metrowerks

Inside std::vector

Howard E. Hinnant
Senior Library Architect

MOTOROLA

October 24, 2003

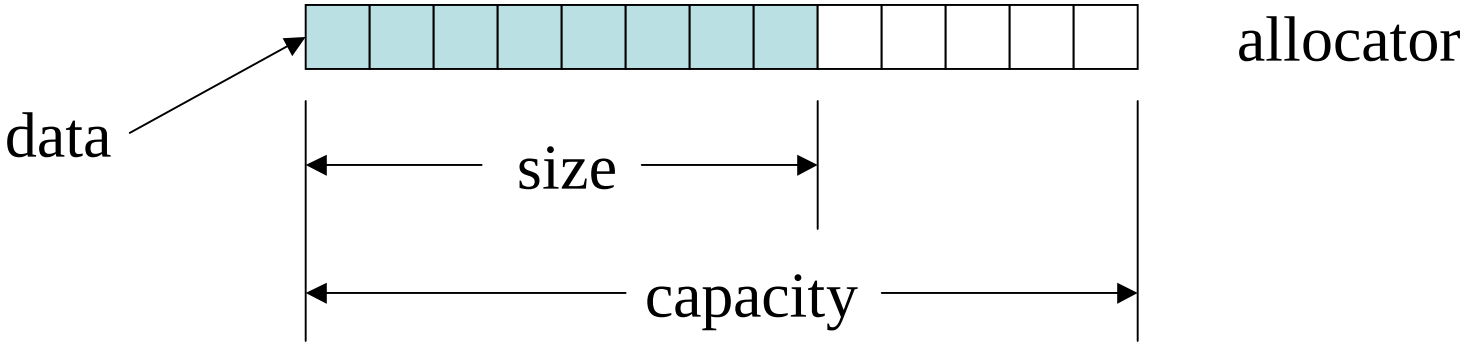
Overview

- **This presentation will cover:**
  - The anatomy of vector
  - Data layout
  - Exception safety
  - Template code bloat reduction
  - Selective optimization based on type traits
  - Restricted template techniques
  - Move semantics
  - Invariant checking
  - Debug iterators

vector interface

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
   ...
   explicit   vector(const Allocator& = Allocator());
   explicit   vector(size_type n, const T& value = T(),
                     const Allocator& = Allocator());
   ...
   void    reserve(size_type n);
   ...
   void    resize(size_type sz, const T& c = T());
   ...
   void    push_back(const T& x);
   void    pop_back();
   ...
   iterator insert(iterator position, const T& x);
   void      insert(iterator position, size_type n, const T& x);
   ...
   iterator erase(iterator position);
   iterator erase(iterator first, iterator last);
   ...
   void      clear();
};
```

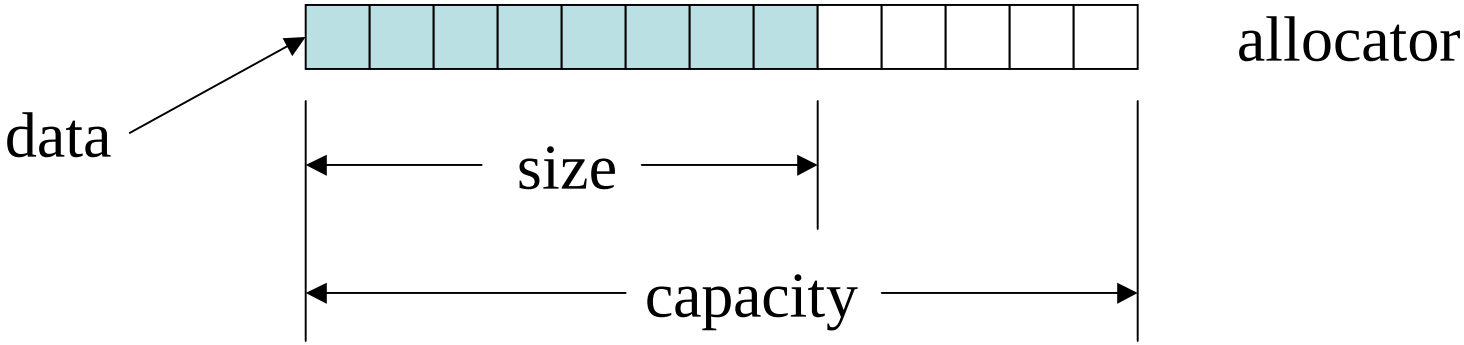
What is a vector?



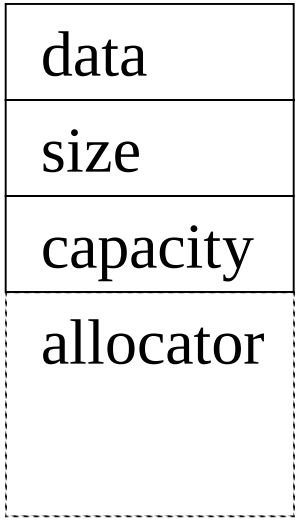
- **A contiguous array of elements**
- **The first “size” elements are constructed (initialized)**
- **The last “capacity - size” elements are uninitialized**
- **Four data members**
  - data pointer
  - size
  - capacity
  - allocator

  } or equivalent

Data layout



- **allocator typically empty class**
  - optimize away space for allocator using compressed_pair<capacity, allocator>
- **Typical vector has only 3 words of overhead**



compresed_pair<T1, T2>

- **Available in boost & Metrowerks.**
- **Derives from T1 and / or T2 only if it is an "empty class".**
  - Takes advantage of "empty base class optimization."
- **Examples:**
  - compressed_pair<int, int> : sizeof = 2*sizeof(int)
  - compressed_pair<int, less<int> > : sizeof = sizeof(int)
  - std::pair<int, less<int> > : sizeof = 2*sizeof(int)

Type traits

- **is_scalar<T>::value**
  - true if T is a scalar, otherwise false.
    - arithmetic types
    - pointers
    - member pointers
    - enums
- **Trivial : member is implicitly defined and…**
- **has_trivial_copy_ctor<T>::value**
  - true if T can be copy constructed with memcpy, otherwise false.
- **has_trivial_assignment<T>::value**
  - true if T can be assigned with memcpy, otherwise false.
- **has_trivial_dtor<T>::value**
  - true if T's destructor does nothing, otherwise false.

int2type

- General purpose utility that will transform a compile time integral value into a unique type.
- Used for compile time polymorphism.

```
template <int I>
struct int2type
{
    static int const value = I;
};
```

Constructor exception safety

- **First try:**

```
template <class T, class A>
vector<T, A>::vector(size_type n, const T& value, const A& a)
    :   data_(0),
        size_(0),
        capacity_(0, a)
{
    if (n > 0)
    {
        data_ = alloc().allocate(n);
        cap() = size_ = n;
        pointer e = data_ + n;
        for (pointer i = data_; i < e; ++i)
            alloc().construct(i, value);
    }
}
```

leaking

Might throw!

Constructor exception safety - 2

- try/catch solution

```
template <class T, class A>
vector<T, A>::vector(size_type n, const T& value, const A& a)
    :   data_(0),
        size_(0),
        capacity_(0, a)
{
    if (n > 0)
    {
        data_ = alloc().allocate(n);
        cap() = size_ = n;
        pointer i;
        pointer e = data_ + n;
        …
```

Constructor exception safety - 3

- **try/catch solution continued:**

```
        try
        {
            for (i = data_; i < e; ++i)
                alloc().construct(i, value);
        }
        catch (...)
        {
            for (pointer j = data_; j < i; ++i)
                alloc().destroy(j);
            alloc().deallocate(data_, cap());
            throw;
        }
    }
}
```

Two approaches:

1. Charge forward and only worry about cleaning up if a problem occurs.
2. Keep things in order at all times so that if a problem occurs, the problem isn't compounded.

(I prefer the second approach)

Constructor exception safety - 4

- **Solution without try/catch:**
  - Create private base class with data members, destructor, but very limited constructors. Base constructors do not acquire resources. No copy constructor!

```
template <class T, class A>
vector<T, A>::vector(size_type n, const T& value, const A& a)
    :   base(a)
{
    base::init(n, value);
}
```

  - base is a fully default constructed object before init() is called.
  - If base::init throws an exception, ~base() is still run, cleaning up all memory.
  - Copy constructor implemented at derived level.

Constructor exception safety - 5

- **Solution without try/catch continued:**

```
template <class T, class A>
vector_base<T, A>::vector_base(const A& a)
    :   data_(0),
        size_(0),
        capacity_(0, a)
{
}
```

```
template <class T, class A>
vector_base<T, A>::~vector_base()
{
    clear();
    if (data_)
        alloc().deallocate(data_, cap());
}
```

```
template <class T, class A>
void
vector_base<T, A>::init(size_type n,
                        const T& value)
{
    if (n > 0)
    {
        data_ = alloc().allocate(n);
        cap() = size_ = n;
        pointer e = data_ + n;
        for (pointer i = data_; i < e; ++i)
            alloc().construct(i, value);
    }
}
```

Constructor exception safety - 6

- **Analysis of vector_base::init**
  - Must have “basic exception safety” guarantee
    - Must not leak memory.
    - Must leave vector_base in a self-consistent state.
    - Does not need “commit or rollback” semantics.
  - If any call to construct() throws, size_ will have the wrong value.
    - Fails self-consistent state test.

Constructor exception safety - 7

- **Got it right!**

```
template <class T, class A>
void
vector_base<T, A>::init(size_type n, const T& value)
{
    if (n > 0)
    {
        data_ = alloc().allocate(n);
        cap() = n;
        pointer e = data_ + n;
        for (pointer i = data_; i < e; ++i, ++size_)
            alloc().construct(i, value);
    }
}
```

If init() throws, ~vector_base() will run.

If allocate throws, vector_base is in a self-consistent state.

vector_base is in a self-consistent state at each step through the loop.

Layered approach

- **vector derives privately from vector_base**
- **vector_base contains all of the non-trivial code**
  - vector is a thin inlined layer above vector_base

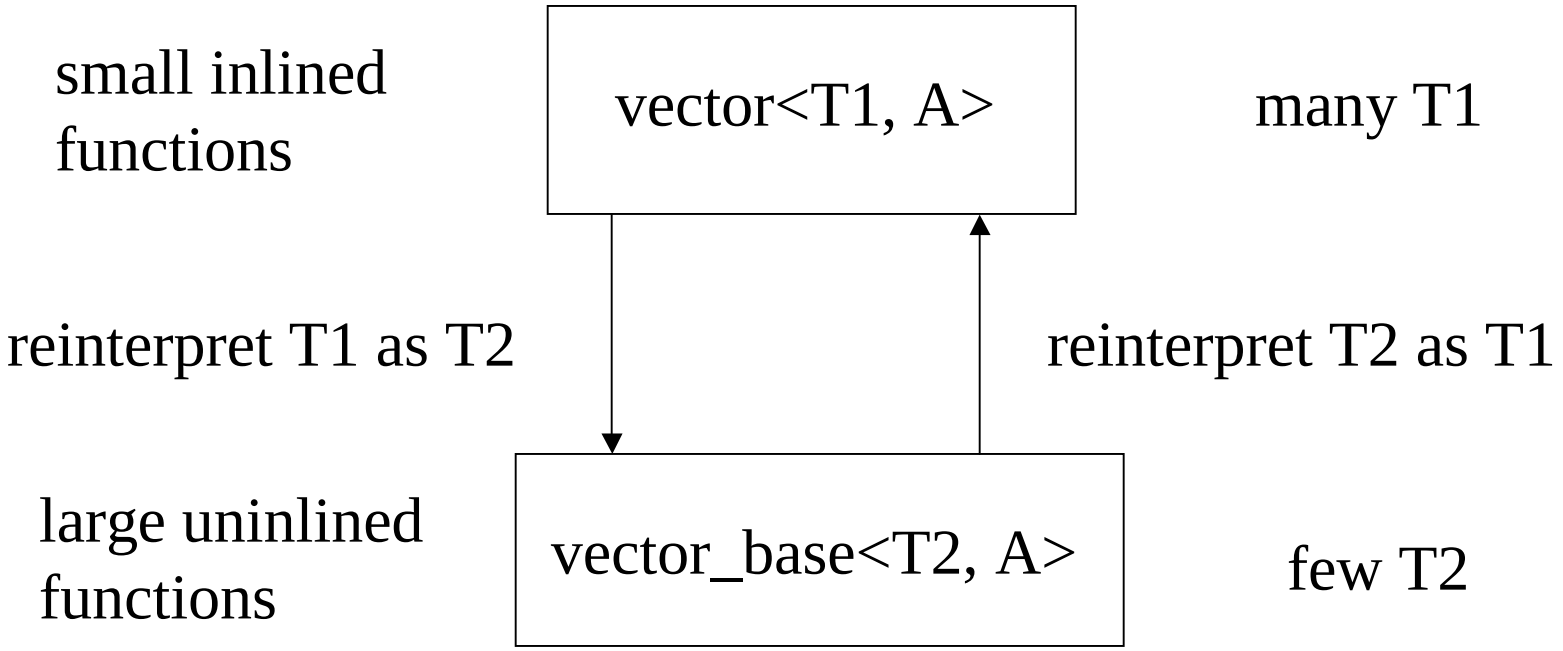
```
template <class T, class A>
class vector
    :   private vector_base<T, A>
{
    ...
};
```

Dual layering aides with exception safety of constructors.

- **Would like to have as few instantiations of vector_base<T, A> as possible.**
- **But since vector is a thin inlined layer, can have many different instantiations without cost.**

Reducing Template Code Bloat

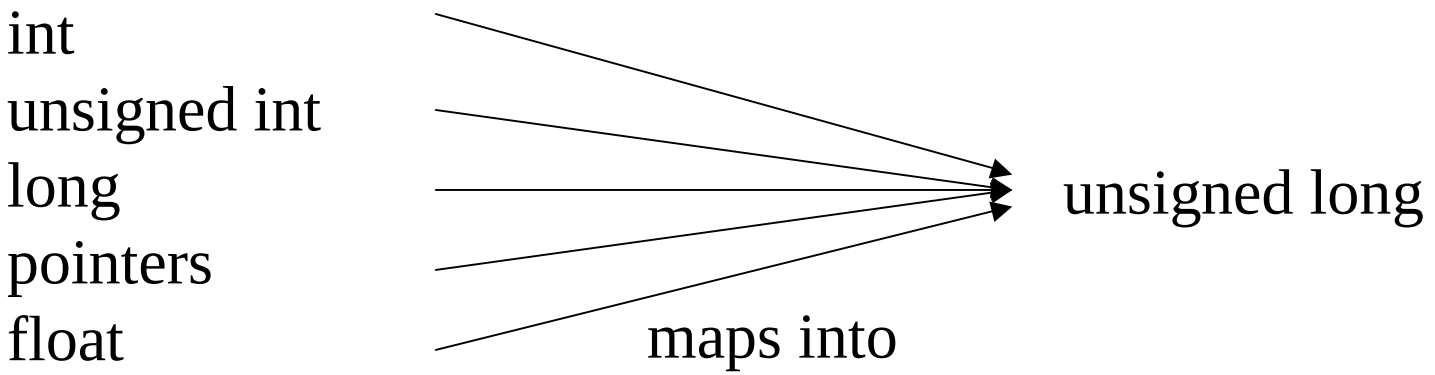
- When T is a scalar, vector_base<T, A> can be considered just a bit bucket, holding enough bits to represent T.
- T1 and T2 can both use the same vector_base as long as T1 and T2 are both scalars and sizeof(T1) == sizeof(T2)



store_as map

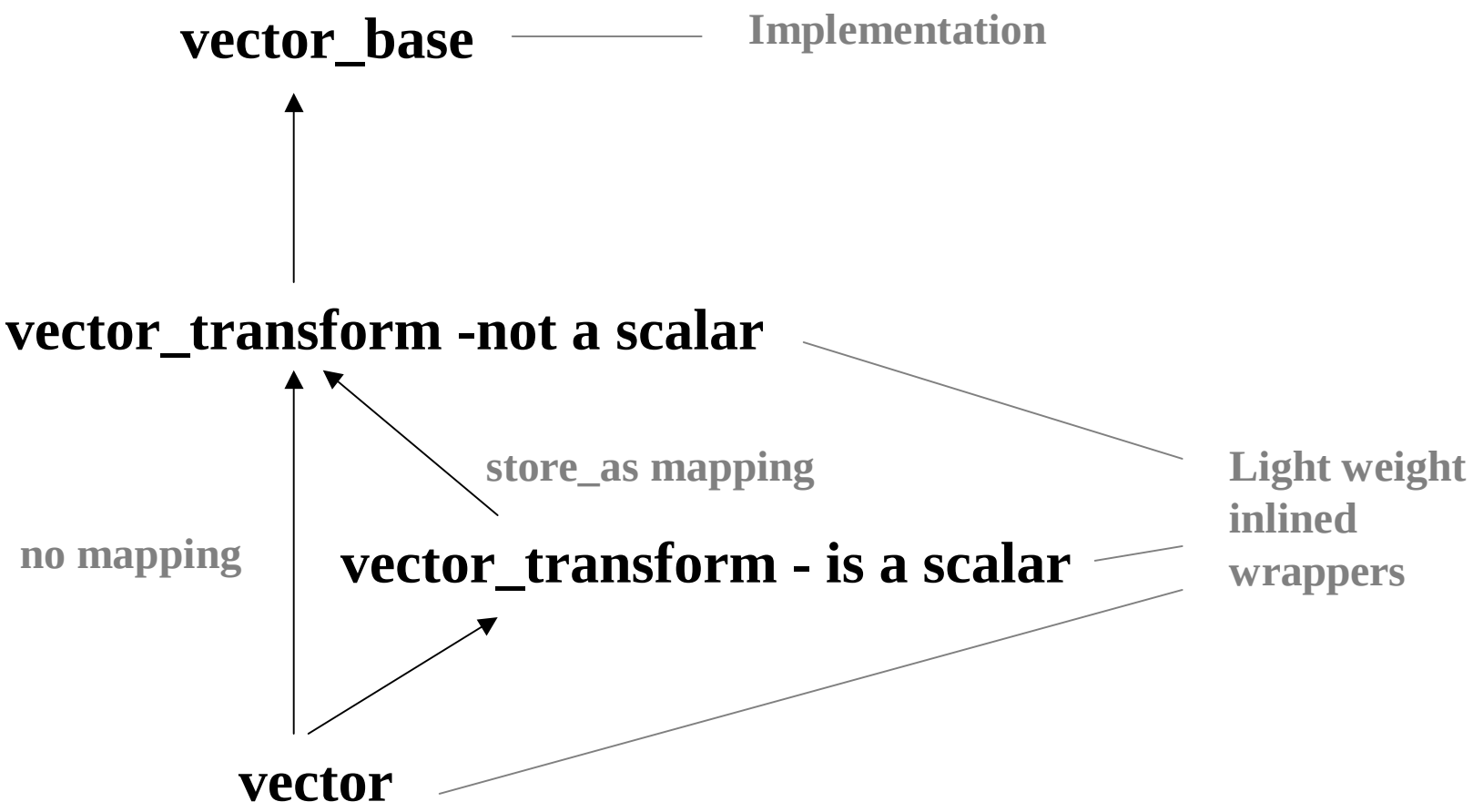
- **Need to map several types T1 into a representative type T2**

For example on popular 32 bit platforms:




```
template <class T> struct store_as              {typedef T type;};
template <>        struct store_as<int>         {typedef unsigned long type;};
template <>        struct store_as<unsigned int> {typedef unsigned long type;};
template <>        struct store_as<long>        {typedef unsigned long type;};
template <class T> struct store_as<T*>          {typedef unsigned long type;};
template <>        struct store_as<float>       {typedef unsigned long type;};
…
```

Hierarchy for template code bloat reduction



Template code bloat reduction

```
template <class T, class A>
class vector_base
{...};

template <class T, class A, bool IsScalar>
class vector_transform
    :   protected vector_base<T, A>
{...};

template <class T, class A>
class vector_transform<T, A, true>
    :   protected vector_transform
        <
            typename store_as<T>::type,
            typename A::rebind<typename store_as<T>::type>::other,
            false
        >
{...};
```

```
template <class T, class A>
class vector
    :   private vector_transform
        <
            T,
            A,
            is_scalar<T>::value
        >
{...};
```

vector_transform - T is a scalar

- **Must reinterpret value_type as base::value_type for information headed down to the base class.**
- **Must do the reverse for return values.**
  - Example:

```
iterator insert(iterator position, const value_type& x)
{
    return (iterator)base::insert
        (
            (typename base::iterator)position,
            (const typename base::value_type&)x
        );
}
```

Summary of layers

• vector_base	• Implements everything but resource acquiring constructors
• vector_transform - not scalar	• Implements resource acquiring constructors.
• vector_transform - scalar	• Maps several scalar types onto a single implementation.
• vector	• Chooses which vector_transform should serve as the implementation.

Factor often used code into private helper functions

- **void allocate(size_type n);**
  - serves init's / constructors
- **void reallocate_nocopy(size_type n);**
  - serves most assign overloads
- **size_type grow_by(size_type n);**
  - serves insert, push_back, resize
- **void append_realloc(size_type n, const value_type& x);**
  - serves push_back, resize
- **void erase_at_end(size_type n);**
  - serves clear, assign, resize, erase
- **etc…**

Use of type traits to optimize erase_at_end

- **Two versions of erase_at end:**
  - T does not have a trivial destructor.
  - T does have a trivial destructor.

```
void erase_at_end(size_type n)
   {erase_at_end(n, int2type<has_trivial_dtor<value_type>::value>());}

void erase_at_end(size_type n, int2type<false>); // not optimized
void erase_at_end(size_type n, int2type<true>);  // optimized
```

erase_at_end - optimized and not

```
template <class T, class A>
void
vector_base<T, A>::erase_at_end(size_type n, int2type<false>)
{
    iterator i = end();
    size_ -= n;
    for (; n > 0; --n)
        alloc().destroy(--i);
}
```

Too big to inline
if ~T() is non-trivial.
24 instructions on PPC.

```
template <class T, class A>
inline
void
vector_base<T, A>::erase_at_end(size_type n, int2type<true>)
{
    size_ -= n;
}
```

assign(n, value) - compile time polymorphism

- **Optimize if all of the special members are trivial**

```
void assign(size_type n, const T& x)
   {assign(n, x, int2type<
          has_trivial_copy_ctor<T>::value &&
          has_trivial_assignment<T>::value &&
          has_trivial_dtor<T>::value
                >());}
```

assign(n, value) - not optimized

```
template <class T, class A>
void
vector_base<T, A>::assign(size_type n, const T& x, int2type<false>)
{
    if (n <= cap()) {
        std::fill_n(data_, min(n, size_), x);
        if (n < size_)
            erase_at_end(size_ - n);
        else if (size_ < n) {
            pointer i = data_ + size_;
            for (n -= size_; n > 0; --n, ++i, ++size_)
                alloc().construct(i, x);
        }
    } else {
        reallocate_nocopy(n);
        for (pointer i = data_; n > 0; --n, ++i, ++size_)
            alloc().construct(i, x);
    }
}
```

156 instructions

assign(n, value) - optimized

```
template <class T, class A>
void
vector_base<T, A>::assign(size_type n, const T& x, int2type<true>)
{
    if (n > cap())
        reallocate_nocopy(n);
    std::fill_n(data_, n, x);
    size_ = n;
}
```

44 instructions

assign(n, value) - really not optimized

```
template<class T, class A>
void
vector<T, A>::assign(size_t n, const T& x)
{
    if (n > cap())
    {
        vector tmp(n, x, get_allocator());
        tmp.swap(*this);
    }
    else if (n > size_)
    {
        std::fill(data_, data_ + size_, x);
        std::uninitialized_fill_n(data_, n - size_, x);
        size_ = n;
    }
    else
        erase(fill_n(data_, n, x), data_ + size_);
}
```

bug!

239 instructions

Requires twice the memory for no reason, making exception more likely to happen.

Needless try/catch clause hidden here.

erase(it, it) is a superset of the functionality needed here.

2 bugs!

assign(n, value) - The ultimate in pessimization, std::text

416 instructions

```
template <class T, class A>
void
vector<T, A>::assign(size_type n, const T& x)
{
    erase(begin(), end());
    insert(begin(), n, x);
}
```

- **For most classes assignment is much more efficient than a destruction followed by a construction.**

Restricting templates to guide overload resolution

- **Sometimes a templated function is not truly generic, for example:**

```
template <class T, class A>
class vector
{
public:
    ...
    template <class InputIterator>
    void
    insert(iterator position,
           InputIterator first, InputIterator last);
    void insert(iterator position, size_type n, const T& x);
    ...
};
```

- **InputIterator must really be an input iterator, not another type, such as int.**

Restricted templates

- **Consider:**

```
std::vector<int> v;
v.insert(v.begin(), 10, 1);
```

- **This is an exact match for the templated (InputIterator) insert function. The non-templated size-value insert function is not an exact match because it involves a trivial conversion of the int 10 to vector::size_type (typically a size_t).**
- **One solution is to "restrict" the templated insert to non-integral types.**

restrict_to

```
template <bool b, class T = void> struct restrict_to          {};
template <class T>                struct restrict_to<true, T> {typedef T type;};
```

- **Also known as enable_if at boost.**
- **The compiler now will not consider the template if InputIterator has integral type.**

```
template <class T, class A>
class vector {
public:
    ...
    template <class InputIterator>
    typename restrict_to
    <
        !is_integral<InputIterator>::value,
        void
    >::type
    insert(iterator position,
           InputIterator first, InputIterator last);

    void insert(iterator position, size_type n, const T& x);
    ...
};
```

restrict_to with constructors

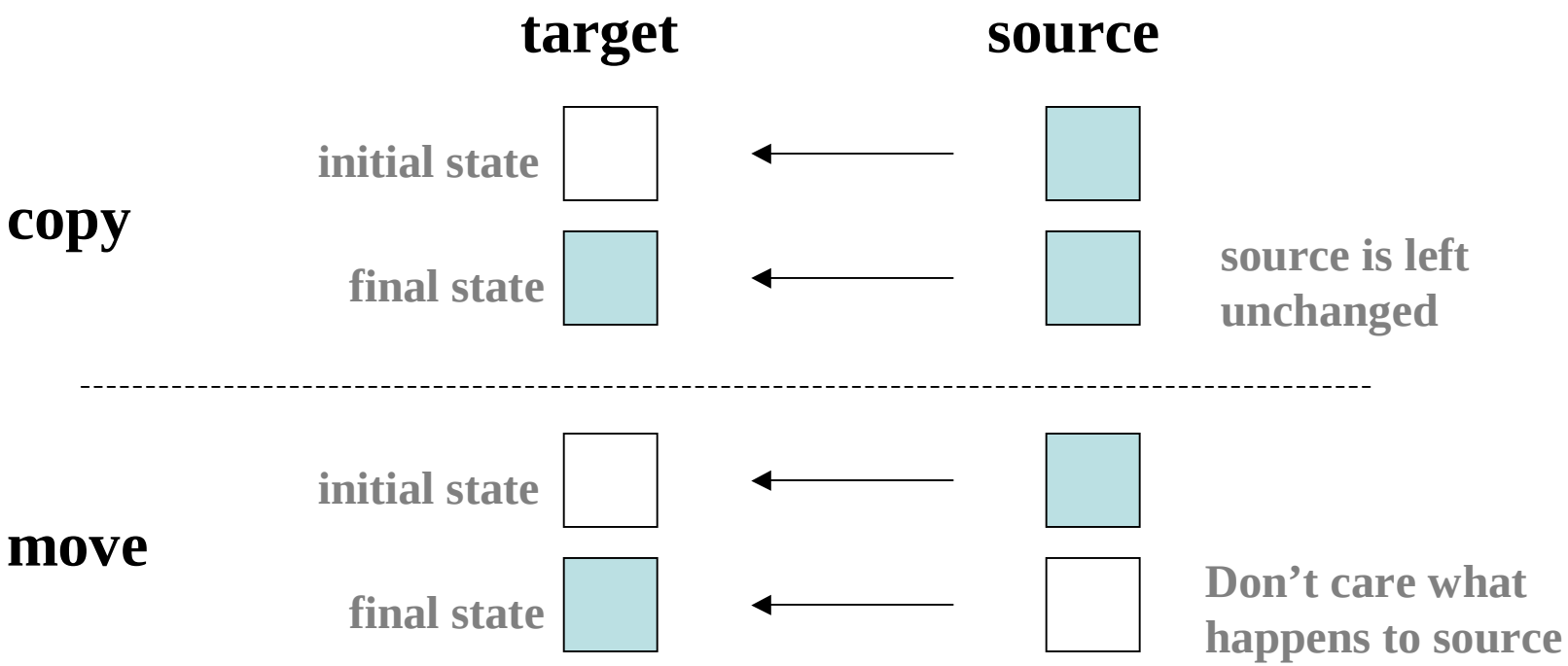
- **restrict_to can be applied to return types, or as a defaulted parameter (needed for restricted templated constructors):**

```
template <class T, class A>
class vector {
public:
    template <class InputIterator>
    vector(InputIterator first, InputIterator last,
            typename restrict_to<!is_integral<InputIterator>::value>::type* = 0);

    vector(size_type n, const T& value);
    ...
};
```

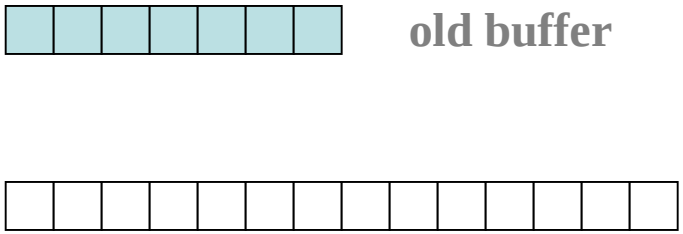
Move semantics

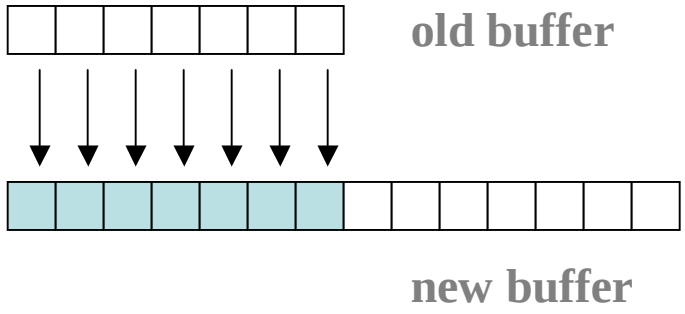
- Move is the ability to cheaply transfer the value of an object from source to target, with no regard for the value of the source after the move.



Move-aware vector

- vector can make good use of move semantics when creating a new internal buffer.

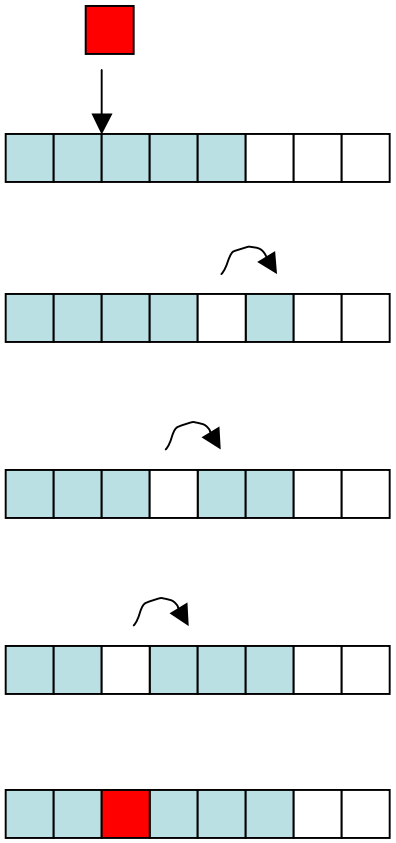




- Elements are **moved** (not copied) to the new buffer.

Move-aware vector

- vector can make good use of move semantics when inserting and erasing within a single buffer



- Elements are **moved** (not copied) within the buffer to create a “hole” for the new element.

Move semantics - current limitations

- **Without language changes, it is practical for vector to use move semantics only for those types it knows about:**
  - vector
  - string
  - list
  - deque
  - map
  - multimap
  - set
  - multiset
  - other non-standard containers (extensions)
- **vector can move these types with memcpy, swap, or other means agreeable to the type being moved.**

Move semantics - timing examples

- vector<string>::insert, no reallocation

```
std::string s(20, ' ');
std::vector<std::string> v;
v.reserve(101);
v.assign(100, s);
clock_t t = clock();
v.insert(v.begin(), s);
t = clock() - t;
```

move semantics 12 times faster

- vector<string>::insert, with reallocation

```
std::string s(20, ' ');
std::vector<std::string> v(100, s);
clock_t t = clock();
v.insert(v.begin(), s);
t = clock() - t;
```

move semantics 23 times faster

Move semantics - timing examples

- vector<string>::erase

```
std::string s(20, ' ');
std::vector<std::string> v(100, s);
clock_t t = clock();
v.erase(v.begin());
t = clock() - t;
```

move semantics 14 times faster

- vector<multiset<string>>::erase

```
std::string s(20, ' ');
std::multiset<std::string> ms;
for (int i = 0; i < 100; ++i)
    ms.insert(s);
std::vector<std::multiset<std::string> > v(100, ms);
clock_t t = clock();
v.erase(v.begin());
t = clock() - t;
```

move semantics 200 times faster

Object invariants

- **Most C++ objects have invariants**
- **Invariants are relationships among the different parts of state of an object which must remain true all of the time, except during the middle of a state change.**
- **Invariants may or may not be observable by clients, but they must still always be true.**
- **vector invariants for the Metrowerks implementation:**
  - size() <= capacity().
  - If data pointer is 0, then capacity() is also 0.
  - If data pointer is not 0, then capacity() > 0.

Checking invariants

- **It is useful in debugging to be able to check an object's invariants at any time.**
  - Useful for debugging the object.
  - Useful for debugging clients of the object.

```
template <class T, class A>
class vector
{
public:
    ...
    bool invariants() const;  // extension
};
```

- **Clients can check for invariants, then take whatever action is appropriate if they are not true.**

assert(vec.invariants());

debug mode

- **In debug mode, vector can check for common mistakes such as:**
  - indexing out of bounds
  - dereferencing invalidated iterators
  - erasing one vector with iterators referencing another vector
  - inserting with invalid iterator
  - comparing iterators from two different vectors
  - pop_back on empty vector

debug mode - 2

• vector_base	• Implements everything but resource acquiring constructors
• vector_transform - not scalar	• Implements resource acquiring constructors.
• vector_transform - scalar	• Maps several scalar types onto a single implementation.
• vector	• Chooses which vector_transform should serve as the implementation. • **Wraps pointers with class iterators for non-debug mode.** • **Adds debug iterators and other debugging checks.**

debug iterator

- **A debug iterator knows what container it points into.**
  - stores pointer back to the container.
- **The container keeps a list of all iterators that point into it.**
- **An iterator can be "invalidated" by removing it from the container's list, and zeroing its container "owner" pointer.**
- **The container knows what member functions can invalidate which iterators.**
  - E.g., insert(position, value) will invalidate all iterators at or past position, and if the insert causes a reallocation, will invalidate all iterators.
- **Iterator invalidation must follow exception safety protocols.**
  - E.g., push_back may only invalidate iterators if there was a reallocation, and if no exception is thrown (even by the copy ctor of T).

debug iterator - insert example

```
iterator insert(iterator position, const value_type& x)
{
    typename base::iterator result;
    {
    __invalidate_past_pos __s(*this, position);
    __invalidate_on_reallocate __c(*this);
    result = base::insert(__iterator2pointer(position), x);
    }  // invalidations happen here
    return __pointer2iterator(result);
}
```

1. **Invalidates iterators at or past position, but only if the insert actually happened.**
2. **Invalidates all iterators if the underlying buffer changes.**
3. **Checks that position is actually "owned" by this container (i.e. has not been invalidated).**

Summary

- **vector is a very simple container … conceptually.**
- **There are many complicating factors for the implementation including:**
  - data layout, optimization of allocator
  - exception safety
  - template code bloat reduction
  - optimized set of private helper functions
  - size and speed optimizations for trivial special members
  - correctly handling the member templates
  - move semantics
  - debugging aids
- **All of these issues are applicable to other C++ objects as well.**