Code at: josuttis.com/download/yow

Let's Move

Hidden Features and Traps of C++ Move Semantics

Nicolai M. Josuttis

josuttis.com

@NicoJosuttis

Yow! 2020

C++ Move

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 Cust` representing customers having good performance

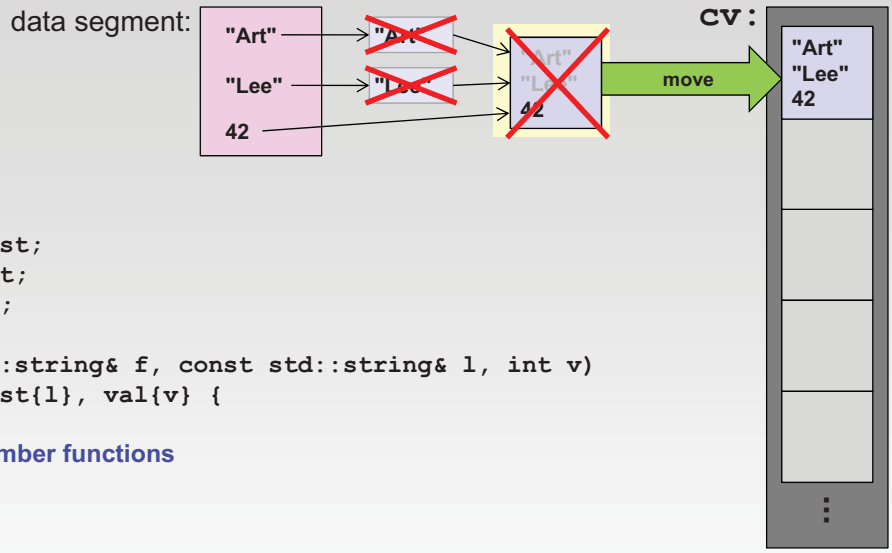
3 @NicoJosuttis

Class `Cust`

```
class Cust {
 private:
  std::string first;   // first name
  std::string last;    // last name
  int         val;     // some value (e.g. a year)

 public:
  Cust(const std::string& f, const std::string& l, int v)
   : first{f}, last{l}, val{v} {
     assert(!last.empty());   // ensure last name is never empty
  }

  void setLast(const std::string& s) {
    assert(!s.empty());        // ensure last name is never empty
    last = s;
  }

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

4 @NicoJosuttis

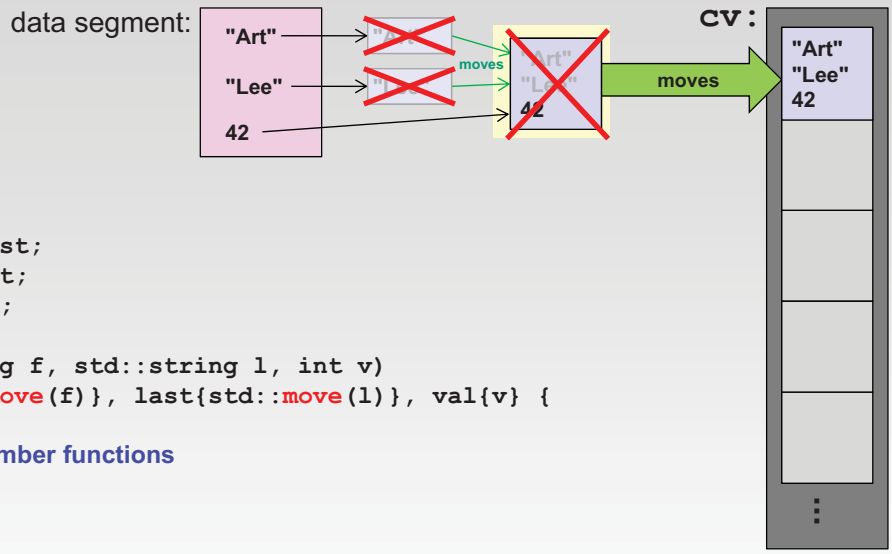
Using Generated Move Semantics




```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;
 public:
  Cust(const std::string& f, const std::string& l, int v)
   : first{f}, last{l}, val{v} {
  }
  ... // no special member functions
};
```

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

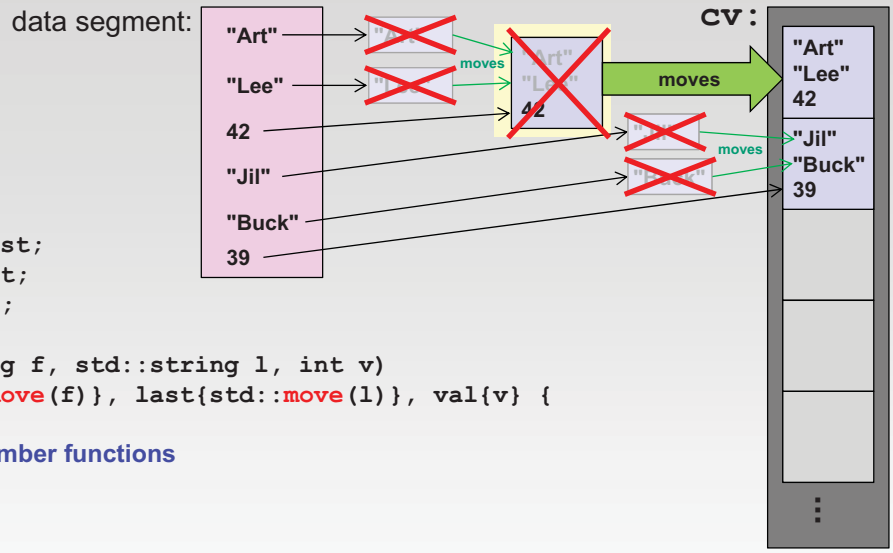

Using Generated and Implemented Move Semantics




```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;
 public:
  Cust(std::string f, std::string l, int v)
   : first{std::move(f)}, last{std::move(l)}, val{v} {
  }
  ... // no special member functions
};
```

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

Using Generated and Implemented Move Semantics




```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;
 public:
  Cust(std::string f, std::string l, int v)
   : first{std::move(f)}, last{std::move(l)}, val{v} {
  }
  ... // no special member functions
};
```

```
std::vector<Cust> cv;
...
cv.push_back(Cust{"Art","Lee",42});   // create customer and copy/move it into cv
cv.emplace_back("Jil","Buck",39);     // create new customer inside cv
```


@NicoJosuttis

Compare Ways to Initialize Members

```
Cust c{"Joe", "Fox"};              // at least 2 mallocs
Cust d{str, "Fox"};                // at least 2 mallocs
Cust e{std::move(str), "Fox"};     // at least 1 malloc
```

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

10 mallocs (4cr + 6cp)

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

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

```
  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} {
  }
};
```

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

@NicoJosuttis

Compare Ways to Initialize Members

```
Cust c{"Joe", "Fox"};            // at least 2 mallocs
Cust d{str, "Fox"};              // at least 2 mallocs
Cust e{std::move(str), "Fox"};   // 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

```
Cust c{"Joe", "Fox"};            // at least 2 mallocs
Cust d{str, "Fox"};              // at least 2 mallocs
Cust e{std::move(str), "Fox"};   // at least 1 malloc
```

Move initialize expensive members from by-value parameters

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

With array:

	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
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         val;

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

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

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

```
Cust readCust();
using namespace std;

Cust c{"Joe","Fox",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
    …
  }
}
```


Getters by Reference

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;

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

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

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

```
…
Cust c{"Joe","Fox",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
    …
  }
}
```

✓

But...

Getters by Reference ?

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;

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

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

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

```
...
Cust c{"Joe","Fox",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;
}
```

Core dump at best

Getters and Range-Based `for` Loop

Range-based `for` loop:

```
reference 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 l,
   : first{std::move(f)}, last{std::
  }

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

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

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

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

See http://wg21.link/p2012 for details

Overload Getters by Reference Qualifiers

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;

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

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

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

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

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
}
```

for rvalues
(temporaries without name, `move()`)
return by value

for lvalues
(objects with a name)
return by reference

Getters with Reference Qualifiers and Move Semantics

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;

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

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

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

```
...
Cust c{"Joe","Fox",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
}
```

Using Class `Cust`

```
std::vector<Cust> coll{{"Salvador", "Dali", 1904},
                       {"Michelangelo", 1475},
                       {"Claude", "Monet", 1840},
                       {"Pablo", "Picasso", 1881},
                      };

print(coll);


std::remove_if(coll.begin(), coll.end(),
               [](const auto& c) {
                 return c.getYear() < 1800;
               });

print(coll);
```

Output:

```
Salvador Dali 1904
 Michelangelo 1475
Claude Monet 1840
Pablo Picasso 1881


Salvador Dali 1904
Claude Monet 1840
Pablo Picasso 1881
  1881
```


Moved-from States

- **Guarantees by the C++ Standard Library**
  - Moved-from objects are in a *valid but unspecified state*
    - No invariants broken
    - All operations work as expected

```
...
foo(std::move(obj));
// obj is in moved-from state
...
```

- **Requirements by the C++ Standard Library**
  - A moved-from object is nothing special
  - Moved-from objects should also support all requirements
    - At least: destruction and assignment
    - But e.g. `sort()` might call < for a moved-from object

Ensure moved-from objects are valid
- destructible
- assignable
- support for all other operations

Deleting Generated Move Semantics

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;

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

  ...

  // disable generated move operations:
  Cust(Cust&&) = delete;
  Cust& operator=(Cust&&) = delete;

  ...
};
```

Never =delete move members

```
Cust getCustomer();      // forward declaration
std::vector<Cust> coll;

coll.push_back(getCustomer()); // ERROR
```


Deleting Generated Move Semantics

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;

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

  ...

  // force not to have generated move operations:
  Cust(const Cust&) = default;
  Cust& operator=(const Cust&) = default;

  ...
};
```

=default copy members to disable move semantics

```
Cust getCustomer();      // forward declaration
std::vector<Cust> coll;

coll.push_back(getCustomer()); // copies
```

"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`

http://github.com/isocpp/CppCoreGuidelines/blob/master/**CppCoreGuidelines.md#Rc-five**

Implementing Move Semantics

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;

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

  ...

  // implement move operations and enable copying:
  Cust(Cust&& c)
   : first{std::move(c.first)}, last{std::move(c.last)}, val{c.val} {
     c.val = 0;
  }
  ... // also implement move assignment
  Cust(const Cust&) = default;
  Cust& operator=(const Cust&) = default;

  ...
};
```

```
Cust getCustomer();      // forward declaration
std::vector<Cust> coll;

coll.push_back(getCustomer()); // moves
coll.push_back(coll[0]);       // copies
```

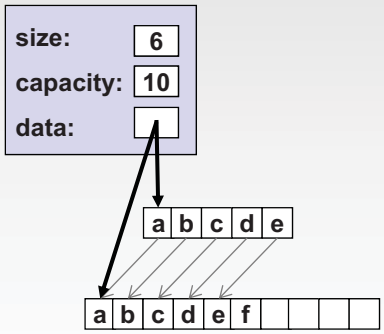
Exception Safety Guarantee for `push_back()`

```
std::vector<std::string> coll{"a", "b", "c", "d", "e"};
coll.push_back("f");
```

Strong exception guarantee

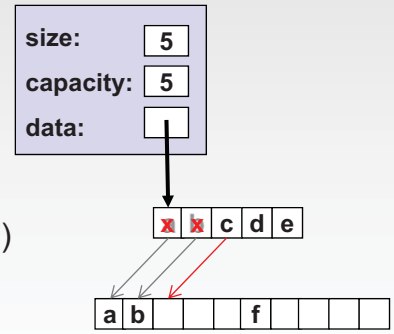
based on

- **we can roll back**:
  - Allocate new memory
  - Insert new value
  - Copy existing elements (element by element)
- **we don't throw**:
  - Delete old elements and free old memory
    - Requirement: destructors do not throw
  - Assign new memory to internal pointer
  - Update size and capacity



Exception Safety Guarantee for `push_back()`

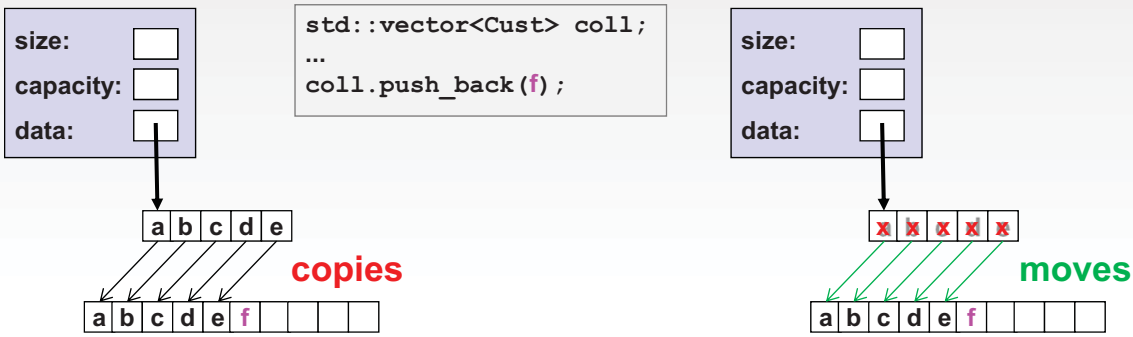
- **Reallocation with move semantics**
- **breaks the strong exception guarantee**
  - Rolling back a move might fail
- **We can't**
  - silently break the strong exception guarantee
    - Existing code would be broken
  - replace **`push_back()`** by something new
    - Too much use
  - require that move constructors don't throw
    - Even the moved-from state (valid but unspecified) might need memory
- **So:**
  std::vector<> moves elements only if it's safe
  - with guarantee that the move constructor does not throw



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)} {
   }
   ...
};
```



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') { // init with 100 'a'
  }

  Str(const Str&) = default;

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

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(v.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)

Performance 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') { // init with 100 'a
  }

  Str(const Str&) = default;

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

```
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(v.capacity() + 1);
  auto t1 = steady_clock::now();

  duration<double, std::milli> d{t1 - t0};
```

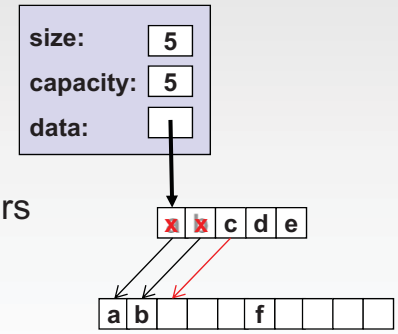
Different platforms!

Program by Howard Hinnant in [c++std-lib-3[illegible]204] (slig

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	~15 ms (Bug in VC++15)	~15 ms
VS2017	1,000,000	170 – 190 ms	18 – 22 ms

Exception Safety Guarantee for push_back()

- **On reallocation,**
  std::vector<> moves elements only if it's safe
  - If the move constructor guarantees not to throw
    - Declare that it doesn't throw
- **But throwing might depend on**
  - members
  - base classes
- **and we might not know their types**
  - Class templates can have any type for members

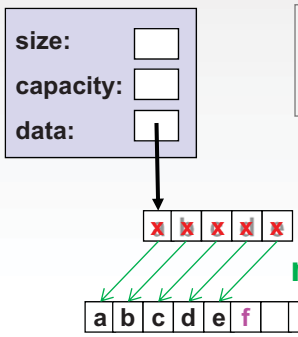


=> We need a way to specify a
conditional guarantee not to throw:
- Keyword noexcept

Vector Reallocation and Conditional noexcept

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;
 public:
  ...
  Cust(Cust&& c) noexcept(std::is_nothrow_move_constructible<std::string>::value)
   : first{std::move(c.first)}, last{std::move(c.last)} val{val}{
  }
  ...
};
```

guarantees not to throw
if `std::string` guarantees
not to throw in its move constructor




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

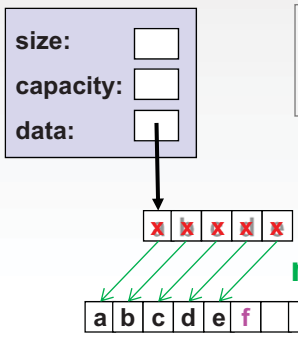

moves, because move constructors of strings don't throw

Vector Reallocation and Generated Move Constructors

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;
 public:
  ...
  Cust(Cust&& c) = default;

  ...
};
```

For move constructors:
- `string` and `vector<>` guarantee not to throw
- Other containers *may or may not* guarantee
- `pair<>`, `tuple<>`, ... guarantee not to throw, if members guarantee not to throw

guarantees not to throw
if all members and base classes guarantee
not to throw in their move constructor
(same when automatically generated)




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


moves, because move constructors of strings don't throw

Lessons Learned

- **For expensive members**
  - Initializing constructors might take by value and `move()`
  - Overload getters for `&&` and `const&`
- **Algorithms move**
- **Moved-from objects**
  - are nothing special for the C++ standard library
    - Functions/algorithms use them as other objects
  - should not break invariants
    - Ideally in a "valid but unspecified state"
- **To disable move semantics `=default` other special members**
  - Breaks naive "*Rule of 5*"
- **Use `noexcept` when *implementing* special move members**

Take Care

Code at: **josuttis.com/download/yow**

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis