

Let's Move

The Hidden Features and Traps of C++ Move Semantics

Nicolai M. Josuttis

josuttis.com

@NicoJosuttis

6/20

Let's Move
©2020 by josuttis.com

josuttis | eckstein
IT communication

Nicolai M. Josuttis

- **Independent consultant**
 - Continuously learning since 1962
- **C++:**
 - since 1990
 - ISO Standard Committee since 1997
- **Other Topics:**
 - Systems Architect
 - Technical Manager
 - SOA
 - X and OSF/Motif



@NicoJosuttis

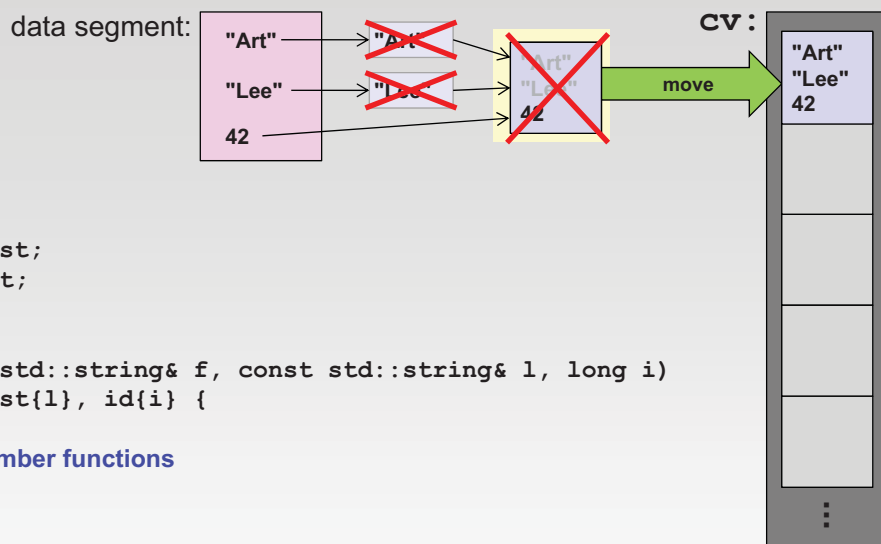
Agenda

Implement and test a **class Email** representing email addresses having good performance

3

 @NicoJosuttis

Using Generated Move Semantics



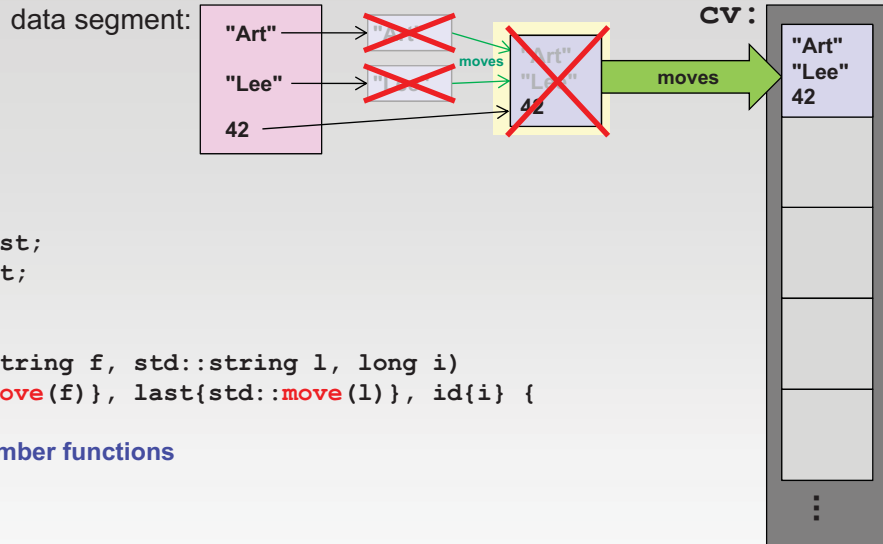
```
class Customer {
private:
    std::string first;
    std::string last;
    long        id;
public:
    Customer(const std::string& f, const std::string& l, long i)
        : first{f}, last{l}, id{i} {
    }
    ... // no special member functions
};
```

```
std::vector<Customer> cv;
...
cv.push_back(Customer{"Art", "Lee", 42}); // create customer and copy/move it into cv
```

6

 @NicoJosuttis

Using Generated and Implemented Move Semantics



```
class Customer {
private:
    std::string first;
    std::string last;
    long        id;
public:
    Customer(std::string f, std::string l, long i)
        : first{std::move(f)}, last{std::move(l)}, id{i} {
    }
    ... // no special member functions
};
```

```
std::vector<Customer> cv;
...
cv.push_back(Customer{"Art", "Lee", 42}); // create customer and copy/move it into cv
```

By-Value and std::move() is almost the Best

```
std::string s = "Joe";
Cust c{"Joe", "Fix"}; // at least 2 mallocs
Cust d{s, "Fix"}; // at least 2 mallocs
Cust e{std::move(s), "Fix"}; // at least 1 malloc
```

```
class Cust {
    Cust(const std::string& f, const std::string& l)
        : first{f}, last{l} {
    }

    Cust(std::string f, std::string l)
        : first{std::move(f)}, last{std::move(l)} {
    }

    Cust(const std::string& f, const std::string& l)
        : first{f}, last{l} {
    }
    Cust(std::string&& f, std::string&& l)
        : first{std::move(f)}, last{std::move(l)} {
    }
    Cust(const std::string& f, std::string&& l)
        : first{f}, last{std::move(l)} {
    }
    Cust(std::string&& f, const std::string& l)
        : first{std::move(f)}, last{l} {
    }
};
```

10 mallocs (4cr + 6cp)

5 mallocs (4cr + 1cp + 7mv)

5 mallocs (4cr + 1cp + 5mv)

By-Value and std::move() is almost the Best

```
std::string s = "Joe";
Cust c{"Joe", "Fix"}; // at least 2 mallocs
Cust d{s, "Fix"}; // at least 2 mallocs
Cust e{std::move(s), "Fix"}; // at least 1 malloc
```

```
class Cust {
    Cust(const std::string& f, const std::string& l)
        : first{f}, last{l} {
    }
    Cust(std::string&& f, std::string&& l)
        : first{std::move(f)}, last{std::move(l)} {
    }
    Cust(const std::string& f, std::string&& l)
        : first{f}, last{std::move(l)} {
    }
    Cust(std::string&& f, const std::string& l)
        : first{std::move(f)}, last{l} {
    }
    Cust(const char* f, const char* l)
        : first{f}, last{l} {
    }
    Cust(const char* f, const std::string& l)
        : first{f}, last{l} {
    }
    Cust(const char* f, std::string&& l)
        : first{f}, last{std::move(l)} {
    }
    Cust(const std::string& f, const char* l)
        : first{f}, last{l} {
    }
    Cust(std::string&& f, const char* l)
        : first{std::move(f)}, last{l} {
    }
};
```

10 mallocs (4cr + 6cp)

5 mallocs (4cr + 1cp + 7mv)

5 mallocs (4cr + 1cp + 5mv)

5 mallocs (4cr + 1cp + 1mv)

Compare Ways to Initialize Members

```
std::string s = "Joe";
Cust c{"Joe", "Fix"}; // at least 2 mallocs
Cust d{s, "Fix"}; // at least 2 mallocs
Cust e{std::move(s), "Fix"}; // at least 1 malloc
```

Constructors should **move initialize** expensive members from **by-value parameters**

```
class Cust {
    std::string first;
    std::string last;
    int idx;
public:
    ... // 1 - 9 constructors
};
```

- classic: 10 mallocs (4cr + 6cp)
- move: 5 mallocs (4cr + 1cp + 7mv)
- allref: 5 mallocs (4cr + 1cp + 5mv)
- all: 5 mallocs (4cr + 1cp + 1mv)

```
class Cust {
    std::string first;
    std::string last;
    int idx;
    std::array<Coord,100> val;
public:
    ... // 1 - 9 constructors
};
```

| | Platform A | Platform B | Platform C |
|--------------------|------------|------------|------------|
| classic: | 8.29763 | 13.2746 | 4.95914 |
| move: | 5.78400 | 5.9336 | 2.74172 |
| allref: | 5.76791 | 5.8211 | 2.41148 |
| all: | 5.75993 | 5.7886 | 2.31567 |
| With array: | | | |
| classic: | 11.03440 | 15.1944 | 7.68108 |
| move: | 8.73324 | 8.6309 | 4.89639 |
| allref: | 8.62878 | 8.5899 | 4.81283 |
| all: | 8.74176 | 8.2674 | 5.38340 |

Getters by Value

```

class Cust {
private:
    std::string first;
    std::string last;
    int         id;

public:
    Cust(std::string f, std::string l, int i)
        : first{std::move(f)}, last{std::move(l)}, id{i} {}

    void setLast(const std::string& s) { last = s; }

    std::string getLast() const {
        return last;
    }
    ...
};

```

```

Cust readCust();
using namespace std;

Cust c{"Joe", "Fix", 42};
auto s = c.getLast();           // OK
cout << c.getLast();           // slow
cout << readCust().getLast();  // slow

vector<Cust> coll;
for (const auto& c : coll) {
    if (c.getLast().empty()) { // slow
        ...
    }
}

```

```

std::string getLast() const {
    return last;
}

```

11

 @NicoJosuttis

Getters by Reference ?

```

class Cust {
private:
    std::string first;
    std::string last;
    int         id;

public:
    Cust(std::string f, std::string l, int i)
        : first{std::move(f)}, last{std::move(l)}, id{i} {}

    void setLast(const std::string& s) { last = s; }

    const std::string& getLast() const {
        return last;
    }
    ...
};

```

```

...
Cust c{"Joe", "Fix", 42};
auto s = c.getLast();           // OK
cout << c.getLast();           // fast
cout << readCust().getLast();  // fast

vector<Cust> coll;
for (const auto& c : coll) {
    if (c.getLast().empty()) { // fast
        ...
    }
}

// loop over chars of the name:
for (char c : readCust().getLast()) {
    cout << c;
}

```

```

const std::string& getLast() const {
    return last;
}

```

**Core dump
at best**

12

 @NicoJosuttis

Getters and Range-Based for Loop

Range-based for loop:

```

auto&& _rg = readCust().getLast(); // lifetime of return value of readCust() ends here
for (auto _pos=_rg.begin(), _end=_rg.end(); _pos!=_end; ++_pos ) {
    char c = *_pos;
    cout << c;
}

```

```

public:
    Cust(std::string f, std::string
        : first{std::move(f)}, last{std
    }

    void setLast(const std::string&
        last = s;
    }

    if (c.getLast().empty()) { // fast
        ...
    }

    // loop over chars of the name:
    for (char c : readCust().getLast()) {
        cout << c; // Fatal Runtime ERROR
    }

```

```

const std::string& getLast() const {
    return last;
}

```

```
};
```

13

 @NicoJosuttis

Overload Getters by Reference Qualifiers

```

class Cust {
private:
    std::string first;
    std::string last;
    int id;

public:
    Cust(std::string f, std::string
        : first{std::move(f)}, last{std
    }

    void setLast(const std::string&
        last = s;
    }

```

```

...
Cust c{"Joe","Fix",42};
auto s = c.getLast(); // OK
cout << c.getLast(); // fast
cout << readCust().getLast(); // fast

vector<Cust> coll;
for (const auto& c : coll) {
    if (c.getLast().empty()) { // fast
        ...
    }
}

// loop over chars of the name:
for (char c : readCust().getLast()) {
    cout << c; // OK
}

```

```

std::string& getLast() && {
    return std::move(last);
}
const std::string& getLast() const& {
    return last;
}

```

```
};
```

for **rvalues** (objects with no name and objects marked with `move()`)
return **by value**

for **lvalues** (objects with a name)
return **by reference**

14

 @NicoJosuttis

Moved-from States

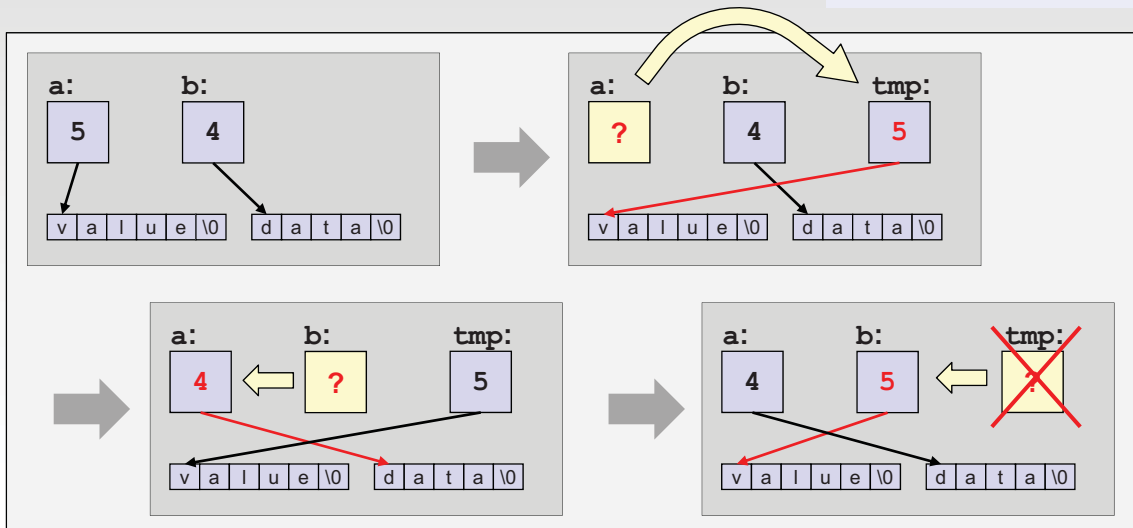
- **Requirements** by the C++ Standard Library
 - A moved-from object is **nothing special**
 - All requirements also apply to moved-from objects
 - Usually: destruction and assignment
 - But `sort()` might call `<` for a moved-from object
 - Moved-from objects should **also support all requirements**
- **Guarantees** by the C++ Standard Library
 - Moved-from objects are in a **valid but unspecified state**
 - No invariants broken
 - All operations work as expected
 - These guarantees fulfill the requirements
- **Ideally you give the same guarantees for your types**

under discussion by the C++ committee

Requirements According to Swap

- **Assign** new values to moved-from objects
 - Note: self-swap should work
- **Destroy** moved-from objects

```
void swap(T& a, T& b) {
    tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
```



Using the Guarantees for Moved-from Objects

```
std::vector<std::string> coll;
...
std::string row;
while (std::getline(myStream, row)) {
    coll.push_back(std::move(row));    // move the line into the vector
}
```

```
std::stack<int> stk;
...
foo(std::move(stk));    // stk gets unspecified state
stk.push(42);
... // do something else without using stk
int i = stk.top();
assert(i == 42);    // should never fail
```

17

 @NicoJosuttis

Invariants Broken by Move Semantics

```
class Email {
private:
    std::string value;
public:
    Email(const std::string& val)
        : value{val} {
        assert(value.find('@') != std::string::npos);
    }
    void setValue(const std::string& val) {
        value = val;
        assert(value.find('@') != std::string::npos);
    }
    std::string getValue() const {
        return value;
    }
    ...
};
```

Invariant:

Objects always have an email address
(`getValue()` has a string with a @)

```
Email e1{"nico@josuttis.de"};
assert(e1.getValue().size() > 0); // holds

Email e2{getEmail()};           // moves
assert(e2.getValue().size() > 0); // holds

Email e3{std::move(e1)};       // moves
assert(e1.getValue().size() > 0); // fails
```

- Moved-from objects can be in invalid/inconsistent states
- Only a problem when using objects after `std::move()`

18

 @NicoJosuttis

Deleting Generated Move Semantics

```

class Email {
private:
    std::string value;
public:
    Email(const std::string& val)
        : value{val} {
        assert(value.find('@') != std::string::npos);
    }
    void setValue(const std::string& val) {
        value = val;
        assert(value.find('@') != std::string::npos);
    }
    std::string getValue() const {
        return value;
    }
    ...
    // disable generated move operations:
    Email(Email&&) = delete;
    Email& operator=(Email&&) = delete;
};

```

• Don't use =delete to disable move semantics

```

Email e1{"nico@josuttis.de"};
assert(e1.getValue().size() > 0); // holds

Email e2{getEmail()}; // compile-time error

Email e3{std::move(e1)}; // compile-time error

```

19

 @NicoJosuttis

Disabling Generated Move Semantics

```

class Email {
private:
    std::string value;
public:
    Email(const std::string& val)
        : value{val} {
        assert(value.find('@') != std::string::npos);
    }
    void setValue(const std::string& val) {
        value = val;
        assert(value.find('@') != std::string::npos);
    }
    std::string getValue() const {
        return value;
    }
    ...
    // force not to have generated move operations:
    Email(const Email&) = default;
    Email& operator=(const Email&) = default;
};

```

• Use =default for special copy members to disable move semantics

```

Email e1{"nico@josuttis.de"};
assert(e1.getValue().size() > 0); // holds

Email e2{getEmail()}; // copies
assert(e2.getValue().size() > 0); // holds

Email e3{std::move(e1)}; // copies
assert(e1.getValue().size() > 0); // holds

```

20

 @NicoJosuttis

"Rule of Five"

- **If one** of the following 5 special member functions **is declared**:

- Copy constructor
- Move constructor
- Copy assignment operator
- Move assignment operator
- Destructor

think carefully about

you should ~~declare~~ **all** of them

- **"Declared"** means one of the following:

- Implemented: `{...}`
- Declaring as being **defaulted**: `=default`
- Declaring as being **deleted**: `=delete`

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rc-five>

21

 @NicoJosuttis

Implementing Move Semantics

```
class Email {
private:
    std::string value;
    bool movedFrom{false};
public:
    Email(const std::string& val)
        : value{val} {
        assert(value.find('@') != std::string::npos);
    }
    ...
    std::string getValue() const {
        return movedFrom ? "" : value; // or assert or throw for guaranteed behavior
    }
    ...
    // implement move operations and enable copying:
    Email(Email&& e)
        : value{e.value}, movedFrom{e.movedFrom} {
        e.movedFrom = true;
    }
    ...
    Email(const Email&) = default;
    Email& operator=(const Email&) = default;
};
```

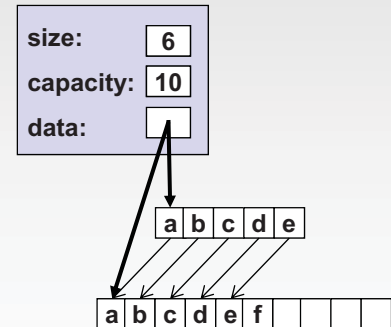
22

 @NicoJosuttis

Exception Safety Guarantee for Vector's `push_back()`

- In C++98/C++03 the guarantee is possible because:

- Reallocation is done by the following steps:
 - allocate new memory
 - assign new value
 - copy old elements (element by element)
 - point of no rollback -----
 - assign new memory to internal pointer
 - delete old elements and free old memory
 - update size and capacity



23

@NicoJosuttis

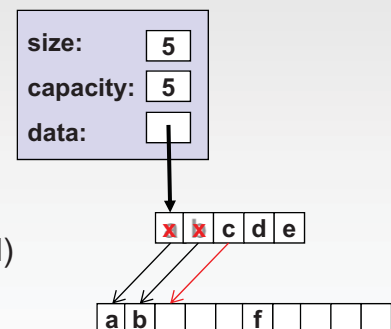
Exception Safety Guarantee for Vector's `push_back()`

- Reallocation with move semantics breaks the strong exception guarantee

- Moving elements might throw but we can't always roll back

- We can't

- silently break the strong exception guarantee
 - Existing code would be broken
- replace `push_back()` by something new
 - Too much use
- require that move constructors don't throw
 - Even the moved-from state (valid but unspecified) might need memory



- So:

`std::vector<>` moves only if it's safe

- with guarantee that the move constructor does not throw

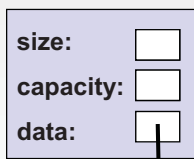
24

@NicoJosuttis

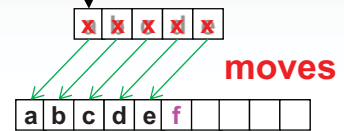
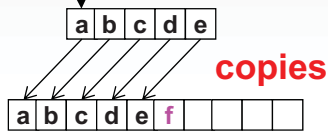
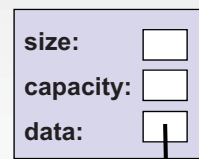
Vector Reallocation and noexcept

```
class Cust {
private:
    std::string name;
public:
    ...
    Cust(const Cust& c) // copy constructor
        : name{c.name} {
    }
    Cust(Cust&& c) // move constructor
        : name{std::move(c.name)} {
    }
    ...
};
```

```
class Cust {
private:
    std::string name;
public:
    ...
    Cust(const Cust& c) // copy constructor
        : name{c.name} {
    }
    Cust(Cust&& c) noexcept // move constructor
        : name{std::move(c.name)} {
    }
    ...
};
```



```
std::vector<Cust> coll;
...
coll.push_back(f);
```



Example With and Without noexcept

```
#include <vector>
#include <string>
#include <chrono>
#include <iostream>

// string wrapper with move constructor:
class Str
{
private:
    std::string s;
public:
    Str()
        : s(100, 'a') {
    }

    Str(const Str&) = default;

    Str(Str&& x) noexcept
        : s{std::move(x.s)} {
    }
};
```

don't use braces here

noexcept optional
measure with and without

```
int main()
{
    using namespace std::chrono;

    // create vector of 1 Million wrapped strings:
    std::vector<Str> v;
    v.resize(1'000'000);

    // measure time to realloc:
    auto t0 = steady_clock::now();
    v.reserve(c.capacity() + 1);
    auto t1 = steady_clock::now();

    duration<double, std::milli> d{t1 - t0};
    std::cout << d.count() << " ms\n";
}
```

with noexcept
10 times faster than
without noexcept

Program by Howard Hinnant in [c++std-lib-35804] (slightly modified)

Example With and Without noexcept

```
#include <vector>
#include <string>
#include <chrono>
#include <iostream>

// string wrapper with move constructor:
class Str
{
private:
    std::string s;
public:
    Str()
        : s(100, 'a') {
    }

    Str(const Str&) = default;

    Str(Str&& x) noexcept
        : s{std::move(x.s)} {
    }
};
```

Different platforms!

Program by Howard Hinnant in [c++std-lib-35-04] (slice)

```
int main()
{
    using namespace std::chrono;

    // create vector of 1 Million wrapped strings:
    std::vector<Str> v;
    v.resize(1'000'000);

    // measure time to realloc:
    auto t0 = steady_clock::now();
    v.reserve(c.capacity() + 1);
    auto t1 = steady_clock::now();

    duration<double, std::milli> dft1 = t0;
```

| | Reallocation of # Elements | Without noexcept | With noexcept |
|---------|----------------------------|----------------------|---------------|
| clang++ | 1,000,000 | 228 – 239 ms | 19 – 22 ms |
| g++49 | 1,000,000 | 15 – 31 ms | 0 ms |
| g++49 | 10,000,000 | 234 – 249 ms | 15 – 31 ms |
| VS2015 | 1,000,000 | Bug in VC++15 ~15 ms | ~15 ms |
| VS2017 | 1,000,000 | 170 – 190 ms | 18 – 22 ms |

27

 @NicoJosuttis

Implementing Move Semantics with noexcept

```
class Email {
private:
    std::string value;
    bool movedFrom{false};
public:
    Email(const std::string& val)
        : value{val} {
        assert(value.find('@') != std::string::npos);
    }
    ...
    std::string getValue() const {
        return movedFrom ? "" : value; // or assert or throw for guaranteed behavior
    }
    ...
    // implement move operations and enable copying:
    Email(Email&& e) noexcept(std::is_nothrow_move_constructible<std::string>::value)
        : value{e.value}, movedFrom{e.movedFrom} {
        e.movedFrom = true;
    }
    ...
    Email(const Email&) = default;
    Email& operator=(const Email&) = default;
};
```

28

 @NicoJosuttis

Lessons Learned

- **Your types automatically provide move semantics**
- **For expensive members**
 - Initializing constructors **take by value and move ()**
 - Overload getters for **&&** and **const&**
- **Moved-from objects**
 - are **nothing special** for the C++ standard library
 - Functions expect all requirements also to work for moved-from objects
 - should not break **invariants**
 - Ideally in a "valid but unspecified state"
- **To disable move semantics =default other special members**
 - breaks "Rule of 5"
- **Use noexcept when implementing special move members**

29

 @NicoJosuttis

Contact



Nicolai M. Josuttis

www.josuttis.com

nico@josuttis.com

 @NicoJosuttis



30

 @NicoJosuttis