COPY ELISION & MOVE SEMANTICS

Plan for Today

- Copy Elision: RVO, NRVO, URVO
- Move Semantics (Motivation)

RAII Vector Class

```
class Vec {
  size_t len{};
  int    *ptr{nullptr};
public:
  Vec() = default;
  ~Vec() { delete [] ptr; }
  Vec(Vec const& rhs)
  : len{rhs.len}, ptr{new int [len]} {
    std::copy(rhs.ptr, rhs.ptr+len, ptr);
  }
  Vec& operator=(Vec const& rhs) {
    Vec copy{rhs};
    copy.swap(*this);
    return *this;
  }
};
```

RAII Classes: Rule of Three

- If your class manages a resource, you'll need to write three special member functions:
  - Destructor to release the resource
  - Copy constructor to clone the resource
  - Copy assignment operator to release current resource and clone resource of assigned object

C++'s Copy Problem

- Perception that C++ is overly fond of copying
  - Pass-by-value means invoking copy constructor
  - Return-by-value means invoking copy constructor
  - Assignment means invoking copy assignment operator
  - STL containers employ value semantics

C++'s Copy Problem

- Based on our understanding of stack-based function semantics in C/C++, one would categorically assert that every invocation of following functions requires invocation of copy ctor

```
void foo(X xx) {
  // use xx
}

int main() {
  X x;
  // use x
  foo(x);
  // use x
}
```

```
X bar() {
  X xx;
  // process xx
  return xx;
}

int main() {
  X x = bar();
  // use x
}
```

C++'s Copy Problem

- Based on our understanding of stack-based function semantics in C/C++, one would categorically assert that every invocation of following functions requires invocation of copy ctor
- Pass-by-reference becomes default mode of transferring resources to functions

Return Value Optimization [RVO]

- Most high-quality C++ implementations allow copy *elision* [that is, *omit copying*] even in cases where copy ctors and dtors may have side effects
  - Copy elision now part of C++17
- This compiler optimization is more commonly called *Return Value Optimization*
  - Avoids copying object that function returns as its value
  - Avoids creation of temporary object
  - Permits function to efficiently return large objects
  - Simplifies function's interface
  - Eliminates scope for issues such as resource leaks

Functions: Pass-by-Value Convention (1/20)

this variable is called *formal parameter* or just *parameter*

```
int myabs(int number) {
  return number < 0 ? -number : number;
}
```

client calls function myabs using function call operator ()

```
int num = 10;
num = myabs(-num)
```

this expression is called *function argument*

1) At runtime, expression (or argument) `-num` is evaluated
2) Result of evaluation is used to initialize parameter `number`
3) Changes made to parameter `number` are localized to function `myabs`
4) Function `myabs` terminates by returning value of type `int`
5) When function `myabs` terminates, variable `number` ceases to exist

Functions: Pass-by-Value Convention (2/20)

- Example

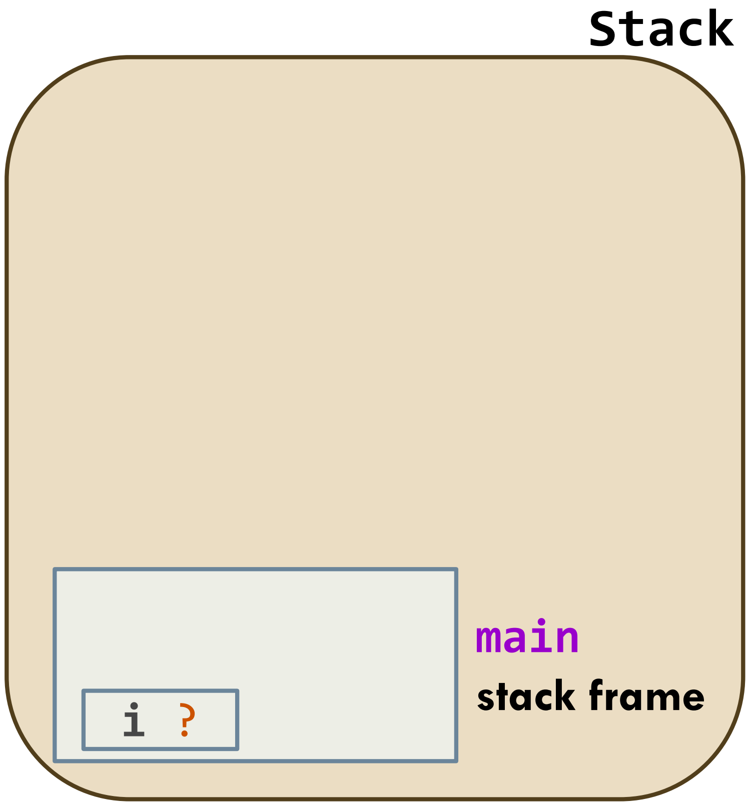
```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

- Output

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```

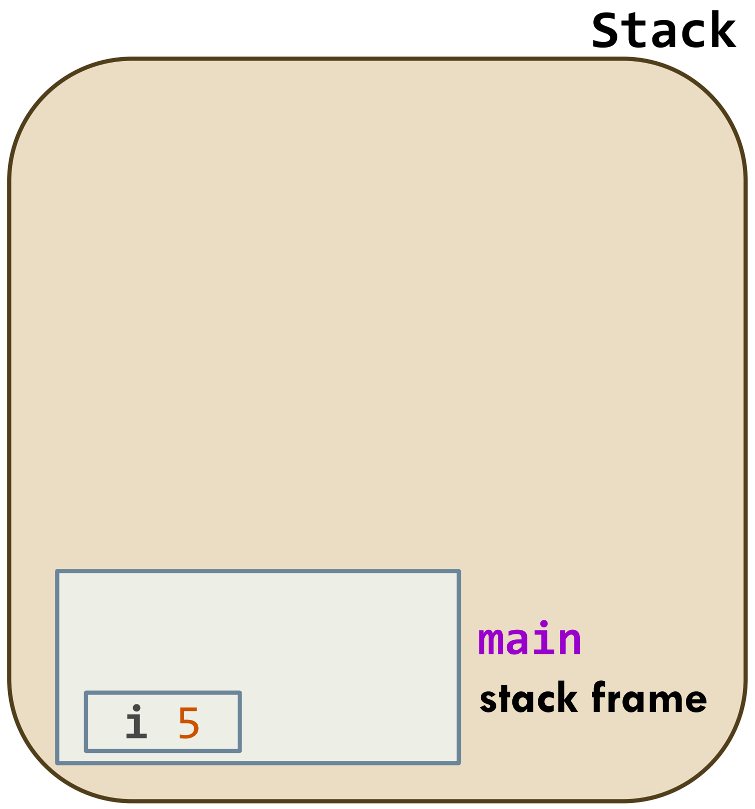
Functions: Pass-by-Value Convention (3/20)

```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



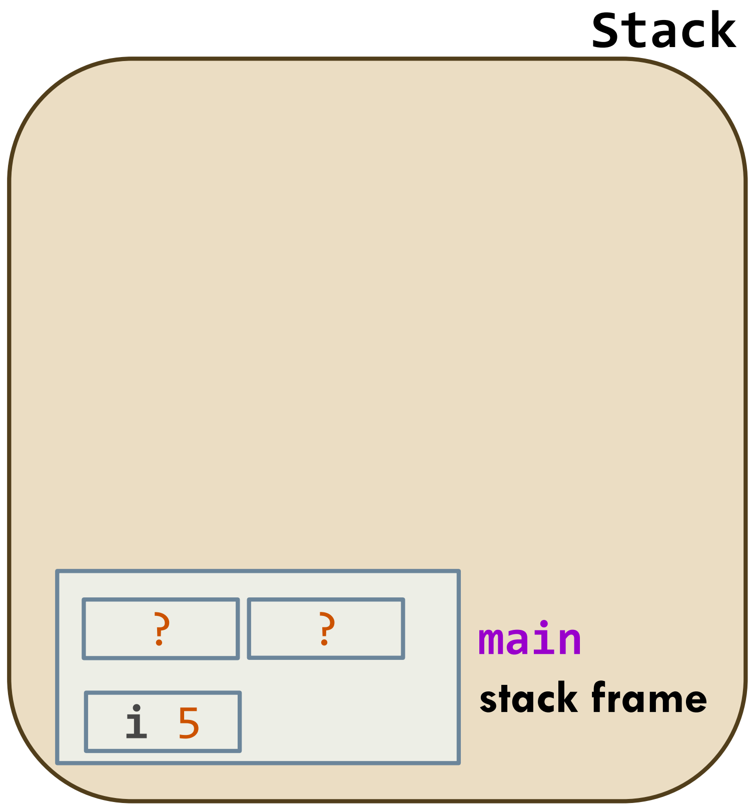
Functions: Pass-by-Value Convention (4/20)

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



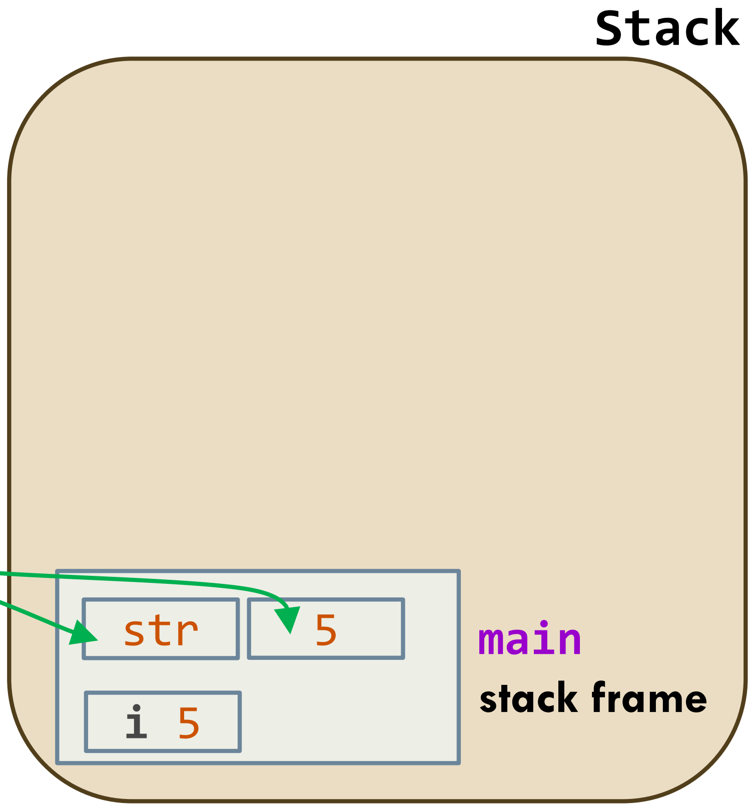
Functions: Pass-by-Value Convention (5/20)

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (6/20)

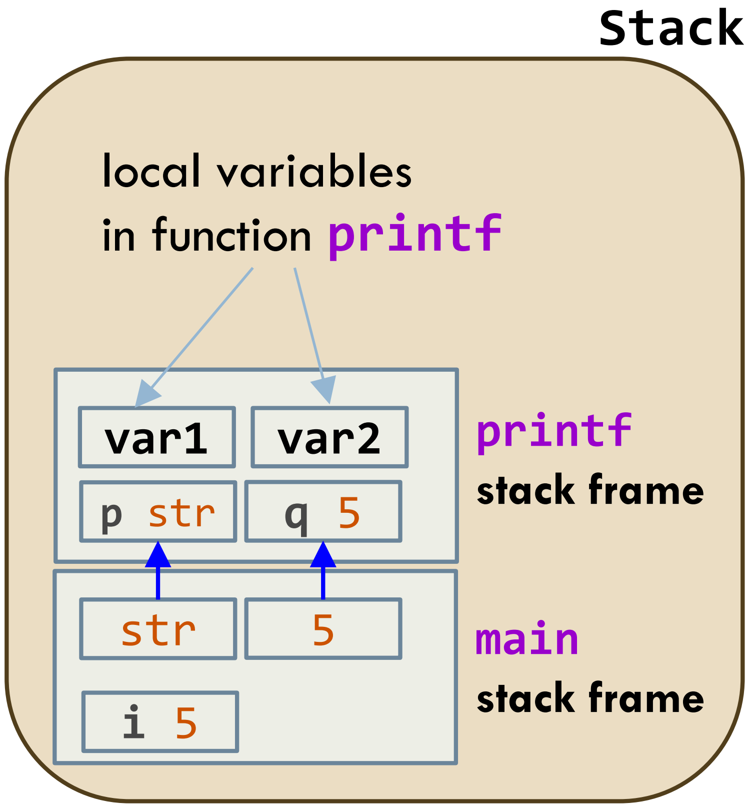
```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (7/20)

```
Before call: i is 5
```

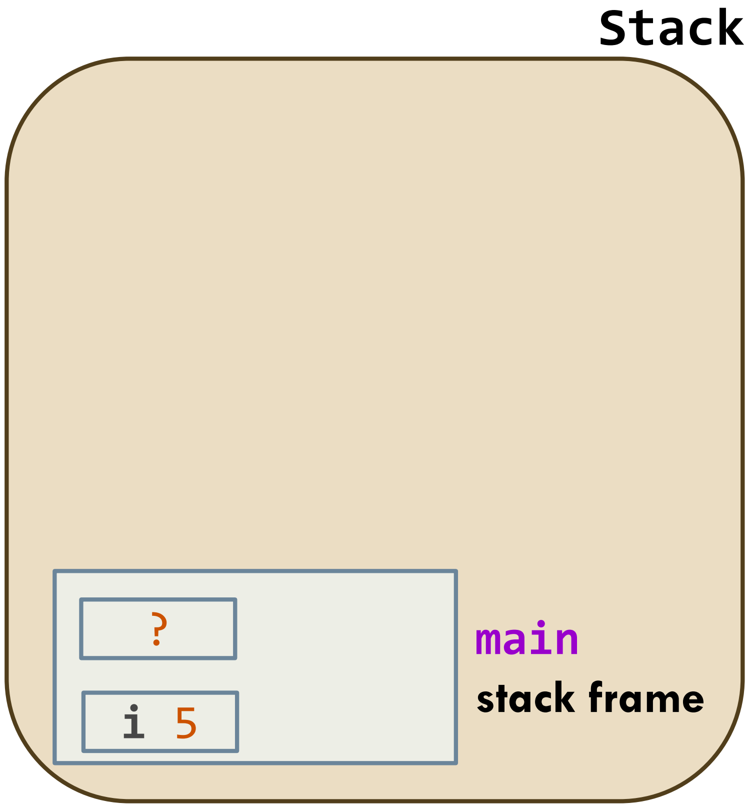
```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (8/20)

```
Before call: i is 5
```

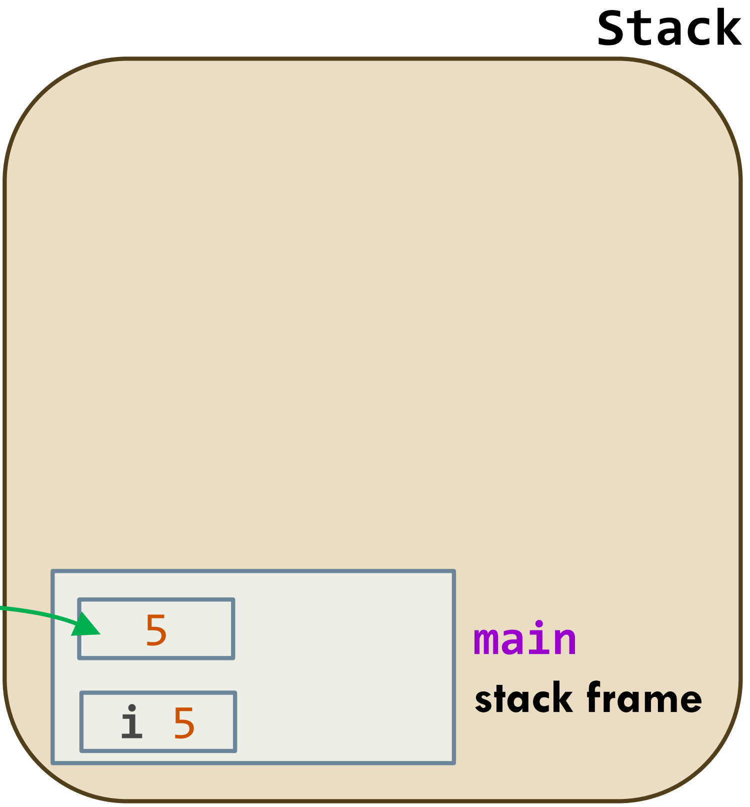
```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (9/20)

```
Before call: i is 5
```

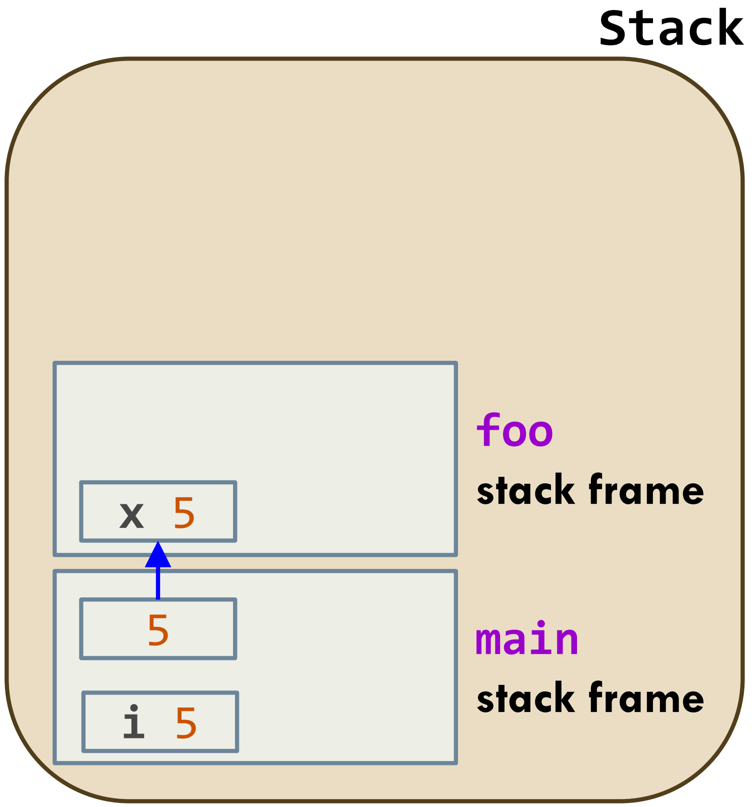
```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (10/20)

```
Before call: i is 5
```

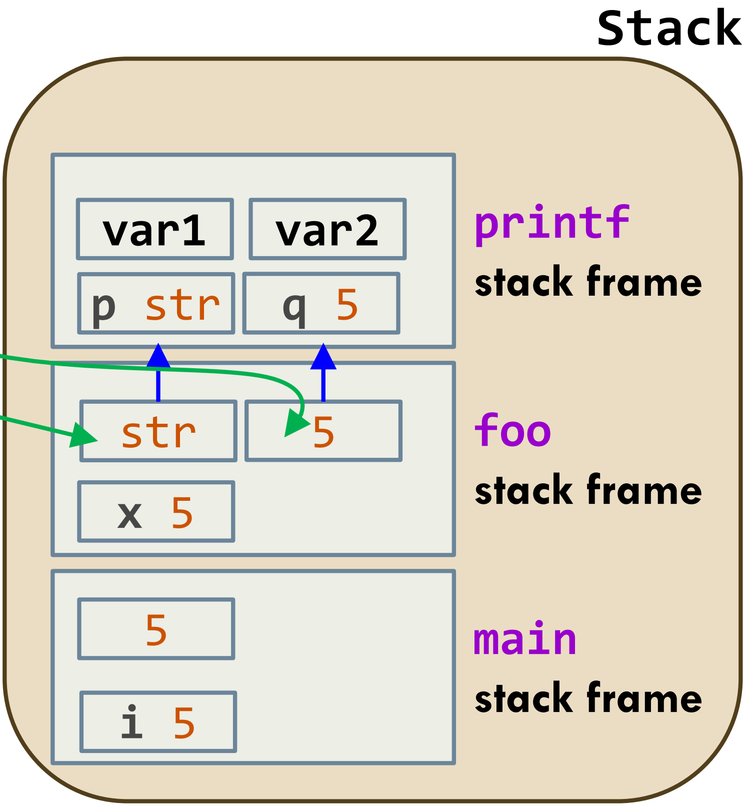
```c
#include <stdio.h>

void foo(int x) {
   printf("In foo, x is %d\n", x);
   x = 10;
   printf("In foo, x is now %d\n", x);
}

int main(void) {
   int i;
   i = 5;
   printf("Before call: i is %d\n", i);
   foo(i); // call to function foo
   printf("After call: i is %d\n", i);
   return 0;
}
```



Functions: Pass-by-Value Convention (11/20)

```
Before call: i is 5
In foo, x is 5
```

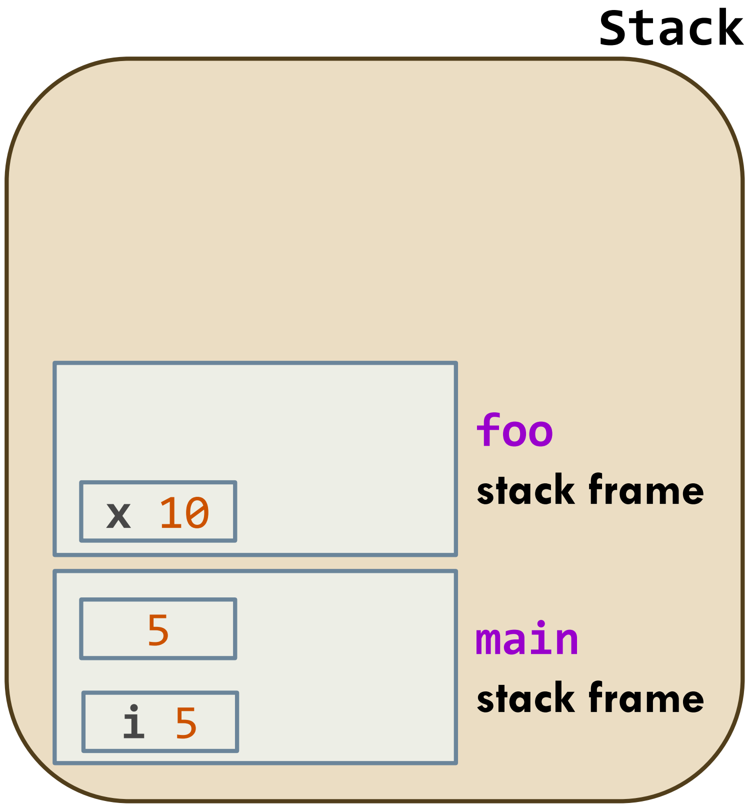
```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (12/20)

```
Before call: i is 5
In foo, x is 5
```

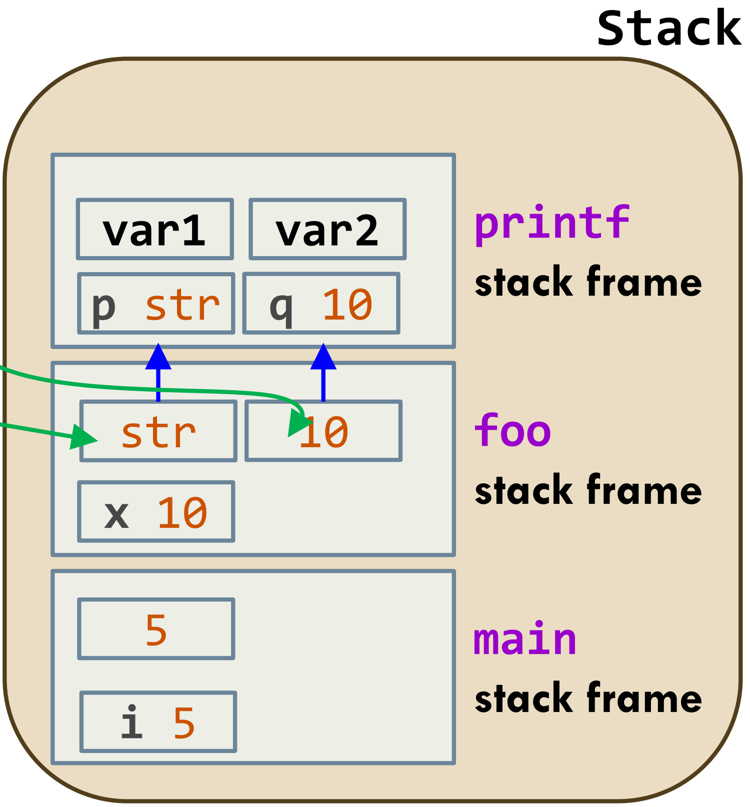
```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (13/20)

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
```

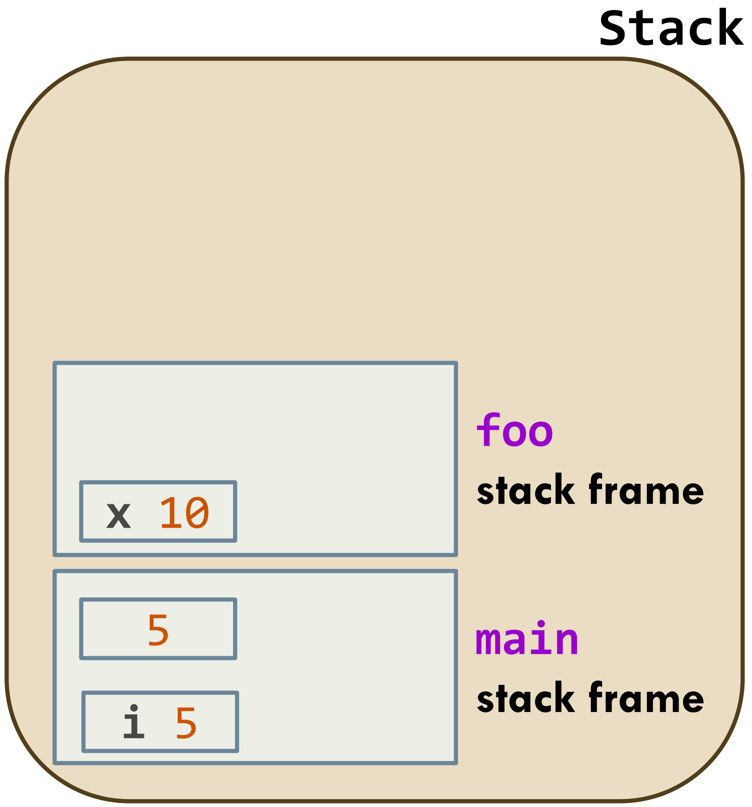
```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (14/20)

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
```

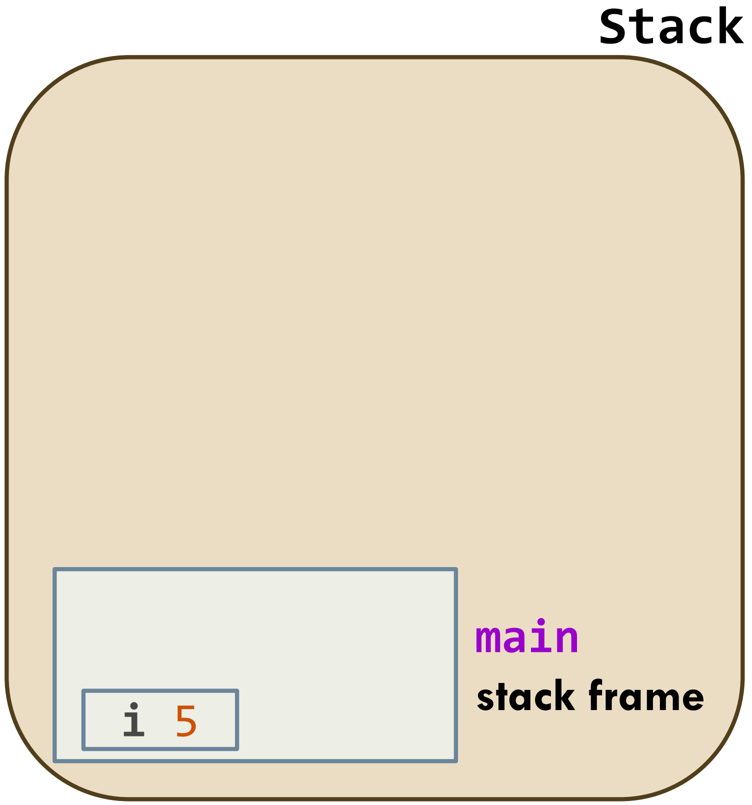
```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (15/20)

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
```

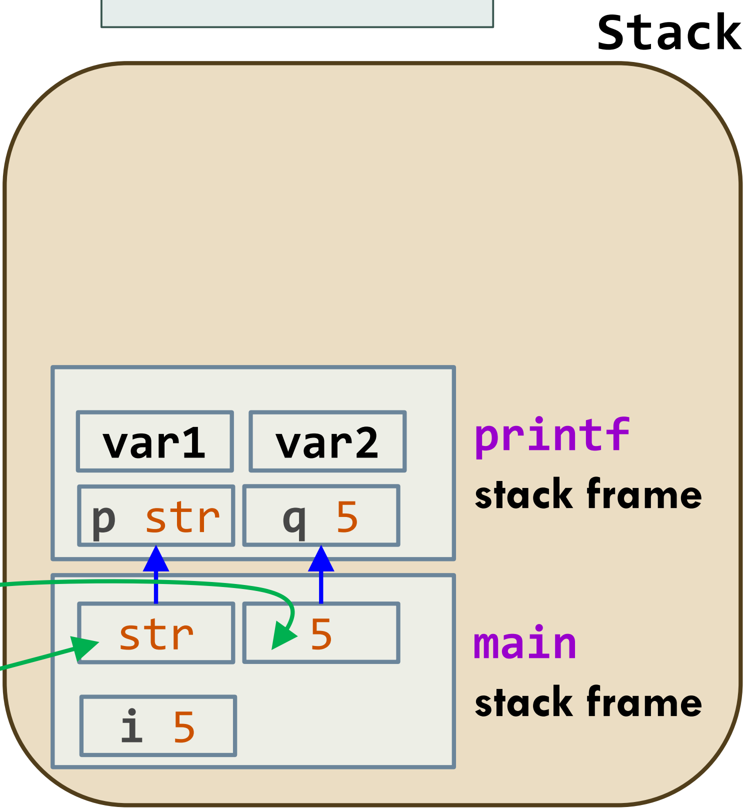
```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (16/20)

```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

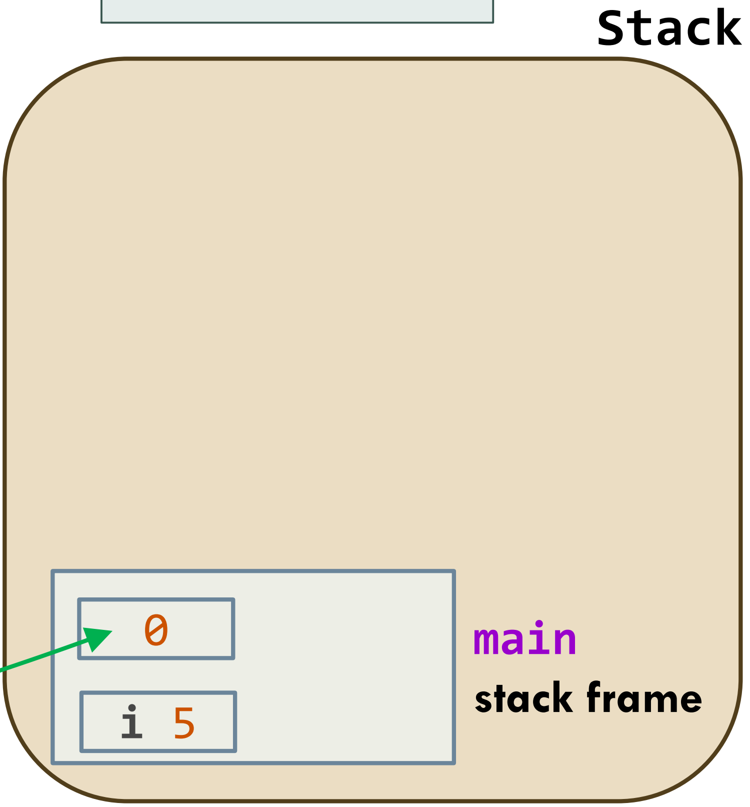
```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```



Functions: Pass-by-Value Convention (17/20)

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

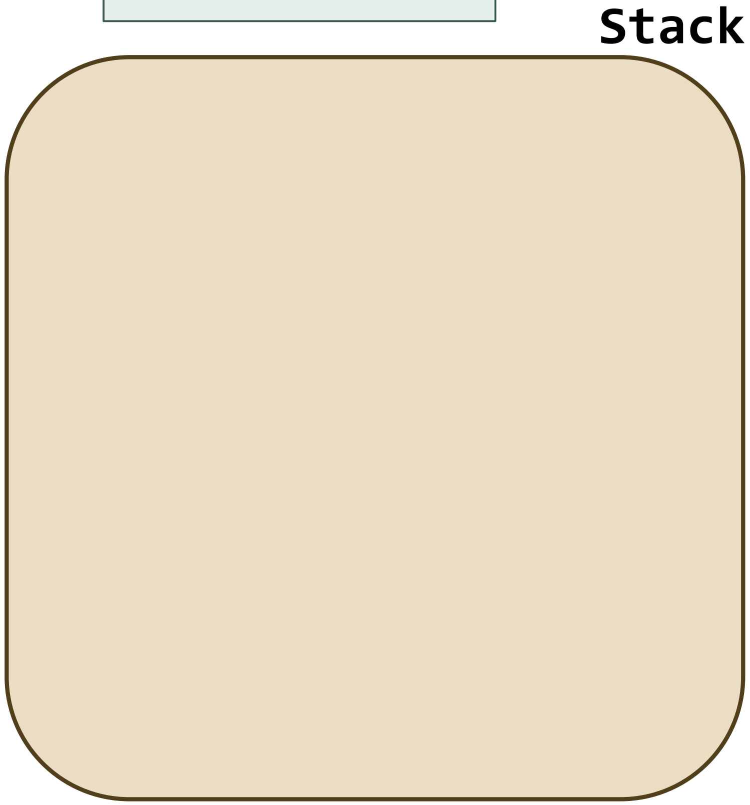
```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```



Functions: Pass-by-Value Convention (18/20)

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



Functions: Pass-by-Value Convention (19/20)

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

Main takeaway:
Inter-function communication uses *pass-by-value* semantics. Using the *stack*, copy of argument `i` is passed to function `foo` to initialize parameter `x`.
Changes made to parameter `x` do not affect argument `i`!!!

Functions: Pass-by-Value Convention (20/20)

- Visualization of program

```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

RVO, NRVO, URVO

- URVO: copy elision of unnamed objects
- NRVO: copy elision of named objects
- RVO: copy elision of named and unnamed objects

Copy Elision in C++17

- Compilers are required to provide copy elision when function returns unnamed [temporary] object
- Not required to provide copy elision when function returns named object
- Whether copy elision helpful or not depends on how function's return value is consumed

URVO

```
// without URVO
Str urvo(char const *prc) {
  return Str{prc}; // 1) ctor
} // 2) copy ctor for unnamed copy
// 3) dtor of temporary

int main() {
  // 4) copy ctor for s
  Str s = urvo("s");
 // 5) dtor for unnamed copy 2)
} // 6) dtor for s
```

```
// with URVO
Str urvo(char const *prc) {
  // 1) ctor for s in calling
  // environment
  return Str{prc};
}

int main() {
  Str s = urvo("s");
} // 2) dtor for s
```

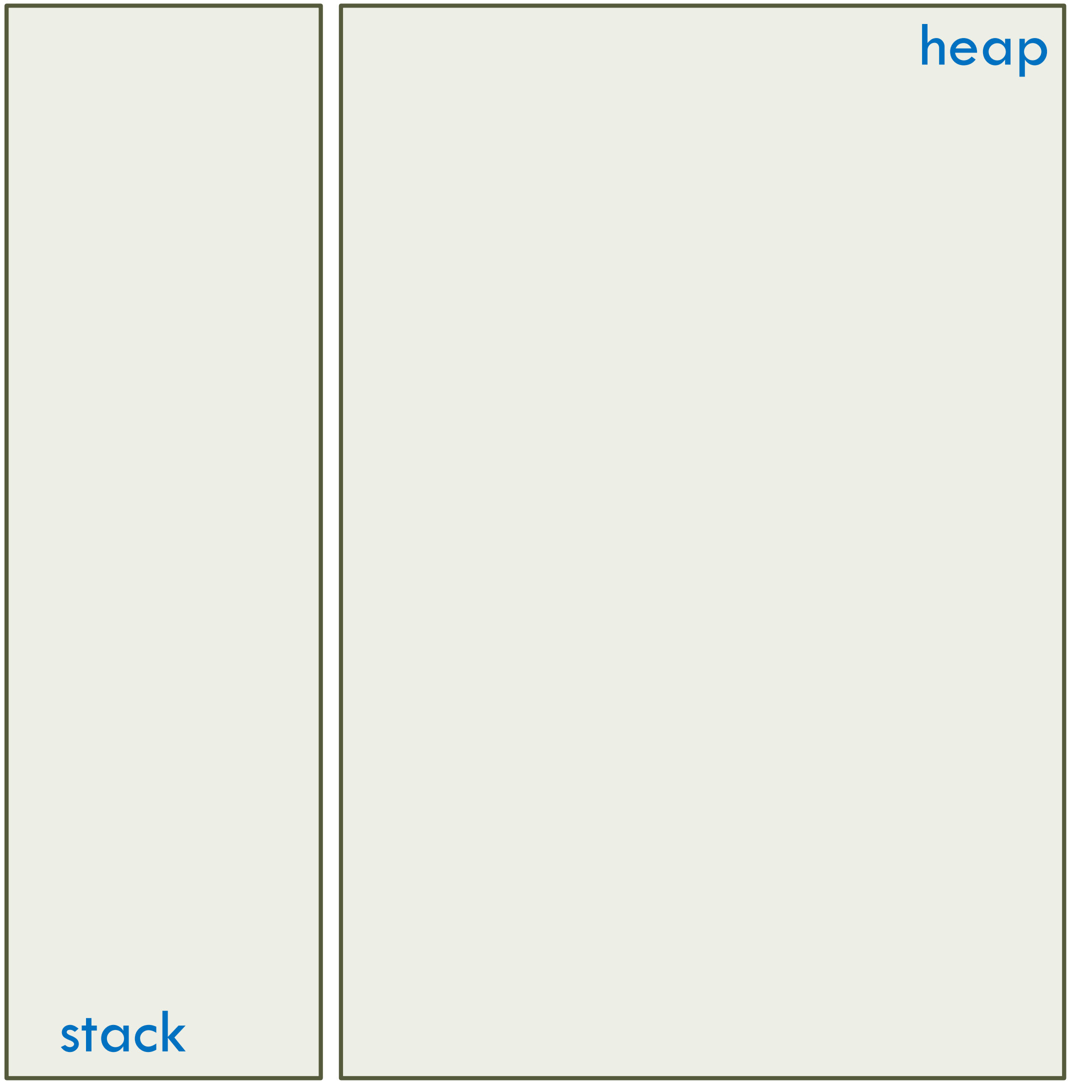
NRVO

```
// without NRVO
Str nrvo(char const *prc) {
  Str x{prc}; // 1) ctor for x
  // process x
  return x; // 2) copy ctor
} // 3) dtor for x

int main() {
  // s constructed by step 2
  Str s = nrvo("s"); s
} // 4) dtor for s
```

```
// with NRVO
Str nrvo(char const *prc) {
 Str x{prc}; // 1) ctor for s
 // process x
 return x;
}

int main() {
  Str s = nrvo("s");
} // 2) dtor for s
```

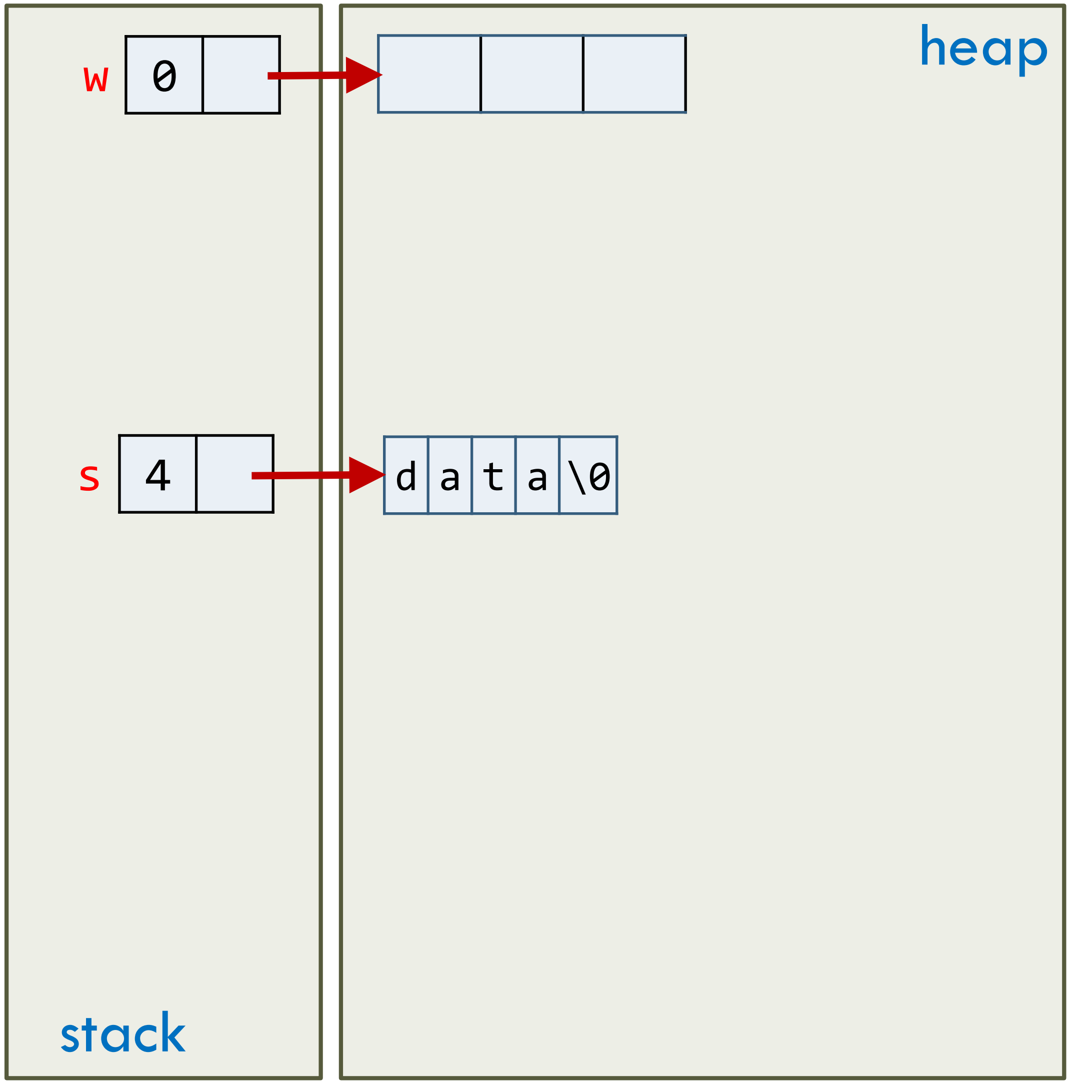
Motivation for RVO

```
std::vector<Str> f98() {
   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 = f98();
```



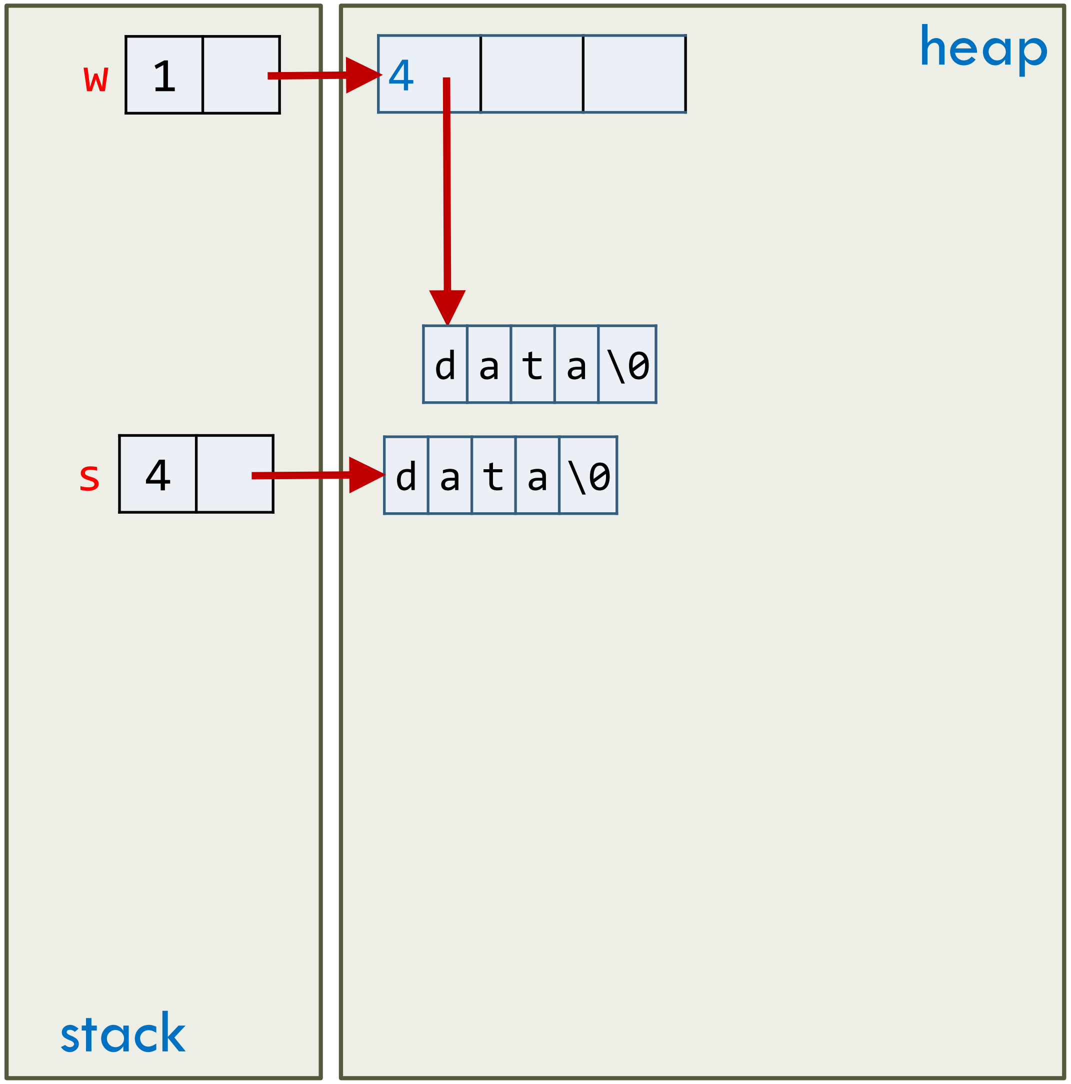
Motivation for RVO

```
std::vector<Str> f98() {
  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 = f98();
```



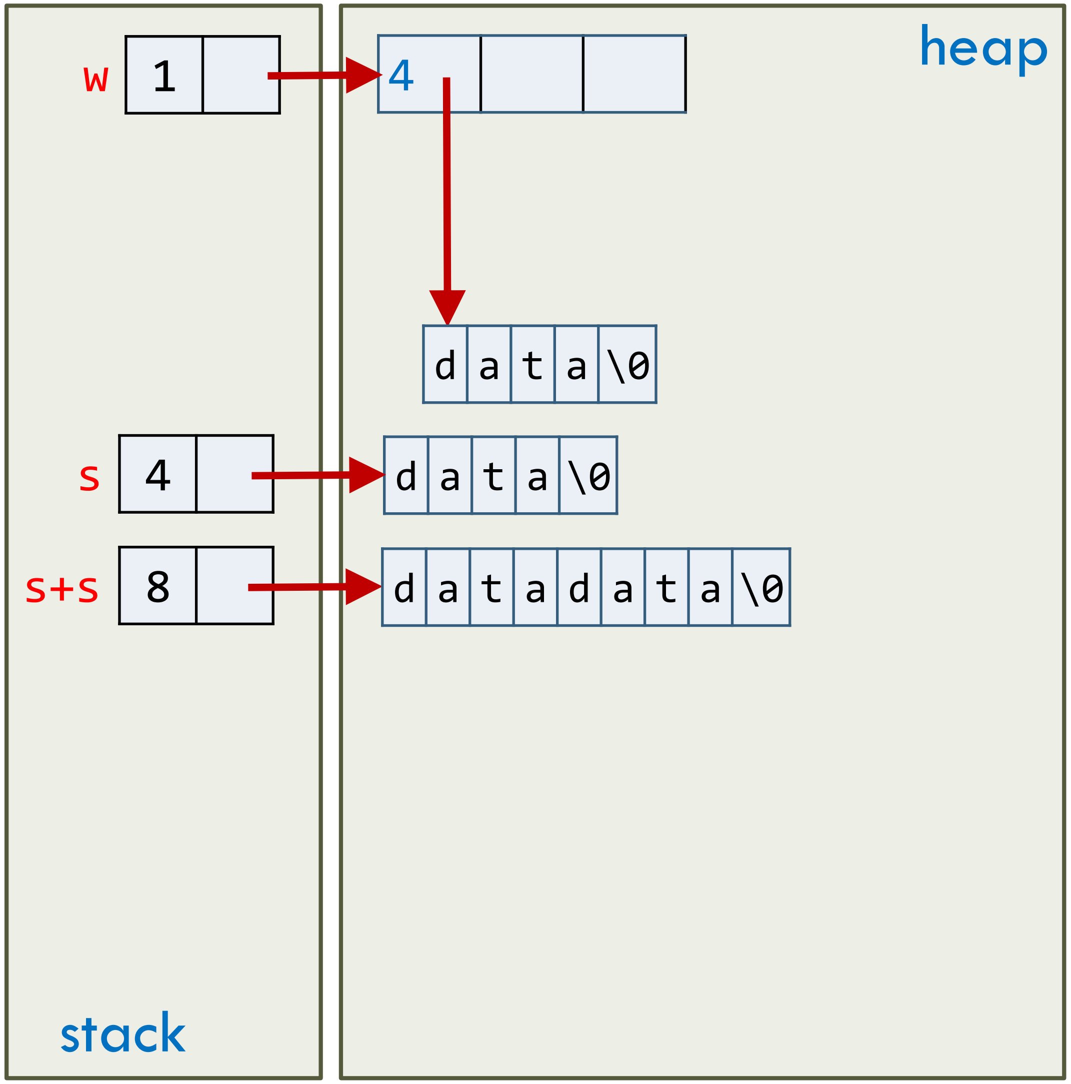
Motivation for RVO

```
std::vector<Str> f98() {
   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 = f98();
```



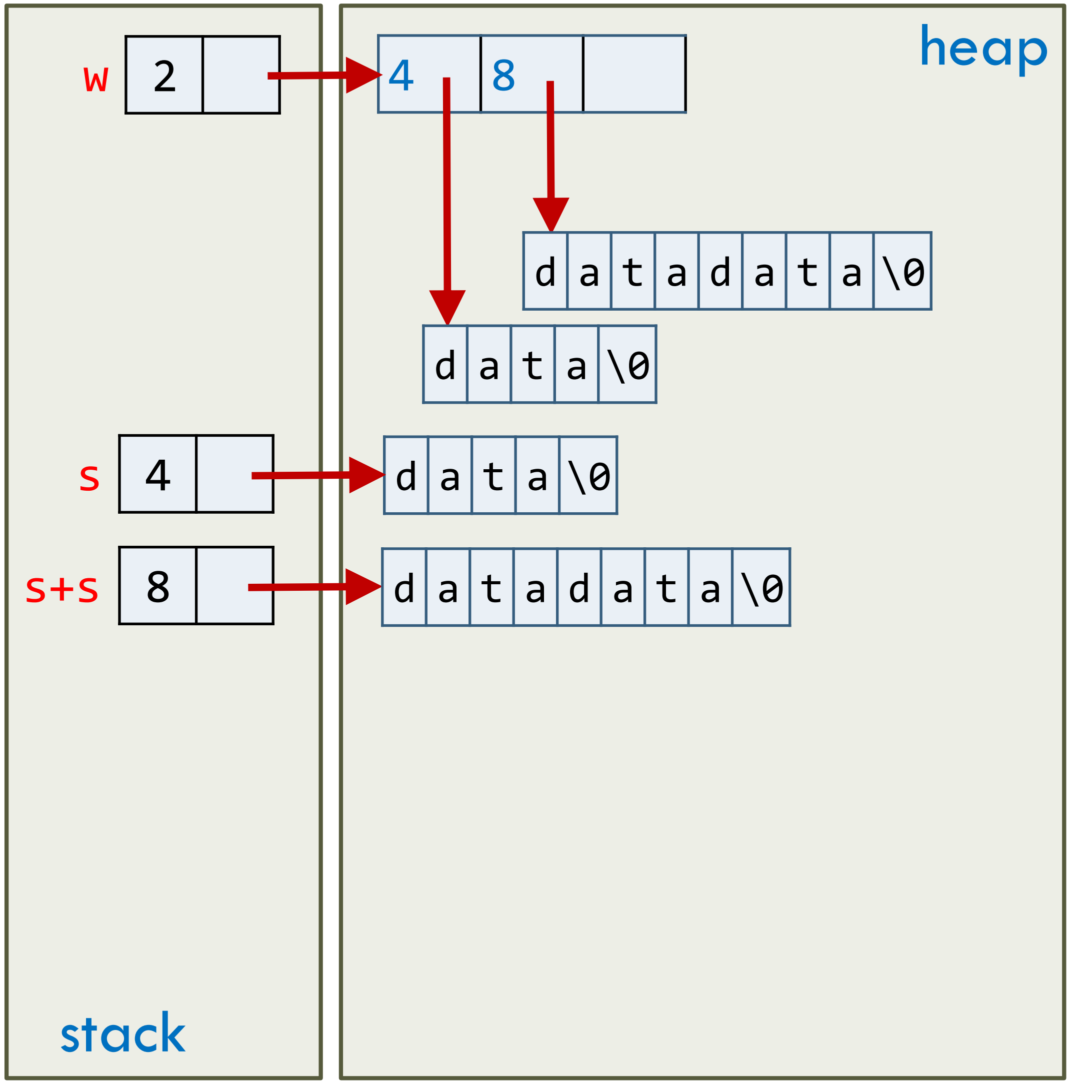
Motivation for RVO

```
std::vector<Str> f98() {
  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 = f98();
```



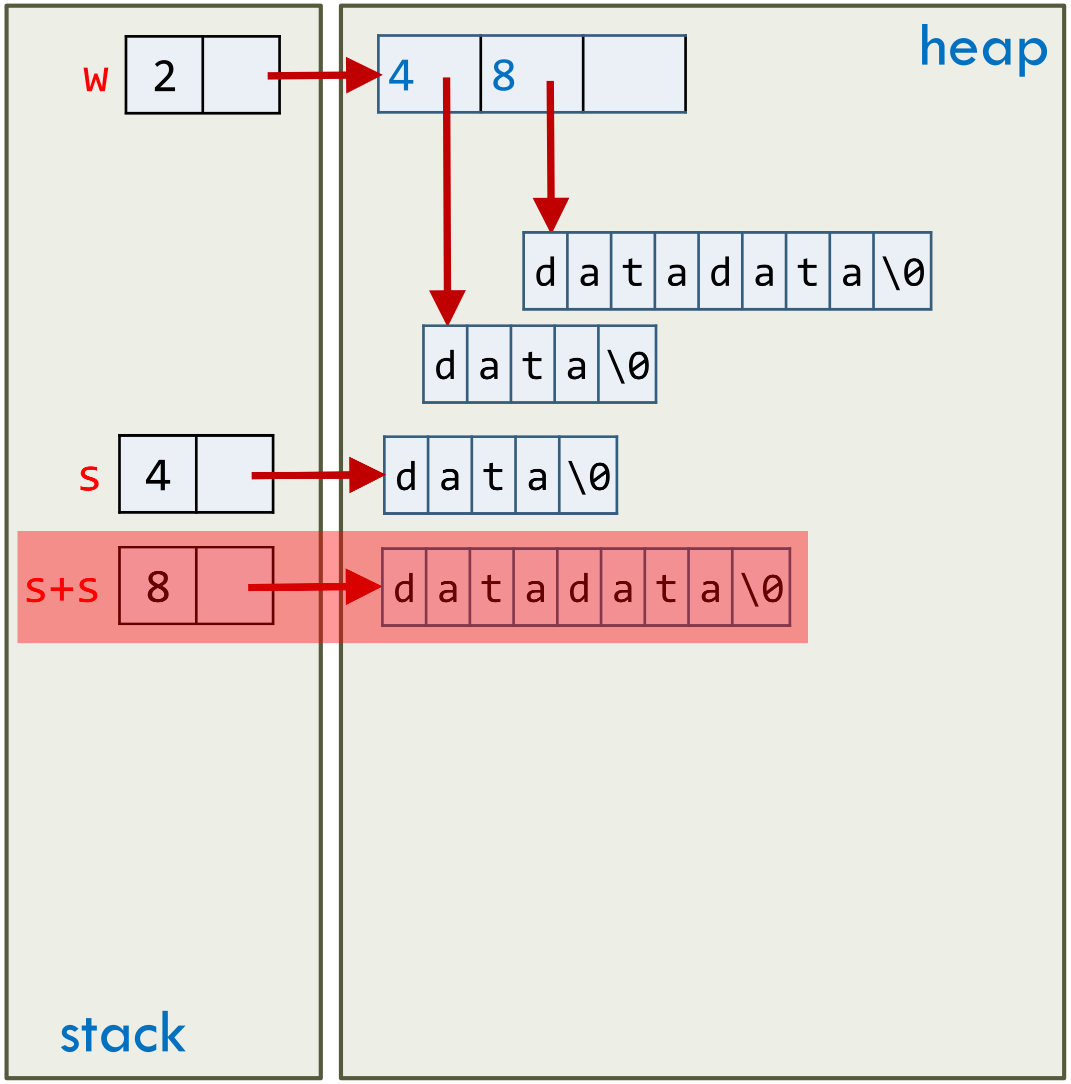
Motivation for RVO

```
std::vector<Str> f98() {
  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 = f98();
```



Motivation for RVO

```
std::vector<Str> f98() {
   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 = f98();
```

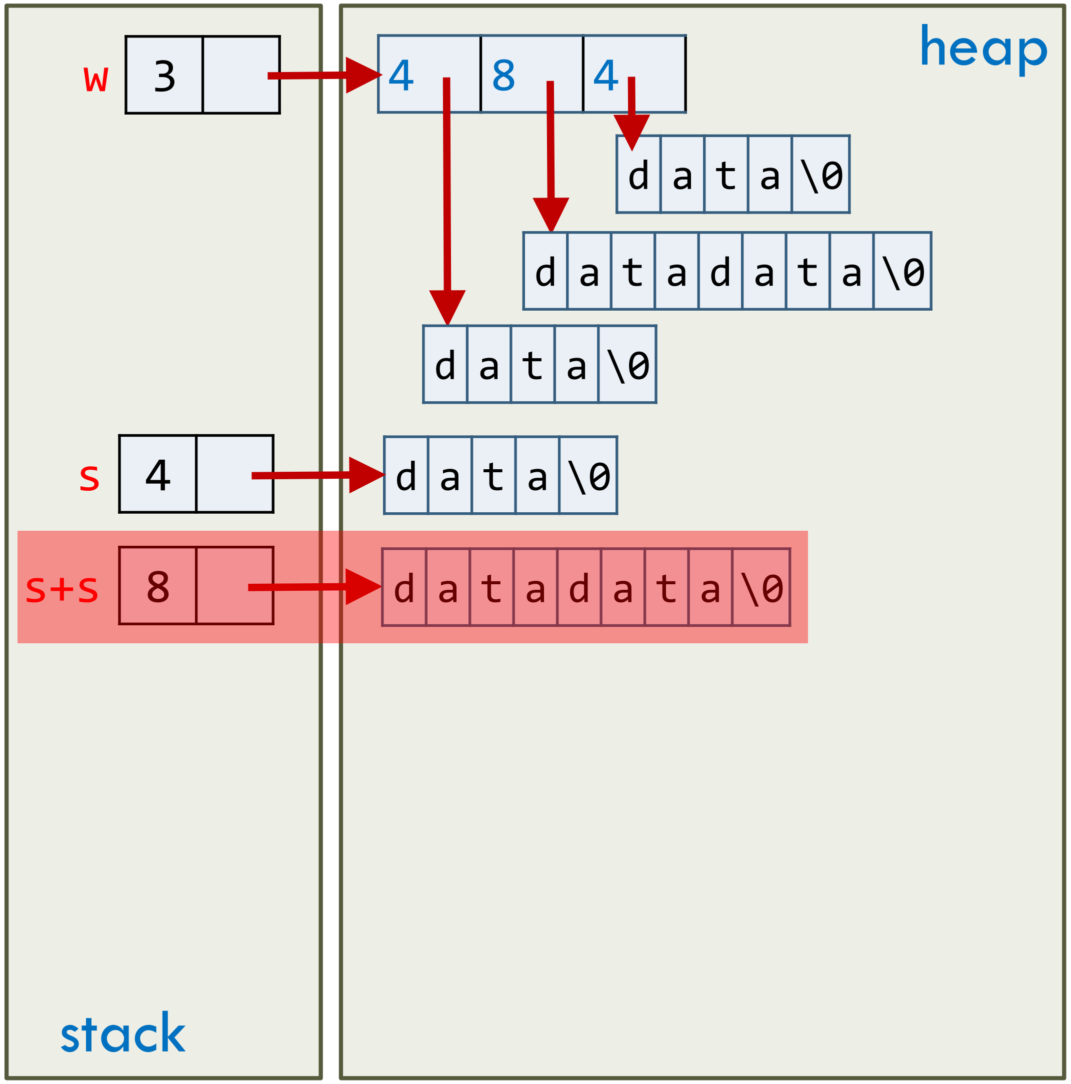


Motivation for RVO

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

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

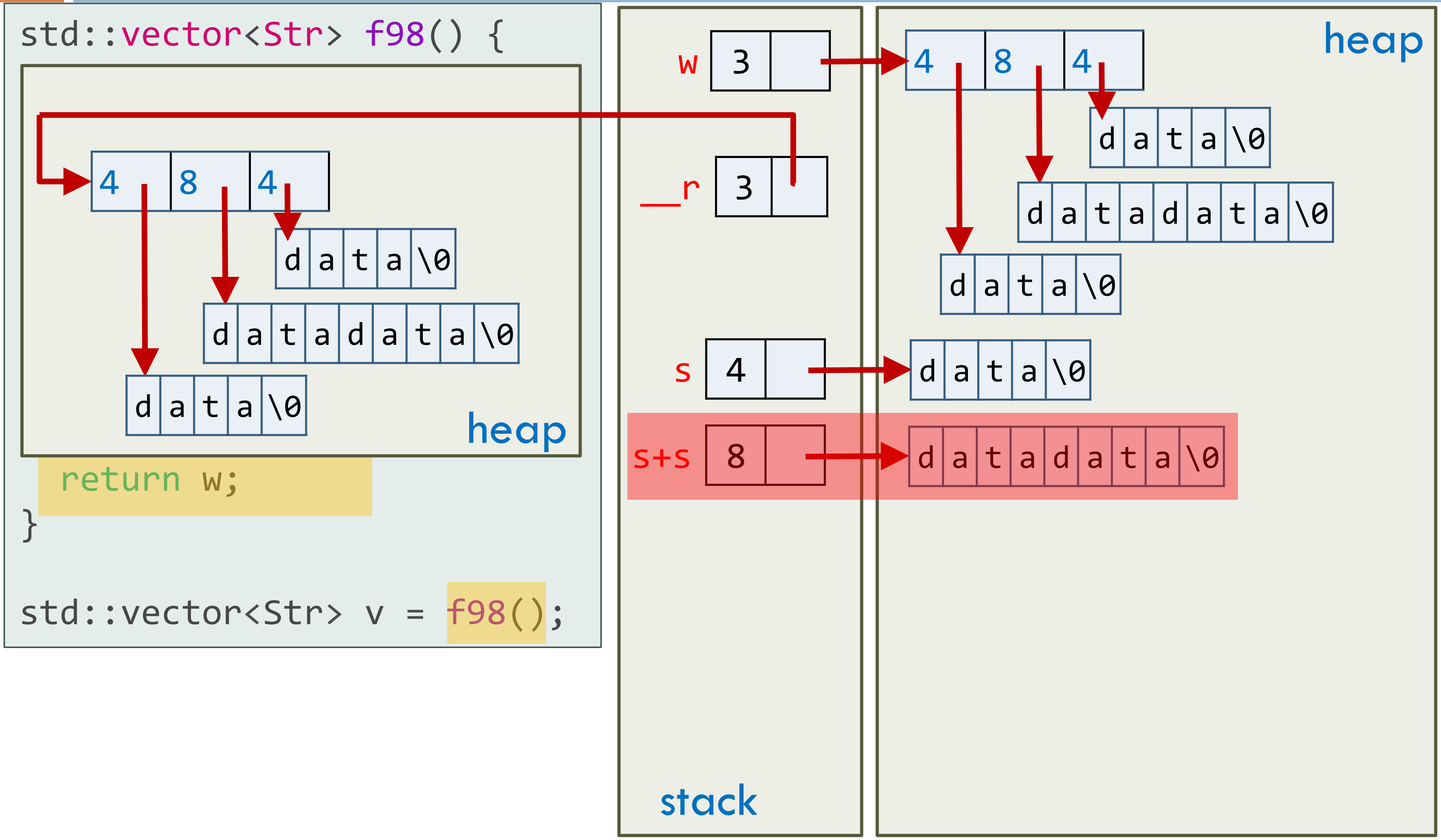
Motivation for RVO

```
std::vector<Str> f98() {
  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 = f98();
```



Motivation for RVO

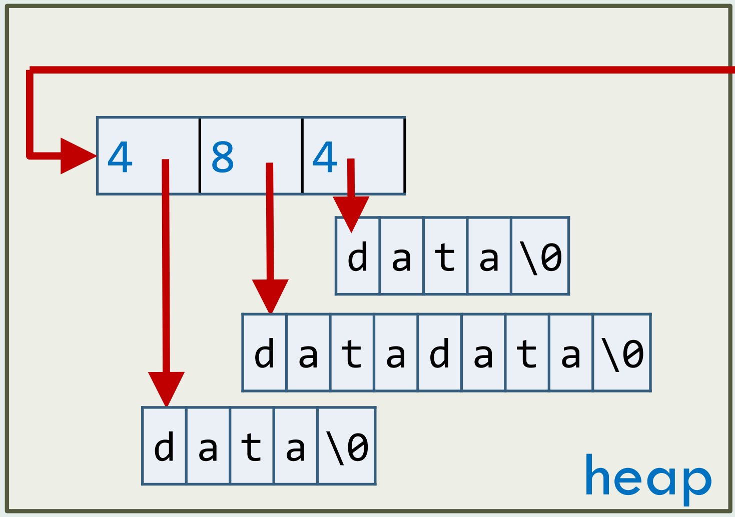
```
std::vector<Str> f98() {
```



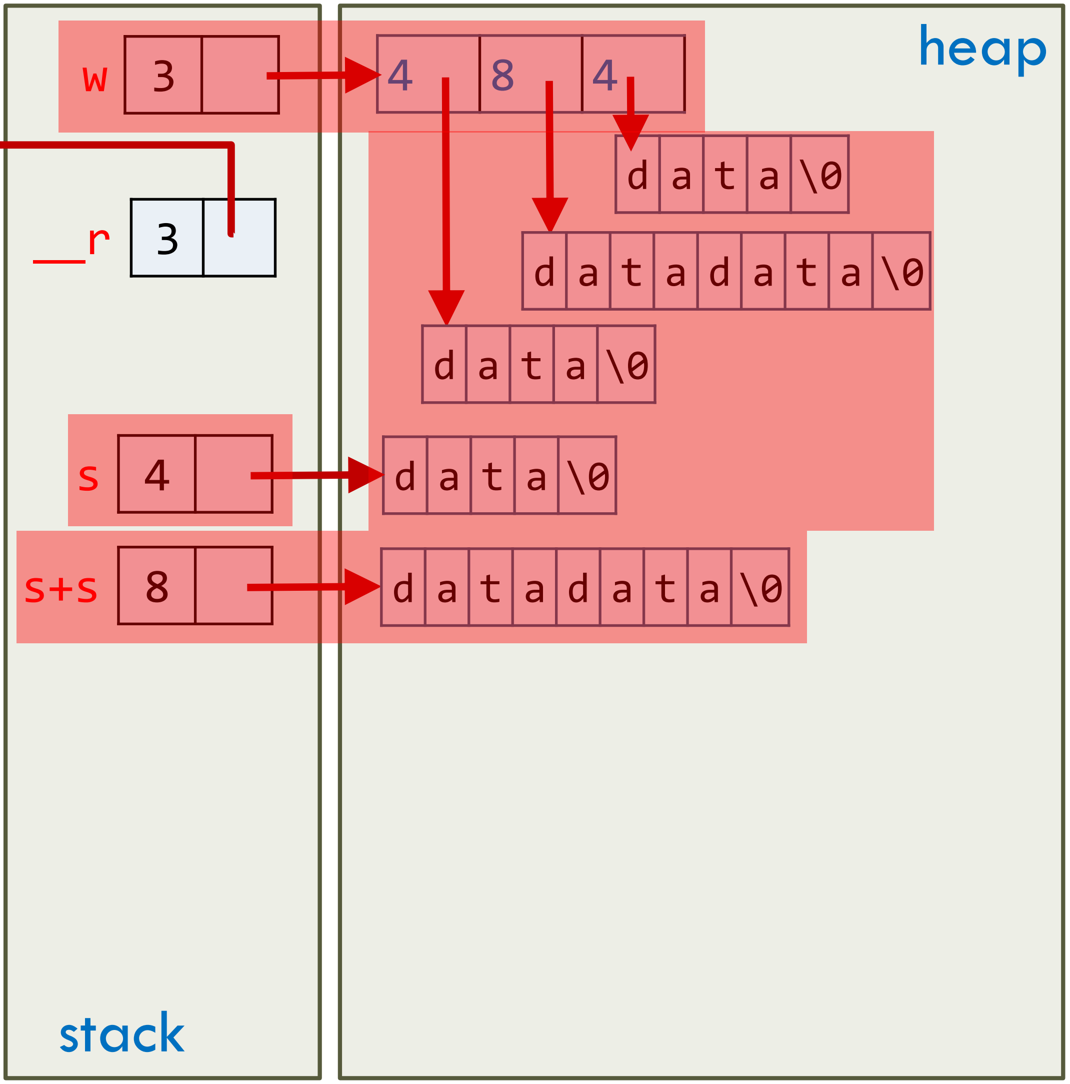
```
  return w;
}

std::vector<Str> v = f98();
```

Motivation for RVO

```
std::vector<Str> f98() {
```

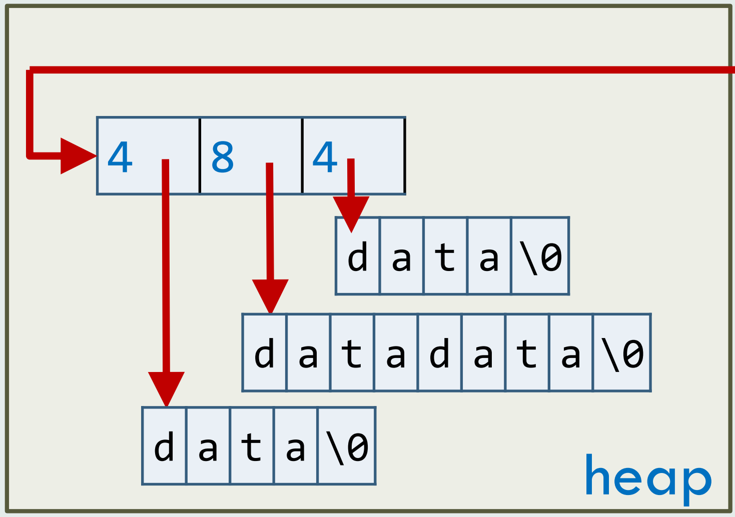


```
    return w;
}

std::vector<Str> v = f98();
```

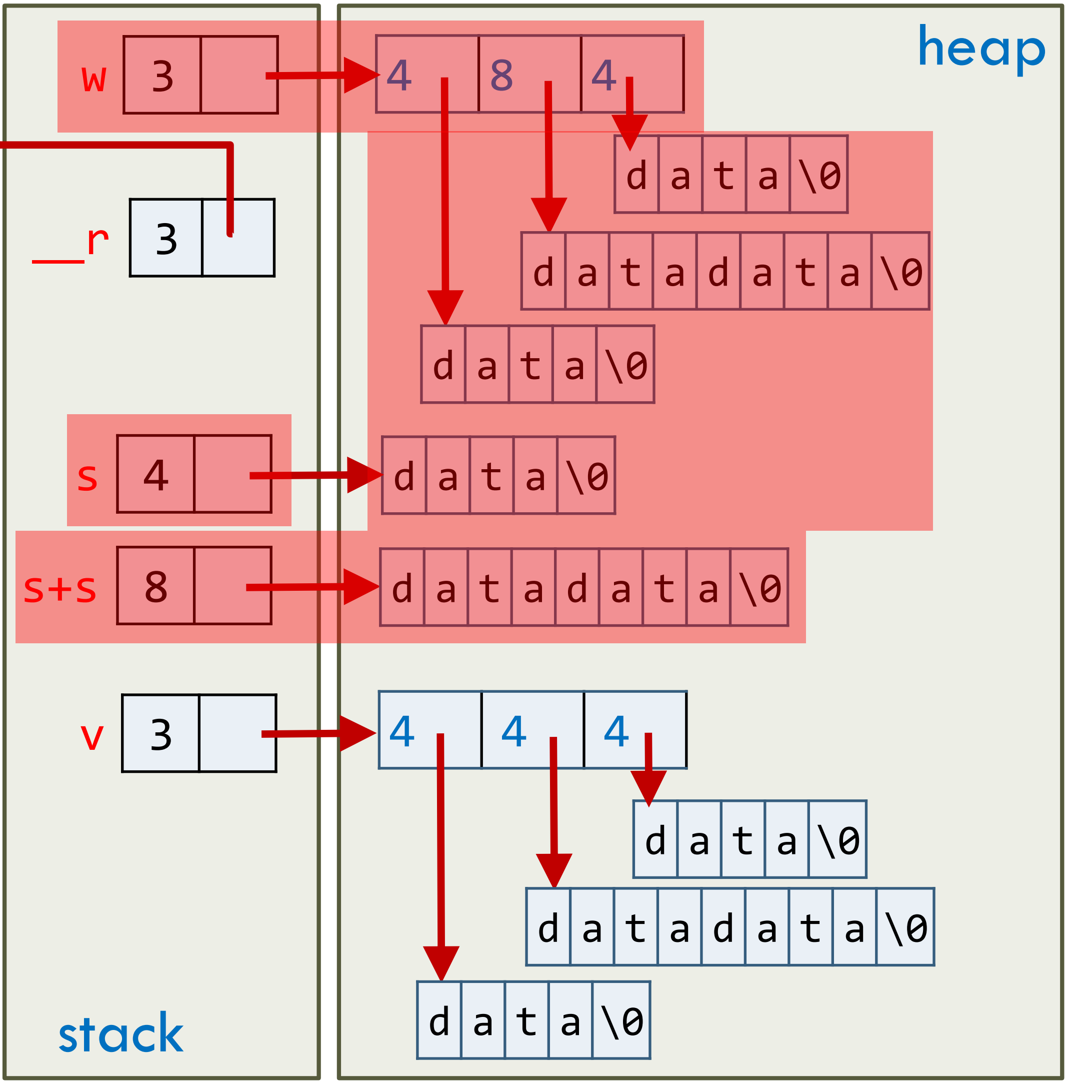


Motivation for RVO

```
std::vector<Str> f98() {
```

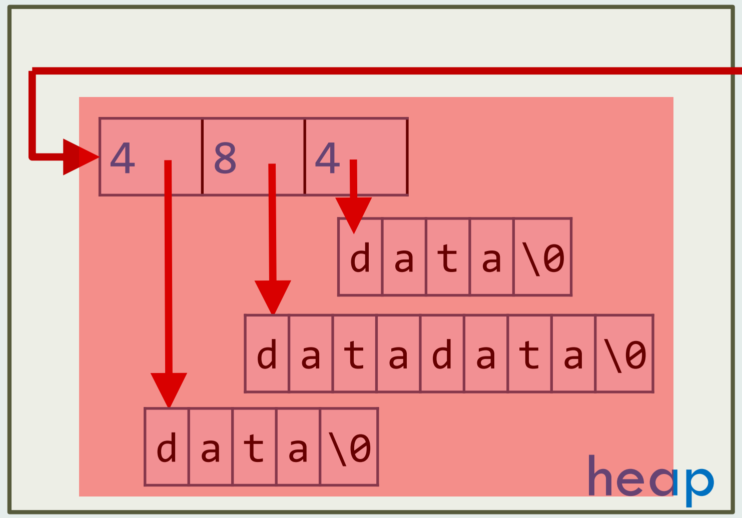


```
  return w;
}

std::vector<Str> v = f98();
```

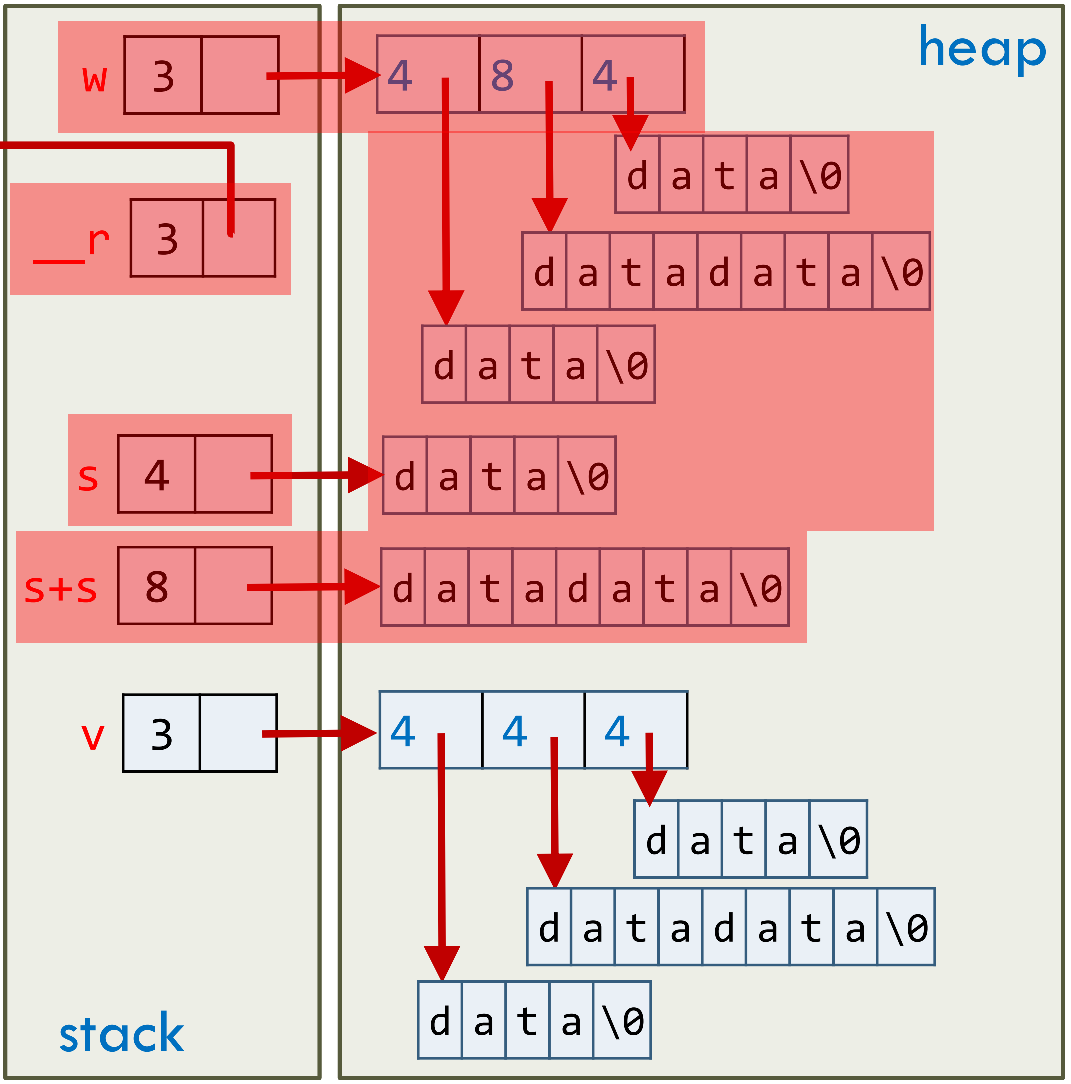


Motivation for RVO

```
std::vector<Str> f98() {
```

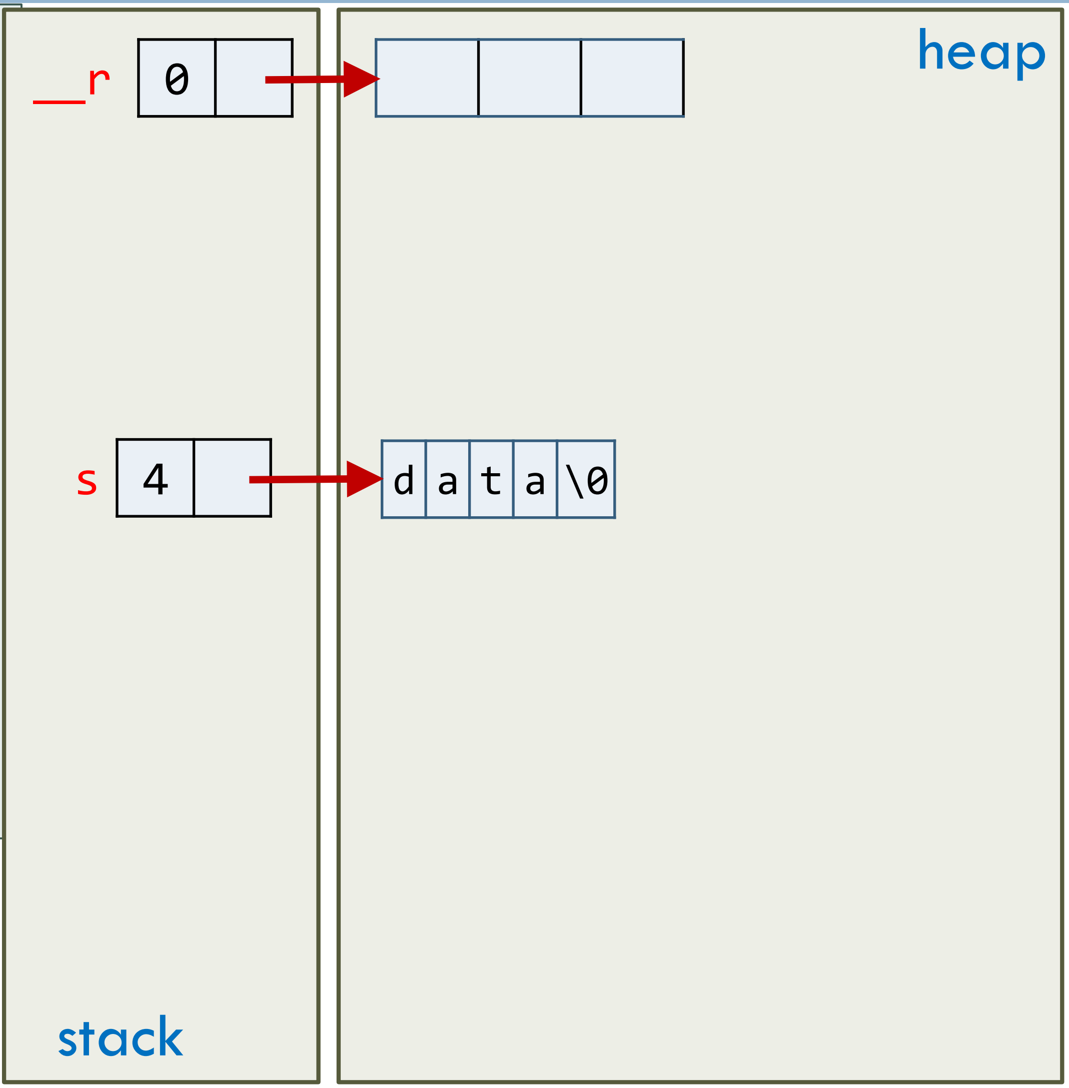


```
  return w;
}

std::vector<Str> v = f98();
```



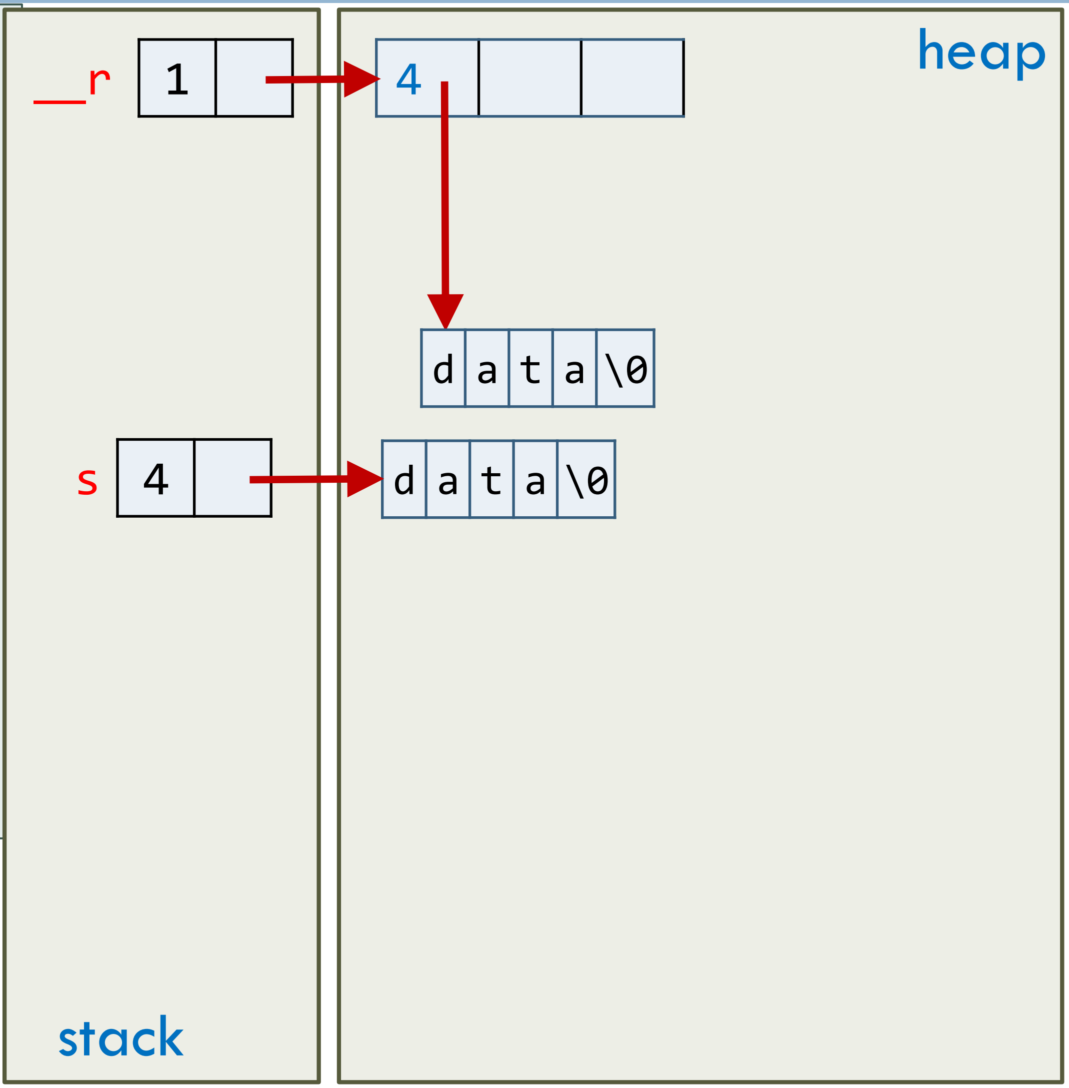
With RVO

```
void f98(vector<Str>& __r) {
  __r.vector<Str>();
  __r.reserve(3);
  Str s = "data";

  __r.push_back(s);
  __r.push_back(s+s);
  __r.push_back(s);

  return;
}

std::vector<Str> v; // no ctor
f98(v);
```



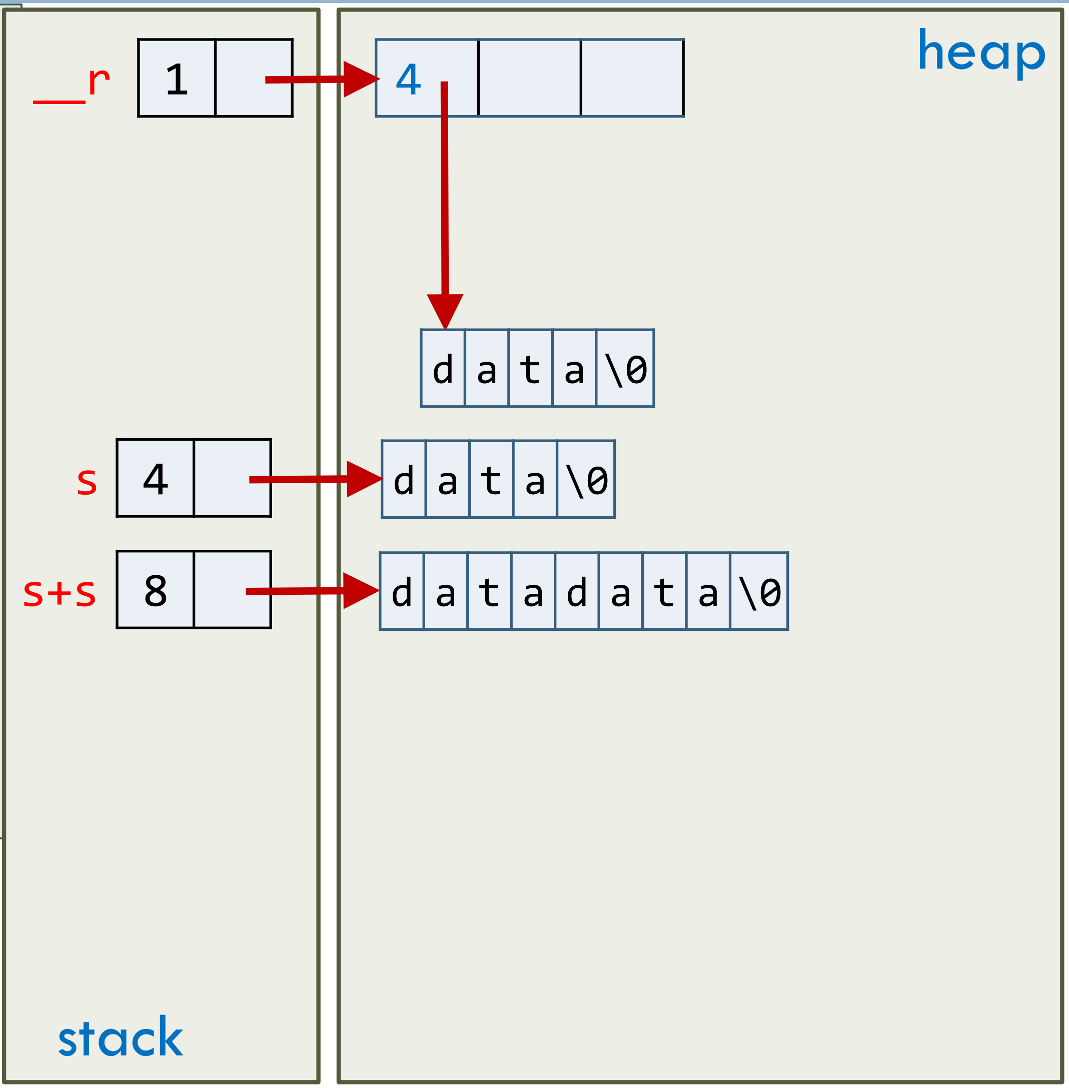
With RVO

```
void f98(vector<Str>& __r) {
   __r.vector<Str>();
   __r.reserve(3);
   Str s = "data";

   __r.push_back(s);
   __r.push_back(s+s);
   __r.push_back(s);

   return;
}

std::vector<Str> v; // no ctor
f98(v);
```



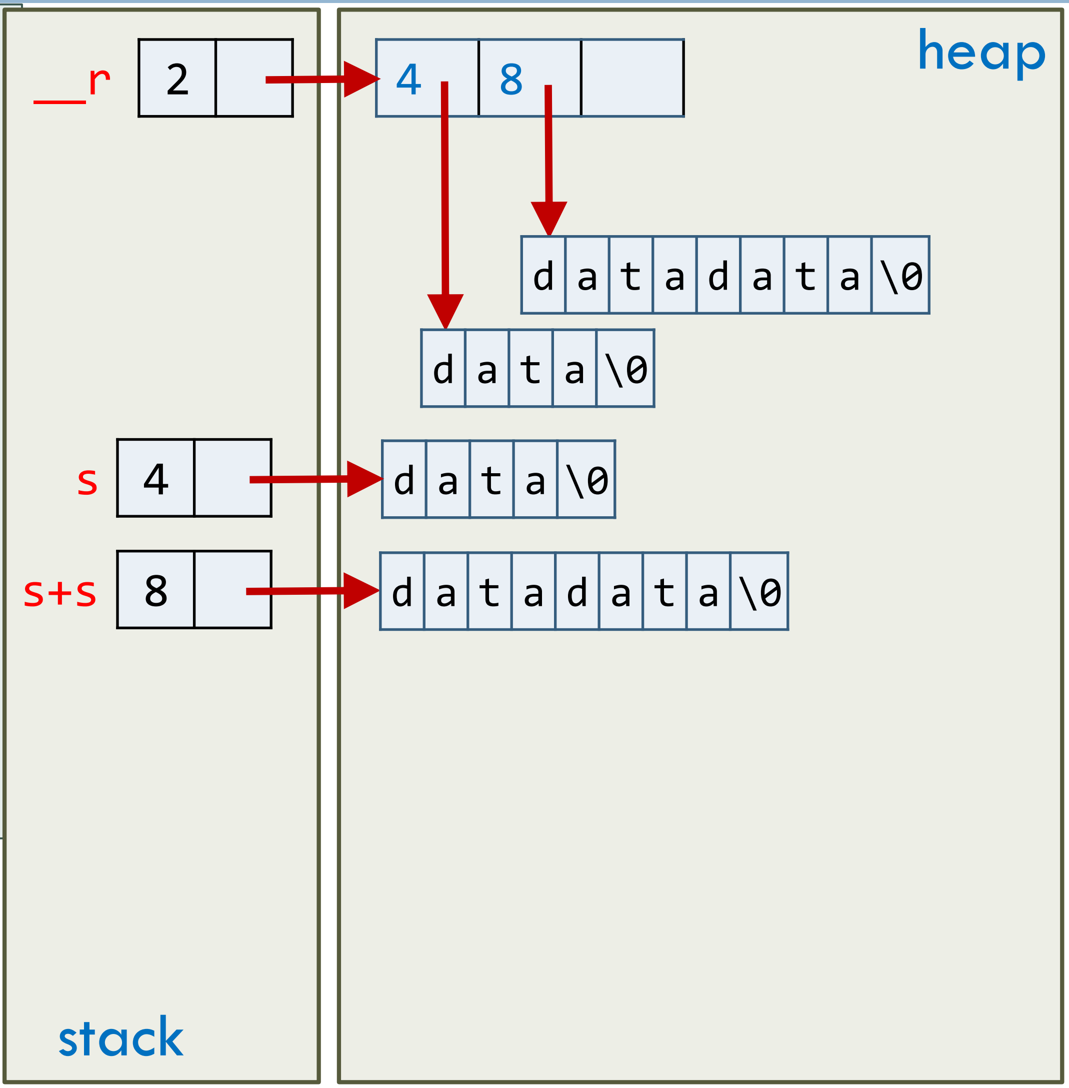
With RVO

```
void f98(vector<Str>& __r) {
   __r.vector<Str>();
   __r.reserve(3);
   Str s = "data";

   __r.push_back(s);
   __r.push_back(s+s);
   __r.push_back(s);

   return;
}

std::vector<Str> v; // no ctor
f98(v);
```



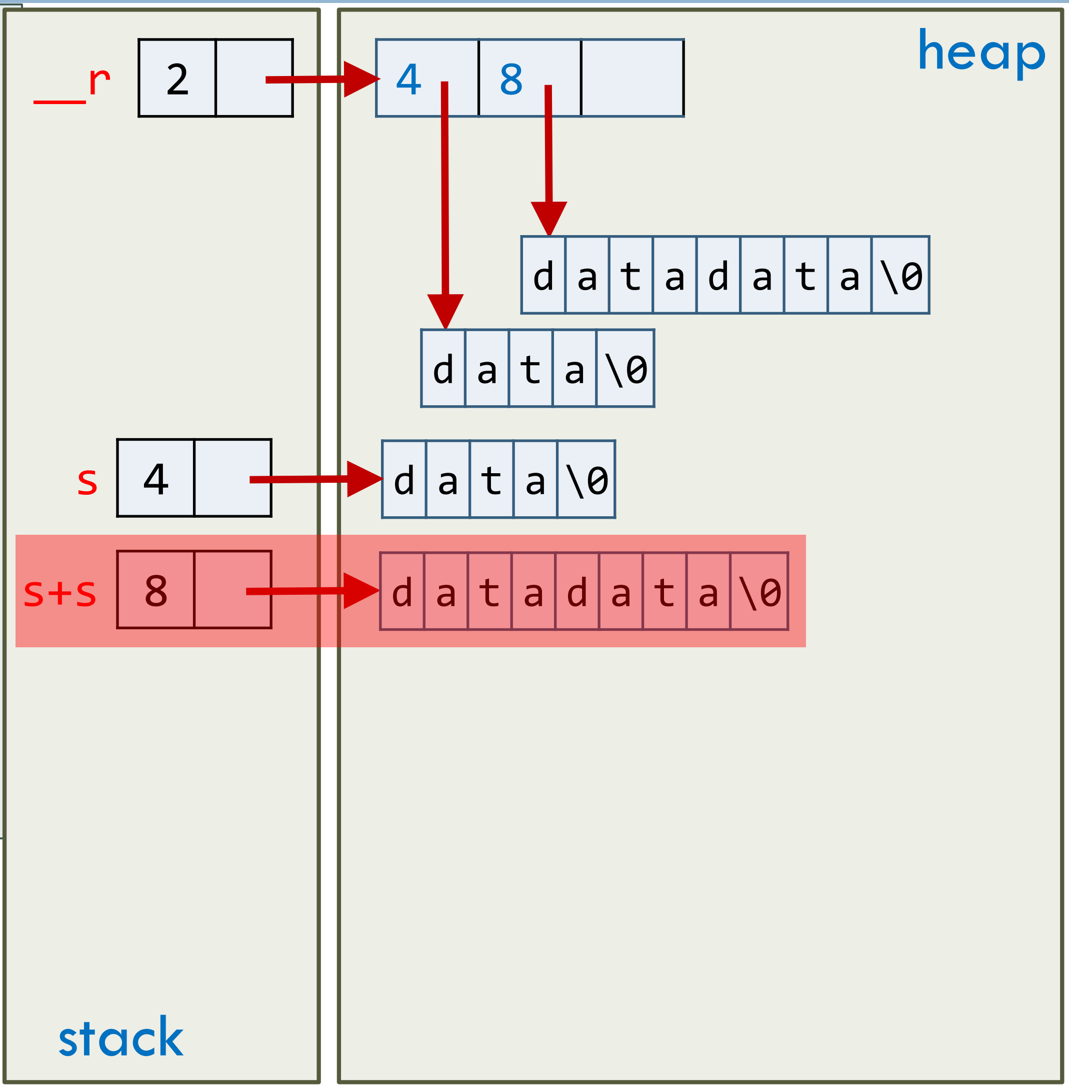
With RVO

```
void f98(vector<Str>& __r) {
  __r.vector<Str>();
  __r.reserve(3);
  Str s = "data";

  __r.push_back(s);
  __r.push_back(s+s);
  __r.push_back(s);

  return;
}

std::vector<Str> v; // no ctor
f98(v);
```



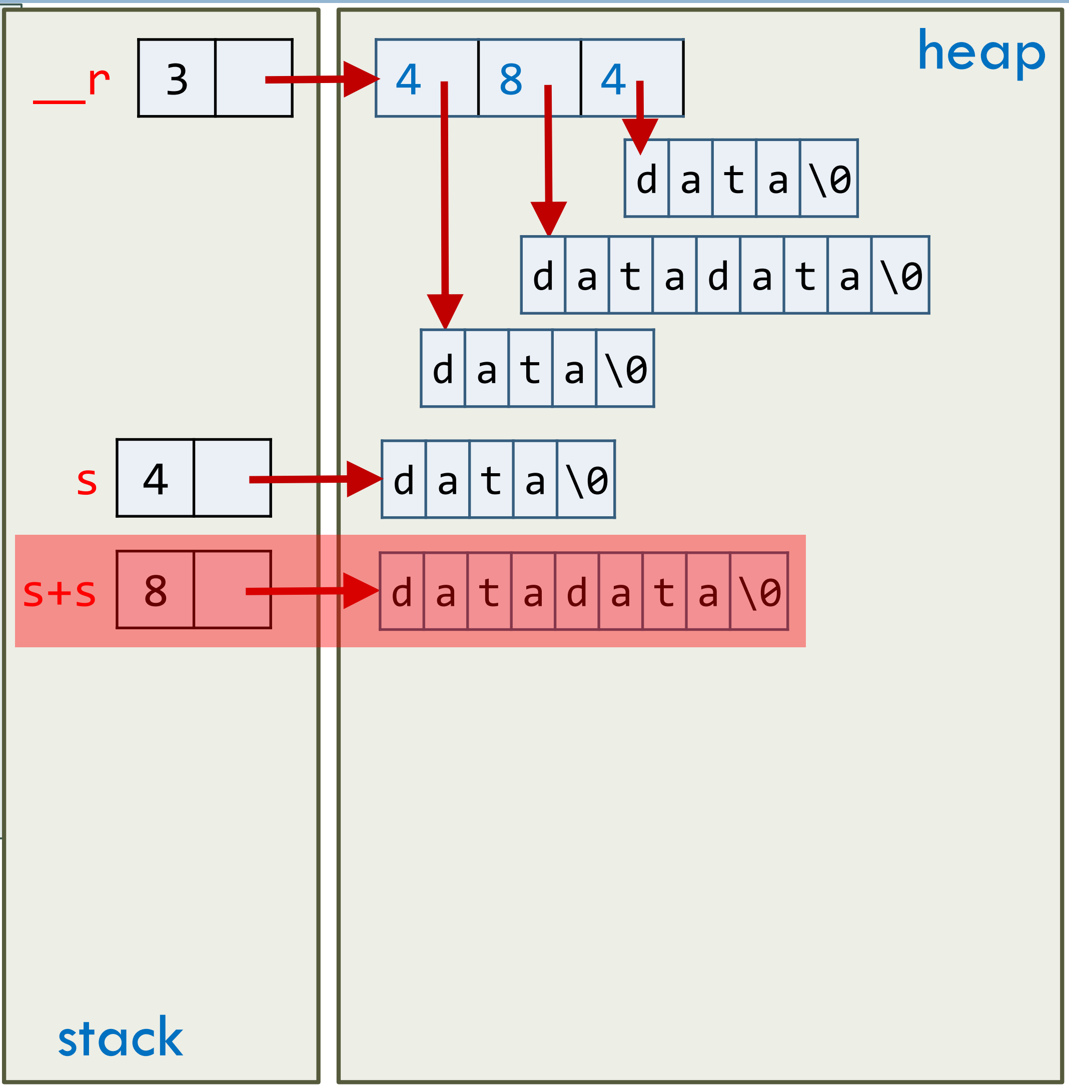
With RVO

```
void f98(vector<Str>& __r) {
  __r.vector<Str>();
  __r.reserve(3);
  Str s = "data";

  __r.push_back(s);
  __r.push_back(s+s);
  __r.push_back(s);

  return;
}

std::vector<Str> v; // no ctor
f98(v);
```



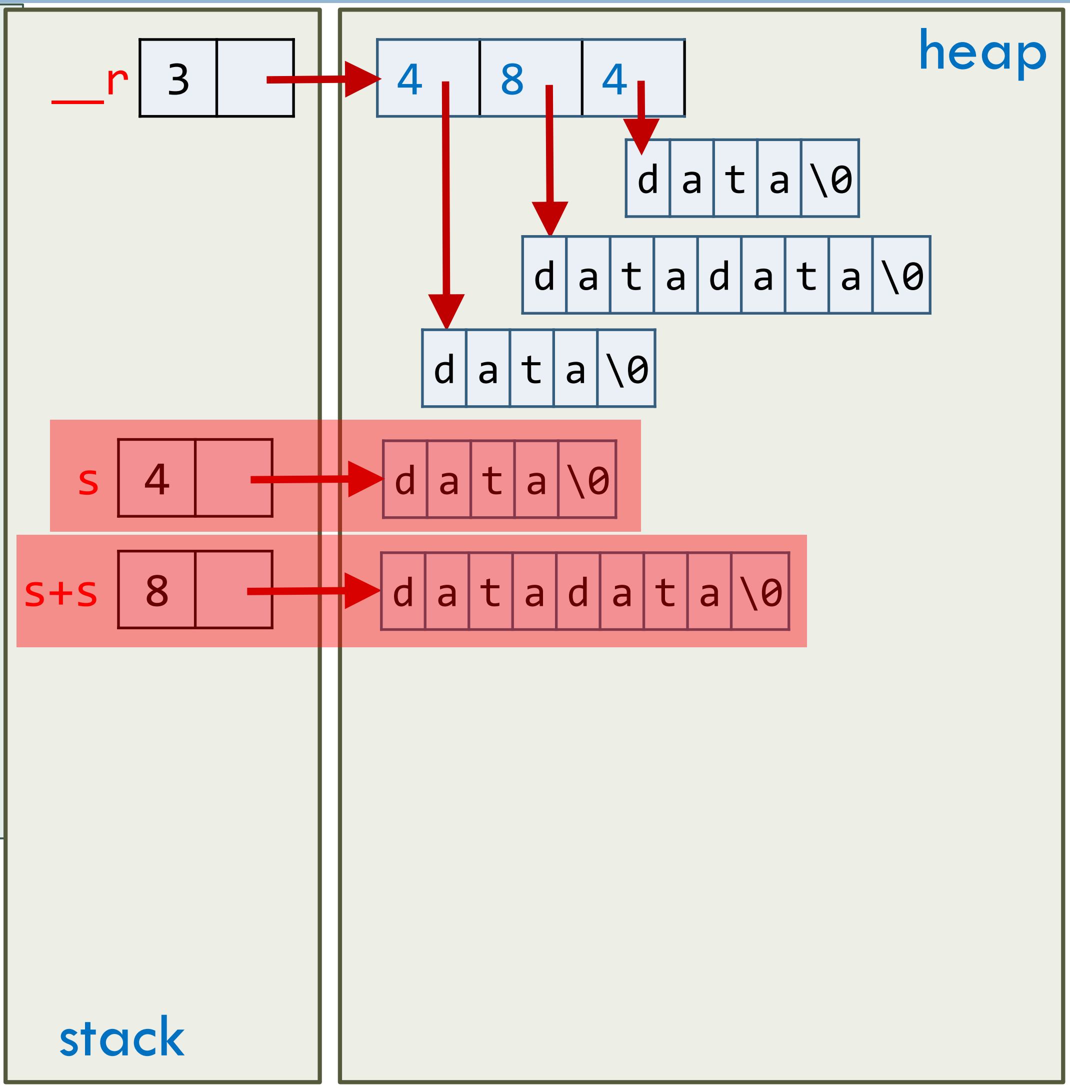
With RVO

```
void f98(vector<Str>& __r) {
  __r.vector<Str>();
  __r.reserve(3);
  Str s = "data";

  __r.push_back(s);
  __r.push_back(s+s);
  __r.push_back(s);

  return;
}

std::vector<Str> v; // no ctor
f98(v);
```



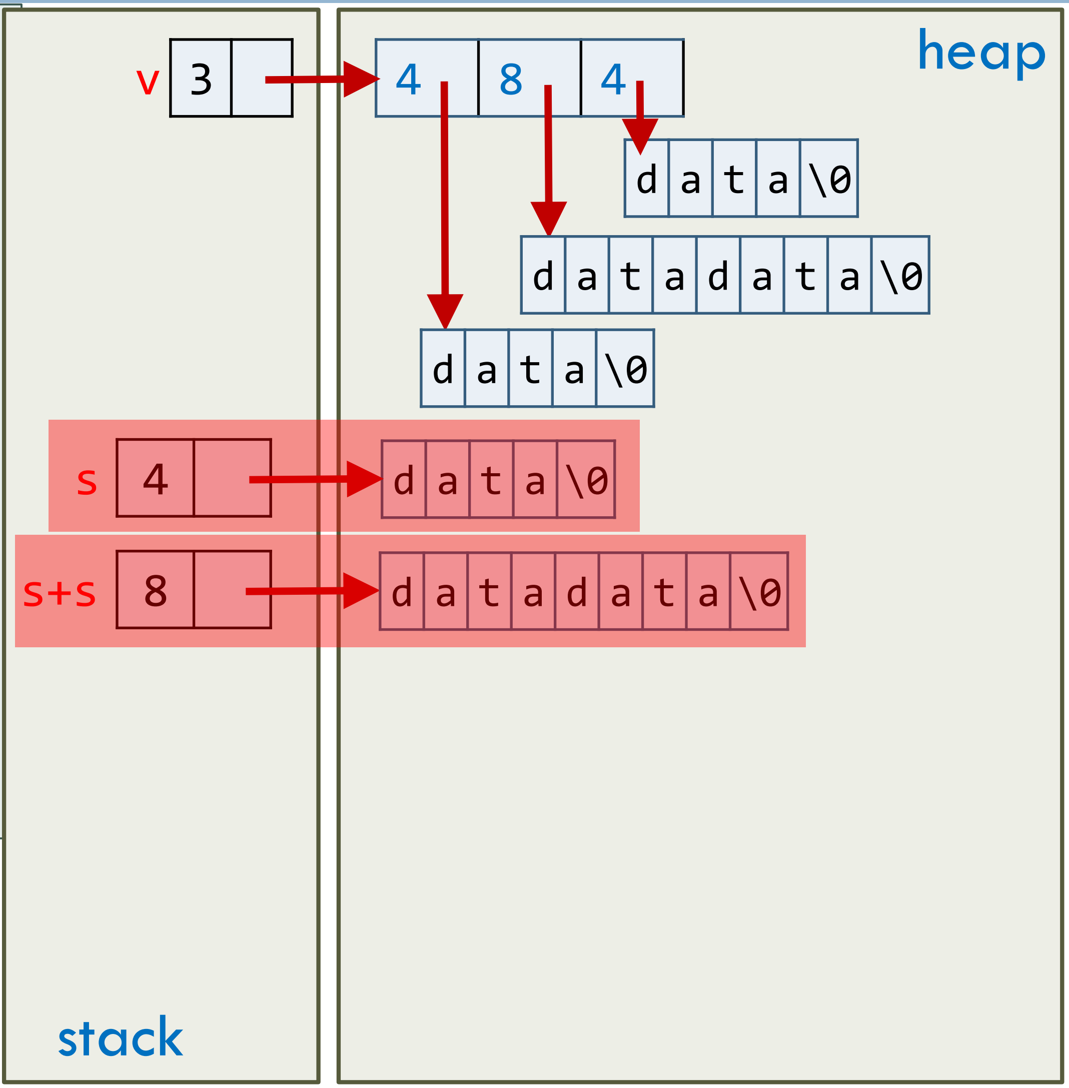
With RVO

```
void f98(vector<Str>& __r) {
   __r.vector<Str>();
   __r.reserve(3);
   Str s = "data";

   __r.push_back(s);
   __r.push_back(s+s);
   __r.push_back(s);

   return;
}

std::vector<Str> v; // no ctor
f98(v);
```



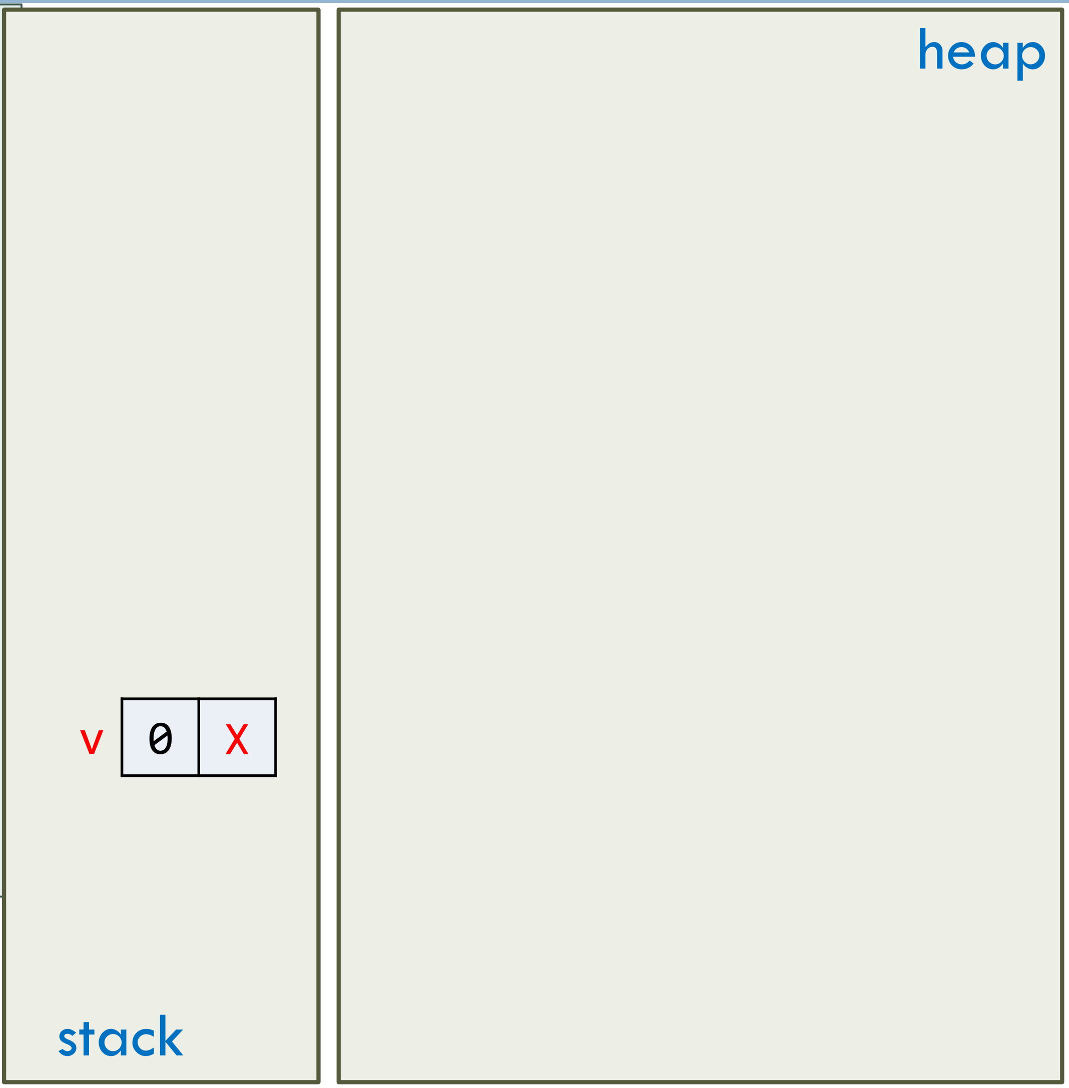
With RVO

```
void f98(vector<Str>& __r) {
  __r.vector<Str>();
  __r.reserve(3);
  Str s = "data";

  __r.push_back(s);
  __r.push_back(s+s);
  __r.push_back(s);

  return;
}

std::vector<Str> v; // no ctor
f98(v);
```



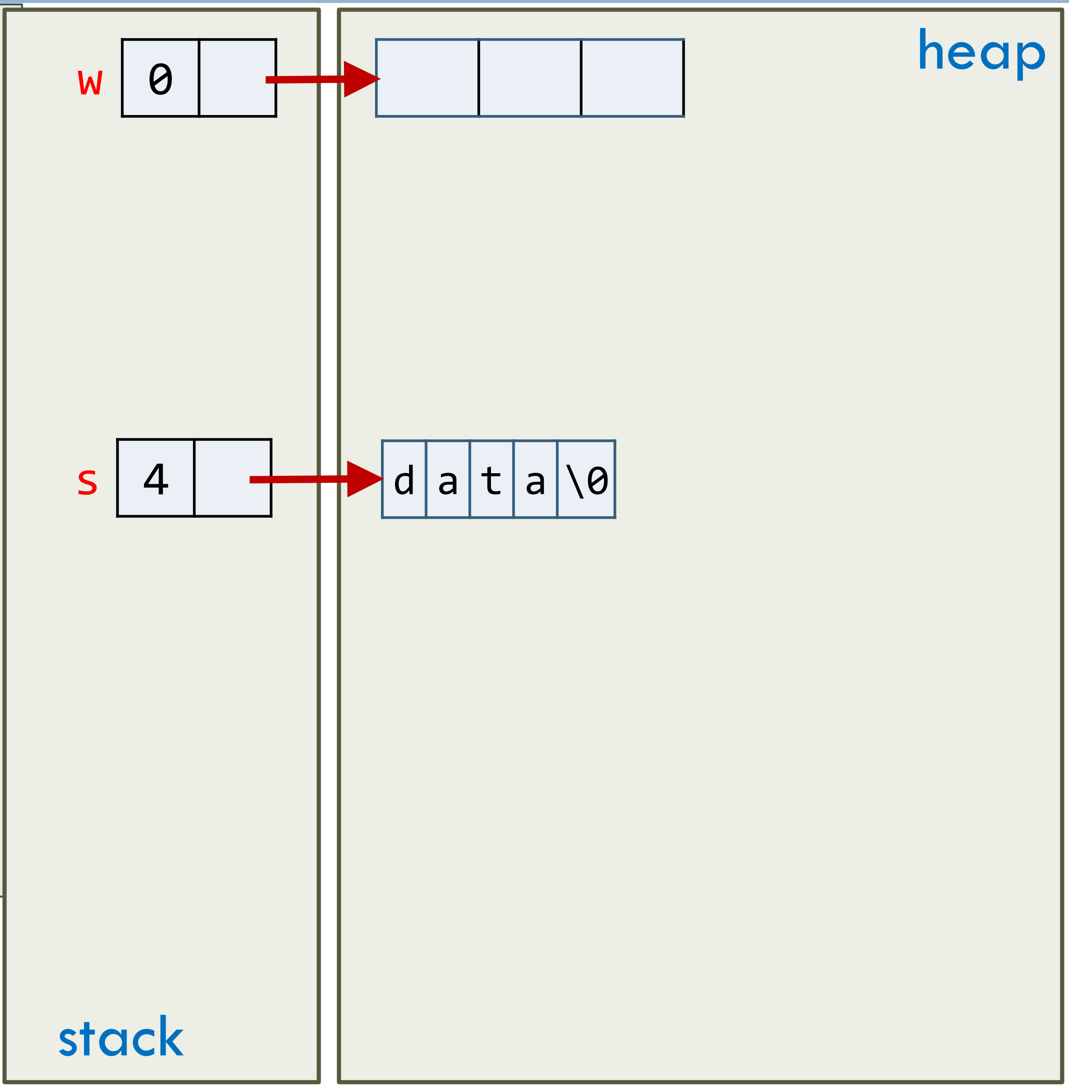
Motivation for Move Semantics

```
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();
```



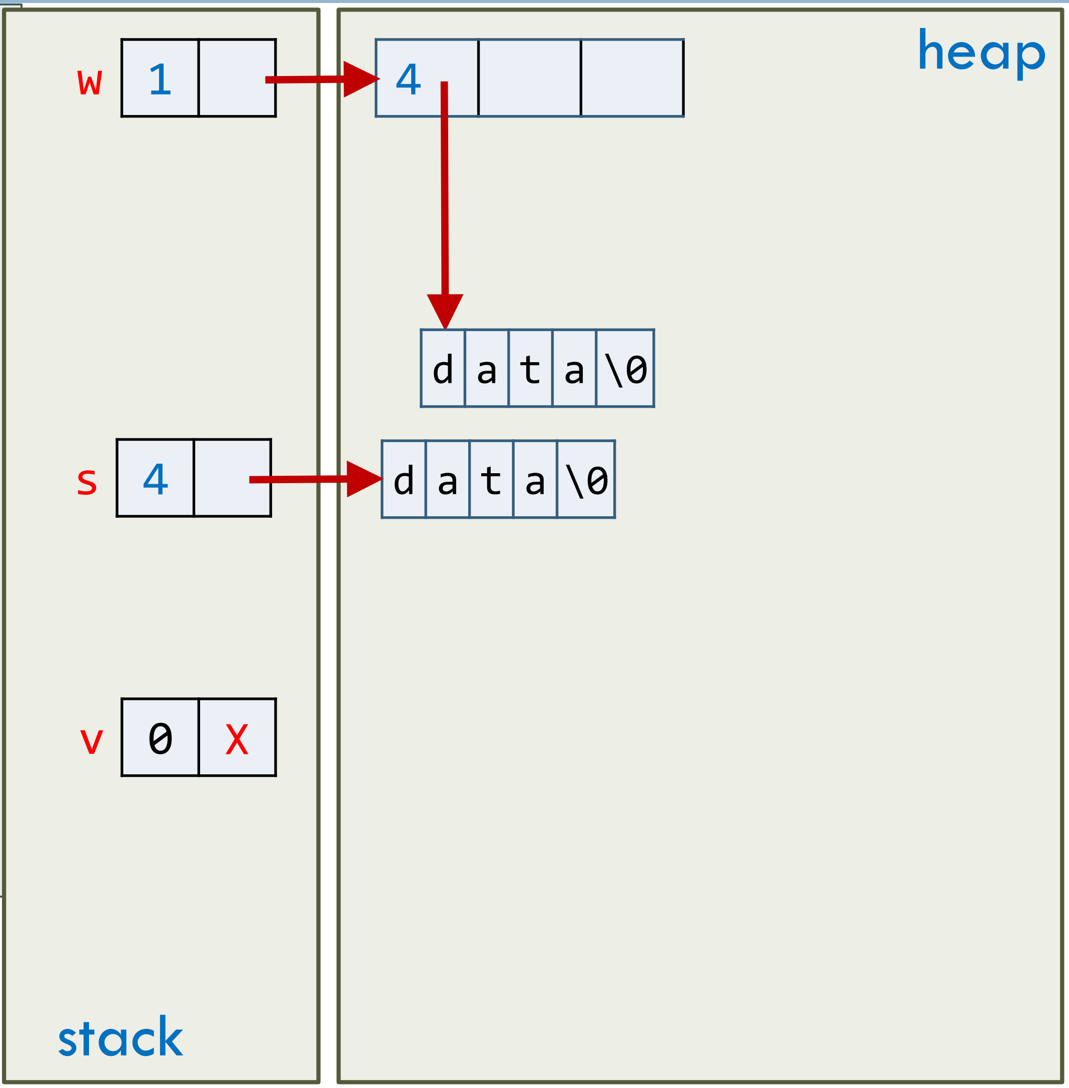
Motivation for Move Semantics

```
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();
```



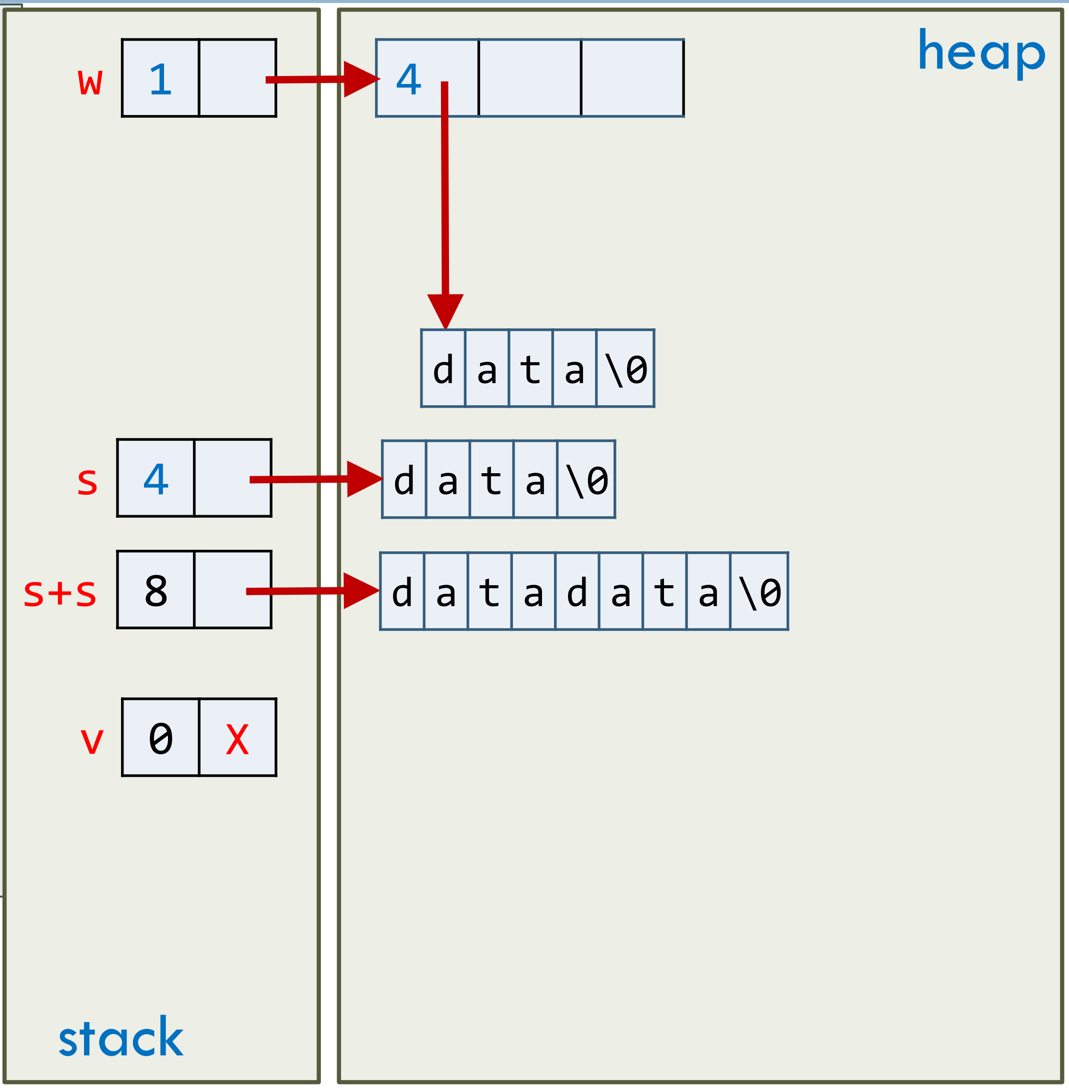
Motivation for Move Semantics

```
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();
```



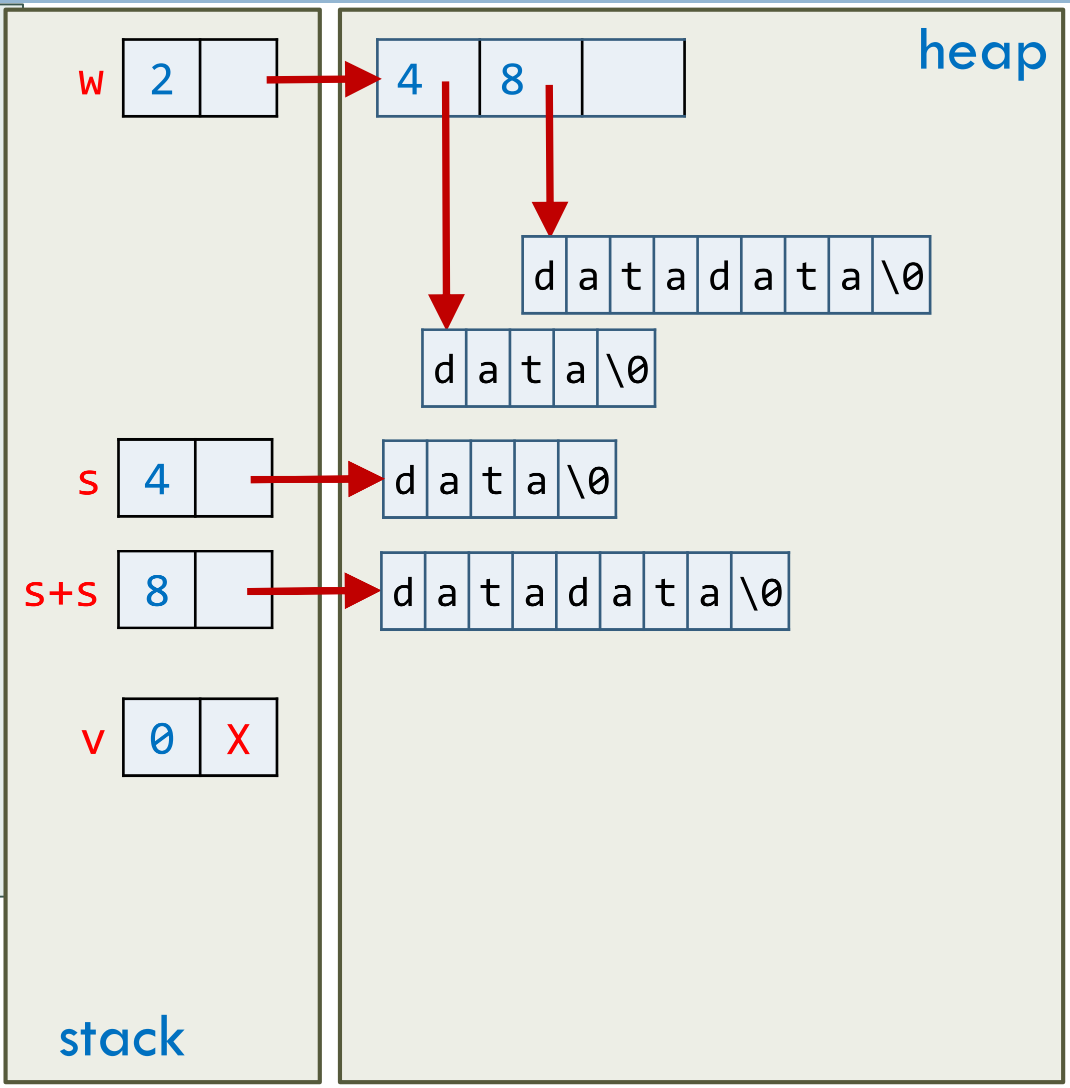
Motivation for Move Semantics

```
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();
```



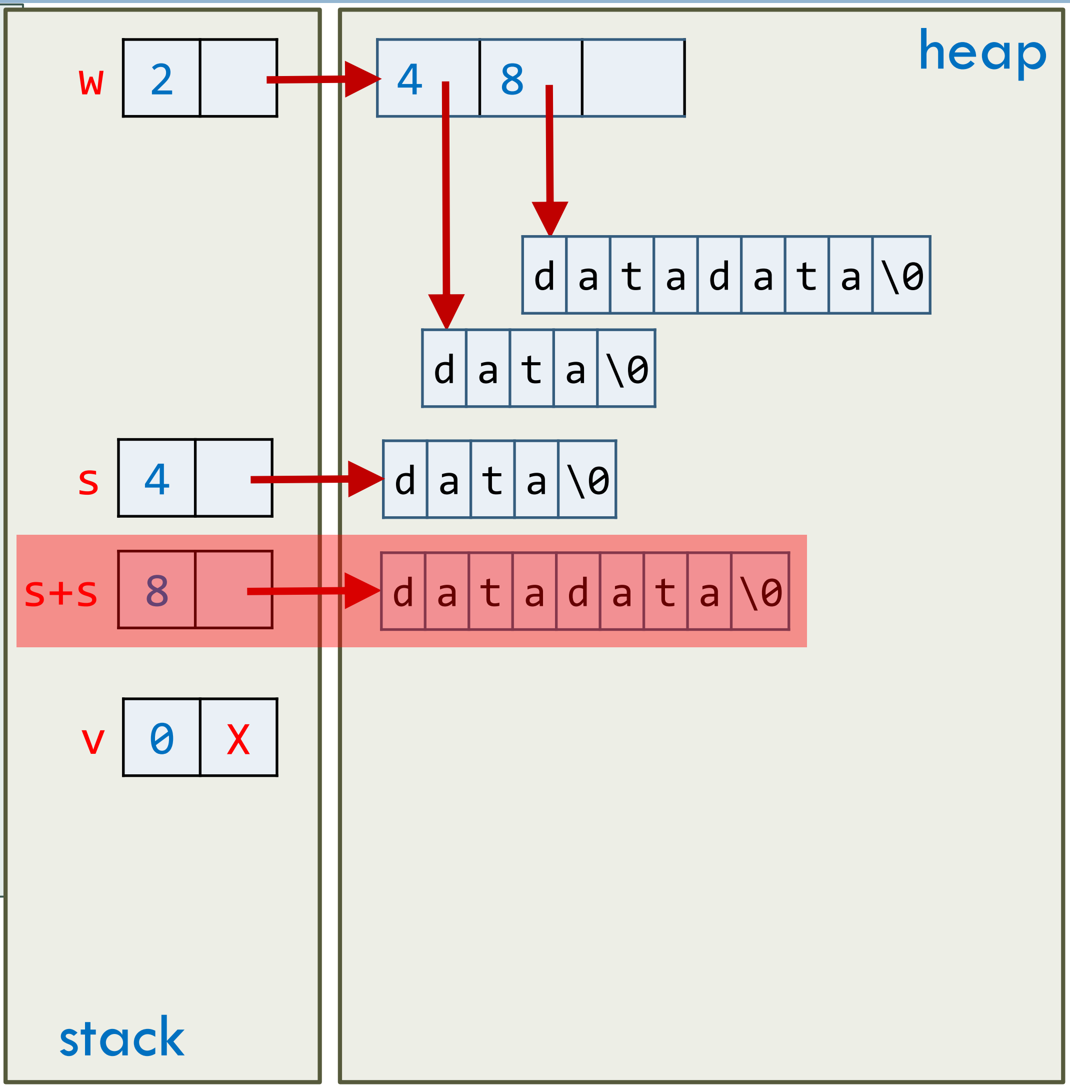
Motivation for Move Semantics

```
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();
```



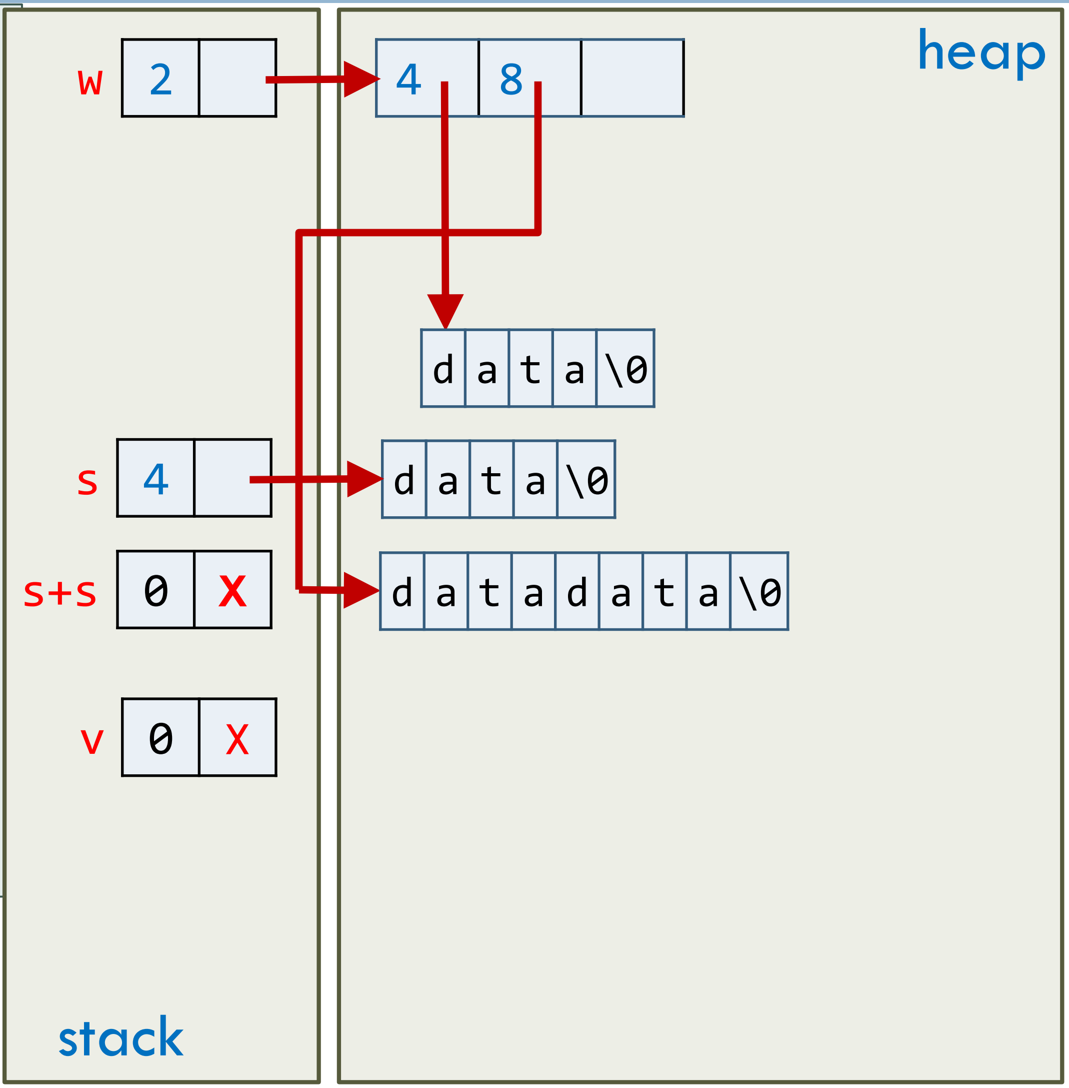
Motivation for Move Semantics

```
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();
```



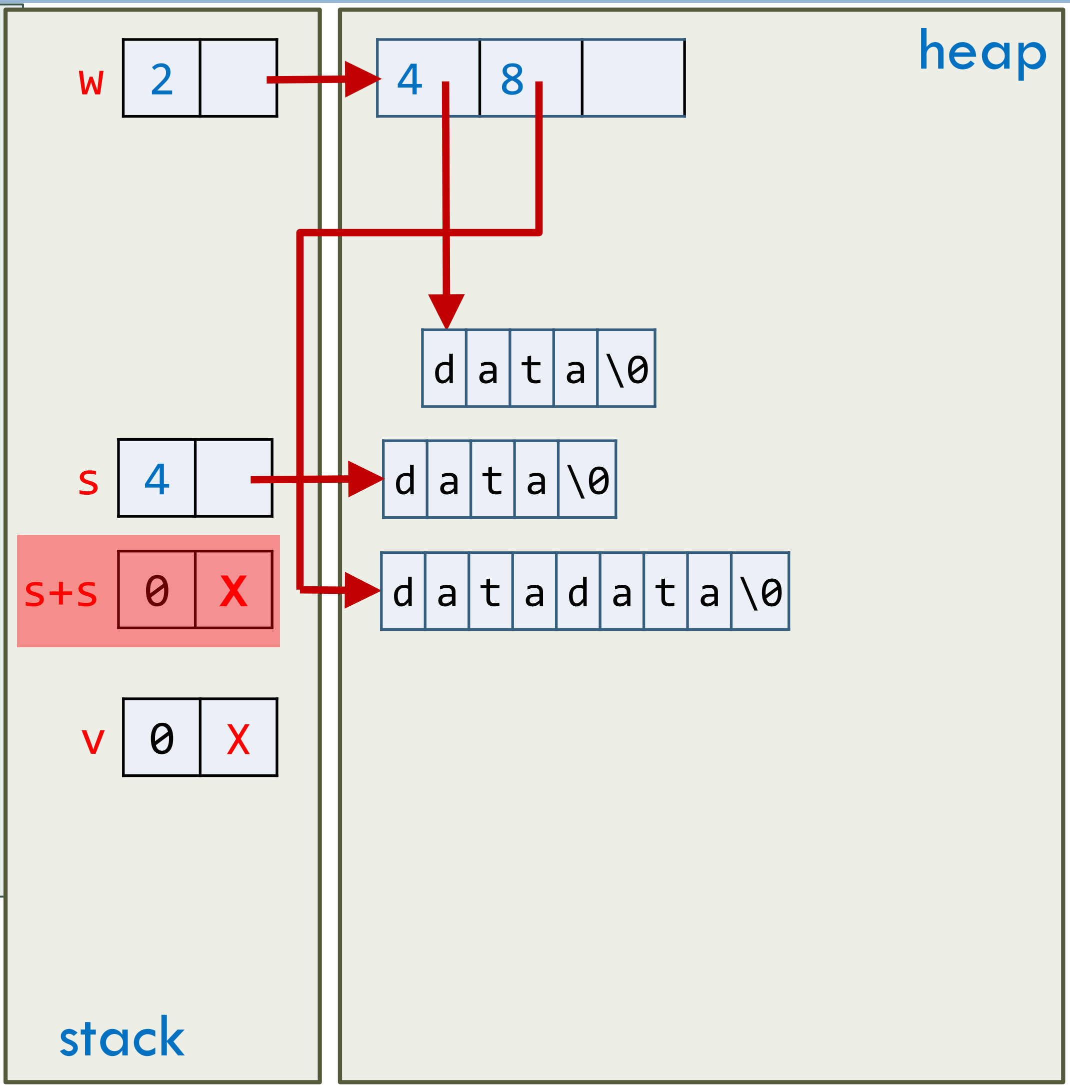
Motivation for Move Semantics

```
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();
```



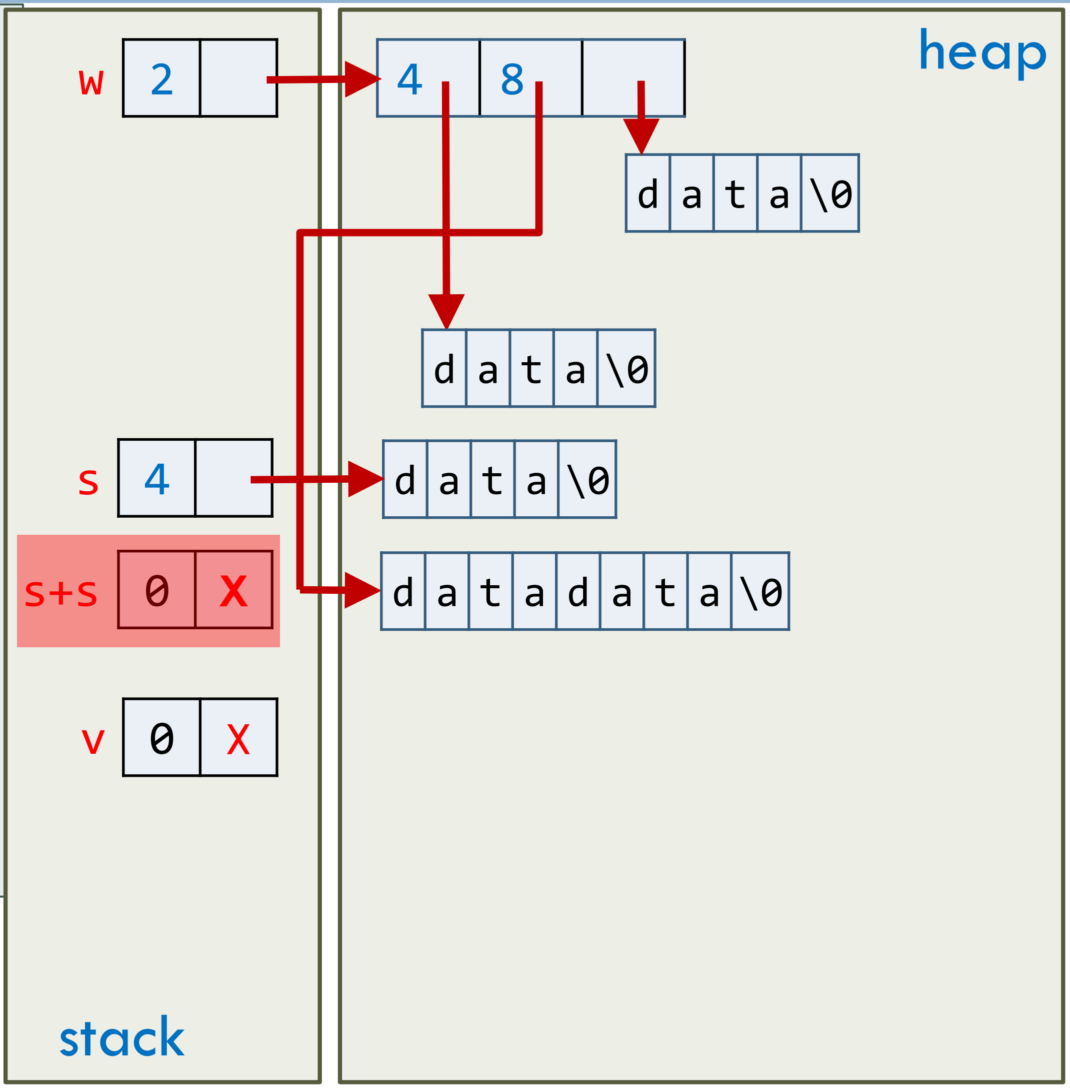
Motivation for Move Semantics

```
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();
```



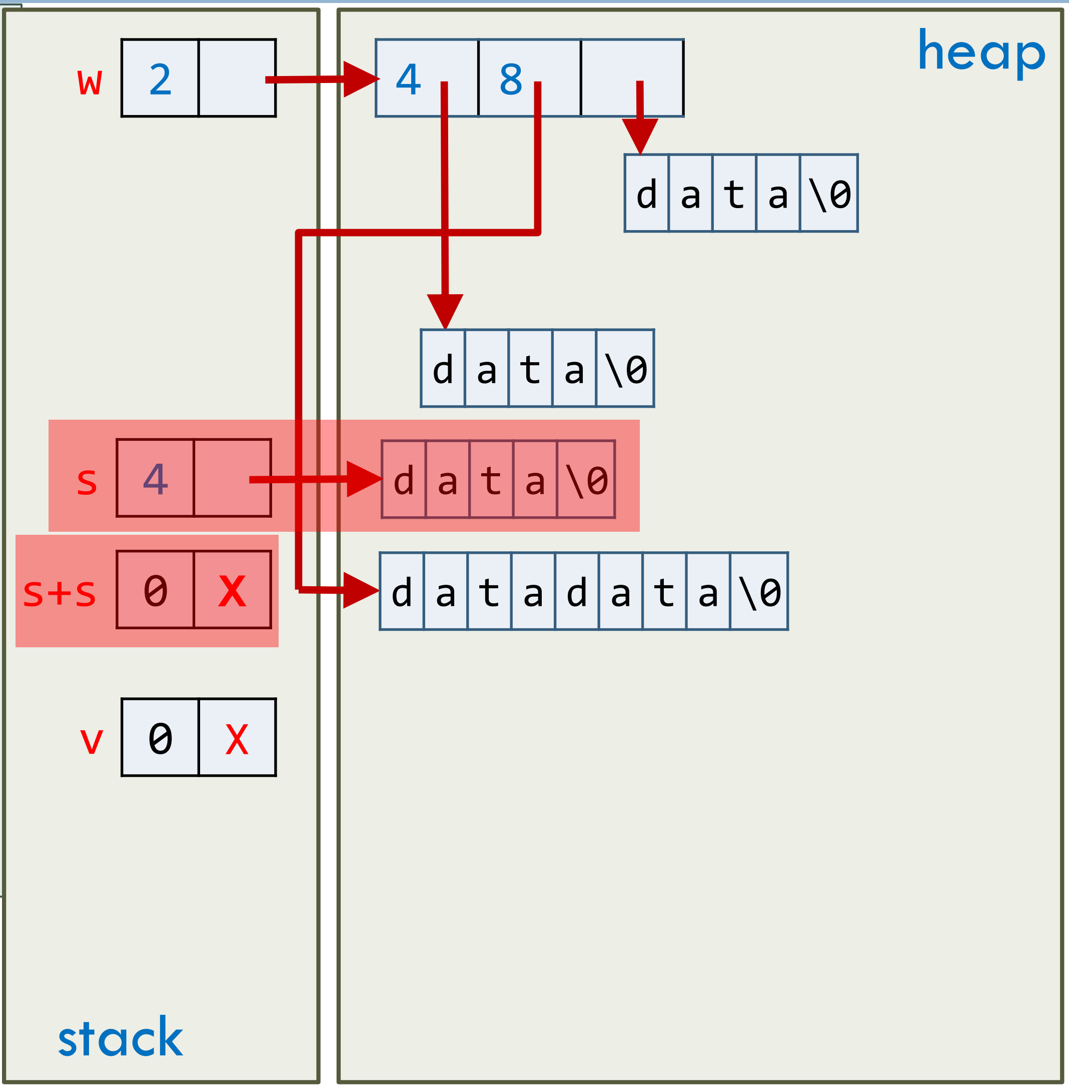
Motivation for Move Semantics

```
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();
```



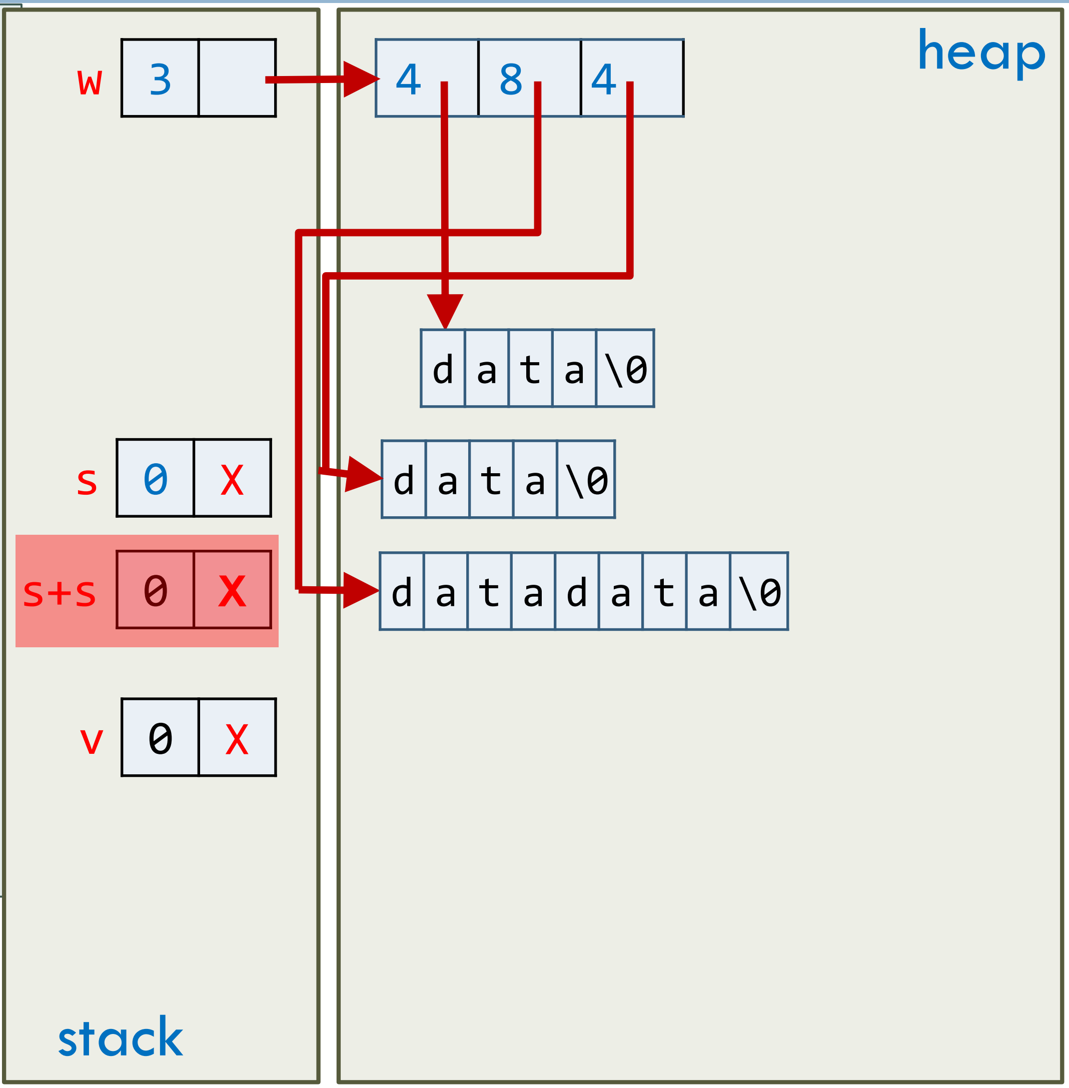
Motivation for Move Semantics

```
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();
```



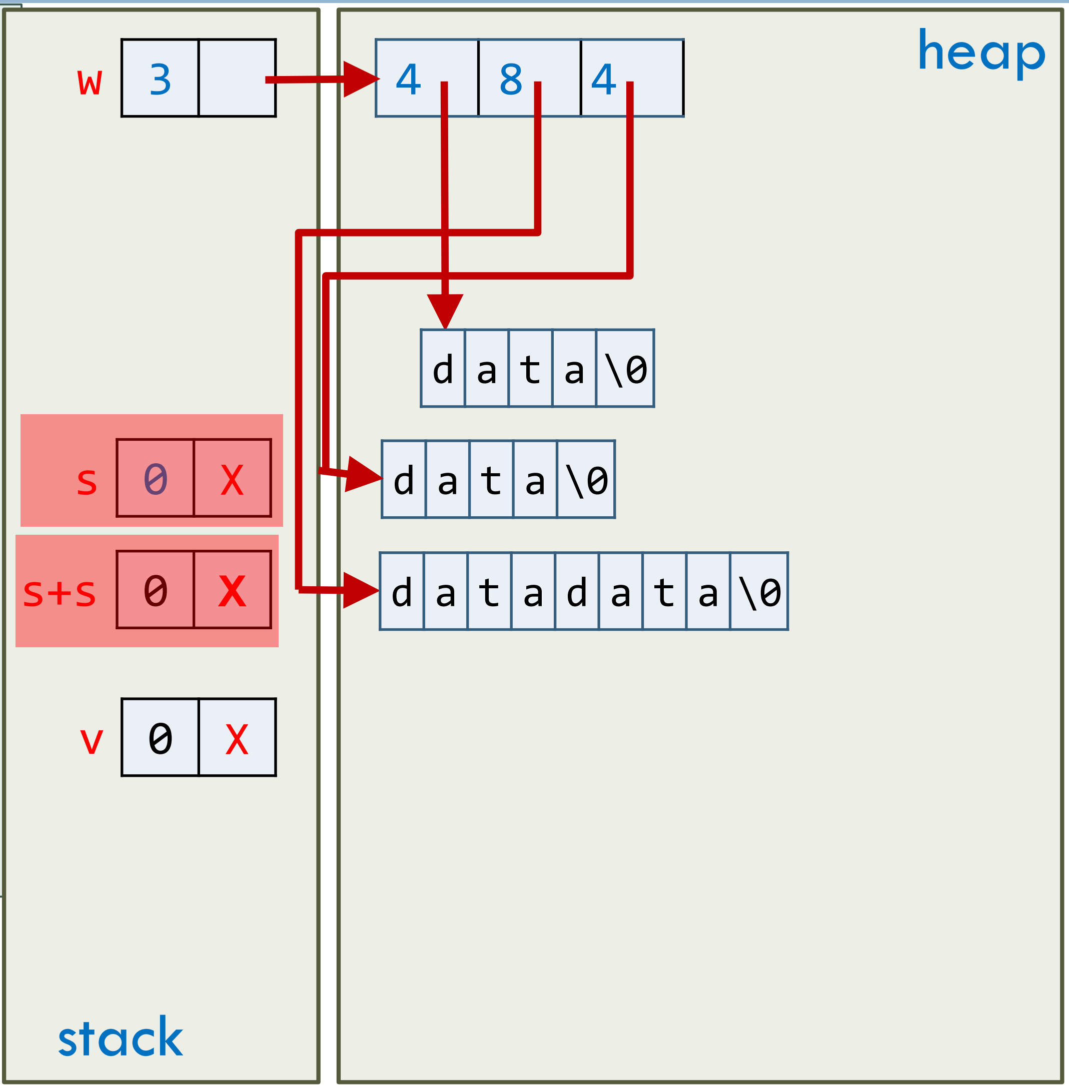
Motivation for Move Semantics

```
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();
```



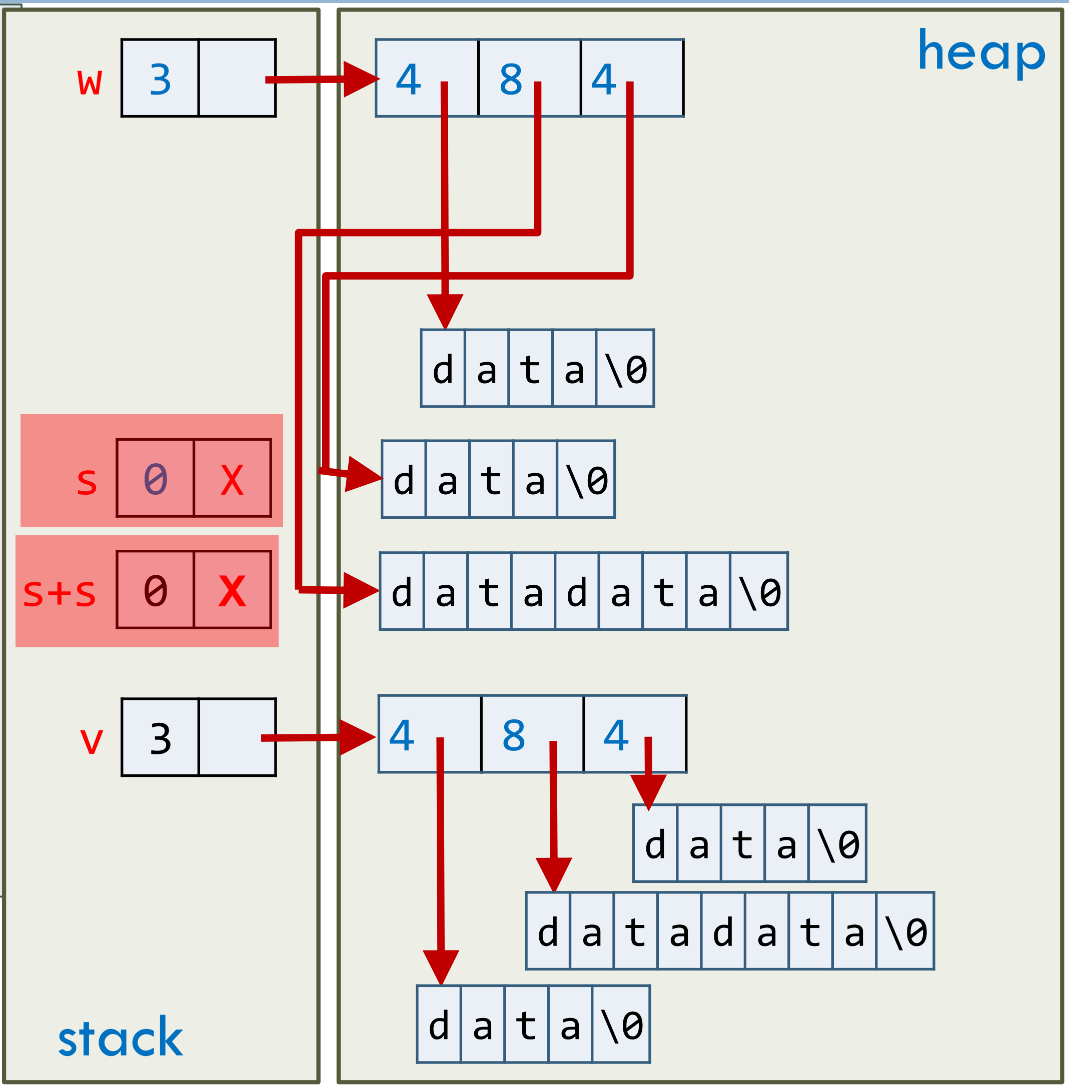
Motivation for Move Semantics

```
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();
```



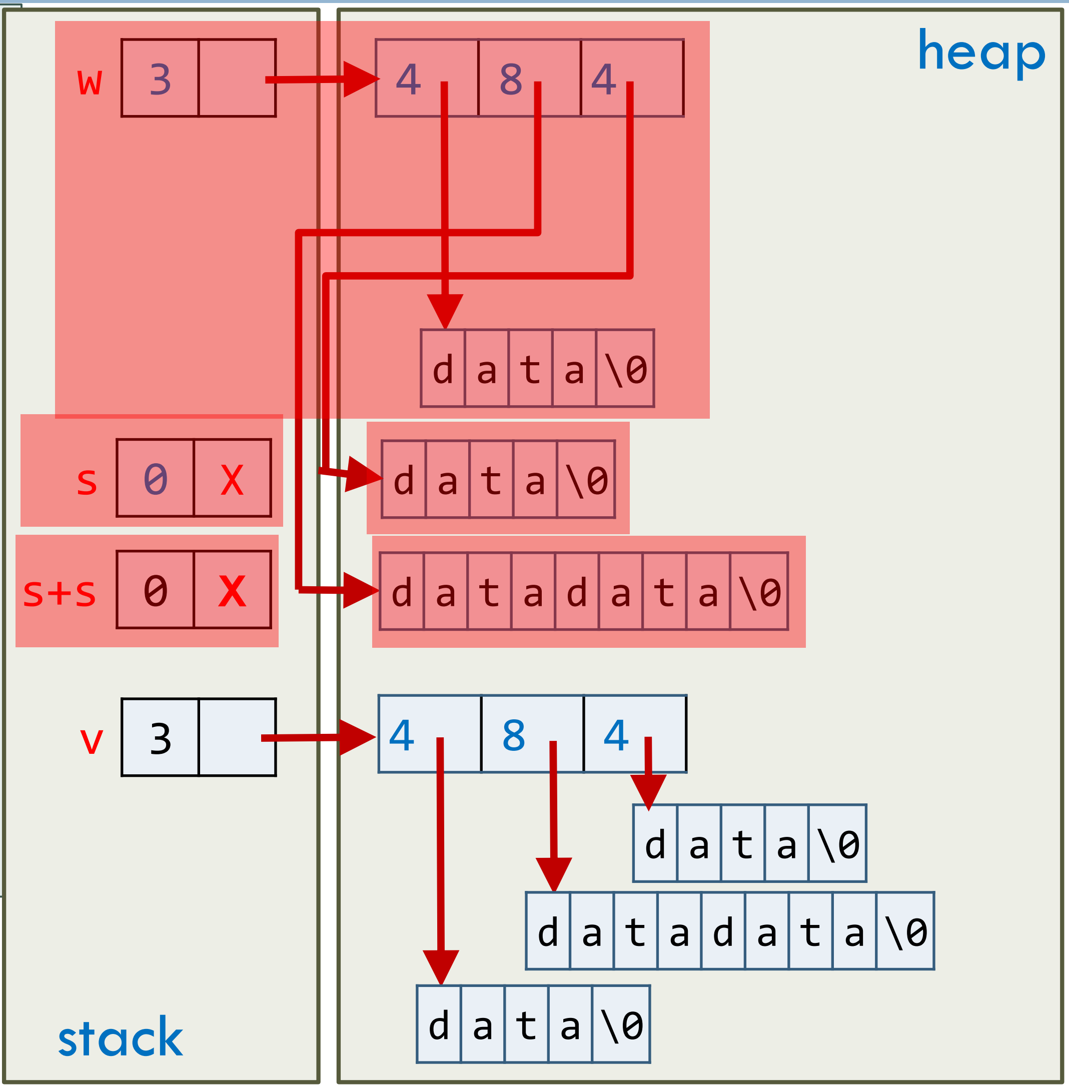
Motivation for Move Semantics

```
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();
```



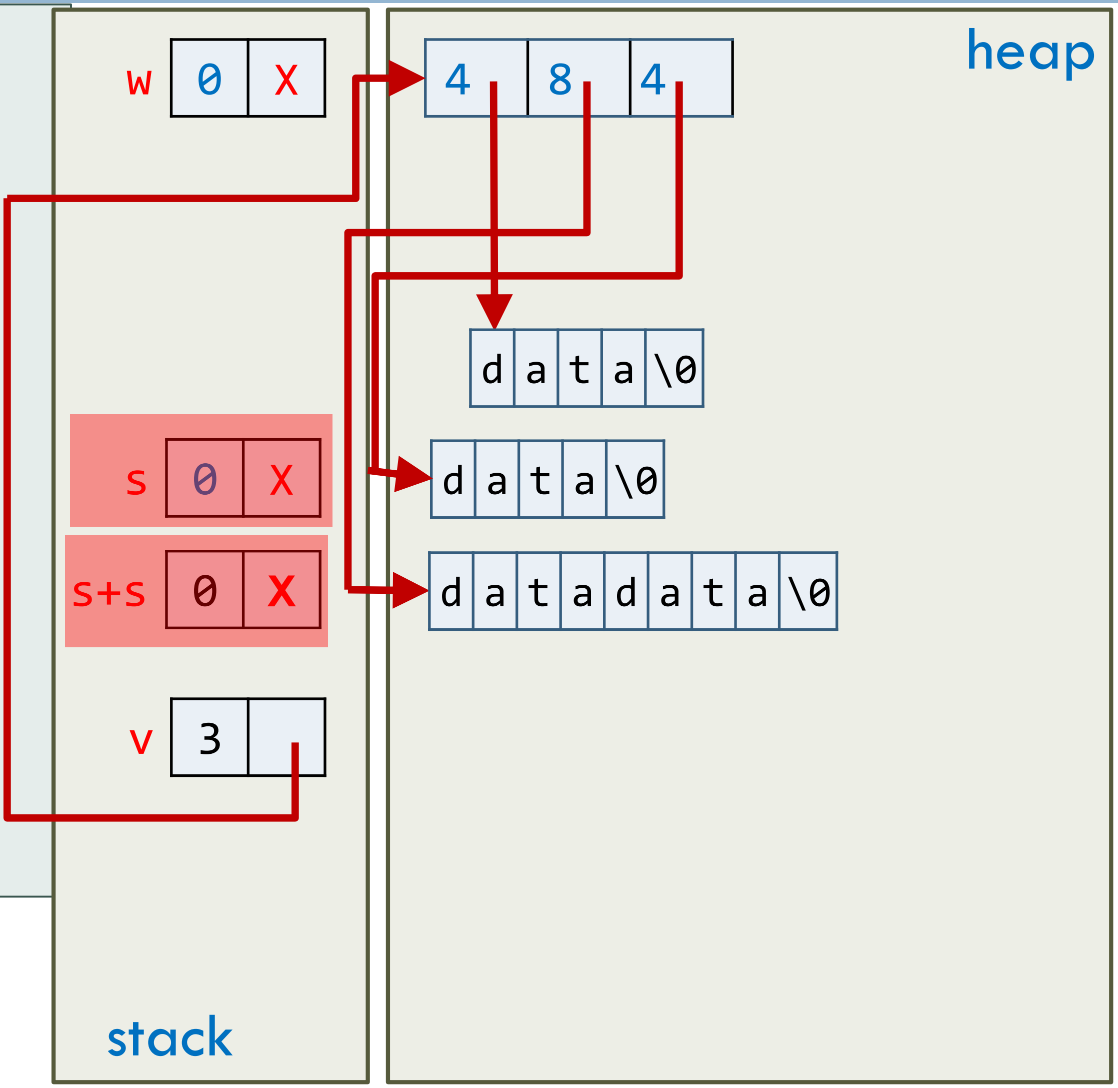
Motivation for Move Semantics

```
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();
```



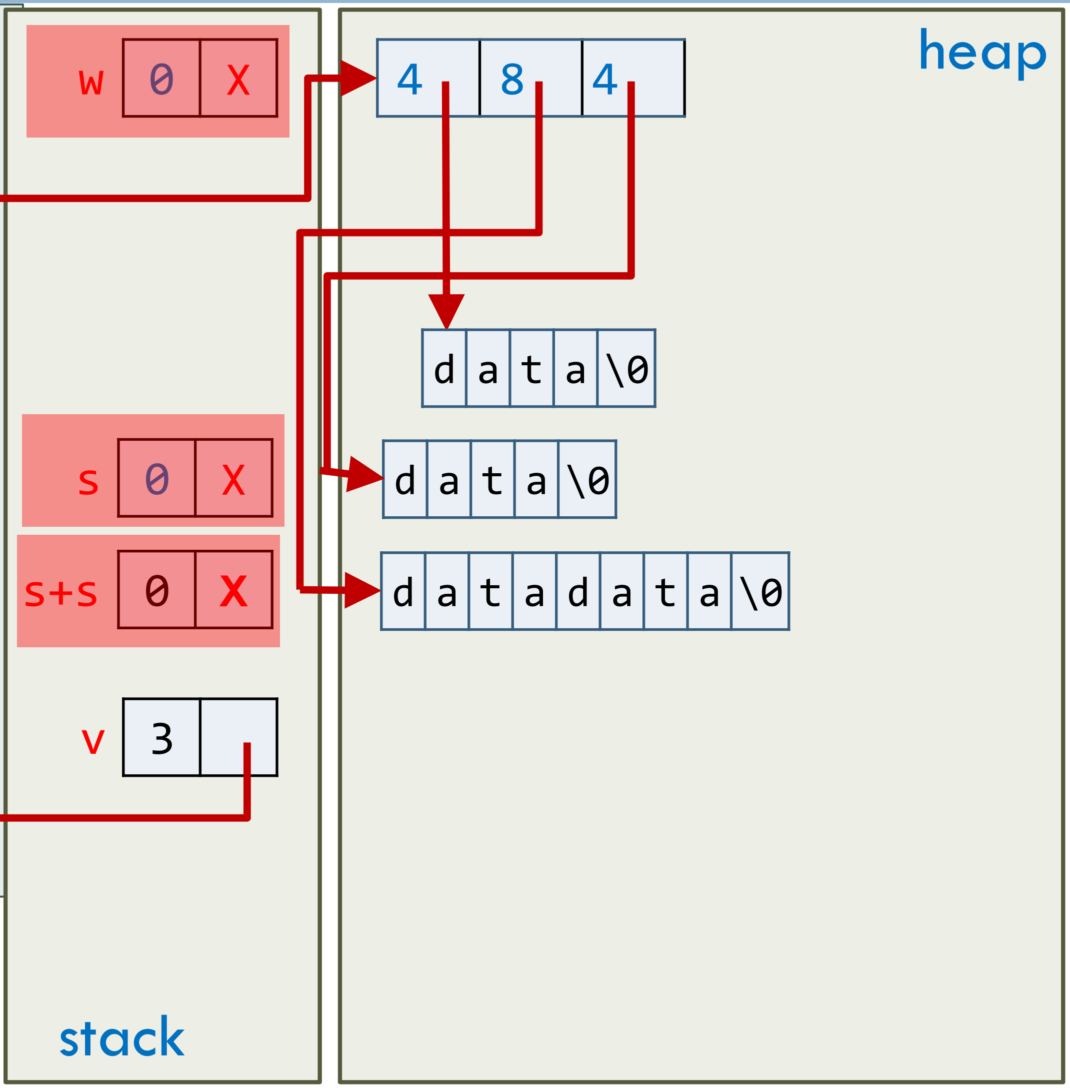
Motivation for Move Semantics

```
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();
```



Next Lecture(s)

- Rvalue references
- Move constructors
- Move assignments
- std::move
- std::swap
- std::forward
- Smart pointers