient ways (as we'll see below). It's that simple.

3 yasli::vector's protocol

So, let's do a bit of analysis: we need to define a protocol for moving objects that `yasli::vector` will use. The protocol should have the following properties:

1. *Genericity.* We need the protocol to work for any type that `yasli::vector` can ever be instantiated with.

2. *Flexibility.* We'd like the protocol to be customizable so that we can implement it in various ways for types with various properties.

3. *Reasonable defaults.* This is an important point. The protocol must implement a conservative, reasonable default that works for any type. You don't want to require anyone using `yasli::vector` to define some special functionality.

4. *Robust.* That means the protocol should be easy to implement and use correctly, and hard to implement and use incorrectly. Robustness is somewhat in tension with reasonable defaults; if you implement the protocol incorrectly, there is a risk that the default protocol will be executed for your type without you realizing it.

Let's now brainstorm some designs that would meet the requirements above. The simplest approach would be to arrange things such that `yasli::vector` calls a template function:

```
// yasli_protocols holds all of
// user-overridable portions of YASLI
namespace yasli_protocols
{
  template <class T>
  T* destructive_move(
      T* begin,
      T* end,
      void* dest);
}
```

The charter of `destructive_move` is to move the data range [`begin`, `end`) to a chunk of memory pointed to by `dest`. Then it returns `dest`, cast to `T*` so as to reveal the new type of the destination. After the operation has completed, the source range is considered to be bits, that it, it has lost type—hence the "destructive" particle in the function name. The default implementation of `destructive_move` would do the conservative thing: copies the data element by element, and then it destroys each element in the source. Then, `vector` would use `destructive_move` religiously whenever it comes about moving data around. The users of `vector` can define their own overloads of `destructive_move` to implement efficient move for their own types.

Unfortunately, this turns out to be a bad design that fails badly on the flexibility and robustness test. In fact, it is the route taken by `std::swap`, with disastruous results. The short version of an explanation is that relying on function overloading is the wrong way to go due to the way name lookup works; I've erased all traces of that trauma from my memory, but you are welcome to read the details [1].

A nicer and more tame way to do things is to rely on a template class that defines a number of functions. Template classes obey simpler rules with regard to name lookup and instantiation. Here's how the declaration of such a protocol would look like:

```
namespace yasli_protocols
{
  // that's more like it
  template <class T>
  struct move_traits
  {
    static T* destructive_move(
        T* begin,
        T* end,
        void* dest);
  };
}
```

Now `yasli::vector<T>` will use `move_traits<T>::destructive_move` whenever it comes about moving data around. Any type that wants can simply specialize `move_traits<T>::destructive_move` to perform the move in some efficient way. The first candidates that come to mind would be primitive

types such as integers or pointers, and here's how the specializations would look like:

```
namespace yasli_protocols
{
  template <>
  struct move_traits<int>
  {
    static int* destructive_move(
        int* b,
        int* e,
        void* d)
    {
      return (int*)memmove(d,
        b,
        (e - b) * sizeof(int));
    }
  };
  template <class T>
  struct move_traits<T*>
  {
    static T** destructive_move(
        T** b,
        T** e,
        void* d)
    {
      return (T**)memmove(
        d,
        b,
        (e - b) * sizeof(T*));
    }
  };
}
```

The first specialization is a total specialization applying to `int` alone, while the second one is a partial specialization that applies to all pointer types. Nothing special (no pun intended). But there are two less-than-perfect things about this design. One is that there's some duplication creeping up. But hey, the real code generated is really one line, and if worst comes to worst, you know that yours truly won't be coy about defining a, um, macro. But the more disturbing problem is a feel of a brute-force approach. We want to implement `move_traits` in a certain way for types that have a specific property (are moveable by using `memmove`, property that all primitive types possess). We end up achieving that by implementing `move_traits` for *each* type sporting that property, and that doesn't bode well. Later, maybe we'll want to implement `move_traits` in a special way for all types that implement the Mojo protocol [6], and that set is simply not enumerable! In other words, this design is not very good at fulfilling the flexibility requirement.

Looks like relying on overloading or template specialization both have shortcomings. Fortunately, although we don't have full access to the Yaslander technology, they do have full access to ours. In particular, they've read an article of important impact, fatefully entitled "Function overloading based on arbitrary properties of types" [**?**]. That article (highly recommended) presents a simple template `enable_if` that allows one to control when a specific overload kicks in. This concept is so interesting, and applies to our needs of defining a protocol so well, it has a short section of its own (at the cost of plagiarizing)—but do read the mentioned article for the full story.

4 Controlling Overloading

Several C++ luminaries arrived independently at the same interesting conclusion: when resolving function overloading in the presence of templates, the compiler might try and silently abandon a lot of "dead ends." For example:

```
void transmogrify(unsigned int) { ... }
template <class T>
typename T::result
transmogify(T) { ... }
...
transmogrify(5);
```

When seeing such code, the compiler gives the second overload of `transmogrify` a chance, by attempting to instantiate it with `int`. It doesn't take a strike of genius, though, to realize that `int::result` is not a type. The interesting bit is that the compiler does not issue a compile-time error at this point, but instead it just removes the function from the candidates set. This phenomenon, coined as the Latin-sounding SFINAE (Substitution Failure Is Not An Error) [9],

led to a simple technique of controlling overloading. First, we define a template class `enable_if` as shown below:

```
template <bool B, class T = void>
struct enable_if
{
  typedef T type;
};
template <class T>
struct enable_if<false, T> {};
```

So, `enable_if` defines `type` if the compile-time Boolean value passed is true, or it doesn't if the Boolean value passed is **`false`**. Now all you have to do is to use `enable_if<expression, T>::type` instead of `T` somewhere in the function signature, and voilà! The Boolean now controls whether the function is ever considered or not.

The applicability of this idea to `move_traits` is hinted to in the "Future work" part of the mentioned article, which says:

> Matching partial specializations of class templates is performed with the same set of rules as function template argument deduction. This means that a partial specialization of a class template can only match if template argument deduction does not fail, and thus the set of rules we describe in the background section can be exploited to enable and disable class template specializations. [...] all that is needed is one extra template parameter with the default value **`void`**. In specializations, this extra parameter is a condition wrapped inside an enabler template.

5 A Protocol Design

Using `enable_if` turns out to be a very good strategy for `move_traits`. This is how the protocol is declared:

```
template <class T, class U=void>
struct move_traits
{
  static T* destructive_move(
      T* begin,
      T* end,
      void* dest);
};
```

Now say we want to specialize for all primitive types, which we can do in a single shot!

```
template <class T>
struct move_traits<T,
  typename enable_if
    <!is_class<T>::value>::type>
{
  static T* destructive_move(
      T* begin,
      T* end,
      void* dest)
  {
    return (T**)memmove(
      d,
      b,
      (e - b) * sizeof(T*));
  }
};
```

Unless you're familiar with Boost's type traits, there's a bit more than a mouthful in the snippet of code above—but nothing you can't handle. The expression `is_class<T>::value` is a compile-time Boolean constant that evaluates to **`true`** if `T` is a class, that is, not a built-in type. Conversely, `!is_class<T>::value` is **`true`** for all built-in types. When you, say, mention `move_traits<float>`, the compiler will attempt to instantiate the more specialized version of `move_traits`. That instantiation succeeds, because `typename enable_if<!is_class<float>::value>::type` evaluates to **`void`**, and as such takes over.

Next, we can easily customize `move_traits` for all classes that implement the Mojo protocol [6]:

```
template <class T>
struct move_traits<T,
  typename enable_if
    <is_base_and_derived<mojo::enabled,
      T>::value>::type>
{
```

```
   ...
};
```

We also can specialize `move_traits` on its first argument, say by matching `std::complex`. Most of the time, `std::complex<T>` can be moved using `memmove`. The only condition is that the compiler doesn't insert some funky additional data with a `complex` object (which no compiler I know of does), condition that can be easily checked with a **`sizeof`** test:

```
template <class T>
struct move_traits<std::complex<T>,
  typename enable_if
    <sizeof(std::complex<T>) ==
      2 * sizeof(T)>
        ::type>
{
   ... use memmove ...
};
```

To conclude this section, defining a protocol as a two-parameters template class as shown above and using `enable_if` to guide specialization scores good on all of our requirements.

6 Initialization

Here's another common request made by `std::vector`'s users. Many programmers want to create an *uninitialized* vector because, for example, they need to fill it with a C-style API function. There is no way of doing that with `std::vector`, and allowing it is tricky because it would create a type-unsafe hole inside `std::vector`. What to do?

The approach of `yasli::vector` is particularly elegant in that it offers initialization functions putting both the responsibility and the accountability in its user's hands. Consider, for example, this constructor:

```
// ... inside yasli::vector ...
template <class F>
void resize_nstd(size_type newSize, F functor);
```

This nonstandard resizing function resizes the vector. If the vector grows, `resize_nstd` doesn't initialize the newly added elements; instead, it calls `F(range_begin, range_end)`, where `range_begin` and `range_end` are pointers pointing to the newly-added range of elements. If you want to leave them uninitialized, write a do-nothing function and pass it in. There's no better way out of the "who's responsible and who's accountable" conundrum.

7 Guarded Assumptions

If you look around `yasli::vector`'s source code, you'll notice a funny-looking piece of code that is called whenever an empty object of type `yasli::vector` is being initialized:

```
// ...inside yasli::vector...
    void init_empty()
    {
#if YASLI_UNDEFINED_POINTERS_COPYABLE == 1
        end_ = beg_;
        eos_ = beg_;
#else
        beg_ = 0;
        end_ = 0;
        eos_ = 0;
#endif
    }
```

The three members `beg_`, `end_`, and `eos_` are the classic "begin, end, and end-of-storage" pointers that all implementations of `std::vector` hold as members. The interesting bit is the true branch of the **`#if`**-guarded code. If `YASLI_UNDEFINED_POINTERS_COPYABLE` is `1`, then the vector is initialized by copying the *uninitialized* pointer `beg_` to the other two pointers! How can that be even close to correctness?

The empty state of a vector is that its begin, end, and end-of-storage pointers have the same value. However, the actual value of the three pointers is irrelevant! The test for emptiness spells `beg_ == end_`, not `beg_ == 0`. Traditionally, the three pointers are initialized with the singular value `NULL`. But on some architectures (such as Intel's Pentium family), you can copy and compare for (in)equality any uninitialized pointers, as long as you don't dereference them. Some other processors don't

allow such manipulation of uninitialized pointers. On the former processors, you can get away with initializing a vector with only two assignments instead of three—a nice 30% speed improvement.

Exploiting uninitialized pointer manipulation when `YASLI_UNDEFINED_POINTERS_COPYABLE` is `1` is a good example of an important piece of Yaslander technology: *guarded assumptions*. A guarded assumption is a system-dependent assumption about the way a hardware-compiler-runtime combo works. The "guard" is a preprocessor definition; if that definition is absent, then YASLI implements conservative standard behavior. If the definition is present, then YASLI is free to exploit that assumption to achieve better execution speed and/or pack data better, with code that invokes undefined behavior according the the C++ standard. Other examples of guarded assumptions—not necessarily present in `vector` per se, are:

- `YASLI_REALLOC_AFTER_NEW`. When this preprocessor symbol is defined to be `1`, then YASLI will assume that it is safe to call `realloc` on a pointer obtained with the global `operator new`.
- `YASLI_STANDARD_OBJECT_LAYOUT`. This means that field are laid out in an "unsurprinsing" manner—there are no hidden fields and the declaration order is respected.
- `YASLI_FLOATS_BITWISE_ZERO`. This means that a floating point value filled with zero bitwise, will also evaluate to `0`.

8 `yasli::vector`—Standard or not?

So the important question that arises is, is `yasli::vector` compliant with the C++ standard's prescription? The answer is either a qualified yes or a qualified no—depending, of course, on what you qualify the answer with.

First, if you grep `yasli::vector`'s header file for "nstd," you will find a number of functions and methods that are, well, not standard, but that ought to be if `std::vector`'s interface would allow for a maximally efficient implementation. Such an example is `yasli::vector<T>::move_back_nstd(T&)`, function that appends an element to the vector by *moving* the content of the passed-in object—as opposed to copying it.

Second, as said, `yasli::vector` commits to using the `yasli_protocol::move_traits` protocol. That, by default, ensures standard-compliant behavior. If you provide your own implementation of the protocol, of course `yasli::vector` will use that one and therefore deviate from the letter of the standard.

So we could say, `yasli::vector` provides some additional functionality that is nonstandard. However, in doing that, it requires for the cooperation of the user, either in the form of providing a custom implementation of `yasli_protocol::move_traits`, or in the form of compulsively calling functions that have "nstd" spelled as part of their name. That makes it easy for you to make your choices.

9 Conclusions

This article presented an enriched interface and a performant implementation for `std::vector`. The existing `std::vector` is of good quality, but fails to address a number of important points, such as memory friendliness, distinction between moving and copying, and resizing without initialization. The actual code, as stolen from the Yaslanders, is available from `http://moderncppdesign.com/code`, and your comments on it are welcome.

10 Acknowledgments

In addition to independently having stolen much knowledge from the Yaslanders and putting it in CodeWarrior 9's `std::vector` implementation, Howard Hinnant provided excellent input for the article that, hopefully, you enjoyed.

References

[1] ISO/IEC IS 14882:1998(E). Issue 226 in C++ Standard Library Active Issues List (Revision 28). Available at `http://gcc.gnu.org/onlinedocs/libstdc++/ext/lwg-active.html#226`.

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

[3] Andrei Alexandrescu. Generic⟨Programming⟩: Typed Buffers (II). *C++ Experts Online*, October 2001. Available at `http://moderncppdesign.com/publications/cuj-10-2001.html`.

[4] Andrei Alexandrescu. Generic⟨Programming⟩: Typed Buffers (III). *C++ Experts Online*, December 2001. Available at `http://moderncppdesign.com/publications/cuj-12-2001.html`.

[5] Andrei Alexandrescu. Generic⟨Programming⟩: A policy-Based `std::string` implementation. *C++ Experts Online*, August 2003. Available at `http://www.cuj.com/documents/s=7992/cujcexp1908alexandr/alexandr.htm`.

[6] Andrei Alexandrescu. Generic⟨Programming⟩: Move constructors. *C++ Experts Online*, February 2003. Available at `http://moderncppdesign.com/publications/cuj-02-2003.html`.

[7] Harmut Kaizer. The boost preprocessor. Available from `http://boost.org`.

[8] Herb Sutter. *Exceptional C++*. Addison-Wesley Longman, 2000.

[9] Daveed Vandevoorde and Nicolai Josuttis. *C++ Templates*. Addison-Wesley Longman, 2003.