# RVALUE REFERENCES & MOVE SEMANTICS

# Plan for Today



- Review of *lvalue* and *rvalue* expressions
- Review of (*lvalue*) references
- What is motivation for move semantics
- How did C++ implement move semantics using *rvalue* references?
- Move Semantics
  - Move Constructor
  - Move Assignment Operator
  - `std::move`

# C: *Lvalues* and *Rvalues* (1/7)



- Original definition from C:
  - Every expression is an *lvalue* or an *rvalue*
  - *Lvalue* [short for *Left value*] expression can appear on left or right hand side of assignment expression

```
int a{11}, b{22};

// a and b are both lvalues
a = b;   // ok
b = a;   // ok
a = a+b; // ok
```

# C: *Lvalues* and *Rvalues* (2/7)



- Original definition from C:
  - Every expression is an *lvalue* or an *rvalue*
  - *Lvalue* (short for *Left value*) expression can appear on left or right hand side of assignment operator
  - *Rvalue* [short for *Right value*] expression can only appear on right hand side of assignment operator

```
double a{}, b{1.1}, c{-2.0};
a = b+c;             // b+c is rvalue
b = std::abs(a*c); // right hand side is rvalue
c = std::pow(b, std::abs(a)); // RHS is rvalue
30 = a*b; // error: rvalue on left of assignment
```

# C: *Lvalues* and *Rvalues* (3/7)



- Addition of `const` type qualifier in C98 complicated definition
  - Every expression is an *lvalue* or an *rvalue*
  - *Lvalue* [short for *locater value*] is expression that refers to identifiable memory location
  - By exclusion, any non-*lvalue* expression is an *rvalue*; think of *rvalue* as "value resulting from expression"

```
double a{}, b{1.1};
double const c{-2.0};
a = b+c; // a is modifiable lvalue; b+c is rvalue
b = std::abs(a*c); // b is modifiable lvalue
c = a*b; // error: non-modifiable lvalue on left
```

# C: *Lvalues* and *Rvalues* (4/7)



- We can use an *lvalue* when an *rvalue* is required, but we cannot use an *rvalue* when an *lvalue* is required!!!
- Useful to visualize a variable as name associated with certain memory locations

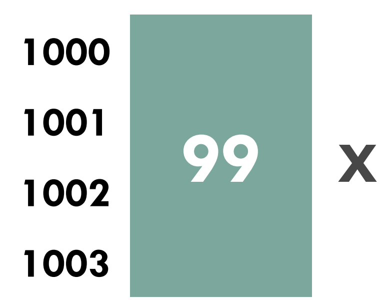
```c
int x = 99;
```

- Sometimes [as an *lvalue*] x means its memory locations and sometimes [as an *rvalue*] x means value stored in those memory locations

here, x means *lvalue*
[the address 1000]

```c
int x = 99;
x = 100;
x = x + 2;
```

while here, x means *rvalue*
[value stored at address 1000]



# C: *Lvalues* and *Rvalues* (5/7)



x = y

Symbol x, in this context, means "address that x represents"

This expression is termed ***lvalue***

*lvalue* means "x's memory location"

*lvalue* is known at compile-time

Symbol y, in this context, means "contents of address that y represents"

This expression is termed ***rvalue***

*rvalue* means "value of y"

*rvalue* is not known until run-time

# C: *Lvalues* and *Rvalues* (6/7)



- Knowledge of *lvalues* and *rvalues* helps in understanding behavior of operators
  - Some operators require *lvalue* operands while others require *rvalue* operands
  - Some operators return *lvalues* while others return *rvalues*

# C: *Lvalues* and *Rvalues* (7/7)



- Most function call expressions are *rvalues*
- Function calls returning pointers are *lvalues*

```
int gi{10}; // variable defined at file scope
int incr(int x) { return x+1; } // evaluates to rvalue
int* foo()      { return &gi; } // evaluates to lvalue

int *pi = foo(); // ok: pi points to gi
*pi = incr(*pi); // ok: gi is 11
incr(*pi) = 10;  // error: incr(*pi) is rvalue
*foo() = 111;    // ok: foo() and operator * are lvalues
++*foo();        // ok: foo() is lvalue;
                 // operator * takes and returns lvalue;
                 // operator ++ takes lvalue
```

# C++: *Lvalues* and *Rvalues* (1/7)



- Ordinary functions can now return references

```
int gi1{1}, gi2{2}; // variable defined at file scope
int* foo() { return &gi1; }       // evaluates to lvalue
int& boo() { return gi2; }        // evaluates to lvalue

int *pi = foo(); // ok: pi points to gi1
*pi = 100;       // ok: gi1 is now 100
*foo() = 111;    // ok: gi1 is now 111
++*foo();        // ok: gi1 is now 112

int x = boo();    // ok: x initialized with value 2
int *px = &boo(); // ok: px points to gi2
boo() = 100;      // ok: gi2 is now 100
++boo();          // ok: gi2 is now 101
int &rx = boo();  // ok: rx is alias to gi2
```

# C++: *Lvalues* and *Rvalues* (2/7)



- *Every expression is either an lvalue or an rvalue*
- *Lvalue* expressions name objects that *persist* beyond single expression until end of scope
  - For example, a variable expression
- *Rvalue* expressions are *temporaries* that evaporate at end of full expression in which they live – at semicolon indicating sequence point
  - Either literals or temporary objects created in the course of evaluating expressions

# C++: *Lvalues* and *Rvalues* (3/7)



- Expression `++x` is *lvalue* while `x++` is *rvalue*
  - `++x` modifies and returns the persistent object
  - `x++` copies original value, modifies persistent object and then returns temporary copy
- *Lvalueness* versus *rvalueness* doesn't care about what an expression does
  - It cares about what the expression names – something persistent or something temporary

# C++: *Lvalues* and *Rvalues* (4/7)



```
int x = 10, *px = &x;    // x and px are lvalues
int foo(std::string);    // foo() evaluates to rvalue
std::string s{"hello"}; // s is lvalue
x = foo(s);              // ok: foo()'s return value is rvalue
px = &foo();             // error: foo()'s return value is rvalue
x = foo("hello");        // temporary string created for call is rvalue
std::vector<int> vi(10); // vi is lvalue
vi[5] = 11;               // ok: vector<T>::op[] returns T&
int& foobar();            // foo() evaluates to lvalue
foobar() = 11;            // ok: foobar()'s return value is lvalue
px = &foobar();           // ok: foobar()'s return value is lvalue
```

# C++: *Lvalues* and *Rvalues* (5/7)



- Another way to distinguish between *lvalueness* and *rvalueness*: Can you take address of expression?
  - Yes – expression is *lvalue*: `&x`, `&*ptr`, `&a[5]`, …
  - No – expression is *rvalue*: `&1029`, `&(x+y)`, `&x++`, …
- Why?
  - Standard says address-of-operator requires *lvalue* operand
  - Taking address of persistent object is fine
  - But, taking address of temporary is dangerous because they evaporate quickly after expression is evaluated!!!

# C++: *Lvalues* and *Rvalues* (6/7)



- User-defined types in C++ introduce some subtleties regarding modifiability and assignability

```
class Str {
public:
  char& operator[](size_t idx);
  char const& operator[](size_t idx) const;
  // ...
};

Str s{"hello"}, t{"world"};
Str const c{"constant"};
s[0] = 'A' + (c[0] - 'a'); // ok: s[0] is 'H'
c[0] = 'C';                // error: c is non-modifiable lvalue
(s+t)[0] = 'C';            // ok but weird since s+t is rvalue
```

# C++: *Lvalues* and *Rvalues* (7/7)



- Lvalue and rvalue xpressions can be either modifiable [non `const`] or non-modifiable [`const`]:

```
std::string s1{"iowa"};
std::string const s2{"ohio"};
std::string f1() { return "texas"; }
std::string const f2() { return "idaho"; }

s1;   // modifiable lvalue
s2;   // const lvalue
f1(); // modifiable rvalue
f2(); // const rvalue
```

# *Lvalue* References (1/6)



- In a type name, notation X& means "*alias to* X"
  - Declared name binds to modifiable *lvalue*
- Called "ordinary" reference in C++98 and now called *lvalue reference*

```
int x {2};
int& rx {x};      // x and rx refer to same int
                  // anywhere you use x, you can now use rx
rx = 4;           // x now becomes 4
int   y {rx};     // y is initialized to 4
int   *pi {&rx}; // pi points to x
int*& rpx {pi};   // rpx is alias to pi
++*rpx;           // increment object that pi points at
```

# *Lvalue* References (2/6)



- In a type name, notation X& means "*reference to* X"
  - Binds to modifiable lvalues
  - Can't bind to `const` *lvalues*: violates `const` correctness
  - Can't bind to *rvalues*, literals, and expressions requiring conversion: modifying temporaries that evaporate [along with modifications made] can lead to bugs

# *Lvalue* References (3/6)



- Why can't *rvalue* bind to *lvalue* reference? To avoid unnecessary bugs!!!

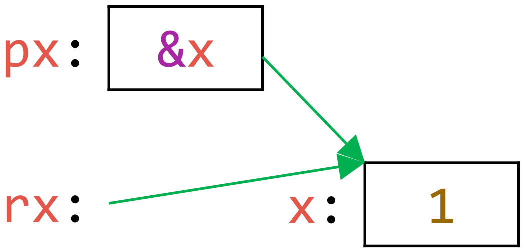
```
void incr(int& ri) { ++ri; }

void foo() {
  double d{10.1};
  incr(d); // suppose expression compiles
  // ...
}
```

If binding of *rvalues* to non`const` *lvalue* references is allowed, then `d` is not incremented
Instead, temporary `int` passed as argument to `incr()` would be the one incremented and then evaporated!!!

# *Lvalue* References (4/6)



- What do we know about *lvalue* references?
  - Must be initialized – no such thing as null reference
  - Value of reference cannot be changed after initialization
  - Operators *don't* operate on reference
  - Cannot get pointer to reference
  - Cannot define array of references
- Basically: a *reference is not an object; instead a reference is an alias to an object*




```
int x{1}, *px{&x}, &rx{x};
```

# *Lvalue* References (5/6)



```
int i{1}, i2{2048};     // i1 and i2 are lvalues of type int
int &r{i}, r2{i2};      // r is alias for i; r2 is an int
int i3{1024}, &ri{i3}; // i3 is an int;
                        // ri is reference bound to i3
int &r3{i3}, &r4{i2};  // both r3 and r4 are references
int &r5{10};            // error: initializer must be lvalue
int &r7 = i*42;         // error: initializer must be lvalue
double dval{3.14};      // dval is lvalue of type double
int &r6{dval};          // error: initializer must be int lvalue
const int ci{1024};     // ci is non-modifiable lvalue
int &r8{ci};            // error: initializer must be nonconst lvalue
```

# *Lvalue* References (6/6)



- Main purpose: Allow functions to change objects defined in scope of calling function [so called "*in/out*" parameters]

```
template <typename T>
void foo(std::vector<T>& rv) {
  // in parameter:  foo uses values in rv
  // out parameter: foo modifies values in rv
}
```

# `const` *Lvalue* References (1/6)



- In a type name, notation `X const&` means "non-modifiable (`const`) *lvalue* reference to `X`"
  - Binds to modifiable *lvalues*
  - Binds to `const` *lvalues*
  - Binds to modifiable *rvalues*
  - Binds to `const` *rvalues*
- Doesn't distinguish between *lvalues* and *rvalues*

# const *Lvalue* References (2/6)



```
int& ri{1};                   // error: lvalue needed
int const& rci{1};            // ok: binds to rvalue
int x{10};                    // lvalue
int const& rcx1{x};           // binds to lvalue
int const y{11};              // non-modifiable lvalue
int const& rcx2{y};           // ok: binds to const lvalue
int foo();                    // modifiable rvalue
int const& rcx3{foo()};       // ok: binds to rvalue
int const boo();              // non-modifiable rvalue
int const& rcx4{boo()};       // ok: binds to const rvalue
int const& rcx5{12.5};        // ok: binds to literal
```

# `const` *Lvalue* References (3/6)



- While initializer for "plain" `X&` must be *lvalue* of type `X`, initializer for `X const&` need not be an *lvalue* or even of type `X`
  - First, implicit type conversion to `X` is applied if necessary
  - Then, resulting value is placed in temporary variable of type `X`
  - Finally, reference makes binding to this temporary variable

# `const` *Lvalue* References (4/6)



```
double& dr{1};             // error
double const& cdr{1}; // ok
```

interpret as

```
// first, create temporary with right value
double const __tmp{(double)1};
// then, use temporary as initializer
// for const lvalue reference
double const& cdr{__tmp};
```

# `const` *Lvalue* References (5/6)



- Main purpose of *lvalue* references:
  - Allow functions to change objects defined in scope of calling function [so called "*in/out*" parameters]
- Main purpose of `const` *lvalue* references:
  - Allow function to refer to objects defined in scope of calling function whose values shouldn't be changed [so called "*in*" parameters]

# `const` *Lvalue* References (6/6)



- Main purpose: Allow function to refer to objects defined in scope of calling function whose values shouldn't be changed [so called "*in*" parameters]

```
// foo is declared to take by reference to avoid
// unnecessary and expensive copies via copy ctor
template <typename T>
T foo(std::vector<T> const& rrv) {
  // in parameter: foo uses values in rrv
}

std::vector<int> vi;
// fill vi with lots of values ...
// use foo to perform an action using vi
foo(vi);
```

# Reference Parameter: *Lvalue* or *Rvalue*? (1/5)



- Reference is a name
- Can we speak about *lvalueness* of this name in context of function parameter?
- Yep!!!
- Are you confused???

# Reference Parameter: *Lvalue* or *Rvalue*? (2/5)



- Inside definition of `test()`, `str` is a name, so it is an *lvalue*!!!

```
string f1()          { return "texas"; }
const string f2() { return "idaho"; }
void test(const string& str);

test(f1());    // f1() evaluates to rvalue
test(f2());    // f2() evaluates to const rvalue
test("iowa"); // string("iowa") is rvalue
string s{"ohio"};
test(s);       // s is lvalue
```

# Reference Parameter: *Lvalue* or *Rvalue*? (3/5)



- Have you ever bound an *rvalue* to a `const` reference and then taken its address?
- Yes – you've!!! Consider class `X`:

```
class X {
private:
  // some data members here
public:
  X(const X&);
  X& operator=(const X&);
  // other member functions
};
```

```
X& X::operator=(const X& rhs) {
  // check for self-assignment
  if (this != &rhs) {
    // clone rhs & assign cloned
    // resources to *this
  }
  return *this;
}
```

# Reference Parameter: *Lvalue* or *Rvalue*? (4/5)



- You might then use class X like this:

```
// function returning rvalue
X foo();

X x1, x2;
// do some stuff ...
x1 = foo();
```

```
x1.operator=( foo() )
```

```
X& X::operator=(const X& rhs) {
  // check for self-assignment
  if (this != &rhs) {
    // clone rhs & assign cloned
    // resources to *this
  }
  return *this;
}
```

*rvalue* returned by `foo()` is bound to parameter `rhs` in `X::op=()`

And, in self-assignment test, we're taking address of `rhs` [which is referencing *rvalue* returned by `foo()`]

# Reference Parameter: *Lvalue* or *Rvalue*? (5/5)



- Reference parameter bound to *lvalue* or *rvalue* is itself an *lvalue*
- And, reference parameter bound to `const` *lvalue* or `const` *rvalue* is `const` *lvalue* reference
- Remember: "*lvalueness versus rvalueness is a property of expressions, not of objects*"

# Reference to *Rvalue*? (1/3)



- What can we do with *modifiable rvalues*?
  - Can't bind reference (`X&`) to modifiable *rvalues*
  - Can't assign things to *rvalues*
- Can they be really modified?
  - Definitely not in C
  - Maybe in C++: Calling non`const` function on modifiable *rvalue* is allowed in C++

# Reference to *Rvalue*? (2/3)



```
class Complex {
  double mReal{0.0}, mImag{0.0};
public:
  // ctors ...
  Complex operator+=(Complex const& rhs);
  // other member functions ...
};
// non-member operator+ overload ...
Complex operator+(Complex const& lhs, Complex const& rhs);

Complex a(1, 2), b(3, 4), c{a+b};
(a+b) = c;
```

Compiles!!! Left hand expression returns a temporary [an *rvalue*] that is assigned before it evaporates!!!

# Reference to *Rvalue*? (3/3)



```
class Complex {
  double mReal{0.0}, mImag{0.0};
public:
  // ctors ...
  Complex operator+=(Complex const& rhs);
  // other member functions ...
};
const Complex
operator+(Complex const& lhs, Complex const& rhs);

Complex a(1, 2), b(3, 4), c{a+b};
(a+b) = c;
```

Doesn't compile anymore since non-member `operator+(Complex const&, Complex const&)` returns non-modifiable *rvalue*

# Motivation for Move Semantics (1/19)



```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

// suppose don't want RVO
std::vector<Str> v;

v = f();
```

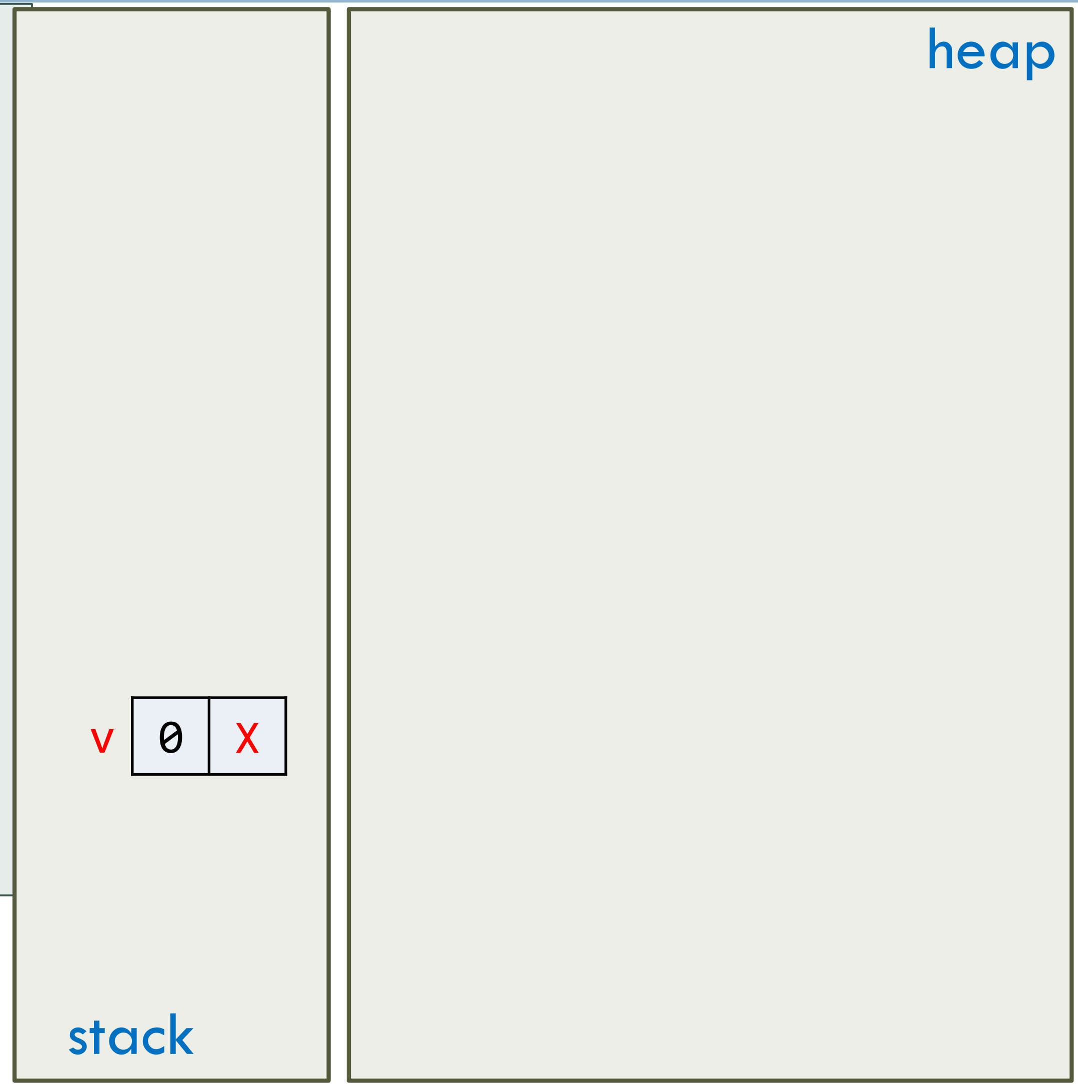
stack

heap

# Motivation for Move Semantics (2/19)



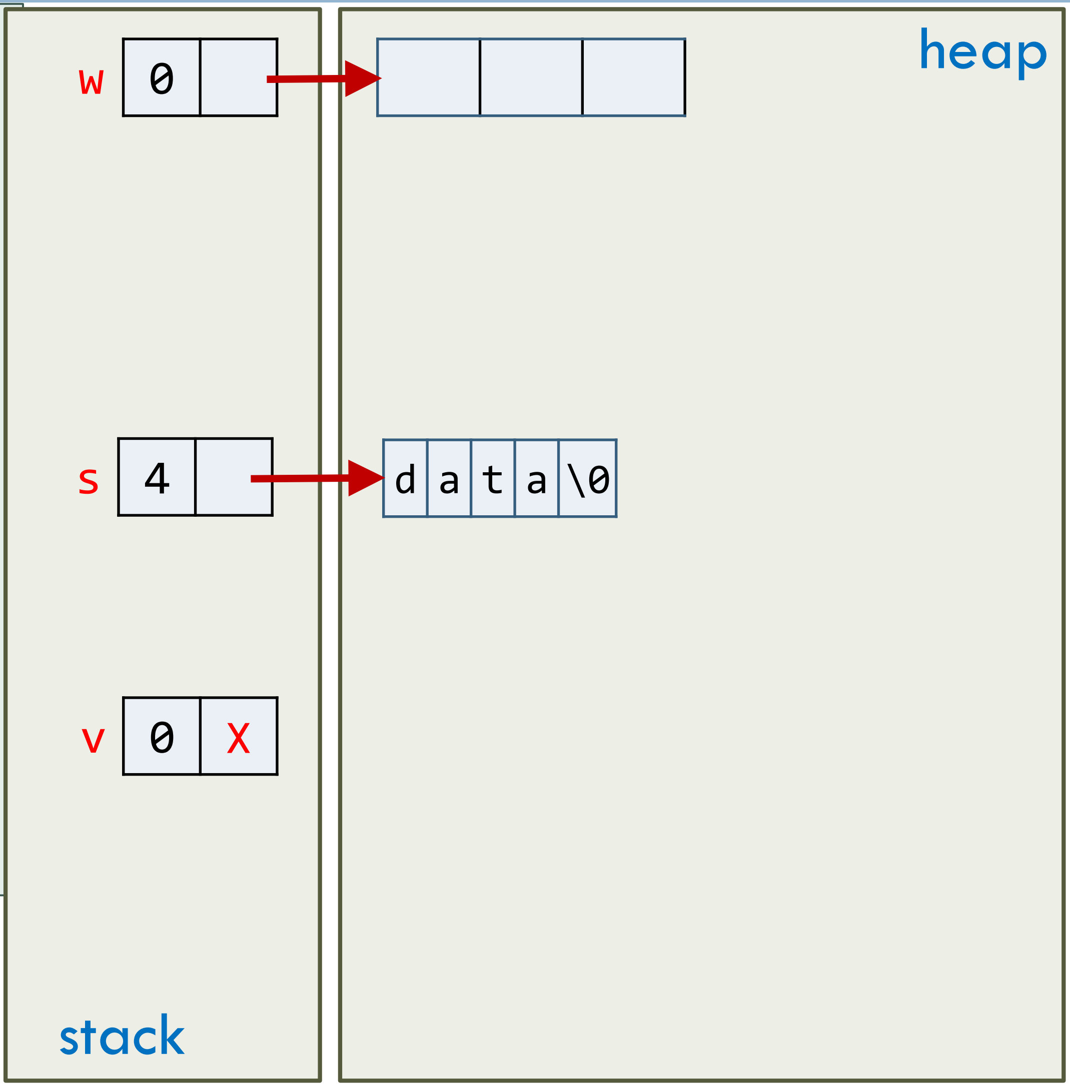
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (3/19)



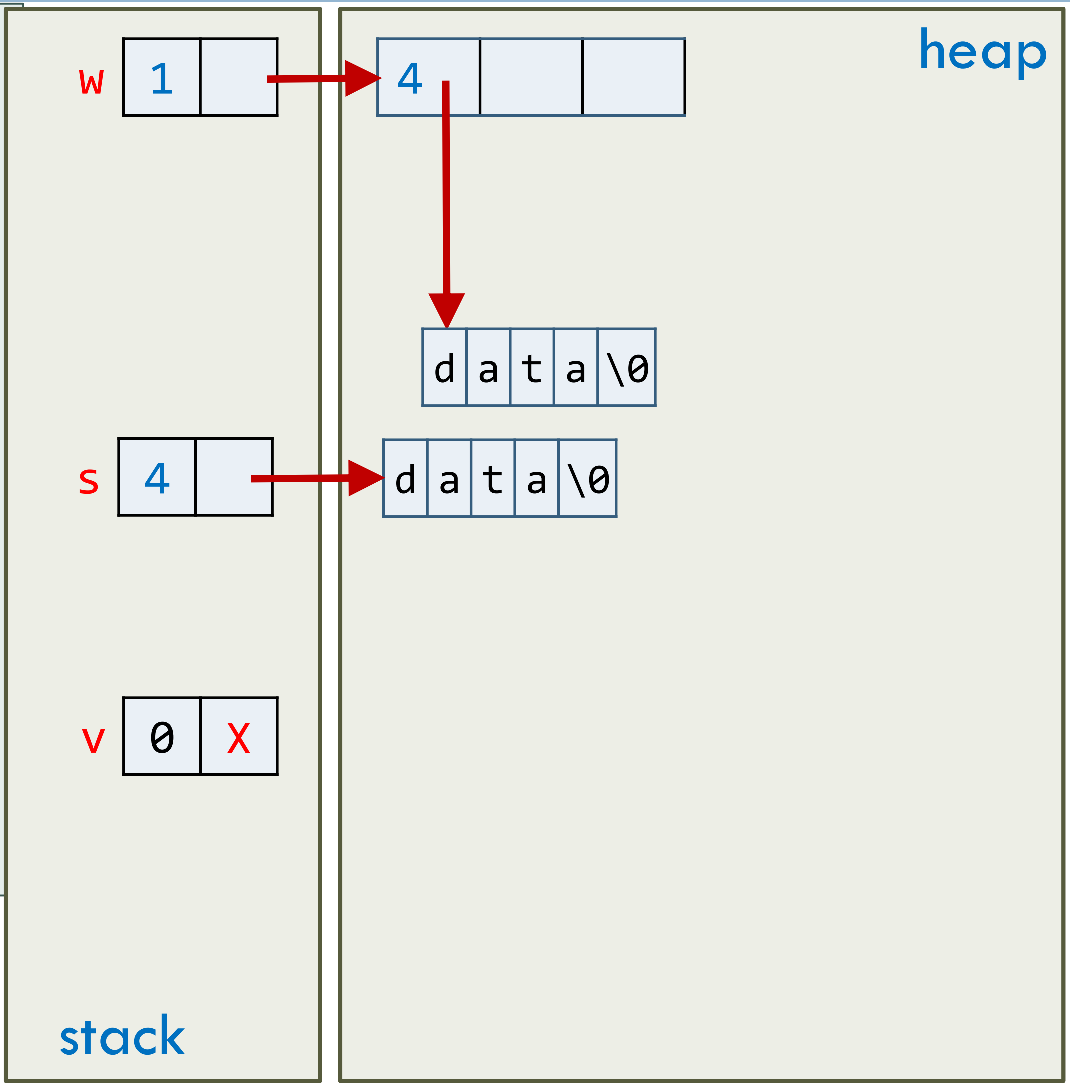
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (4/19)



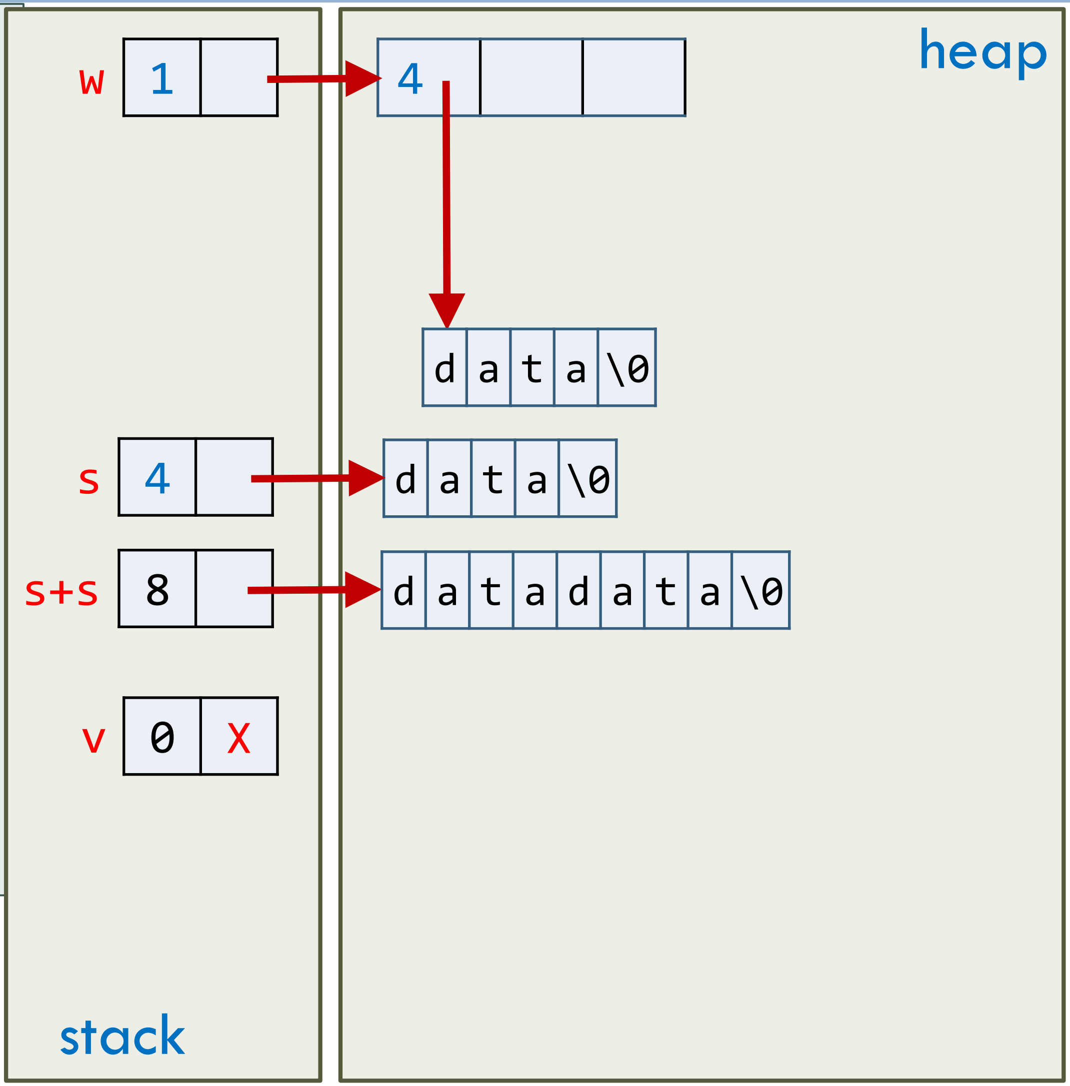
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (5/19)



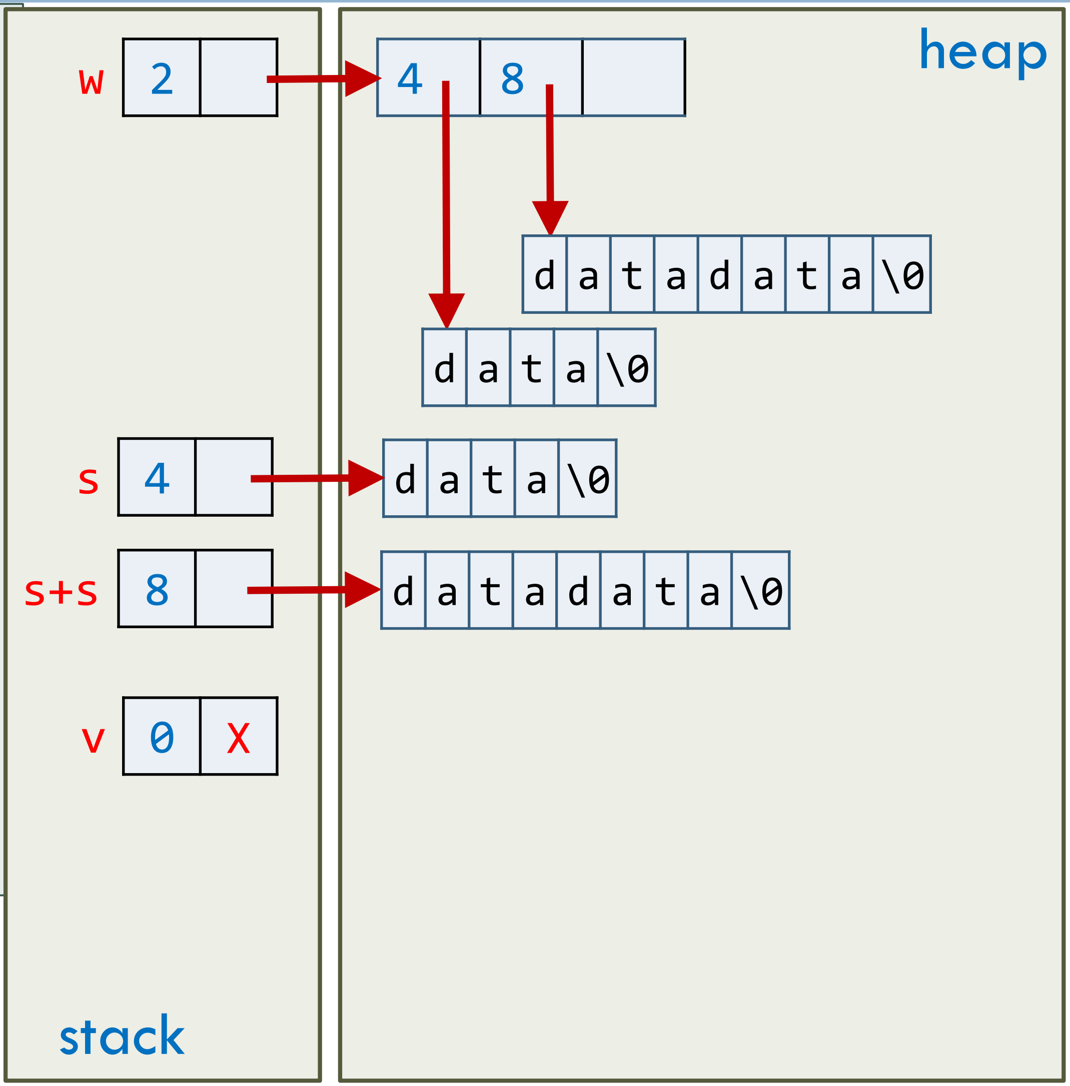
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (6/19)



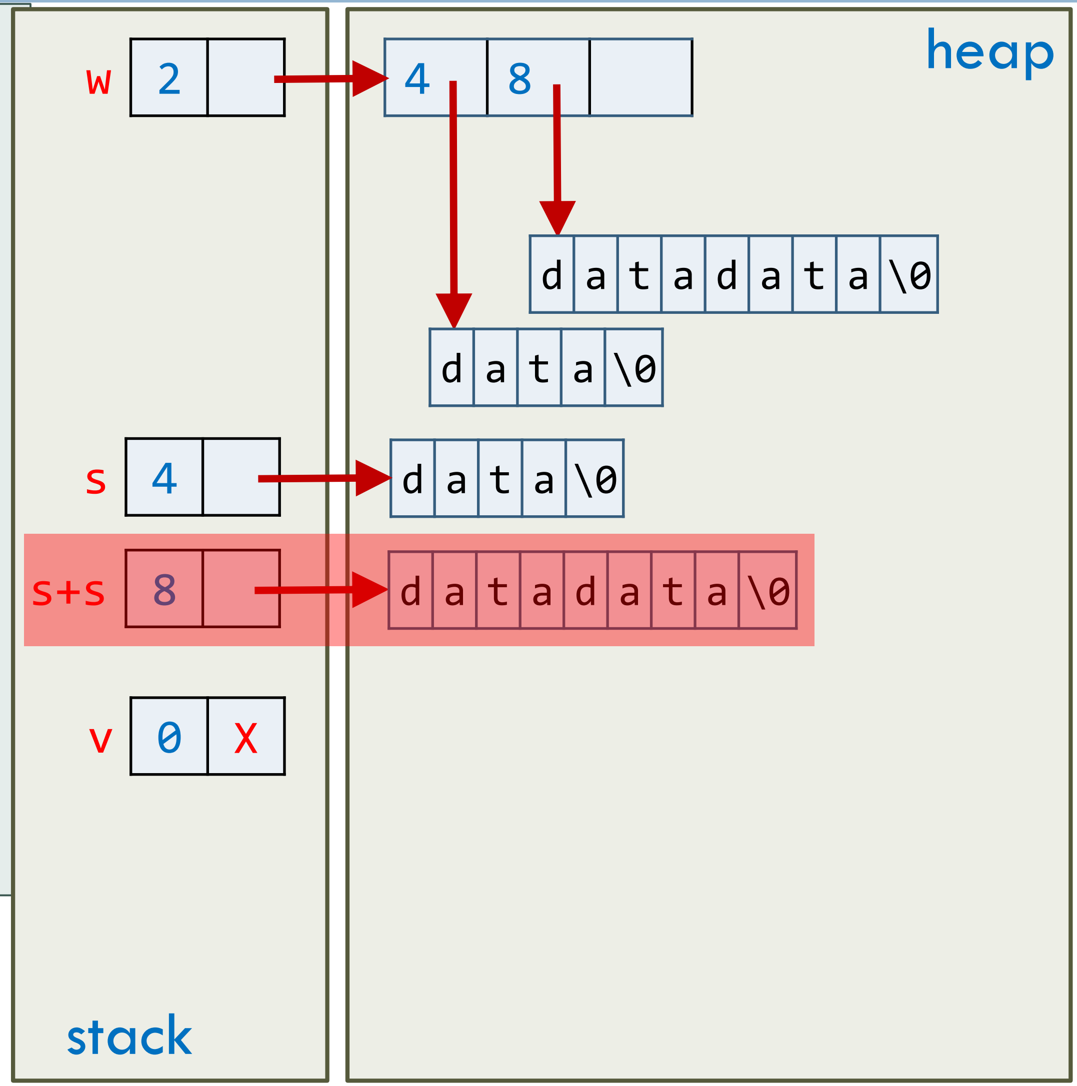
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (7/19)



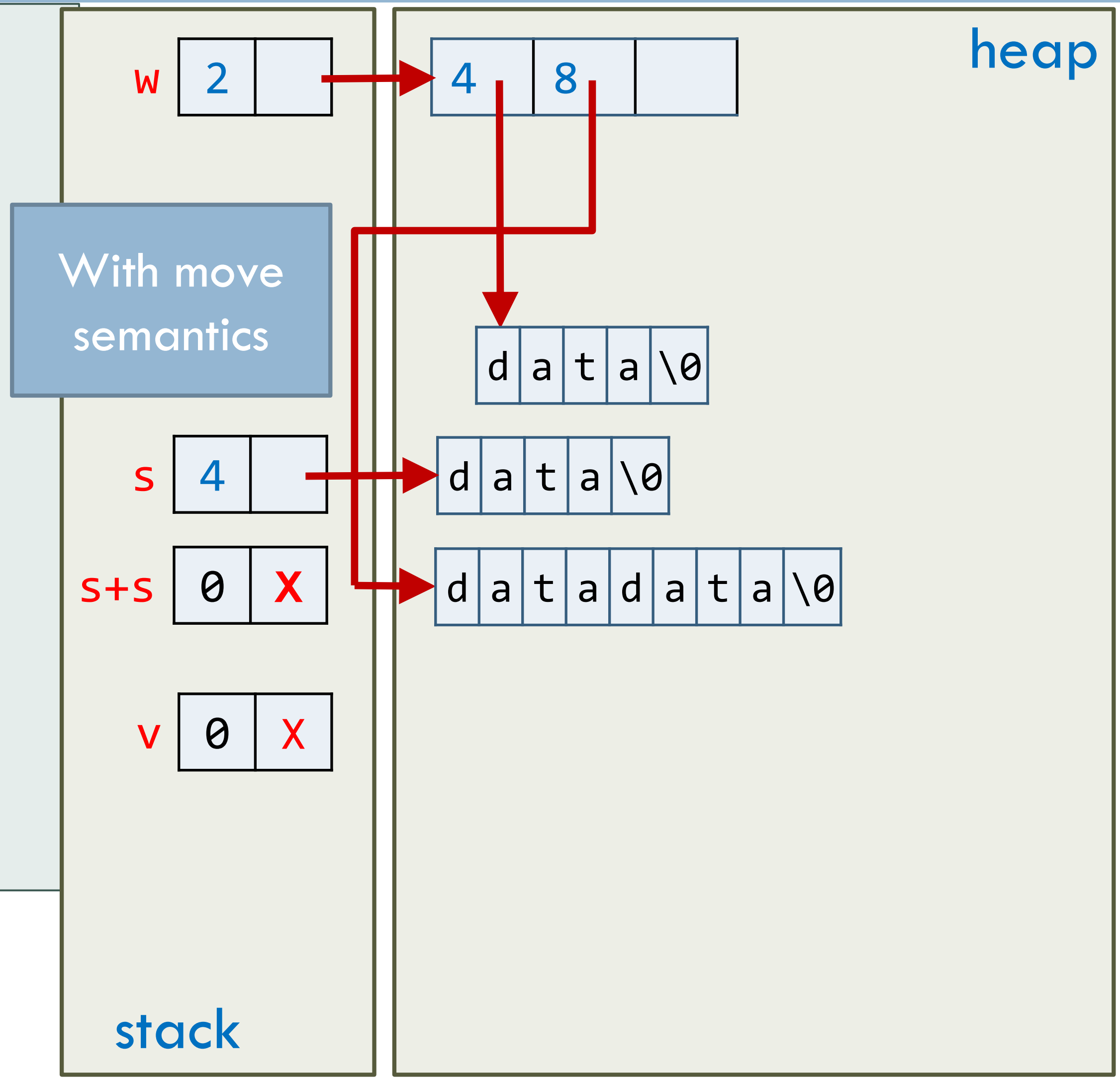
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (8/19)



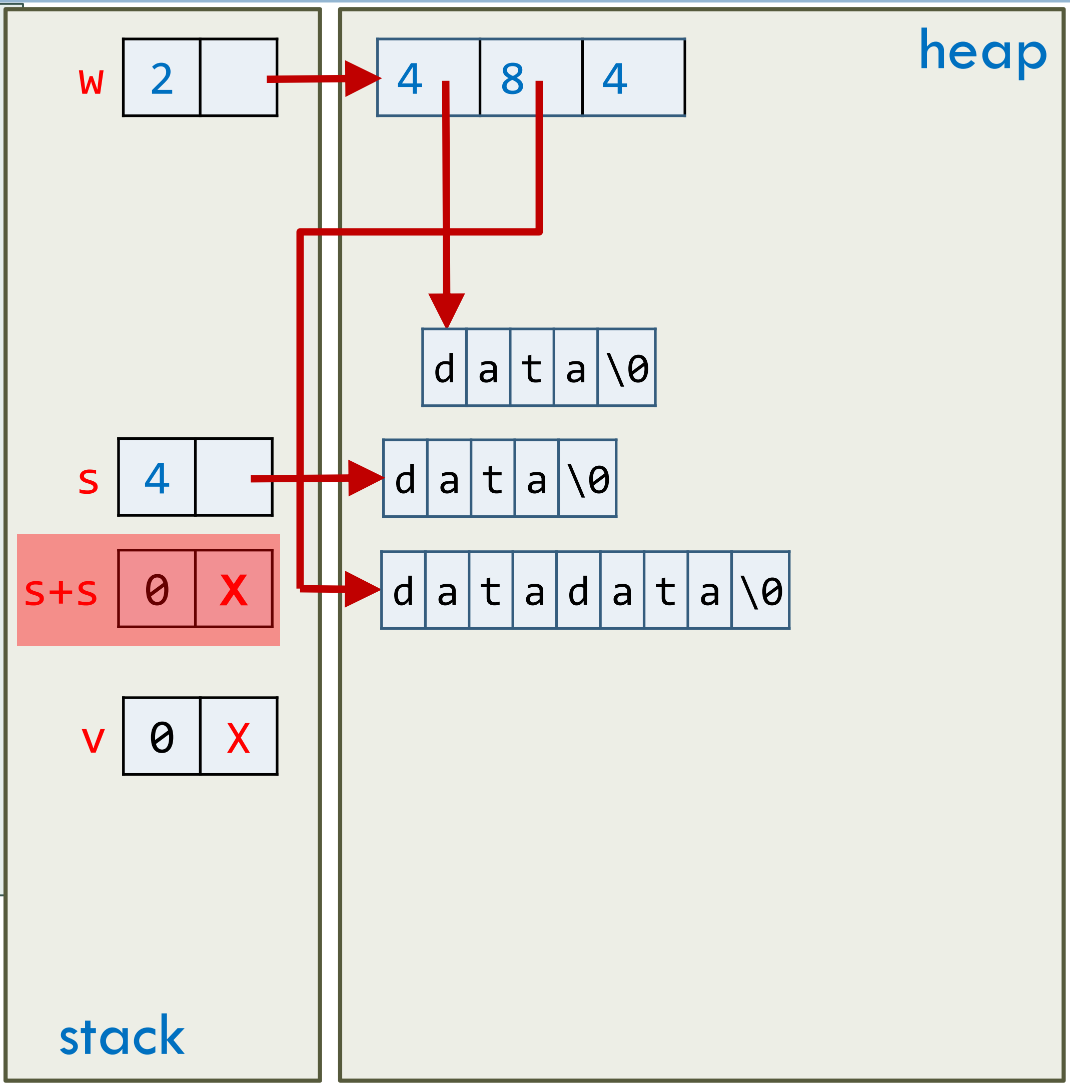
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (9/19)



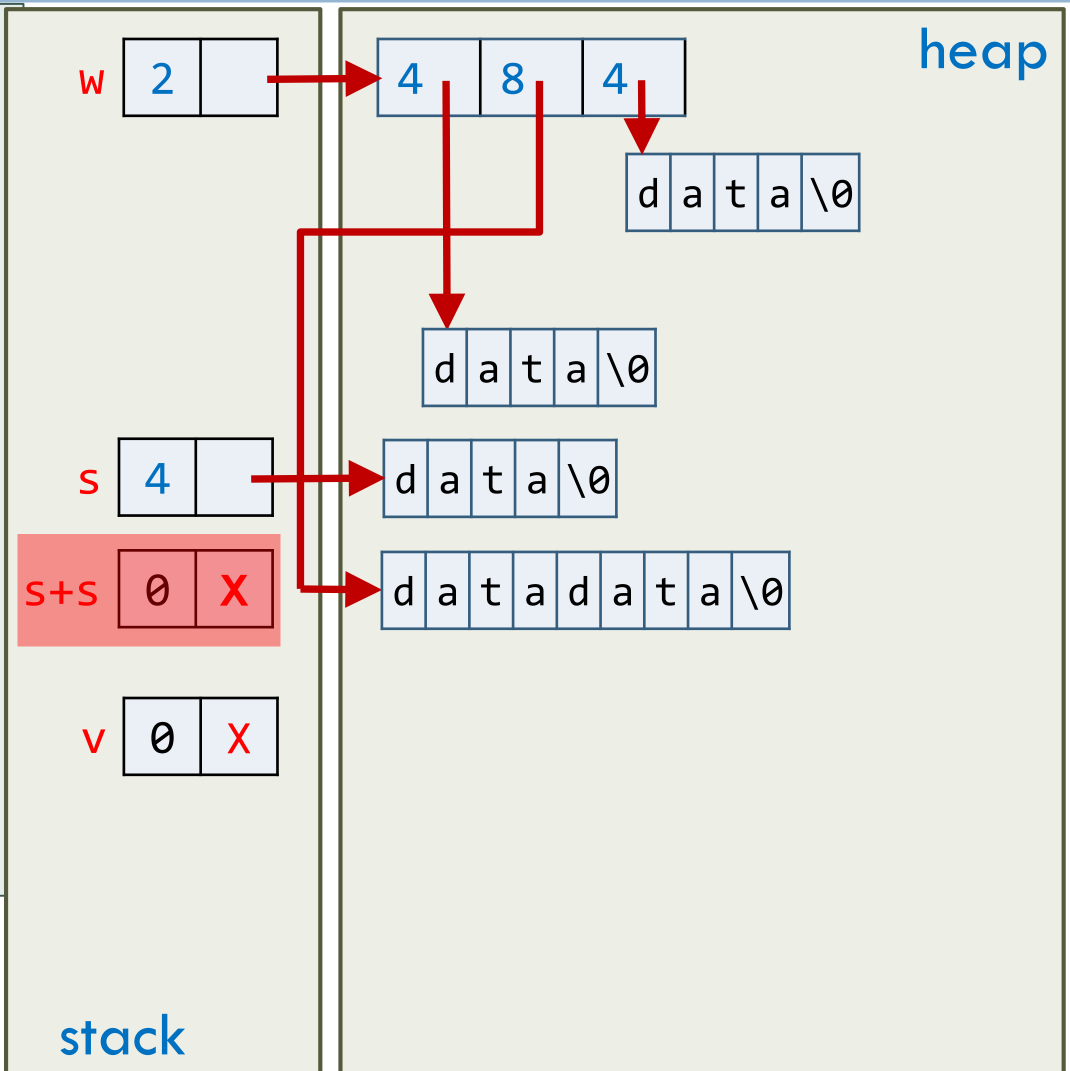
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (10/19)



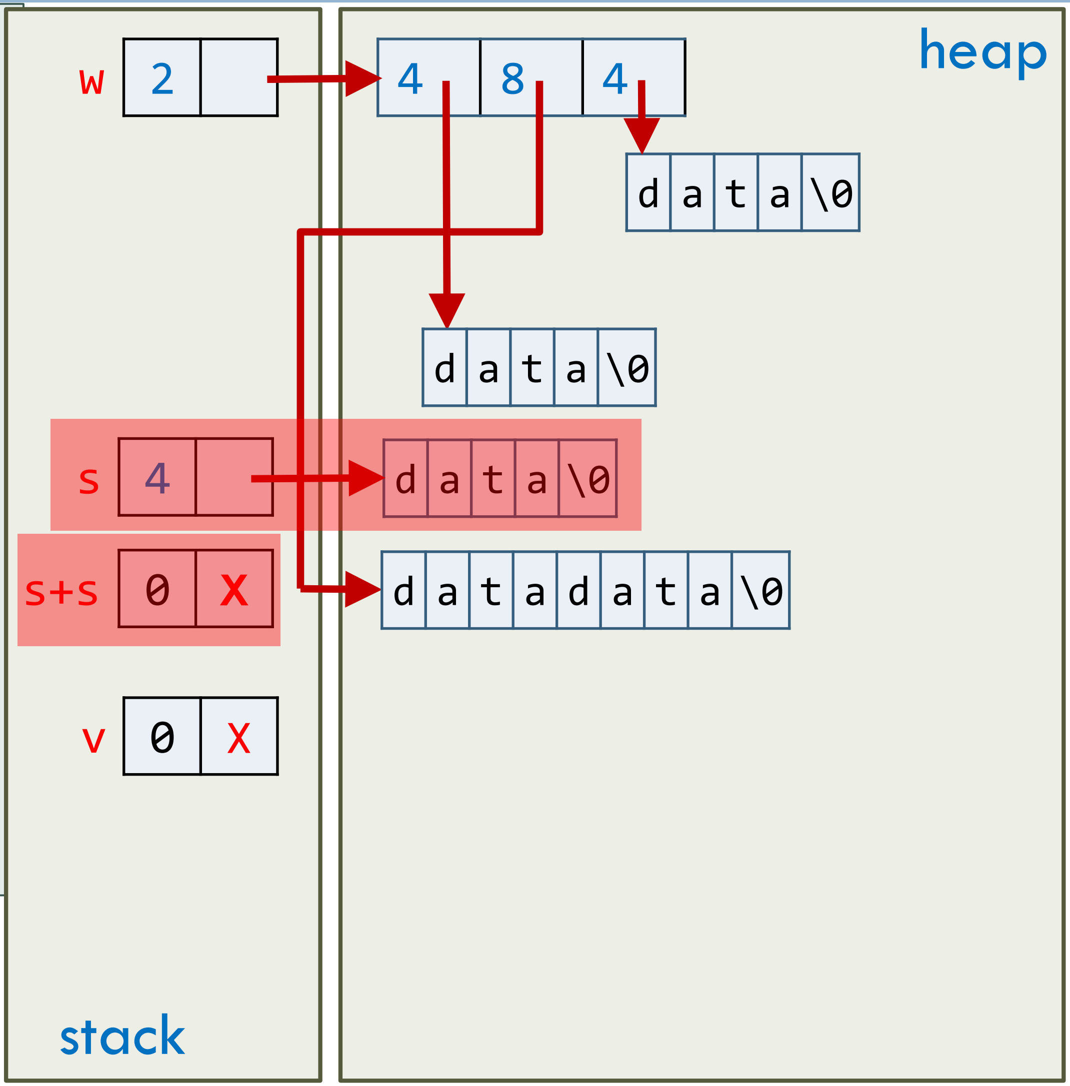
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (11/19)



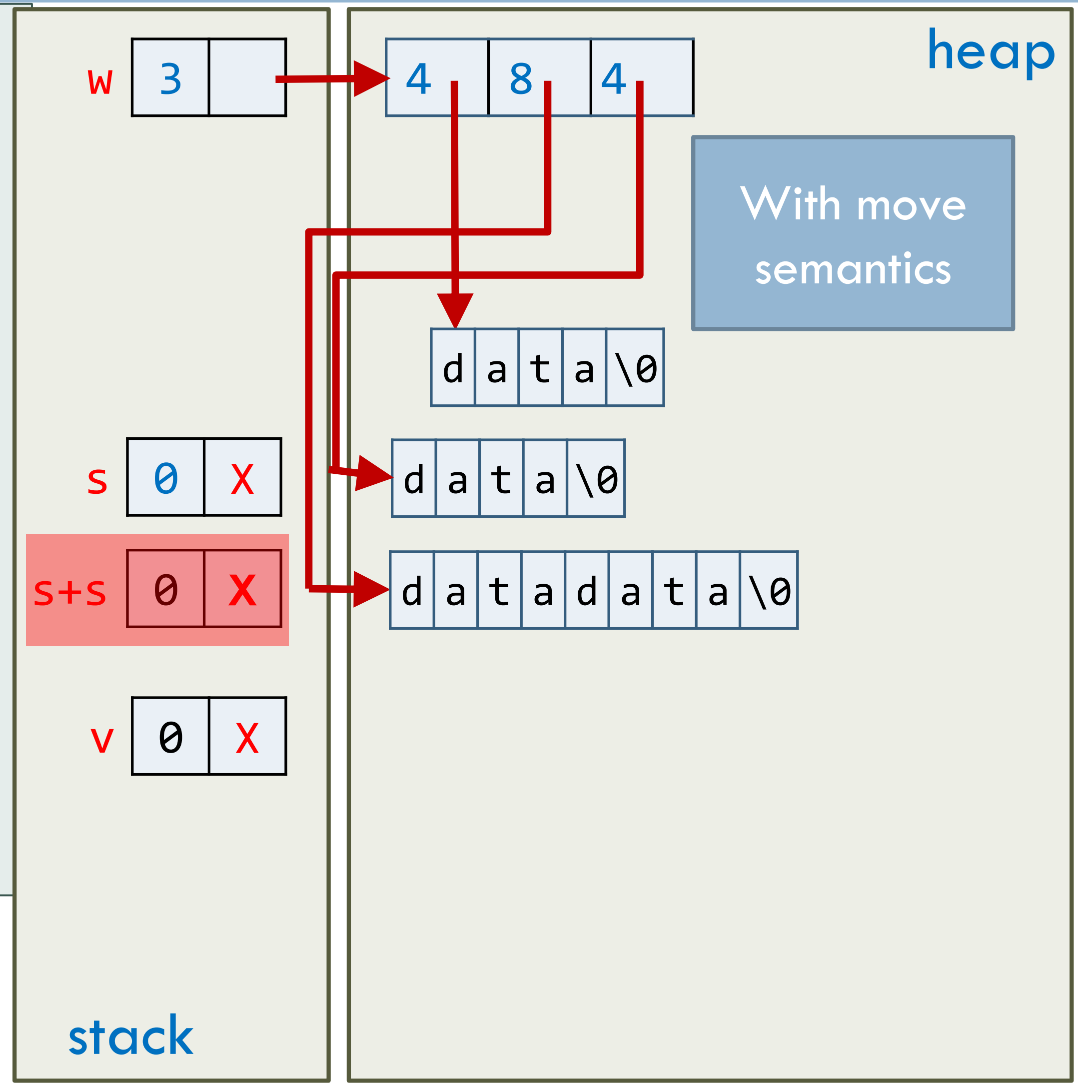
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(s);

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (12/19)



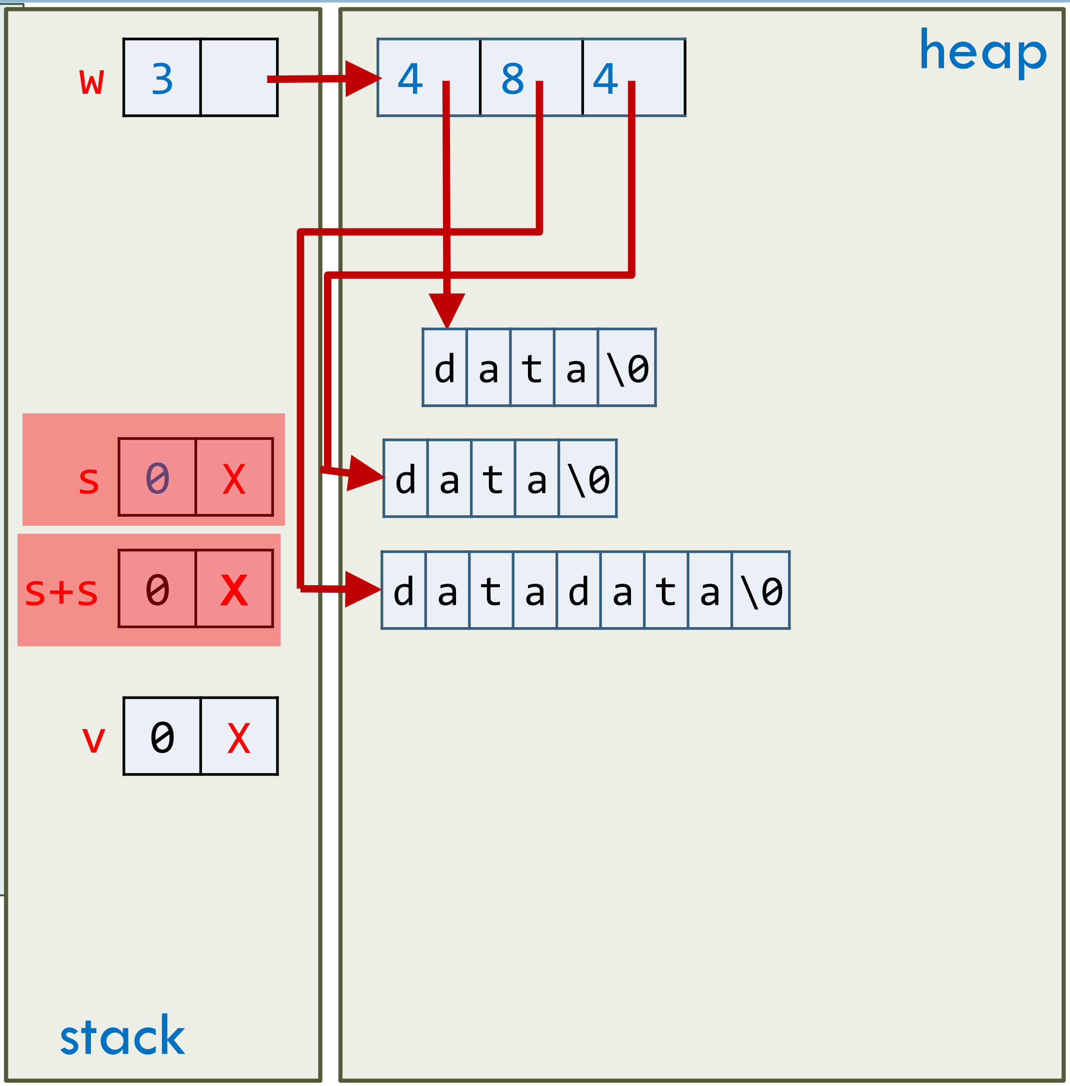
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(std::move(s));

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (13/19)



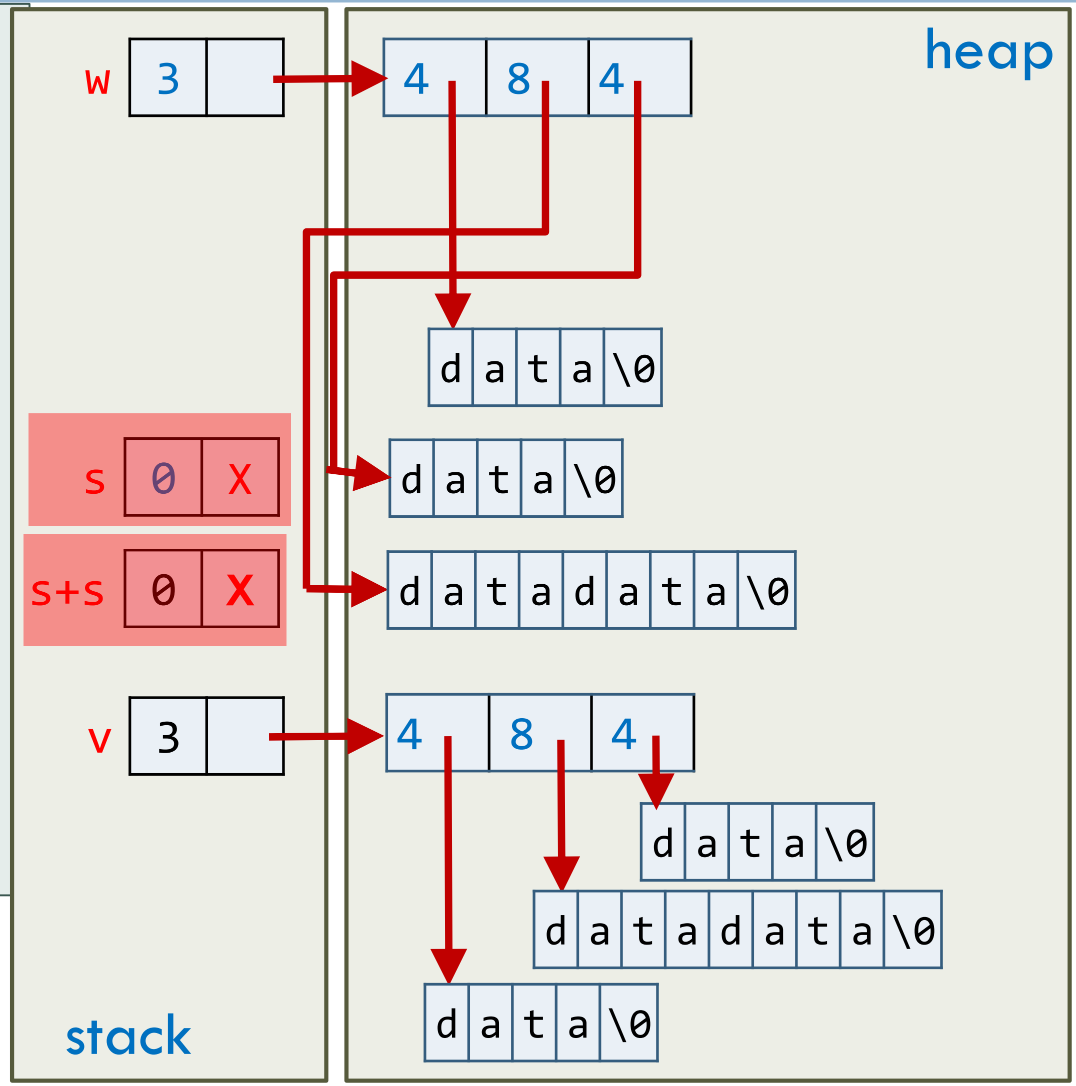
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(std::move(s));

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (14/19)



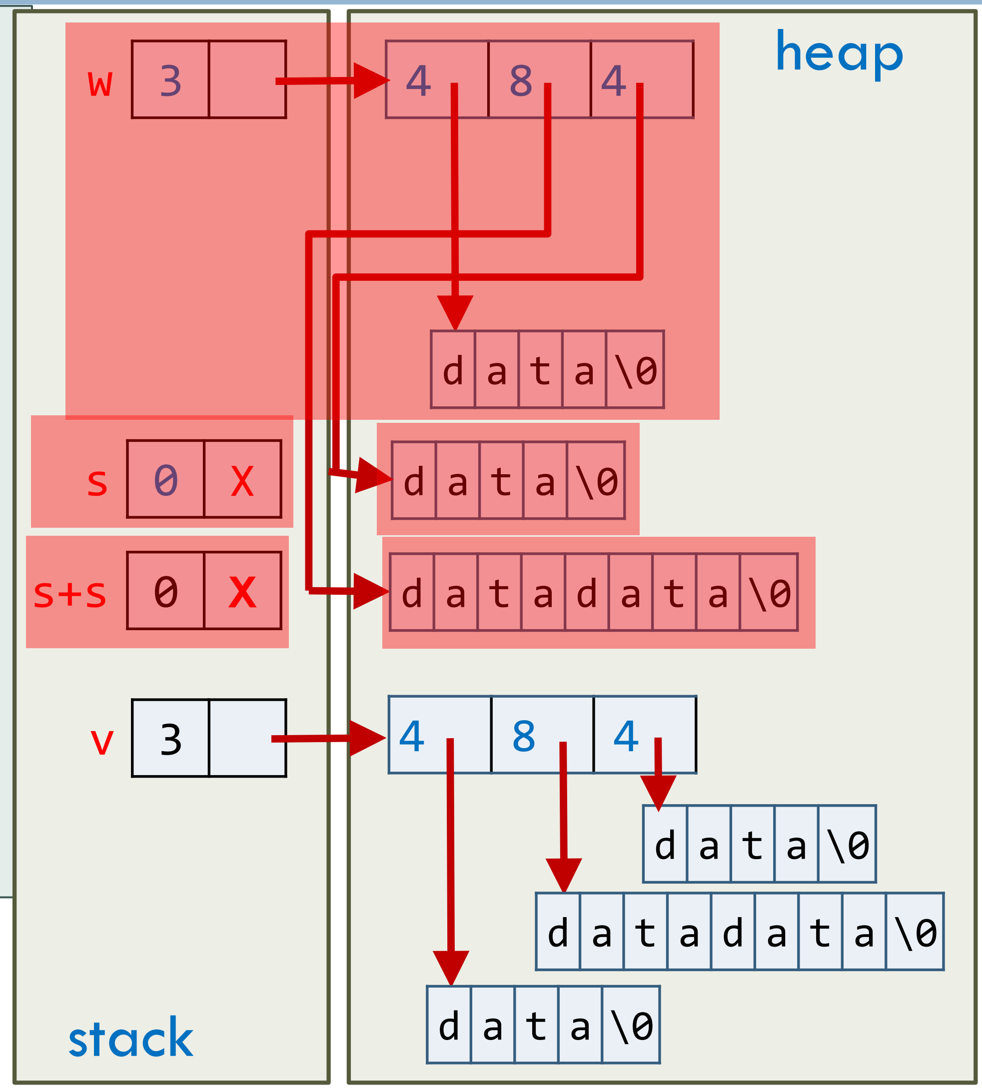
```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(std::move(s));

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (15/19)



```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(std::move(s));

  return w;
}

std::vector<Str> v;

v = f();
```

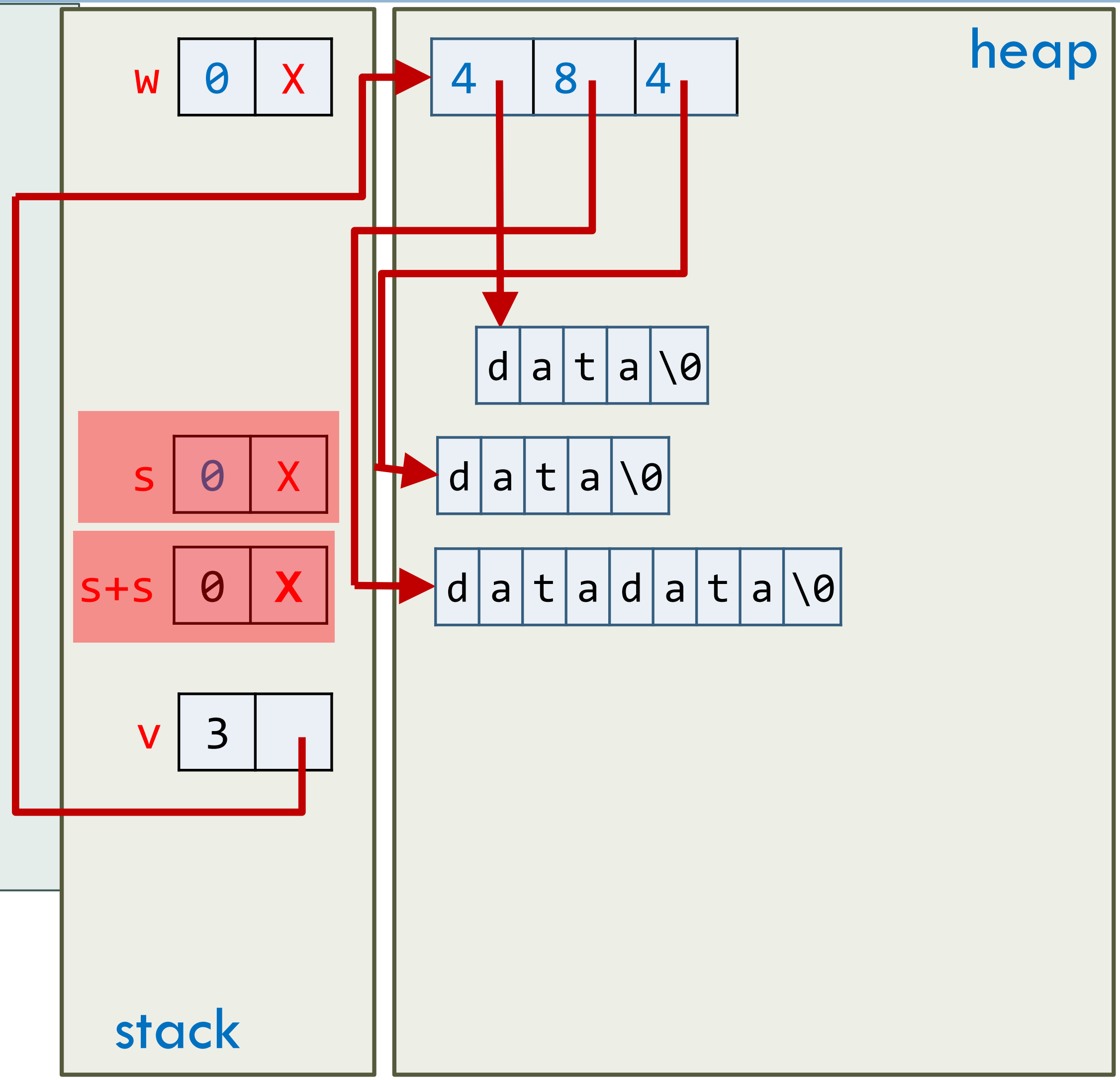


# Motivation for Move Semantics (16/19)



```cpp
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(std::move(s));

  return w;
}

std::vector<Str> v;

v = f();
```

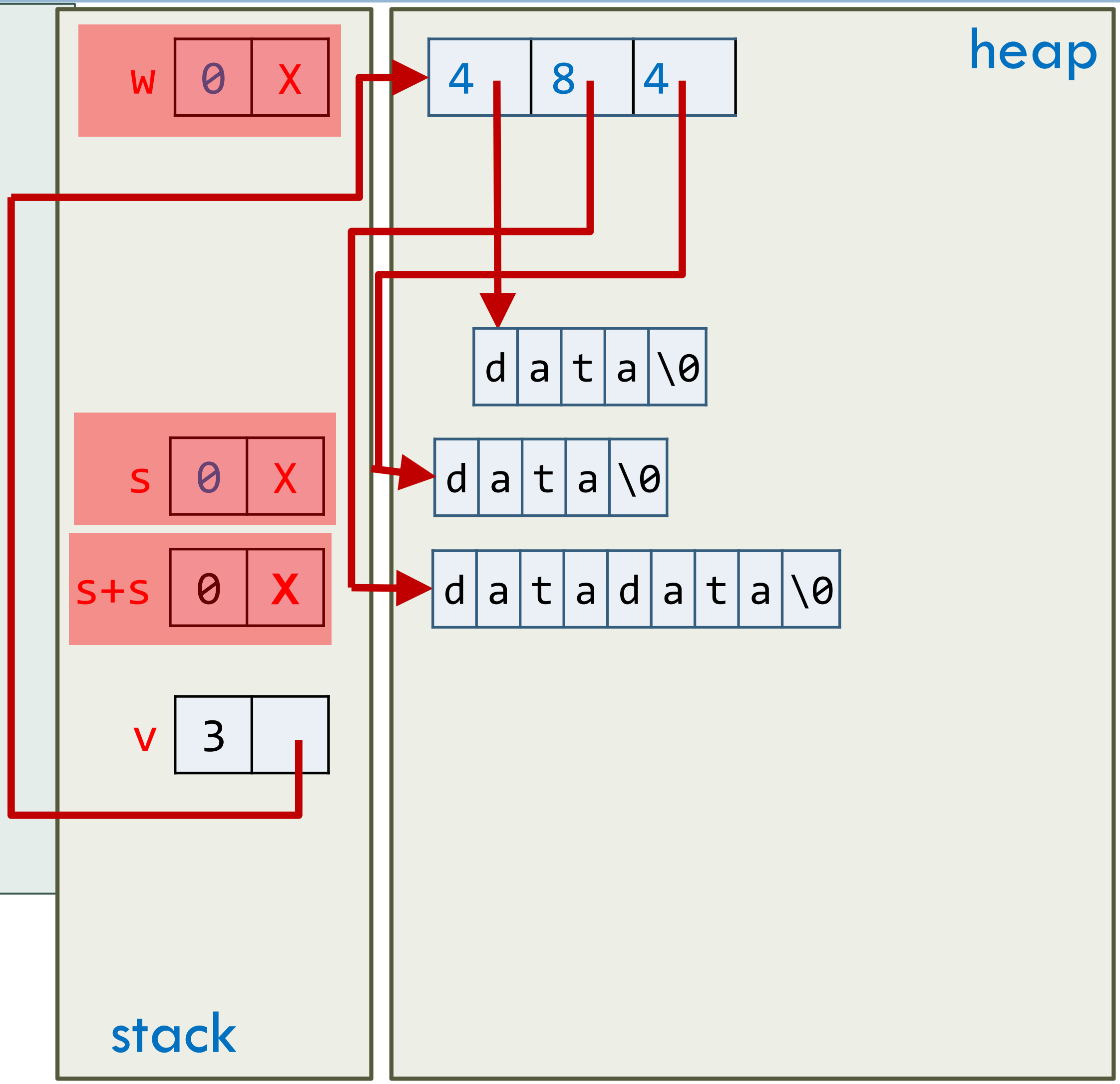
With move semantics



# Motivation for Move Semantics (17/19)

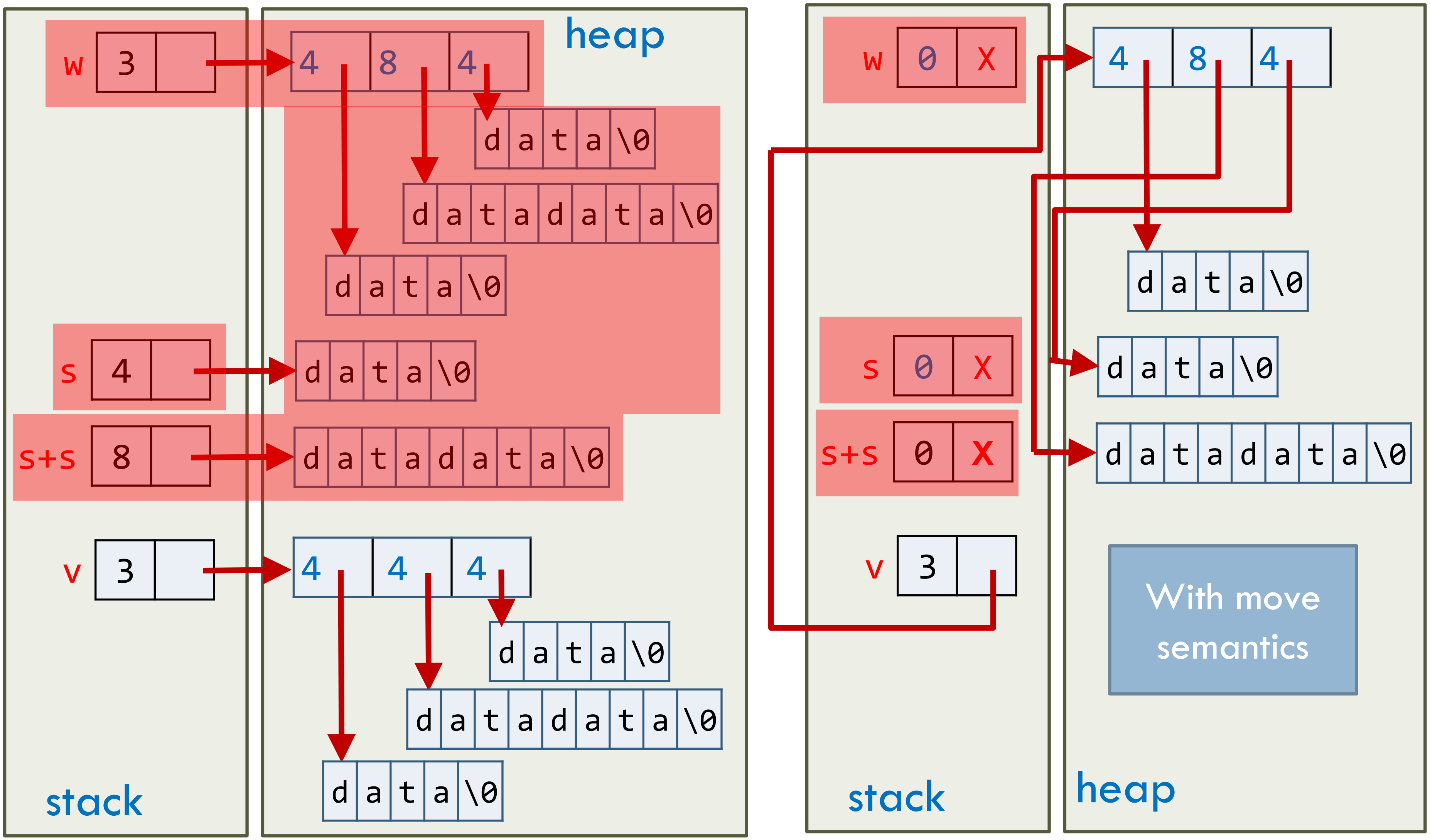


```
std::vector<Str> f() {
  std::vector<Str> w;
  w.reserve(3);
  Str s = "data";

  w.push_back(s);
  w.push_back(s+s);
  w.push_back(std::move(s));

  return w;
}

std::vector<Str> v;

v = f();
```



# Motivation for Move Semantics (18/19)





# Motivation for Move Semantics (19/19)



- At the end, with move semantics we're in same state as without using move semantics, but lots of copies have been removed

# What Is Needed From Language?



- C++ must recognize move opportunities and take advantage of them
  - How to recognize?
  - How to take advantage?

# What C++ Can Do Now? (1/5)



- Copy assignment operator for class Str [a typical handle class] looks like this:

```
Str& Str::operator=(Str const& rhs) {
  // ...
  // 1) make clone of what rhs.ptr refers to
  // 2) destruct resource that ptr refers to
  // 3) attach clone to ptr
  // ...
}
```

# What C++ Can Do Now? (2/5)



- Similar reasoning applies to copy ctor

```
Str:: Str(Str const& rhs) {
  // ...
  // 1) make clone of what rhs.ptr refers to
  // 2) attach clone to ptr
  // ...
}
```

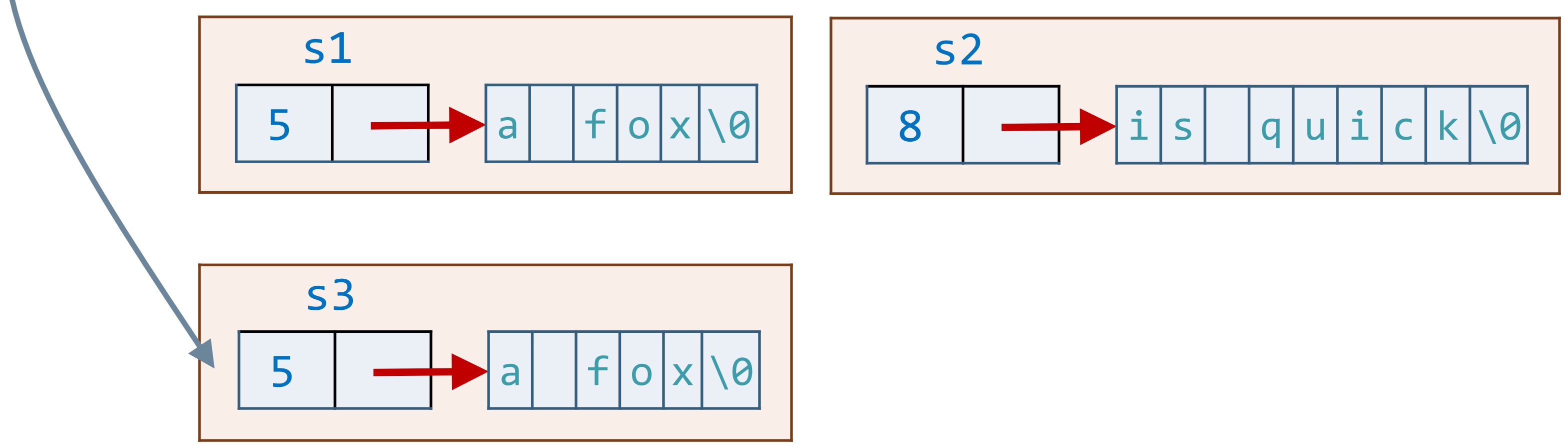
# What C++ Can Do Now? (3/5)



- When we do this

```
Str s1{"a fox"}, s2{"is quick"};
Str s3{s1};
s3 = s2;
```

- We want



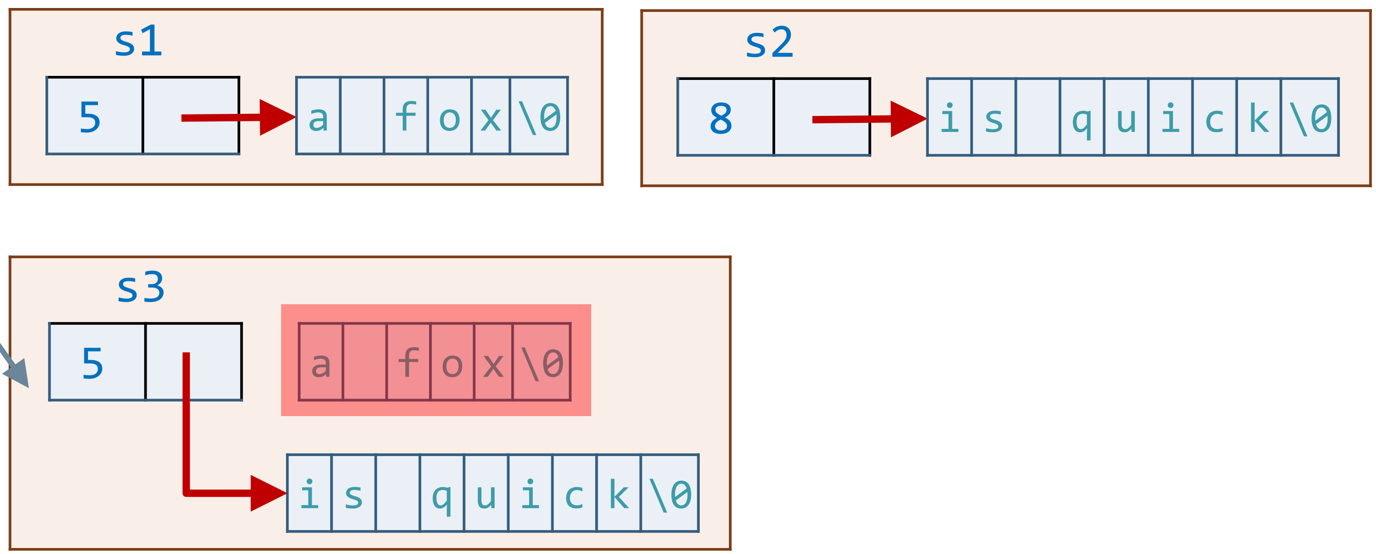
# What C++ Can Do Now? (4/5)



- When we do this

```
Str s1{"a fox"}, s2{"is quick"};
Str s3{s1};
s3 = s2;
```

- We want



# What C++ Can Do Now? (5/5)



```
Str operator+(Str const& lhs, Str const& rhs) {
  Str tmp{lhs};
  tmp += rhs;
  return tmp;
}

Str operator+(Str const& lhs, char const *rhs) {
  Str tmp{lhs};
  tmp += rhs;
  return tmp;
}
```

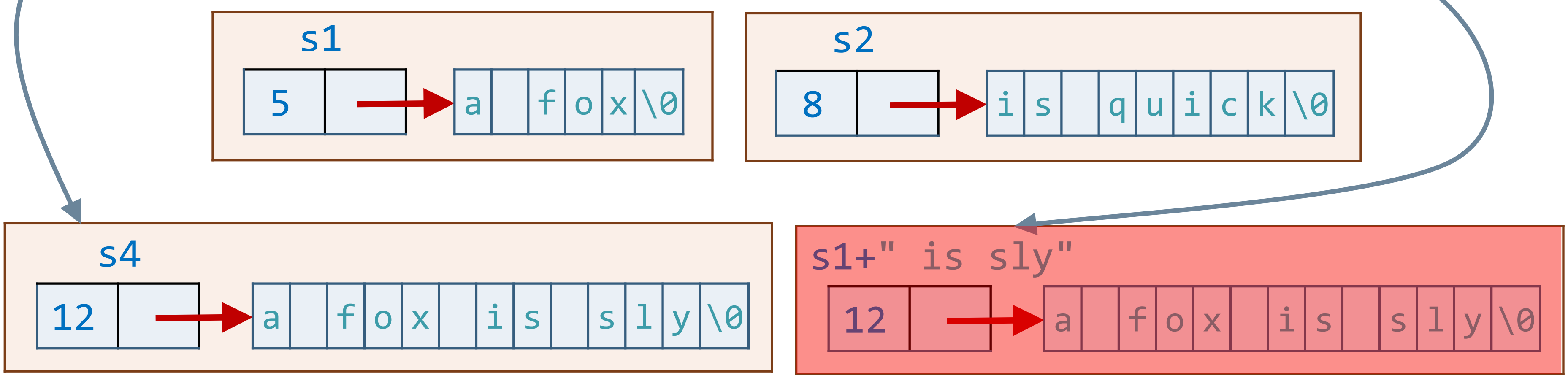
# What We Want From C++? (1/7)



- And when we do this

```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```

- We don't want



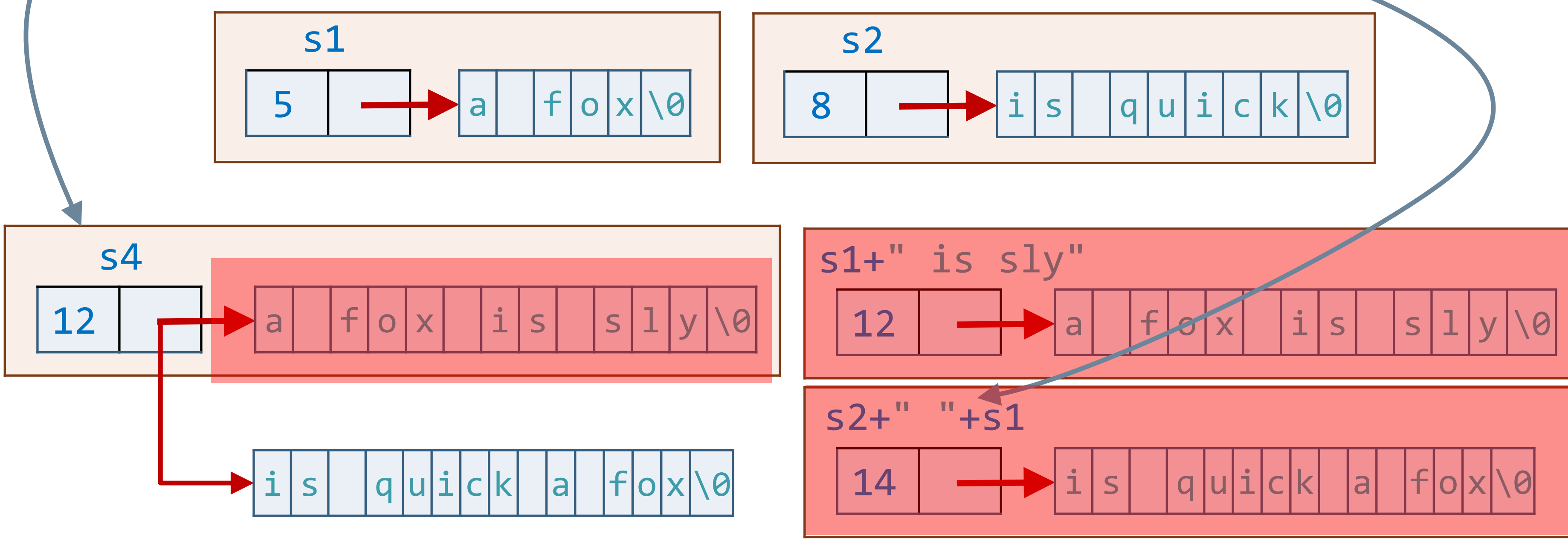
# What We Want From C++? (2/7)



- And when we do this

```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```

- We don't want



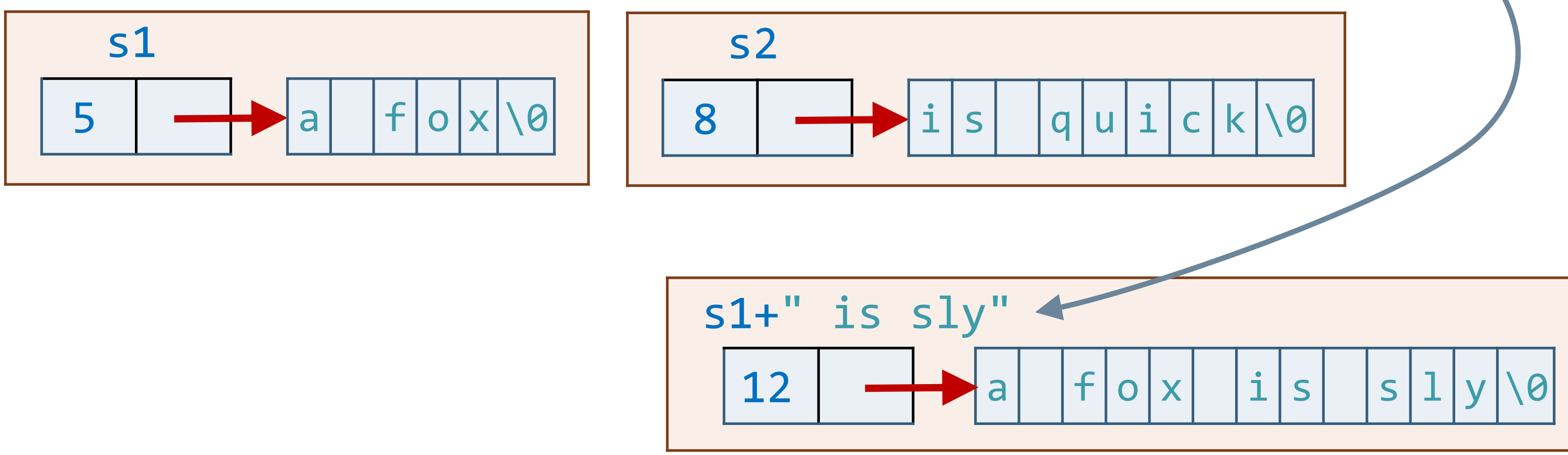
# What We Want From C++? (3/7)



- Instead,

```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```

- We want



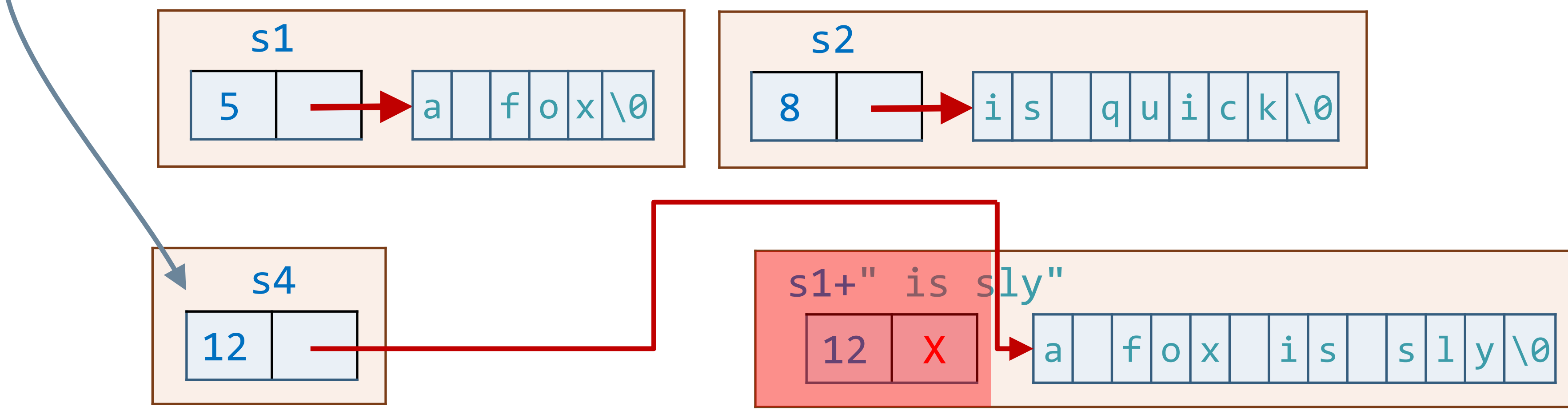
# What We Want From C++? (4/7)



- Instead,

```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```

- We want



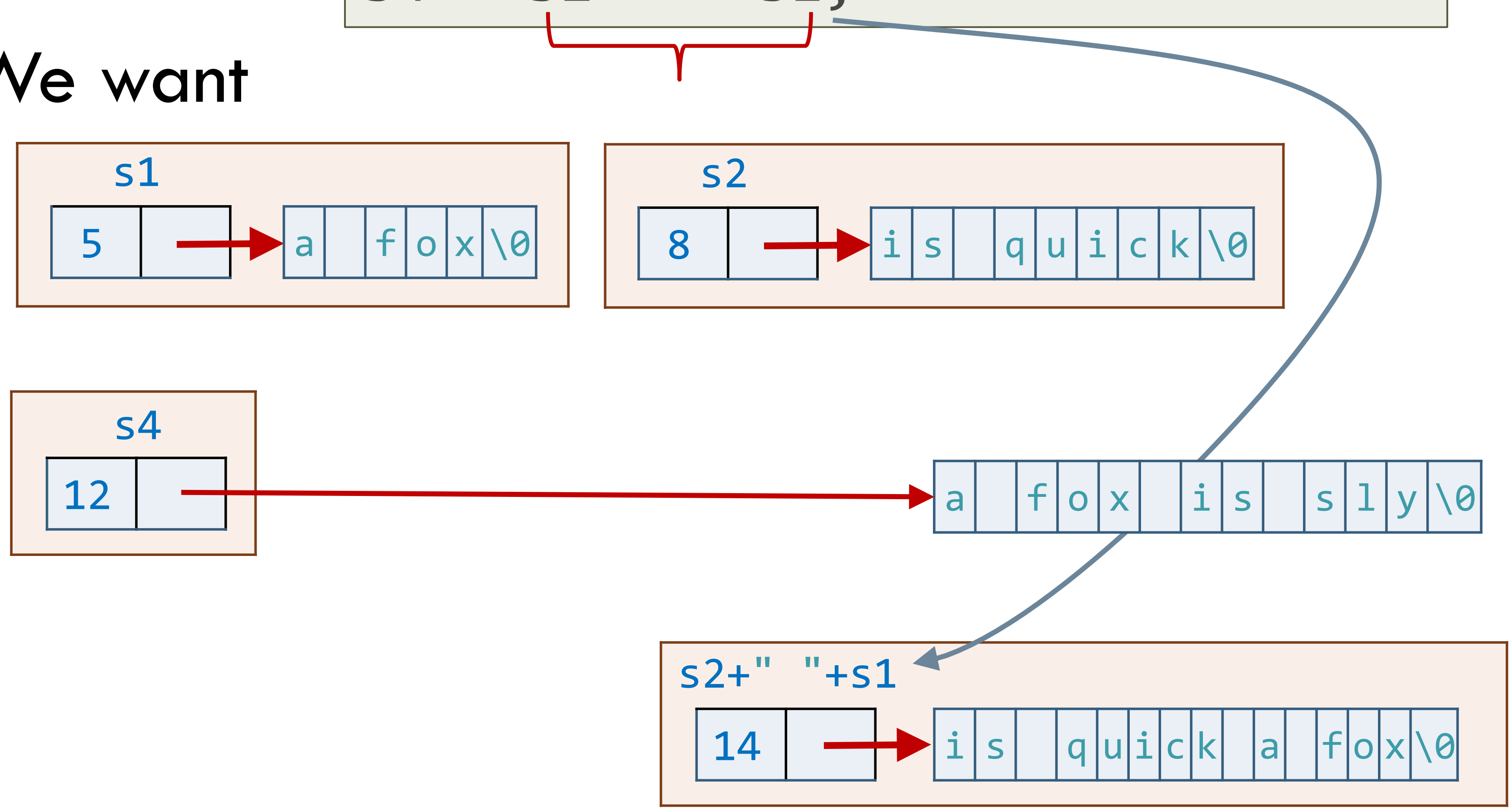
# What We Want From C++? (5/7)



- Instead,

```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```

- We want



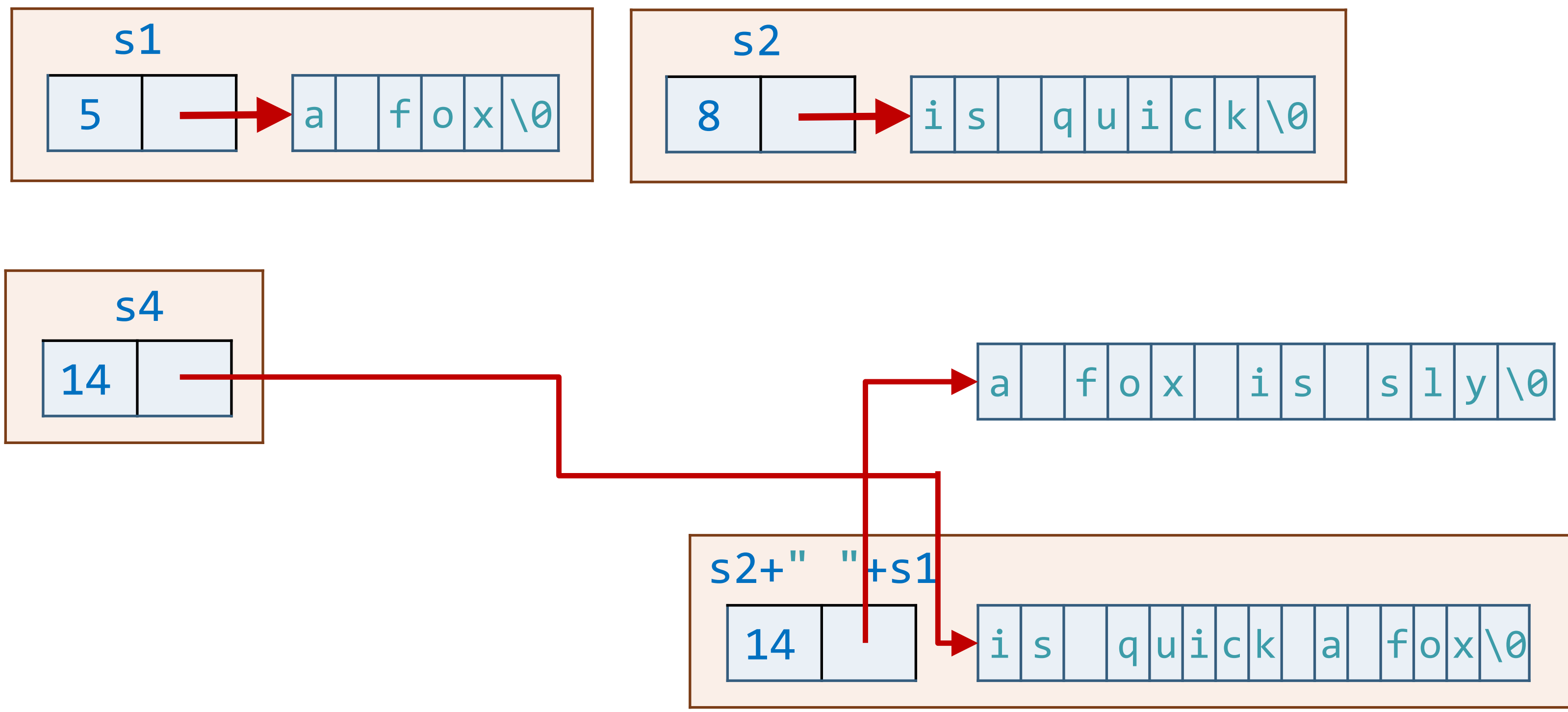
# What We Want From C++? (6/7)



- Instead,

```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```

- We want



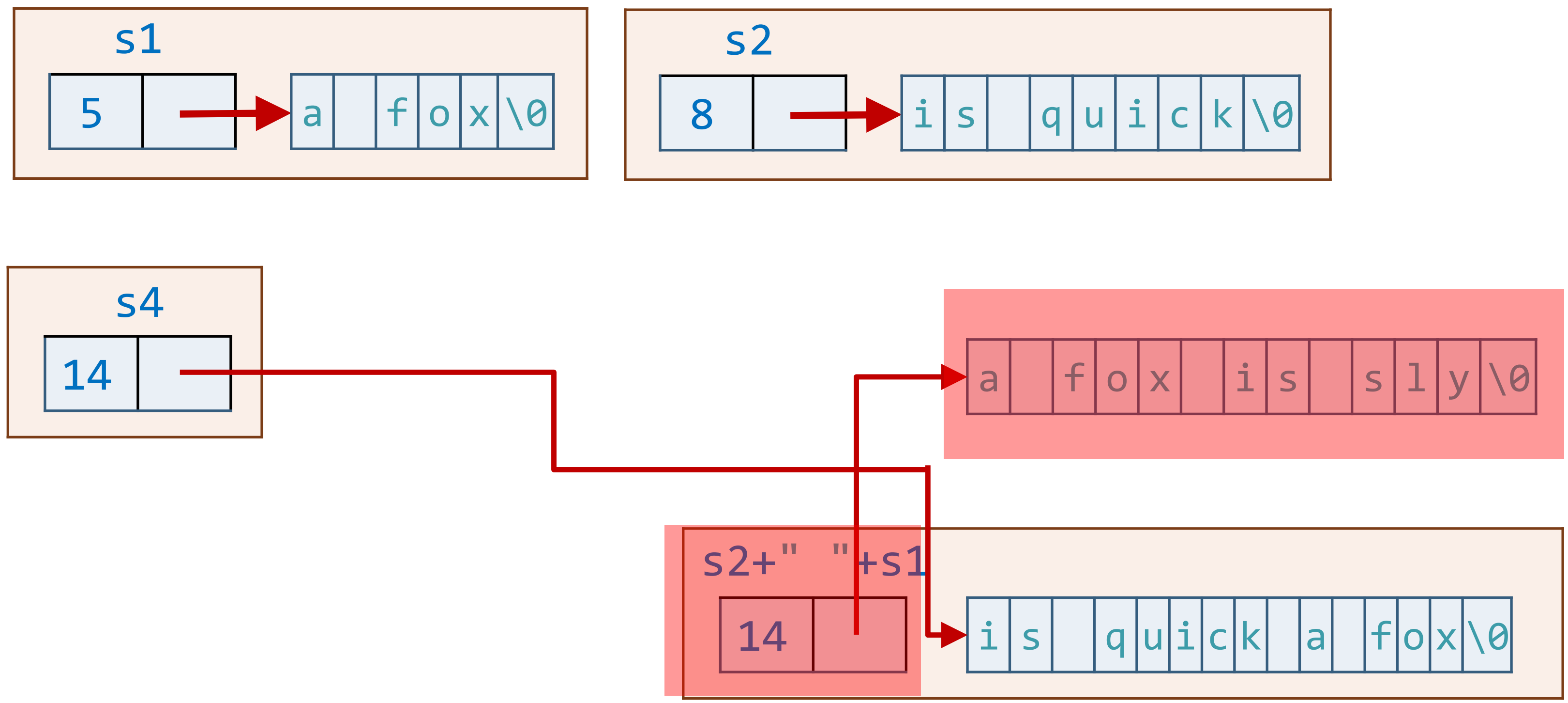
# What We Want From C++? (7/7)



- Instead,

```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```

- We want



# Move and *lvalues*



- Moving is dangerous when source is *lvalue*
  - *lvalue* is persistent and will continue to exist after move
  - May be referred to after move

```
std::string s1{"a fox"}, s2{"is quick"};
std::string s3{s1}; // author expects s1 to
                    // be deep copied to s3
s2 = s3; // s3 is also deep copied to s2
         // after s2 destructs its resource
// ...
// s1, s2, and s3 continue to be used
```

# Move and *rvalues*



- Moving is safe when source is *rvalue*
  - Source is bound to temporary
  - Temporary will not be used again because it will evaporate at end of statement

```
std::string s1{"a fox"}, s2{"is quick"};
std::string s4{s1+" is sly"}; // rvalue source: move ok
s4 = s2+" "+s1; // rvalue source: move ok
std::string s5{s2}; // lvalue source: copy needed
s2 = s4; // lvalue source: copy assignment needed
// ...
// s1, s2, s4 and s5 continue to be used
```

# What Is Needed From C++? (1/4)



- When source is *lvalue*, we want to provide usual copy ctor and copy assignment for some handle class X:

```
X(X const& rhs)
: member initializer list {
  // usual copy semantics
}

X& operator=(X const& rhs) {
  // usual copy semantics
}
```

# What Is Needed From C++? (2/4)



- In special case when source is *rvalue*, we want to provide move ctor and move assignment:

```
X(something that is rvalue)
: member initializer list {
  // move source resources to *this
  // set source resources to null state
}

X& operator=(something that is rvalue) {
  // exchange resources between source and *this
}
```

# What Is Needed From C++? (3/4)



- We want C++ to provide this conditional behavior via an overload

# What Is Needed From C++? (4/4)



```
X(X const& rhs) // for lvalues and const lvalues
: member initializer list {
  // usual copy semantics
}

X(<mystery type> rhs)  // something that is a rvalue
: member initializer list {
  // move source resources to *this
  // set source resources to null state
}

X& operator=(X const& rhs) { // for lvalues and const lvalues
  // usual copy semantics
}

X& operator=(<mystery type> rhs) {// something that is a rvalue
  // exchange resources between source and *this
}
```

# What is `mystery type`? (1/2)



- Must be some reference type
  - Copy ctor and assignment operator already defined to take `const` *lvalue* reference
- Given two overloaded copy ctors or copy assignment functions where one is *lvalue* reference and other is `mystery type`, overloads must provide following behavior
  - *rvalues* must prefer `mystery type`
  - *lvalues* must prefer `const` *lvalue* reference

# What is `mystery type`? (2/2)



- Given two overloaded copy ctors or copy assignment functions where one is *lvalue* reference and other is `mystery type`, overloads must provide following behavior
  - *rvalues* must prefer `mystery type`
  - *lvalues* must prefer ordinary reference
- C++11 came up with new type for `mystery type` called *rvalue reference*

# *Rvalue* References (1/2)



- If X is any type, then X&& is called *rvalue* reference to X
- Behavior exactly opposite of *lvalue* reference:
  - Can only bind to an *rvalue*, but not to an *lvalue*

# Rvalue References (2/2)



```
std::string var{"iowa"};   // var evaluates to lvalue
std::string f();           // f evaluates to rvalue

std::string& l1 {var};     // bind l1 to lvalue var
std::string& l2 {f()};     // error: f() is rvalue
std::string& l3 {"ohio"};  // error: can't bind to
                           // rvalue [temporary]

std::string&& r1 {var};    // error: var is lvalue
std::string&& r2 {f()};    // bind r2 to rvalue [temporary]
std::string&& r3 {"iowa"}; // bind r3 to rvalue
                           // [unnamed temporary]

std::string const& c1{var};    // bind c1 to lvalue
std::string const& c2{f()};    // bind c2 to rvalue
std::string const& c3{"iowa"}; // bind c3 to rvalue
```

# *Rvalue* References: Main Purpose



- *Rvalue* reference binds to temporary object
- User of reference can [and typically will] modify object assuming it will be vaporized
  - Implement "destructive read" for optimization of what would have required a copy
- `const` *rvalue* references are not used
  - Cannot implement "destructive read" on non-modifiable value

# *Rvalue* References: Access



- Accessed exactly like object referred to by *lvalue* reference or ordinary variable name

```
std::string f(std::string&& s) {
  s[0] = (s.size()) ? toupper(s[0]) : s[0];
  return s;
}
```

# References: Recap



- Idea of having more than one reference type is to support different uses of objects:
  - *Lvalue* references: to refer to objects whose value we want to change [so called *in/out parameters*]
  - `const` *lvalue* references: to refers to objects whose value we don't want to change [*in parameters*]
  - *Rvalue* references: to refer to objects whose value we don't need to preserve after usage [*"will move from" in parameters*]
  - `const` rvalue references: not used

# Implementing Move Ctor (1/2)



- We can define Str's move ctor to simply take representation from its source and replace it with empty Str [which is cheap to destroy]

In modern C++, a function can specify that it doesn't throw exceptions by providing a noexcept specification

```
Str::Str(Str&& rhs) noexcept
    : len(rhs.len), ptr{rhs.ptr} {
  rhs.len = 0;
  rhs.ptr = nullptr;
 }
```

# Implementing Move Ctor (2/2)



```
Str::Str(Str&& rhs) noexcept
      : len(rhs.len), ptr{rhs.ptr} {
   rhs.len = 0;
   rhs.ptr = nullptr;
}
```

After a move operation, the “moved-from” object must remain a valid, destructible object but users may make no assumptions about the value!!!

# Implementing Move Assignment (1/4)



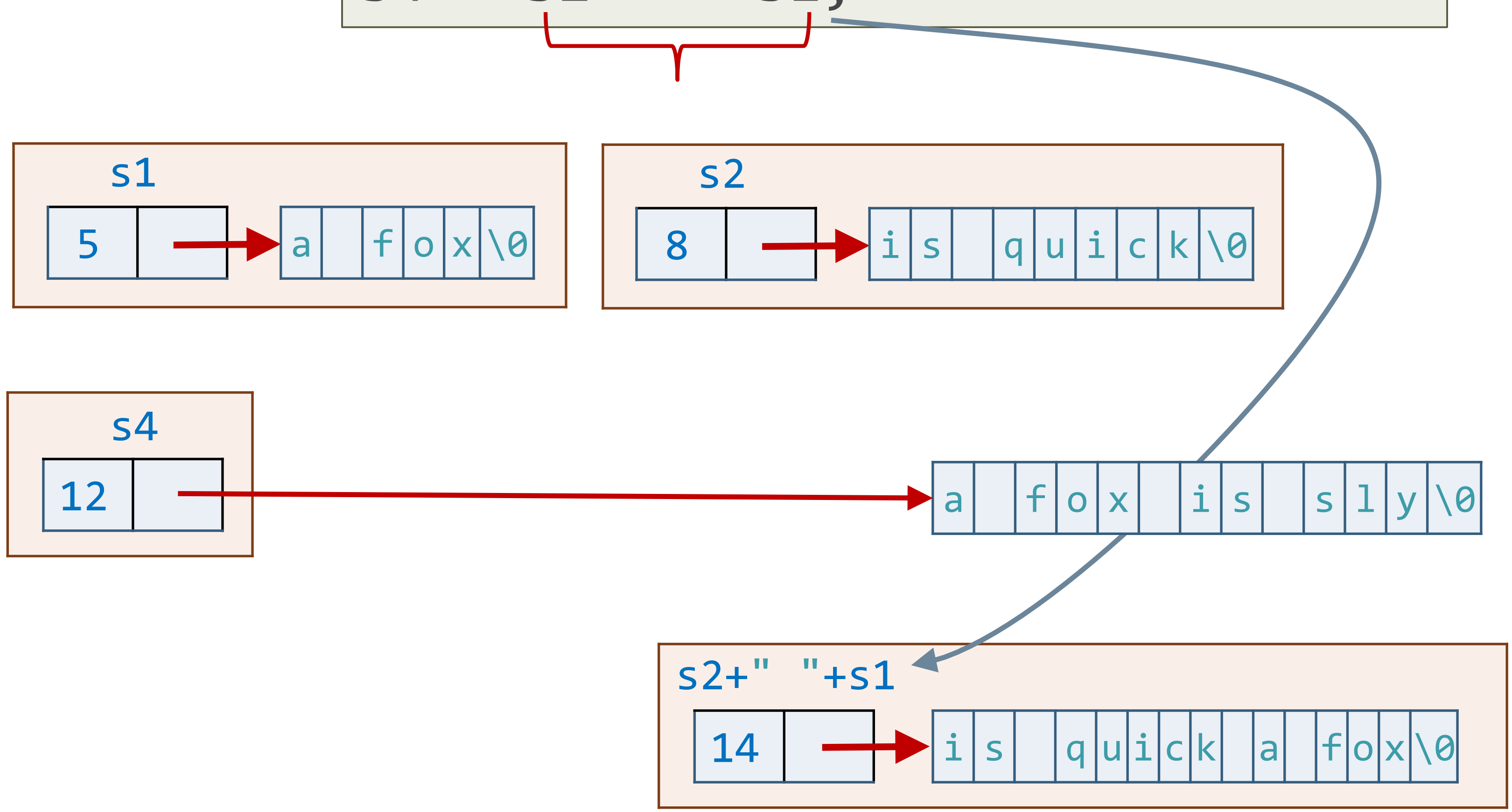
- Idea behind using a swap is that source is just about to be destroyed
  - So just let destructor for source do necessary cleanup work for us

```
Str& Str::operator=(Str&& rhs) noexcept {
  std::swap(len, rhs.len);
  std::swap(ptr, rhs.ptr);
  return *this;
}
```

# Implementing Move Assignment (2/4)



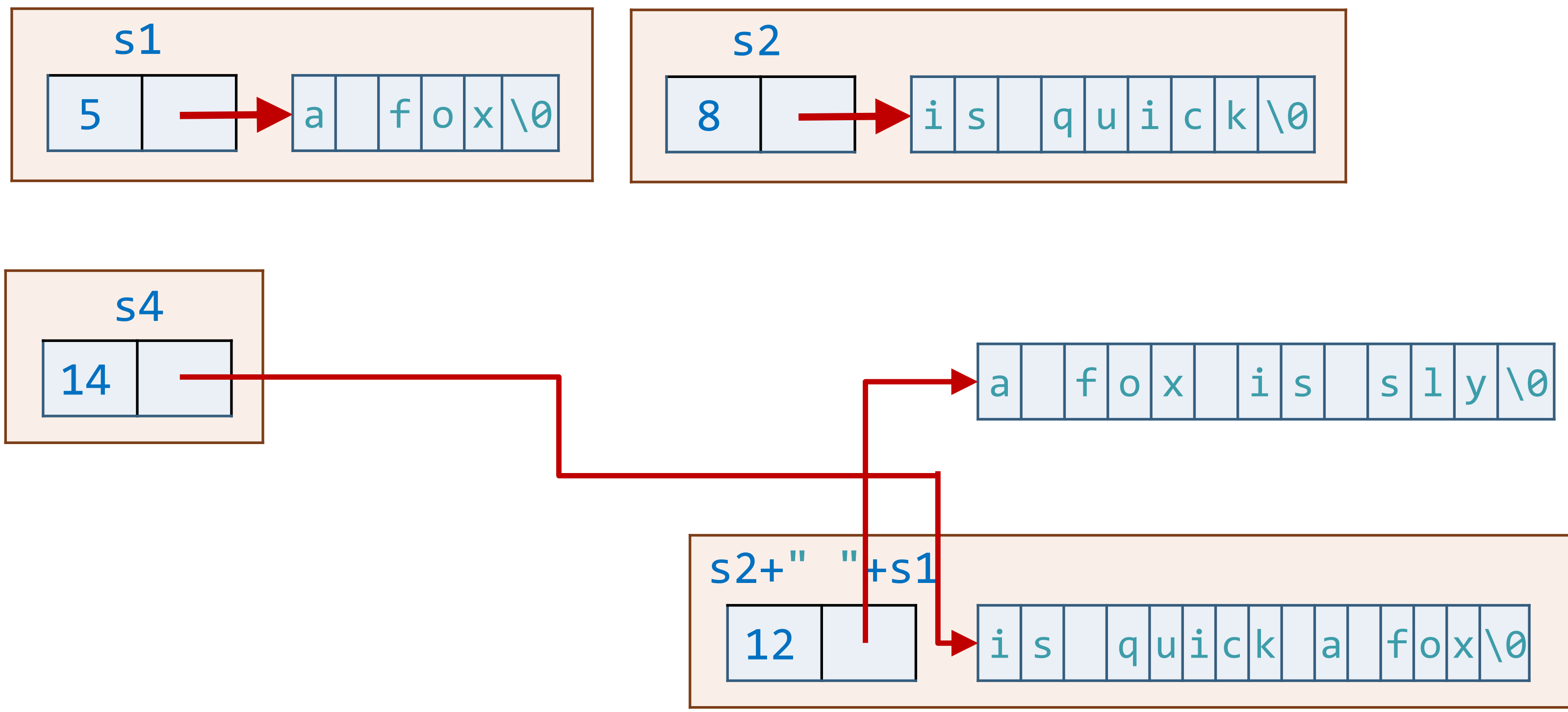
```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```



# Implementing Move Assignment (3/4)



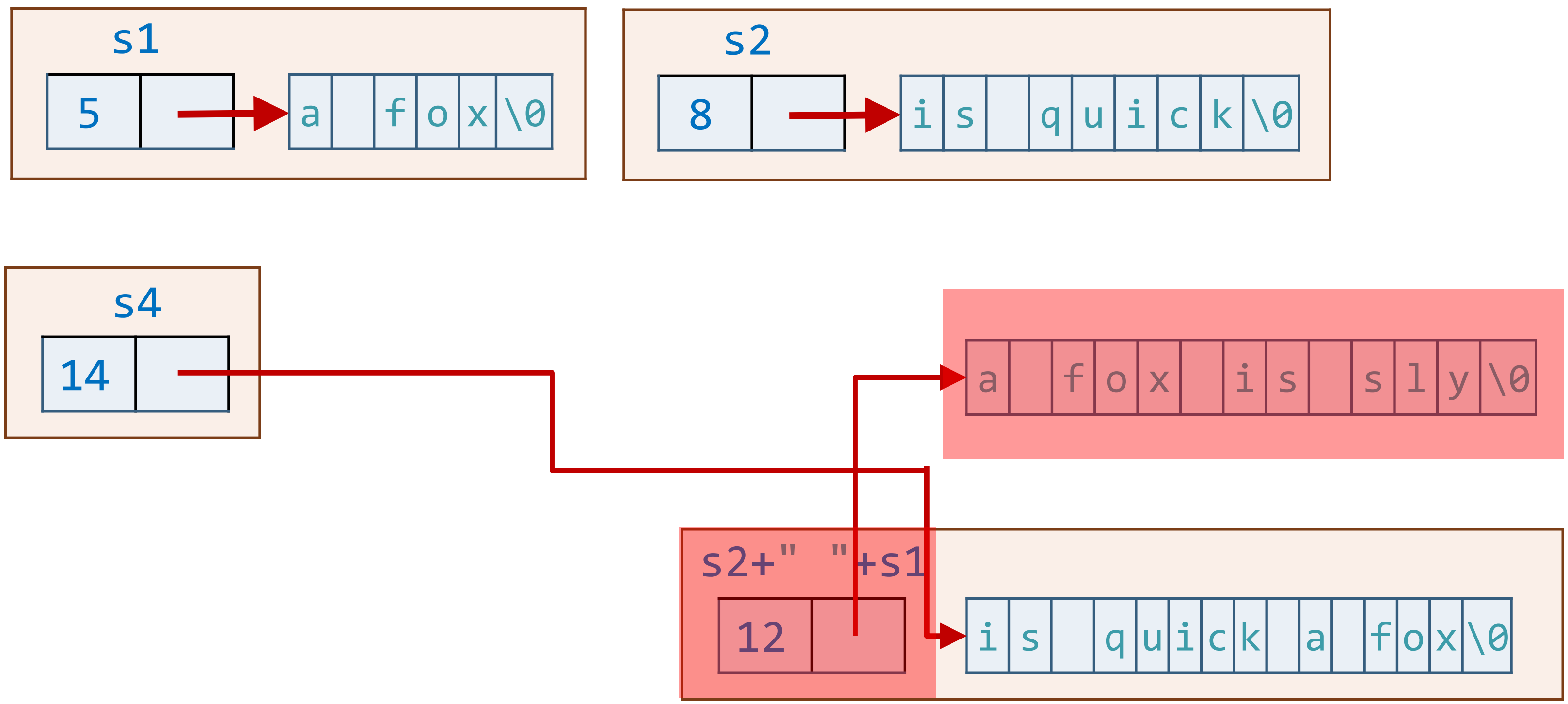
```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```



# Implementing Move Assignment (4/4)



```
Str s1{"a fox"}, s2{"is quick"};
Str s4{s1 + " is sly"};
s4 = s2+" "+s1;
```



# Special Member Functions



- By default, compiler generates each of these functions if a program uses it:
  - Copy constructor: `X(const X&)`
  - Copy assignment: `X& op=(const X&)`
  - Move constructor: `X(X&&) noexcept`
  - Move assignment: `X& op=(X&&) noexcept`
  - Destructor: `~X()`
  - Default constructor [only if other ctors are not defined]: `X()`

# Rule Of Five



- All five copy-control members [excluding default ctor] must be thought of as a unit
- If you're defining any of these special member functions for a class, you should define them all

# Synthesized Operations



| | default ctor | dtor | copy ctor | copy assignment | move ctor | move assignment |
|---|---|---|---|---|---|---|
| none defined | defaulted | defaulted | defaulted | defaulted | defaulted | defaulted |
| any ctor | not declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| default ctor | user declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| dtor | defaulted | user declared | defaulted | defaulted | not declared | not declared |
| copy ctor | not declared | defaulted | user declared | defaulted | not declared | not declared |
| copy assignment | defaulted | defaulted | defaulted | user declared | not declared | not declared |
| move ctor | not declared | defaulted | deleted | deleted | user declared | not declared |
| move assignment | defaulted | defaulted | deleted | deleted | not declared | user declared |

# Synthesized Operations: Caveats (1/2)



- Any constructor defined, default constructor is not generated
- Copy operation or a destructor defined, no move operation is synthesized *at all*
  - When class doesn't have a move operation, corresponding copy operation is used in place of move thro' normal function matching
- Move operations defined, copy operations must also be defined since they are deleted by default

# Synthesized Operations: Caveats (2/2)



- Compiler will synthesize move ctor and move assignment *only if* class doesn't define any of its own copy-control functions *and only if* all data members can be move constructed and move assigned, respectively

```
struct X {
  int i;          // built-in type can be moved
  std::string s; // string defines its own move operations
};
struct hasX {
  X mem; // X will have synthesized move ctors
};

X x, x2 = std::move(x);        // uses synthesized move ctor
hasX hx, hx2 = std::move(hx); // uses synthesized move ctor
```

# Move or Copy? (1/4)



- When both move and copy operations are defined, how does compiler know when it can use move operation rather than copy operation?
  - Rvalues are moved, lvalues are copied
  - For return value, move ctor [RVO takes precedence] or assignment are chosen

# Move or Copy? (2/4)



- When both move and copy operations are defined, how does compiler know when it can use move operation rather than copy operation?
  - Rvalues are moved, lvalues are copied
- *Rvalue* expression can be bound to an *rvalue* reference [but not to an *lvalue* reference]
- *Lvalue* expression can be bound to an *lvalue* reference [but not to an *rvalue* reference]

# Move or Copy? (3/4)

95

```
void foo(vector<int>&);          // 1
void foo(vector<int> const&); // 2
void foo(vector<int>&&);        // 3

void g(vector<int>& vi,
       vector<int> const& cvi) {
  foo(vi);                    // ???
  foo(cvi);                   // ???
  foo(vector<int>{1,2,3}); // ???
}
```

# Move or Copy? (4/4)



```
void foo(vector<int>&);          // 1
void foo(vector<int> const&);    // 2
void foo(vector<int>&&);         // 3

void g(vector<int>& vi,
       vector<int> const& cvi) {
  foo(vi);                       // call 1
  foo(cvi);                      // call 2
  foo(vector<int>{1,2,3});       // call 3
}
```

# Forcing Move Semantics (1/4)



- Sometimes, programmer knows that an object won't be used again, even though compiler does not

```
// old-style swap
template <typename T>
void swap(T& a, T& b) {
  T tmp{a}; // now there are two copies of a
  a = b;    // now there are two copies of b
  b = tmp;  // now there are two copies of tmp
}
```

# Forcing Move Semantics (2/4)



- Notice we don't want any copies at all – we just want to move values of a, b, and tmp
- swap() becomes expensive when T is type string, vector, …

```
// old-style swap
template <typename T>
void swap(T& a, T& b) {
  T tmp{a}; // two copies of a
  a = b;    // two copies of b
  b = tmp;  // two copies of tmp
}
```

# Forcing Move Semantics (3/4)



- We can tell that to compiler:

```
// almost perfect swap
template <typename T>
void swap(T& a, T& b) {
  T tmp{static_cast<T&&>(a)};
  a = static_cast<T&&>(b);
  b = static_cast<T&&>(tmp);
}
```

# Forcing Move Semantics (4/4)



- Result value of `static_cast<T&&>(x)` is an *rvalue* of type `T&&` for `x`
- Move operation optimized for *rvalues* can now use its optimization for *lvalue* `x`
  - If type `T` has move ctor or move assignment, it will be used
  - Otherwise, copy ctor or copy assignment will be used

# std::move() (1/5)



- Use of static_cast in swap() is verbose
- Standard library provides move()
  - move(x) means static_cast<X&&>(x) where x is type of X

```
// almost perfect swap
template <typename T>
void swap(T& a, T& b) {
  T tmp{std::move(a)}; // steal from a
  a = std::move(b);      // steal from b
  b = std::move(tmp);  // steal from tmp
}
```

# `std::move()` (2/5)



- `std::move()` is standard library function returning an *rvalue* reference to its argument `x`
  - `move(x)` means "give me *rvalue* reference to `x`"
  - `move(x)` *doesn't move anything* – instead, it provide *rvalue* reference to `x` that allows user to take `x`'s resources

# `std::move()` (3/5)



- That is, `move()` is used to tell compiler that an object will not be used anymore in certain scope, so that its value can be moved and an empty object is left behind
  - Use `move()` when intent is to "steal representation" of an object with a move operation
  - Only safe use of `x` after a `move(x)` is destruction or as target for assignment

# `std::move()` (4/5)



- Using `std::move()` wherever we can, provides following benefits:
  - Significant performance gains when using standard algorithms and operations
  - STL requires *copyability* of types used as container values – in most cases *moveability* is enough
  - We can define types that are *moveable* but not *copyable* such as `std::unique_pointer`

# `std::move()` (5/5)



- What happens if we try to swap objects of type that doesn't have move functions?
  - We copy and pay the price
- But, in this case, how does compiler evaluate `move(x)` to call to a copy function?
  - Doesn't `move(x)` mean `static_cast<T&&>(x)`?

# Overloading Rules for *Rvalue* and *Lvalue* References (1/8)



- We can declare following overloads

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
void foo(X&&);        // 4) in & moved-from parameters
```

# Overloading Rules for *Rvalue* and *Lvalue* References (2/8)



- If you only implement *pass-by-value*

```
void foo(X);            // 1) in parameters: inexpensive
void foo(X&);           // 2) in-out parameters
void foo(X const&);     // 3) in parameters: expensive
void foo(X&&);          // 4) in & moved-from parameters
```

- `foo` can be [possibly expensively] called for *lvalue*, `const` *lvalue*, *rvalue*, and `const` *rvalue* parameters since argument is copied
  - Pass-by-value is used for *inexpensive values*

# Overloading Rules for *Rvalue* and *Lvalue* References (3/8)



- If you only implement *pass-by-lvalue-reference*

```
void foo(X);            // 1) in parameters: inexpensive
void foo(X&);           // 2) in-out parameters
void foo(X const&);     // 3) in parameters: expensive
void foo(X&&);          // 4) in & moved-from parameters
```

- `foo` can be *inexpensively* called for *lvalues* but not for `const` *lvalues*, *rvalues*, and `const` *rvalues*

# Overloading Rules for *Rvalue* and *Lvalue* References (4/8)



- If you only implement

```
void foo(X);            // 1) in parameters: inexpensive
void foo(X&);           // 2) in-out parameters
void foo(X const&);     // 3) in parameters: expensive
void foo(X&&);          // 4) in & moved-from parameters
```

- foo can be called for everything *inexpensively*: *lvalues* , const *lvalues*, *rvalues*, and const *rvalues*
- However, not possible to distinguish between *lvalues* and *rvalues*

# Overloading Rules for *Rvalue* and *Lvalue* References (5/8)



- If you only implement

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&);   // 3) in parameters: expensive
void foo(X&&);        // 4) in & moved-from parameters
```

- `foo` can only be called on *rvalues* but not on *lvalues*, `const` *lvalues*, and `const` *rvalues*

# Overloading Rules for *Rvalue* and *Lvalue* References (6/8)



- If you implement these two

```
void foo(X);            // 1) in parameters: inexpensive
void foo(X&);           // 2) in-out parameters
void foo(X const&);     // 3) in parameters: expensive
void foo(X&&);          // 4) in & moved-from parameters
```

- You can distinguish between *lvalues* AND `const` *lvalues*, *rvalues*, and `const` *rvalues*

# Overloading Rules for *Rvalue* and *Lvalue* References (7/8)



- If you implement these two

```
void foo(X);            // 1) in parameters: inexpensive
void foo(X&);           // 2) in-out parameters
void foo(X const&);     // 3) in parameters: expensive
void foo(X&&);          // 4) in & moved-from parameters
```

- You can distinguish between `const` *lvalues*, *rvalues*, and `const` *rvalues* AND *rvalues*

# Overloading Rules for *Rvalue* and *Lvalue* References (8/8)



- All of this means if class doesn’t provide move semantics and has only copy ctor and copy assignment operator, these will be called for *rvalue* references
- Thus, `std::move()` means *to call move semantics, if provided, and copy semantics otherwise*

# Reference Arguments: Rules of Thumb (1/2)



- How do we choose among the many ways of passing arguments?

# Reference Arguments: Rules of Thumb (2/2)



- Use pass-by-value for small objects (<= 16B)
- Use pass-by-`const` references to pass large values that you don't need to modify [*in* parameters]
- Return result as `return` value rather than modifying an object thro' an argument [return value optimization or move assignment]
- Use *rvalue* references to implement *move* and *forwarding*
- Use pass-by-reference only if you have to
- Pass a pointer if "no object" is valid alternative