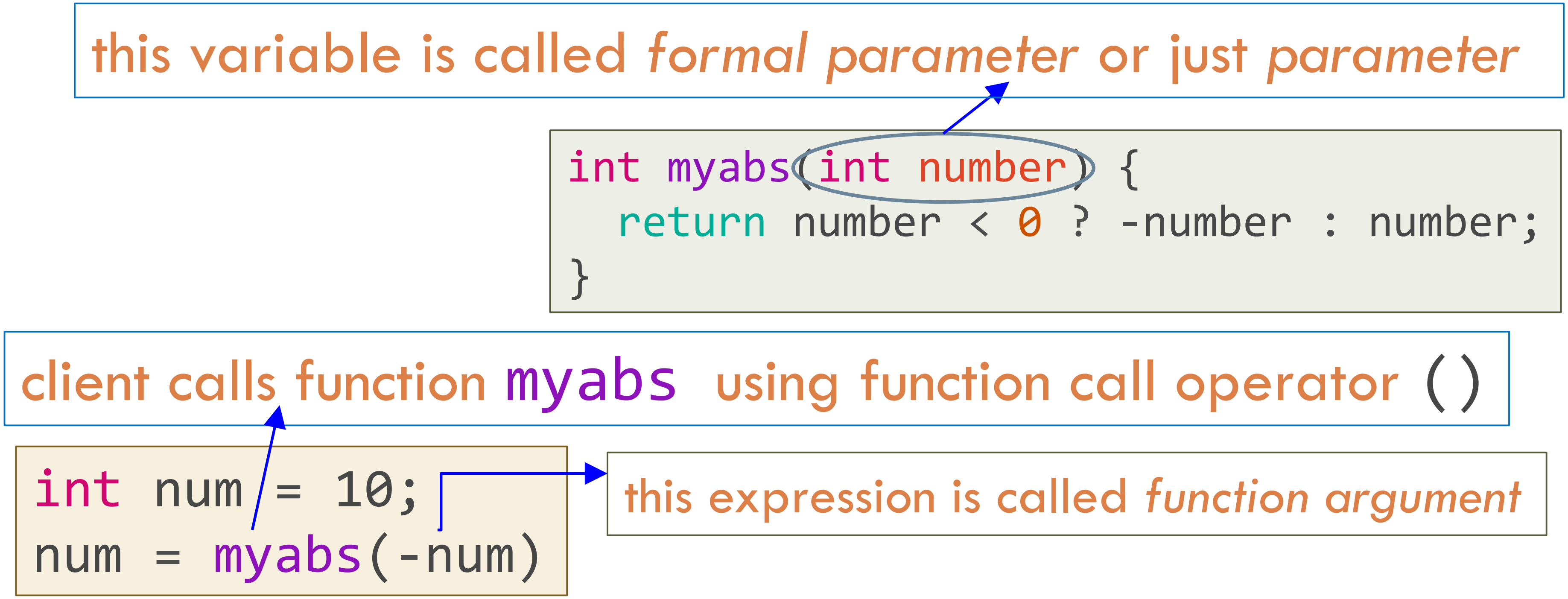
MODERN C++ DESIGN PATTERNS

Return Value Optimization & Copy Elision by Prasanna Ghali

Plan for Today

- Copy Elision: RVO, NRVO, URVO

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

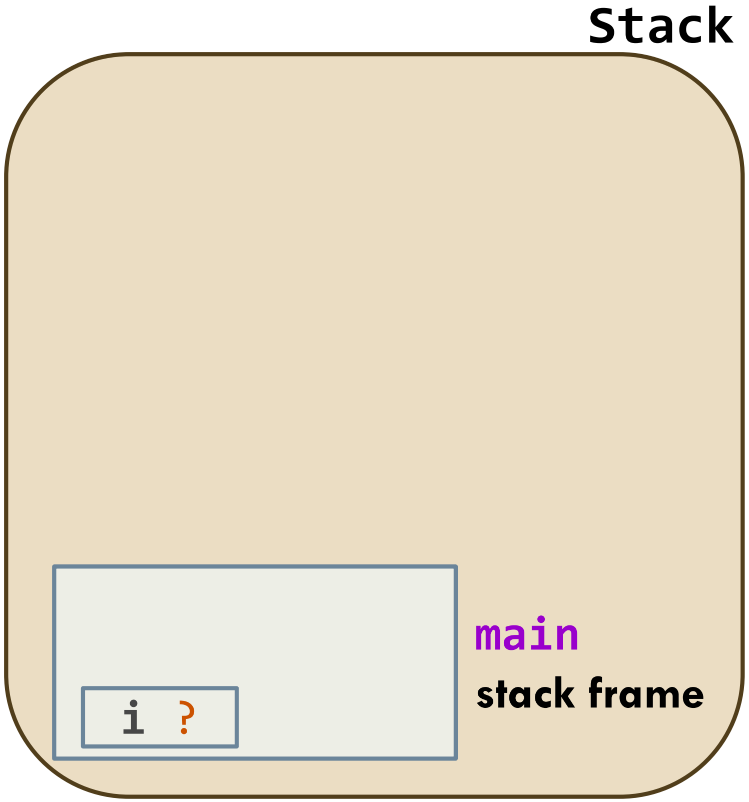
```c
#include <stdio.h>

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;
}
```

□ 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)

```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;
}
```



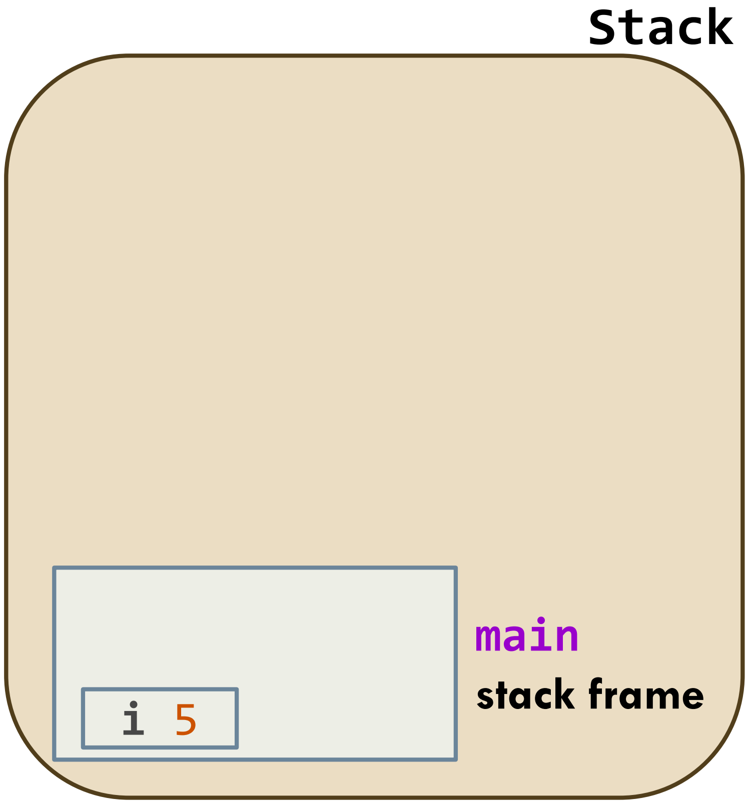
Stack

i ?

main
stack frame

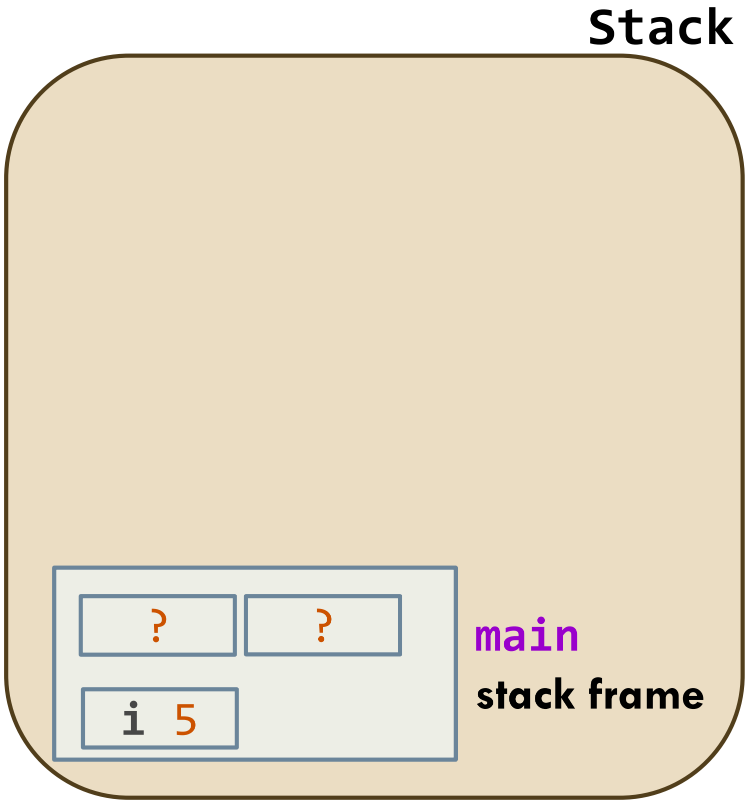
Functions: Pass-by-Value Convention (4/20)

```
#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 (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;
}
```



Stack

?

?

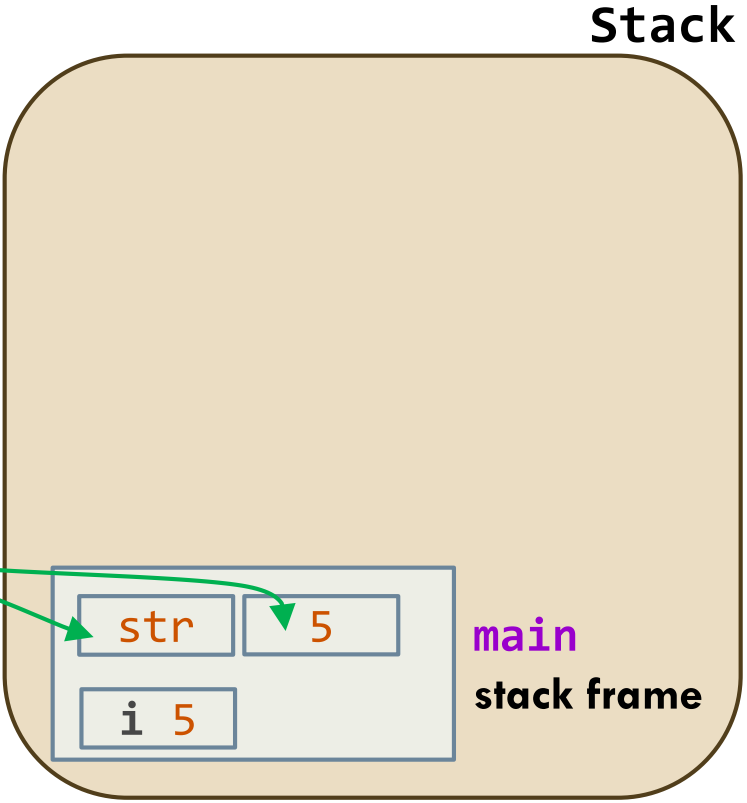
i 5

main

stack frame

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

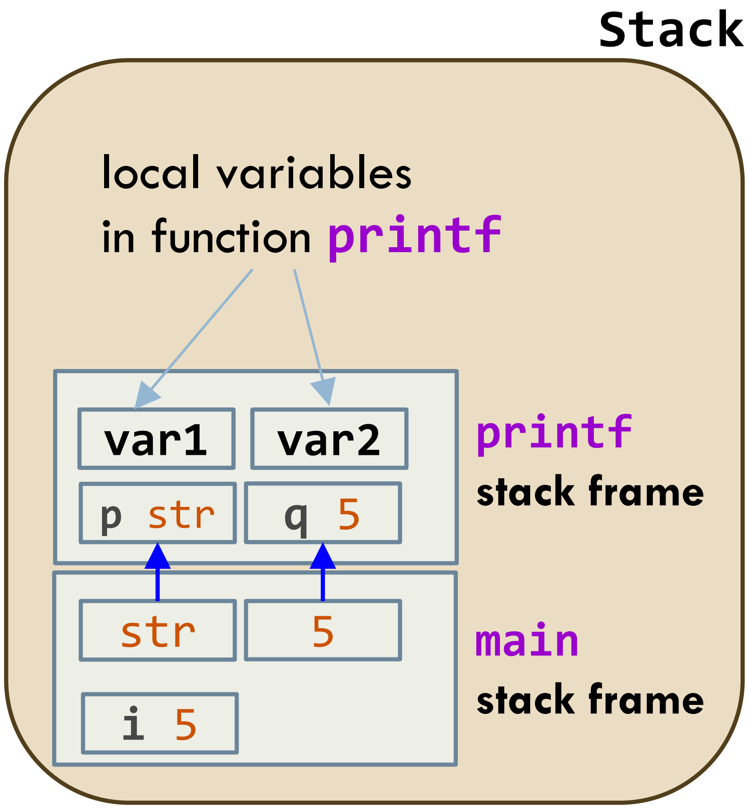
```c
#include <stdio.h>

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 (7/20)

Before call: i 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;
}
```



Stack

local variables
in function **printf**

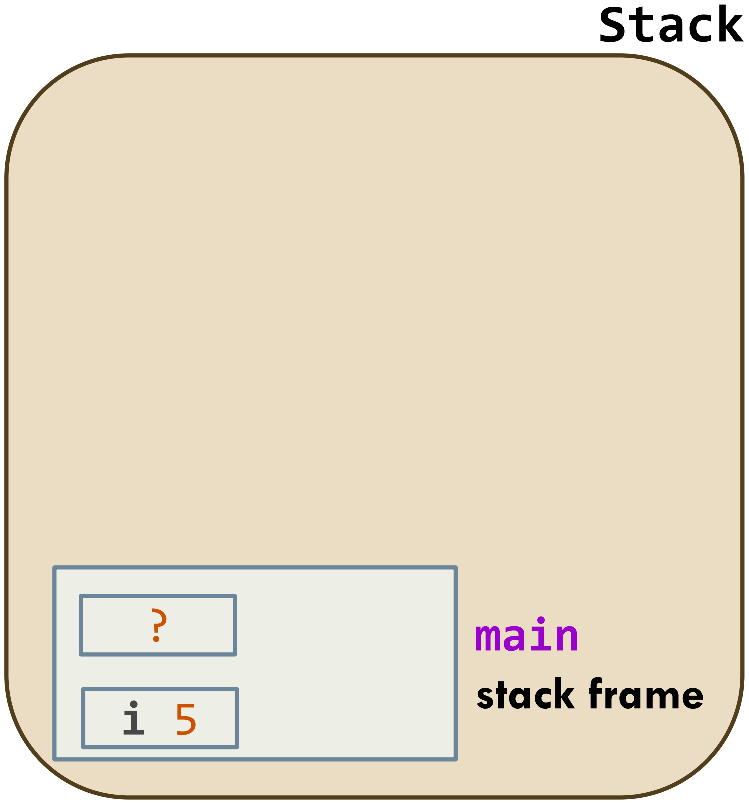
printf
stack frame

main
stack frame

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

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

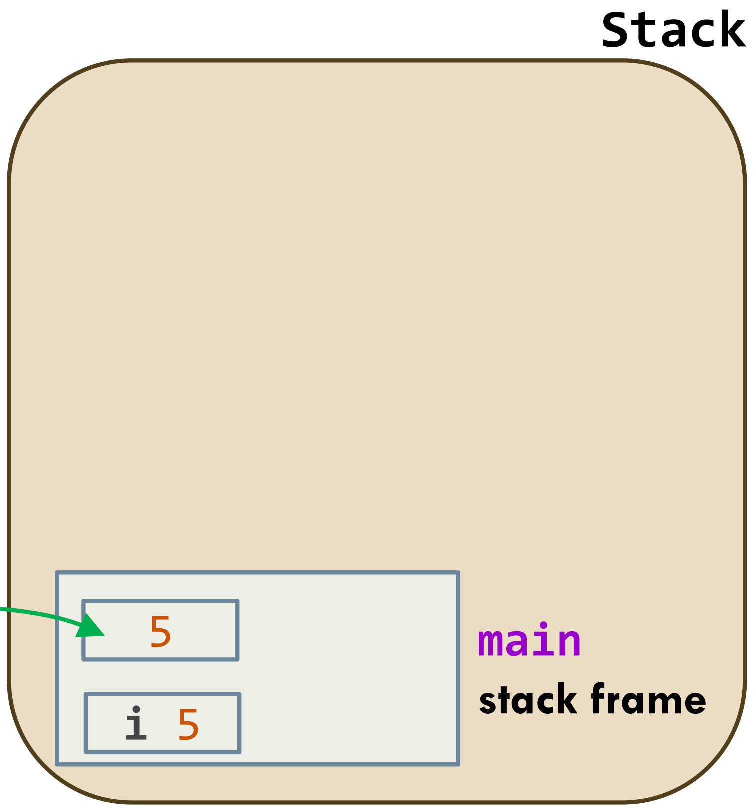
```c
#include <stdio.h>

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
```

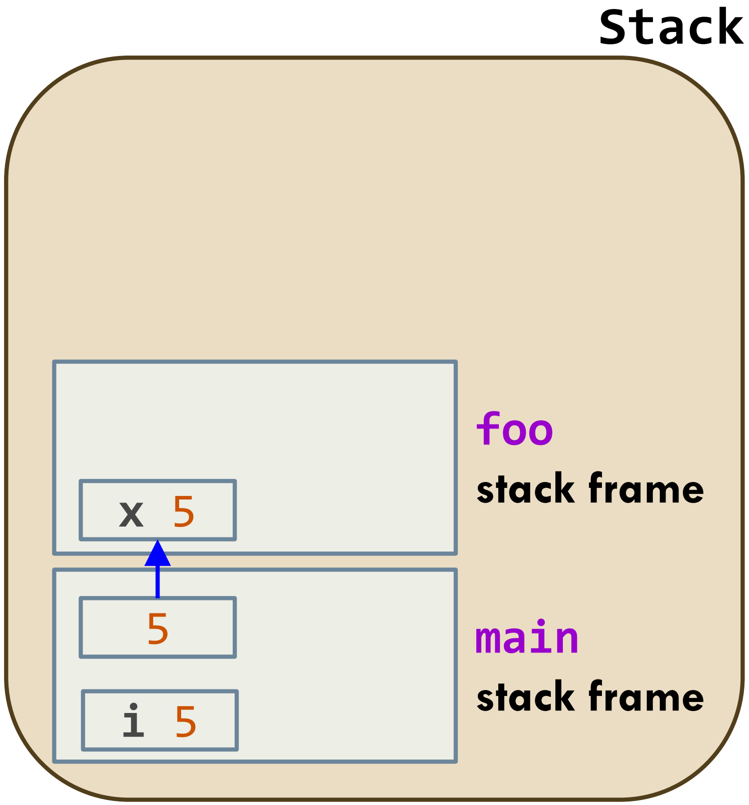
```c
#include <stdio.h>

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
```

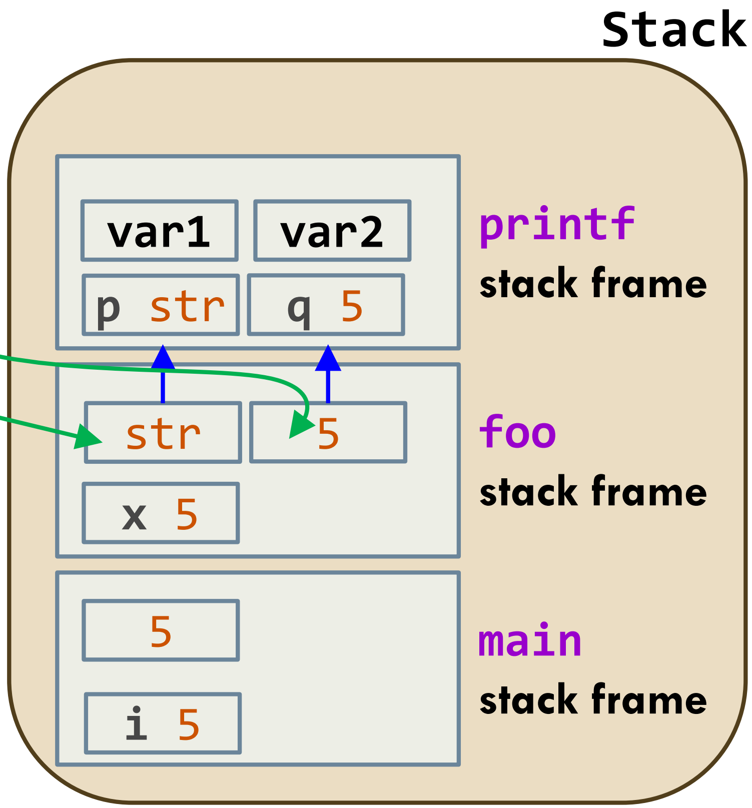
```
#include <stdio.h>

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
```

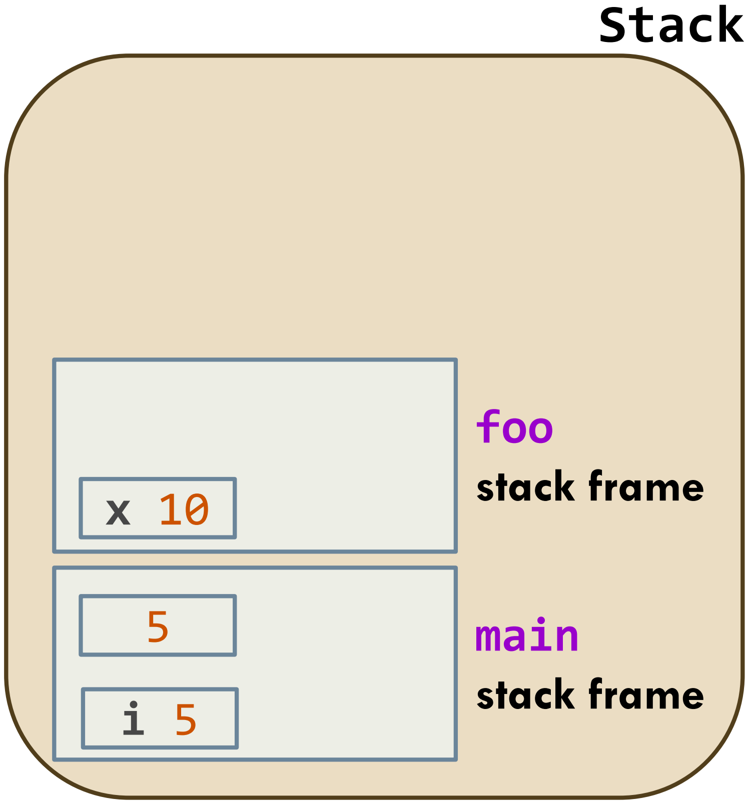
```c
#include <stdio.h>

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 (12/20)

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

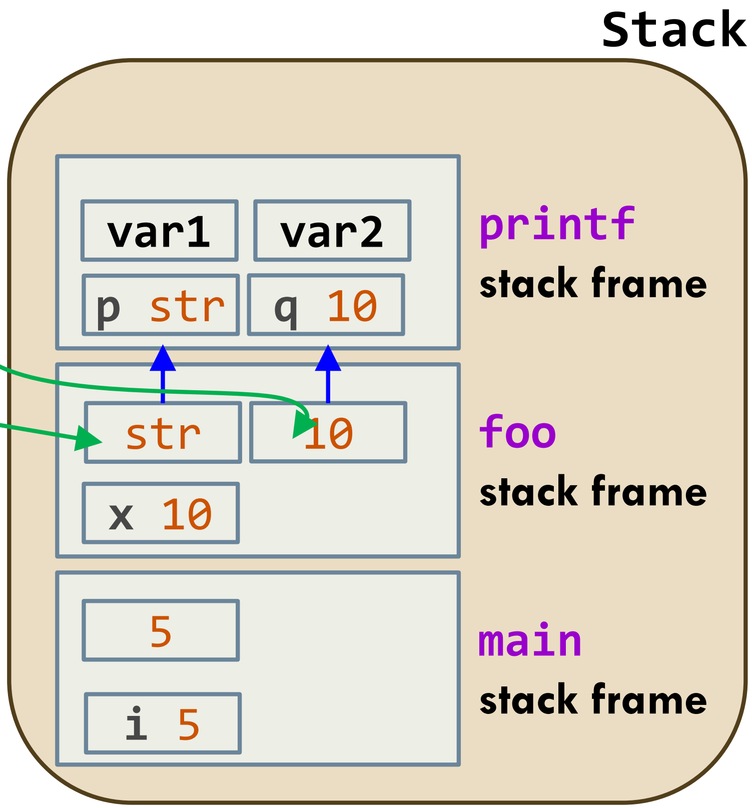
```c
#include <stdio.h>

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
```

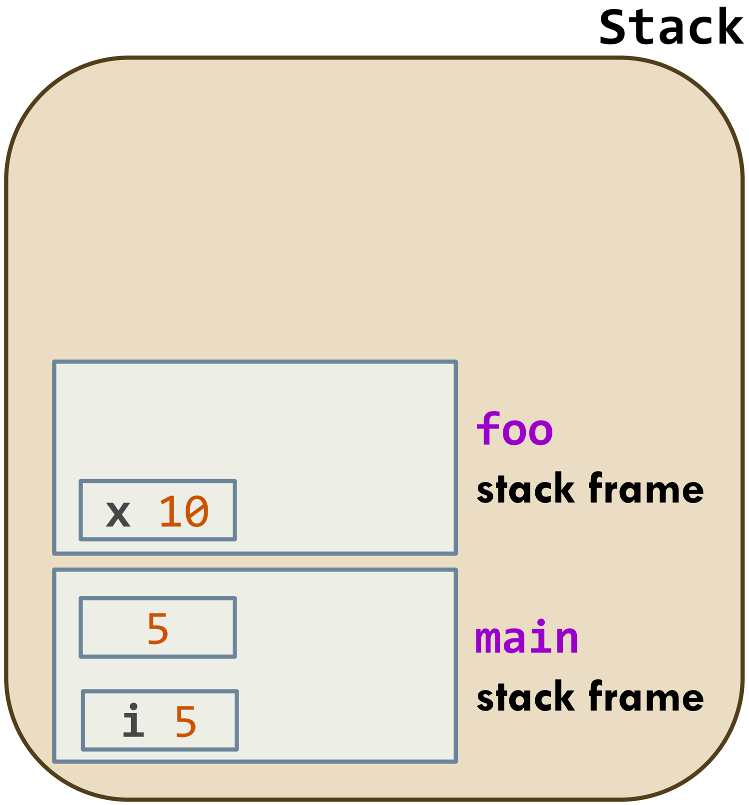
```c
#include <stdio.h>

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
```

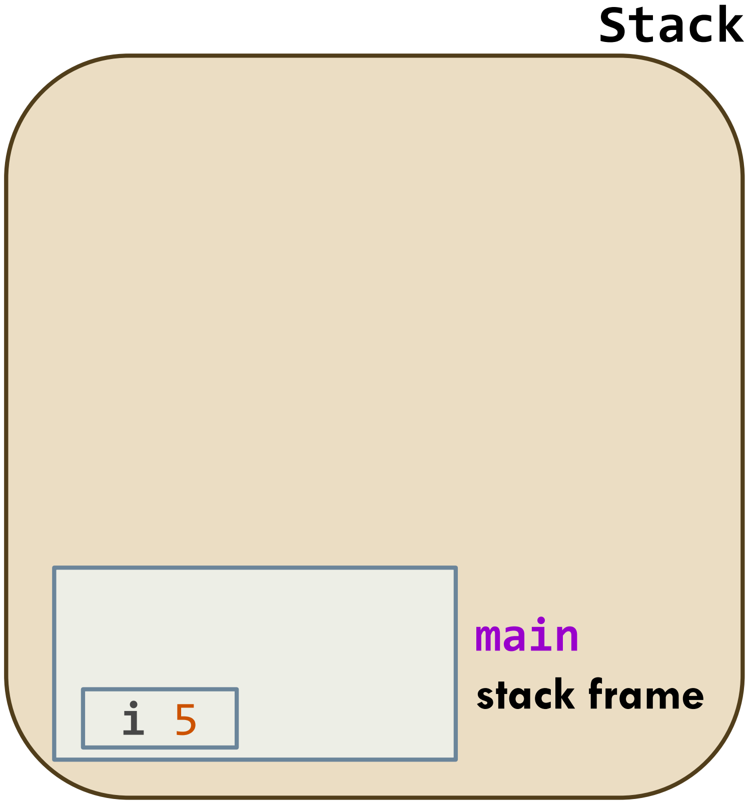
```c
#include <stdio.h>

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
```

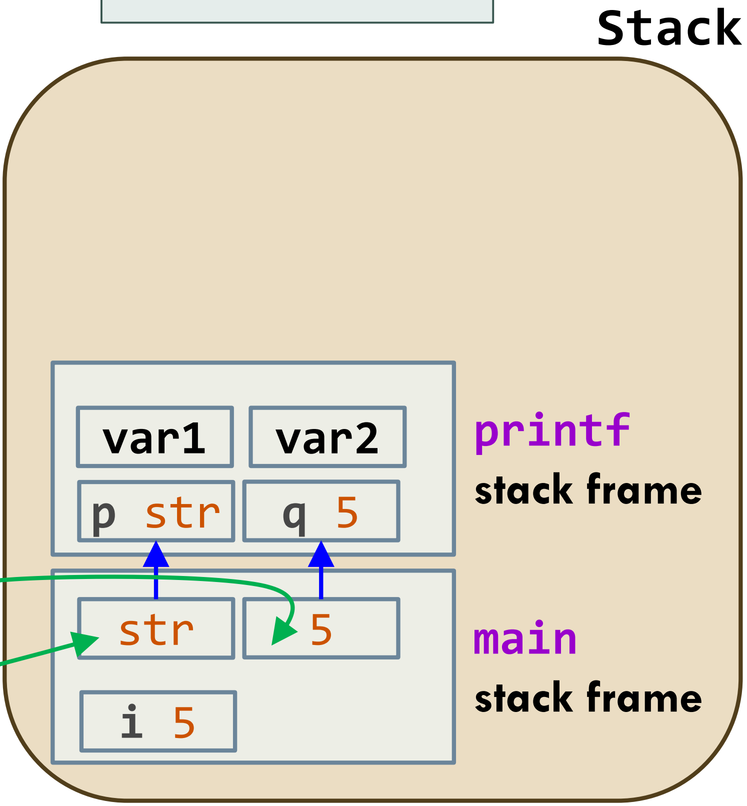
```c
#include <stdio.h>

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;
}
```

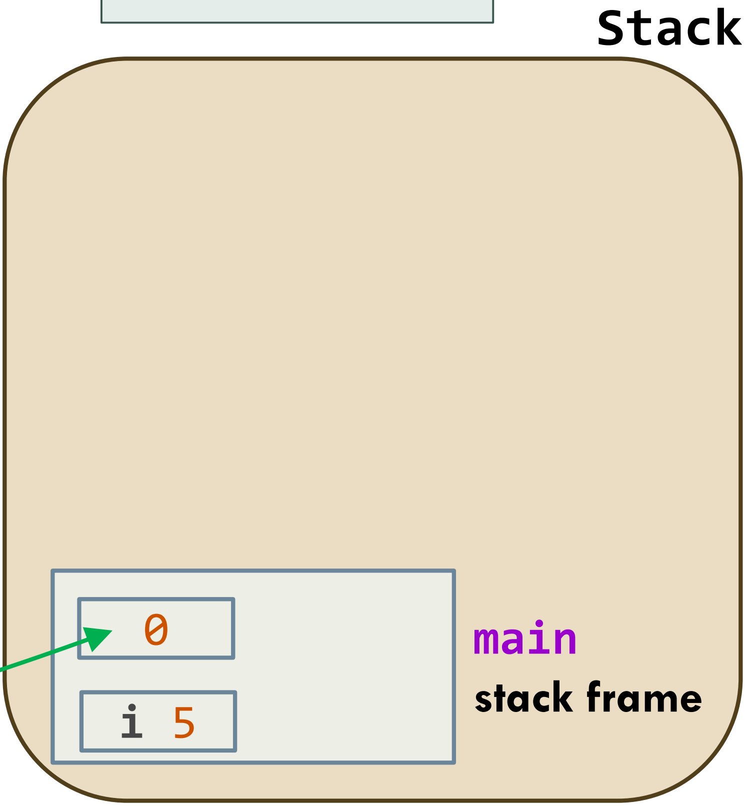
```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```



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

```
#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;
}
```

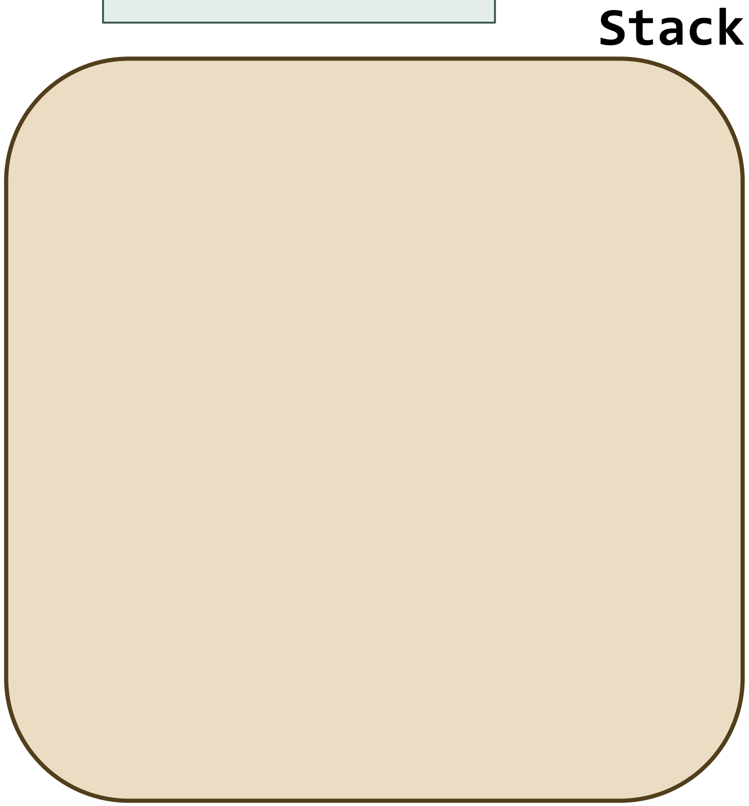
```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```



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)

```
#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;
}
```

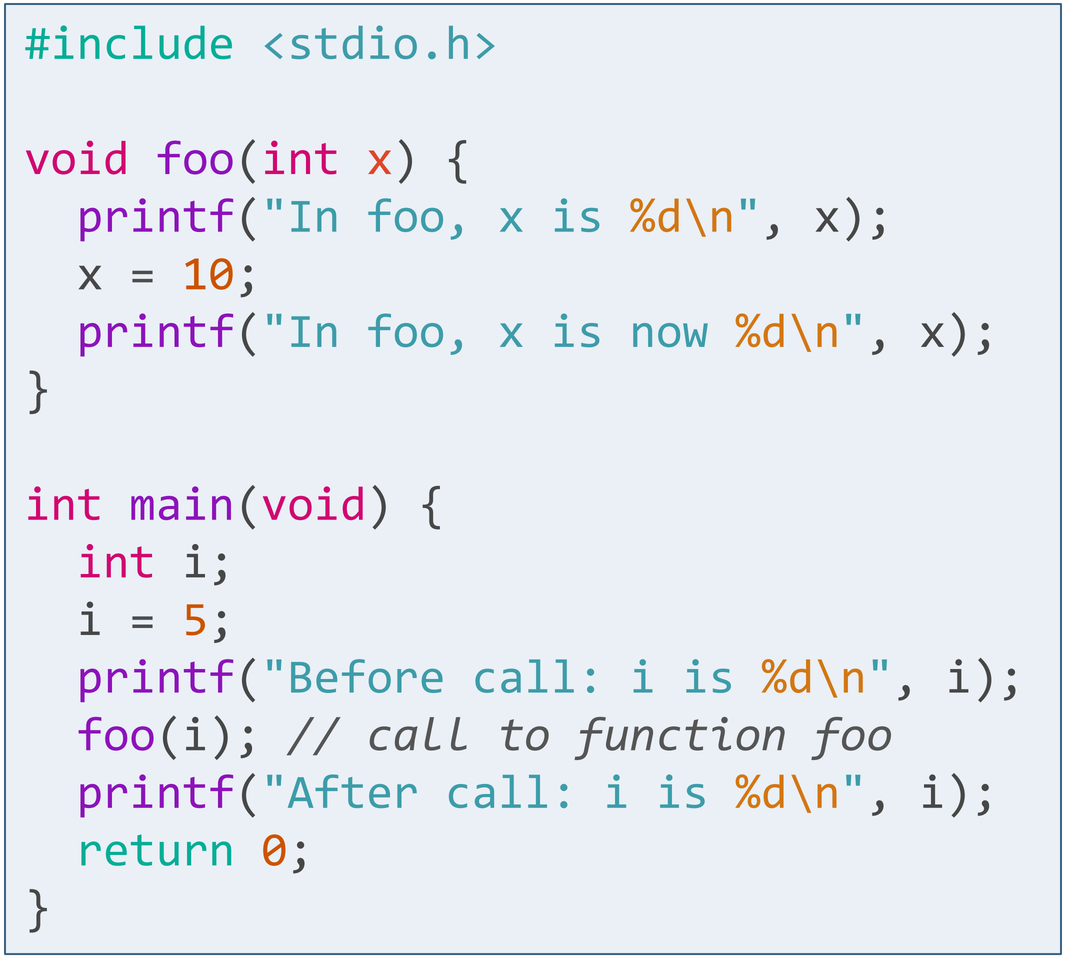
```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```

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;
}
```

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 (1/3)

- 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 (2/3)

□ Based on our understanding of stack-based function semantics in C/C++, we 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 (3/3)

- To avoid unnecessary copies, pass-by-reference becomes default mode of transferring resources to functions

```
~~void foo(X xx) {~~
void foo(X const &xx) {
  // use xx
}

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

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

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

What is Copy Elision?

When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the ctor selected for the copy/move operation and/or the dtor for the object have side effects.

In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object.

This elision of copy/move operations, called copy elision, is permitted in a return statement in a function with a class return type, when the expression is the name of a non-volatile automatic object with the same type (ignoring cv-qualification) as the function return type.

Copy Elision in C++17

- Compilers are required to provide copy elision when function returns unnamed temporary object
- In some cases, 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

Advantages of Copy Elision

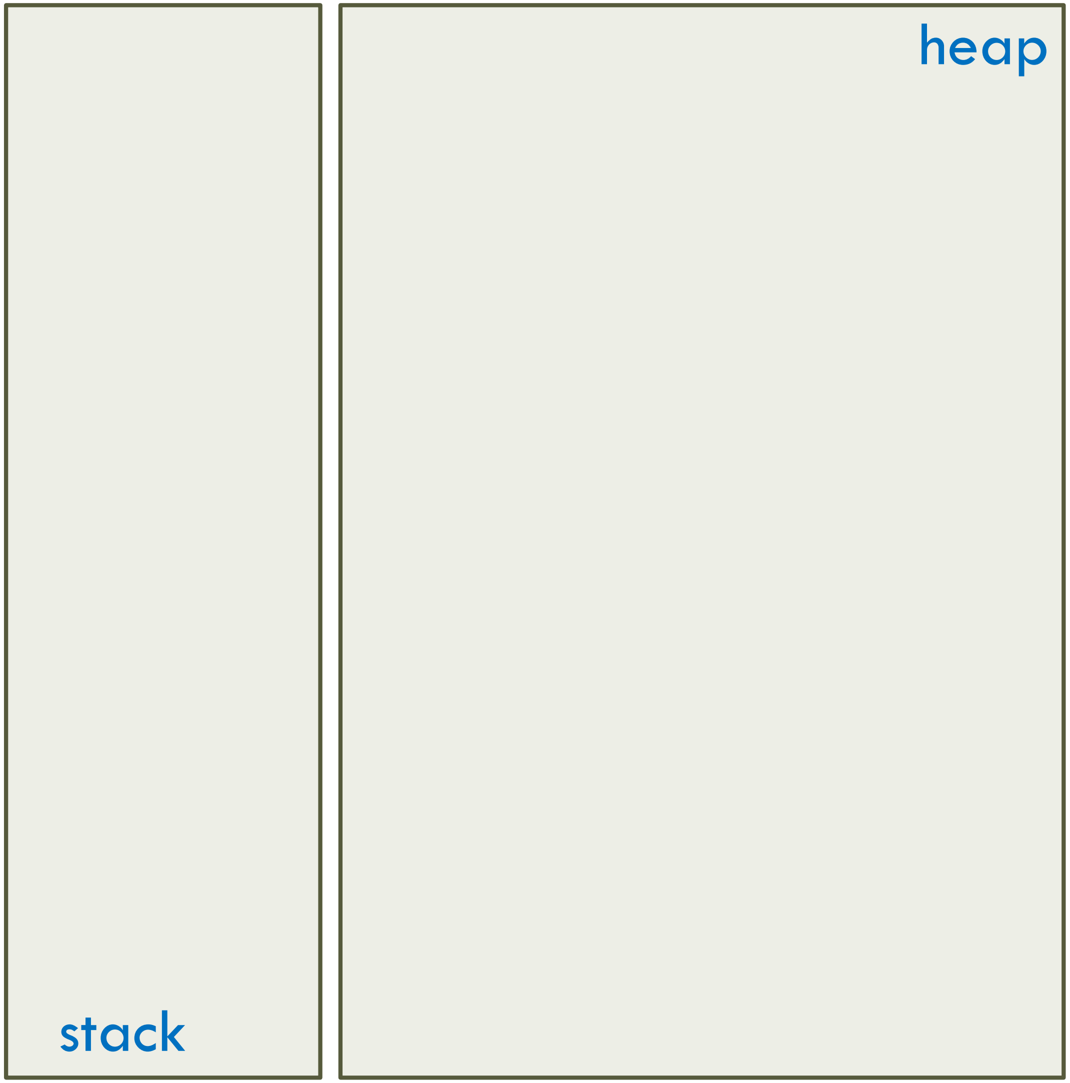
- Avoids copying object that function returns as its value by avoiding creation of temporary object on stack
- Permits function to efficiently return large objects
- Simplifies function’s interface
- Eliminates possibilities for issues such as resource leaks from arising

RVO, NRVO, URVO

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

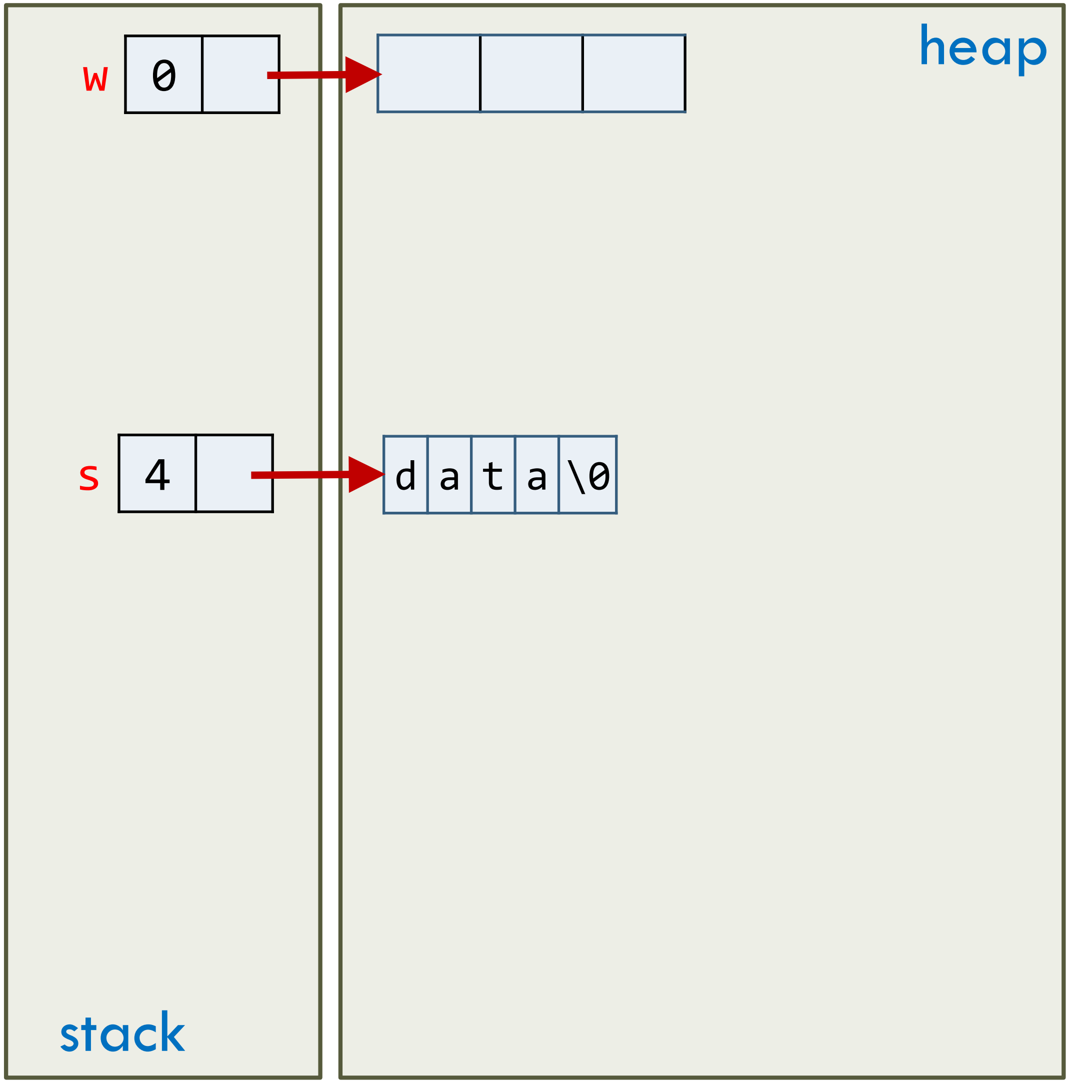
Motivation for RVO (1/12)

```
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 (2/12)

```
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();
```

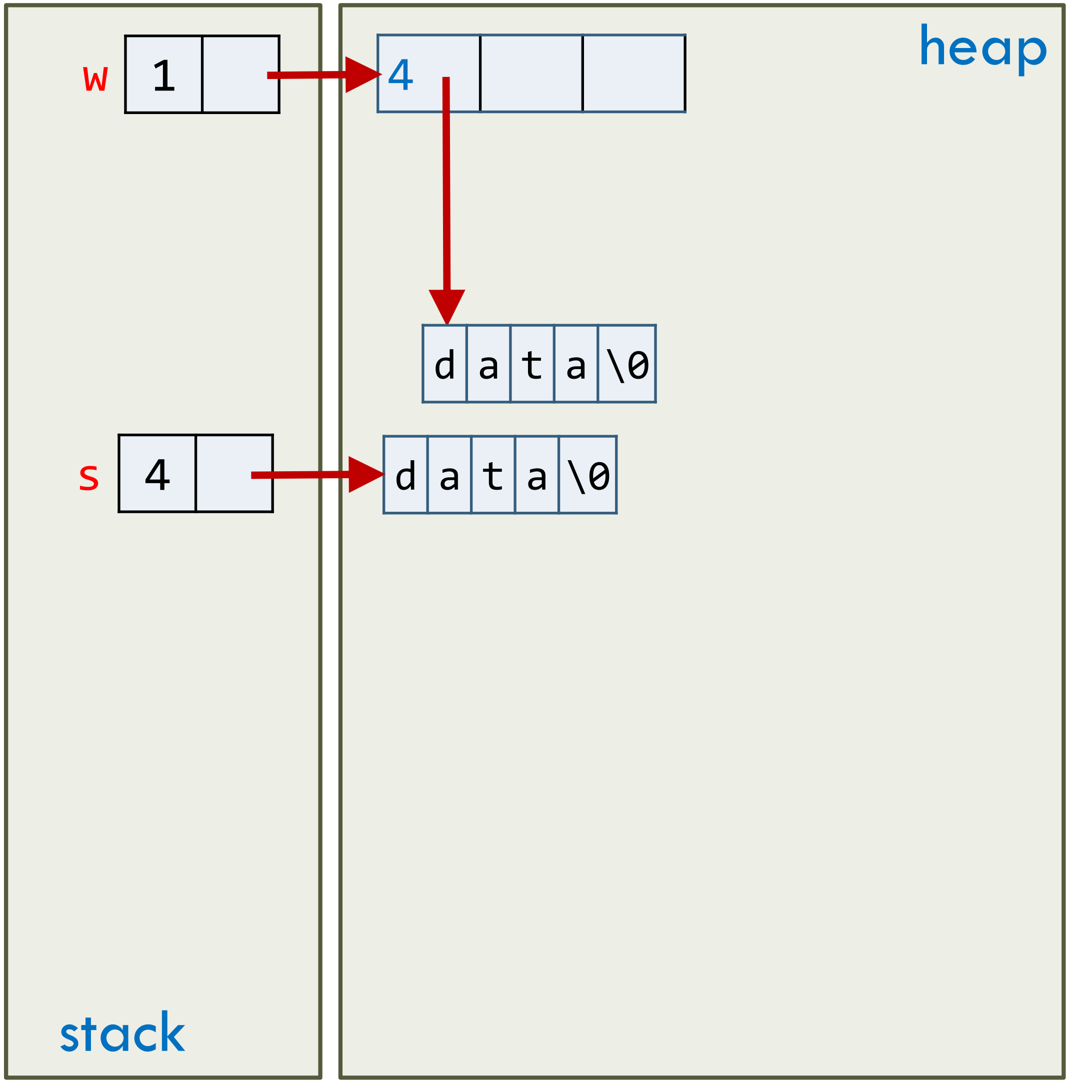


Motivation for RVO (3/12)

```
Str operator+(Str const& lhs, Str const& rhs) {
  Str tmp{lhs}; // initialize tmp with copy of lhs
  tmp += rhs;   // add
  return tmp;
}

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

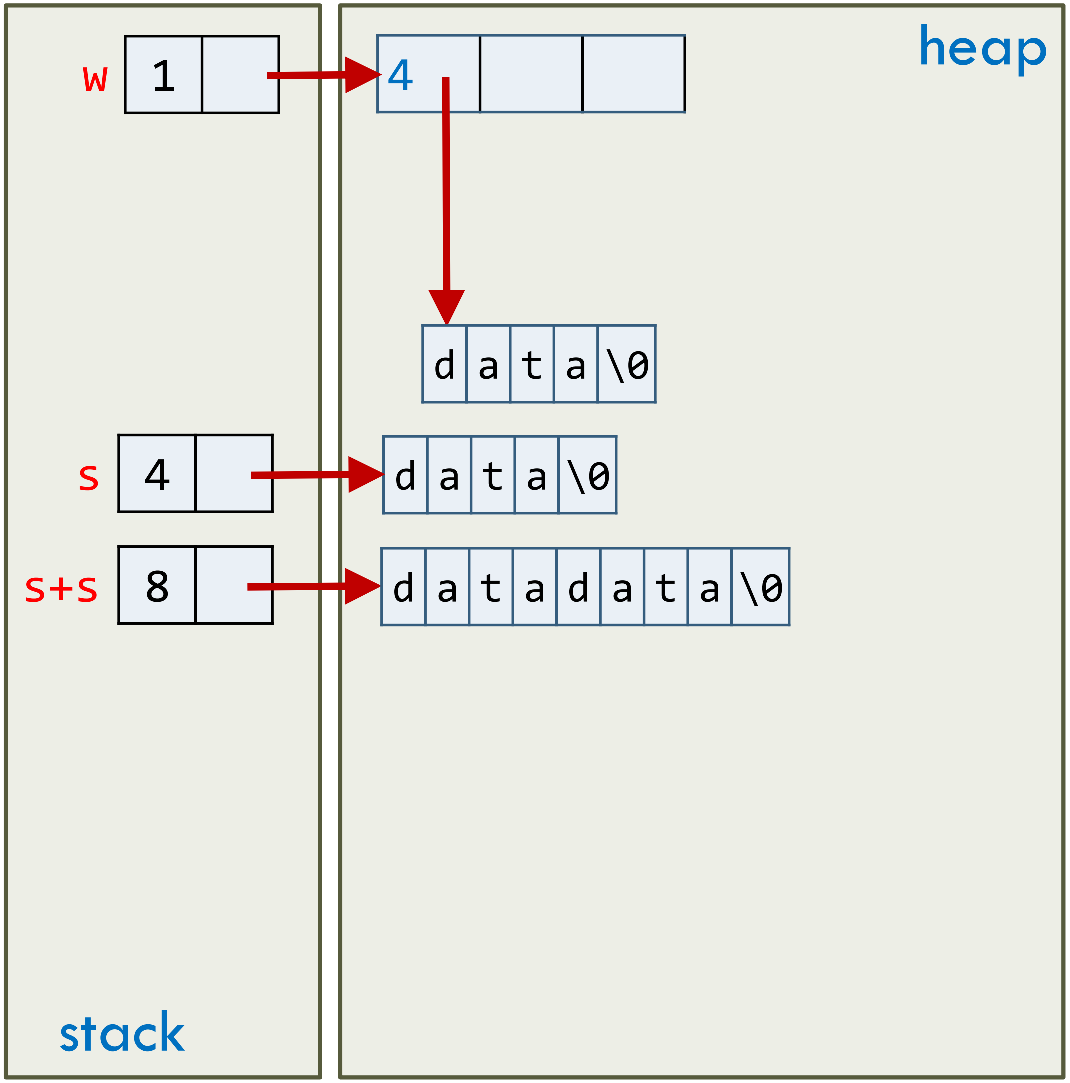
Motivation for RVO (4/12)

```
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();
```



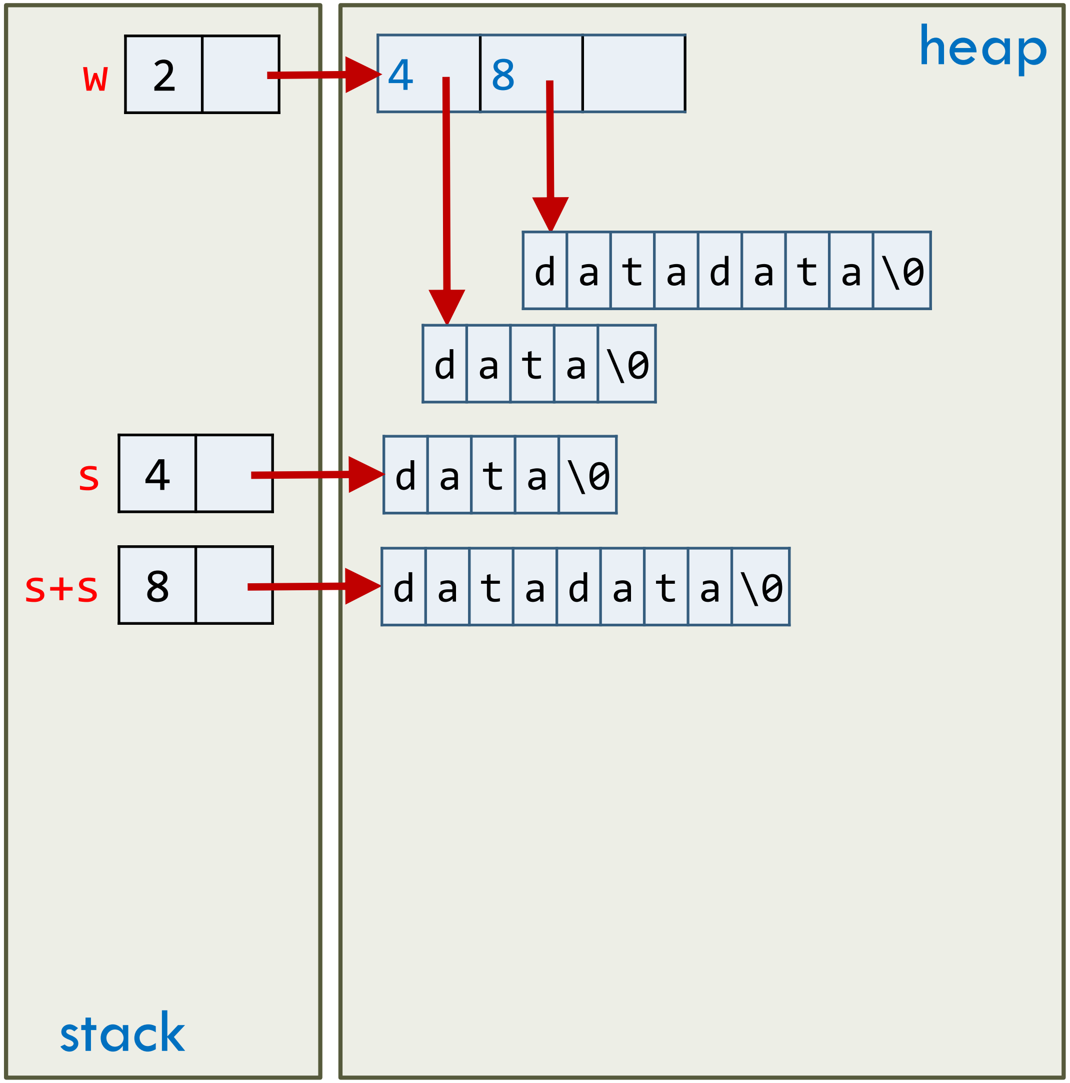
Motivation for RVO (5/12)

```
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();
```



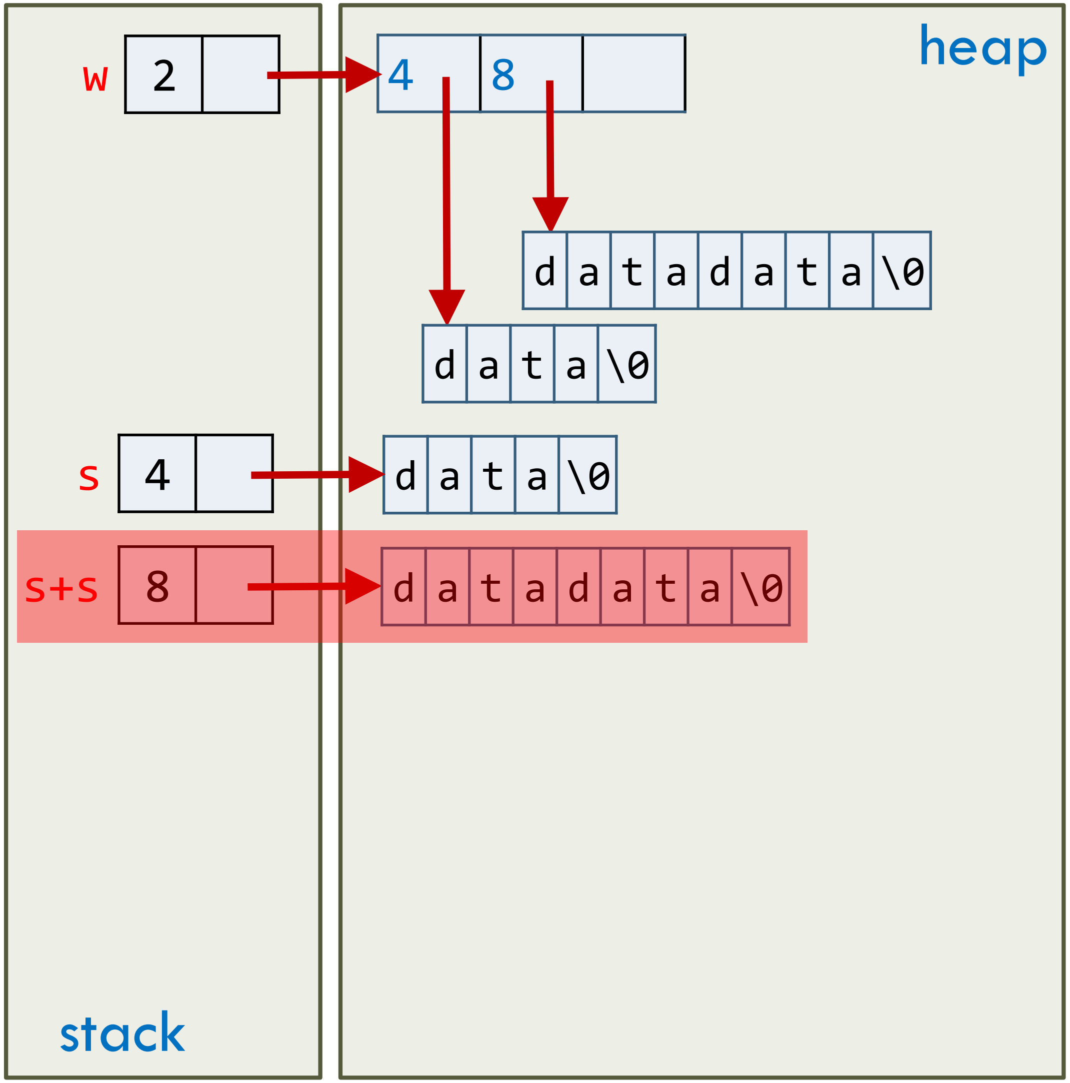
Motivation for RVO (6/12)

```
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();
```



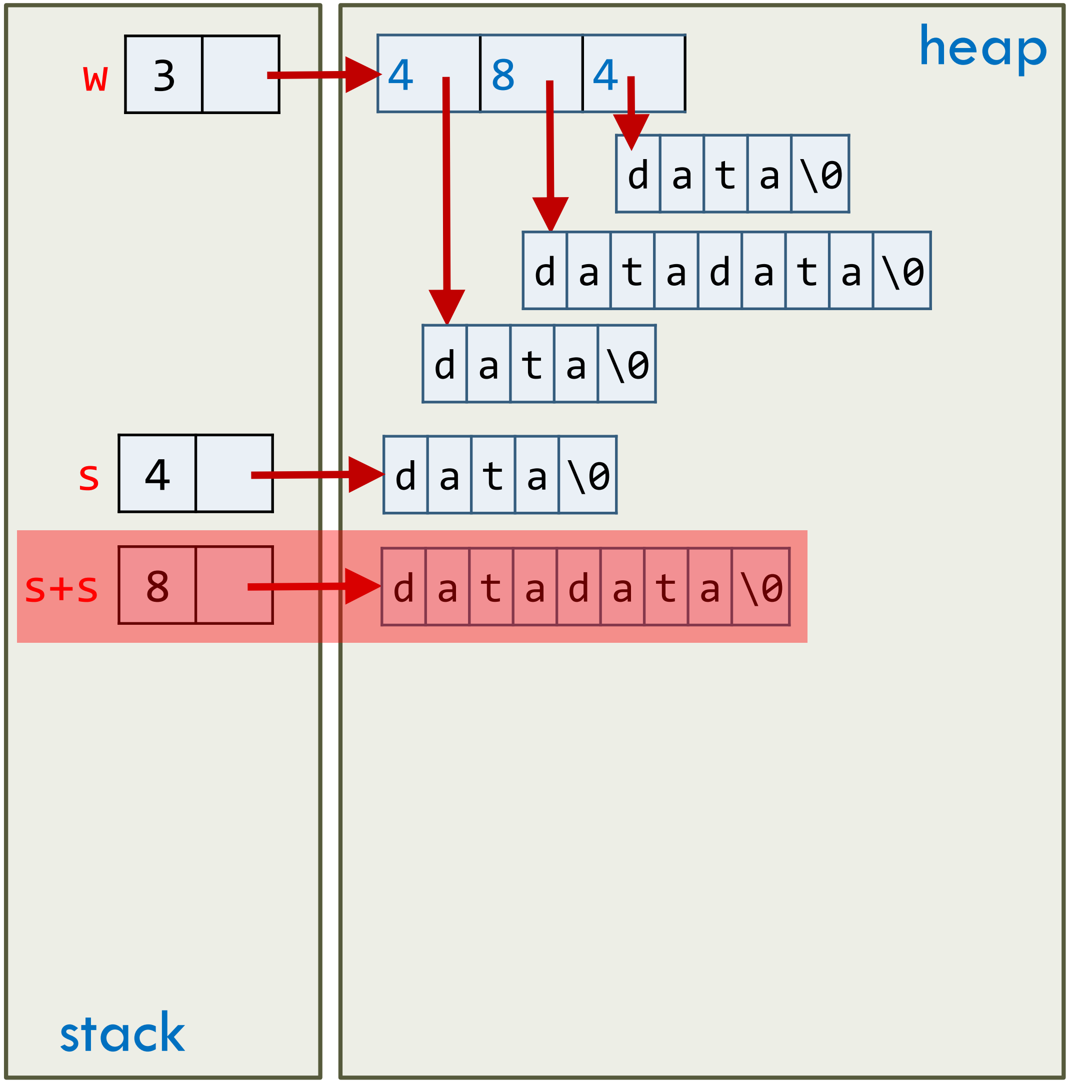
Motivation for RVO (7/12)

```
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();
```

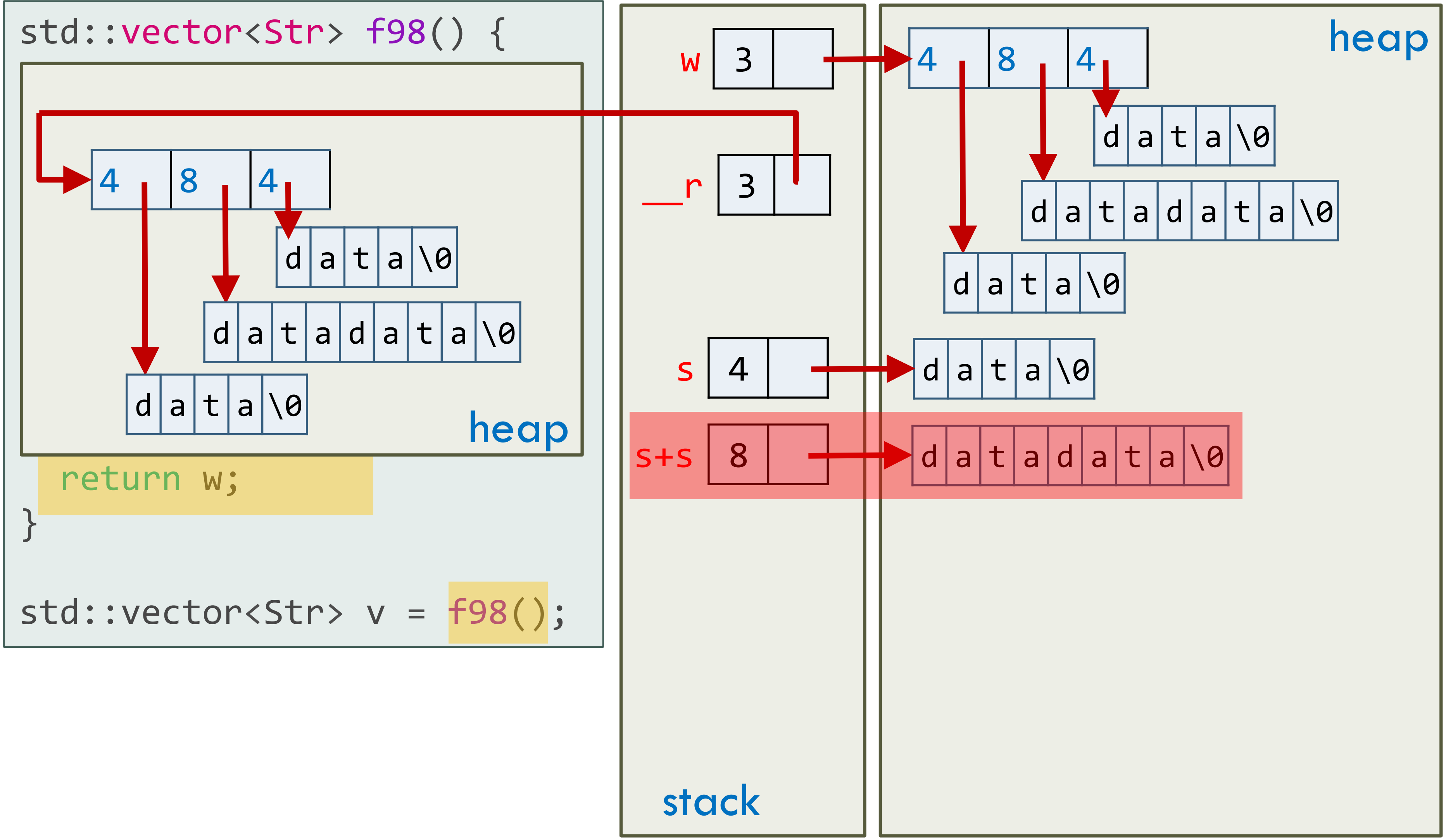


Motivation for RVO (8/12)

```
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();
```



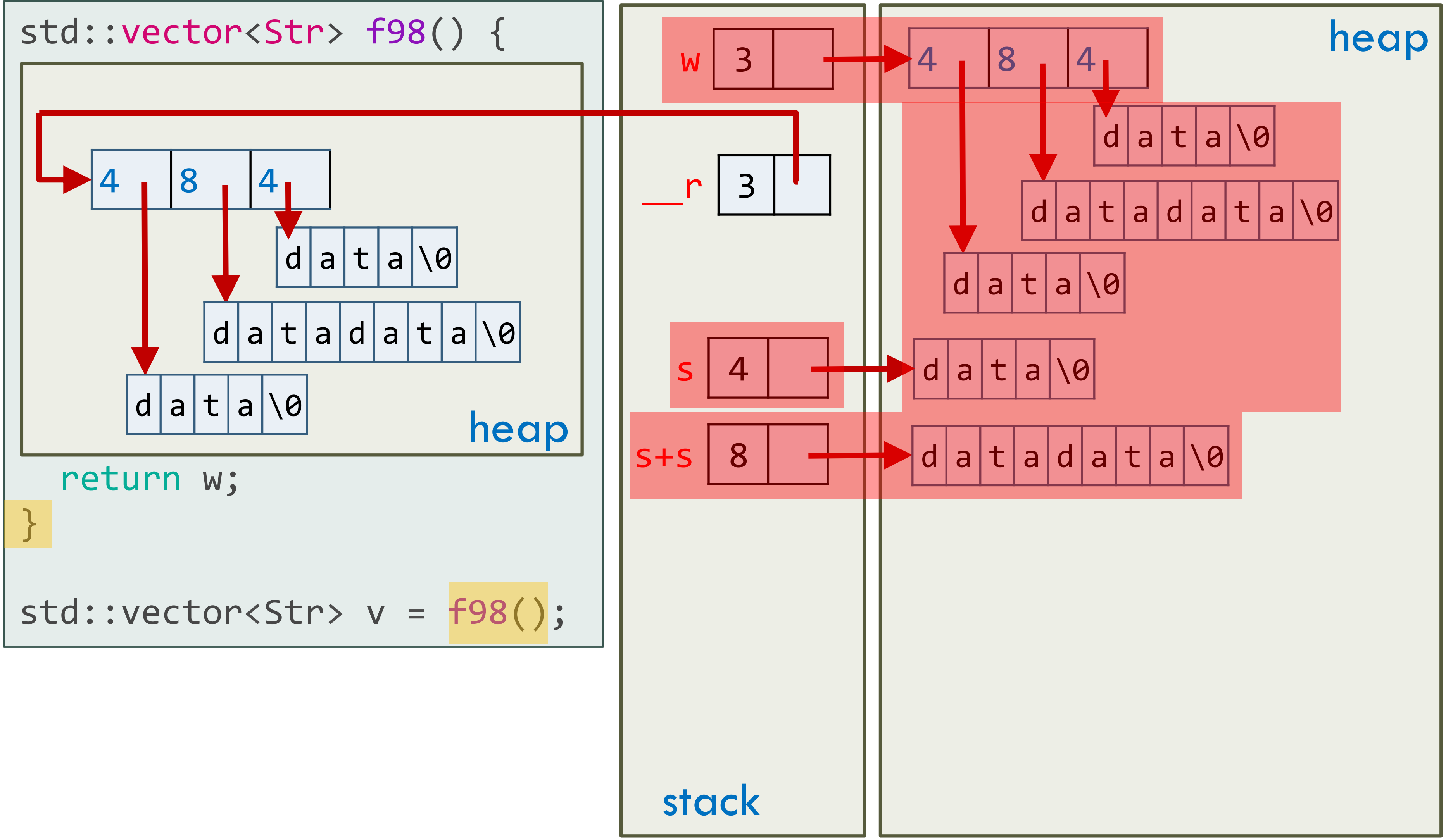
Motivation for RVO (9/12)



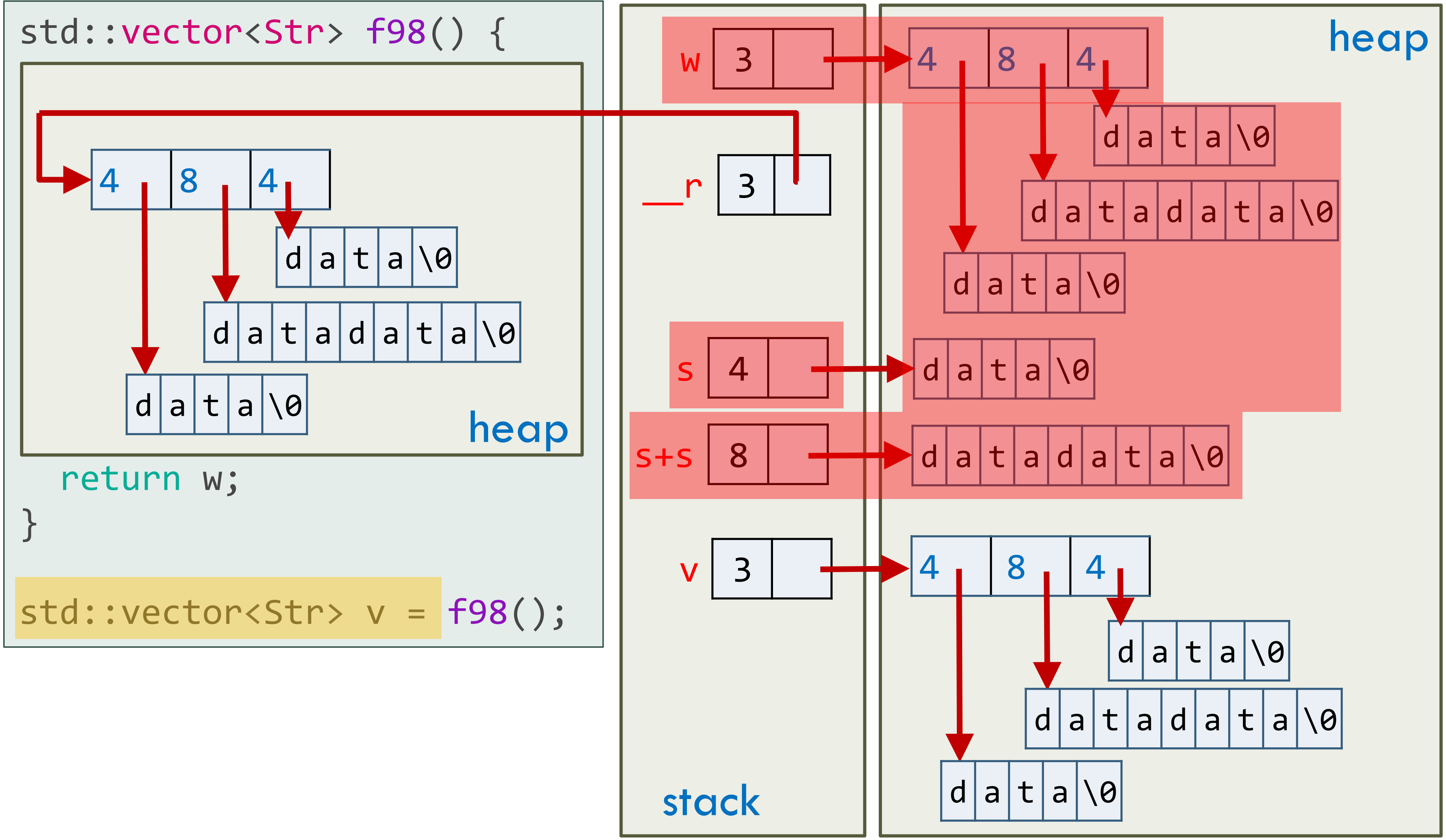
```
std::vector<Str> f98() {
  return w;
}

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

Motivation for RVO (10/12)

```
std::vector<Str> f98() {
    return w;
}

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

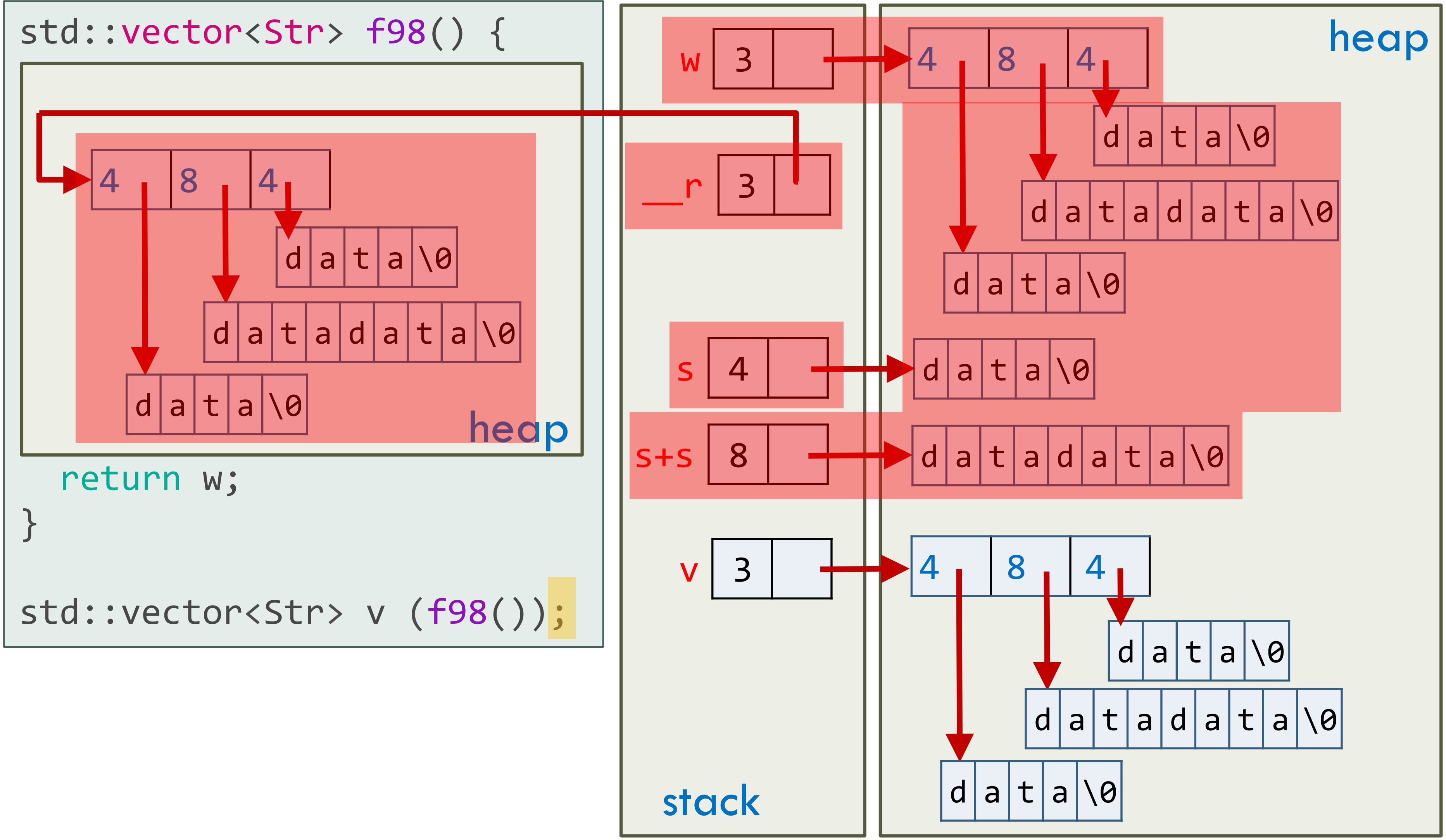


Motivation for RVO (11/12)



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

  return w;
}

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

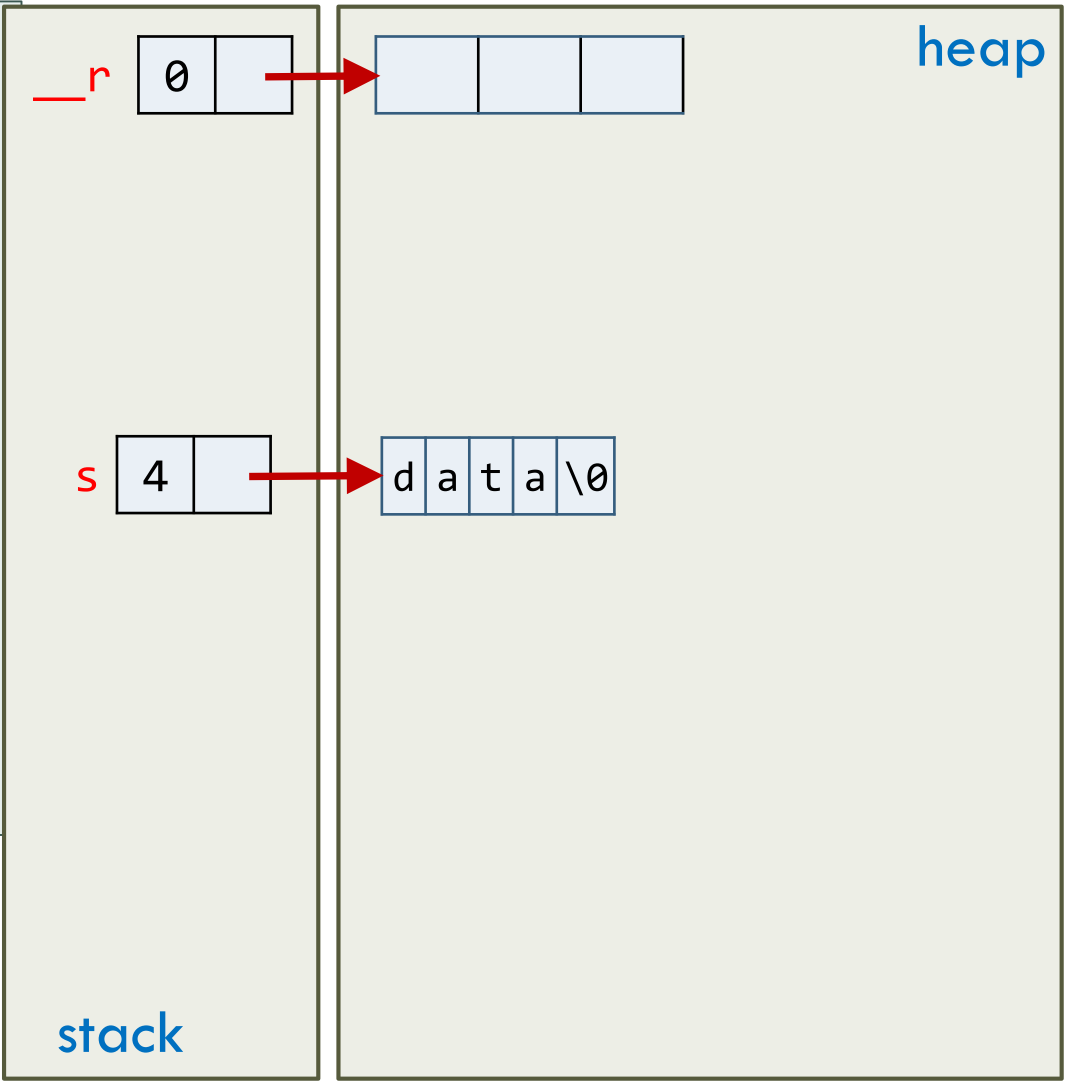
Motivation for RVO (12/12)



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

  return w;
}

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

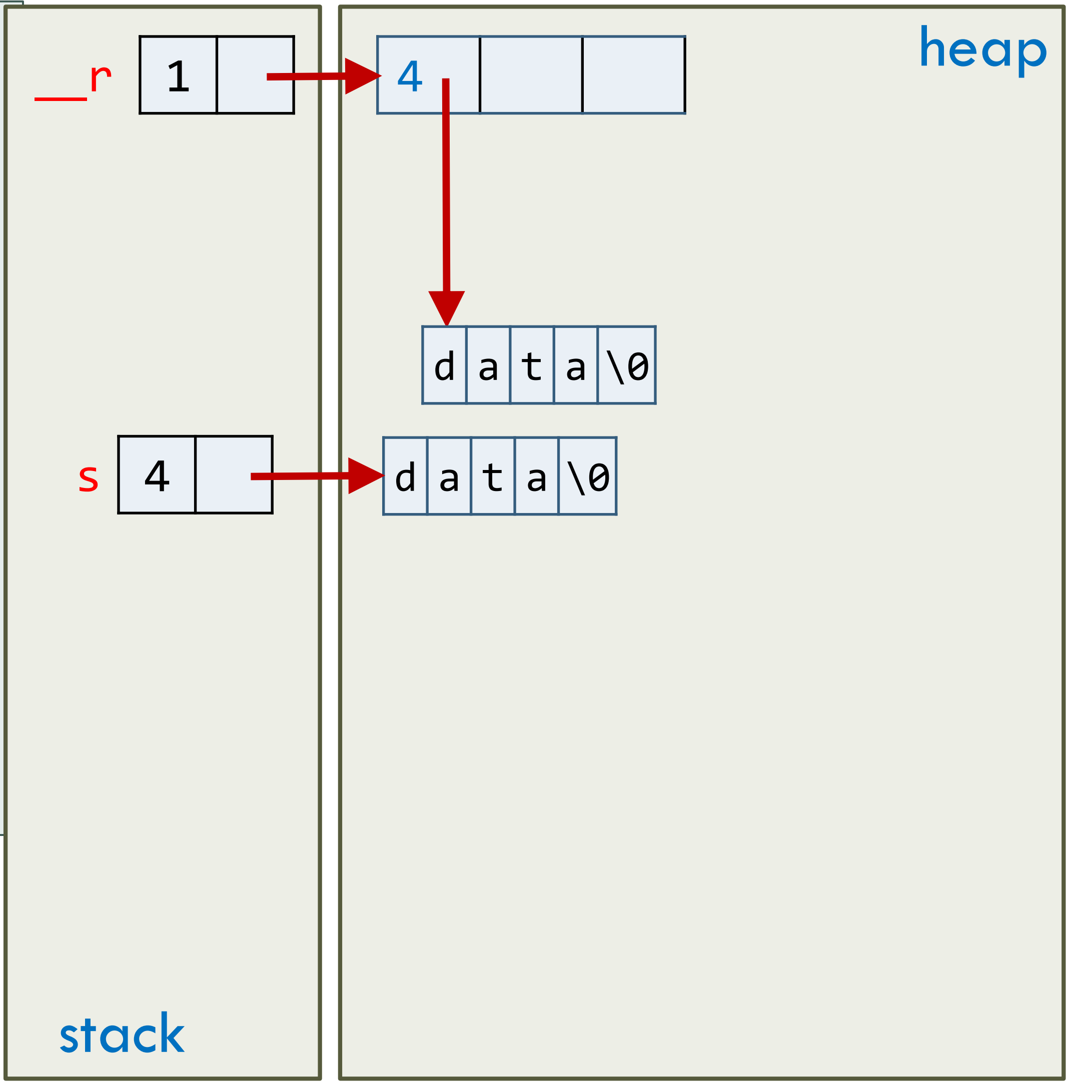
With RVO (1/8)

```
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);
```



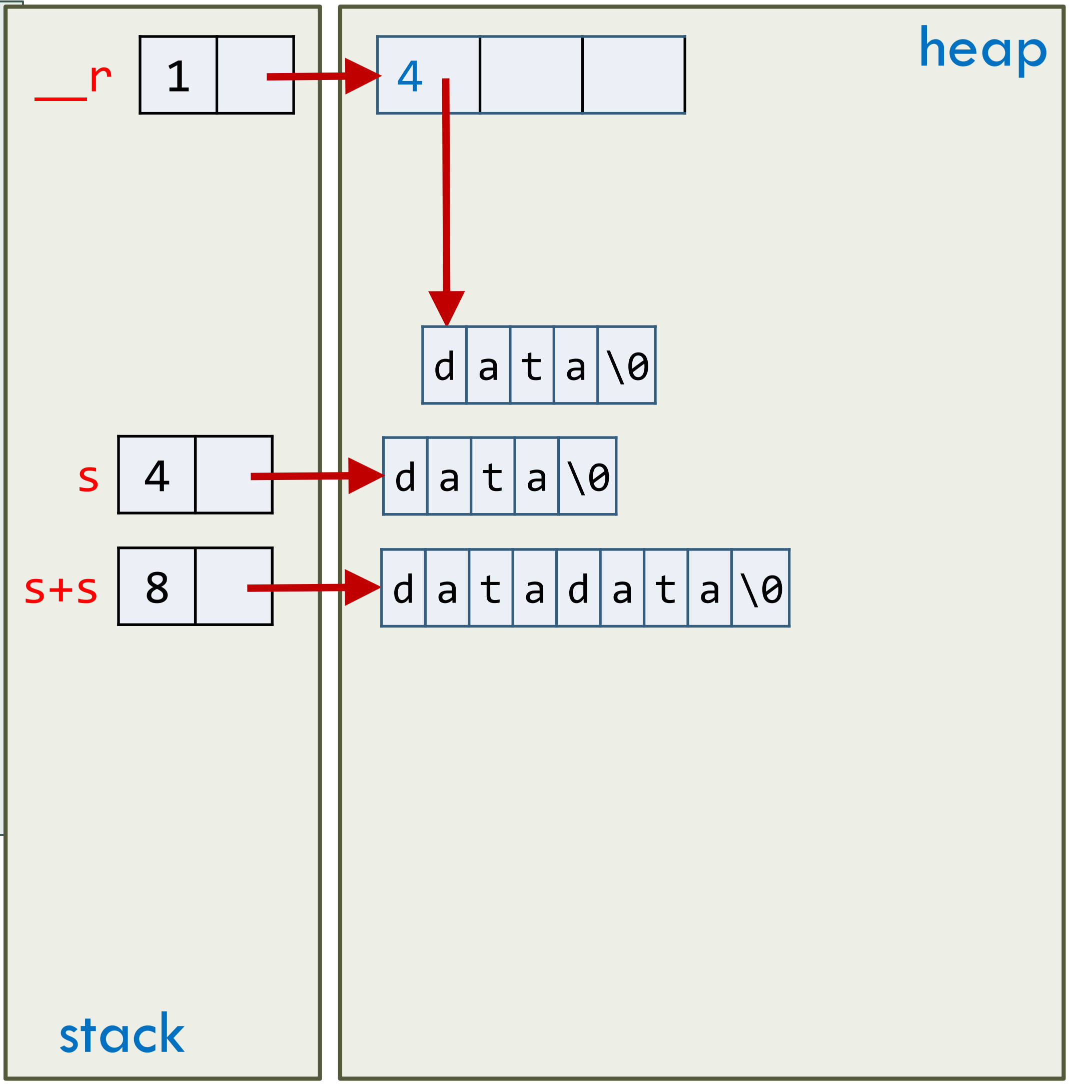
With RVO (2/8)

```
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);
```



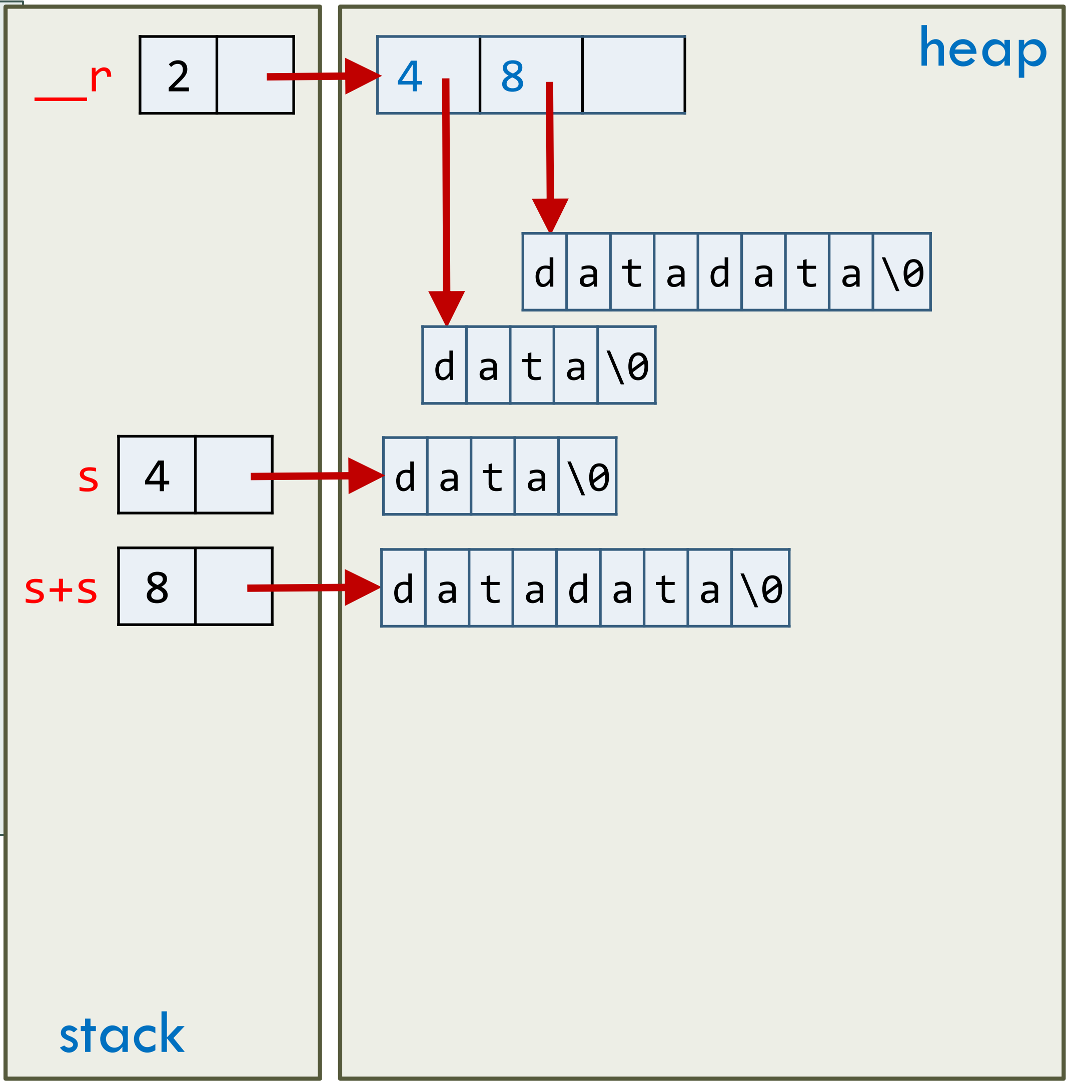
With RVO (3/8)

```
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);
```



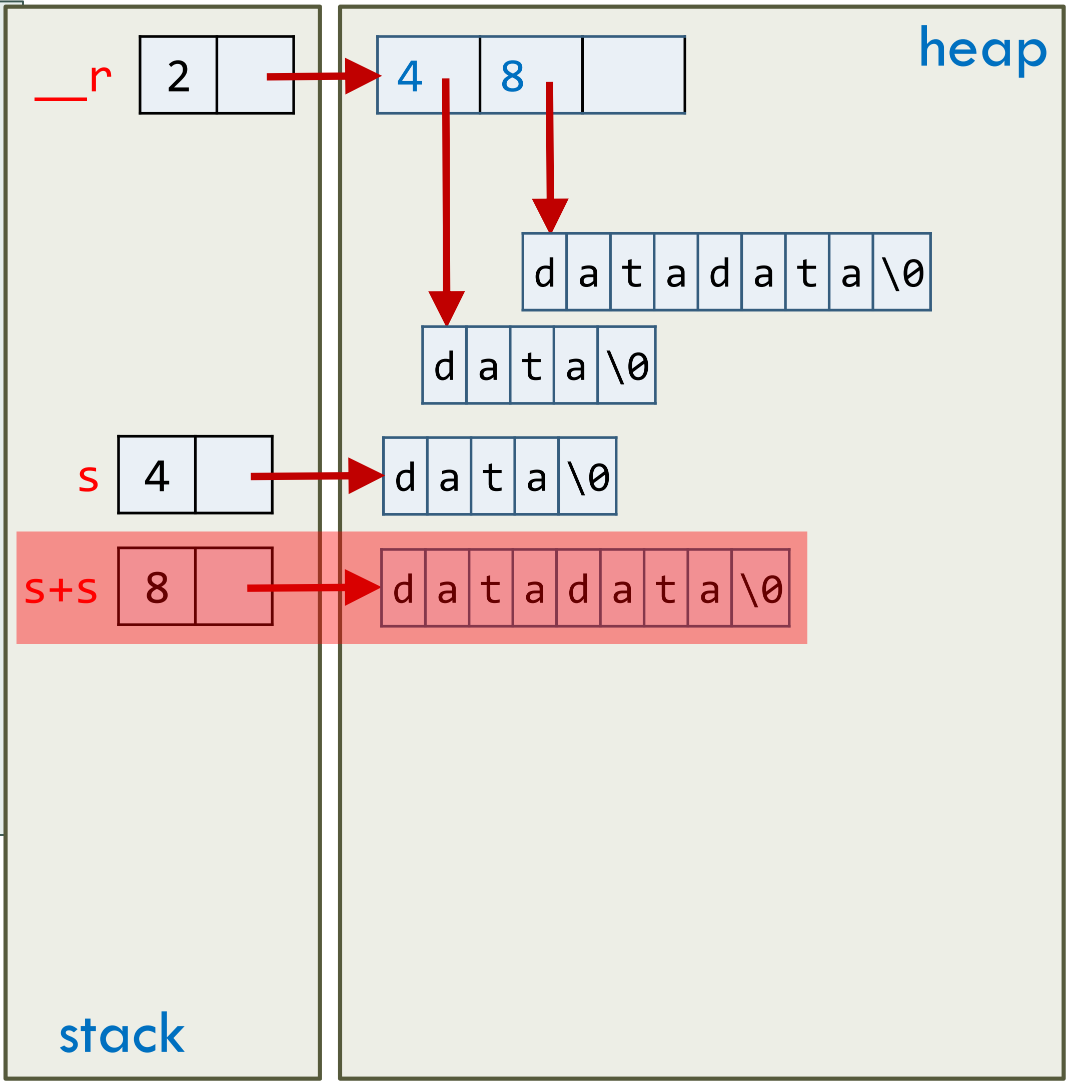
With RVO (4/8)

```
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);
```



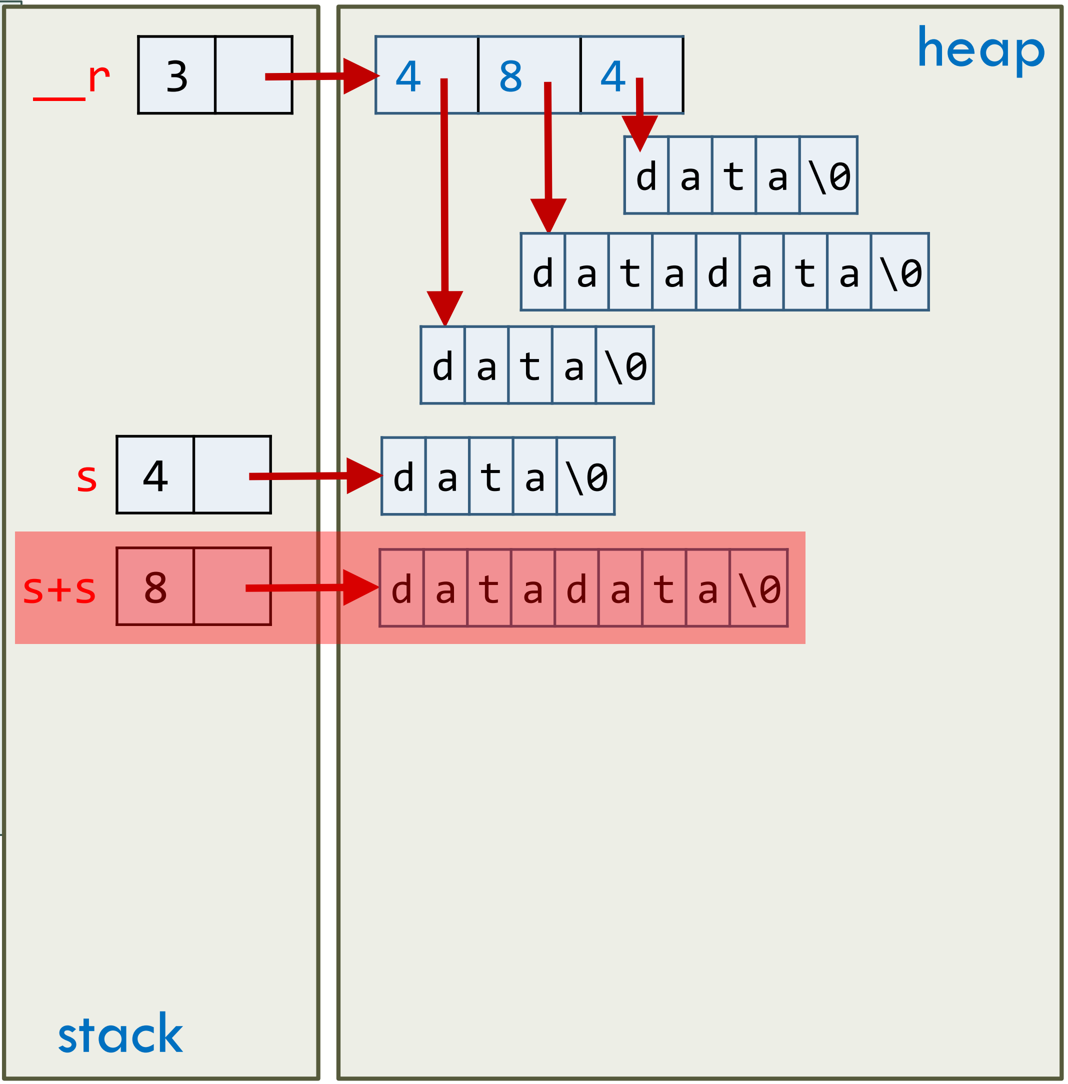
With RVO (5/8)

```
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);
```



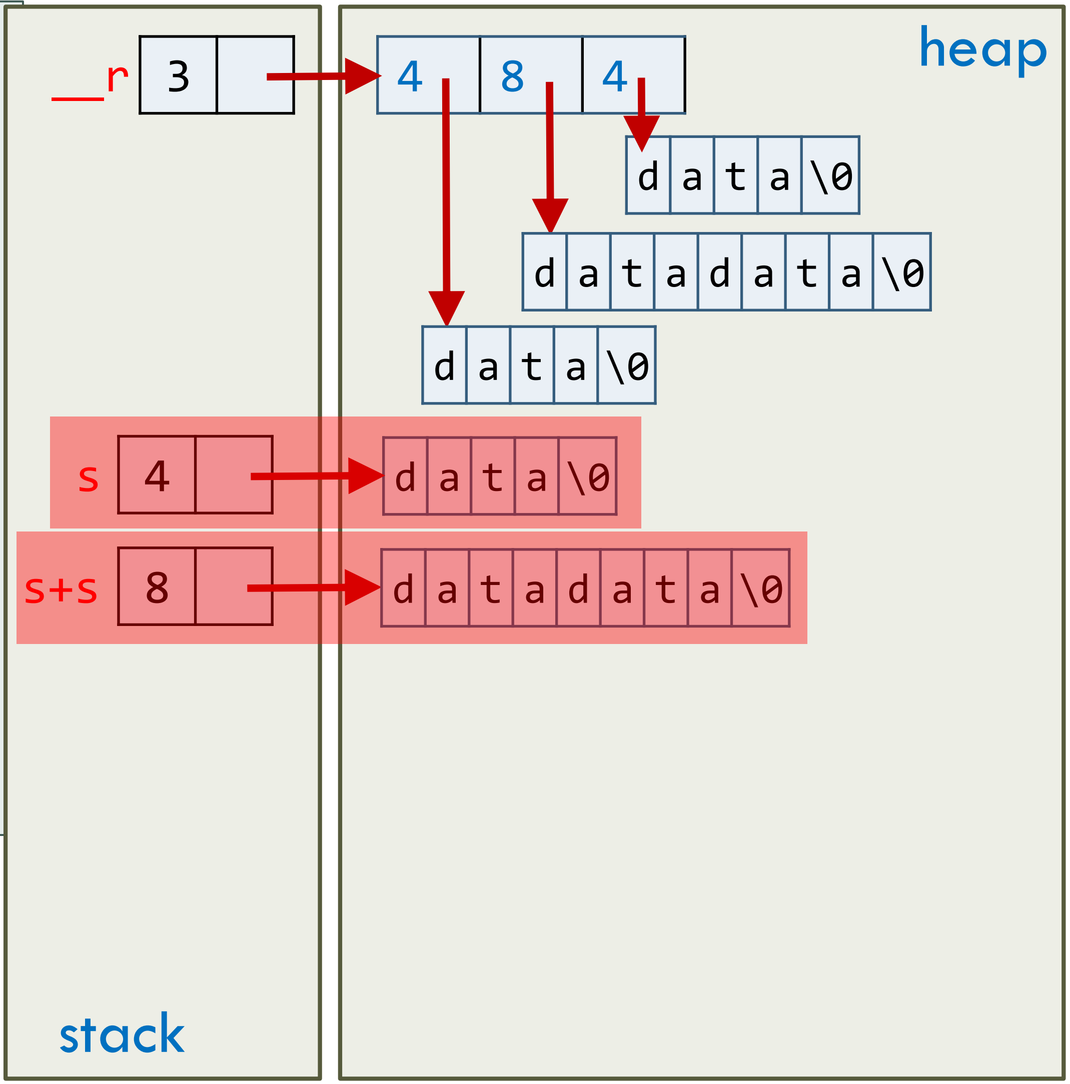
With RVO (6/8)

```
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);
```



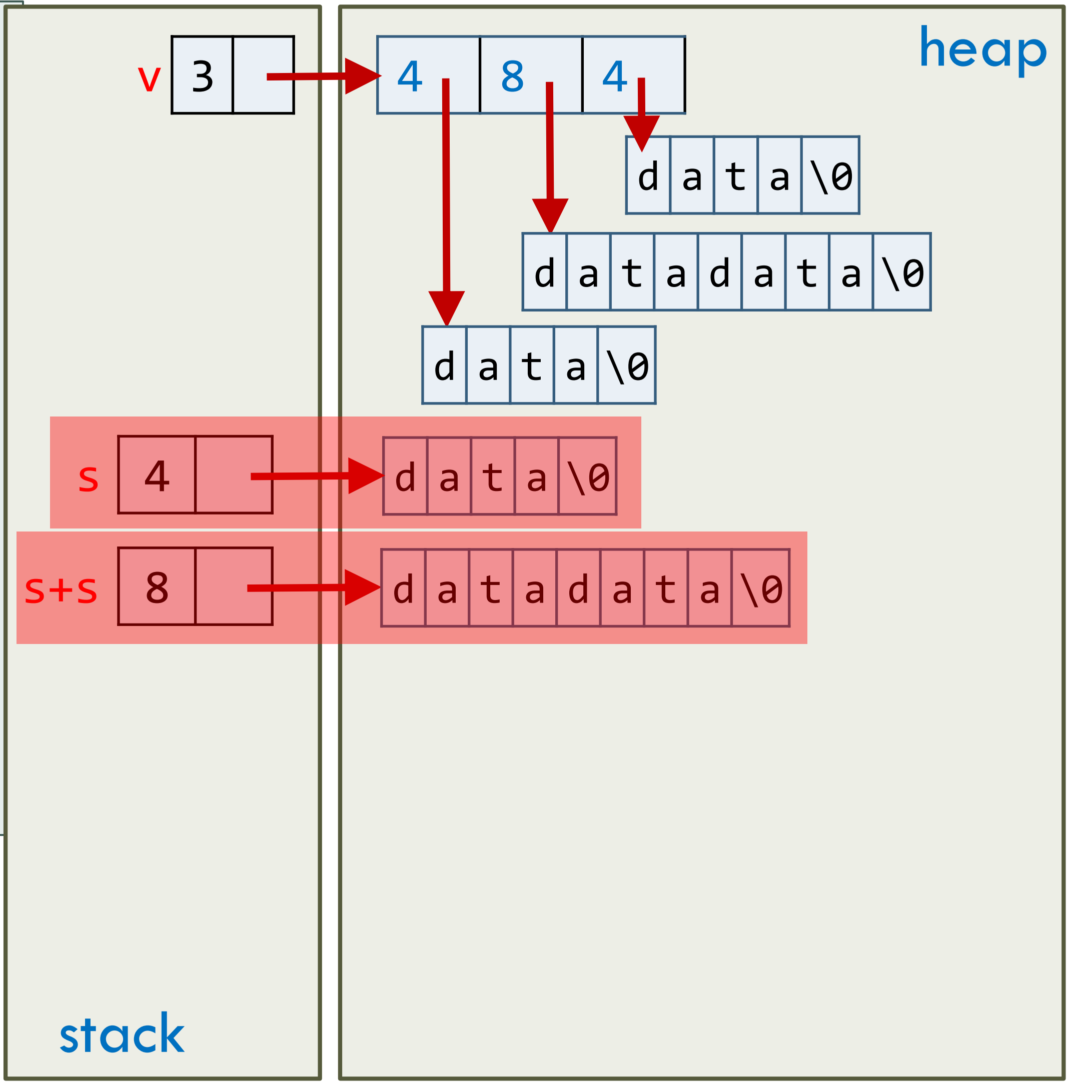
With RVO (7/8)

```
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 (8/8)

```
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);
```



URVO (1/2)

- □ Old technique available in many compilers since C++98 to elide copies when returning unnamed [that is, temporary] objects

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

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

```
// with URVO: 1 ctor
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
```

URVO (2/2)

- Always enabled in MSVC
- Always enabled in g++17 and beyond
- Can be disabled in pre-g++17 using `–fno-elide-constructors` option
- Remember not to have side effects in copy constructor and destructor because it will be elided in MSVC and g++-17 and beyond!!!

NRVO (1/2)

- □ Elide copies when returning named objects

```
// without NRVO: 3 ctors
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 copy ctor
  // using temporary from step 2
  Str s {nrvo("s")}; // copy ctor
                     // dtor from 2
} // 4) dtor for s
```

```
// with NRVO: 1 ctor
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
```

NRVO (2/2)

- MSVC doesn’t perform NRVO optimization without our help!!!
  - Can be enabled using optimization level `/O2`
- Enabled in g++11 and beyond
  - Can be disabled in g++11 using `-fno-elide-constructors` option
  - Cannot be disabled since g++17 …
- Remember not to have side effects in copy constructor because it will be elided!!!

Limitations: Multiple Return Paths (1/3)

- Compilers ~~may not~~ perform NRVO if different paths return different named objects
  - ~~Both g++17 and MSVC [even with /O2] cannot perform NRVO~~

```
// NRVO ~~not~~ possible ...
Str nrvo_ifelse(int x) {
  Str a{"a"}, b{"b"};
  if (x%2) {
    return a;
  } else {
    return b;
  }
}
```

Limitations: Multiple Return Paths (2/3)

□ NRVO ~~may be~~ possible if same named object is returned from different paths

- ~~MSVC [even with /02] cannot perform NRVO~~

```
// NRVO ~~not~~ possible ...
Str nrvo_ifelse(int x) {
  Str a{"a"}, b{"b"};
  if (x%2) {
    return a;
  } else {
    return b;
  }
}
```

```
// NRVO ~~may be~~ possible ...
Str nrvo_ifelse(int x) {
  Str a;
  // process a ...
  if (x%2) {
    a = "a";
    return a;
  } else {
    a = "b";
    return a;
  }
}
```

Limitations: Multiple Return Paths (3/3)

- Even better to have URVO by returning unnamed object is returned from different paths
  - Enabled in both g++17 and MSVC

```
// NRVO ~~may be~~ possible ...
Str nrvo_ifelse(int x) {
  Str a;
  // process a ...
  if (x%2) {
    a = "a";
    return a;
  } else {
    a = "b";
    return a;
  }
}
```

```
// URVO is always possible ...
Str nrvo_ifelse(int x) {
  if (x%2) {
    return Str{"a"};
  } else {
    return Str{"b"};
  }
}
```

Limitations: Assignments

- □ Copy elision performed only if return value is used as initializer for receiving variable
- □ Subtle logic error causes RVO to be disabled

```
// 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
```

```
Str urvo(char const *prc) {
  return Str{prc}; // 2) ctor
} // 3) assign 2) to 1)
  // 4) dtor for 2)

// URVO disabled ...
int main() {
  Str s{"a"}; // 1) ctor
  s = urvo("s");
} // 5) dtor for s
```

Copy Elision: Pass-By-Value

- Copy elision can also be performed when passing by value …

```
void foo(Str s) {
  std::cout << s;
} // 2) dtor for foo’s parameter

int main() {
  foo(Str()); // 1) ctor for foo’s parameter
 // other statements ...
}
```

C++'s Copy Problem: Revisit (1/2)

□ To avoid unnecessary copies, pass-by-reference becomes default mode of transferring resources to functions

```
~~void foo(X xx) {~~
void foo(X const &xx) {
  // use xx
}

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

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

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

C++'s Copy Problem: Revisit (2/2)

- □ To avoid unnecessary copy, pass by `const`-reference to callee
- □ Return by value from callee

```
~~void foo(X xx) {~~
void foo(X const &xx) {
  // use xx
}

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

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

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