Lvalue & Rvalue

What are lvalue and rvalue?

Compile the following code:

```
int foo() {
    return 2;
}
int main() {
    foo() = 2;
    return 0;
}
```

What are lvalue and rvalue?

Compile the following code:

```
int foo() {
    return 2;
}
int main() {
    foo() = 2;
    return 0;
}
```

You get the following error:

```
In function 'main': error: lvalue required as left operand of assignment
```

What are lvalue and rvalue?

And with the following code:

```
int& foo() {
    return 2;
}
int main() {
    foo() = 2;
    return 0;
}
```

What are lvalue and rvalue?

And with the following code:

```
int& foo() {
    return 2;
}
int main() {
    foo() = 2;
    return 0;
}
```

You get the following error:

```
function 'int& foo()': error: invalid
initialization of non-const reference of type
'int&' from an rvalue of type 'int'
```

A simple definition

- An *lvalue* (locator value) represents an object that occupies some identifiable location in memory (i.e. has an address)

- *rvalues* are defined by exclusion, by saying that every expression is either an *lvalue* or an *rvalue*. Therefore, from the above definition of *lvalue*, an *rvalue* is an expression that does not represent an object occupying some identifiable location in memory

Lvalue & Rvalue: definition by examples

```cpp
int a = 1;
a = 5; // Lvalue = Rvalue, Ok
a = a; // Lvalue = Lvalue, Ok
5 = a; // Rvalue = Lvalue Comp. error
5 = 5; // Rvalue = Rvalue Comp. error
(a+1) = 5; // Rvalue = Rvalue Comp. error
```

Lvalue & Rvalue: definition by examples

```
int a = 1;
int foo(int *input) { return *input; }


foo(&a); // both &a and foo(&a) are rvalues


int *p = &a;


foo(p) // now only foo(p) is rvalue,
       // p is an lvalue
```

Lvalue & Rvalue: definition by examples

```
int a = 1;
int& foo() { return a; }


foo() = 42;  // ok, foo() returns an lvalue


int* p1;
p1 = &foo(); // ok, foo() returns an lvalue


const int b = 1;
b = 2;          // ERROR: b is an lvalue
                // but it cannot be assigned
```

Lvalue & Rvalue

Lvalues: every value that has a name

- variables, references …

Rvalues: every value that has no name

- numbers, temporaries …

Temporary: a result of expression that isn't stored in a named variable.

Temporaries could be accessed only in the expression where they were created

- a+5 creates a temporary int with value 6

Conversions between lvalues and rvalues

lvalues can be converted to rvalues **implicitly**

```
int a = 1;      // a is an lvalue
int b = 2;      // b is an lvalue
int c = a + b; // + needs rvalues, so a and b
               // are converted to rvalues and
               // an rvalue is returned
```

Conversions between lvalues and rvalues

rvalues can be converted to lvalues **explicitly**

The unary '*' (dereference) operator can take an rvalue argument and produces an lvalue

```
int arr[] = {1, 2};
int* p = &arr[0];
*(p + 1) = 10; // OK: p + 1 is an rvalue, but
               // *(p + 1) is an lvalue
```

Conversions between lvalues and rvalues

The unary '&' (address-of) operator takes an lvalue argument and produces an rvalue

```
int var = 10;
int* bad_addr = &(var + 1); // ERROR: lvalue
            // required as unary '&' operand
int* addr = &var; // OK: var is an lvalue
&var = 40; // ERROR: lvalue required as left
           // operand of assignment
```

R/L value and References

non-const reference – only to a non const l-value.

const reference – to both l-value and r-value

```
int lv = 1;
const int clv = 2;
int& lvr1 = lv;
int& lvr2 = lv + 1; // error! why?
int& lvr3 = clv;    // error! why?
const int& cr1 = clv;
const int& cr2 = 5 + 5;
```

Rvalue reference
“move semantics”

C++ 11 feature

X&&

Rvalue reference

Rvalue refernce

Rvalue reference binds only to rvalue objects.

Syntax:

- `reference_type && reference_name`

Examples:

- `int && r1 = 3;`
- `int i=3, j=5;`
- `int && r2 = i+j;`
- `int && r3 = i; // error! (i is lvalue)`

Function returning rvalue

```
int foo(int* input)
{
  return *input;
}
int a = 3;
int& r = foo(&a); //error!
int&& r = foo(&a); //Ok!
```

rvalue-lvalue function overloading

```
void foo( int& input ){…}
void foo( int&& input ){…}

int main()
{
   int i = 3;
   foo(i);  // will call the first foo
   foo(3);  // will call the second foo
```

std::move

```cpp
#include <utility>
void foo( int& input){…}
void foo( int&& input){…}

int main()
{
   int i = 3;
   foo(std::move(i));// will call the second foo
   foo(3);           // will call the second foo
```

So we have the power to treat named variables and temporaries differently

➔ let’s see why this is useful and why it’s probably the most important new feature in C++11

Let’s look at the following string class

```
#ifndef MYSTRING_H
#define MYSTRING_H

class MyString
{
public:

   // Constructors
   MyString(const char* str );
   MyString(const MyString& o);
   // Destructor
   ~MyString();

private:

   int   _length;
   char* _string;
};

#endif
```

```
#include “MyString.h”

MyString::MyString(const char *str)
{
   _length = strlen(str)+1;
   _string = new char[_length];
   strcpy(_string, str);
}

MyString::MyString(const
                   MyString& o)
   : MyString(o._string)
{}


MyString::~MyString() {
   delete[] _string;
}
```

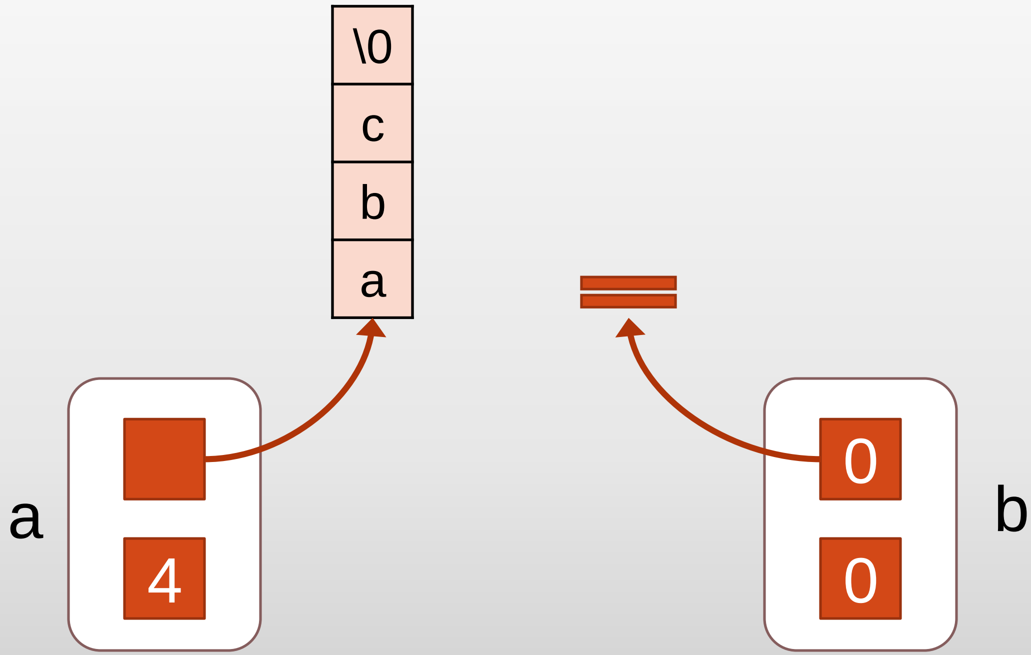
Move semantics - motivation

```
MyString returnString(MyString input){
  MyString output(input);
  return output;
}
(1) MyString a("hello");      // regular ctor
(2) MyString b(a);               // copy ctor
(3) MyString c(returnString(a)); // copy ctor
```

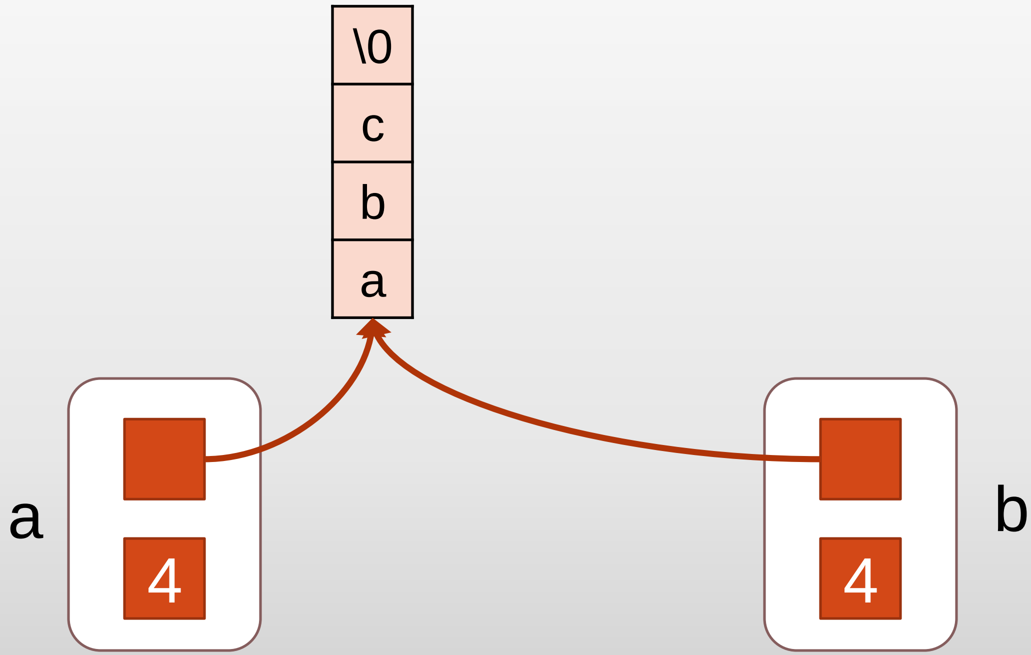
We could make (3) much more efficient if we use shallow copy there!!

Different Copy Diagrams

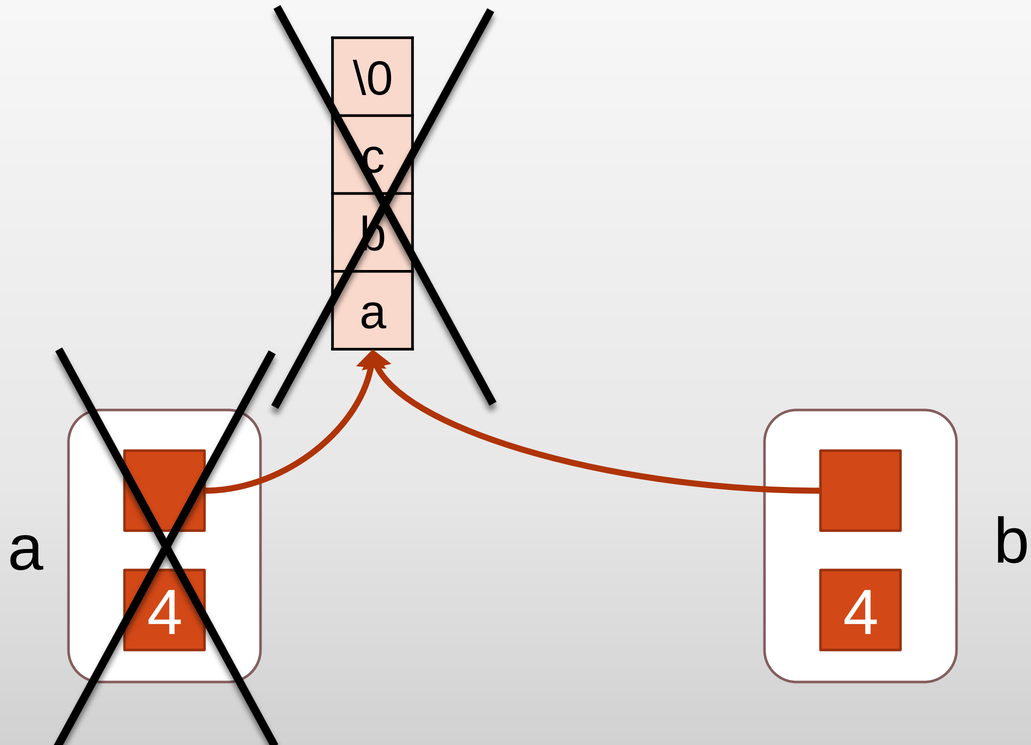
Shallow Copy A to B (1)



Shallow Copy A to B (2)

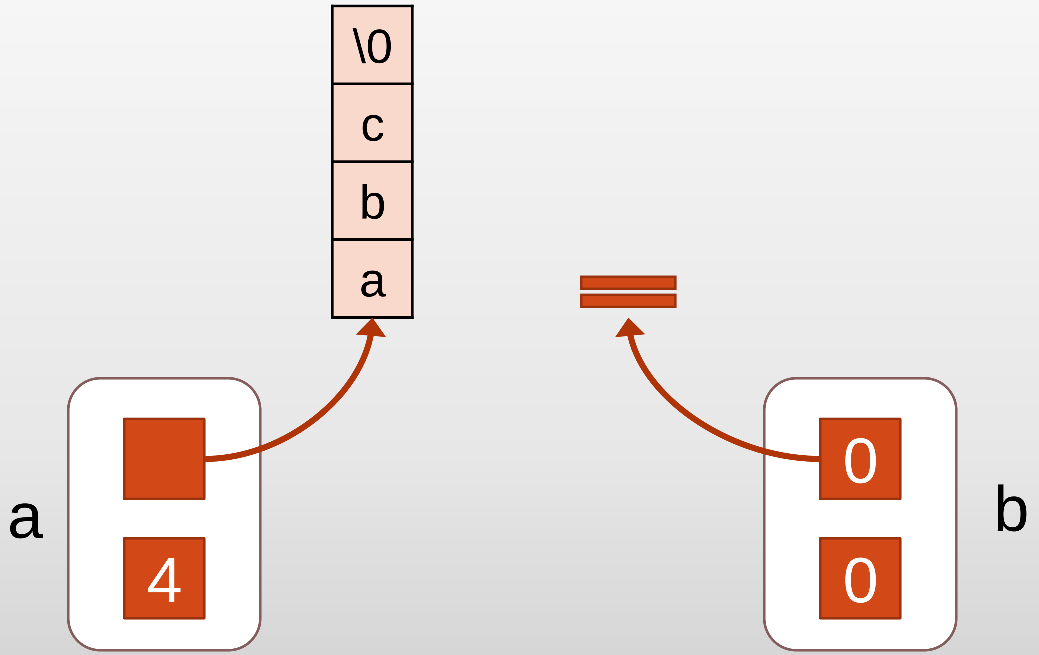


Shallow Copy A to B (3)

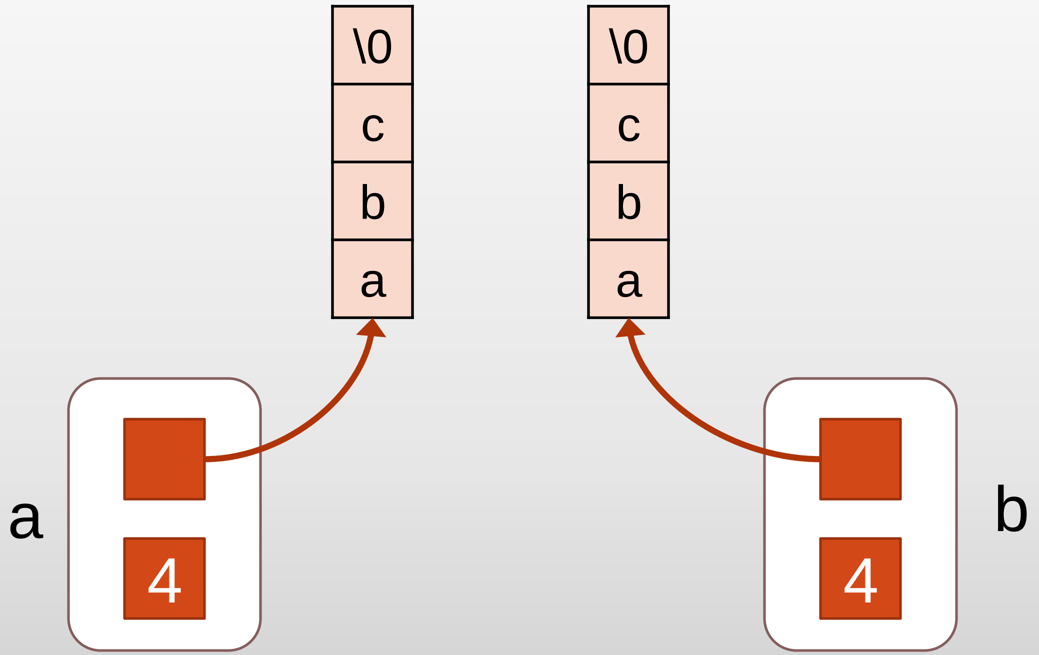


This is the default copy constructor – and you can see why in this case we should either not allow it, or implement it differently (next slide…)

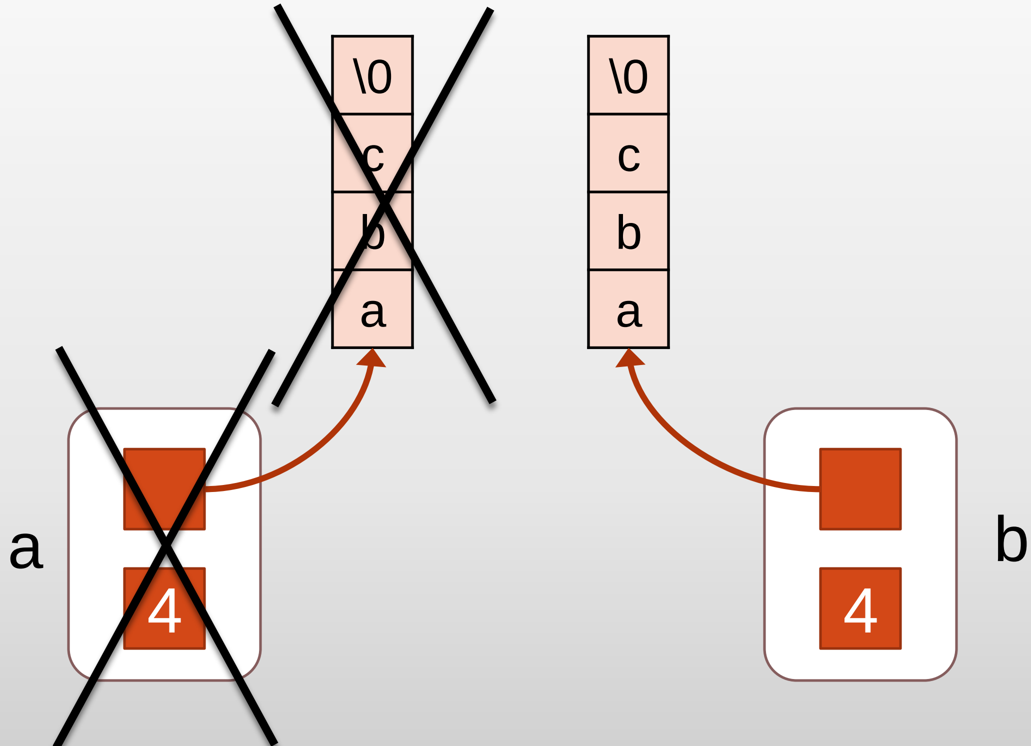
Deep Copy A to B (1)



Deep Copy A to B (2)

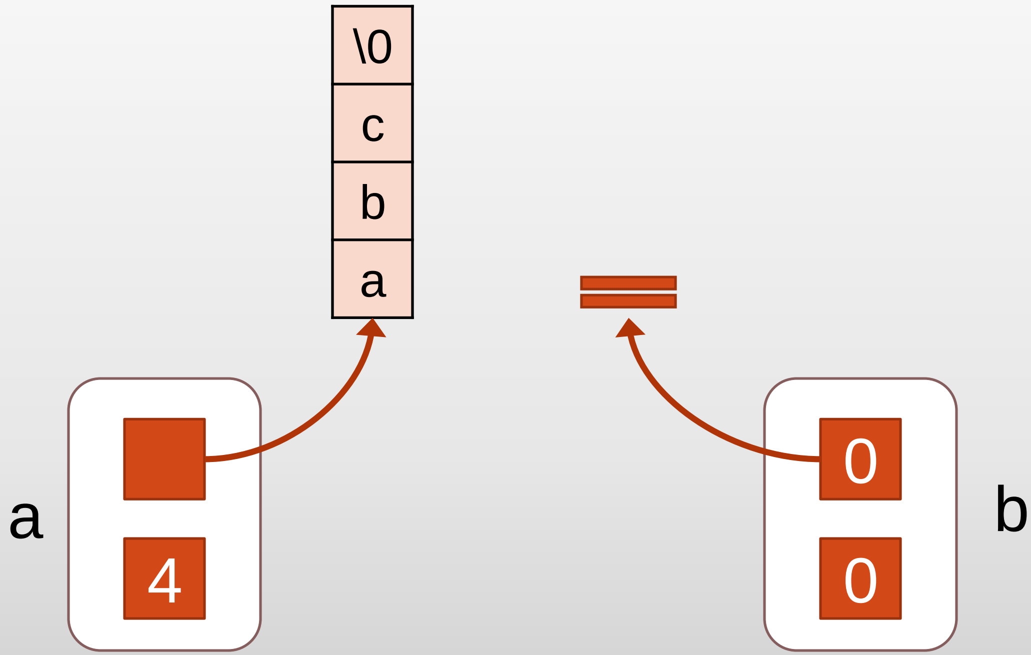


Deep Copy A to B (3)

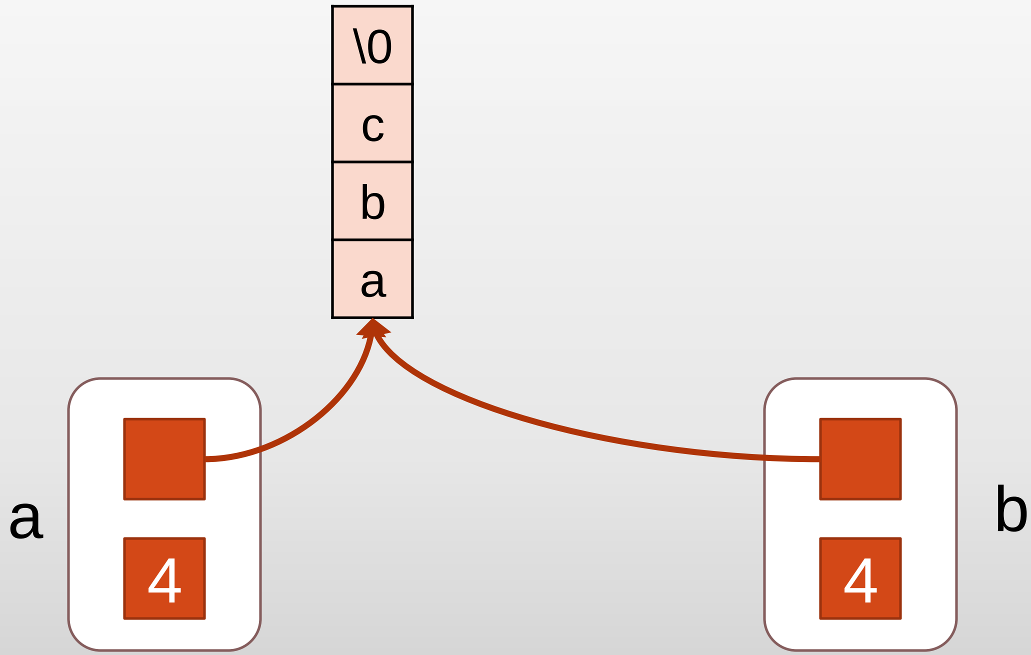


Now if ‘a’ changes or dies, ‘b’ is not affected

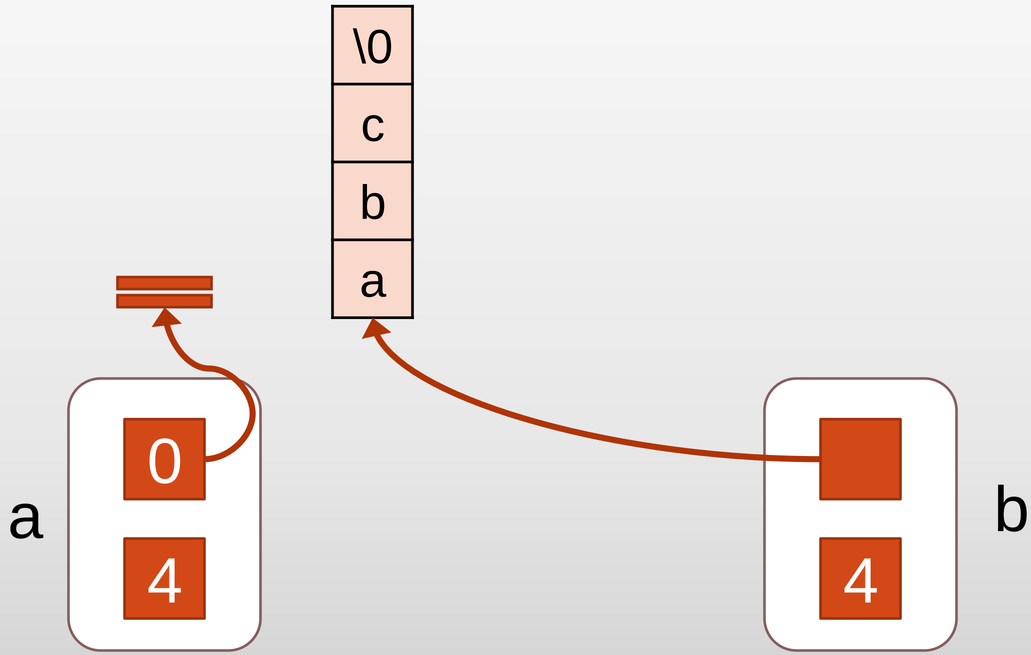
Move (temporary) A to B (1)



Move (temporary) A to B (2)

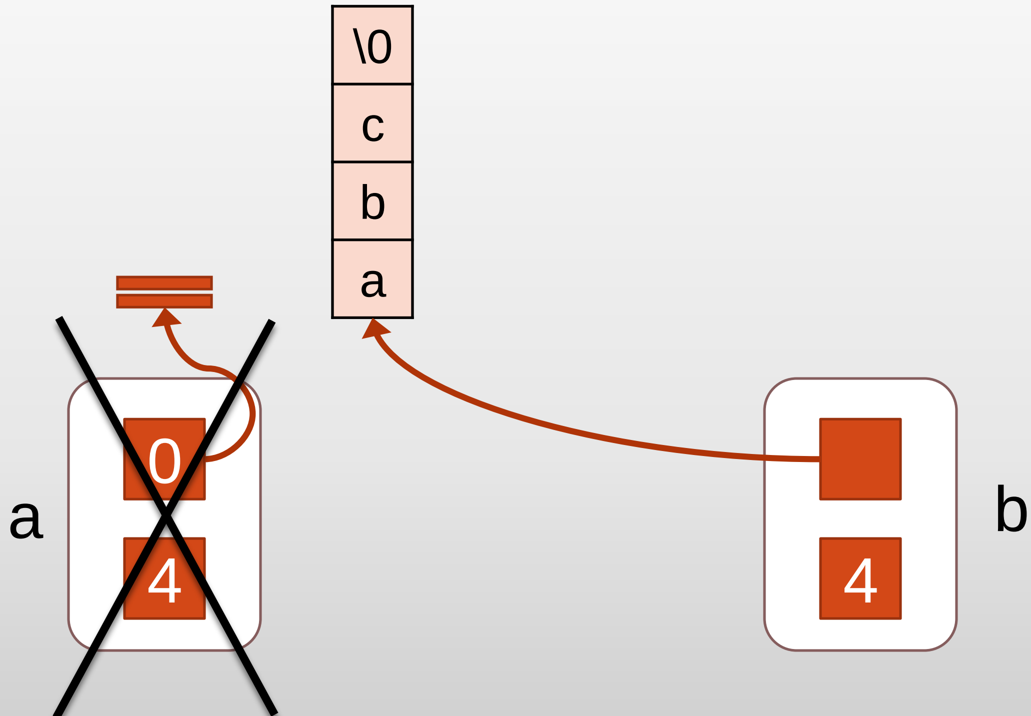


Move (temporary) A to B (3)



Smart shallow copy: ‘b’ steals ‘a’ internals, but leave it in a well defined state, so a later call to the destructor won’t become a problem. ‘a’ now is free to die peacefully

Move (temporary) A to B (4)



Smart shallow copy: ‘b’ steals ‘a’ internals, but leave it in a well defined state, so a later call to the destructor won’t become a problem. ‘a’ now is free to die peacefully

Move and Copy

```
MyString(const MyString& other); // copy constructor
MyString(MyString&& other); // move constructor
MyString returnString(MyString input) // returns some string

MyString::MyString(MyString&& other) // move ctor implementation
{
   _string = other._string;
   other._string = nullptr;
   _length = other._length;
}

int main {
  MyString a(“abc”); // regular ctor
  MyString b(a); // copy
  MyString c(std::move(a)); // move (swaps/steals)
  MyString d(returnString(b)); // move (swaps/steals)
}
```

Move operator=

```
MyString& operator=(MyString&& other)
{
  if (this != &other)
  {
    delete[] _string;
    _string = other._data;
    other._string = nullptr;
    _length = other._length;
  }
  return *this;
}
```

Calling operator=

```
MyString returnString(MyString& input){
  MyString output(input);
  return output;
}
MyString a(“hello”), b(“world”), c(“labcpp”);

(2) b = a; // calls the regular op=

(3) c = returnString(a); // calls the move op=
```

The Rule of Five

So now, the rule of three becomes the rule of five

If we implement one of the following, and want to allow move semantics, we probably need to implement all of them:

- the destructor
- copy constructor
- copy assignment operator
- move constructor
- move assignment operator

Read more here:

http://en.cppreference.com/w/cpp/language/rule_of_three

When is the (copy c-tor || move c-tor) executed?

- Explicit call
- Parameter passed to function by value
- Parameter passed to function by reference
- Return value from function by value
- Return value from function by reference
- In the command MyClass a=b;
(b is also of type MyClass)

Copy and Swap idiom

Read more at:
http://stackoverflow.com/a/3279550/2586599

(by GManNickG)

How can we implement the Big Five [destructor, copy constructor, copy assignment operator, move constructor, move assignment operator]

The *copy-and-swap idiom* is an elegant solution, assisting us to achieve two things:

- avoiding code duplication
- providing a strong exception guarantee

First implement the following: Copy Ctor, Move Ctor, and Dtor

This should be straight forward…

(see the code in previous slides)

Then implement **one** operator=

We'll see now how…

Implementing operator= (failed solution)

```
MyString& operator=(const MyString& other) {
  if (this != &other)
  {
    delete[] _string;
    _string = nullptr;
    _length = other._length;
    _string = _length ? new char[_length] : nullptr;
    strcpy(_string, other._string);
  }
  return *this;
}
```

Implementing operator= (failed solution)

```
MyString& operator=(const MyString& other) {
  if (this != &other) // usually unnecessary
  {
    delete[] _string; // what if new below throws?
    _string = nullptr;
    _length = other._length; // copy ctor code dup
    _string = _length ? new char[_length] : nullptr;
    strcpy(_string, other._string);
  }
  return *this;
}
```

Implementing operator= (partial fix)

```
MyString& operator=(const MyString& other) {
if (this != &other) // usually unnecessary
{ // get the new data ready before replacing
  int newLen = other._length; // copy ctor code dup
  char* newStr = newLen ? new char[newLen] : nullptr;
  strcpy(newStr, other._string);
  // now replace
  delete[] _string;
  _length = newLen;
  _string = newStr;
}
return *this;
}
```

Implementing operator= (partial fix)

```
MyString& operator=(const MyString& other) {
  // get the new data ready before replacing
  int newLen = other._length; // copy ctor code dup
  char* newStr = newLen ? new char[newLen] : nullptr;
  strcpy(newStr, other._string);
  // now replace
  delete[] _string;
  _length = newLen;
  _string = newStr;

  return *this;
}
```

Implementing operator= (better solution)

```
MyString& operator=(MyString other) {
    swap(*this, other);
    return *this;
}
```

The swap function

```
friend void swap(MyString& first, MyString& second)
{
    using std::swap;
    // by swapping the members of two classes,
    // the two classes are effectively swapped
    swap(first._length, second._length);
    swap(first._string, second._string);
}
```

Implementing operator= with the "copy and swap" idiom

```
MyString& operator=(MyString other) {
    swap(*this, other);
    return *this;
}
```

- Receives the parameter by value
  - Calls copy ctor for lvalue arguments
  - Calls move ctor for rvalue arguments
- Clean up is implicit (no leaks and no checks)
  - What if `_string` is null?
  - What if `_string` points to allocated space?
- No need to check self-assignment (why?)
- Better exception safety (later)
- Need to implement a swap function

Copy elision & Return Value Optimization (RVO)

RVO – return value optimization

How many MyString objects will be created (and how)?

```
MyString foo(MyString str) {
    return str;
}
int main () {
    MyString str1("Paul"), str2;
    str2 = foo(str1);
    MyString str3 = foo(str2);
}
```

5-7

```
1: 1 char* ctor and 1 default ctor
2: 1/2 copy ctor (+1 operator=)
3: 1/2/3 copy ctor
(cannot know for sure because of RVO)
```

```
MyString foo(MyString str) {
    return str;
}
int main () {
    1 MyString str1("Paul"), str2;
    2 str2 = foo(str1);
    3 MyString str3 = foo(str2);
}
```

example

```
class A
{
public:
    A(int k):i(k) {};
    A(const A& other)
    {
        cout << "copy c-tor for "
             << other.i << endl;
        i=other.i;
    }
    ~A()
    {
        cout << "I'm being destructed - “
             << i << endl;
    }
    A& operator=(const A& other)
    {
        i=other.i;
        cout << "in operator = " << endl;
        return *this;
    }
private:
    int i;
};
```

Note!
No printing here

```
A foo(A m) {
    A x(9);
    cout << "before return“ << endl;
    return x;
}
void main() {
    A m(2);
    A b=foo(m);
    cout << "after func" << endl;
}
```

With “-fno-elide-constructors”

```
class A
{
public:
   A(int k):i(k) {};
   A(const A& other)
   {
      cout << "copy c-tor for "
           << other.i << endl;
      i=other.i;
   }
   ~A()
   {
      cout << "I'm being destructed - “
           << i << endl;
   }
   A& operator=(const A& other)
   {
      i=other.i;
      cout << "in operator = " << endl;
      return *this;
   }
private:
   int i;
};
```

```
A foo(A m) {
   A x(9);
   cout << "before return“ << endl;
   return x;
}
void main() {
   A m(2);
   A b=foo(m);
   cout << "after func" << endl;
}
```

```
copy c-tor for 2
before return
copy c-tor for 9
I'm being destructed – 9
copy c-tor for 9
I'm being destructed – 9
I'm being destructed – 2
after func
I'm being destructed – 9
I'm being destructed - 2
```

With "-fno-elide-constructors"

```
class A
{
public:
   A(int k):i(k) {};
   A(const A& other)
   {
      cout << "copy c-tor for "
           << other.i << endl;
      i=other.i;
   }
   ~A()
   {
      cout << "I'm being destructed - "
           << i << endl;
   }
   A& operator=(const A& other)
   {
      i=other.i;
      cout << "in operator = " << endl;
      return *this;
   }
private:
   int i;
};
```

```
A foo(A m) {
   A x(9);
   cout << "before return" << endl;
   return x;
}
void main() {
   A m(2);
   A b=foo(m);
   cout << "after func" << endl;
}
```

```
copy c-tor for 2
before return
copy c-tor for 9
I'm being destructed - 9
copy c-tor for 9
I'm being destructed - 9
I'm being destructed - 2
after func
I'm being destructed - 9
I'm being destructed - 2
```

Can be optimized

With optimization

```
class A
{
public:
   A(int k):i(k) {};
   A(const A& other)
   {
      cout << "copy c-tor for "
           << other.i << endl;
      i=other.i;
   }
   ~A()
   {
      cout << "I'm being destructed - “
           << i << endl;
   }
   A& operator=(const A& other)
   {
      i=other.i;
      cout << "in operator = " << endl;
      return *this;
   }
private:
   int i;
};
```

```
A foo(A m)
{
   A x(9);
   cout << "before return“ << endl;
   return x;
}


void main()
{
   A m(2);
   A b=foo(m);
   cout << "after func" << endl;
}
```

```
copy c-tor for 2
before return
I'm being destructed – 2
after func
I'm being destructed – 9
I'm being destructed - 2
```

0/1/2 ctors calls

With copy elision and passing temporary

```
class A
{
public:
   A(int k):i(k) {};
   A(const A& other)
   {
      cout << "copy c-tor for "
           << other.i << endl;
      i=other.i;
   }
   ~A()
   {
      cout << "I'm being destructed - “
           << i << endl;
   }
   A& operator=(const A& other)
   {
      i=other.i;
      cout << "in operator = " << endl;
      return *this;
   }
private:
   int i;
};
```

```
A foo(A m)
{
   A x(9);
   cout << "before return“ << endl;
   return x;
}


void main()
{
   A b=foo(A(2));
   cout << "after func" << endl;
}
```

With copy elision and passing temporary

```
class A
{
public:
   A(int k):i(k) {};
   A(const A& other)
   {
      cout << "copy c-tor for "
           << other.i << endl;
      i=other.i;
   }
   ~A()
   {
      cout << "I'm being destructed - “
           << i << endl;
   }
   A& operator=(const A& other)
   {
      i=other.i;
      cout << "in operator = " << endl;
      return *this;
   }
private:
   int i;
};
```

```
A foo(A m)
{
   A x(9);
   cout << "before return“ << endl;
   return x;
}


void main()
{
   A b=foo(A(2));
   cout << "after func" << endl;
}
```

```
before return
I'm being destructed – 2
after func
I'm being destructed – 9
```

With copy elision and passing temporary

```
class A
{
public:
   A(int k):i(k) {};
   A(const A& other)
   {
      cout << "copy c-tor for "
           << other.i << endl;
      i=other.i;
   }
   ~A()
   {
      cout << "I'm being destructed - “
           << i << endl;
   }
   A& operator=(const A& other)
   {
      i=other.i;
      cout << "in operator = " << endl;
      return *this;
   }
private:
   int i;
};
```

```
A foo(A m)
{
   A x(9);
   cout << "before return“ << endl;
   return x;
}


void main()
{
   A b=foo(A(2));
   cout << "after func" << endl;
}
```

```
before return
I'm being destructed – 2
after func
I'm being destructed – 9
```

No copy ctor calls at all

But can the compiler just skip functions

- What if we print something in the copy constructor?

- What if we do something even more important?

RVO - Return Value Optimization

The C++ standard allows a compiler to perform any optimization, as long as the resulting executable exhibits the **same observable behaviour *as if* all the requirements of the standard have been fulfilled**

However:
RVO is particularly notable in C++ for being allowed to **change the observable behavior** of the resulting program!

- Supported on most compilers
  - Sometimes configurable
- Don't put important logic in copy/move constructors or destructors
- You need to know the standard
  - That will always be a starting point to understand what is going on

RVO and move constructor

- Copy elision can also omit move constructor calls
- To enable RVO or move constructor be applied for returned values, we should **return non-const objects when returning by value**

RVO and move constructor

There are cases where the compiler cannot use RVO, e.g:

```
#include <string>
std::string f(bool cond = false) {
  std::string first("first");
  std::string second("second");
  return cond ? first : second;
}
int main() {
  std::string result = f();
}
```

In those cases, having a move constructor invoked instead of copy constructor can increase performance

Common forms of copy elision

C++

named return value optimization

```
Thing f() {
   Thing t;
   return t;
}
Thing g = f();
```

return value optimization

```
Thing f() {
   return Thing();
}
Thing g = f();
```

temporary is passed by value

```
void f(Thing t) {...}
f(Thing());
```

exception is thrown and caught by value

```
int main() {
   try {
      Thing t;
      throw t;
   } catch (Thing g) {
   }
}
```