C++11 Move

l-value / r-value `c++03`

```
int fct(int const & a) {
    return a;
}

int main() {
    // the right way: lvalues left of = / rvalues right of =
    int l = 10;
    int const lc = fct(l);

     // the wrong way: rvalues left of =
    fct(l) = l;
    10 = lc;

    // lvalues may also be on the right side

    return 0;
}
```

the valueness is **independent** of the type (i.e. there are `int` lvalues and `int` rvalues)

l-value

- “can be on left of =“
- Has an identity
- Examples:
  - int lval;
  - int const cval;
  - ++lval;
  - std::cout;
  - std::cout << lval;
  - a=b;
  - a+=b;
  - fct(); // int & fct()

r-value

- “can only be on right of =“
- Does not have an identity
- Examples:
  - 1+2;
  - lval++;
  - fct(); // int fct()
  - a == b;
  - &lval;

r-value references

```
int main() {

    int l = 10;

    int & lref = l;

    int & lref = 10; // fails!
    int const & clref = 10; // ok

    // the new c++11 rvalue reference
    int && rref = 10; // works

    // so what have we gained?
    // Not much without move-constructors / move-assignment

    return 0;
}
```

„Factory“ Function Problem

```
std::vector<int> all_int_prime_nr() { std::vector<int> res; ... return res; };
void all_int_prime_nr_ref(std::vector<int> & pnr);

int main() {
    // looks like it needs one ctor, one copy-ctor and one dtor
    // don‘t we have at some point two (huge) copies?
    // actually, this is no problem due to „copy elision“, only one ctor
    std::vector<int> pnr = all_int_prime_nr();

    // only one ctor, perhaps more efficient
    std::vector<int> pnr_efficient;
    all_int_prime_nr_ref(pnr_efficient);

    // but as soon as there is no construction anymore the problem arises
    std::vector<int> pnr_later;
    pnr_later = all_int_prime_nr();

    return 0;
}
```

std::vector (c++03)

```
template<typename T, typename Alloc = allocator<T> >
class vector {

    public:
    using size_type = uint32_t;
    using value_type = T;

    vector();
    ~vector();
    vector(vector const & rhs);
    vector & operator=(vector const & rhs);

    void push_back(value_type const & val);
    void reserve(size_type const & new_cap);
    void clear();

    size_type const & size() const     { return size_; }
    size_type const & capacity() const { return capacity_; };

    private:
    void check_capacity(); // small helper
    value_type * data_;
    size_type size_;
    size_type capacity_;
};
```

std::vector structors

```
// constructor
vector(): data_(nullptr)
        , size_(0)
        , capacity_(0)
{}

// copy-constructor
vector(vector const & rhs): data_(Alloc::allocate(rhs.size()))
                          , size_(0)
                          , capacity_(rhs.size()) {

    for(size_type i = 0; i < rhs.size(); ++i)
        Alloc::construct(data_ + size_++, rhs[i]);
}

// destructor
~vector() {
    clear();
    Alloc::deallocate(data_, capacity());
}
```

push_back (c++03)

```
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
void check_capacity() {
    if(size() == capacity()) {
        if(capacity() == 0)
            reserve(1);
        else
            reserve(2*capacity());
    }
}
void reserve(size_type const & new_cap) {
    value_type * new_data = Alloc::allocate(new_cap);

    for(size_type i = 0; i < size(); ++i)
        Alloc::construct(new_data + i, data_[i]);

    for(size_type i = 0; i < size(); ++i)
        Alloc::destruct(data_ + i);

    Alloc::deallocate(data_, capacity());
    data_ = new_data;
    capacity_ = new_cap;
}
```

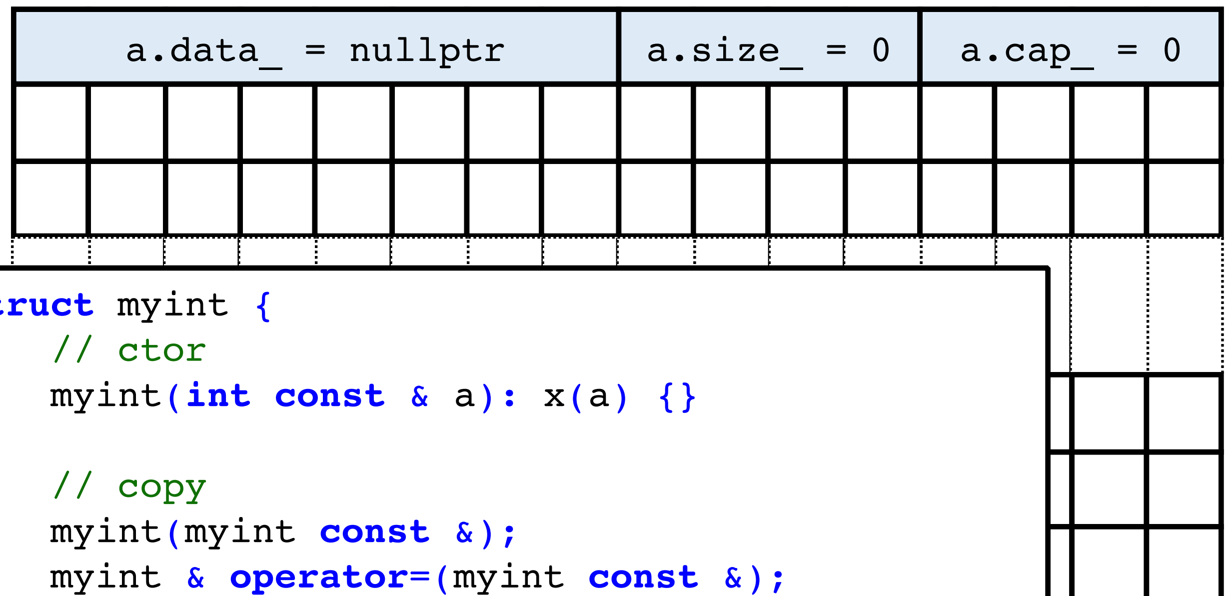
two loops for the strong exception guarantee

allocator (c++03)

```
template<typename T>
struct allocator {

    static T * allocate(std::size_t const & n) {
        // this form of new returns n * sizeof(T) bytes memory
        return static_cast<T*>(::operator new(n * sizeof(T)));
    }
    static void deallocate(T * p, std::size_t const & n) {
        ::operator delete(p);
    }

    static void construct(T * p, T const & t) {
        // this is called a „placement new“. It doesn‘t allocate any memory
        new (p) T(t); // copy-ctor call
    }
    static void destruct(T * p) { p->~T(); }
};
```

what happens in memory

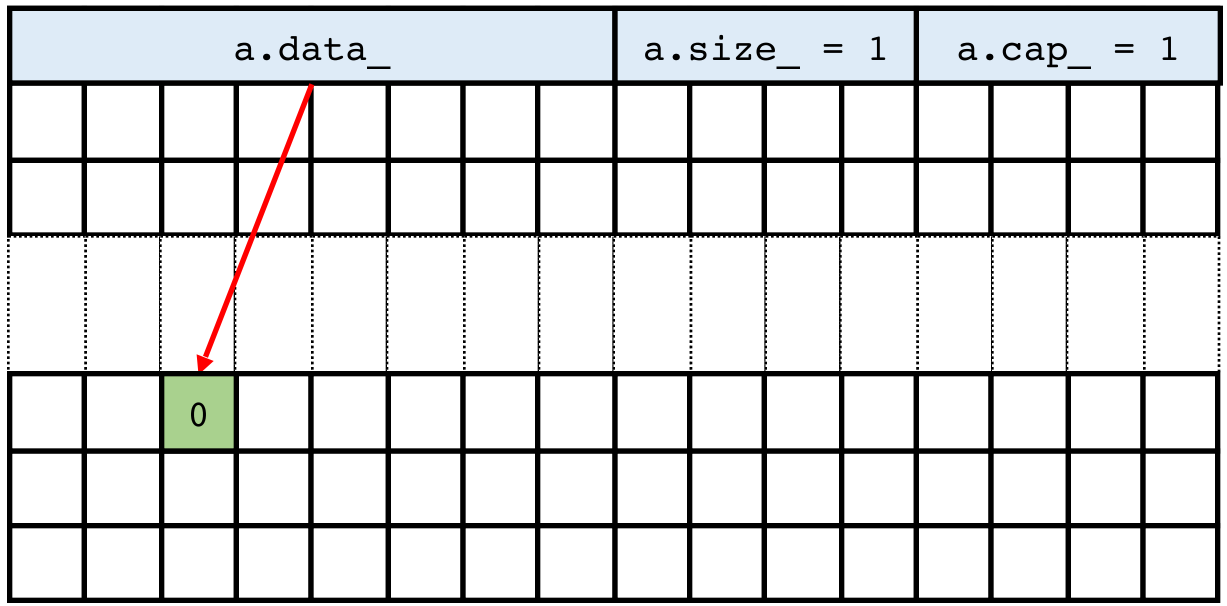
```
int main() {
    vector<myint> a;



    return 0;
}
```




```
struct myint {
    // ctor
    myint(int const & a): x(a) {}

    // copy
    myint(myint const &);
    myint & operator=(myint const &);

    int8_t x;
};
```

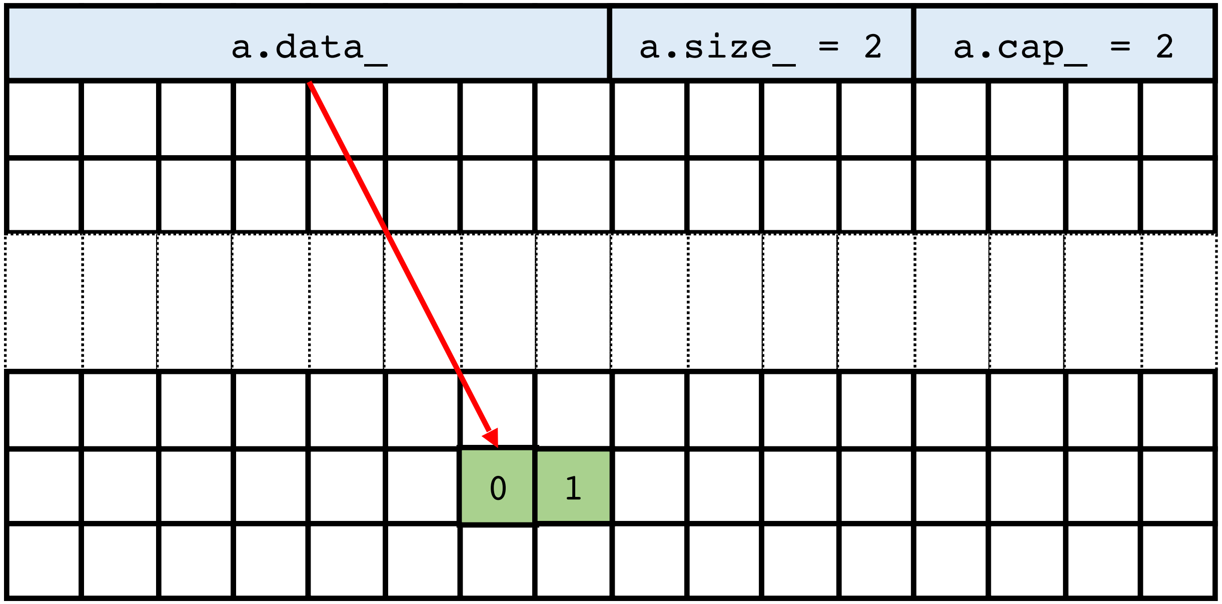
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);




    return 0;
}
```



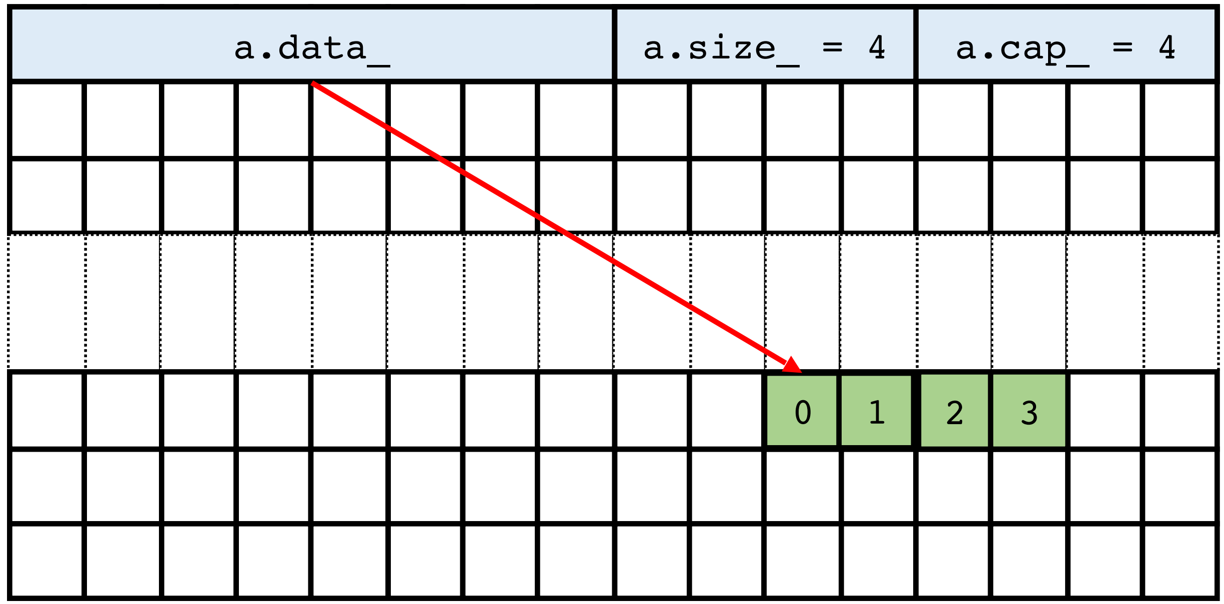
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);






    return 0;
}
```



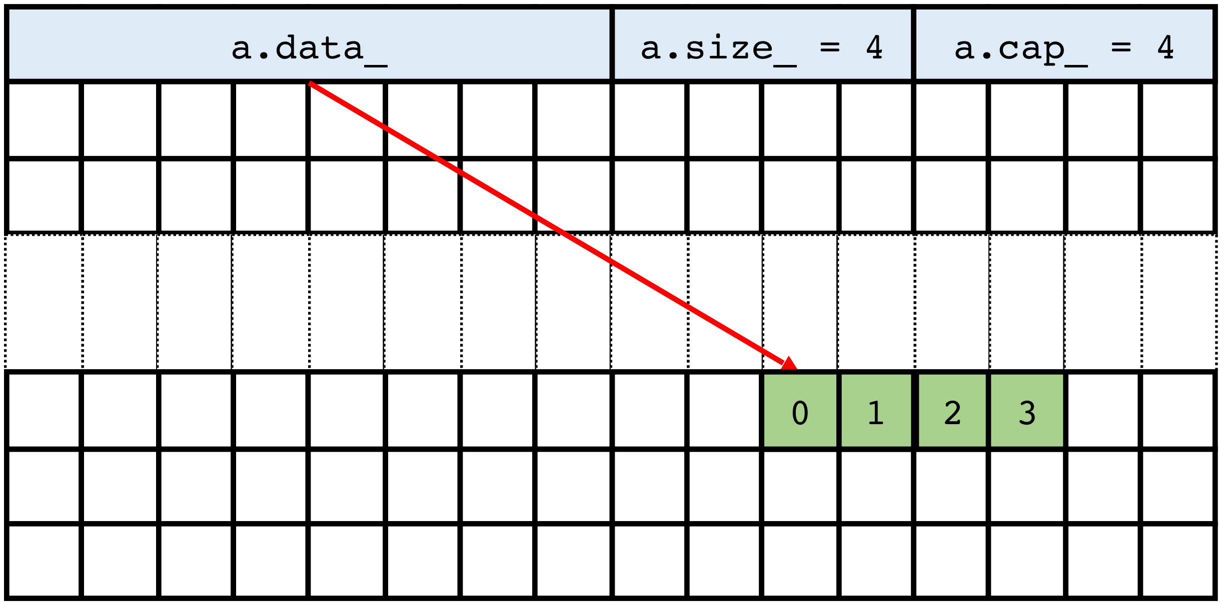
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);




    return 0;
}
```



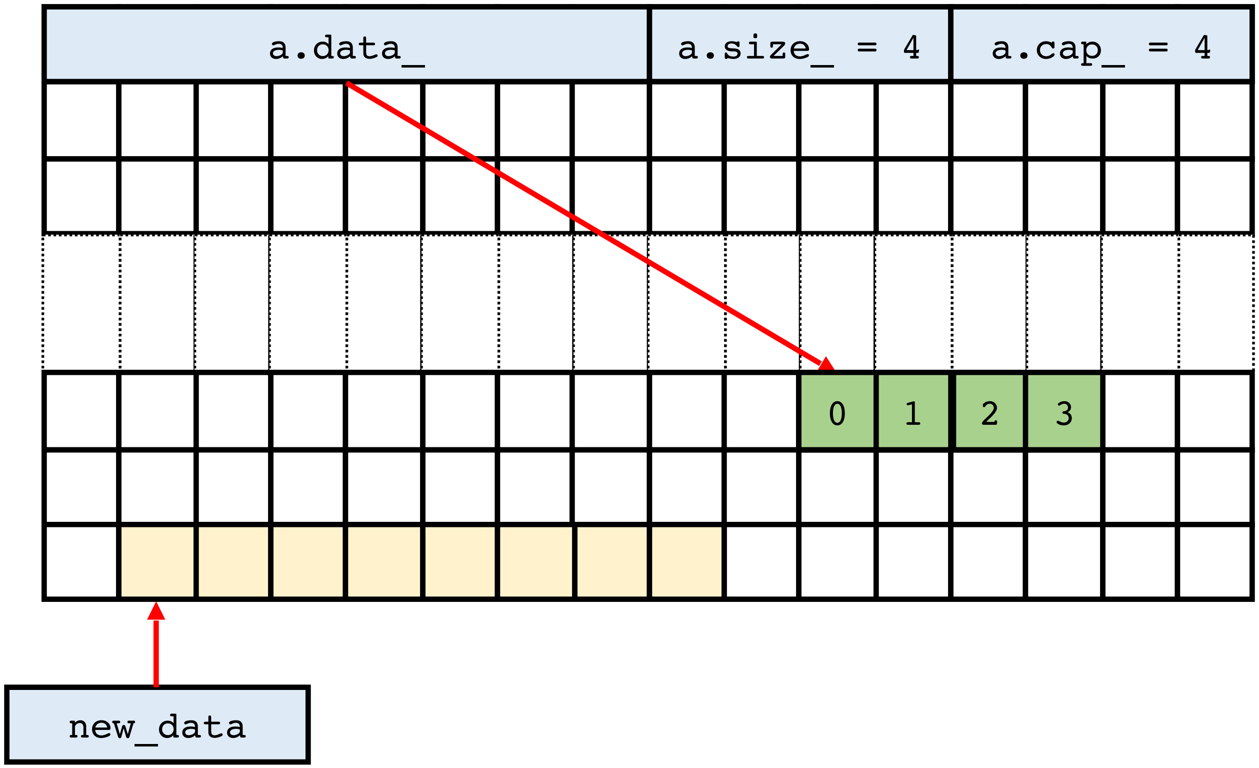
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);



    return 0;
}
```



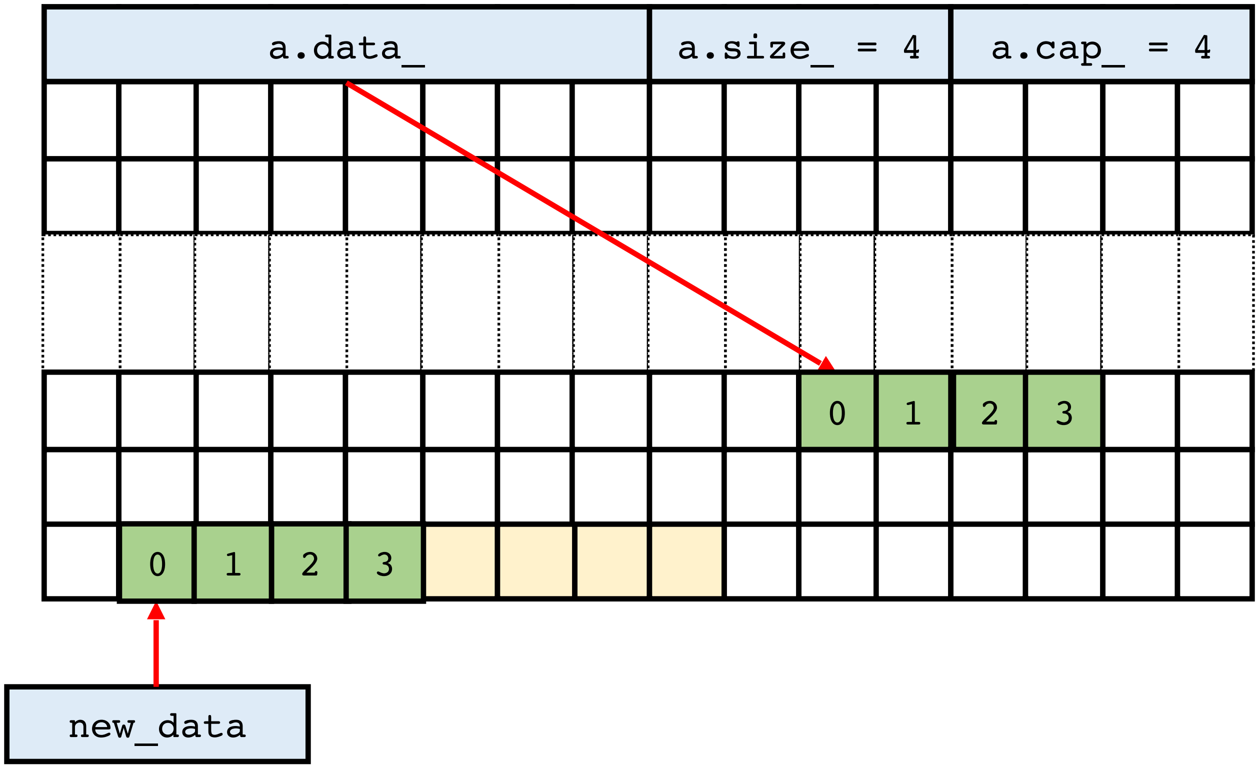
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);



    return 0;
}
```



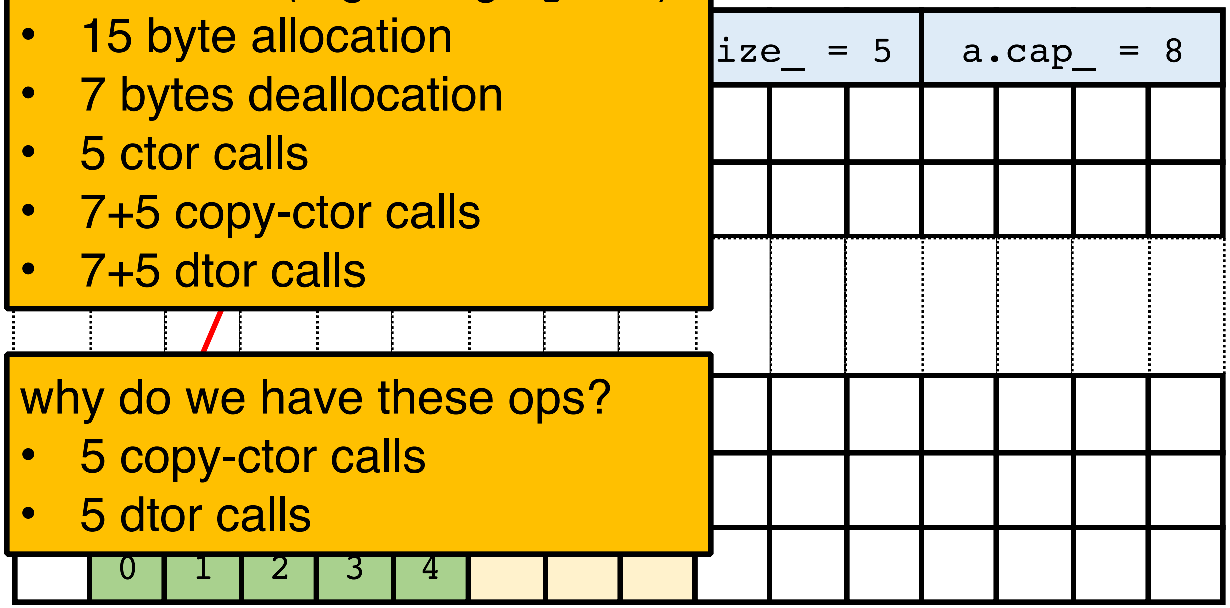
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);



    return 0;
}
```



what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);



    return 0;
}
```



this needed (regarding `myint`):

- 15 byte allocation
- 7 bytes deallocation
- 5 ctor calls
- 7+5 copy-ctor calls
- 7+5 dtor calls

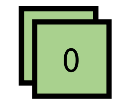
why do we have these ops?

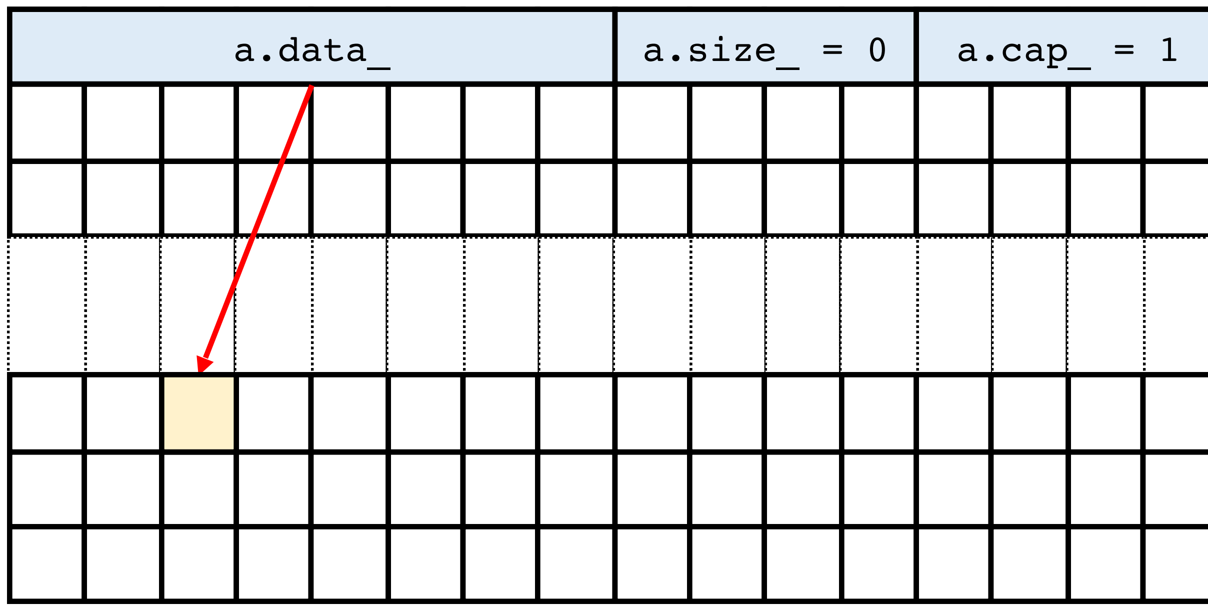
- 5 copy-ctor calls
- 5 dtor calls

additional temporary copy

```
int main() {
    vector<myint> a;

    a.push_back(0);




    return 0;
}
```

```
// if an int is passed, val is a temporary object
void push_back(value_type const & val) {
    check_capacity();
```

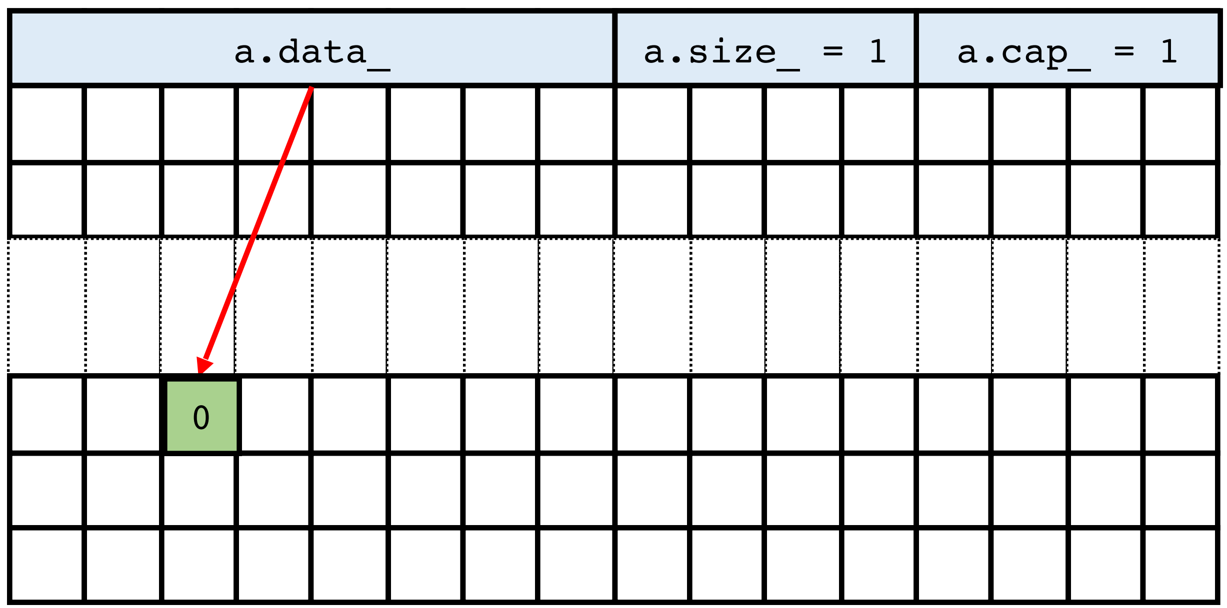




additional temporary copy

```
int main() {
    vector<myint> a;

    a.push_back(0);

    return 0;
}
```

```
// if an int is passed, val is a temporary object
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
```



we can remove these additional copy-ctor calls next week

std::vector structors

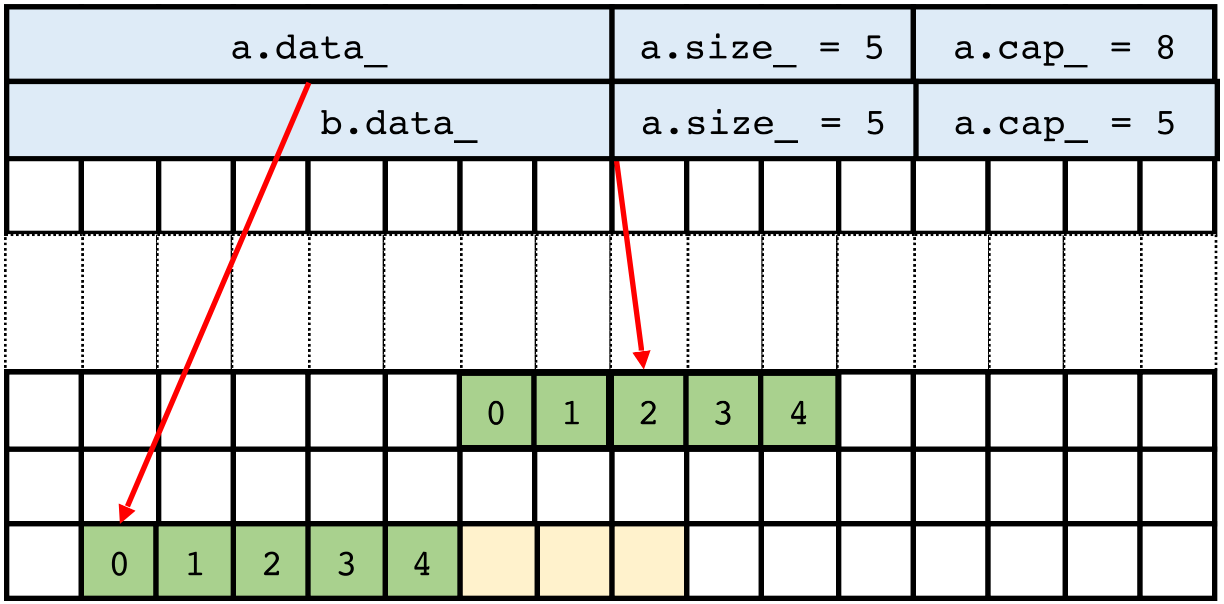
```
// constructor
vector(): data_(nullptr)
        , size_(0)
        , capacity_(0)
{}

// copy-constructor
vector(vector const & rhs): data_(Alloc::allocate(rhs.size()))
                          , size_(0)
                          , capacity_(rhs.size()) {

    for(size_type i = 0; i < rhs.size(); ++i)
        Alloc::construct(data_ + size_++, rhs[i]);
}

// destructor
~vector() {
    clear();
    Alloc::deallocate(data_, capacity());
}
```

copy-ctor

```
int main() {
    …
     // copy-ctor is called
    vector<myint> b(a);
    return 0;
}
```

this needed (regarding `myint`):

- `a.size()` byte allocation
- `a.size()` copy-ctor calls



why move?

- copying from A to B and later delete A (like in `reserve`) seems inefficient
- we want to reuse A‘s resources to construct B
- some entities are non-copyable (i.e. threads) but they are usually moveable

`std::vector` move ctor

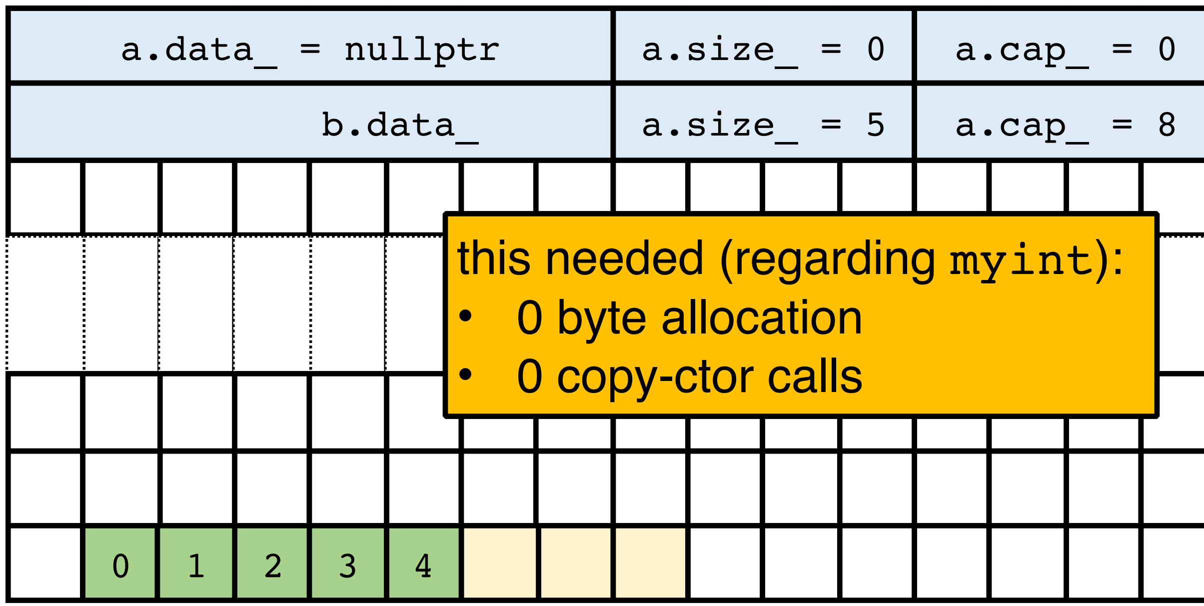
```
// constructor
vector(): data_(nullptr)
        , size_(0)
        , capacity_(0)
{}

// move-constructor (note: rhs is not const)
vector(vector && rhs):
                     data_(rhs.data_)
                   , size_(rhs.size_)
                   , capacity_(rhs.capacity_)
{
    // rhs's invariants must not be broken!
    // without the following lines we only have a shallow copy...
    rhs.data_ = nullptr;
    rhs.size_ = 0;
    rhs.capacity_ = 0;
}
```

semantic convention: the moved-from object **must be left in a valid state** (i.e. not break it‘s invariants)

move-ctor

```
int main() {
    …
     // move-ctor is called
    vector<myint> b(std::move(a));
    return 0;
}
```

`std::move` does not move anything! it casts the object to an r-value reference, so that the overload mechanism prefers the move-ctor (better match)



this needed (regarding `myint`):

- 0 byte allocation
- 0 copy-ctor calls

`std::move` implementation

```
template<typename T>
constexpr std::remove_reference_t<T> && move(T && t) noexcept {
    return static_cast<std::remove_reference_t<T> &&>(t);
}
```

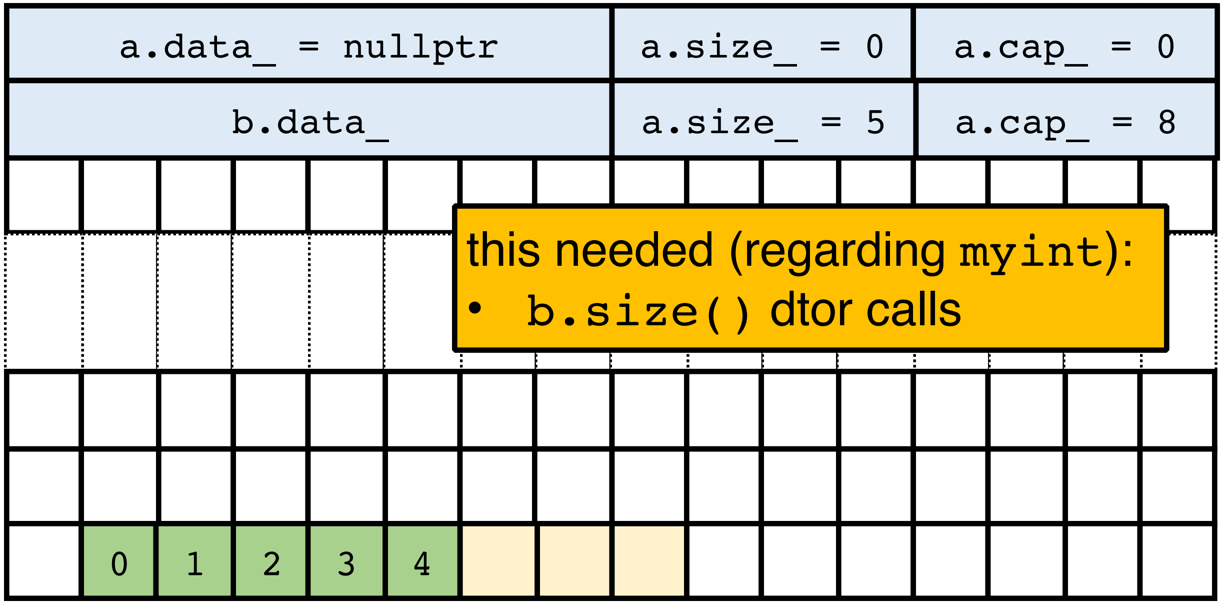
`std::move` does not move anything! it casts the object to an r-value reference, so that the overload mechanism prefers the move-ctor (better match)

std::vector move assign

```
// constructor
vector(): data_(nullptr)
        , size_(0)
        , capacity_(0)
{}

// move-assignment (note: rhs is not const)
 vector & operator=(vector && rhs) {
    using std::swap;
    swap(data_, rhs.data_);
    swap(size_, rhs.size_);
    swap(capacity_, rhs.capacity_);
    rhs.clear(); // semantically optional, the std::vector clears the rhs

    return (*this);
}
```

move-assign

```
int main() {
    …
     // move-assign is called
    vector<myint> b;
    b = std::move(a);
    return 0;
}
```



conclusion

move operation are almost always (much) faster than their corresponding copy operation!

Especially for classes that contain dynamically allocated data (e.g. vector / pimpl)

improve `reserve` with move

```
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
void check_capacity() {
    if(size() == capacity()) {
        if(capacity() == 0)
            reserve(1);
        else
            reserve(2*capacity());
    }
}
void reserve(size_type const & new_cap) {
    value_type * new_data = Alloc::allocate(new_cap);

    for(size_type i = 0; i < size(); ++i)
        Alloc::construct(new_data + i, std::move(data_[i]));

    for(size_type i = 0; i < size(); ++i)
        Alloc::destruct(data_ + i);

    Alloc::deallocate(data_, capacity());
    data_ = new_data;
    capacity_ = new_cap;
}
```

is this enough to enable the move-ctor instead of the copy-ctor?

improve `allocator` with move

```
template<typename T>
struct allocator {

    static T * allocate(std::size_t const & n);
    static void deallocate(T * p, std::size_t const & n);

    static void construct(T * p, T const & t) {
        // this is called a „placement new“. It doesn‘t allocate any memory
        new (p) T(t);
    }
    static void construct(T * p, T && t) {
        new (p) T(std::move(t));
    }

    static void destruct(T * p) { p->~T(); }
};
```

is this enough now to use the move-ctor of `myint` instead of the copy-ctor?

`myint` needs move support

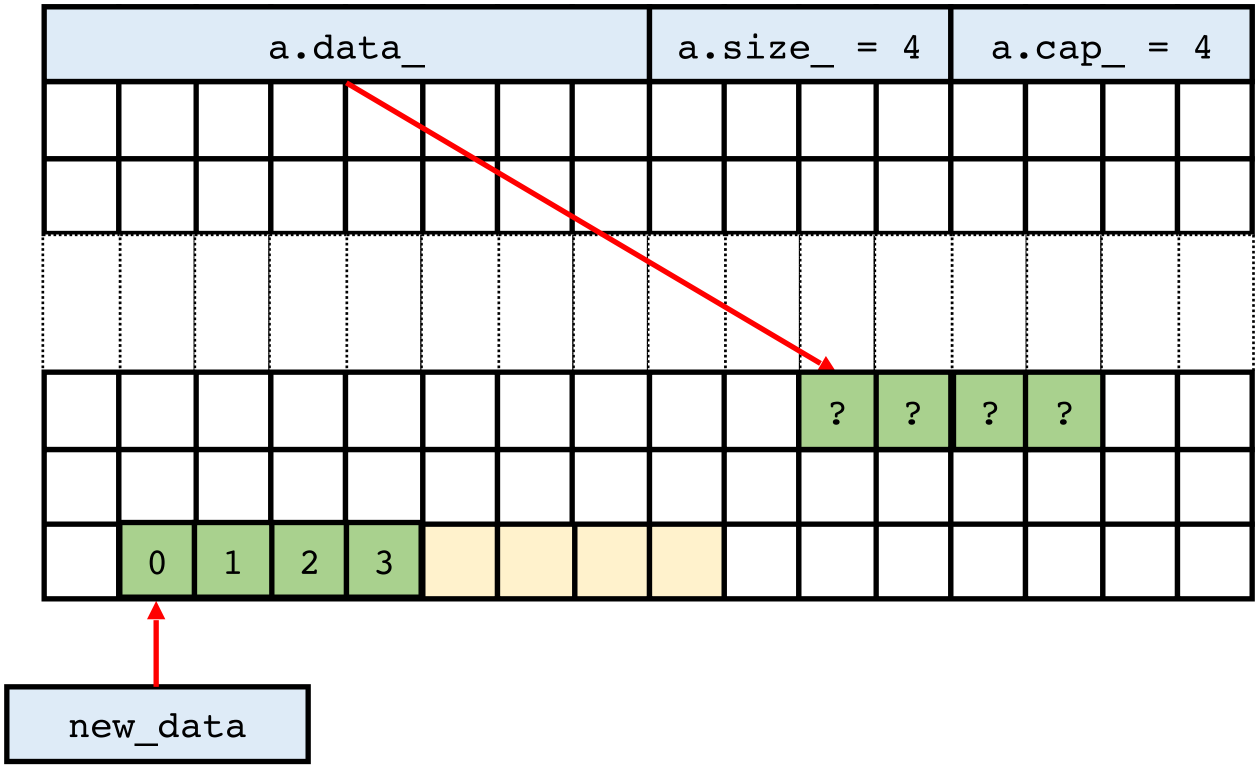
```
struct myint {
    // ctor
    myint(int const & a): x(a) {}

    // copy (disables move)
    myint(myint const &);
    myint & operator=(myint const &);

    // move
    myint(myint &&);
    myint & operator=(myint &&);

    int8_t x;
};
```

		compiler generates (if possible)					
		default ctor	***copy ctor***	***copy assign***	***move ctor***	***move assign***	***dtor***
we declare	***any ctor***	**no**	**yes**	**yes**	**yes**	**yes**	**yes**
	copy ctor	**yes**		**yes***	**no**	**no**	**yes**
	copy assign	**yes**	**yes***		**no**	**no**	**yes**
	move ctor	**yes**	**no**	**no**		**no**	**yes**
	move assign	**yes**	**no**	**no**	**no**		**yes**
	dtor	**yes**	**yes***	**yes***	**no**	**no**	

yes* should be **no** (backwards compatible)

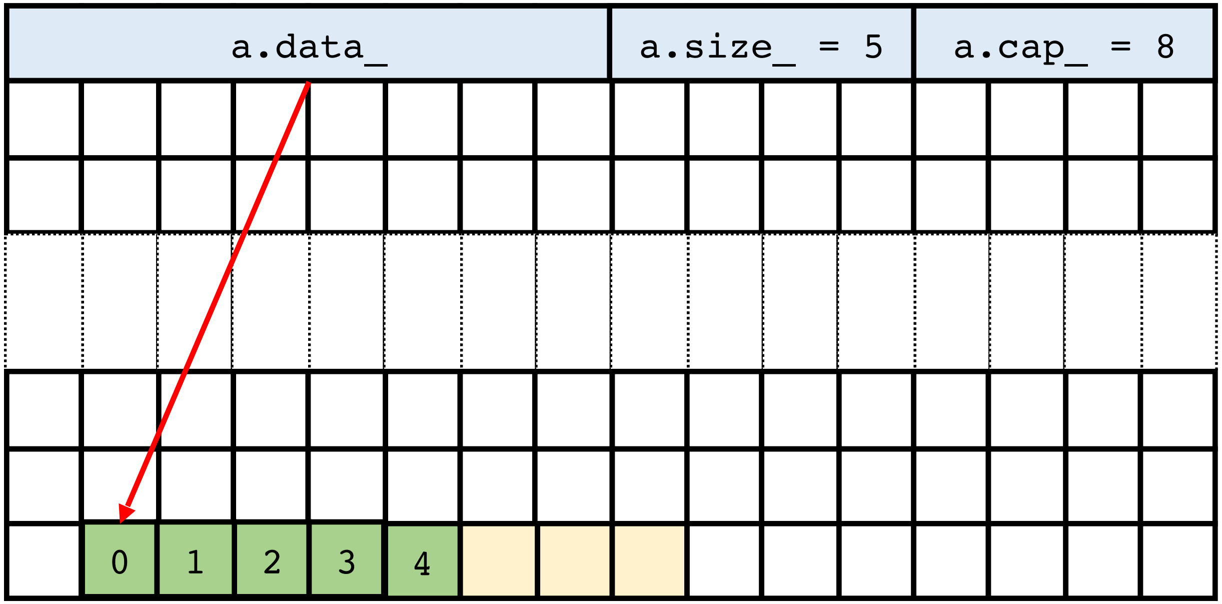
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);



    return 0;
}
```



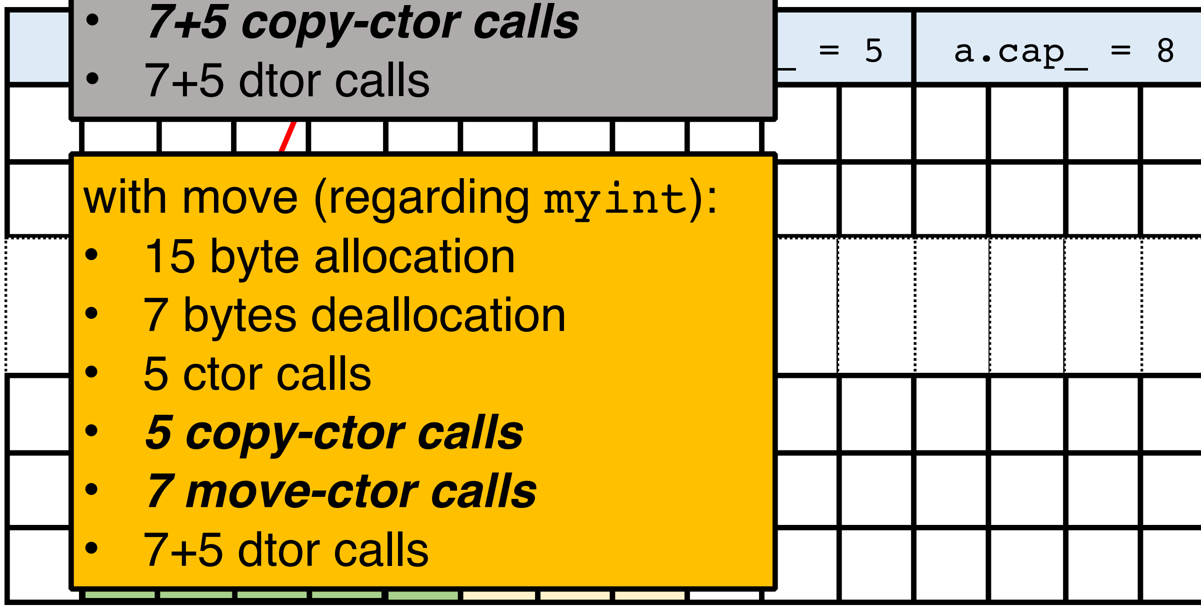
what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);



    return 0;
}
```



what happens in memory

```
int main() {
    vector<myint> a;

    a.push_back(0);
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);



    return 0;
}
```



before with copy only:

- 15 byte allocation
- 7 bytes deallocation
- 5 ctor calls
- ***7+5 copy-ctor calls***
- 7+5 dtor calls

with move (regarding `myint`):

- 15 byte allocation
- 7 bytes deallocation
- 5 ctor calls
- ***5 copy-ctor calls***
- ***7 move-ctor calls***
- 7+5 dtor calls

exception safety in `reserve`

```
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
void check_capacity() {
    if(size() == capacity()) {
        if(capacity() == 0)
            reserve(1);
        else
            reserve(2*capacity());
    }
}
void reserve(size_type const & new_cap) {
    value_type * new_data = Alloc::allocate(new_cap);

    for(size_type i = 0; i < size(); ++i)
        Alloc::construct(new_data + i, std::move(data_[i]));

    for(size_type i = 0; i < size(); ++i)
        Alloc::destruct(data_ + i);

    Alloc::deallocate(data_, capacity());
    data_ = new_data;
    capacity_ = new_cap;
}
```

why is the strong exception guarantee broken?

exception safety in `reserve`

```
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
void check_capacity() {
    if(size() == capacity()) {
        if(capacity() == 0)
            reserve(1);
        else
            reserve(2*capacity());
    }
}
void reserve(size_type const & new_cap) {
    value_type * new_data = Alloc::allocate(new_cap);

    for(size_type i = 0; i < size(); ++i)
        Alloc::construct(new_data + i, std::move_if_noexcept(data_[i]));

    for(size_type i = 0; i < size(); ++i)
        Alloc::destruct(data_ + i);

    Alloc::dealloc
    data_ = new_da
    capacity_ = new_cap;
}
```

„move if you can - copy if you must“ strategy

noexcept

```
// does not throw
// we (!) promis the compiler with noexcept that this fct does not throw
int f1(int const & a) noexcept {
    return a;
}
// could still be declared noexcept, but the compiled code
// could erase your hard disk if something is thrown ;)
int f2(int const & a) {
    throw std::runtime_error("never call this");
    return a;
}
// exception neutral function
// doesn‘t throw itself but calls something that may
int g1(int const & a) noexcept(noexcept(f1(a))) {
    return f1(a);
}
int g2(int const & a) noexcept(noexcept(f2(a))) {
    return f2(a);
}
int main() {
    std::cout << noexcept(f1(1)) << std::endl; // 1
    std::cout << noexcept(g1(1)) << std::endl; // 1
    std::cout << noexcept(f2(1)) << std::endl; // 0
    std::cout << noexcept(g2(1)) << std::endl; // 0
```

noexcept

```
// does not throw
// we (!) promis the compiler with noexcept that this fct does not throw
int f1(int const & a) noexcept {
    return a;
}
// could still be declared noexcept, but the compiled code
// could erase your hard disk if something is thrown ;)
int f2(int const & a) {
    throw std::runtime_error("never call this");
    return a;
}
// exception neutral function
// doesn't throw itself but calls s
int g1(int const & a) noexcept(noex
    return f1(a);
}
int g2(int const & a) noexcept(noex
    return f2(a);
}
int main() {
    std::cout << noexcept(f1(1)) << std::endl; // 1
    std::cout << noexcept(g1(1)) << std::endl; // 1
    std::cout << noexcept(f2(1)) << std::endl; // 0
    std::cout << noexcept(g2(1)) << std::endl; // 0
}
```

`noexcept` allows for:

- many compiler optimizations
- better exception guarantees

noexcept

```
struct myint {
    // ctor
    myint(int const & a): x(a) {}

    // copy (disables move)
    myint(myint const &) noexcept;
    myint & operator=(myint const &) noexcept;

    // move
    myint(myint const &) noexcept;
    myint & operator=(myint const &) noexcept;


    int8_t x;
};
```

use `noexcept` wherever possible. It may allows libraries to run faster/safer versions of their implementation.