Back to Basics: Move Semantics

NICOLAI JOSUTTIS

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

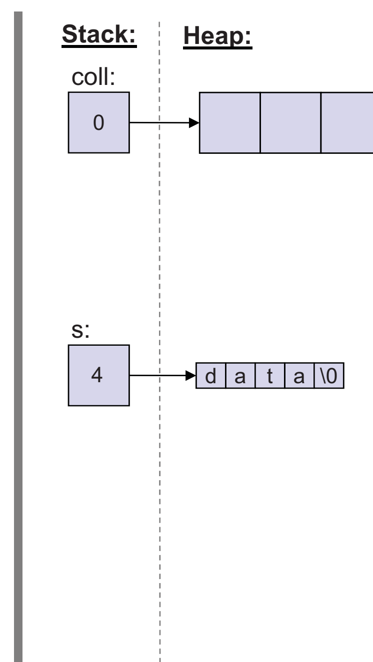
C++
©2021 by josuttis.com

C++ Move Semantics

Motivation

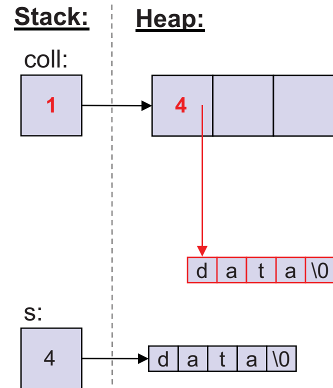
Copy Semantics

```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);
```



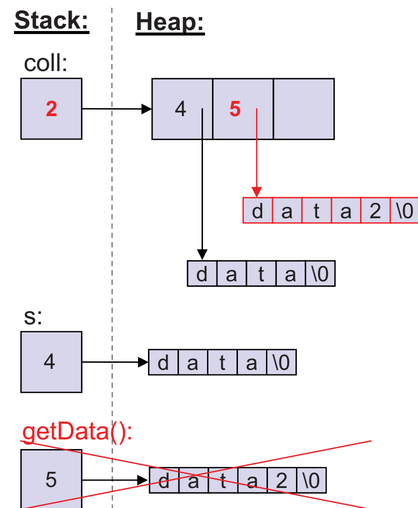
Copy Semantics

```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);
```



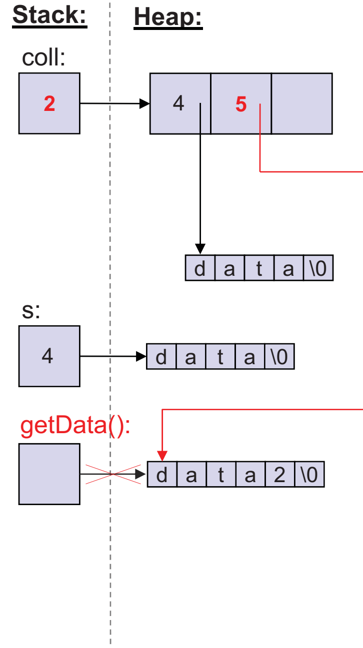
Copy Semantics (with C++98/C++03)

```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);

  coll.push_back(getData());
```



Move Semantics (since C++11)

```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);

  coll.push_back(getData());
```



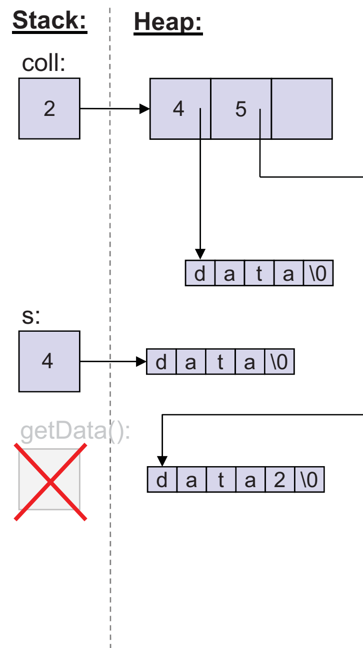
©2021 by josuttis.com
josuttis | eckstein
IT communication

Move Semantics (since C++11)

```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);

  coll.push_back(getData());
```

destruct temporary

Avoid objects with names



©2021 by josuttis.com
josuttis | eckstein
IT communication

C++ Move Semantics

`std::move()`

Objects with Names

- **Often you can't avoid objects with names:**
  - When you need an object/value multiple times:

```
std::string str{getData()};
…
coll1.push_back(str);              // copy (still need the value of str)
coll2.push_back(str);              // copy (but no longer need the value)
```

  - When you deal with parameters:

```
void reinit(std::string& s) {
  history.push_back(s);            // copy (but no longer need the value)
  s = getDefaultValue();
}

// read line-by-line from myStream and store them all in a collection:
std::string row;
while (std::getline(myStream, row)) {  // read next line into row
  allRows.push_back(row);              // and copy it to collection of all rows
}
```

`std::move()` for Objects with Names

- `std::move()`: *"I no longer need this value here"*
  - When you need an object/value multiple times:

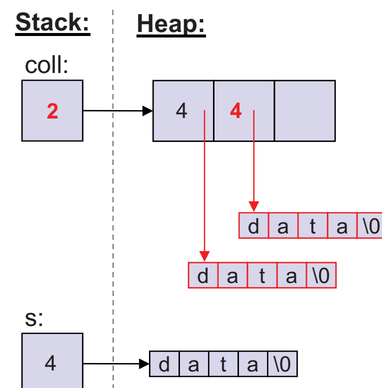
```
std::string str{getData()};
…
coll1.push_back(str);                  // copy (still need the value of str)
coll2.push_back(std::move(str));       // move (ok, no longer need the value)
```

  - When you deal with parameters:

```
void reinit(std::string& s) {
  history.push_back(std::move(s));     // move (ok, no longer need the value)
  s = getDefaultValue();
}

// read line-by-line from myStream and store them all in a collection:
std::string row;
while (std::getline(myStream, row)) {  // read next line into row
  allRows.push_back(std::move(row));   // and move it to collection of all rows
}
```


Processing Multiple Times

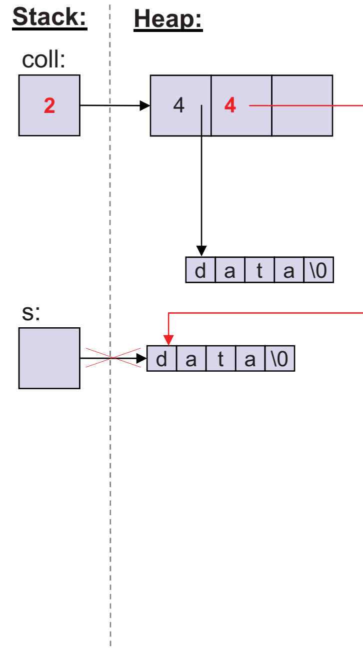
```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);

  coll.push_back(s);  // would copy again
```



std::move()

```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);

  coll.push_back(std::move(s));
```

"*I no longer need the value of* s *here*"



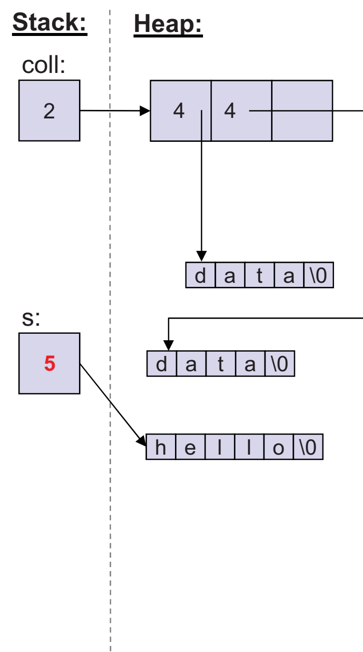
C++
©2021 by josuttis.com
13
josuttis | eckstein
IT communication

Valid but Unspecified State

```
std::vector<std::string> coll;
coll.reserve(3);

  std::string s{getData()};

  coll.push_back(s);

  coll.push_back(std::move(s));
```

A moved-from library object is in a valid but unspecified state

```
  std::cout << s;      // OK, some value
  int i = s.size();    // OK, consistent value
  s.append('.');       // OK, size() >= 1
  char c1 = s[0];      // OK, some value
  char c2 = s[5];      // Error (Undef.Behav.)
  s = "hello";         // OK, specified state
  ...
```



C++
©2021 by josuttis.com
14
josuttis | eckstein
IT communication

Re-using Objects after `std::move()`

- Yes, it makes sense to re-use objects after `std::move()`

```
// read line-by-line from myStream and store it in a collection:
std::vector<std::string> allRows;
std::string row;
while (std::getline(myStream, row)) {   // read next line into row
  allRows.push_back(std::move(row));    // and move it to somewhere
}
```

```
// swap two strings
void swap(std::string& a, std::string& b) {
  std::string tmp{std::move(a)};
  a = std::move(b);     // assign new value to moved-from a
  b = std::move(tmp);   // assign new value to moved-from b
}
```

C++
©2021 by josuttis.com
15
josuttis | eckstein
IT communication

C++ Move Semantics

RValue References

C++
©2021 by josuttis.com
16
josuttis | eckstein
IT communication

Vectors Without Move Semantics (C++03)

- **Containers have value semantics**
  - New elements are copied into the container
  - Passed arguments are not modified
- **This leads to unnecessary copies with C++98/C++03**

```
template <typename T>
class vector {
  public:
    ...
    // copy elem into the vector:
    void push_back(const T& elem);
    ...
};
```

unnecessary copies in C++98 / C++03

```
std::vector<std::string> coll;
std::string s = getData();
...
coll.push_back(s);              // copy s into coll

coll.push_back(getData());      // copy temporary into coll

coll.push_back(s+s);            // copy temporary into coll

coll.push_back(s);              // copy s into coll again
                                // (no longer need s)
return coll;
```


©2021 by josuttis.com

Vectors With Move Semantics (C++11)

- **With rvalue references you can provide move semantics**
- **Rvalue references bind to rvalues**
  - Caller no longer needs the value
  - May *steal* but keep valid

```
template <typename T>
class vector {
  public:
    ...
    // copy elem into the vector:
    void push_back(const T& elem);
    ...
    // move elem into the vector:
    void push_back(T&& elem);
    ...
};
```

now named lvalue reference

declares rvalue reference

```
#include <utility>             // declares std::move()

std::vector<std::string> coll;
std::string s = getData();
...
coll.push_back(s);              // copy s into coll

coll.push_back(getData());      // move temporary into coll

coll.push_back(s+s);            // move temporary into coll

coll.push_back(std::move(s));   // move s into coll
                                // (no longer need s)
return coll;
```


©2021 by josuttis.com

Strings With Move Semantics

- **Move semantics is usually implemented in:**
  - a **move constructor**
  - a **move assignment operator**

  as optimized copying
  - Steals by keeping the source valid

```
class string {
  private:
    int len;          // current number of characters
    char* data;       // array of characters

  public:
    // create a full copy of s:
    string (const string& s)
     : len{s.len} {                   // copy length
      if (len > 0) {                  // if not empty
        data = new char[len+1];       // - new memory
        memcpy(data, s.data,          // - copy chars
               len+1);
      }
    }
```

Copy constructor uses const lvalue reference

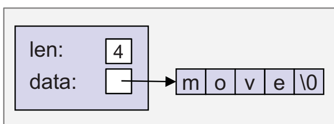
Move constructor uses rvalue reference

```
  public:
    // create a copy of s with its content moved:
    string (string&& s)
     : len{s.len},               // copy length and
       data{s.data} {            // copy pointer to memory
        s.data = nullptr;        // erase memory at source
                                 //  to really move ownership
        s.len = 0;
    }
    ...
};
```

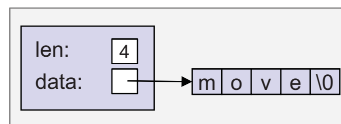

Strings With Move Semantics

```
coll.push_back(x)
```

```
string x = "move";
x: s:
```




```
string e1 = x;
e1:
```



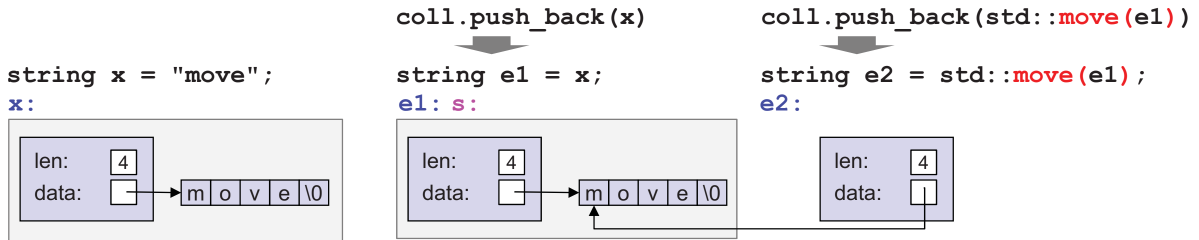

```
class string {
  private:
    int len;          // current number of characters
    char* data;       // array of characters

  public:
    // create a full copy of s:
    string (const string& s)
     : len{s.len} {                   // copy length
      if (len > 0) {                  // if not empty
        data = new char[len+1];       // - new memory
        memcpy(data, s.data,          // - copy chars
               len+1);
      }
    }
```

Copy constructor uses const lvalue reference

Move constructor uses rvalue reference

```
  public:
    // create a copy of s with its content moved:
    string (string&& s)
     : len{s.len},               // copy length and
       data{s.data} {            // copy pointer to memory
        s.data = nullptr;        // erase memory at source
                                 //  to really move ownership
        s.len = 0;
    }
    ...
};
```

Strings With Move Semantics




```
class string {
  private:
    int len;          // current number of characters
    char* data;       // array of characters

  public:
    // create a full copy of s:
    string (const string& s)
     : len{s.len} {                    // copy length
      if (len > 0) {                   // if not empty
        data = new char[len+1];        // - new memory
        memcpy(data, s.data,           // - copy chars
               len+1);
      }
    }
```

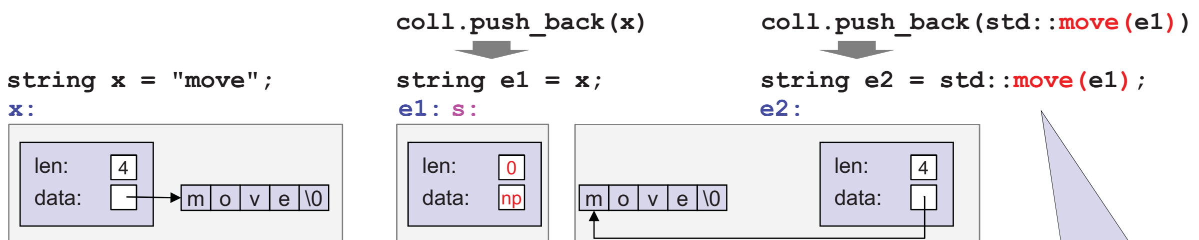
Copy constructor uses const lvalue reference

```
  public:
    // create a copy of s with its content moved:
    string (string&& s)
     : len{s.len},             // copy length and
       data{s.data} {          // copy pointer to memory
        s.data = nullptr;      // erase memory at source
                               //  to really move ownership
        s.len = 0;
    }
    ...
};
```

Move constructor uses rvalue reference

©2021 by josuttis.com 21 josuttis | eckstein IT communication

Strings With Move Semantics



`std::move(e1)` is equivalent to `static_cast<string&&>(e1)`


```
class string {
  private:
    int len;          // current number of characters
    char* data;       // array of characters

  public:
    // create a full copy of s:
    string (const string& s)
     : len{s.len} {                    // copy length
      if (len > 0) {                   // if not empty
        data = new char[len+1];        // - new memory
        memcpy(data, s.data,           // - copy chars
               len+1);
      }
    }
```

Copy constructor uses const lvalue reference

```
  public:
    // create a copy of s with its content moved:
    string (string&& s)
     : len{s.len},             // copy length and
       data{s.data} {          // copy pointer to memory
        s.data = nullptr;      // erase memory at source
                               //  to really move ownership
        s.len = 0;
    }
    ...
};
```

Move constructor uses rvalue reference

©2021 by josuttis.com 22 josuttis | eckstein IT communication

Overloading on References

- `void foo(const Type&)`
  - pass value without creating a copy
  - can bind to everything
- `void foo(Type&)`
  - pass named entity to return a value
  - only non-`const` named object (lvalues)
- `void foo(Type&&)`
  - pass value that is no longer needed
  - only objects without name or with `move()` (rvalues)
- `void foo(const Type&&)`
  - possible, but semantic contradiction
  - usually covered by `const Type&`

read-only access
- in parameter to read

write access
- (in)out parameter

move access
- in parameter to adopt value

```
std::vector<std::string> coll;
…
const std::string s = getData();
…
coll.push_back(std::move(s));      // copies
```

Don't use `const` if you later move

C++

©2021 by josuttis.com

josuttis | eckstein IT communication

Overloading on References

- `void foo(const Type&)`
  - pass value without creating a copy
  - can bind to everything
- `void foo(Type&)`
  - pass named entity to return a value
  - only non-`const` named object (lvalues)
- `void foo(Type&&)`
  - pass value that is no longer needed
  - only objects without name or with `move()` (rvalues)
- `void foo(const Type&&)`
  - possible, but semantic contradiction
  - usually covered by `const Type&`

read-only access
- in parameter to read

write access
- (in)out parameter

move access
- in parameter to adopt value

```
std::vector<std::string> coll;
…
void insert(const std::string& s) {
  coll.push_back(std::move(s));  // copies
}
```

C++

©2021 by josuttis.com

josuttis | eckstein IT communication

Overloading on References

- `void foo(const Type&)`
  - pass value without creating a copy
  - can bind to everything
- `void foo(Type&)`
  - pass named entity to return a value
  - only non-`const` named object (lvalues)
- `void foo(Type&&)`
  - pass value that is no longer needed
  - only objects without name or with `move()` (rvalues)
- `void foo(const Type&&)`
  - possible, but semantic contradiction
  - usually covered by `const Type&`

read-only access
- in parameter to read

write access
- (in)out parameter

move access
- in parameter to adopt value

```
const std::string getValue(); // forward decl.
...
std::vector<std::string> coll;
...
coll.push_back(getValue());    // copies
```

Don't use `const` when returning by value

C++ ©2021 by josuttis.com

josuttis | eckstein IT communication

Basics of Move Semantics

- **Move Semantics allows to**
  - optimize copying
  - by semantically stealing resources from a source
  - Implemented by dealing with rvalue references (*type*`&&`)
- **Ideally supported by temporary objects**
  - Create them on the fly
    - when passing arguments
    - in `return` statements
- **`std::move()`**
  - signals "*I no longer need this value here*"
  - A moved-from objects is in a valid but unspecified state
    - Any operation without any assumption about the value is fine

Avoid objects with names

```
// instead:
MyType x{42, "hello"};
foo(x);   // x no longer used

// better:
foo(MyType{42,"hello"});

// or:
foo(std::move(x));
```

C++ ©2021 by josuttis.com

josuttis | eckstein IT communication

C++ Move Semantics

Move Semantics for Classes

Basic Move Support

- **Guarantees for library objects** (§17.6.5.15 [lib.types.movedfrom]**):**
  - "Unless otherwise specified, ... moved-from objects shall be placed in a valid but unspecified state."
- **Copy as Fallback**
  - If no move semantics is provided, copy semantics is used
    - You can disable this fallback

Used by **Move-Only Types** such as std::thread, streams, std::unique_ptr<>

- **Default move operations are generated**
  - Move constructor and move assignment operator
    - move members

  but only if this can't be a problem
  - Only if there is no user-declared special member function
    - No copy constructor
    - No assignment operator
    - No destructor

Move Semantics and Special Member Functions

```
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 copy constructor

                                        // no move constructor

    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.val << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;

Cust c1{"Joe", "Fox", 77};
v.push_back(std::move(c1));           // moves c1
std::cout << "c1: " << c1 << '\n';    // c1: [77: ??? ???]
```

- **Move semantics** is **enabled** because no other special member function is user-declared
- Unless a move is not implementable

C++

©2021 by josuttis.com

29

josuttis | eckstein
IT communication

Move Semantics and Special Member Functions

```
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} {
    }
    Cust(const Cust&) = default; // copy constructor

                                        // no move constructor

    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.val << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;

Cust c1{"Joe", "Fox", 77};
v.push_back(std::move(c1));           // copies c1
std::cout << "c1: " << c1 << '\n';    // c1: [77: Joe Fox]
```

- **Move semantics** is **disabled** because of user-declared other special member function
- **Copying** used as **fallback**

C++

©2021 by josuttis.com

30

josuttis | eckstein
IT communication

Move Semantics and Special Member Functions

```
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} {
    }
    Cust(const Cust& c)             // copy constructor
     : first{c.first}, last{c.last}, val{c.val} {
    }
    Cust(Cust&& c) noexcept         // move constructor
     : first{std::move(c.first)}, last{std::move(c.last)}, val{c.val} {
        c.val *= -1;
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.val << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;

Cust c1{"Joe", "Fox", 77};
v.push_back(std::move(c1));           // moves c1
std::cout << "c1: " << c1 << '\n';    // c1: [-77: ??? ???]
```

Mark move constructor with `noexcept`, if implemented and it never throws

Parameter `c` has no move semantics unless marked with `move()` again,
(the **caller** no longer needs the value, but we might need it multiple times)

Rules for Special Member Functions

user declaration of ↓ / forces →	default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	undeclared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
copy constructor	undeclared	user declared	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
copy assignment	defaulted	defaulted	user declared	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
move constructor	undeclared	deleted	deleted	user declared	undeclared (fallback disabled)	defaulted
move assignment	defaulted	deleted	deleted	undeclared (fallback disabled)	user declared	defaulted
destructor	defaulted	defaulted	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	user declared

(move constructor / move assignment rows: can't copy)

Adopted from http://foonathan.net/2019/02/special-member-functions/
with friendly permission by Howard Hinnant

Move Semantics in Polymorphic Classes

- **Declared virtual destructors disable move semantics**
  - Moving special member functions are not generated
  - If and only if a polymorphic base class has members expensive to copy, it might make sense to declare/define move operations
- **Don't declare destructors in derived classes** (unless you have to)

```
class Person {                      // polymorphic base class with virtual functions
 protected:
   std::string id;                  // to support move semantics for id, declare move functions
 public:
   ...
   virtual void print() const = 0;
   virtual ~Person() = default;     // disables move semantics for members
};

class Customer : public Person {    // derived polymorphic class
 protected:
   std::vector<int> data;           // move semantics for data enabled without special function
 public:
   ...
   virtual void print() const override;
   virtual ~Customer() = default;   // disables move semantics for members
};
```

(The line `virtual ~Customer() = default;` is crossed out.)

C++ Move Semantics

Perfect Forwarding

Motivation for Perfect Forwarding 1/2

- **Overloading for `const`/non-`const` lvalues and rvalues:**

```
class C {
  …
};
void foo(const C&);        // read-only access (binds to all values)
void foo(C&);              // write access (binds to non-const lvalues)
void foo(C&&);             // move access (binds to non-const rvalues)

C v;
const C c;
foo(v);                    // calls foo(C&)
foo(c);                    // calls foo(const C&)
foo(C{});                  // calls foo(C&&)
foo(std::move(v));         // calls foo(C&&)
```

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Motivation for Perfect Forwarding 2/2

- **Forward move semantics in helper functions:**

```
class C {
  …
};
void foo(const C&);        // read-only access (binds to all values)
void foo(C&);              // write access (binds to non-const lvalues)
void foo(C&&);             // move access (binds to non-const rvalues)
void callFoo(const C& x) {
  foo(x);                  // x is const lvalue       => calls foo(const C&)
}
void callFoo(C& x) {
  foo(x);                  // x is non-const lvalue => calls foo(C&)
}
void callFoo(C&& x) {
  foo(std::move(x));       // x is non-const lvalue => needs std::move() to call foo(C&&)
}
C v;
const C c;
callFoo(v);                // calls foo(C&)
callFoo(c);                // calls foo(const C&)
callFoo(C{});              // calls foo(C&&)
callFoo(std::move(v));     // calls foo(C&&)
```

use `std::move()` to forward move semantics

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Perfect Forwarding

- **Perfect forwarding of parameters:**
  1. Template parameter
  2. Declaring the parameter as `&&` of the template parameter
  3. `std::forward<>()`

```
void foo(const C&);     // read-only access (binds to all values)
void foo(C&);           // write access (binds to non-const ...)
void foo(C&&);          // move access (binds to non-con...)

template <typename T>
void callFoo(T&& x)     // x is a universal (or forwarding) reference
{
  foo(std::forward<T>(x));   // perfectly forwards move semantics
}

C v;
const C c;
callFoo(v);             // foo(std::forward<T>(x))  =>  foo(x)
callFoo(c);             // foo(std::forward<T>(x))  =>  foo(x)
callFoo(C{});           // foo(std::forward<T>(x))  =>  foo(std::move(x))
callFoo(std::move(v));  // foo(std::forward<T>(x))  =>  foo(std::move(x))
```

Universal / forwarding ***reference*** (community / C++ standard term)
- Can refer to `const` and non-`const`
- Can refer to rvalue and lvalue

`std::forward<>()` is `std::move()` only for rvalues

C++

©2021 by josuttis.com

josuttis | eckstein IT communication

The Two Meanings of `&&` Declarations

- **&& declares**
  - For types: raw rvalue references
  - For template params: universal/forwarding references

```
class Type {
  ...
};

void foo(Type&& x)     // rvalue reference
{
  std::is_const<Type>::value   // always false
  ...
  // perfectly forward x:
  bar(std::move(x));
  // x has valid but unspecified state
}

Type v;
const Type c;

foo(v);               // ERROR
foo(c);               // ERROR
foo(Type{});          // OK
foo(std::move(v));    // OK
```

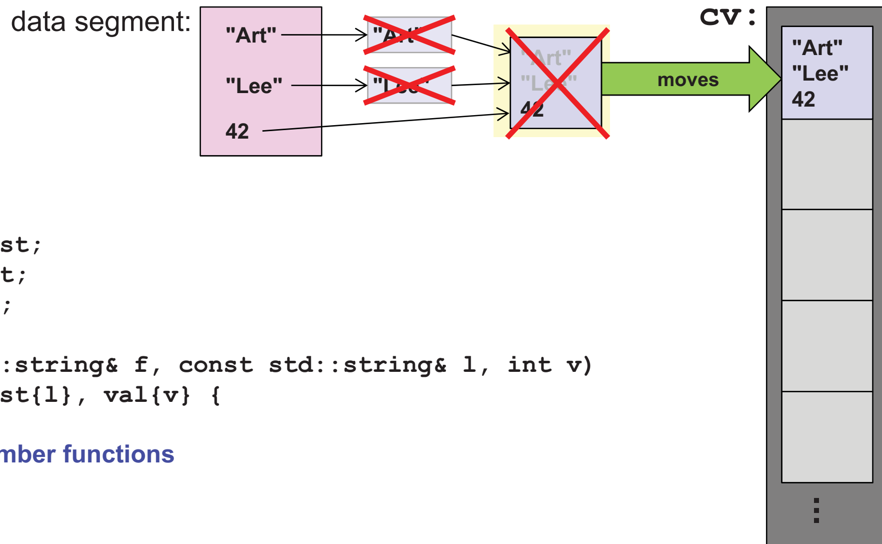
```
class Type {
  ...
};

template<typename T>
void foo(T&& x)        // universal reference
{
  std::is_const<T>::value      // maybe true or false
  ...
  // perfectly forward x:
  bar(std::forward<T>(x));
  // x has valid but unspecified state
}

Type v;
const Type c;

foo(v);               // OK, x is non-const
foo(c);               // OK, x is const
foo(Type{});          // OK, x is non-const
foo(std::move(v));    // OK, x is non-const
```

C++

©2021 by josuttis.com

josuttis | eckstein IT communication

Using Generated and Implemented Move Semantics



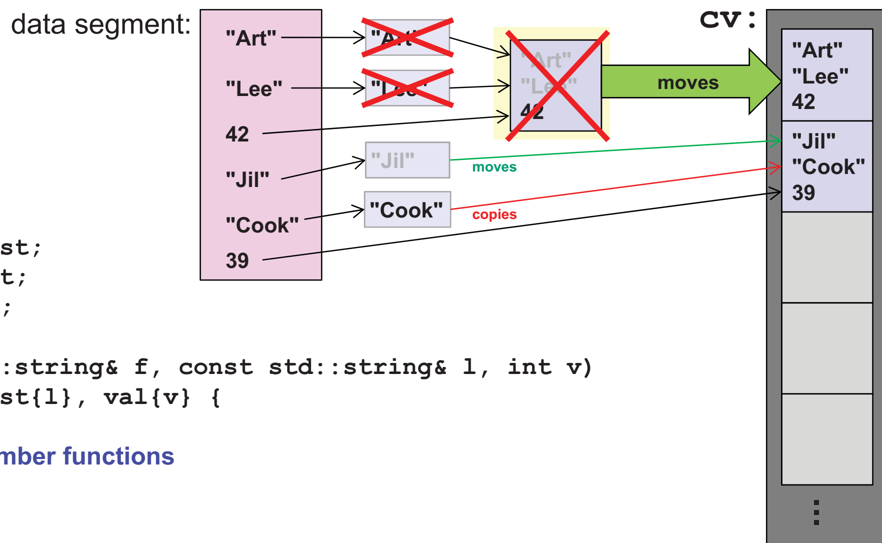

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;
 public:
  Cust(const std::string& f, const std::string& l, int v)
   : first{f}, last{l}, val{v} {
  }
  ... // no special member functions
};

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

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Perfect Forwarding with Emplace Functions



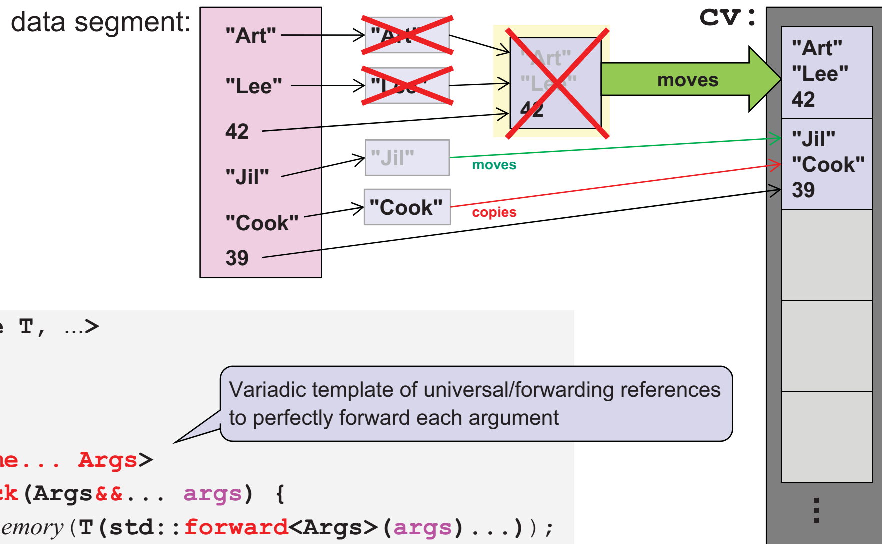

```
class Cust {
 private:
  std::string first;
  std::string last;
  int         val;
 public:
  Cust(const std::string& f, const std::string& l, int v)
   : first{f}, last{l}, val{v} {
  }
  ... // no special member functions
};

std::vector<Cust> cv;

cv.push_back(Cust{"Art","Lee",42});    // create customer and copy/move it into cv

std::string first{"Jil"};
std::string last{"Cook"};
cv.emplace_back(std::move(first), last, 39);    // create new customer inside cv
```

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Perfect Forwarding with Emplace Functions




```
template <typename T, ...>
class vector {
 public:
  ...
  template<typename... Args>
  void emplace_back(Args&&... args) {
    place_element_in_memory(T(std::forward<Args>(args)...));
  }
  ...
};
```

Variadic template of universal/forwarding references to perfectly forward each argument

```
std::string first{"Jil"};
std::string last{"Cook"};
cv.emplace_back(std::move(first), last, 39);   // create new customer inside cv
```

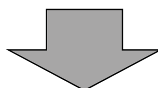
C++
©2021 by josuttis.com

josuttis | eckstein
IT communication

Perfect Passing with auto&&

- **To perfect forward a return value**
  - declare returned value as **auto&&**
    - Universal/forwarding reference without being a template parameter
  - and forward
- **For example:**

```
// pass return value of compute() to process():
process(compute(t));   // OK, perfect
```



Note: A reference extends the lifetime of a temporary.

```
// same, but doing something between compute() and process():
auto&& val = compute(t);
...
process(std::forward<decltype(val)>(val));
```

C++
©2021 by josuttis.com

josuttis | eckstein
IT communication

`auto&&` as Universal/Forwarding Reference

- **Reference that**
  - can *universally* refer
    - to temporary objects and objects marked with `move()` (rvalues)
    - to named objects (lvalues)
  - keeps its non-constness
  - is useful for *perfect forwarding*

```
std::string returnTmpString();
const std::string& returnConstStringRef();
std::string s{"some lvalue"};

const auto& s1 = s;                        // OK, s1 is const
const auto& s2 = returnTmpString();        // OK, s2 is const
const auto& s3 = returnConstStringRef();   // OK, s3 is const

auto& s4 = s;                              // OK, s3 is not const
auto& s5 = returnTmpString();              // ERROR: cannot bind non-const lvalue reference to rvalue
auto& s6 = returnConstStringRef();         // OK, s6 is const

auto&& s7 = s;                             // OK, s7 is not const
auto&& s8 = returnTmpString();             // OK, s8 is not const
auto&& s9 = returnConstStringRef();        // OK, s9 is const
```

`const auto&` can refer to everything, **but** `const`

`auto&` **cannot** refer to everything

`auto&&` can refer to everything and **keeps non-constness**

C++20: Universal/forwarding References for Ranges and Views

- **`const` views might not support iterating**
  - They might have to modify their state while iterating
  - Use universal/forwarding references when passed by reference

```
template<typename T>
void print(const T& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

Better:

```
template<typename T>
void printElems(T&& coll)
```

or:

```
void printElems(auto&& coll)
```

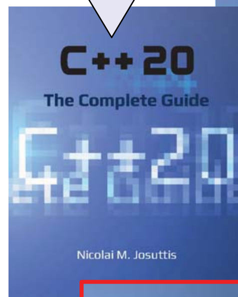
```
std::vector vec{1, 2, 3, 4, 5};

print(vec);                                // OK
print(vec | std::views::drop(3));          // OK
print(vec | std::views::filter(...));      // ERROR without universal reference
std::list lst{1, 2, 3, 4, 5};
print(lst | std::views::drop(3));          // ERROR without universal reference
```

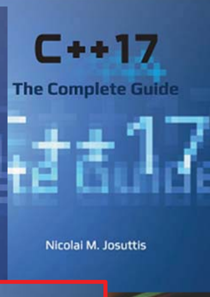
Thank You!

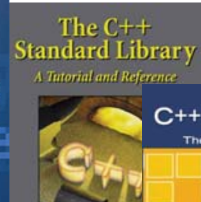
Nicolai M. Josuttis

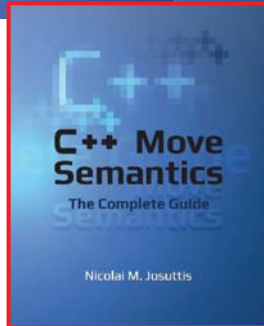
www.josuttis.com
nico@josuttis.com
@NicoJosuttis

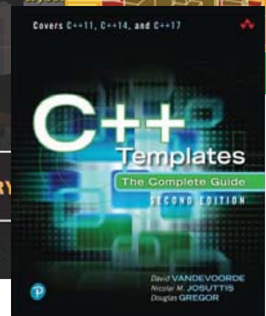
First draft is out
cppstd20.com











©2021 by josuttis.com