RVALUE REFERENCES & MOVE SEMANTICS

Plan for Today

- Rvalue References
- Move Semantics
  - Move Constructor
  - Move Assignment Operator
  - std::move

Quiz on References


```cpp
void mutate(std::string& s);
void observe(std::string const& s);
std::string three();
std::string const four();

std::string       one{"one"};
std::string const two{"two"};

// which of these calls are valid?
observe(one);     observe(two);
observe(three()); observe(four());

mutate(one);     mutate(two);
mutate(three()); mutate(four());
```

Lvalues and *Rvalues*

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

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

Lvalues and *Rvalues*

- 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;
b = std::abs(a*c);
```

Lvalues and *Rvalues*

- 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

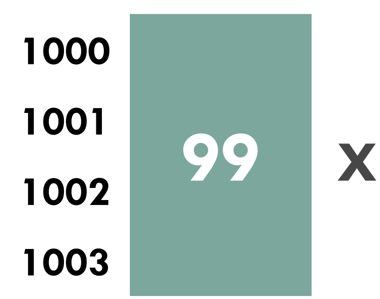
```
int x = 99;
```

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

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

here, x means *lvalue*

while here, x means *rvalue*



Lvalues and *Rvalues*

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

Lvalues and *Rvalues*

- Useful to understand behavior of operators
- Some operators require *lvalue* operands while others require *rvalue* operands
- Some operators return *lvalues* while others return *rvalues*

Lvalues and *Rvalues*

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

```
int gi{10};

int incr(int x) { return x+1; }
int* foo()      { return &gi; }

int *pi = foo(); // ok
*pi = incr(*pi); // ok
incr(*pi) = 10;  // error
*foo() = 111;    // ok
++*foo();        // ok
```

Lvalues and *Rvalues*: C++

- 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{"hello"};
Str const c{"constant"};
s[0] = A' + (c[0] - 'a'); // ok
c[0] = 'C';   // error
(s+t)[0] = 'C';  // ok but weird
```

Lvalues and *Rvalues*: C++

- Ordinary functions can now return references

```
int gi{10};

int& boo() {
  return gi;
}

int *pi = &boo();
boo() = 100; // ok
++boo(); // ok
int x = boo(); // ok
```

Lvalues and *Rvalues*: C++

- C++ standard: *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
  - They are either literals or temporary objects created in the course of evaluating expressions

Lvalues and *Rvalues*: C++

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

Lvalues and *Rvalues*: C++


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

Lvalues and *Rvalues*: C++

- 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

Lvalues and *Rvalues*: C++

- Either modifiable (non`const`) or non-modifiable (`const`):

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

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

References

- In a type name, notation X& means "reference to X"
  - Binds to modifiable lvalues

```
int   x {2};
int& rx {x};     // x and rx refer to same int
rx = 4;          // x now becomes 4
int y {rx};      // y initialized to 4
int *pi {&rx}; // points to x
```

References

- 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 modifiable rvalues – modifying temporaries that evaporate along with modifications can lead to bugs
  - Can't bind to `const` rvalues - for above reasons
- Called "ordinary" reference in C++98 and now called *lvalue reference*

References

- Obvious implementation: constant pointer (to object to which it refers) that is dereferenced each time it is used
- Main purpose: to refer to objects that we wish to change (so called "in/out" parameters)

References

- What do we know about lvalue references?
  - Must be initialized – no such thing as a null reference
  - No operator operates on a reference
  - Value of a reference cannot be changed after initialization
  - Cannot get a pointer to a reference
  - Cannot define an array of references
  - Basically, a reference is not an object

References


```
int i{1}, i2{2048}; // i1 and i2 are ints
int &r{i}, r2{i2};  // r is reference bound to i; r2 is an int
int i3{1024}, &ri{i3}; // i3 is an int;
                       //  ri is reference bound to int
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};
int &r6{dval};   // error: initializer must be int  lvalue
const int ci{1024};
int &r8{ci};      // error: initializer must be non-const  lvalue
```

Why Can't Rvalue Bind to Reference?

- Causes unnecessary bugs

```
void incr(int& ri) { ++ri; }
void foo() {
 double d = 10.1;
 incr(d);
}
```

- If binding of rvalues to non`const` references is allowed, then `d` will not be incremented
- Instead, temporary `int` that must be created to pass to `incr()` will be the one incremented

`const` References (1/5)

- In a type name, notation `X const&` means "non-modifiable (`const`) reference to `X`"
  - Binds to modifiable lvalues
  - Binds to `const` lvalues
  - Binds to modifiable rvalues
  - Binds to `const` rvalues

`const` References (2/5)


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

`const` References (3/5)

- 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` References (4/5)

- Consider:

```
double& dr = 1; // error – lvalue needed
double const& cdr { 1 }; // ok
```

- Interpretation of this last initialization:

```
double const temp {(double) 1};
double const& cdr { temp };
```

- First, create a temporary with right value
- Then, use temporary as initializer for `const` reference

`const` References (5/5)

- Main purpose of `const` references: to refer to objects whose values we don’t want to change or shouldn’t change (so called in parameters)

Reference: Lvalue or Rvalue? (1/2)

- Reference is a name
- Can we speak about lvalueness of this name?
- Yep!!!
- Reference bound to lvalue or rvalue is itself an lvalue
- And, reference bound to `const` lvalue or `const` rvalue is `const` lvalue reference
- Remember: "*lvalueness versus rvalueness is a property of expressions, not of objects*"

Reference: Lvalue or Rvalue? (2/2)

- Have you ever bound an rvalue to a `const` reference and then taken its address?
- Any examples?

Reference to Rvalue?

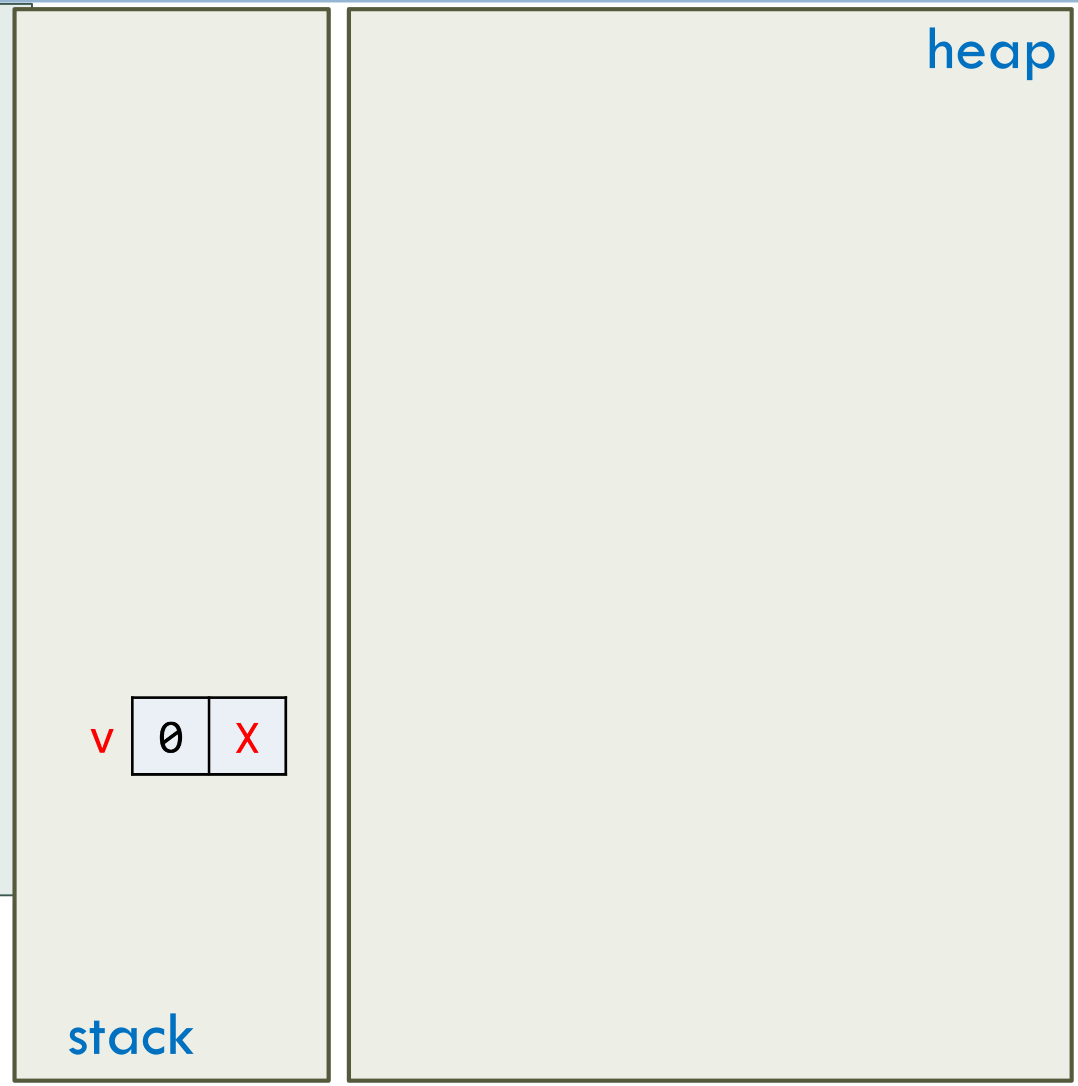
- What can we do with modifiable rvalues?
  - Can't bind reference (X&) to modifiable rvalues
  - Can't assign things to rvalues
- Can they be really modified?
  - Definitely not in C
  - Maybe in C++
- Calling a non`const` member function on a modifiable rvalue is allowed in C++98
- What about C++11?

Quiz on References


```
void mutate(std::string& s);
void observe(std::string const& s);
std::string three() { return "three"; }
std::string const four() { return "four"; }

std::string one("one");
std::string const two("two");

// which of these calls are valid?
observe(one);        observe(two);
observe(three());  observe(four());

mutate(one);        mutate(two);
mutate(three()); mutate(four());
```

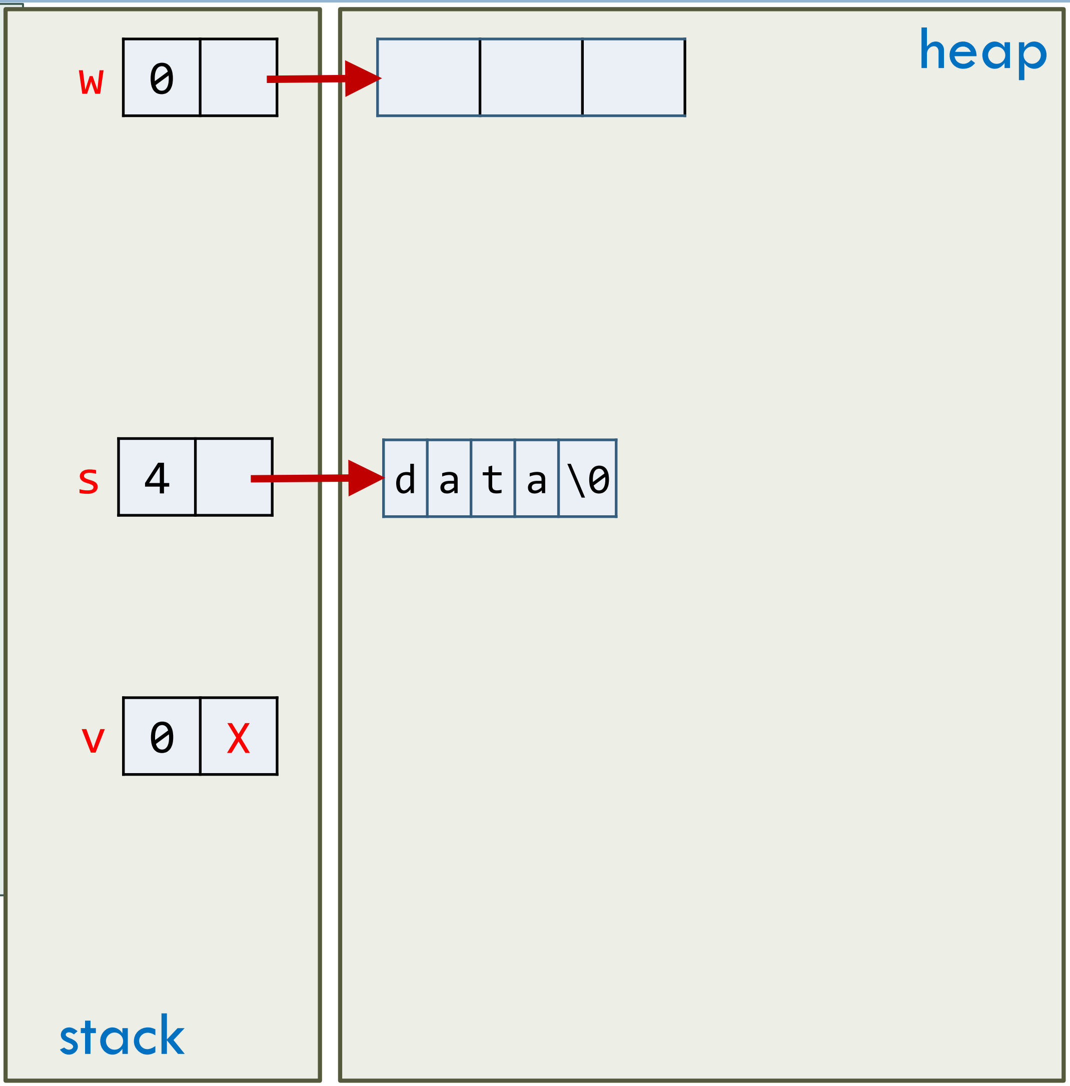
Motivation for Move Semantics


```
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();
```



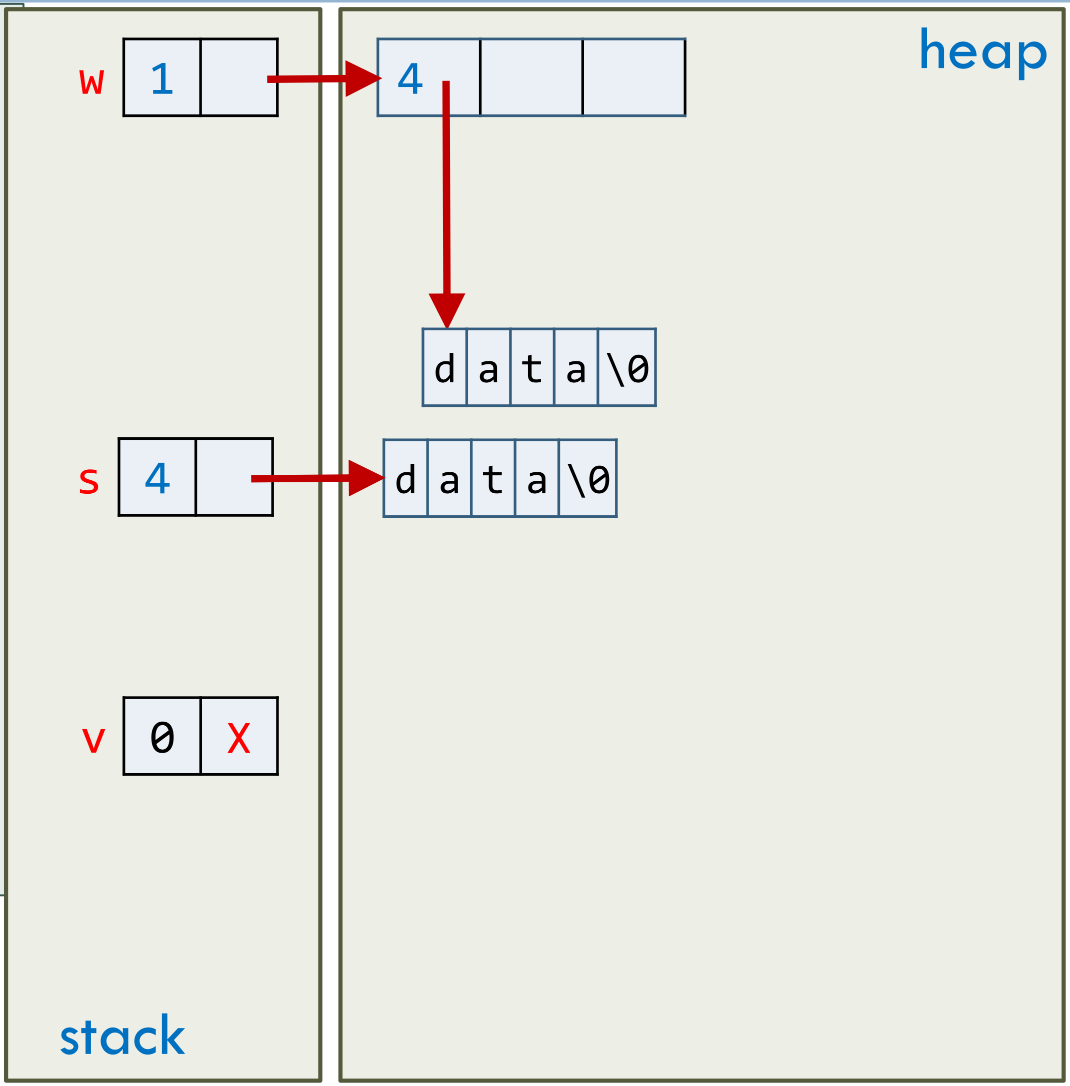
Motivation for Move Semantics


```
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();
```



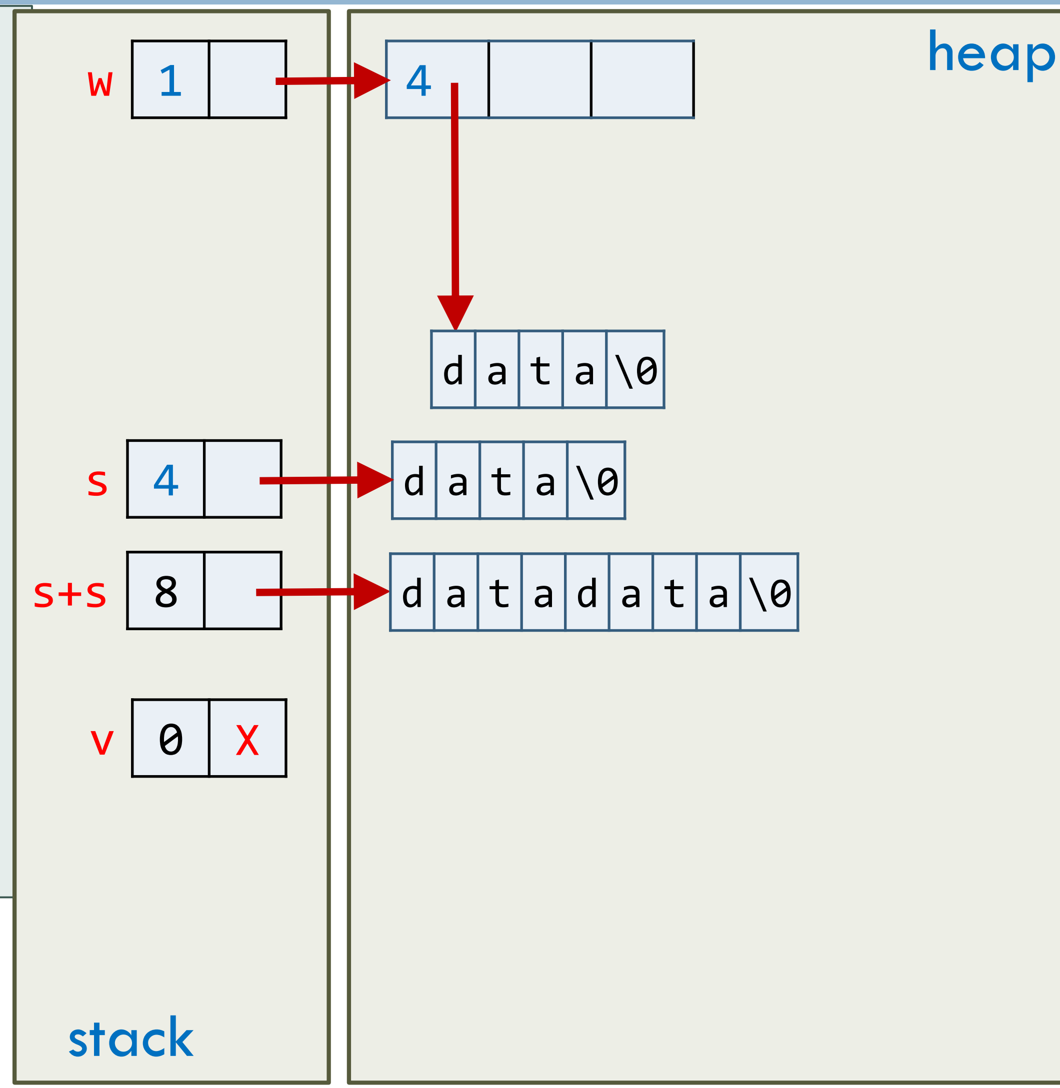
Motivation for Move Semantics


```
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();
```



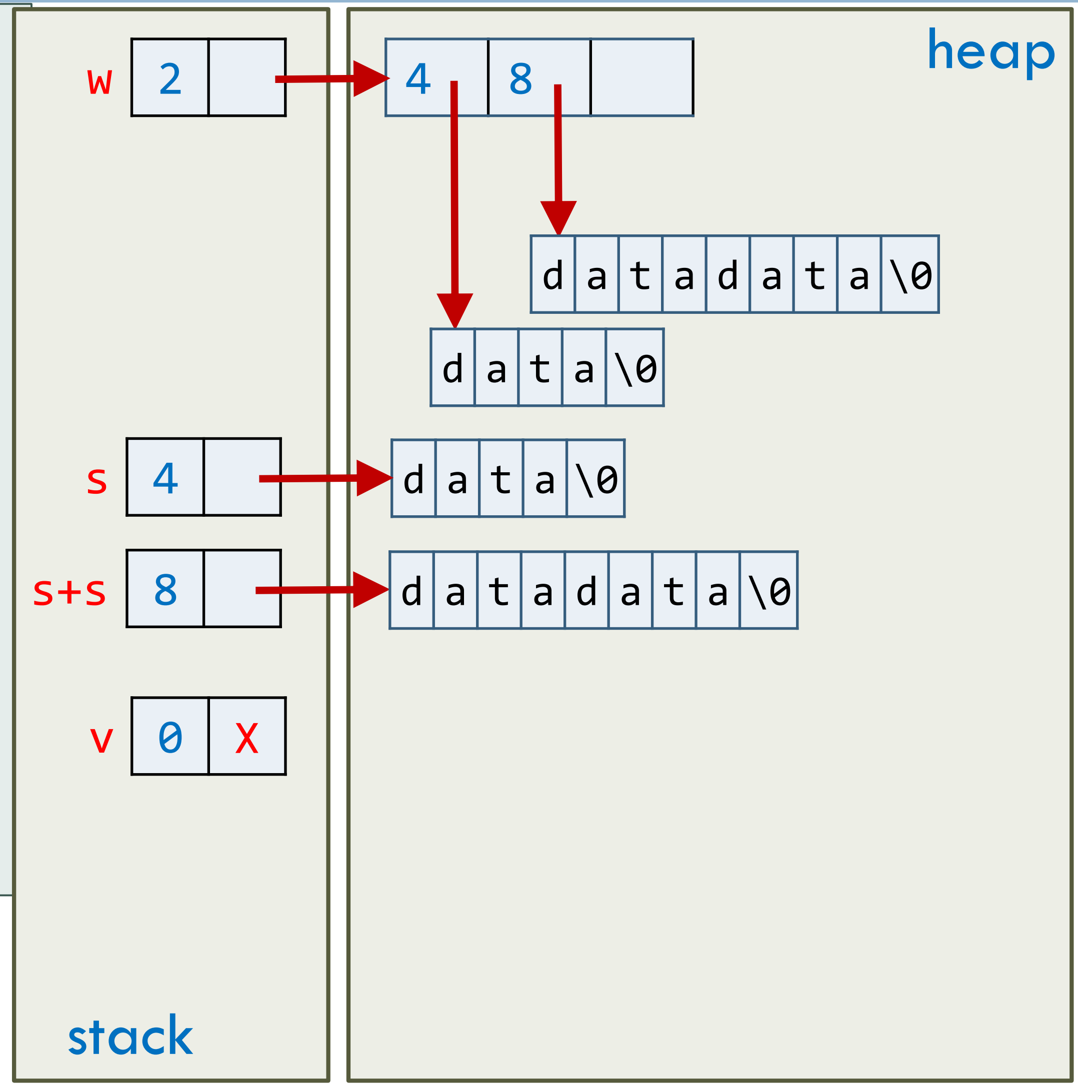
Motivation for Move Semantics


```
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();
```



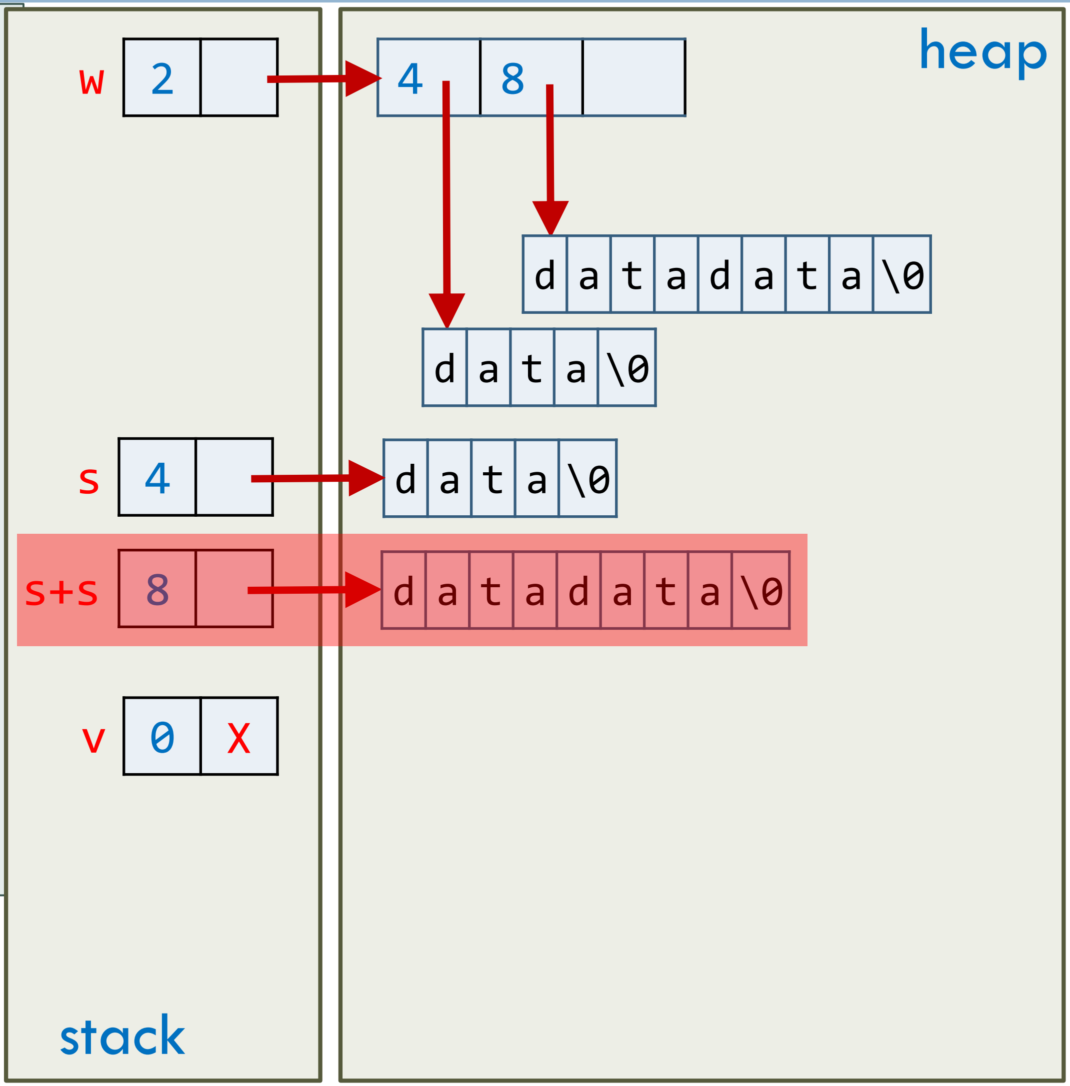
Motivation for Move Semantics


```
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();
```



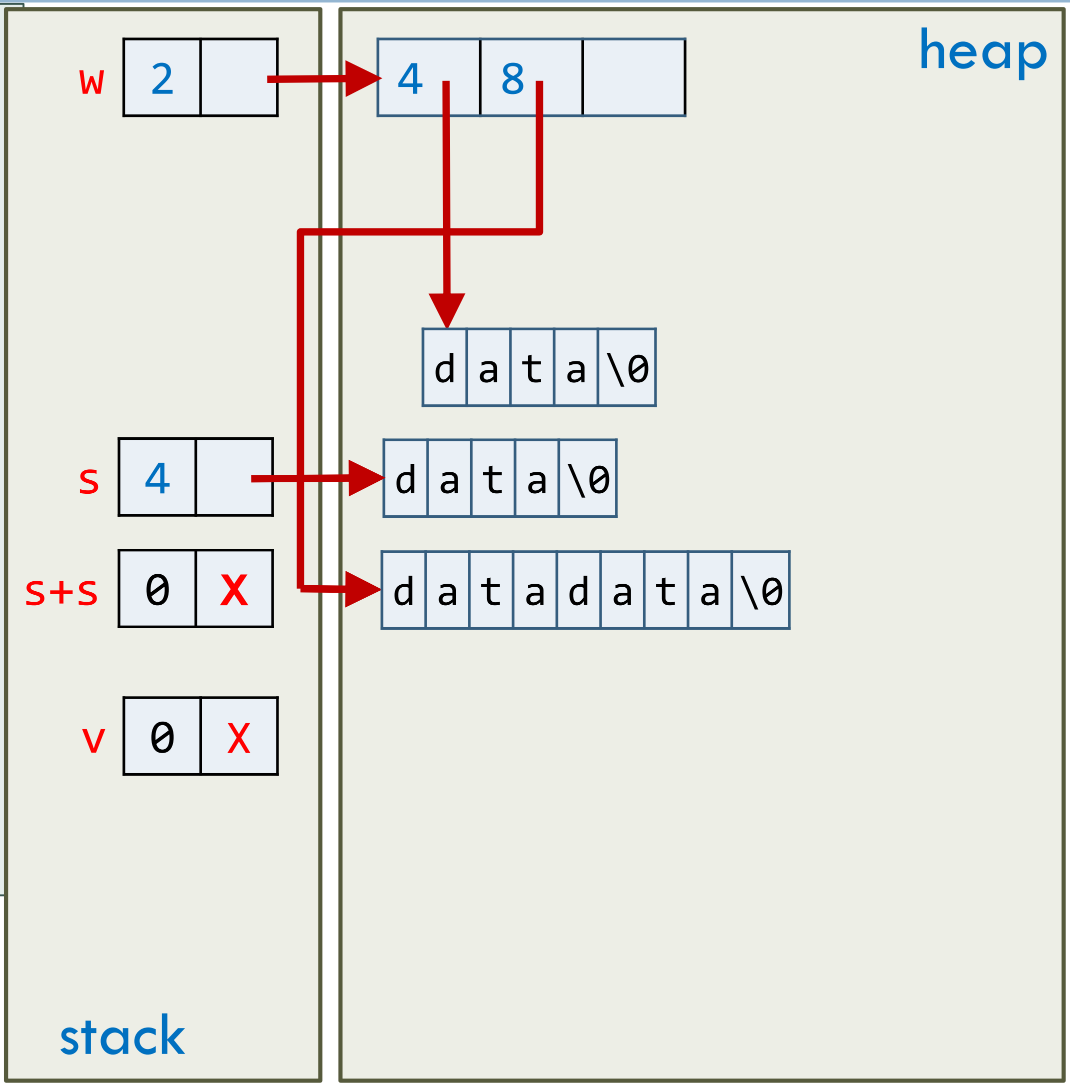
Motivation for Move Semantics


```
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();
```



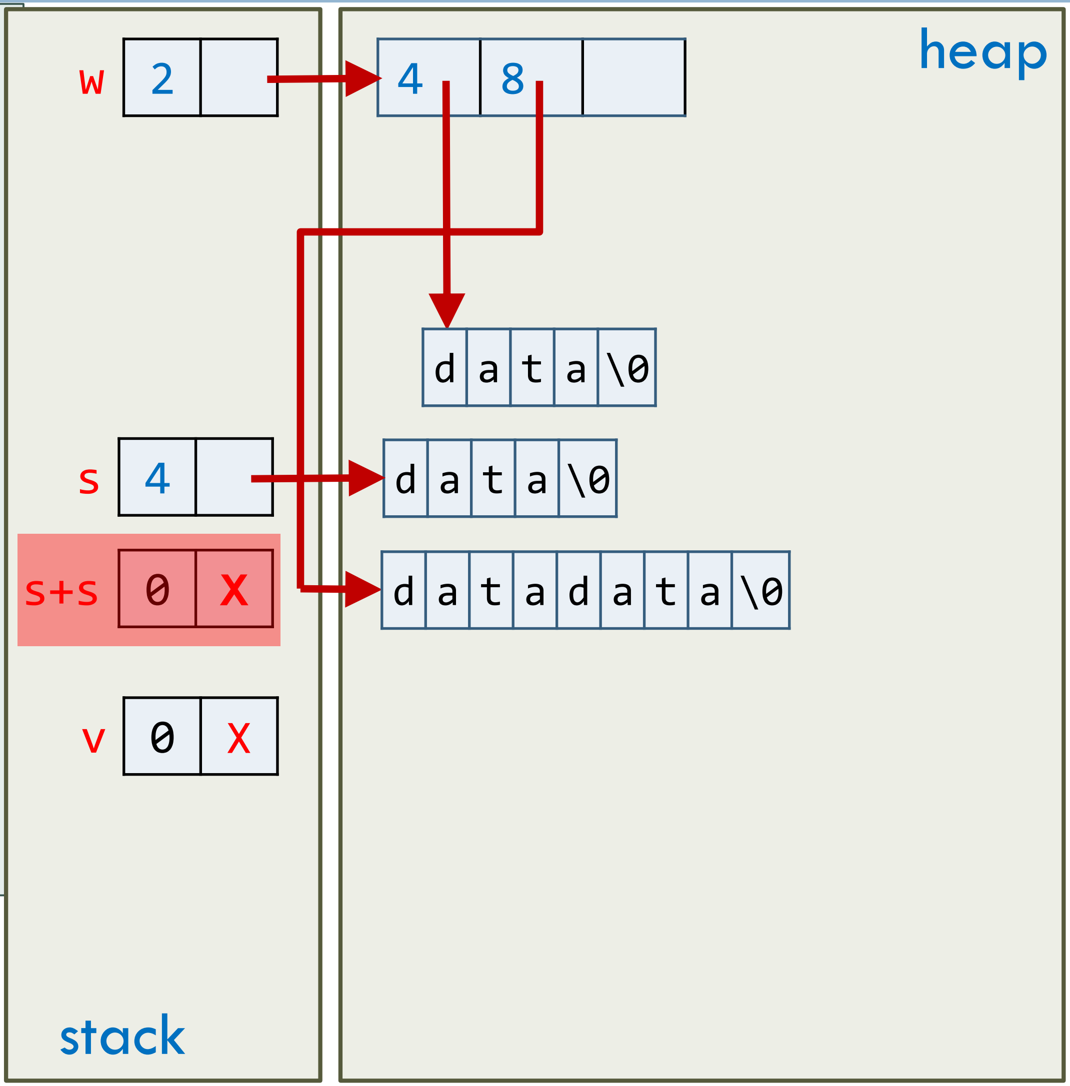
Motivation for Move Semantics


```
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();
```



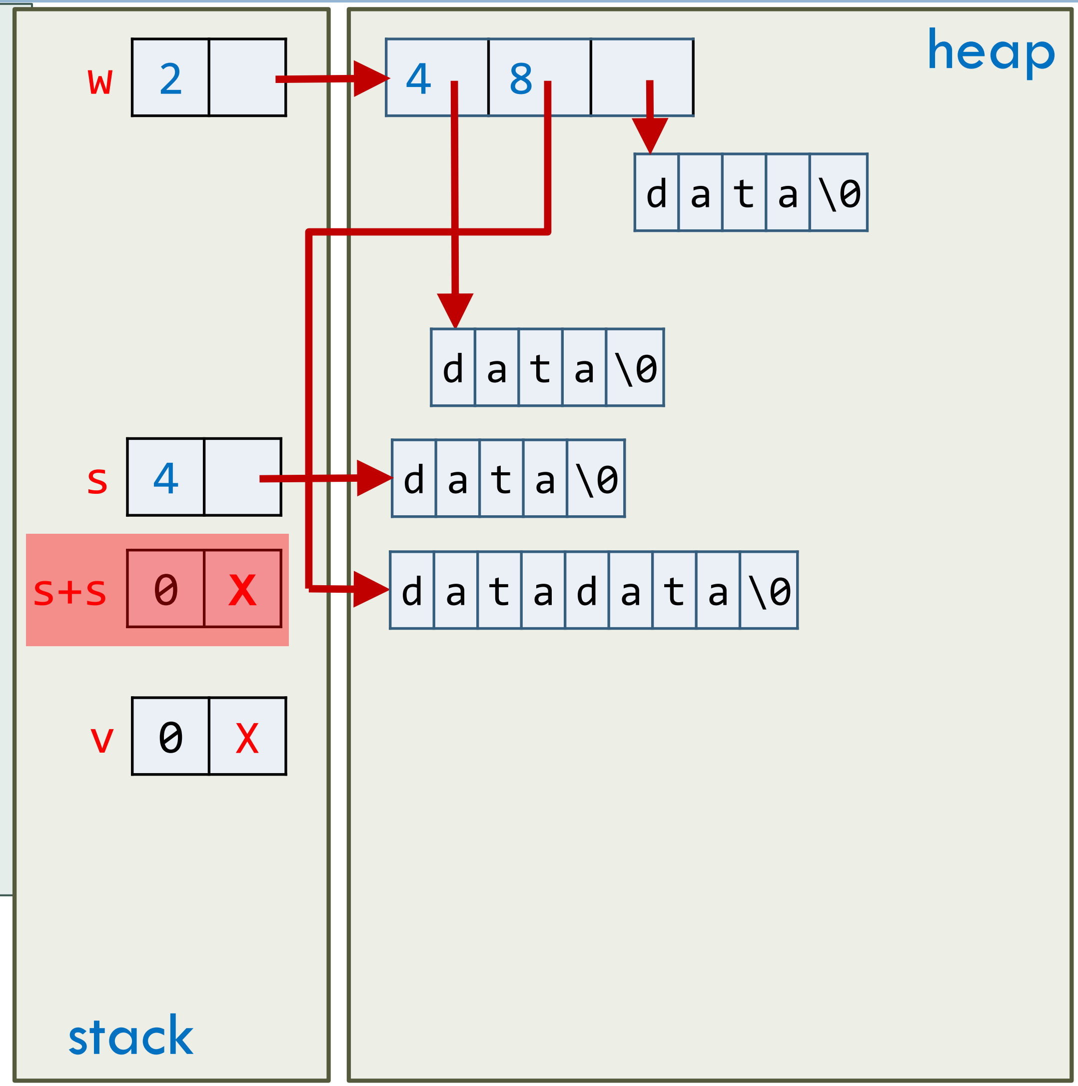
Motivation for Move Semantics


```
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();
```



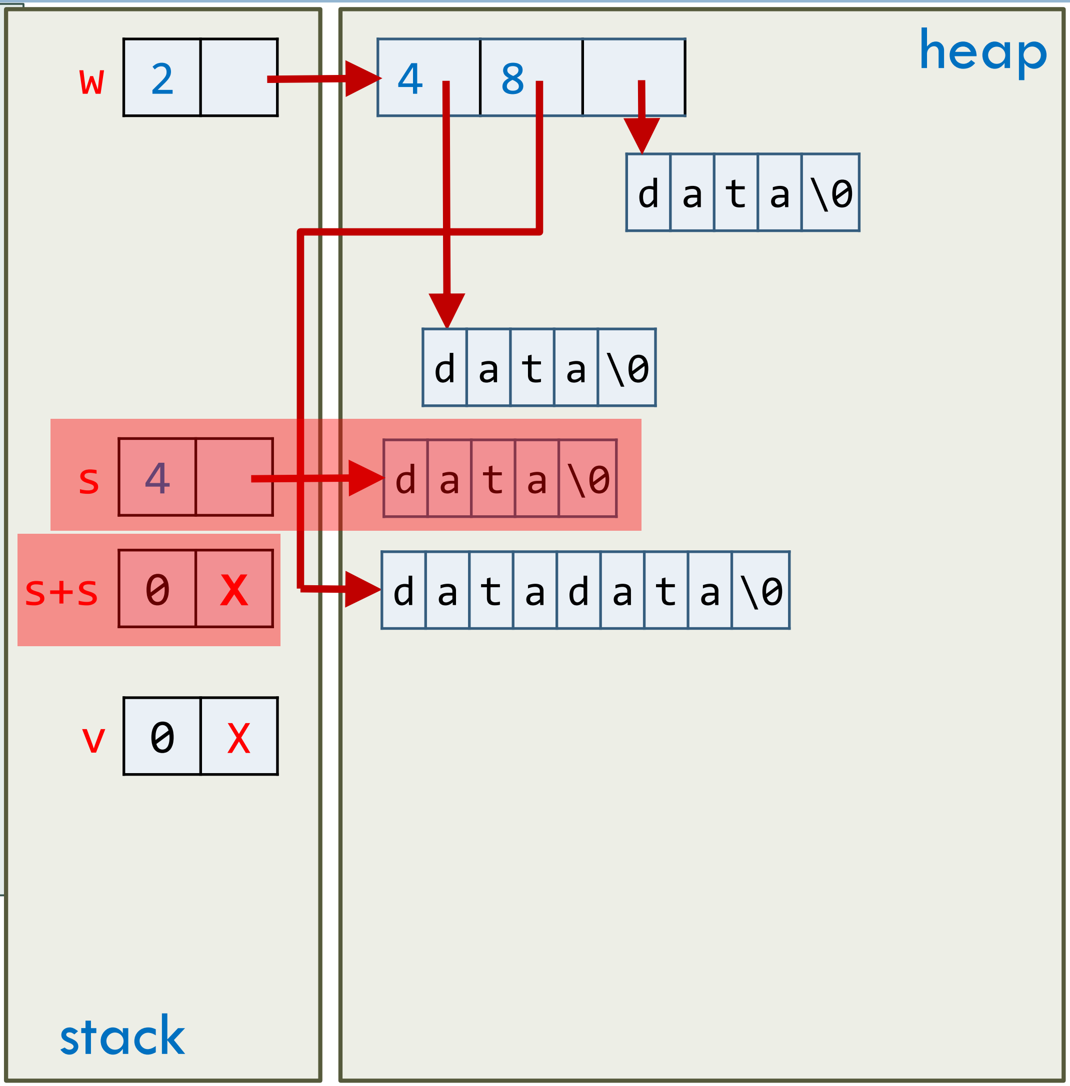
Motivation for Move Semantics


```
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();
```



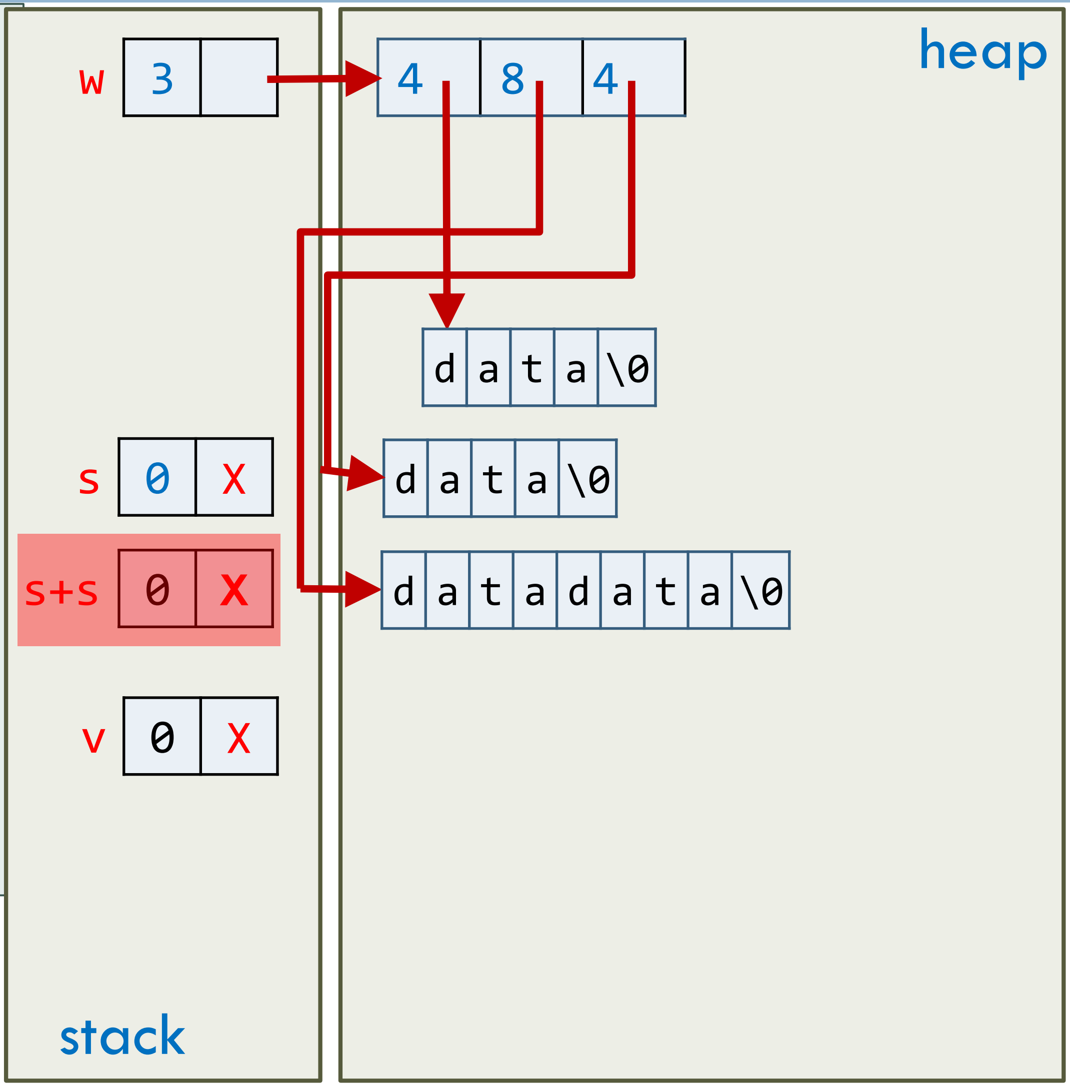
Motivation for Move Semantics


```
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();
```



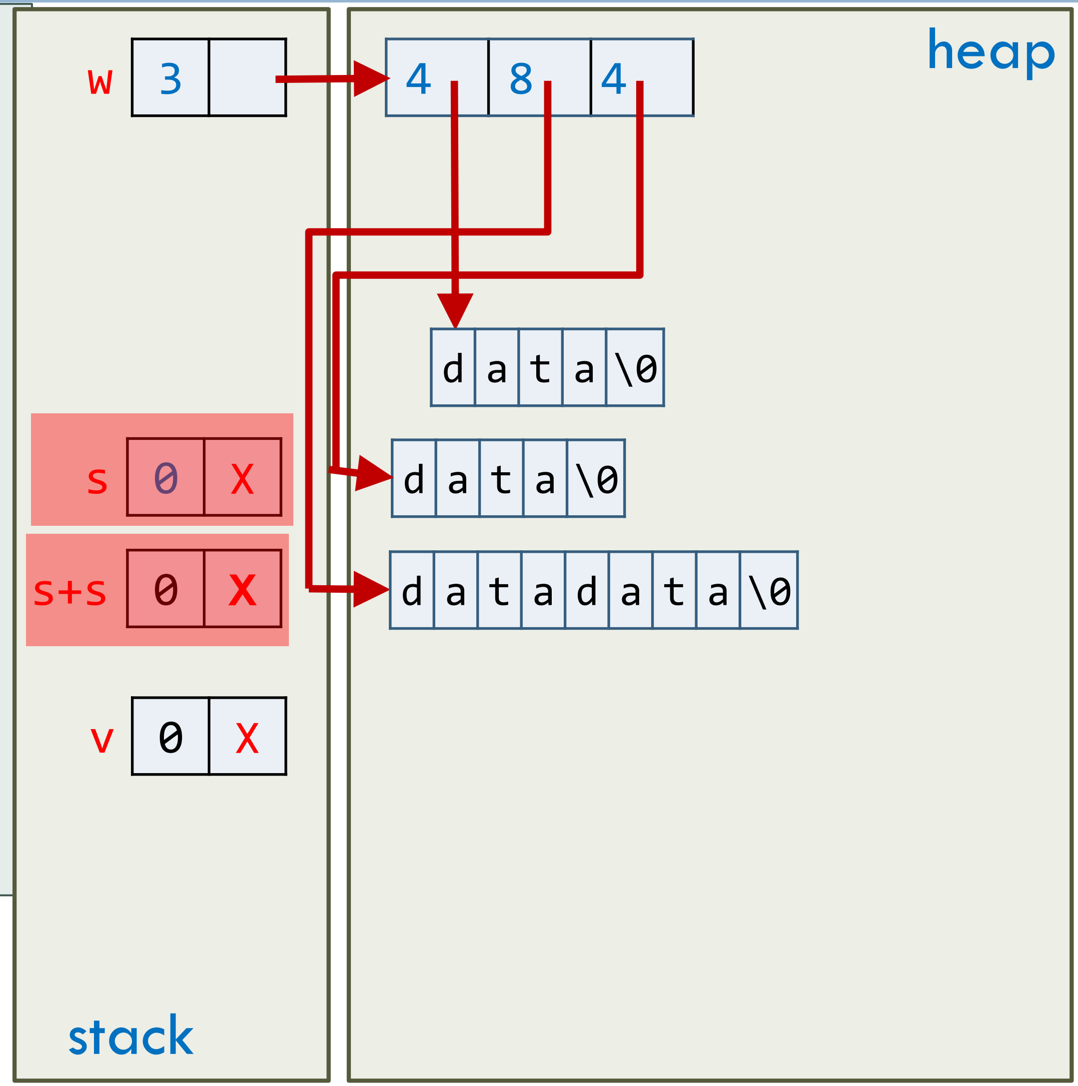
Motivation for Move Semantics


```
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();
```



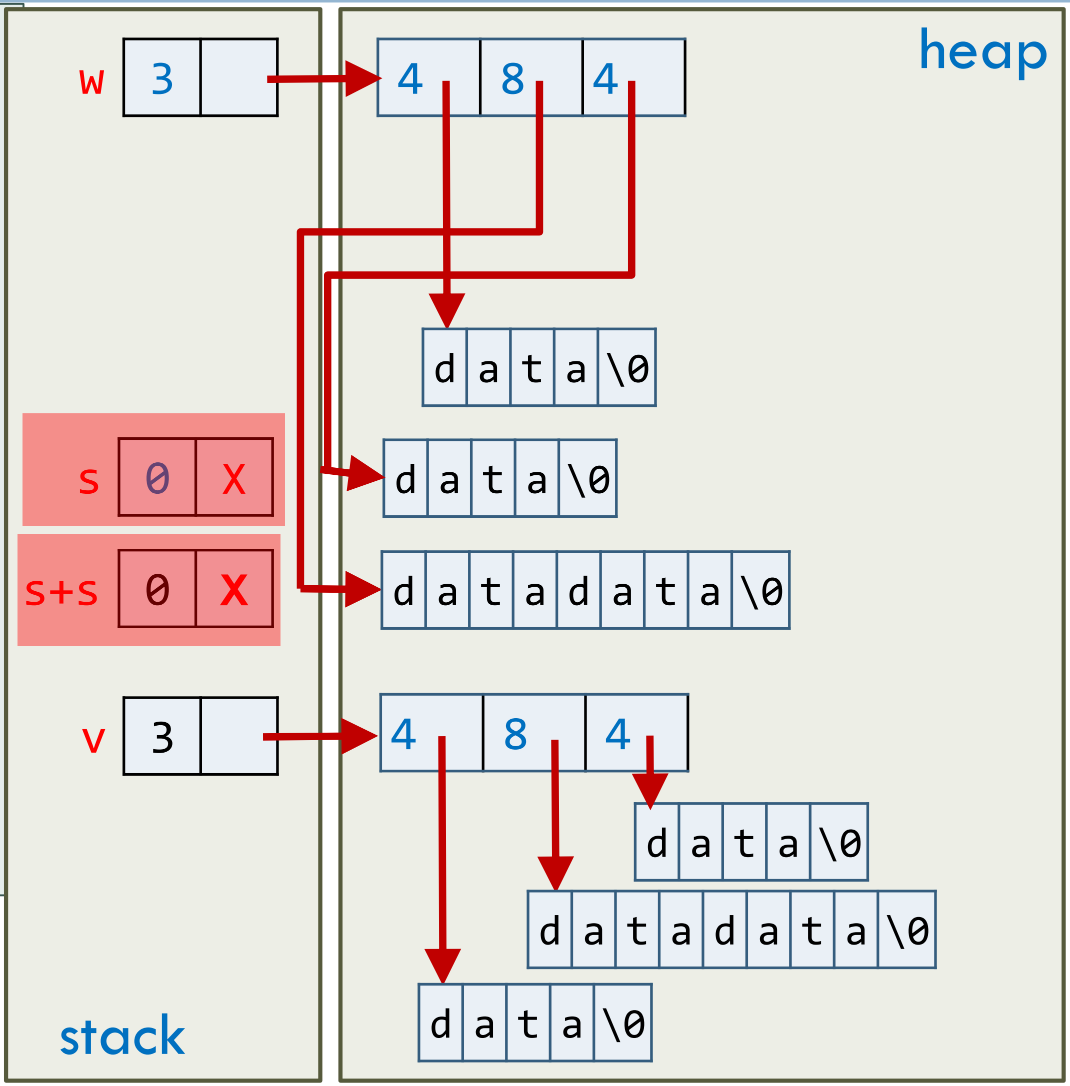
Motivation for Move Semantics


```
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();
```



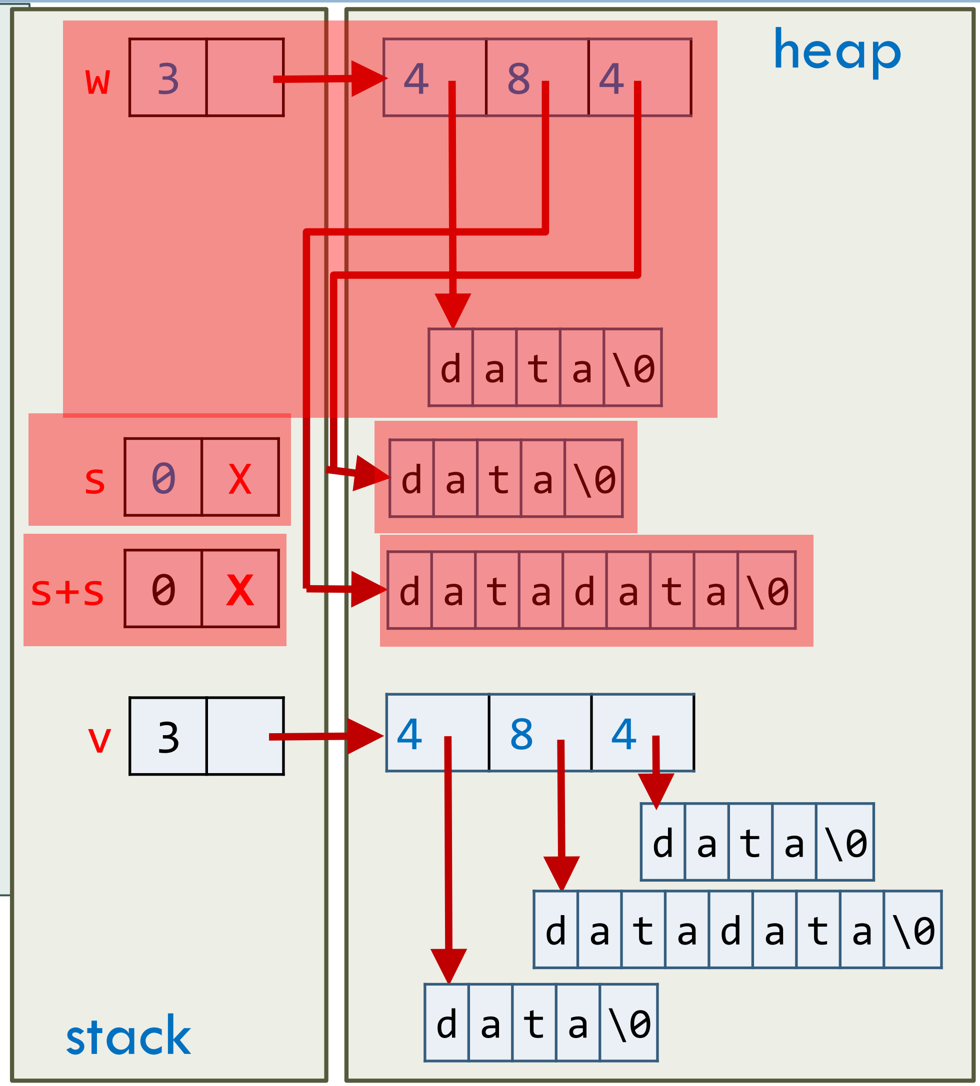
Motivation for Move Semantics


```
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();
```



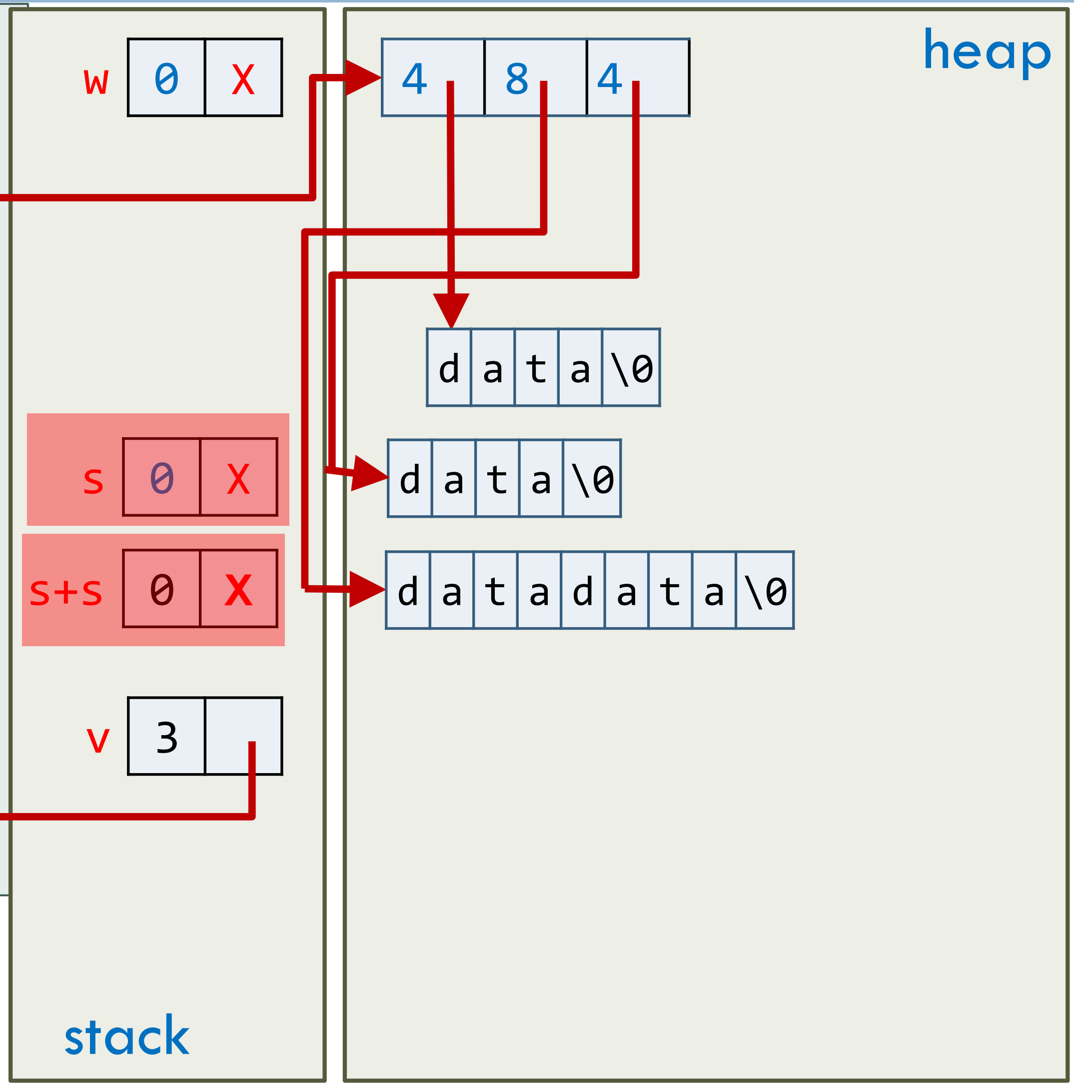
Motivation for Move Semantics


```
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();
```



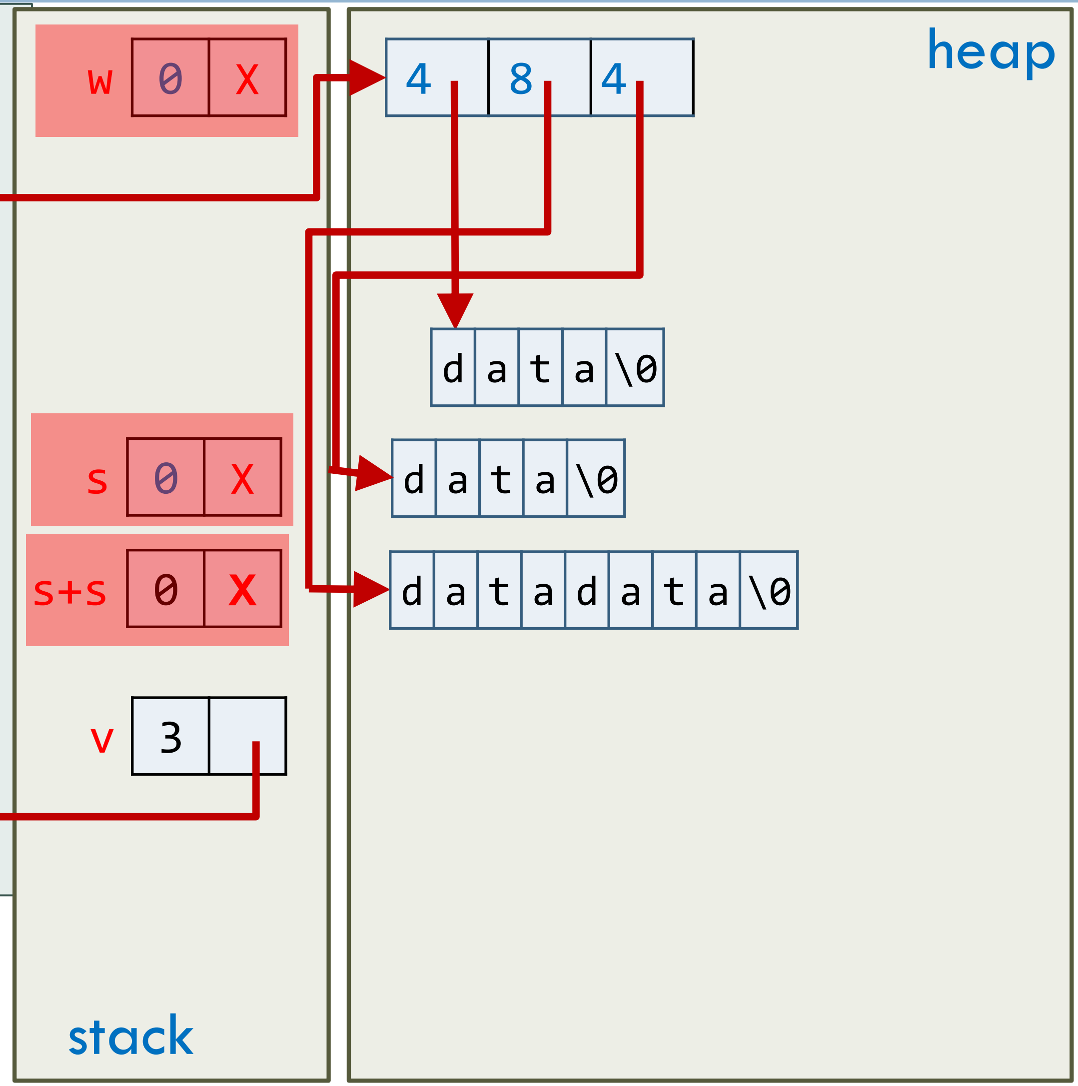
Motivation for Move Semantics


```
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();
```

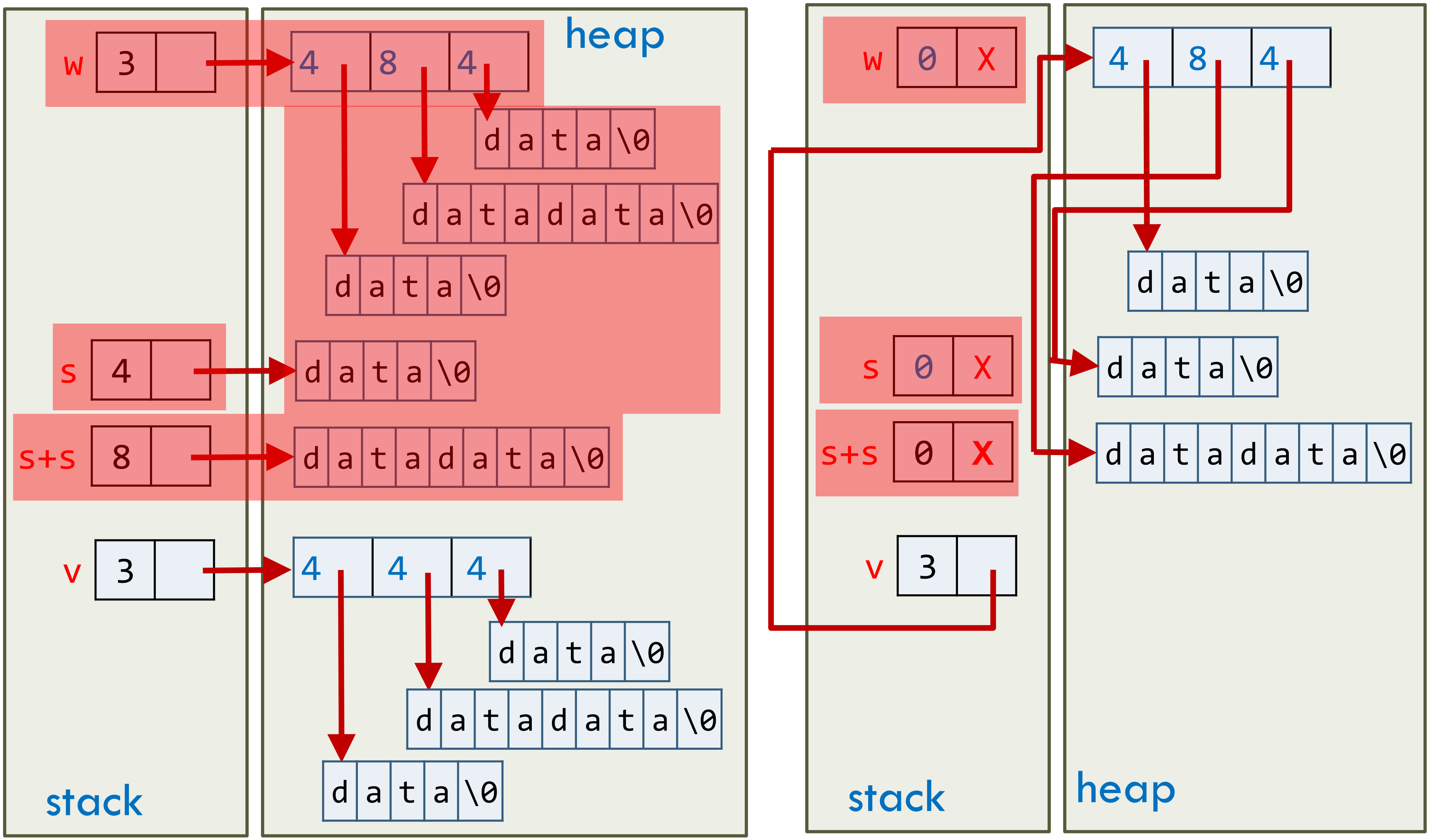


Motivation for Move Semantics


```
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



Motivation for Move Semantics

- At the end, we're in same state as without using move semantics, but with something significant gained:
  - Assuming RVO, avoided five unnecessary copies
  - Able to now use vector of strings naively like built-in type
  - Returning vector of strings and assigning it to an existing string is no longer performance issue

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?

- Copy assignment operator for class `Str` (a typical handle class) looks like this:

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

What C++ Can Do Now?

- Similar reasoning applies to copy ctor

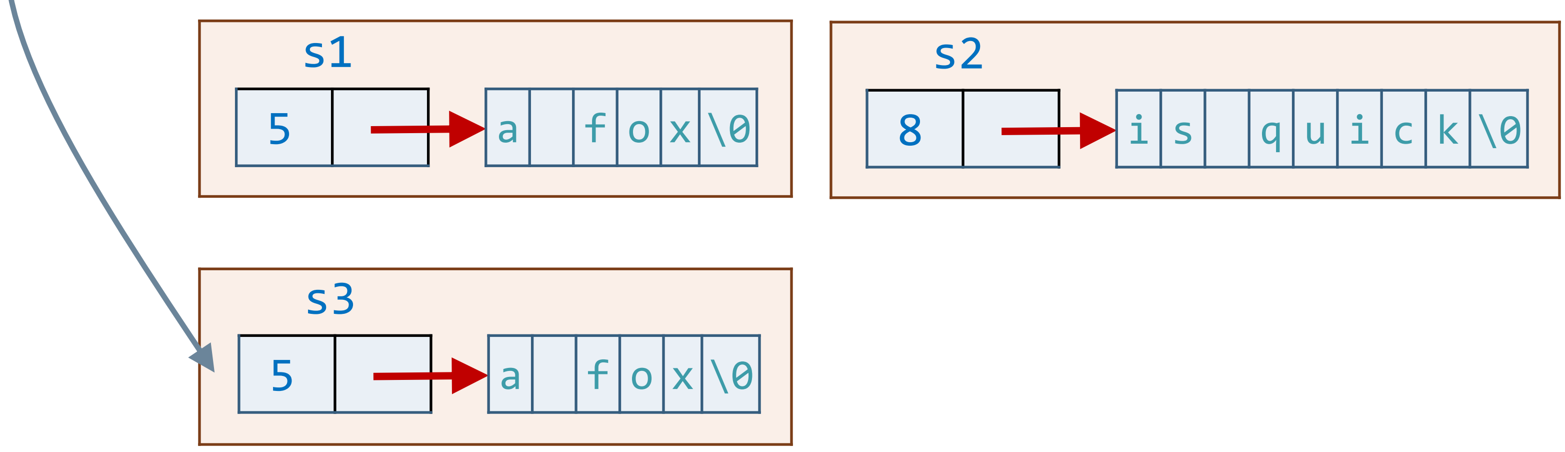
```
Str:: Str(Str const& rhs) {
  // ...
  // make clone of what rhs.ptr refers to
  // attach clone to ptr
  // ...
}
```

What C++ Can Do Now?

- When we do this

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

- We want

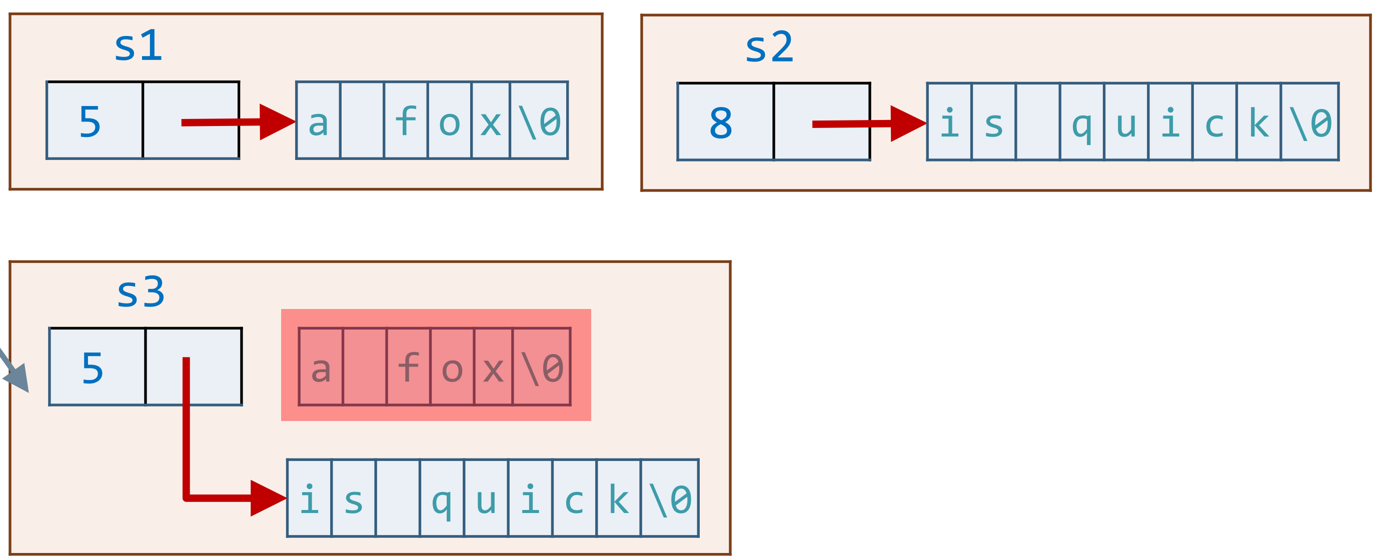


What C++ Can Do Now?

- When we do this

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

- We want



What Can C++ Do Now?

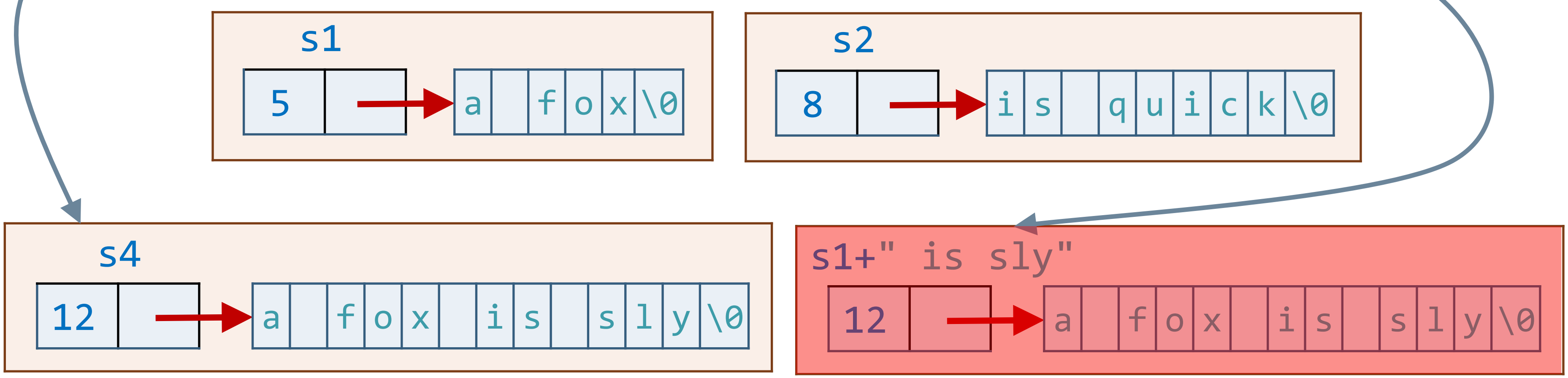

```
Str operator+(Str const& lhs, Str const& rhs) {
  Str tmp{lhs};
  tmp += rhs;
  return tmp;
}

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

What We Want From C++?

- And when we do this

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

- We don’t want

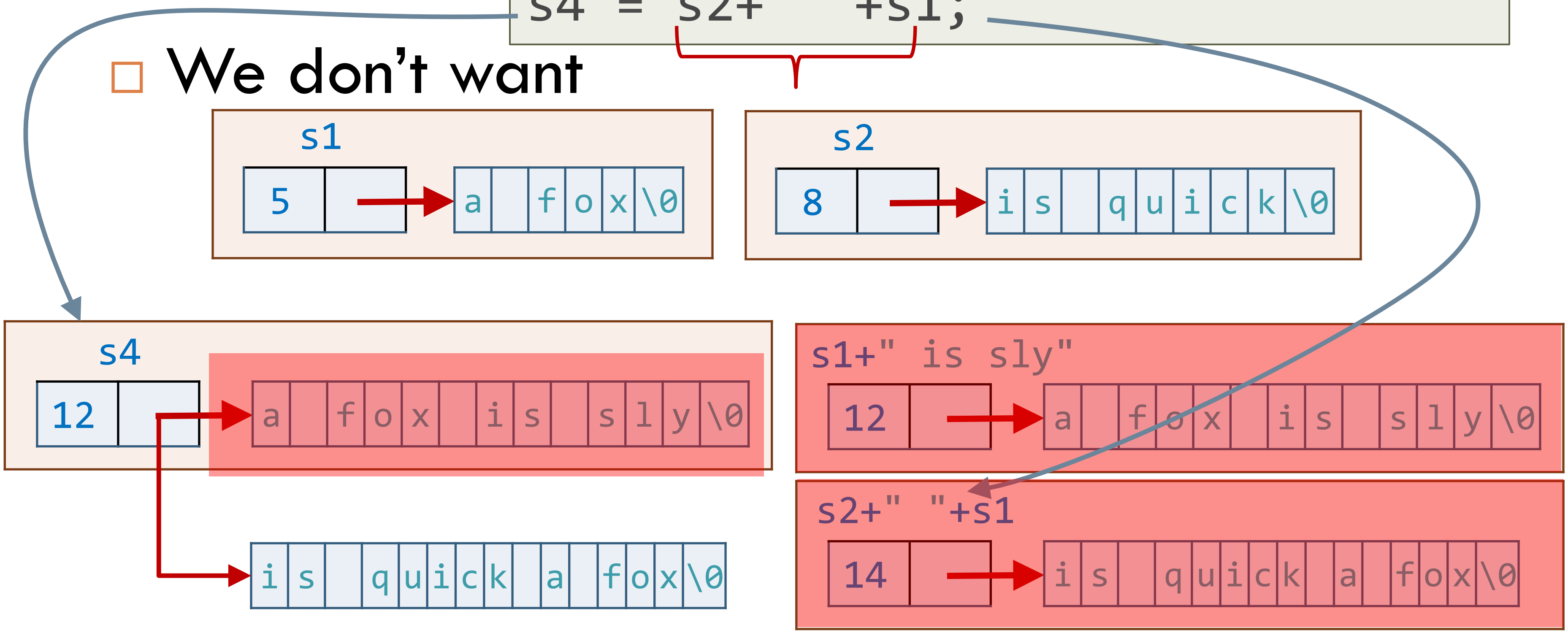


What We Want From C++?

- And when we do this

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

- We don't want

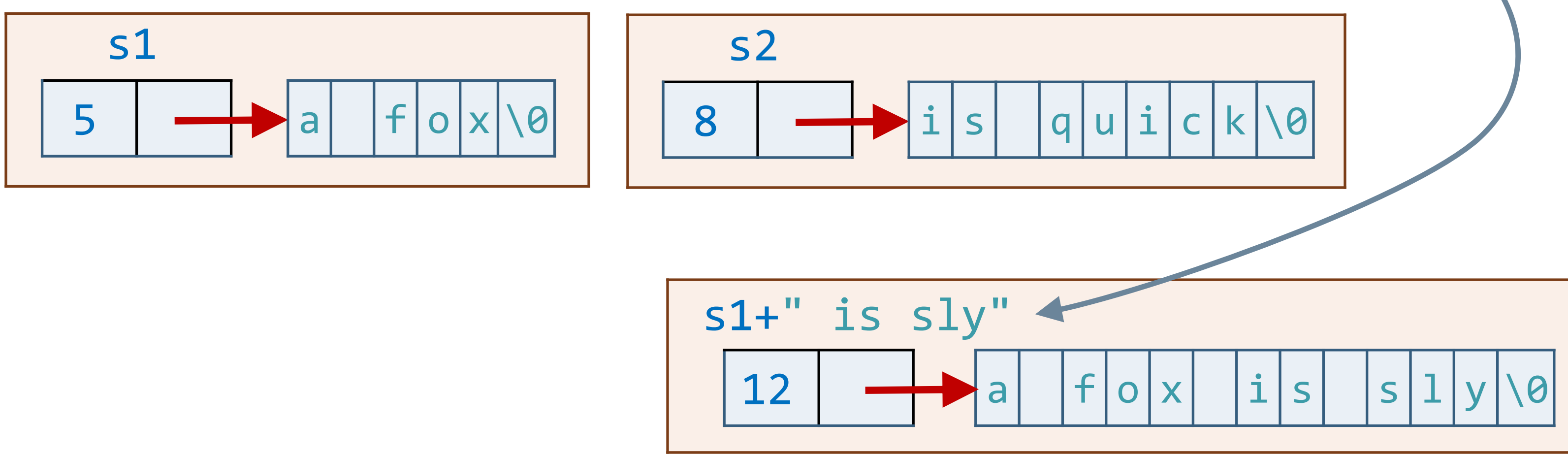


What We Want From C++?

- Instead,

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

- We want

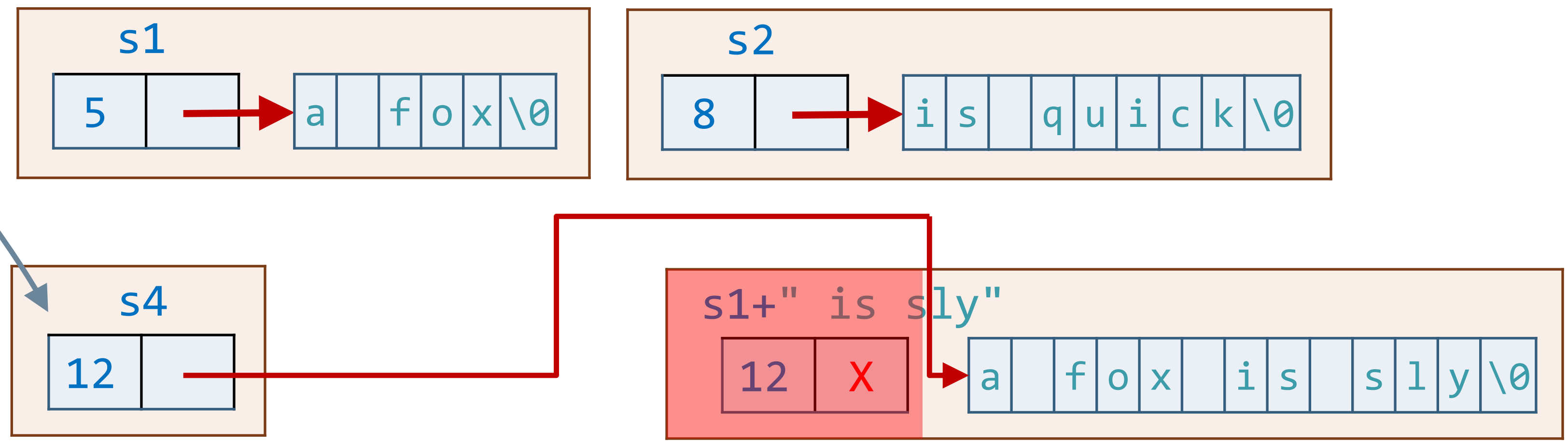


What We Want From C++?

- Instead,

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

- We want

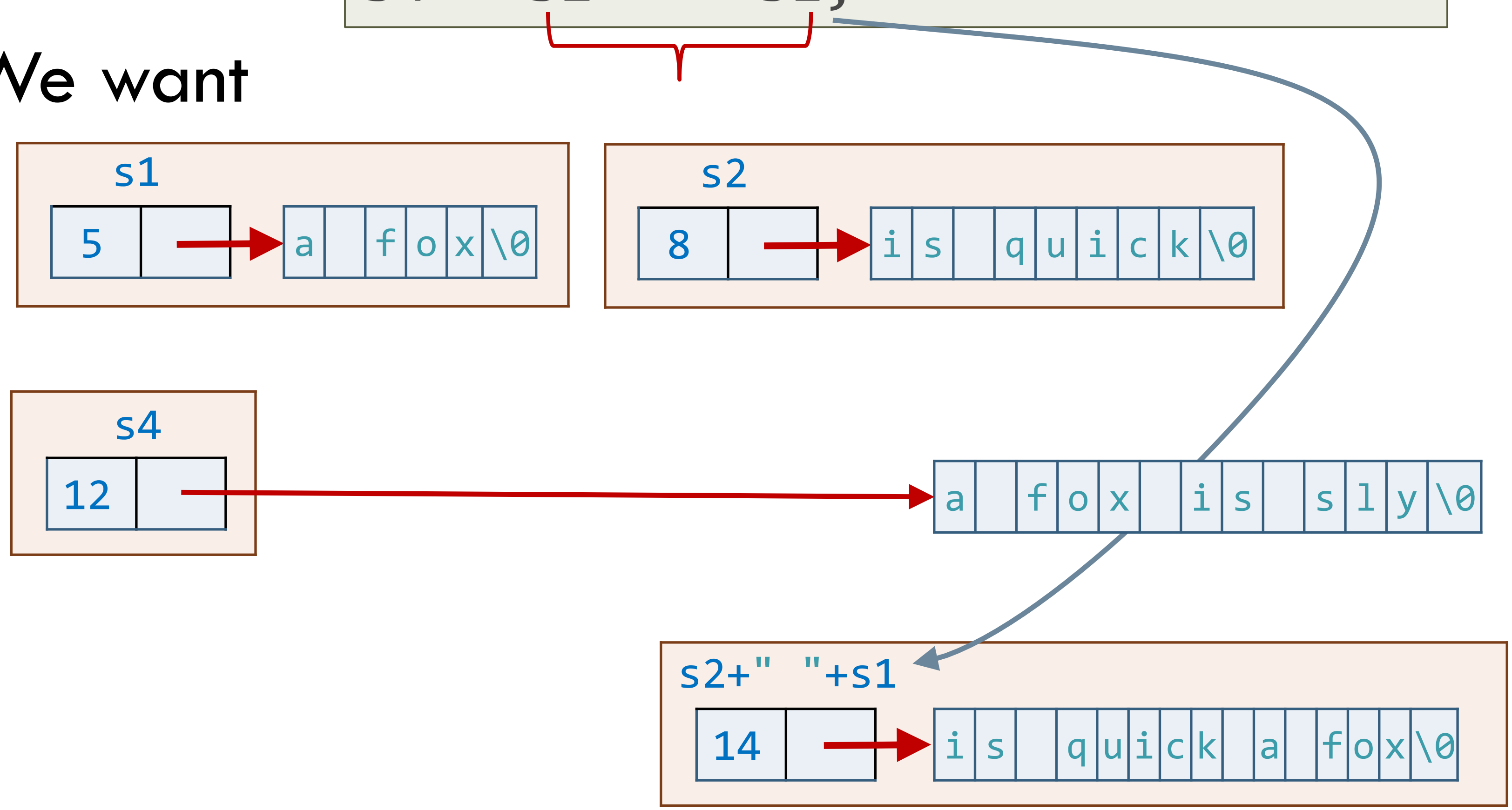


What We Want From C++?

- Instead,

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

- We want

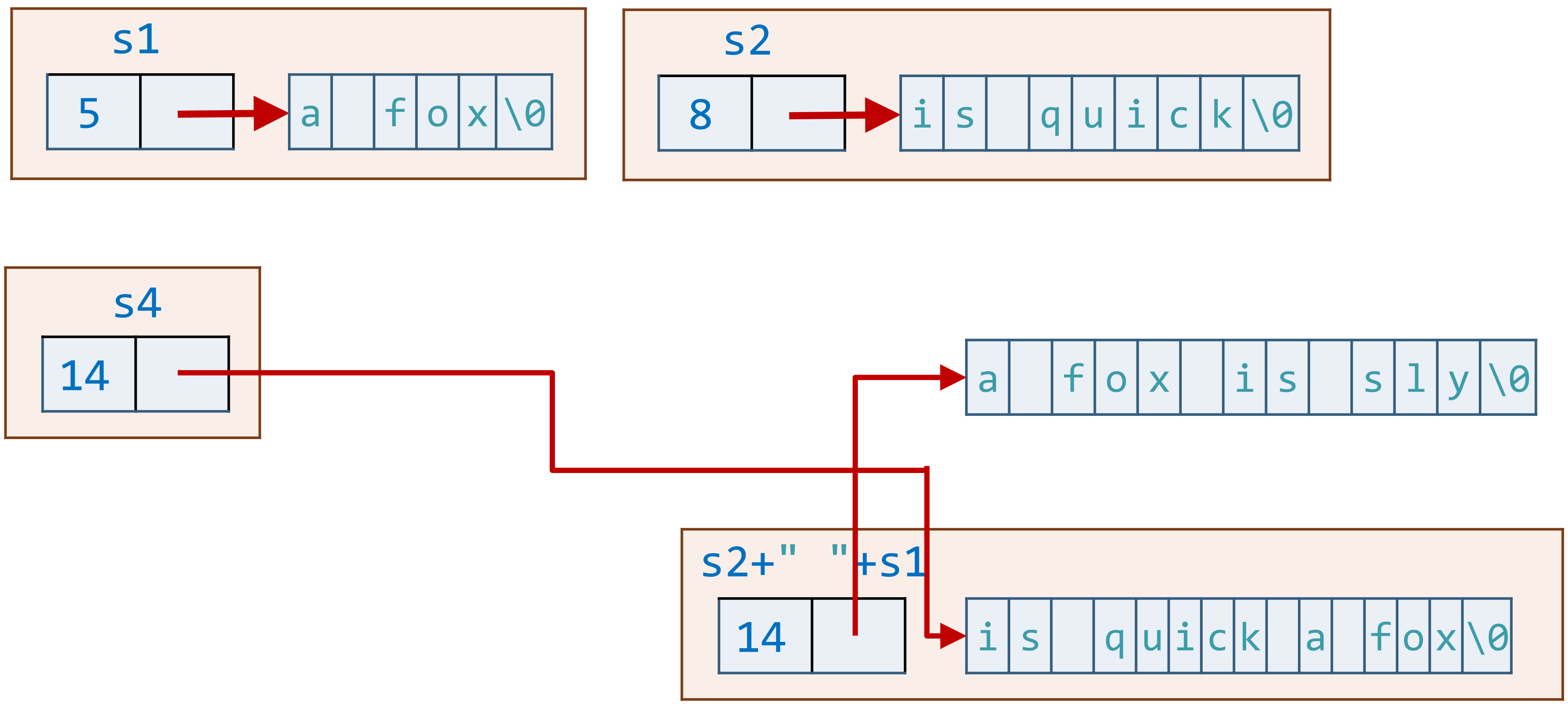


What We Want From C++?

- Instead,

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

- We want

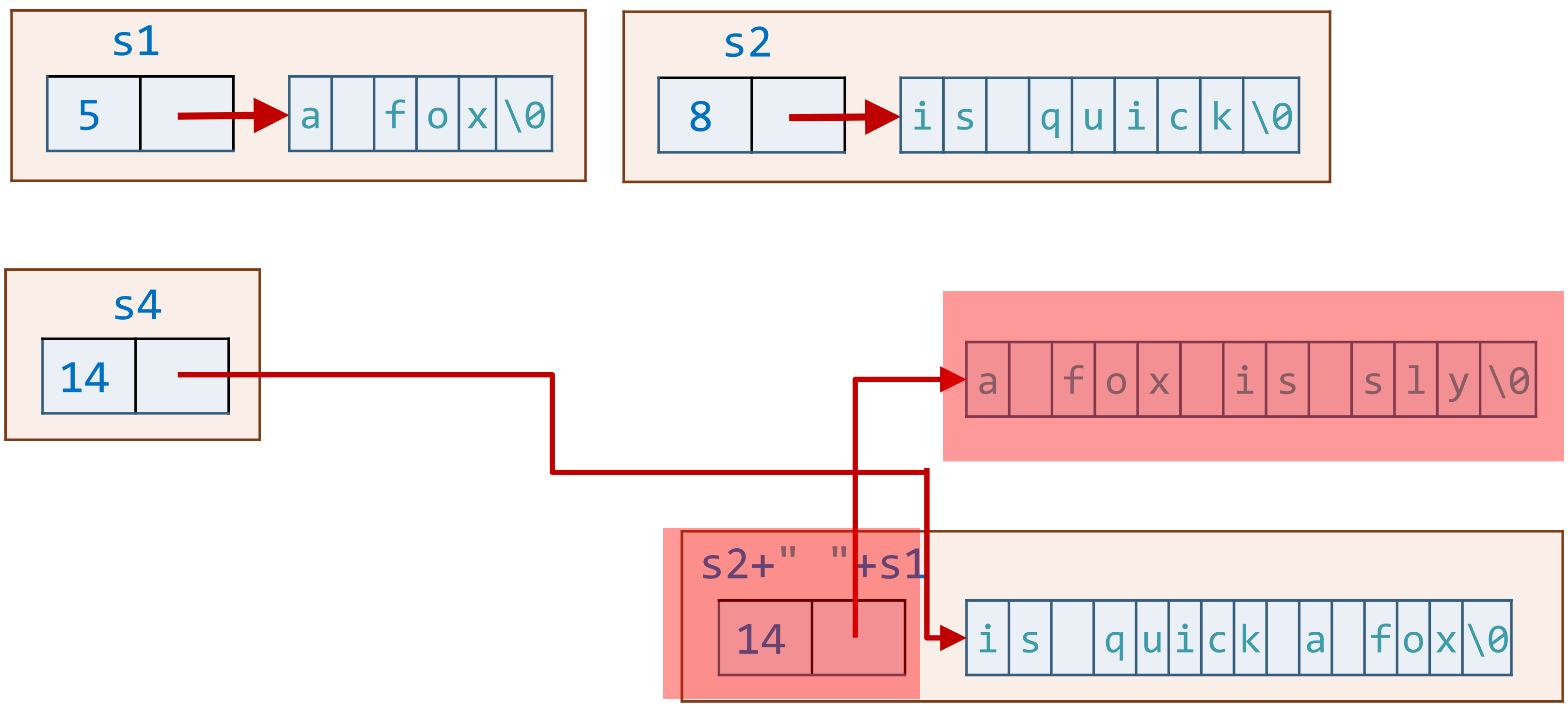


What We Want From C++?

- 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 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++?

- 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++?

- 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++?

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

```
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++?

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

```
X(<mystery type> rhs)
: member initializer list {
  // move resources of rhs to *this
  // set resources of rhs to null state
}

X& operator=(<mystery type> rhs) {
  // exchange resources between rhs and *this
}
```

What is `mystery type`?

- Must be some reference type
  - Copy ctor and assignment operator already defined to take ordinary reference
- Given two overloaded copy ctors or copy assignment functions where one is ordinary reference and other is `mystery type`, 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 an 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)


```
string var{"Iowa"};
string f();

string& r1 {var};   // bind r1 to lvalue var
string& r2 {f()};   // error: f() is rvalue
string& r3 {"Ohio"}; // error: can’t bind to temporary

string&& rr2{var};// error: var is lvalue
string&& rr1{f()};// bind rr1 to rvalue (a temporary)
string&& rr3{"Iowa"}; // rr3 refers to temporary
                      // encapsulating resource "Iowa"
const string& cr{"Iowa"}; // ok: make temporary
                          // and bind to cr
```

Rvalue References: Main Purpose

- Rvalue reference refers to a temporary object
- User of reference can (and typically will) modify object assuming it will not be used again
  - Implement "destructive read" for optimization of what would have required a copy

Rvalue References: Access

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

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

References: Recap

- Basic idea of having more than one kind of reference is to support different uses of objects:
  - Non`const` 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*)

Implementing Move Ctor

- 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)

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

Implementing Move Assignment


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

- 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

Default Operations

- By default, compiler generates each of these operations if a program uses it:
  - Default constructor: `X()`
  - Copy constructor: `X(const X&)`
  - Copy assignment: `X& op=(const X&)`
  - Move constructor: `X(X&&) noexcept`
  - Move assignment: `X& op=(X&&) noexcept`
  - Destructor: `~X()`

Rule Of Five

- All five copy-control members (excluding default ctor) must be thought of as a unit
- If a class defines any of these operations, it usually should define them all

Default Operations: Caveats

- If you define one or more of these operations, compiler suppresses generation of related operations:
  - If you declare any constructor, default constructor is not generated
  - If you declare a copy operation or a destructor, no move operation is synthesized

How Does Compiler Logic Work?

- How does compiler know when it can use a move operation rather than copy operation?
- In few cases, language rules say it can
  - Such as for `return` value – because next action is defined to destroy value

Move or Copy?


```
template<class T> class vector {
// ...
 vector(const vector& r); // copy r’s stuff
 vector(vector&& r);      // "steal" r’s stuff
};

vector<string> s{"head"};   // use ctor
vector<string> s2{s};       // use copy ctor
vector<string> s3{s+"tail"};// pick move ctor
```

Move or Copy?


```
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
}
```

How Does Compiler Logic Work?

- How does compiler know when it can use a move operation rather than copy operation?
- In few cases, language rules say it can
  - Such as for `return` value – because next action is defined to destroy value
- In general, you have to tell by giving an rvalue reference argument

Forcing Move Semantics (1/4)

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

```
template<typename T>
void swap(T& a, T& b) { // old-style swap
  T tmp{a};// two copies of a
  a = b;     // two copies of b
  b = tmp;  // now, we've two copies of tmp
}
```

Forcing Move Semantics (2/4)

- What if `T` is type for which it can be expensive to copy elements (`string`, `vector`, …)?
- Then, `swap()` becomes quite expensive
- Serious problem
- We didn't want any copies at all - just want to move values of `a`, `b`, and `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`
- Operation optimized for rvalues can now use its optimization for `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() (1/7)

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

```
template<class T> // almost perfect "swap"
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()` (2/7)

- `swap()` is “almost perfect” because it will swap only lvalues not rvalues

```
void f(vector<int>& v) {
// ...
 swap(v, vector<int>{1,2,3,4}); //error
// ...
}
```

`std::move()` (3/7)

- Solution is to augment `swap()` by two overloads:

```
template<class T> void swap(T&&, T&);
template<class T> void swap(T&, T&&);
```

- Standard-library takes this approach

`std::move()` (4/7)

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

`std::move()` (5/7)

- That is, `move()` is used to tell compiler that an object will not be used anymore in a context, 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()` (6/7)

- 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
  - This means that we can use types that are moveable but not copyable such as `unique_pointer`

`std::move()` (7/7)

- 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<X&&>(x)`?

Overloading Rules for Rvalue and Lvalue References

- 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

- If you implement only

```
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 lvalues but not for rvalues

Overloading Rules for Rvalue and Lvalue References

- If you implement only

```
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 lvalues and for rvalues

Overloading Rules for Rvalue and Lvalue References

- If you implement either

```
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
```

or

```
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 rvalues

Overloading Rules for Rvalue and Lvalue References

- If you 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 on rvalues but not on lvalues

Overloading Rules for Rvalue and Lvalue References

- 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 (1/6)

- Reference used to specify function argument so that function can change value of an object passed to it:

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

void foo() {
  int x = 1;
  increment(x); // x = 2
}
```

Reference Arguments (2/6)

- To keep program readable, often best to avoid functions that modify their arguments:

```
int next(int v) { return v+1; }

void boo() {
  int x = 1;
  increment(x); // x = 2
  x = next(x);  // x = 3
}
```

Reference Arguments (3/6)

- An rvalue expression can be bound to an rvalue reference (but not to an lvalue reference)
- An lvalue expression can be bound to an lvalue reference (but not to an rvalue reference)
- For example:

Reference Arguments (4/6)


```
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
}
```

Reference Arguments (5/6)

- If you implement `foo(X&)` but not `foo(X&&)`, `foo` can be called only on lvalues and not on rvalues
- If you implement `foo(const X&)` but not `foo(X&&)`, `foo` can be called on both lvalues and rvalues
  - Not possible to distinguish between lvalues and rvalues
- If you only implement `foo(X&&)`, `foo` can be called only on rvalues but not on lvalues

Reference Arguments (6/6)

- How do we choose among the many ways of passing arguments?
- Follow Stroustrup's rules of thumb (page 318)

Rules Of Thumb

- 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 a 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

Rules Of Thumb

- 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 a 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