`auto_value:` Transfer Semantics for Value Types

Richard Harris

The problem of eliminating unnecessary copies is one that many programmers have addressed at one time or another. These notes propose an alternative to one of the most common techniques, copy-on-write. We'll begin with a discussion on smart pointers, and why we need more than one type of them. We'll then look at the relationship between smart pointers and the performance characteristics of some simple string implementations, including one that supports copy-on-write. I'll suggest that a different choice of smart pointer better captures our intent, and show that what we were trying to do doesn't achieve half as much as we thought it would.

Finally, I hope to show that whilst the problem we set out to solve turns out to be a bit of a non-issue, this technique has a side effect that can be exploited to dramatically improve the performance of some complex operations.

`notes::notes()`

I'd like to begin by recalling Jackson's Rules of Optimization.

> Rule 1: Don't do it.
> Rule 2 (for experts only): Don't do it yet.

This is probably a little presumptuous of me, but I'd like to add something:

> Harris's Addendum: Nyah, nyah. I can't hear you.

I'll admit it's not very mature, but I think it accurately reflects how we all *really* feel. No matter how much a programmer preaches, like Knuth, that premature optimisation is the root of all evil, I firmly believe that deep down they cannot help but recoil at inefficient code.
Don't believe me? Well, ask yourself which of the following function signatures you'd favour:

```
    void f(std::vector<std::string> strings);
    void f(const std::vector<std::string> &strings);
```

Thought so.
But you mustn't feel bad about it, the desire to write optimal code is a *good* thing. And I can prove it.
In their seminal 2004 paper, Universal Limits on Computation, Krauss and Starkman demonstrated that the universe will only be able to process another 1.35×10^{120} bits during its lifetime. Count them. Just 1.35×10^{120}. If Moore's Law continues to hold true, we'll run out of bits in 600 years.
So we can stop feeling guilty about our oft-criticised drive to optimise, because wasted CPU cycles are accelerating the heat death of the universe.
Will nobody think of the children?
Act responsibly.
Optimise.

OK, that's a little disingenuous. Knuth actually said that in 97% of cases we should forget about small efficiencies. I suspect that we'd all agree that using const references by default for value types generally falls into the 3% that we shouldn't ignore. There is, after all, a world of difference between choosing the more efficient of two comparably complex statements and significantly increasing the complexity of your code in the name of a relatively small efficiency gain.
There is a grey area though. Sometimes it really is worth increasing the complexity of your code for relatively small gains. Especially when the particular source of inefficiency occurs regularly within your code base and you can hide that complexity behind a nice tightly defined class interface.

These notes are going to take a look at those most profligate of wastrels, temporaries.
But since the direct route rarely has the most interesting views, we're going to set off with a discussion on smart pointers.

auto_ptr

Let's start by taking a look at the definition of `auto_ptr`

```
template<typename X>
class auto_ptr
{
public:
  typedef X element_type;

  explicit auto_ptr(X *p = 0) throw();
  auto_ptr(auto_ptr &p) throw();
  template<class Y> auto_ptr(auto_ptr<Y> &p) throw();
  auto_ptr(auto_ptr_ref<X> p) throw();
  ~auto_ptr() throw();

  auto_ptr & operator=(auto_ptr &p) throw();
  template<class Y> auto_ptr & operator=(auto_ptr<Y> &p) throw();
  auto_ptr & operator=(auto_ptr_ref<X> p) throw();

  template<class Y> operator auto_ptr_ref<Y>() throw();
  template<class Y> operator auto_ptr<Y>() throw();

  X & operator*() const throw();
  X * operator->() const throw();
  X * get() const throw();

  X * release() throw();
  void reset(X *p = 0) throw();

private:
  X *x_;
};
```

It's a little bit more complicated than you'd expect isn't it?
This is because, like HAL, it has been driven ever so slightly barking mad from being given two competing responsibilities. The first of these is to tie the lifetime of an object to the scope in which it's used.
For example:

```
void
f()
{
  const auto_ptr<T> t(new T);
  //...
} //object is destroyed here
```

The destructor of the `auto_ptr` deletes the object it references, ensuring that it is properly destroyed no matter how we leave the scope.
The second responsibility is to safely transfer objects from one location to another.
For example:

```
auto_ptr<T>
f()
{
  return auto_ptr<T>(new T);
}

void
g(auto_ptr<T> t)
{
  //...
} //object is destroyed here

void
h()
{
  auto_ptr<T> t;
  t = f(); //ownership is transferred from f here
  g(t);    //ownership is transferred to g here
}
```

The two roles are distinguished by the constness of the `auto_ptr` interface. The const member functions of `auto_ptr` manage lifetime control, whereas the non-const member functions manage ownership transfer.
For example, by making the variable `t` in the function `h` in the previous example const, we can ensure that the compiler will tell us if we accidentally try to transfer its ownership elsewhere:

```
void
h()
{
  const auto_ptr<T> t;
  t = f(); //oops
  g(t);    //oops
}
```

And herein lies the problem. We want const `auto_ptrs` to jealously guard their contents, so we make the arguments to the transfer constructor and assignment operator *non-const* references. But, unnamed temporaries, such as function return values, can only be bound to *const* references, making it difficult to transfer ownership of an object out of one function and into another.
For example:

```
void
h()
{
  g(f()); //now I'm confused
}
```

This is where the mysterious `auto_ptr_ref` class comes to our rescue. You'll note that `auto_ptr` has a non-const conversion to `auto_ptr_ref` and that there's a conversion constructor that takes an `auto_ptr_ref`. So a non-const unnamed temporary can be converted to an `auto_ptr_ref`, which will in turn transfer ownership to an `auto_ptr` via the conversion constructor.
Neat, huh?

Well, perhaps. But we could almost certainly do better by giving each of `auto_ptr's` personalities its own body. We'll do this by introducing a new type, `scoped_ptr`, to manage object lifetimes and stripping those responsibilities from `auto_ptr`.
The definition of `scoped_ptr` is as follows:

```
template<typename X>
class scoped_ptr
{
public:
  typedef X element_type;

  explicit scoped_ptr(X *p = 0) throw();
  explicit scoped_ptr(const auto_ptr<X> &p) throw();
  ~scoped_ptr() throw();

  X & operator*() const throw();
  X * operator->() const throw();
  X * get() const throw();

  auto_ptr<X> release() throw();

private:
  scoped_ptr(const scoped_ptr &);             //not implemented
  scoped_ptr & operator=(const scoped_ptr &); //not implemented

  X * x_;
};
```

Those in the know will see the similarity with the boost `scoped_ptr` (www.boost.org). This is only natural since I pretty much just swiped it from there.
As with the original `auto_ptr`, the constructors take ownership of the objects passed to them and the destructor destroys them. The principal change is that it is no longer legal to copy or assign to `scoped_ptrs` (the unimplemented private copy constructor and assignment operator are there to suppress the automatically generated ones).
We can continue to use `scoped_ptr` to tie object lifetime to scope:

```
void
f()
{
  scoped_ptr<T> t(new T);
  //...
} //object is destroyed here
```

But we can no longer use it to transfer ownership:

```
scoped_ptr<T> //oops, no copy constructor
f()
{
  return scoped_ptr<T>(new T);
}

void
g(scoped_ptr<T> t) //oops, no copy constructor
{
  //...
}

void
h()
{
  scoped_ptr<T> t;
  t = scoped_ptr<T>(new T); //oops, no assignment operator
}
```

Now let’s have a look at how giving up responsibility for lifetime control changes `auto_ptr`:

```
template<typename X>
class auto_ptr
{
public:
  typedef X element_type;

  explicit auto_ptr(X *p = 0) throw();
  auto_ptr(const auto_ptr &p) throw();
  template<class Y> auto_ptr(const auto_ptr<Y> &p) throw();
  ~auto_ptr() throw();

  const auto_ptr & operator=(const auto_ptr &p) const throw();
  template<class Y>
    const auto_ptr & operator=(const auto_ptr<Y> &p) const throw();

  X & operator*() const throw();
  X * operator->() const throw();

  X * release() const throw();

private:
  mutable X *x_;
};
```

OK, so I lied a little.
We haven't so much lost the ability to control object lifetimes with `auto_ptr` as made it a little less attractive. The constructors still take ownership of the objects passed to them and the destructor still destroys them, but holding on to them is difficult.
This is because the object pointer is now *mutable*, allowing it to be changed even through const member functions. Usually mutable is reserved for members that can change whilst the object maintains the appearance of constness (caches, for example), an idea typically described as logical constness. Here, to be honest, it's a bit of a hack. We need to tell the compiler to abandon all notions of constness for `auto_ptrs` and unfortunately we can't do that (unnamed temporaries rear their problematic heads again). So we lie to the compiler. We tell it that "no really, this function is const" and use mutability to change the object pointer anyway.

We can still use `auto_ptr` to transfer object ownership from one place to another:

```
auto_ptr<T>
f()
{
  return auto_ptr<T>(new T);
}

void
g(auto_ptr<T> t)
{
  //...
} //object is destroyed here

void
h()
{
  auto_ptr<T> t;
  t = f(); //ownership is transferred from f here
  g(t);    //ownership is transferred to g here
}
```

But we might run into problems if we try to use it to control object lifetime:

```
T
h()
{
  const auto_ptr<T> t;
  g(t);      //ownership is transferred to g here
  return *t; //oops
}
```

It’s precisely because this new `auto_ptr` is so slippery, that I’ve added a release method to boost’s `scoped_ptr`. This enables us to use the `scoped_ptr` to control the lifetime of the object within a function and `auto_ptr` to control its transfer during function return.
For example:

```
auto_ptr<T>
f()
{
  scoped_ptr<T> t;
  //...
  return t.release();
}

void
g()
{
  scoped_ptr<T> t(f());
}
```

Henceforth, when we refer to `auto_ptr`, we will mean this new version, having the sole responsibility of ownership transfer.

shared_ptr

Another approach to managing object lifetimes is to allow multiple references to share ownership of an object. This is achieved by keeping the object alive for as long as something is referring to it.
There are two common techniques used to do this, the correct approach and the easy approach. Guess which one we’re going to look at.
That’s right. Reference counting.

Reference counting works by keeping a count of the number of active references to an object and deleting it once this count drops to zero. Each time a new reference is taken, the count is incremented and each time a reference is dropped, the count is decremented. This is generally achieved by requiring the referencing entity to explicitly register its interest or disinterest in the object.
The chief advantage of using reference counting to implement shared ownership semantics is that it’s relatively simple compared to the alternative.
The chief disadvantage occurs when an object directly or indirectly holds a reference to itself, such as when two objects hold references to each other. In this situation, the reference count will not fall to zero unless one of the objects explicitly drops the reference to the other. In practice, it can be extremely difficult to manually remove mutual references since the ownership relationships can be arbitrarily complex. If you would like to bring a little joy into someone’s life, ask a Java programmer about it.

We can automate much of the book-keeping required for reference counting by creating a class to manage the process for us. In another shameless display of plagiarism I’m going to call this class `shared_ptr`:

```
template<typename X>
class shared_ptr
{
public:
  typedef X element_type;

  shared_ptr() throw();
  template<class Y> explicit shared_ptr(Y * p);
  shared_ptr(const shared_ptr &p) throw();
  template<class Y> shared_ptr(const shared_ptr<Y> &p) throw();
  template<class Y> explicit shared_ptr(const auto_ptr<Y> &p);
  ~shared_ptr() throw();

  shared_ptr & operator=(const shared_ptr &p) throw();
  template<class Y>
    shared_ptr & operator=(const shared_ptr<Y> &p) throw();
  template<class Y>
    shared_ptr & operator=(const auto_ptr<Y> &p) throw();

  X & operator*() const throw();
  X * operator->() const throw();
  X * get() const throw();

  void reset(X *p = 0) throw();
  bool unique() const throw();
  long use_count() const throw();

private:
  X *x_;
  size_t *refs_;
};
```

The reference count is pointed to by the member variable `refs_` and is incremented whenever a `shared_ptr` is copied (either through assignment or construction) and decremented whenever a `shared_ptr` is redirected (either through assignment or reset) or destroyed.
For example:

```
void
f()
{
  shared_ptr<T> t(new T); //*refs_==1

  {
    shared_ptr<T> u(t); //*refs_==2
    //...
  } //*refs_==1

  //...
} //*refs_==0, object is destroyed here
```

Reference counting blurs the distinction between object lifetime control and transfer by allowing many entities to simultaneously “own” an object.

```
shared_ptr<T>
f()
{
  return shared_ptr<T>(new T);
}

void
g(shared_ptr<T> t) //++*refs_
{
  //...
} //--*refs_

void
h()
{
  shared_ptr<T> t;
  t = f(); //ownership is transferred from f here
  g(t);    //ownership is shared with g here
}
```

The sequence of events in the above example runs as follows:

```
call h
shared_ptr()

call f
new T
shared_ptr(T *)                          //*refs_=1
shared_ptr(const shared_ptr &)           //++*refs_
~shared_ptr                              //--*refs_
exit f

shared_ptr::operator=(const shared_ptr &) //++*refs_
~shared_ptr                              //--*refs_

call g
shared_ptr(const shared_ptr &)           //++*refs_
~shared_ptr                              //--*refs_
exit g

exit h
~shared_ptr                              //--*refs_
```

As you can see, at the end of `h`, the reference count is zero and the object is consequently destroyed.
Since the object has more than one owner we must exercise caution when using it or, more specifically, when changing its state.
For example:

```
void
f(shared_ptr<T> t)
{
  //...
}

void
g()
{
  shared_ptr<T> t(new T);
  shared_ptr<const T> u(t);
  f(t);    //ownership is shared with f here
  //state of u is uncertain here
}
```

Of course, this is just pointer aliasing in a spiffy new suit and as such shouldn't come as much of a surprise. I mean, *nobody* makes that mistake these days, do they?
Well, almost nobody.
Well, certainly not standard library vendors.
Well, probably not standard library vendors.
Well, probably not very often.

string

The last time I saw this problem was in a vendor supplied `std::string`. Well, not this problem exactly, but it was related to incorrect use of reference counted objects. I shan't name names, but it was a company you all know and many of you respect. When I finally tracked down what was causing my program to crash I was stunned.
Now you may be wondering how these sorts of problems could possibly relate to `string`, after all it's a value type not an object type. The reason is that this particular `string` used a common optimisation technique known as copy-on-write, or COW for short, and this technique relies upon reference counting.

Before we describe how COW works, let's take a look at a naïve implementation of a string class:

```
class string
{
public:
  typedef char         value_type;
  typedef char *       iterator;
  typedef char const * const_iterator;
  typedef size_t       size_type;
  //...

  string();
  string(const char *s);
  string(const string &s);

  string & operator=(const string &s);
  string & operator=(const char *s);

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...

private:
  size_type size_;
  scoped_array<char> data_;
};
```

A `scoped_array` (identical to `scoped_ptr`, except that it uses `delete[]` instead of `delete`) ensures that the data is deleted when a string goes out of scope.

The constructors are pretty straightforward:

```
string::string() : size_(0), data_(0)
{
}

string::string(const char *s) : size_(strlen(s)),
                                data_(size)
{
  std::copy(s, s_size_, data_.get());
}

string::string(const string &s) : size_(s.size_),
                                  data_(new char[size_])
{
  std::copy(s.data_.get(), s.data_.get()+size_, data_.get());
}
```

The copy and conversion constructors allocate new strings whose lifetimes are managed by the `scoped_array` member `data_`. The constructor bodies then simply copy the strings.
Nothing particularly surprising.

The assignment operators are a little more complex, though. These days the recommended way to implement assignment is with the copy and swap idiom.
This is usually expressed as:

```
T &
T::operator=(const T &t)
{
  T tmp(t);
  swap(tmp);
  return *this;
}
```

The chief advantage of this approach (besides its relative simplicity) is that it is guaranteed to leave the object in its original state if an exception is thrown during the copy operation. This is because we don't commit the change until we swap the object with the temporary copy and the call to `swap` is guaranteed not to throw.
In the case of `string`, the `swap` member function could be defined as:

```
void
string::swap(string &s)
{
  std::swap(size_, s.size_);
  std::swap(data_, s.data_);
}
```

Unfortunately `std::swap` will need to assign a new value to both `data_` and `s.data_` and since `scoped_array` doesn't allow us to rebind the pointer, it doesn't provide an assignment operator or `reset` member function. If we want `scoped_array` to consider classes as scopes, we'll need to provide some mechanism to enable assignment.
Adding assignment operators to `scoped_array` would dramatically weaken its guarantee that it will only ever point to one array.

Thankfully, we can achieve what we want by adding a `swap` member function instead, defined as follows:

```
void
scoped_array::swap(scoped_array &a)
{
  std::swap(x_, a.x_);
}
```

This still weakens `scoped_array`'s guarantee, but in a way that's less appealing to use for ownership transfer than an assignment operator or `reset` member function. Well, that's my story and I'm sticking to it.
We can now redefine `string`'s `swap` member:

```
void
string::swap(string &s)
{
  std::swap(size_, s.size_);
  data_.swap(s.data_);
}
```

Now, the naïve approach is perfectly good for short strings, but those string copies in the constructors, and indirectly in the assignment operator, start to look pretty ominous in the face of very long strings. For example, consider the sequence of events when storing the result of a function call:

```
string
f()
{
  string s("hello, world");
  return s;
}

void
g()
{
  string s = f();
}
```

So what exactly happens when we call `g`?

```
call g

call f
copy “hello, world” into s
copy s into return temporary
exit f

copy return temporary into s
exit g
```

Yikes. I count two completely unnecessary copies of the data. Now that’s not such a problem for “hello, world”, but will hit hard for the King James Bible, for example.

So how does copy-on-write help?
Well, it seeks to eliminate unnecessary copies by deferring them until they can no longer be avoided. This is typically when an element of the string is about to be changed, hence the name. Until that moment, a “copy” of a string simply holds a reference to the original.
This is related to, but subtly different from, reference counting for shared ownership. Unlike shared ownership, COW allows multiple references to the underlying data of an object only so long as the external behaviour is unaffected.
With shared ownership, we expect multiple references to be able to change the observed state of the object. With COW, this must be avoided at all costs. We use reference counting merely to avoid making multiple copies of provably identical data.
For example:

```
void
f()
{
  string s1(“hello, world”);
  string s2(s1); //s2 can hold a reference to s1’s data here
  std ::cout << s2 << std::endl;
  s2.replace(0, 5, “goodbye”); //s2 must copy s1’s data here
}
```

Until we actually change the state of `s2` in the last line of `f`, the behaviour of the function is unchanged whether `s1` and `s2` share their state or not.

Let’s have a look at how we might implement a string class that supports COW:

```
class string
{
public:
  typedef char         value_type;
  typedef char *       iterator;
  typedef char const * const_iterator;
  typedef size_t       size_type;
  //...

  string();
  string(const char *s);
  string(const string &s);

  string & operator=(const string &s);
  string & operator=(const char *s);

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...

private:
  size_type size_;
  shared_array<char> data_;
};
```

The point to note about this definition is that the data is stored in a `shared_array` (like `shared_ptr`, except that it uses `delete[]`) rather than as a `scoped_ptr`. With this we will share, rather than copy, data for as long as possible.
To see how this works, let's take a closer look at a few member functions.
First, the constructors:

```
string::string() : size_(0), data_(0)
{
}

string::string(const char *s) : size_( strlen(s)),
                                data_(new char[size_])
{
  std::copy(s, s+size_, data_.get());
}

string::string(const string &s) : size_(s.size_), data_(s.data_)
{
}
```

The sharp-eyed amongst you will have noticed that the copy constructor is superfluous since the compiler generated version would have done exactly the same thing.
The important point though is that when we construct a `string` with a C-style point to character array, the data is copied, whereas when we copy construct a `string`, the data is shared.

Now let's have a look at the two flavours of `begin` to see how we make sure that we don't accidentally change a `string's` value through this shared data:

```
string::const_iterator
string::begin() const
{
  return data_.get();
}
```

Well, the first version is pretty simple. Since we're returning a `const_iterator` (defined as a pointer to const `T`), it's not possible to change the contents of the string so we can simply return the pointer to the start of the character data.
Granted, `const_cast` could throw a spanner in the works, but I doubt anyone would be too upset if we simply declared casting between iterator types undefined behaviour and ignored the problem.
Clearly, it's the non-const version of the function that is of interest:

```
string::iterator
string::begin()
{
  if(data_.get() && !data_.unique())
  {
    char *s = data_.get();
    data_.reset(new char[size_]);
    std::copy(s, s+size_, data_.get());
  }

  return data_.get();
}
```

And here we see the mechanism at work. If more than one `string` is referring to the data, we copy it before we allow a means to change it to escape. The same check for uniqueness and subsequent copy must be present in every function that presents an opportunity to change the string.

In the following example:

```
void
f()
{
  const string s1(“hello, world”);
  string s2(s1);
  string::iterator i = s2.begin();
  *i = ‘y’;
}
```

we have the following sequence of events:

```
construct s1
copy “hello, world”

construct s2
share “hello, world”

s2.begin
fail uniqueness check
copy data
return iterator

assign ‘y’ to start of s2
```

Given that we have already established the sharpness of your eyes, I have no doubt that you have raced ahead of me and spotted that this code isn’t remotely fit for purpose.

To hammer home why, consider the following example:

```
void
f()
{
  string s2(“hello, world”);
  string::iterator i = s2.begin();
  const string s1(s2);
  *i = ‘y’; //oops, s1 has changed too
}
```

Let’s take a look at the sequence of events this time:

```
construct s2
copy “hello, world”

s2.begin
pass uniqueness check
return iterator

construct s1
share “hello, world”

assign ‘y’ to start of s1 and s2
```

It seems there was a potential alias that we overlooked, the iterator itself.
This is harder than I expected. Perhaps I should be more forgiving of my unnamed library vendor.
Worse still, even if we correctly identify and protect all possible aliases, we still have made a rather sweeping assumption. Namely, that `strings` will only be referred to by a single thread.

In a multi-threaded program, it is quite possible that a COW string’s data could be manipulated by more than one thread at the same time. To illustrate the problems that this can lead to, consider what happens when two copies are made of a string.

```
void
f(const string &s)
{
  string s1(s);
}

void
g(const string &s)
{
  string g1(s);
}
```

In a single threaded program, `f` and `g` must be called sequentially:

```
string s(“hello, world”);
f(s);
g(s);
```

leading to the following sequence of events:

```
construct s
copy “hello, world”

call f
share “hello, world”
release “hello, world”
exit f

call g
share “hello, world”
release “hello, world”
exit g
```

and everything goes smoothly.

For multi-threaded programs, the problem is that sharing and releasing the string data are not atomic operations. Specifically, they must read the reference count, manipulate it and finally write it. Unfortunately a thread may be interrupted during these steps.
For example, if we were to call `f` and `g` on separate threads:

```
string s(“hello, world”);
run_threaded(f, s);
run_threaded(g, s);
```

we might observe the following sequence of events:

```
construct s
copy “hello, world”

call f
call g

f: read reference count      (count == 1)
f: increment reference count (count == 2)
g: read reference count      (count == 1)
f: write reference count     (count == 2)
g: increment reference count (count == 2)
g: write reference count     (count == 2)
oops
```

Because `g` read the reference count before `f` had committed its change, we’ve lost the record of `f`’s interest in the string. This is certainly going to lead to interesting behaviour at some point in the program.
To protect ourselves from this eventuality, we need to ensure that only one thread can read or manipulate the reference count at any given time.

Fortunately there’s a specific threading tool to do this, the mutex (or mutual exclusion) lock. Unfortunately it’s not free. And since we need to lock each uniqueness check we’ll need a lot of locks. So many that it’s perfectly possible that our checks end up taking longer than making a copy of the string, rendering the entire approach rather pointless.
I’ve heard of at least one string implementation that sought to reduce the number of mutex locks by having a single mutex for *every* string in the program. This does cut down on the cost of creating locks, but has the unfortunate side effect that every access of every string is synchronised, rather defeating the point of multi-threaded string manipulation.

So is it worth it?

For my part, the chief justification for using COW is that I hate temporaries. Or, more accurately, I hate the expense of copying them left, right and centre. Winds me right up, it does.
Recall that with our naïve string, storing the result of a function call involved *two* unnecessary copies of the data.

Hold on a sec.

Did I say two *unnecessary* copies?
Doesn’t the C++ language specifically allow compilers to avoid making unnecessary copies?
There I go with my little lies again. It’s been quite a while since compilers would create any temporaries at all in this situation. C++ specifically allows such temporaries to be side-stepped (elided, in the terminology of the standard) and the result of the call to `f` to be constructed directly into `s`.

There are two situations where a C++ compiler is legally allowed to avoid copying an object.
Firstly when a function return expression is the name of a local object of the same type as the return value, the object can be constructed directly into the return value rather than copied. For example:

```
string
f()
{
  string s;
  s = "hello, world";
  return s;
}
```

In this case, rather than constructing `s`, manipulating it and copying it into the return value a C++ compiler can, by treating `s` as an alias for the return value, simply construct the return value and manipulate it directly, saving a copy.
Secondly when a temporary that has not been bound to a reference would be copied to an object of the same type, the temporary can be constructed directly into the object. For example:

```
string t = f();
```

In this case, the temporary return value of `f` can be treated as an alias of `t`, saving a copy.
Note that the two optimisations can be applied to the same statement. In the above example, this means that the string `s` in the function `f` can be treated as an alias for the string `t`, effectively eliminating two copies.

So are there any situations where unnecessary copies still exist?
Well, firstly there’s the case when strings are passed by value, but are not consequently changed. But const references capture this behaviour perfectly, so this doesn’t seem to be particularly compelling.
Secondly there’s the case when a function has multiple points of return. For example:

```
string
f(bool b)
{
  string s, t;
  s = "hello, world";
  t = "goodbye, world";

  if(b) return s;
  return t;
}
```

In such cases it can be difficult for the compiler to predict which of the return expressions should be treated as an alias for the return value. In this case, should the compiler pick `s` or `t`?
Of course, a simple restructuring of the code would make things easier for our beleaguered compiler:

```
string
f(bool b)
{
  string s;
  if(b) s = "hello, world";
  else  s = "goodbye, world";

  return s;
}
```

But there will inevitably be situations where such restructuring is difficult, or at least unwieldy.
Finally there’s the case when a temporary is assigned to a string. In this case COW *may* save us a copy. For example:

```
string
f()
{
  string s(“hello, world”);
  return s;
}

string
g()
{
  string s(“goodbye, world”);
  return s;
}

void
h()
{
  string s = f();
  //...
  s = g(); //COW may save us a copy here
}
```

The reason why COW won’t always save a copy in this case is a little subtle.

As I mentioned before, the recommended way to implement assignment is with the copy and swap idiom:

```
T &
T::operator=(const T &t)
{
  T tmp(t);
  swap(tmp);
  return *this;
}
```

In this form, we will definitely pay for a copy during assignment of a naïve `string`. This is because the compiler binds the temporary to a const reference, rather than to a newly constructed object so it can’t be elided. On the following line we copy construct the temporary, but by then it’s too late, since the function can’t know if the reference is to a temporary or a named variable.

Fortunately, we can rewrite the code to take advantage of copy elision:

```
T &
T::operator=(T t)
{
  swap(t);
  return *this;
}
```

Now the temporary is passed directly to the copy constructor of the named argument, and is therefore a candidate for the copy elision rule. The result of a function call can be constructed directly into `t` which is then swapped with the object.

So that's that for COW then, isn't it?

Well, COW can also save us a copy or two when we *might* need to change a string, but aren't certain. For example, consider a function to strip a trailing newline from a `string`:

```
string
f(string s)
{
  if(!s.empty() && *s.rbegin()=='\n')
  {
    return s.substr(0, s.size()-1);
  }
  else
  {
    return s;
  }
}

void
g()
{
  string s("Hello, world\n");
  //...
  std::cout << f(s) << std::endl;
}

void
h()
{
  string s("Hello, world");
  //...
  std::cout << f(s) << std::endl;
}
```

In the function `g`, `string` makes a copy since the original `string` has a trailing newline. In `h`, however, no change is required so no copy will be made.
We can get the same benefit, however, if we rewrite the code so that a copy is only made if we need one.
For example:

```
void
f(const string &s)
{
  if(!s.empty() && *s.rbegin()=='\n')
  {
    std::cout << s.substr(0, s.size()-1) << std::endl;
  }
  else
  {
    std::cout << s << std::endl;
  }
}
```

or:

```
void
g()
{
  string s("Hello, world\n");
  //...
  if(!s.empty() && *s.rbegin()=='\n')
  {
    std::cout << s.substr(0, s.size()-1) << std::endl;
  }
  else
  {
    std::cout << s << std::endl;
  }
}

void
h()
{
  string s("Hello, world");
  //...
  if(!s.empty() && *s.rbegin()=='\n')
  {
    std::cout << s.substr(0, s.size()-1) << std::endl;
  }
  else
  {
    std::cout << s << std::endl;
  }
}
```

OK, I'll admit it, this isn't really a very compelling argument. We'd either need a different version of the function for every operation we want to perform on the resulting string or we'd have to let the newline stripping logic leak out into the calling function, neither of which are particularly attractive prospects.

Well, that and the fact that a more sophisticated COW string wouldn't make a copy if you were just creating a sub-string. Adding offset and length members to `string` would allow sub strings to share a reference into the original string, delaying the copy until we do something *really* destructive like change some characters.

Nevertheless, I contend that this aspect of COW does not confer that big an advantage.

Consider the following function:

```
void
f(const string &s)
{
  if(!s.empty())
  {
    string::const_iterator first = s.begin();
    string::const_iterator last  = s.end();

    --last;
    while(first!=last) std::cout << *first++;
    if(*first!='\n')   std::cout << *first;
  }
}
```

This still has the unfortunate property that we'd need one function for each operation, but gains the significant advantage of making no copies whatsoever. Furthermore, with a few minor changes, we have something that looks suspiciously like a function from `<algorithm>`:

```
template<class BidIt, class T, class UnOp>
void
f(BidIt first, BidIt last, T t, UnOp op = UnOp())
{
  if(first!=last)
  {
    --last;
    while(first!=last) op(*first++);
    if(*first!=t)      op(*first);
  }
}
```

This being a generic algorithm to perform an operation, in our case printing, on every element in the iterator range except the last if it is equal to `t`.

Personally, I tend to view the STL less as a library than as a way of life, or at least I did until my girlfriend threatened to leave me if I didn't start washing once in a while. Now I guess I see it more as a way of programming.
If the STL doesn't have the algorithm or data structure I want I implement it myself and add it to my own extensions library. This can be a lot of work at the outset, but starts paying dividends fairly quickly.
I doubt I'd consider this a candidate for my extensions library, but my point is that if you *really* care about the efficiency of your string processing, I very much doubt that you would rely on delayed copy optimisation. A much better approach is to think very carefully about the algorithms you need to perform and to implement them in an efficient manner.

So, given my assertion that we can write our code in such a way that copy-on-write optimisation is unnecessary and that it has synchronisation problems to boot, is this kind of approach really worth further consideration?

Well, the growing opinion is no. Some string implementations (notably STLport) are turning to alternative optimisations such as the short string optimisation and expression templates.
The short string optimisation involves adding a small array member to the string class.
For example:

```
 class string
 {
 public:
   typedef char         value_type;
   typedef char *       iterator;
   typedef char const * const_iterator;
   //...

   string();
   string(const char *s);
   string(const string &s);

   string & operator=(const string &s);
   string & operator=(const char *s);

   const_iterator begin() const;
   const_iterator end() const;
   iterator begin();
   iterator end();
   //...

 private:
   char data_[16];
   char *begin_;
   char *end_;
};
```

When the string is less than 16 characters long the array `data_` can be used to store it in its entirety, eliminating the need for a relatively expensive allocation when copying. Longer strings must be allocated from the free store as usual. The `begin_` and `end_` pointers are used both to manage the extent of the string and determine whether the internal array or the free store have been used to store it, enabling the destructor to ensure that the memory is correctly released when the string is destroyed.

Note that this optimisation does not reduce the number of characters that are copied when strings are copied. Instead it relies upon the speed of allocating and copying memory on the stack rather than the free store.

Expression templates are altogether more complicated beasts and their precise mechanics are beyond the scope of these notes. Put simply they work by deferring string manipulation operations until the result is actually assigned to something. For example, in the code snippet:

```
string s1 = "Hello";
string s2 = "world";
string s3 = s1 + ", " + s2;
```

the strings `s1`, `", "` and `s2` are not actually concatenated until `s3`'s constructor requires the result. At this point a triple concatenation is performed, rather than the two double concatenations that a simple string implementation would perform. This saves both the creation of a temporary sub string and copying it into the final string.
In fact, expression templates are a very powerful technique that can be used for far more sophisticated optimisations than simply eliminating copies.

Despite this, I'm not yet willing to dismiss COW like optimisations out of hand, although you'll have to be patient whilst I make a brief digression before explaining why.

auto_string

Recall that it's `auto_ptr` rather than `shared_ptr` that is designed to manage ownership transfer. So, why not consider `auto_ptr` rather than `shared_ptr` semantics to eliminate the copy?
Let's look at a slightly different definition of `string`:

```
 class string
 {
 public:
   typedef char          value_type;
   typedef char *        iterator;
   typedef char const *  const_iterator;
   typedef size_t        size_type;
   //...

   string();
   string(const char *s);
   string(const string &s);
   string(const auto_string &s);

   string & operator=(const string &s);
   string & operator=(const auto_string &s);
   string & operator=(const char *s);

   auto_string release();

   const_iterator begin() const;
   const_iterator end() const;
   iterator begin();
   iterator end();
   //...

 private:
   size_type size_;
   scoped_array<char> data_;
};
```

Here, the data is stored in a `scoped_array` (like `scoped_ptr` but uses `delete[]`) rather than a `shared_array` and we've picked up a few extra functions, all of which refer to a new type, `auto_string`.
Let's have a look at the definition of `auto_string`:

```
class auto_string
{
  friend class string;

  auto_string(string::size_type n,
              const auto_array<char> &s) throw();

  string::size_type size_;
  auto_array<char> data_;
};
```

So, `auto_string` is simply a wrapper for an `auto_array` (which also uses `delete[]`) that hides its data from everything but `string`.

This gives us an explicit mechanism for stating that we wish to transfer ownership of a `string`, through the `release` member function:

```
auto_string
string::release()
{
  return auto_string(size_, data_.release());
}
```

The constructor and assignment operator can now be overloaded to take advantage of the fact that we are transferring ownership:

```
string::string(const auto_string &s) : size_(s.size_),
                                       data_(s.data_)
{
}

string &
string::operator=(const auto_string &s)
{
  string tmp(s);
  swap(tmp);
  return *this;
}
```

Now, when we assign an `auto_string` returned from a function to a `string`:

```
auto_string
f()
{
  string result(“hello, world”);
  return result.release();
}

void
g()
{
  string s;
  s = f();
}
```

we have the following sequence of events:

```
call g
default construct s;

call f
construct “hello, world” into result
transfer s into temporary auto_string
transfer temporary auto_string into return value
exit f

transfer auto_string into tmp
swap data with tmp
exit g
```

and at no point is the string data copied.
The advantage this has over the `shared_array` version is that at every step of copy construction or assignment, the string data is owned by one and only one object. In other words, we no longer have aliasing of the data and consequently don’t have to deal with the headaches that that causes.

Of course, since the compiler can optimise the copy away anyway, all we have succeeded in doing is to create a slightly better version of a completely pointless optimisation.
Well, not quite.
Thank you for your patience, by the way.

vector

There is, in fact, another benefit to explicit lifetime control but this is much better illustrated with a class representing a mathematical vector.
To start, let’s have a brief recap of `valarray`.
Just kidding:

```
class vector
{
public:
  typedef double         value_type;
  typedef double *       iterator;
  typedef double const * const_iterator;
  typedef size_t         size_type;
  //...

  vector();
  vector(const vector &v);
  vector(const auto_vector<T> &v);
  explicit vector(size_t n);

  vector & operator=(const vector &v);
  vector & operator=(const auto_vector &v);

  auto_vector release();

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...

private:
  size_type size_;
  scoped_array<double> data_;
};
```

The principal difference between this and a `std::vector` is that we want to support mathematical vector operations.
Let’s use vector addition as an example:

```
vector
operator+(const vector &l, const vector &r)
{
  vector tmp(l);
  tmp += r;
  return tmp;
}
```

Now this isn’t the most efficient implementation, creating an uninitialised vector and filling it with the result of the addition would have 3n read and write operations whereas this has 4n. Still, it does enable us to reuse the in-place addition operator.

Note that we are assuming that both in-place addition and out-of-place addition require 2n operations, with out-of-place addition incurring a further n to copy the results. Strictly speaking this is not the case since the processor must make 3n reads and writes from main memory in both cases.
However, for many processors in-place addition will make much better use of the processor cache and as a result will generally be more efficient.
A few crude experiments with my compiler supported this, showing that out-of-place addition did indeed take approximately 1.5 times as long as in-place addition, so we shall maintain these complexity assumptions as a convenient fiction.

We can effectively address the efficiency concern when `l` is bound to a function return value by redefining the operator thus:

```
vector
operator+(vector l, const vector &r)
{
  l += r;
  return l;
}
```

Now we are exploiting copy elision to reduce the number of reads and writes to 2n, even better than if we were to use an uninitialised vector to store the result.
Unfortunately, we’ll still pay for the extra copy when `l` is a variable. This is especially irritating if `r` is a function return value and hence a candidate for in-place addition. Since reference to `T` and value of `T` are afforded the same status during function overload resolution we can’t use overloading to address this.
Unless we use a different type for our temporaries. Like `auto_vector`, for example:

```
auto_vector operator+(const vector &l, const vector &r);
auto_vector operator+(const auto_vector &l, const vector &r);
auto_vector operator+(const vector &l, const auto_vector &r);
auto_vector operator+(const auto_vector &l, const auto_vector &r);
```

The implementation of each overload will look like our first version, except that we will prefer to construct the temporary from an `auto_vector` whenever possible, so that we can avoid copying one of the arguments:

```
auto_vector
operator+(const auto_vector &l, const vector &r)
{
  vector tmp(l);
  tmp += r;
  return tmp.release();
}
```

So the long awaited advantage of this approach is that it allows us to indicate when we no longer care what happens to an object, which we can exploit both for transfer initialisation and for transforming out-of-place operations into in-place operations.

Note that by returning `auto_vector`s from the addition operators we can eliminate the construction of a series of temporaries in a compound expression:

```
auto_vector
f()
{
  vector x;
  //...
  return x.release();
}

auto_vector
g()
{
  vector x;
  //...
  return x.release();
}

auto_vector
h()
{
  vector x;
  //...
  return x.release();
}

void
i()
{
  vector sum = f()+g()+h();
}
```

With the original implementation of addition, the cost of the summation would have been:

- 2 copies to temporary values at 2n read/writes each
- 2 in-place additions at 2n read/writes each

This gives us a grand total of 8n read/write operations.

Recall that if we were willing to forgo reuse of the in-place addition operator, we could remove one of the read/write operations from creating each of the temporary values, making the cost of summation:

- 2 additions at 2n read/writes each
- 2 copies to temporary values at n read/writes each

Reducing the total to 6n read/write operations.

With `auto_vector`, however, this becomes:

- 5 transfers to temporary values at O(1) read/writes each
- 2 in-place additions at 2n read/writes each

Giving us a final total of 4n + O(1) read/write operations. Not too shabby.

There are two principal disadvantages to this approach.
Firstly we have to write overloaded versions of every function and secondly we have to write a lot of boilerplate code.
Don’t we?

The crux of the first problem is that we only want to overload a function if there is an advantage to us to do so. In other words, we only want to overload functions which can exploit the reuse of temporary objects. A lot of functions can’t and we really don’t want to make work for ourselves by having to overload them.
For example:

```
double
abs(const vector &v)
{
  double sum_square = 0.0;
  vector::const_iterator first = v.begin();
  vector::const_iterator last  = v.end();

  while(first!=last)
  {
    sum_square += *first * *first;
    ++first;
  }

  return sqrt(sum_square);
}
```

Thankfully, in such cases it's perfectly OK to pass an `auto_vector` instead of a `vector`. This is because the compiler is allowed to bind a const reference to a temporary that is the result of a conversion. And a `vector` can be conversion constructed from an `auto_vector`.

The second problem is a little trickier to resolve.

auto_value

What we really need is a class that can manage the value transfer semantics for us. Much like `auto_ptr` does for pointers.

The class definition is actually pretty straightforward:

```
template<typename X>
class auto_value
{
public:
  typedef X element_type;

  explicit auto_value(X &x) throw();
  auto_value(const auto_value &x) throw();

  auto_value & operator=(const auto_value &x) throw();

  X & get() const throw();

private:
  mutable X x_;
};
```

It's the implementation that's the problem.
What we really need to find is a generic way to transfer ownership of the controlled value to and from the `auto_value`.
Fortunately, for most of the types we are interested in, one already exists in `swap`.
Let's see how we can use it to implement the member functions of `auto_value`:

```
template<typename X>
auto_value<X>::auto_value(X &x)
{
  x_.swap(x);
}

template<typename X>
auto_value<X>::auto_value(const auto_value &x)
{
  x_.swap(x.get());
}

template<typename X>
auto_value<X> &
auto_value<X>::operator=(const auto_value &x)
{
  x_.swap(x.get());
  return *this;
}
```

The chief problem with using `swap` rather than transferring the state directly is that we have to default construct a value before we can swap the transferred value in. This pretty much limits `auto_value` to types with relatively inexpensive default constructors.
Note that we've supplied an explicit transfer constructor for the original value. This is more in keeping with the `auto_ptr` interface and removes the need to add a `release` method to the class.

There's not much we can do about the conversion constructor and assignment operator that need to be implemented in the class itself.
Fortunately, these are pretty simple:

```
X::X(const auto_value<X> &x)
{
  swap(x.get());
}

X &
X::operator=(const auto_value<X> &x)
{
  swap(x.get());
  return *this;
}
```

We could make it easier still if we were to abandon our sensibilities and define these functions inline using a macro.
I can't quite bring myself to write it though.

There's only one thing left that's really lacking from the `auto_value` interface and that's `operator.`. This would allow us to use the `auto_value` as a proxy for calls to the transferred object's member functions in the same way that `operator*` and `operator->` do for `auto_ptr`.
Shame it doesn't exist.
Fortunately, there's another way to do this.

That other way is inheritance. If our `auto_value` were to inherit from rather than contain the value it controls, it would trivially be able to act as a proxy.
Let's have a look at the changes we need to make:

```
template<typename X>
class auto_value : public X
{
public:
  typedef X element_type;

  explicit auto_value(X &x) throw();
  auto_value(const auto_value &x) throw();

  auto_value & operator=(const auto_value &x) throw();
};
```

The unfortunate side effect of this is that the `auto_value` and the value it controls are now the same entity, and hence a const `auto_value` implies a const value. Now, you'll recall that function return values can't be bound to non-const references and that `swap` is a non-const member function that has a non-const reference argument.

That's right. We can't use `swap` to implement our transfer semantics anymore. Not unless we cast away the constness, that is:

```
template<typename X>
auto_value<X>::auto_value(X &x)
{
  swap(x);
}

template<typename X>
auto_value<X>::auto_value(const auto_value &x)
{
  swap(const_cast<auto_value &>(x));
}

template<typename X>
auto_value<X> &
auto_value<X>::operator=(const auto_value &x)
{
  swap(const_cast<auto_value &>(x));
  return *this;
}
```

Pretty slick, I'm sure you'll agree.
There's just one tiny little problem with this code. Hardly even worth mentioning.
It's implementation-defined whether or not this will invoke the dreaded undefined behaviour.
There's a clause in the C++ standard stating that compilers are allowed to bind const references to function returns in one of two ways. They can either bind to the value itself, or they can copy it into a const value and bind to that.
Seems innocuous enough, but there's another clause that states that trying to modify a const value results in undefined behaviour. Which we all know could mean anything from doing exactly what you expect to washing away our coding sins with the cleansing fire of a thermonuclear explosion.
I can't think of any compiler vendors who'd go for the latter option though.
Well, on reflection…
Actually, no, not even them.

`transfer`

Fortunately we can avoid invoking undefined behaviour if we are willing to force users of `auto_value` to do a little additional work.
Specifically, we'll need them to implement an ownership transfer mechanism that *will* work for const objects. Let's call it `transfer` and have a look at how `vector` might implement it:

```
class vector
{
public:
  typedef double             value_type;
  typedef double *           iterator;
  typedef double const *     const_iterator;
  typedef size_t             size_type;
  typedef auto_value<vector> auto_vector;
  //...

  vector();
  vector(const vector &v);
  vector(const auto_vector<T> &v);
  explicit vector(size_t n);

  vector & operator=(const vector &v);
  vector & operator=(const auto_vector &v);

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...

protected:
  void transfer(const auto_vector &v) const;

private:
  mutable scoped_array<double> data_;
};
```

Firstly, we've added a typedef that defines `auto_vector` as an `auto_value` of `vector`. Secondly, we've added a protected member function `transfer` to manage the ownership transfer. Finally, we've made the `data_` member mutable so that the const `transfer` function can affect it.

A protected member function? Yeah, yeah, I know.
If you're really bothered by it you can grant friendship to `auto_value<vector>` and make it private instead. The point is that the `transfer` member must be accessible by the derived `auto_value<vector>`, but probably shouldn't be public since it will violate the perfectly reasonable expectation that const objects won't change.

Some of you may also balk at the thought of forcing client classes to make their state mutable.
Well, I can give two reasons why you shouldn't worry too much about it.
Firstly, we have no choice. By making the state mutable we are allowed to change it even if the object is *really* const. This enables us to neatly side step the clause in the standard that allows compilers to copy a return value into a const object.
Secondly, it doesn't matter. Const methods are already allowed to change the state of the object.
You may find the second point a little surprising, so I'll elaborate.

The `vector` class, like most container types, stores its state in a memory buffer. If you took a look at the definition of `scoped_array`, you'd notice that the access member functions are all const, but return non-const pointers or references. This means that the elements of the array can be changed even when accessed through a const method. Whilst this may seem a little counter intuitive, it's exactly how a pointer data member would behave. They key point to note is that constness doesn't propagate through the pointer.
Namely:

```
X * const x;
```

and:

```
X const * const x;
```

do *not* mean the same thing. The former defines an immutable pointer to a mutable value and the latter an immutable pointer to an immutable value, and it is the former that is implicitly applied to pointer data members in a const member function.
The behaviour we've come to expect from our container classes, that const member functions will not change the value of any items in the container, is enforced by the programmer rather than the compiler. When implementing container classes we must always be careful not to change the state through const member functions since the compiler is unlikely to warn us that we are doing so. Making the state mutable therefore has little effect on the effort we must spend designing the class.

Now we're finished discussing the design choices, let's take a look at the implementation:

```
void
vector::transfer(const auto_value<vector> &v) const
{
  data_.swap(v.data_);
}
```

So `transfer` is actually just a `swap` for const objects, which shouldn't be particularly surprising since we've been using `swap` for ownership transfer from the start.
We'll need to provide new implementations of the conversion constructor and assignment operator used for ownership transfer:

```
vector::vector(const auto_value<vector> &v)
{
  transfer(v);
}

vector &
vector::operator=(const auto_value<vector> &v)
{
  transfer(v);
  return *this;
}
```

Again, these shouldn't be particularly suprising. We've just replaced the calls to `swap` with calls to `transfer`.
Finally, we need to rewrite the `auto_value` member functions to make use of the `transfer` function:

```
template<typename X>
auto_value<X>::auto_value(X &x)
{
  transfer(x);
}

template<typename X>
auto_value<X>::auto_value(const auto_value &x)
{
  transfer(x);
}

template<typename X>
auto_value<X> &
auto_value<X>::operator=(const auto_value &x)
{
  transfer(x);
}
```

Yet again, we have simply replaced the calls to `swap` with calls to `transfer`.

So, there we have it, a simple class implementing explicit ownership transfer that requires just a few trivial member functions and a relaxed attitude to constness in its client classes.
If we are willing to accept this compromise, `auto_value` provides the same functionality as a custom made value transfer type with a lot less work.
Recalling the `auto_vector` addition operators:

```
typedef auto_value<vector> auto_vector;

auto_vector operator+(const vector &l, const vector &r);
auto_vector operator+(const auto_vector &l, const vector &r);
auto_vector operator+(const vector &l, const auto_vector &r);
auto_vector operator+(const auto_vector &l, const auto_vector &r);
```

We can implement these operators in the same way as before, by constructing a temporary `vector` from an `auto_vector` argument (where there is one) and performing the addition in-place:

```
auto_vector
operator+(const auto_vector &l, const vector &r)
{
  vector tmp(l);
  tmp += r;
  return auto_vector(tmp);
}
```

In fact, since the `auto_value` copy constructor transfers ownership, we can further simplify these operators by passing the `auto_value` arguments by value. For example:

```
auto_vector operator+(auto_vector l, const vector &r);
```

Now, instead of transferring the `auto_vector` into a temporary `vector`, we can exploit the fact that `auto_vector` inherits from `vector` to perform in-place addition directly:

```
auto_vector
operator+(auto_vector l, const vector &r)
{
  l += r;
  return l;
}
```

saving us a little bit of typing and a few calls to `transfer`.

Once again, if we don't care about reusing temporaries we simply provide a single version of a function:

```
double abs(const vector &v);
```

As before, this function will work for both `vectors` and `auto_vectors` although now this is because `auto_vector` inherits from `vector` and can therefore be bound directly to the reference.

namespace mojo

In his article Generic<Programming>: Move Constructors (Dr Dobb's Journal, 2003), Alexandrescu describes the Mojo framework for *automatically* detecting temporaries and using move semantics for them.
By its very nature this technique must rely upon implicit, rather than explicit, conversion to transfer types and hence requires a little extra work to ensure that temporaries are bound to the correct transfer type.
The upshot of this is that three overloads, rather than two, must be provided for each function that exploits move semantics. Nevertheless, this is arguably a more sophisticated solution to the problem.

T &&t

To close, it's worth mentioning that the value transfer semantics we've worked so hard to implement are trivial using the above notation.

If you're wondering why on Earth I didn't just go ahead and use it instead of leading you down the garden path, it's because it isn't C++. Not yet.

The notation is for a new kind of reference proposed for the next version of C++, the rvalue reference. The rvalue reference differs from the familiar reference (hereafter known as an lvalue reference) in eight ways. Paraphrasing the proposal slightly:

1. A non-const rvalue can bind to a non-const rvalue reference.
2. Overload resolution rules prefer binding rvalues to rvalue references and lvalues to lvalue references.
3. Named rvalue references are treated as lvalues.
4. Unnamed rvalue references are treated as rvalues.
5. The result of casting an lvalue to an rvalue reference is treated as an rvalue.
6. Where elision of copy constructor is currently allowed for function return values, the local object is implicitly cast to an rvalue reference.
7. Reference collapsing rules are extended to include rvalue references.
8. Template deduction rules allow detection of rvalue/lvalue status of bound argument.

For the purpose of eliminating copies, behaviours 1-3 are of particular interest. Put simply, they mean that a non-const temporary can be bound to a reference type through which we *can* legally modify them. Specifically, we now have four kinds of reference:

```
    typedef T &&       Ref1; //non-const reference to rvalue
    typedef T const && Ref2; //const reference to rvalue
    typedef T &        Ref3; //non-const reference to lvalue
    typedef T const &  Ref4; //const reference to lvalue
```

In the same way that a non-const object is preferentially bound to a non-const reference, an rvalue is preferentially bound to an rvalue reference and an lvalue is preferentially bound to an lvalue reference. Given the following function signatures:

```
    f(T &&t);        //1
    f(T const &&t); //2
    f(T &t);         //3
    f(T const &t);  //4
```

and the following references:

```
    T &&t1;
    T const && t2;
    T & t3;
    T const & t4;
```

The rules mean that:

`t1` binds to overloads 1, 2, 3 and 4 in that order of preference
`t2` binds to overloads 2 and 4 in that order of preference
`t3` binds to overloads 3, 4, 1 and 2 in that order of preference
`t4` binds to overloads 4 and 2 in that order of preference

The original justification for disallowing the binding of rvalues to non-const references was that it was generally a mistake to do so. Changing an object which is about to go out of scope and be destroyed will, after all, lose those changes.
This is especially nasty when the object in question is a temporary resulting from an implicit conversion:

```
void f(long &i);

void g()
{
  char c = 0;
  f(c); //oops
  //...
}
```

Since the character `c` is promoted to a long with an implicit conversion, `f` receives a reference to the temporary long that is the result of that conversion. The upshot of which is that, generally to the surprise and consternation of the programmer, `c` never changes.

Move semantics, however, have brought to light a situation in which this is a valid thing to want to do. The new rvalue references provide a mechanism by which we can express that we deliberately wish to change the value of a temporary object whilst avoiding the original problem.
Specifically, overloading on non-const rvalue reference and const lvalue reference enables us to distinguish between destructively reusing a temporary object and non-destructively referring to a non-temporary object.

Which is, of course, exactly what we've been striving to achieve.

`notes::~notes()`

So has this all been a tremendous waste of time?
I hope not.

Firstly, I hope that this has given you cause to think a little more about how you can exploit ownership control.
Secondly, I hope that you may find `auto_value`, or something like it, a useful stop gap.
And finally, I hope that you've enjoyed it.

If not, I refer you to Harris's Addendum:

> Nyah, nyah. I can't hear you.

`#include`

Krauss and Starkman. *Universal Limits on Computation* (arXiv:astro-ph/0404510 v2, 2004).
Alexandrescu. *Generic<Programming>: Move Constructors* (Dr Dobb's Journal, 2003).
Hinnant, Abrahams and Dimov. *A Proposal to Add an Rvalue Reference to the C++ Language* (ISO/IEC JTC1/SC22/WG21 Document Number N1690, 2004).

With thanks to Kevlin Henney for his review of these notes and Astrid Osborn, Keith Garbutt and Niclas Sandstrom for proof reading them.