Understanding &&

By Scott Schurr for C++ Now 2014

Disclaimer

- These slides are based on C++11 and C++14
- Future revisions of the standard may render these slides obsolete

Helpers

```
#include <type_traits>          // numerous helpers
#include <cassert>              // assert
#include <string>               // std::string
#include <vector>               // std::vector
#include <memory>               // std::unique_ptr<>


#define STATIC_ASSERT(...) \
      static_assert(__VA_ARGS__, #__VA_ARGS__)
```

Topics

- **Move Motivation and Background**
- Implementing Move
- Universal References and Perfect Forwarding
- Overloading With Universal References
- Summary

Mooove Motivation and Background

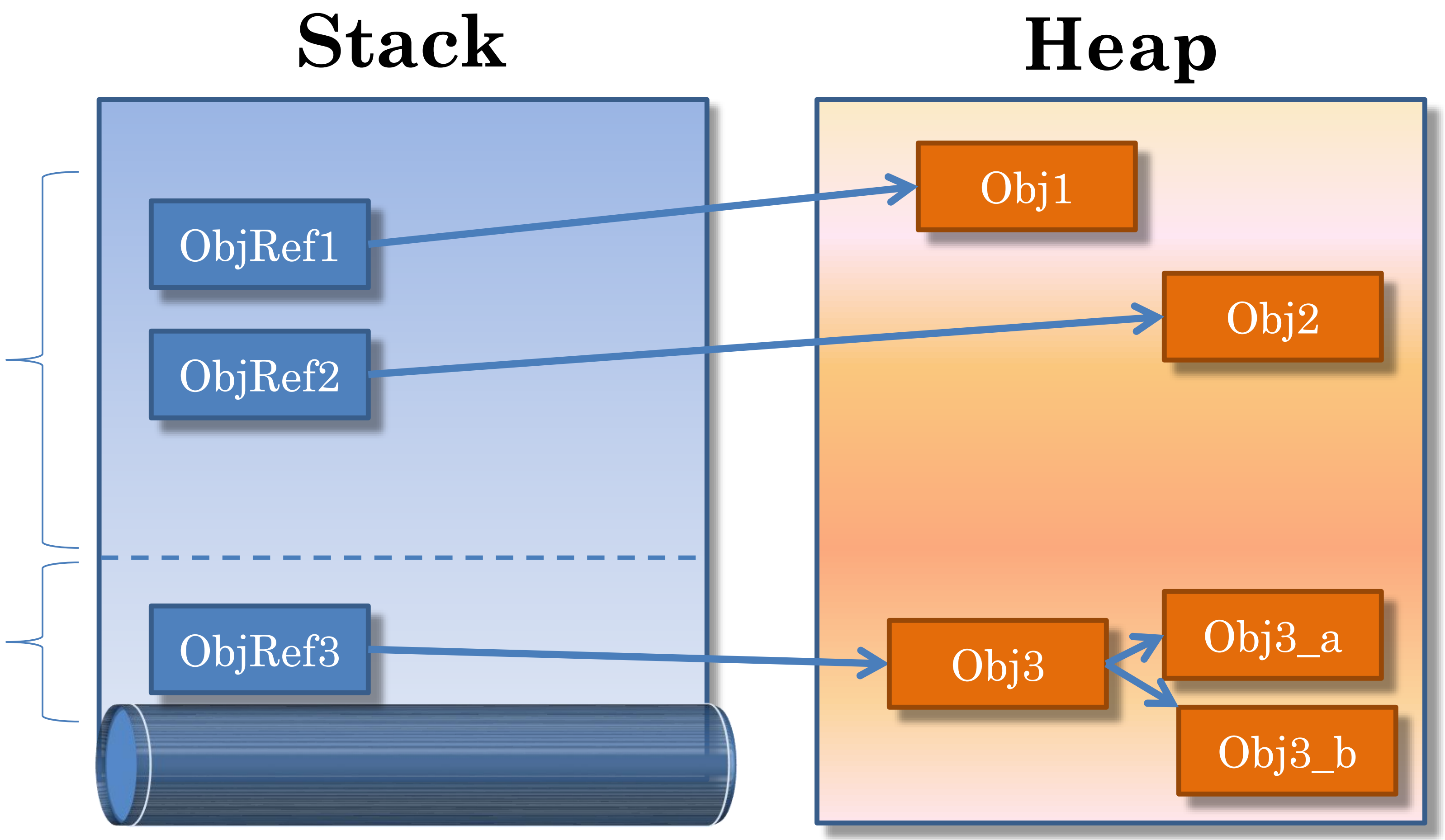
Why Move Semantics?

- Primarily to improve performance in specific, important cases
- Secondarily allows non-copyable types to move from one scope to another
  - E.g, return an fstream from a function

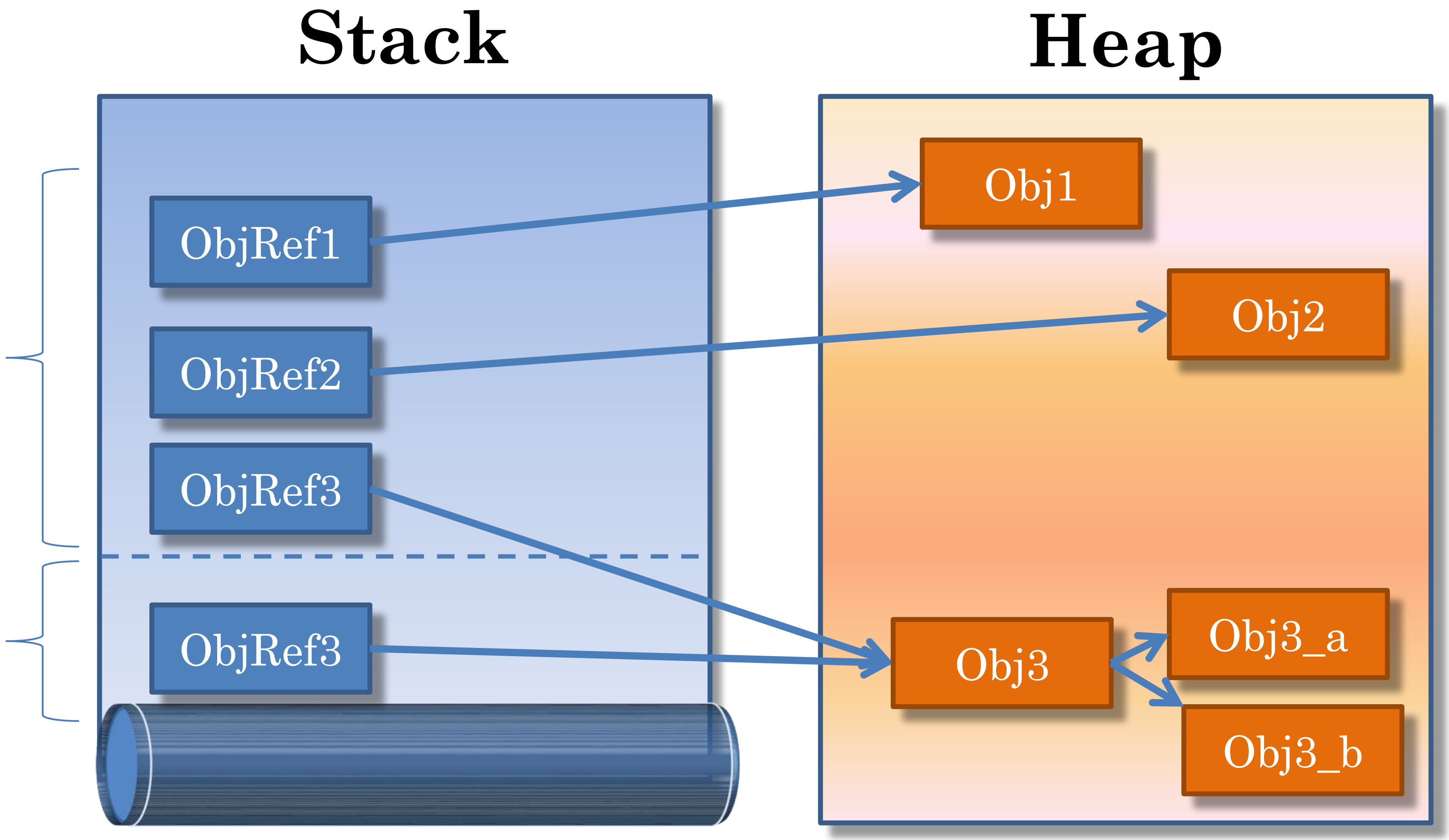
Performance in C++98

- Java released in 1995
- Java was generally slower than C++98
- Java was faster when passing big objects
- Why?

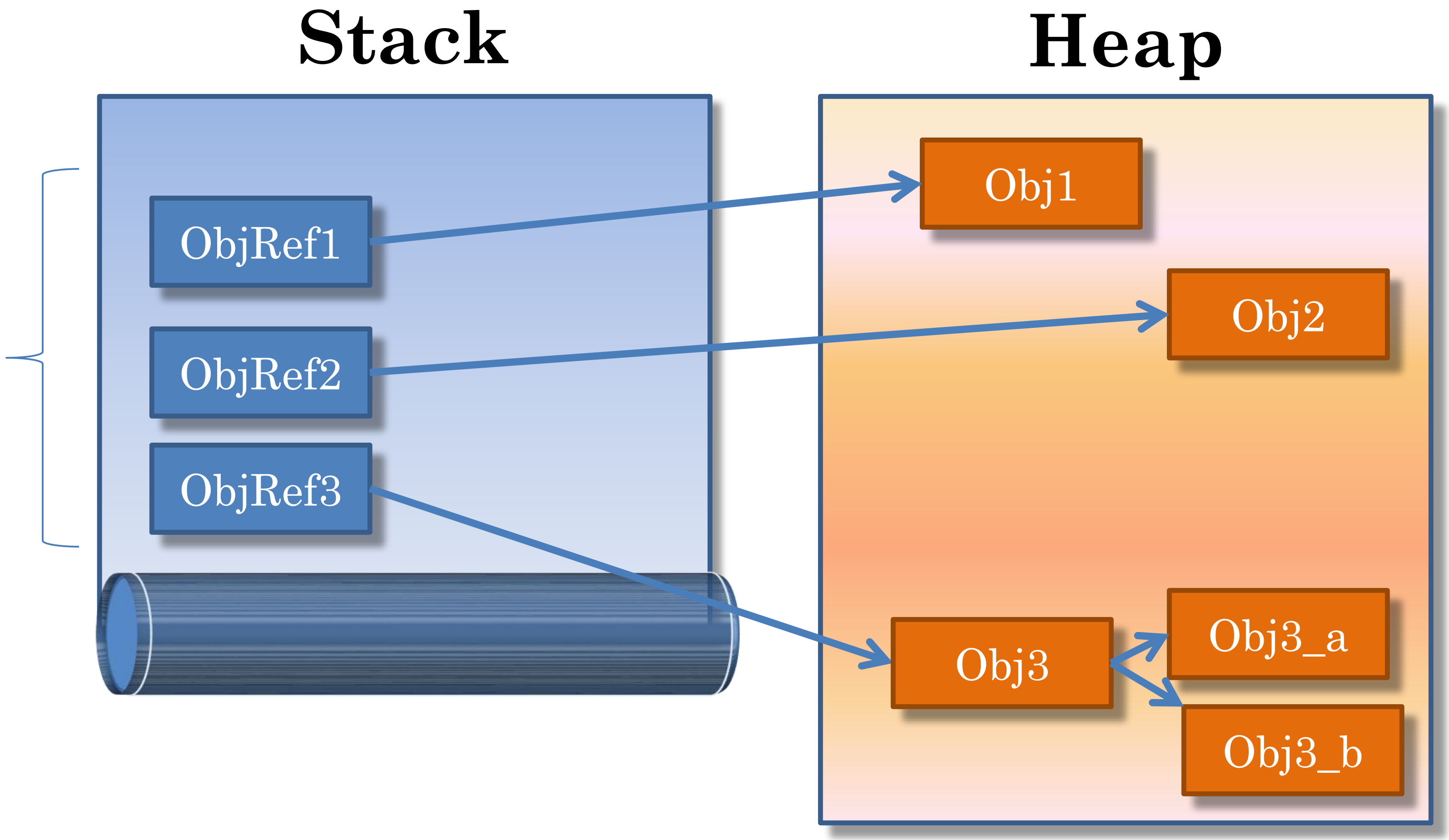
Local Java Objects



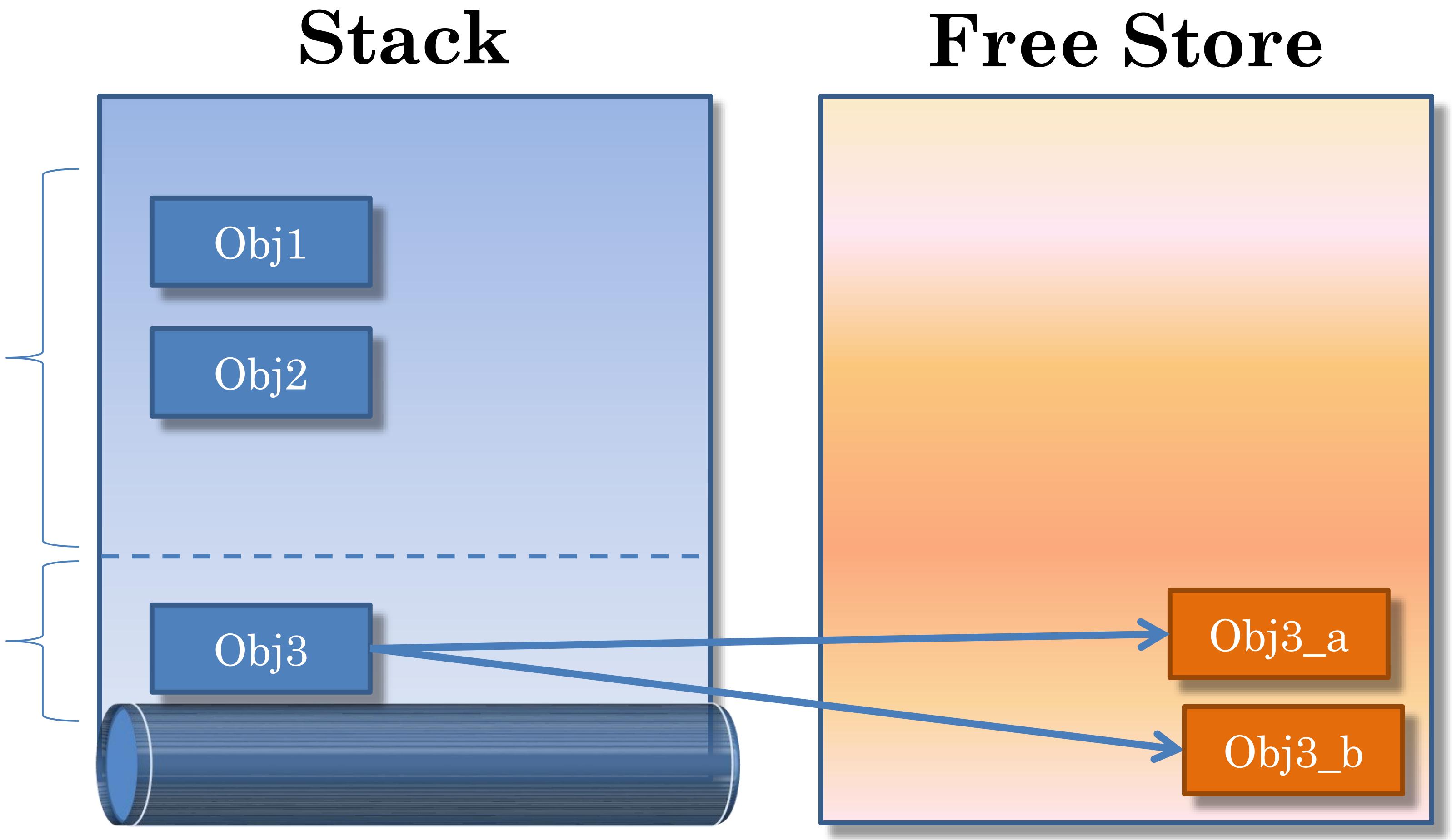
Returning a Java Object pt 1



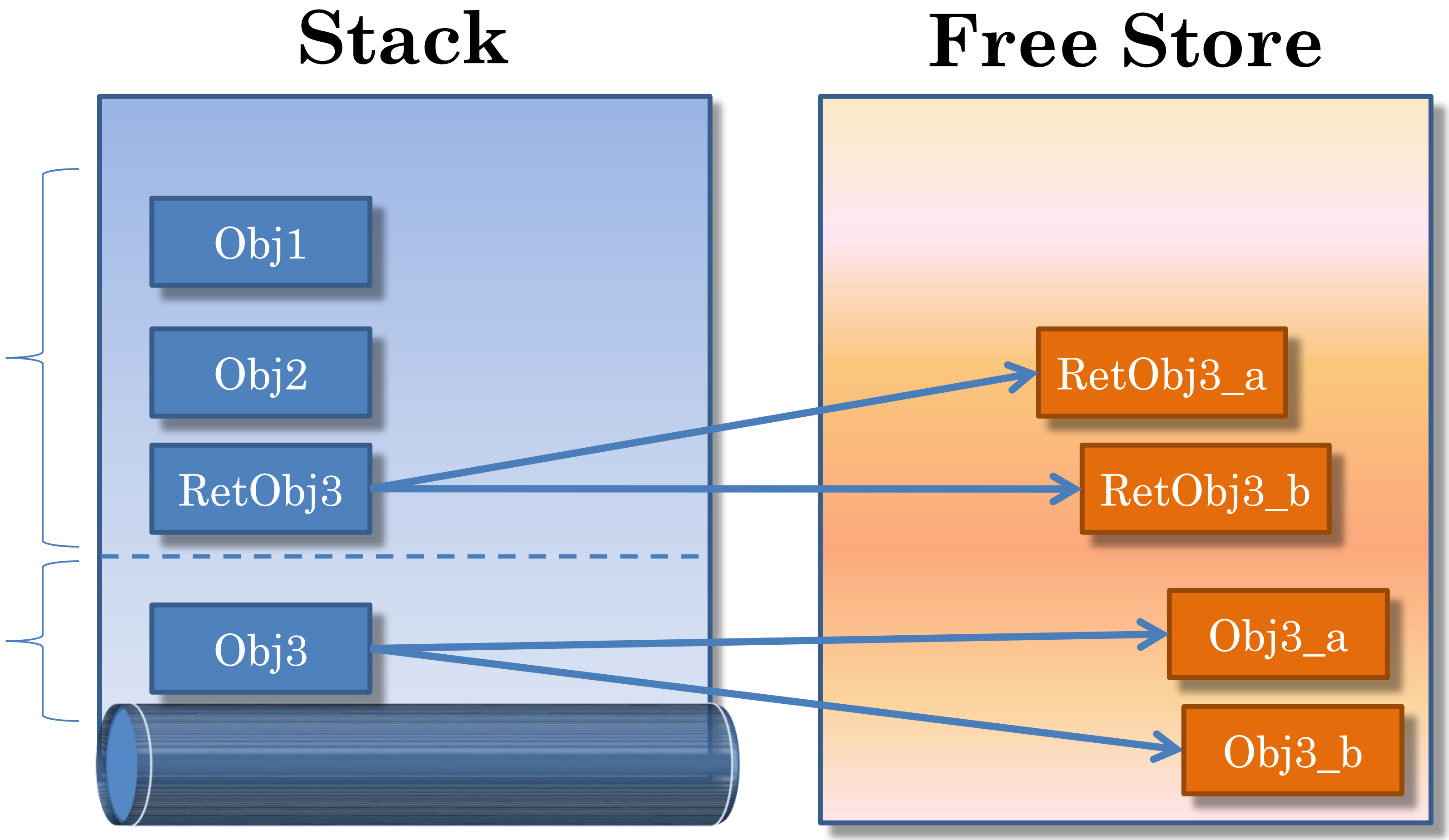
Returning a Java Object pt 2



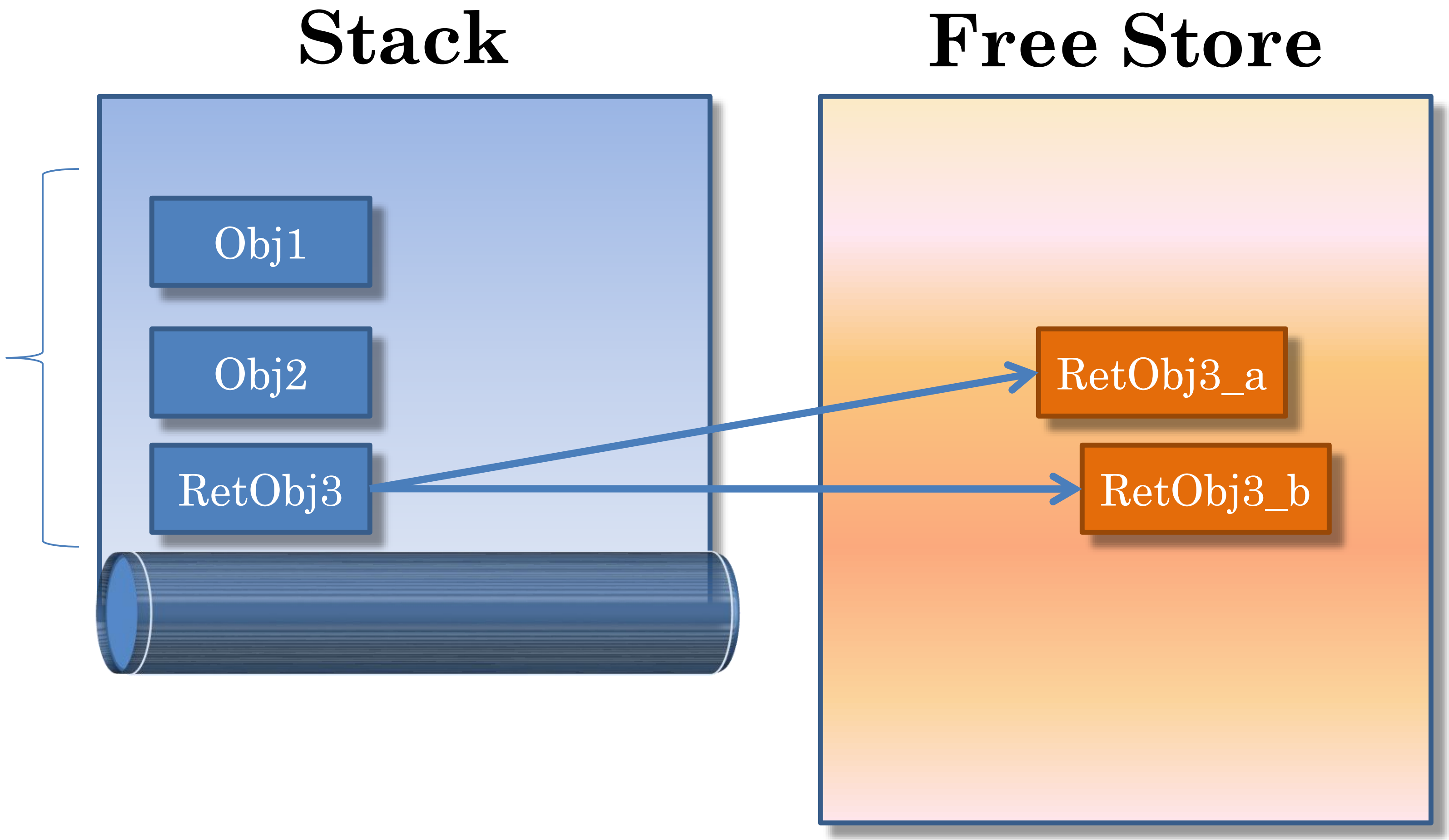
Local C++ Objects



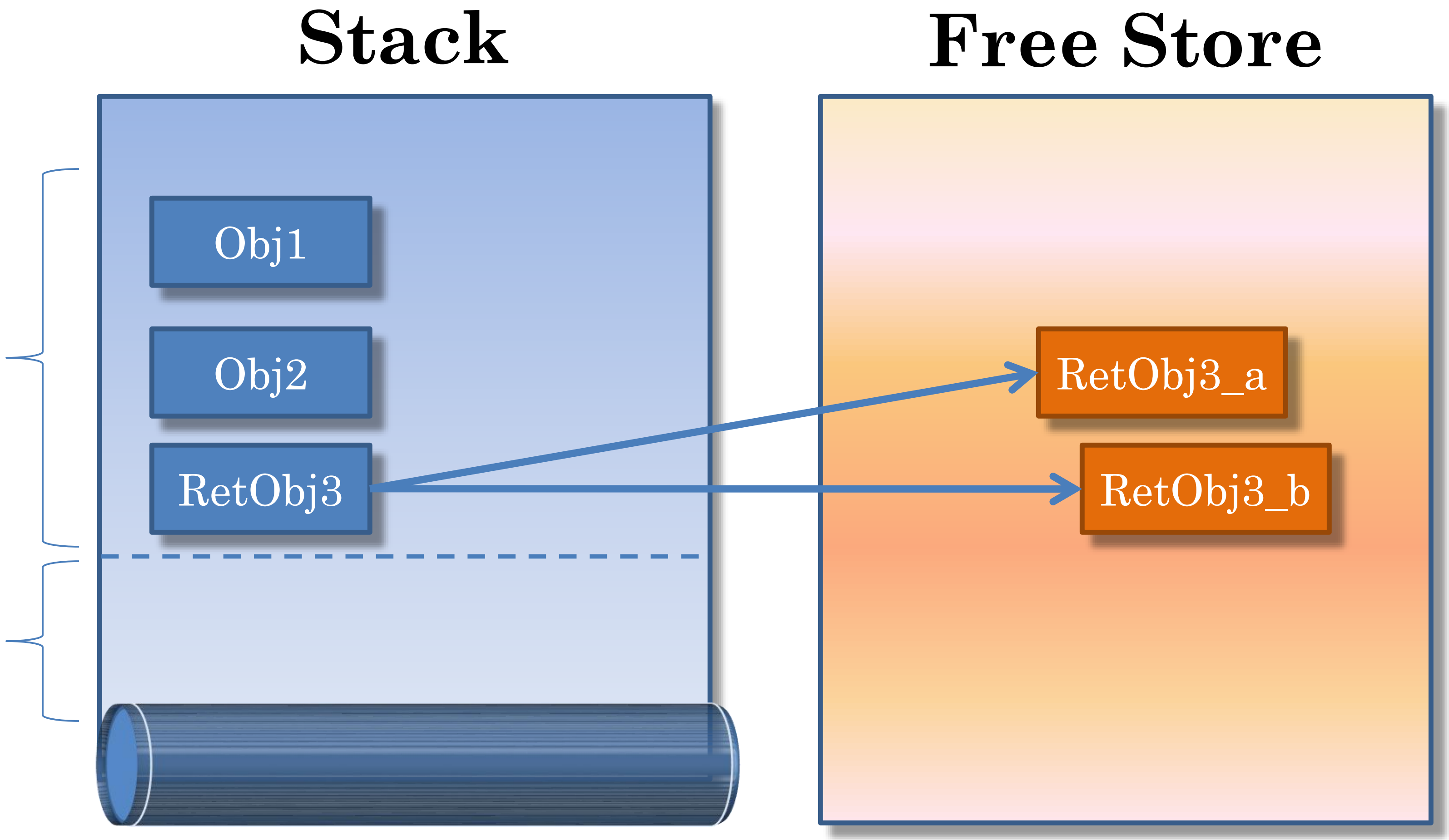
Returning a C++89 Object pt 1



Returning a C++89 Object pt 2



C++98 Return Value Optimization (RVO)

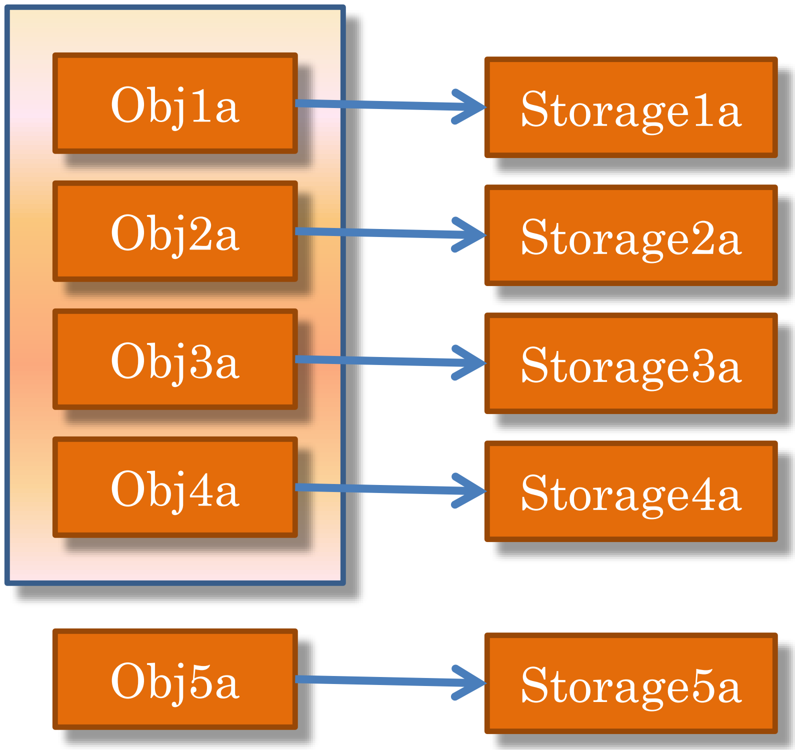


RVO Limits

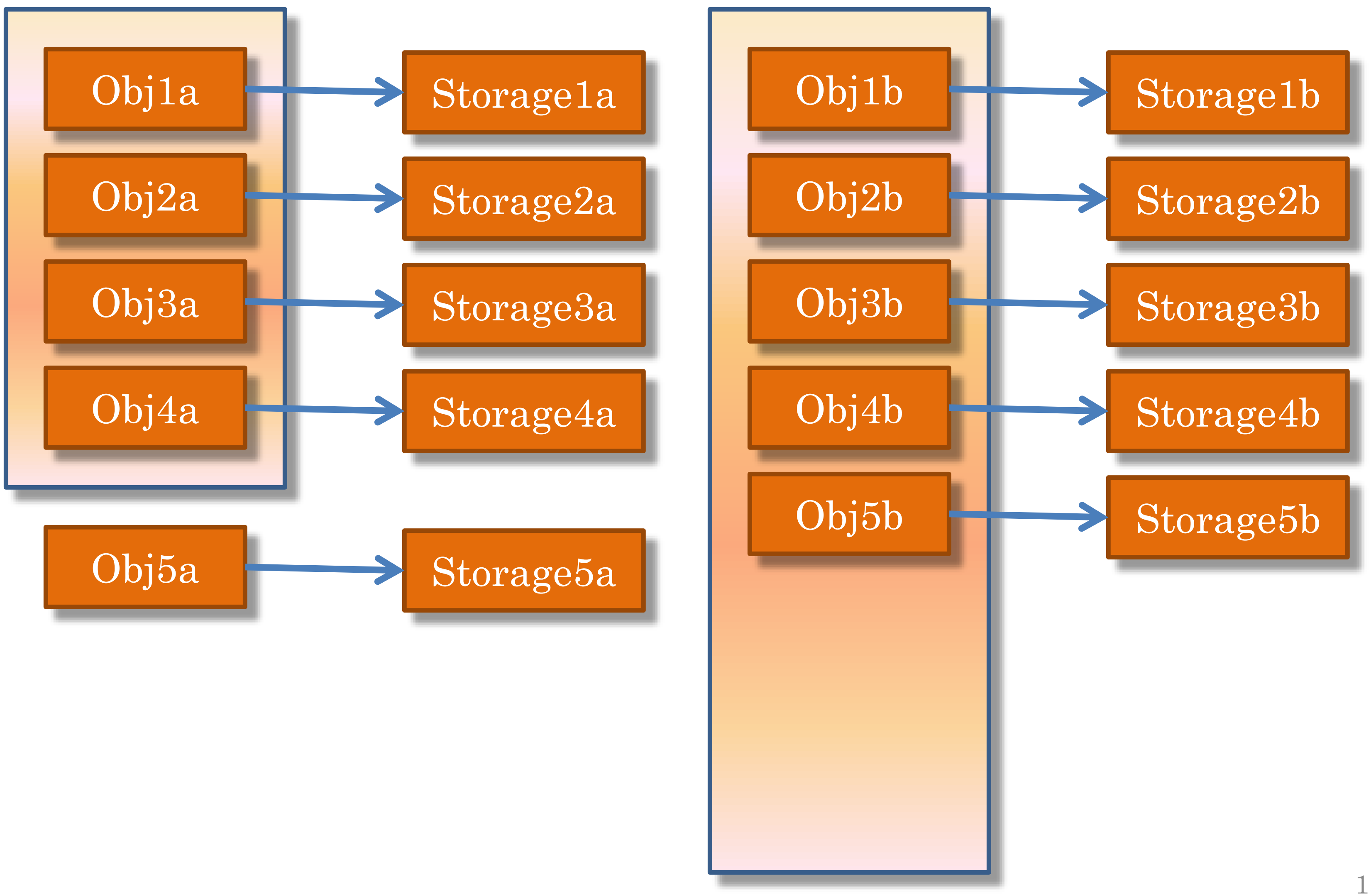
RVO does not work for conditional returns

```
std::vector<std::string> retStringVec(int w)
{
    std::vector<std::string> vA;
    std::vector<std::string> vB;
    if (w > 0) {
        return vA;
    }
    return vB;
}
```

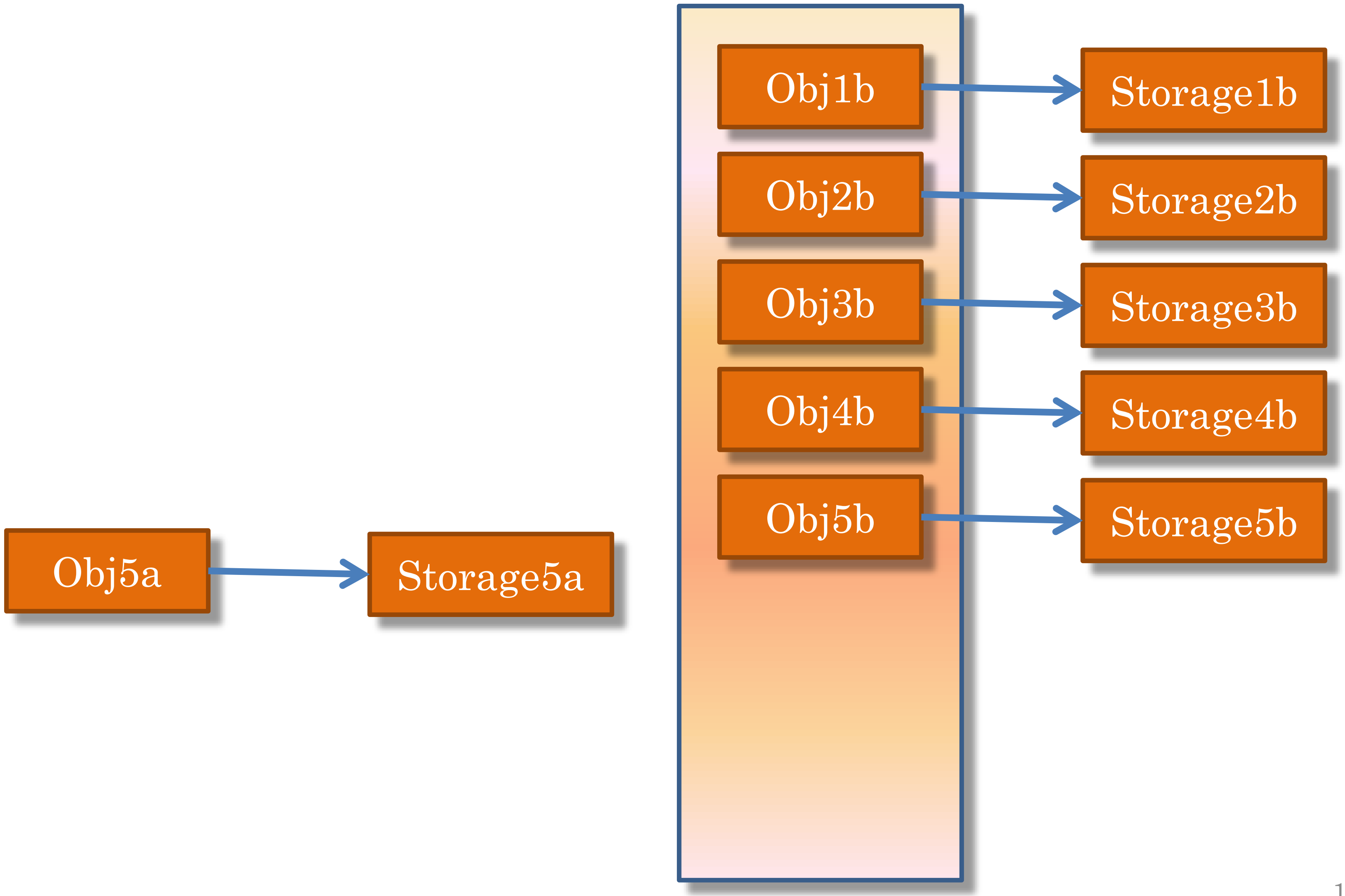
C++98 Vector push_back



C++98 Vector push_back



C++98 Vector push_back



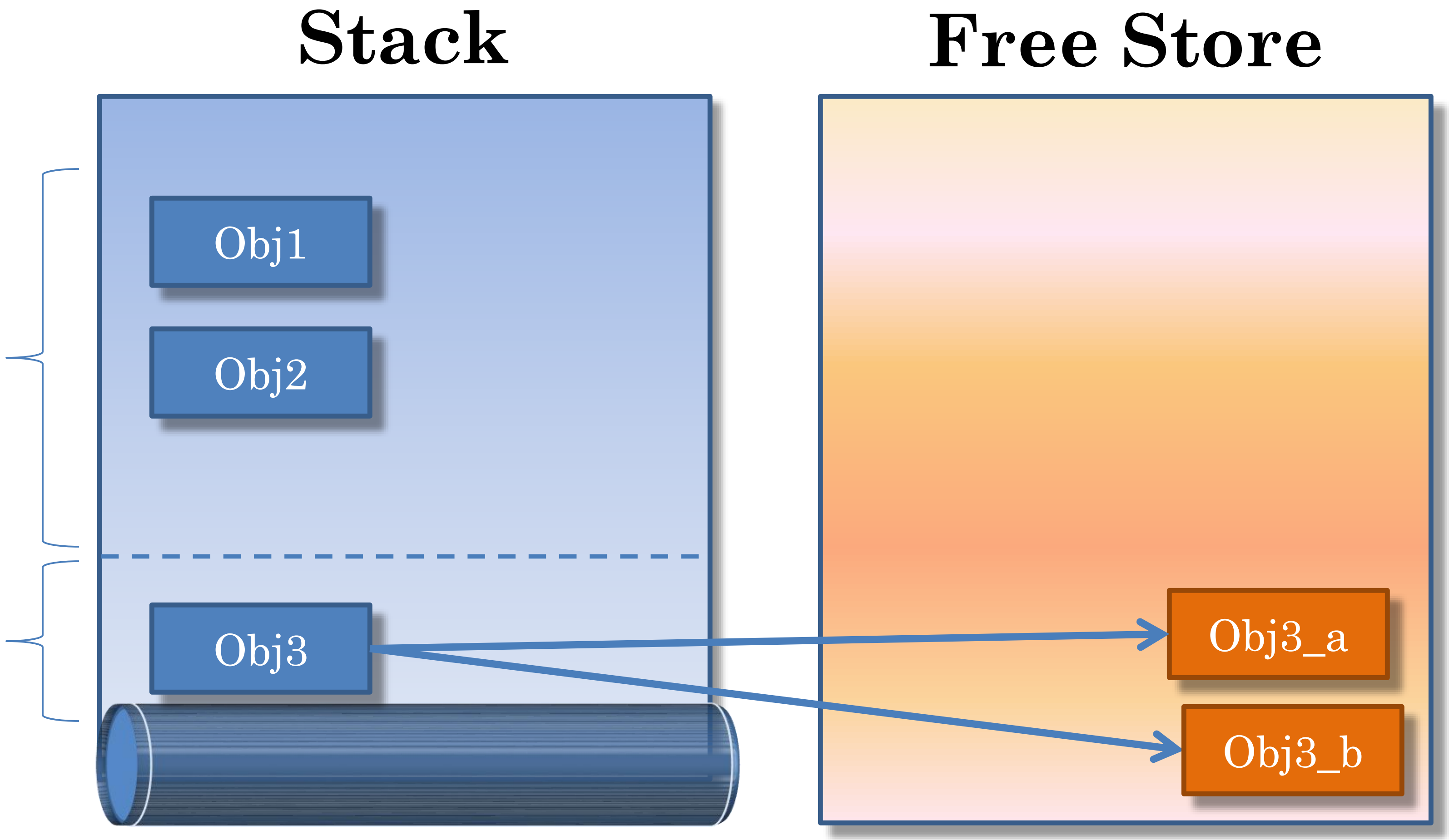
C++11: Move Semantics

- First paper is N1377 Sept 10, 2002
  - Howard Hinnant
  - Peter Dimov
  - Dave Abrahams
- Follow-on papers: N1610, N1690, N1770, N1771, N1856-N1562, N1952, N2027, N2118, N2812, N2831, N2844, N2855, N3010, N3030

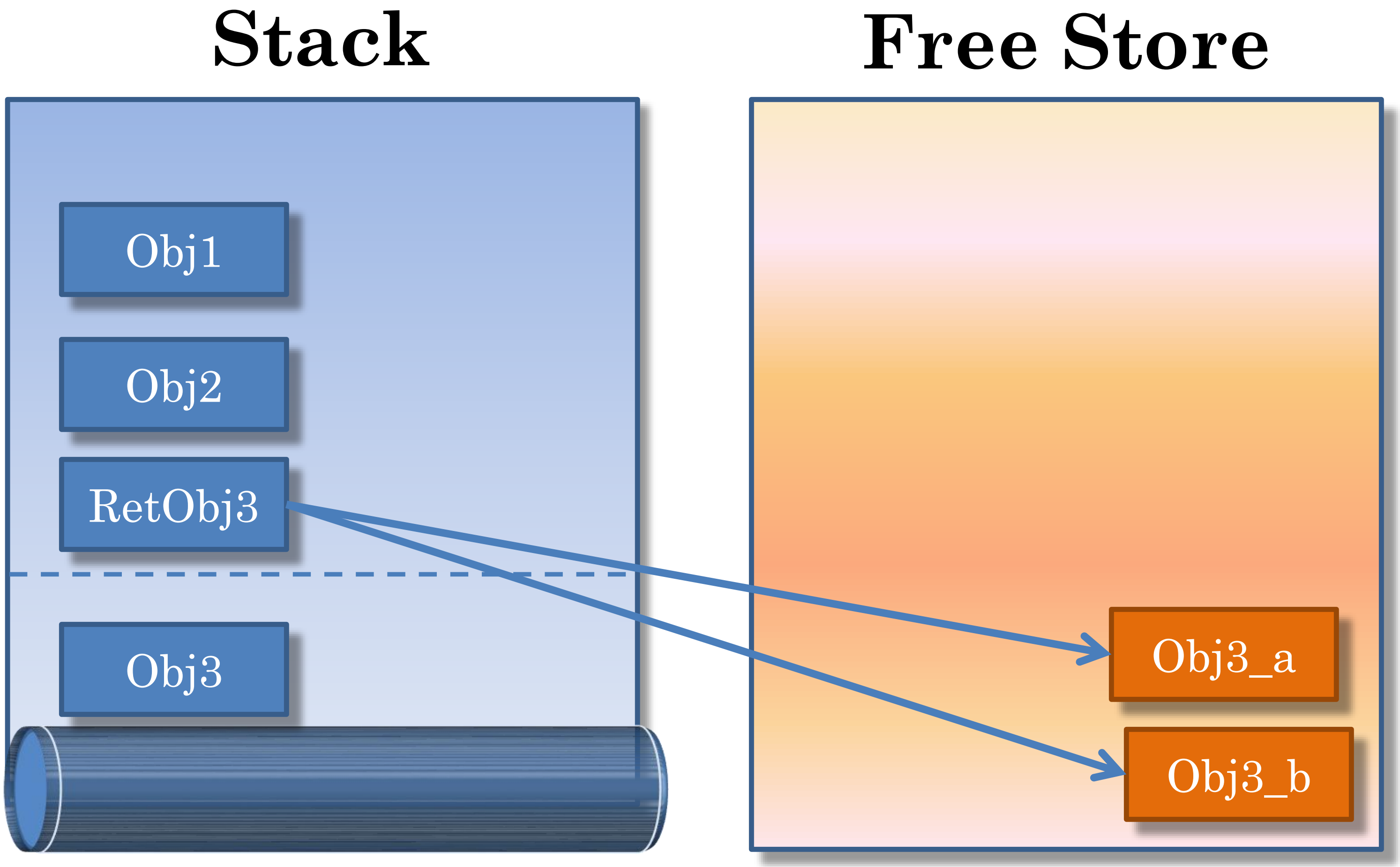
N1377

- Proposed language support to identify temporaries
- Proposed means to explicitly treat non-temporary as temporary
- Introduced terminology:
  - *rvalue reference*
  - *move*

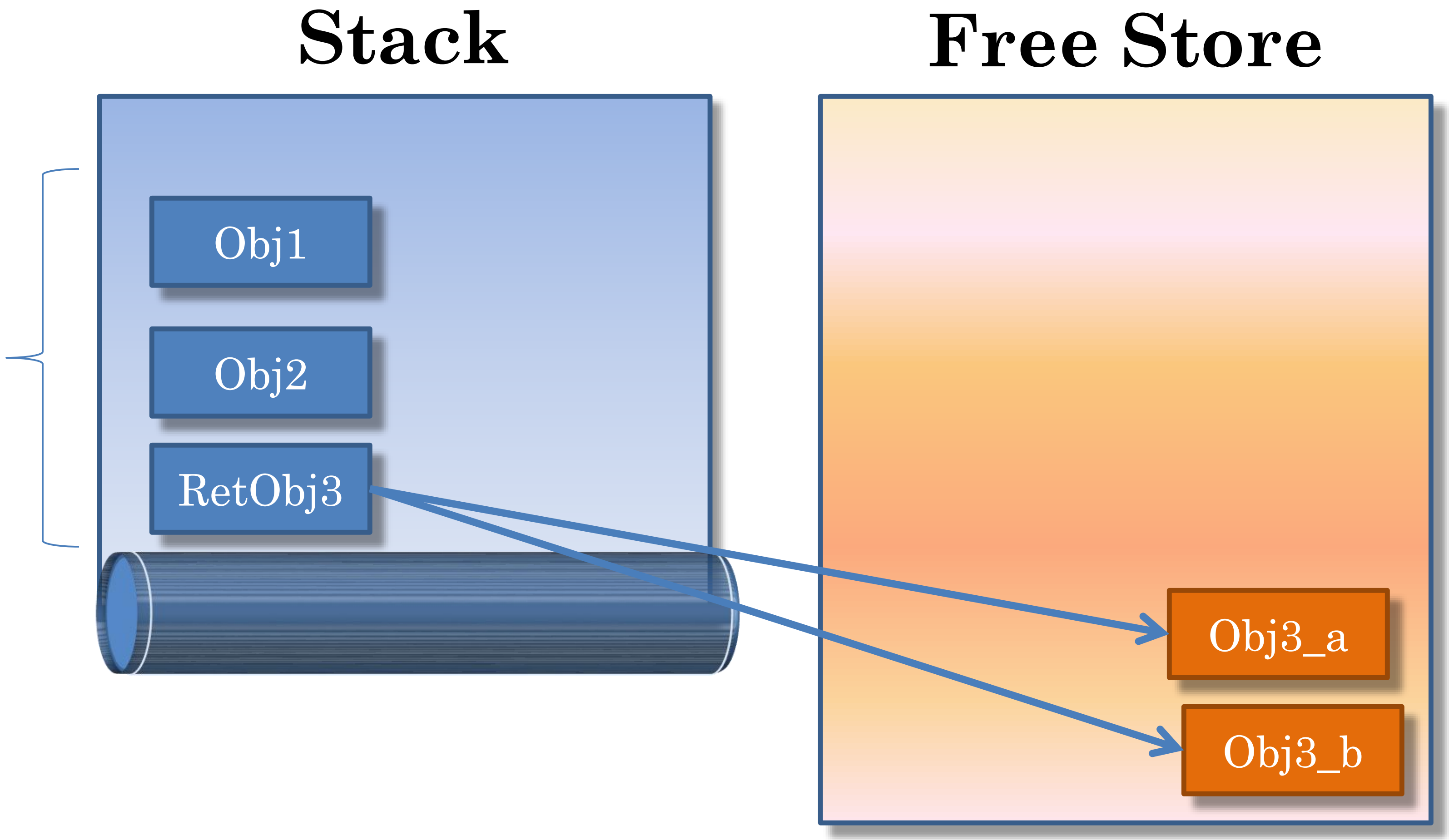
Moving Return pt 1



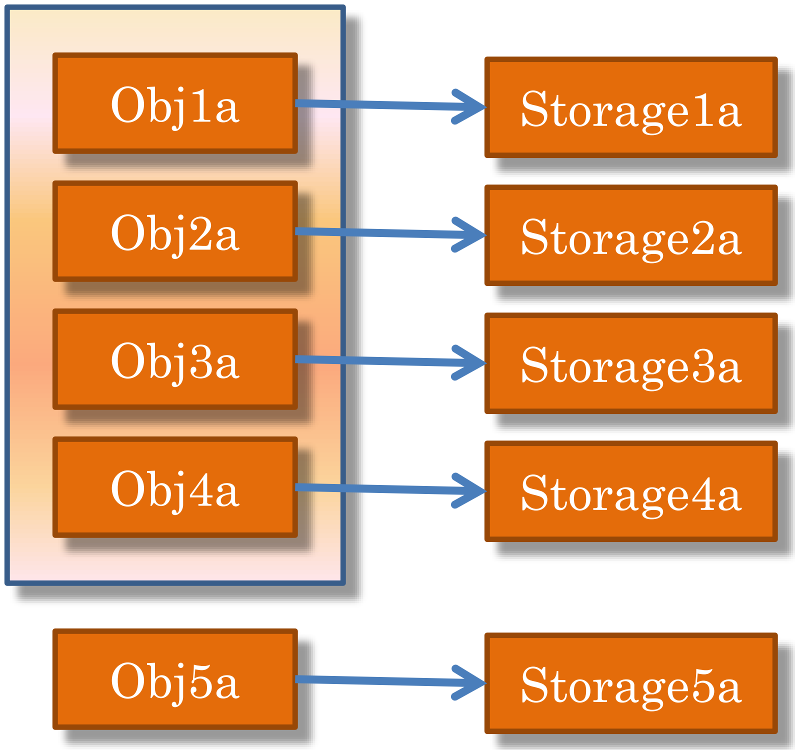
Moving Return pt 2



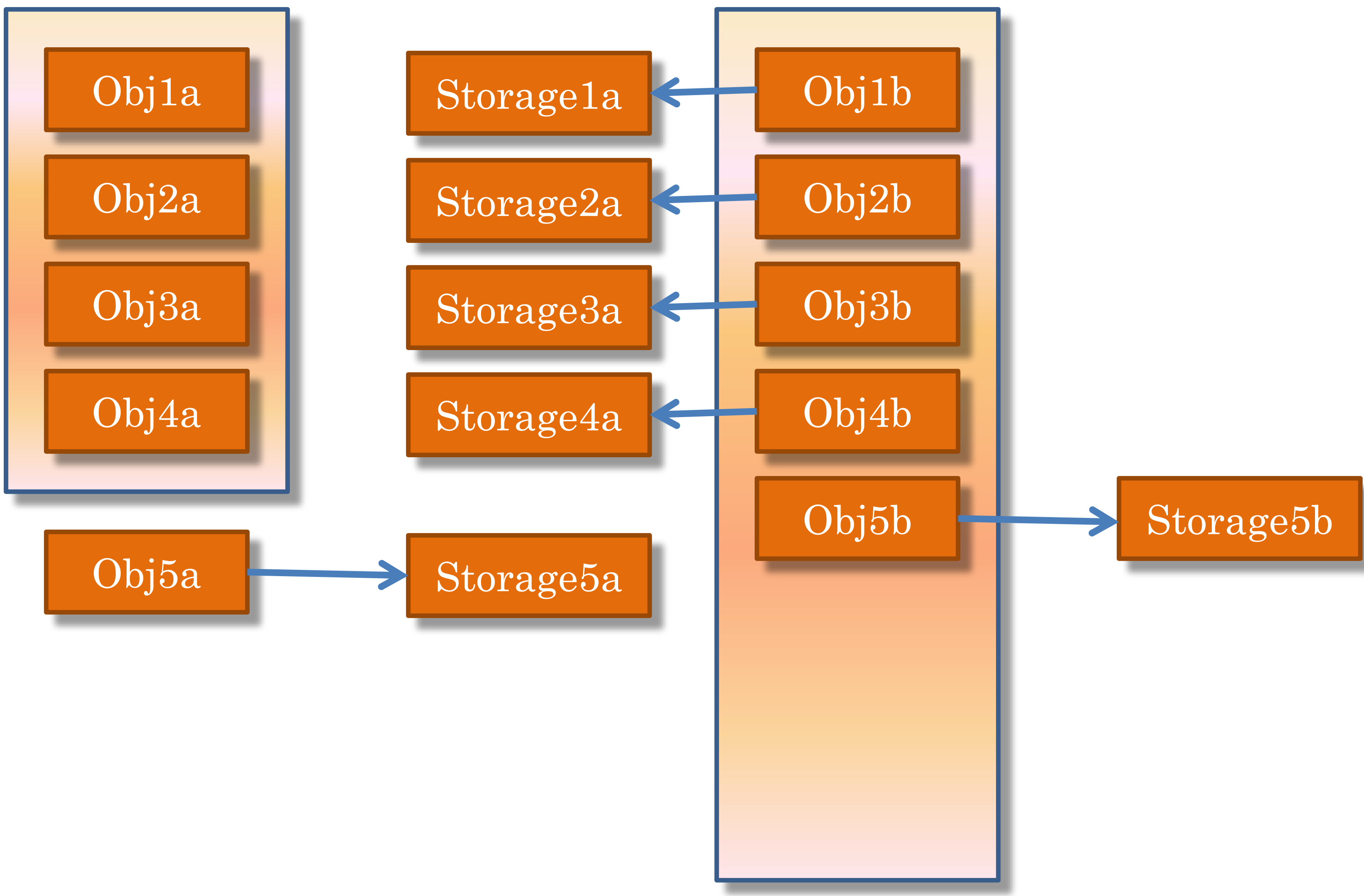
Moving Return pt 3



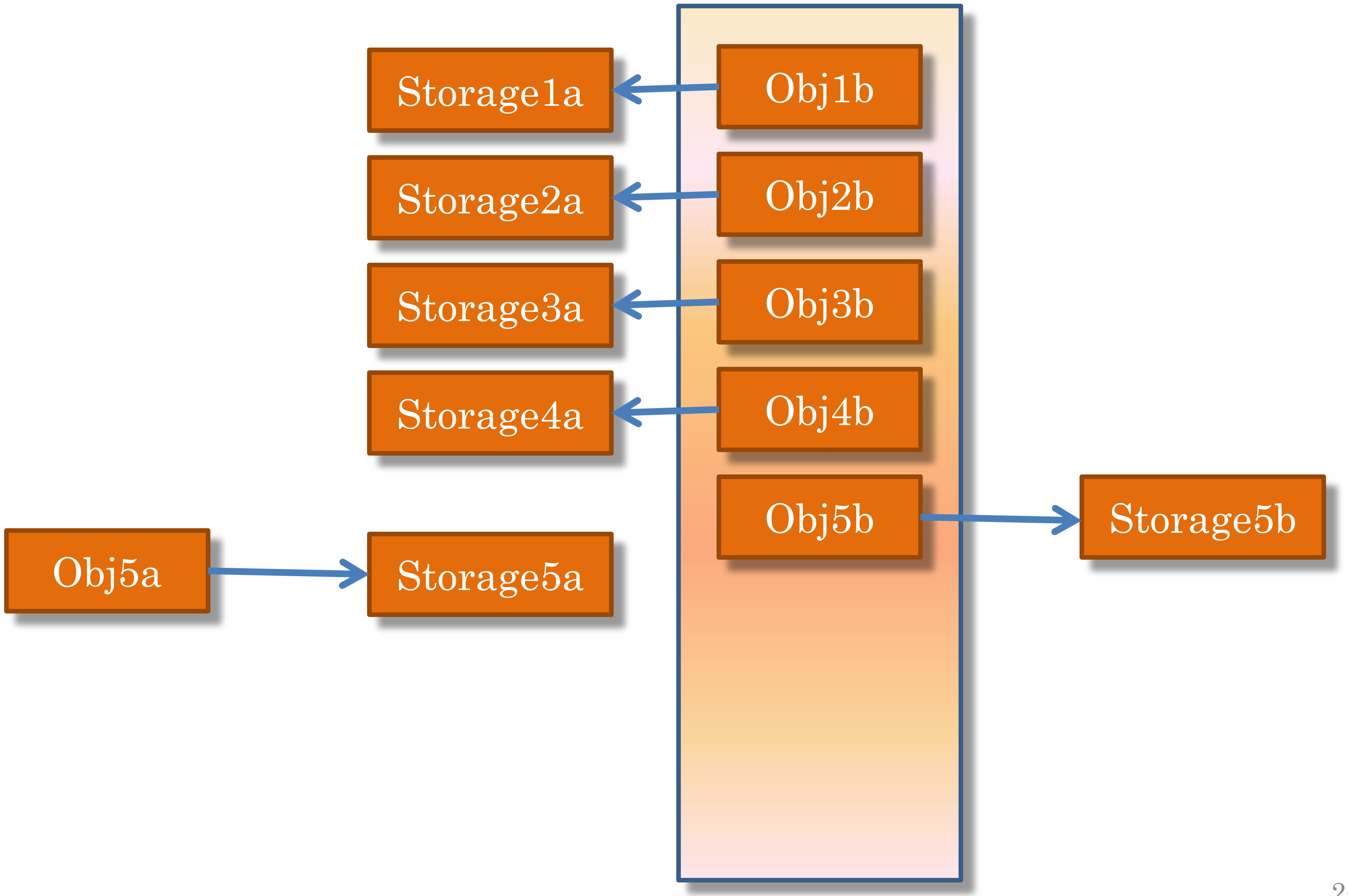
Moving push_back pt 1



Moving push_back pt 2



Moving push_back pt 3



Types That Can Move

- Must have state stored outside the object
  - Free store, referenced by pointer
  - OS object, referenced by handle
- Must have a valid moved-from state
  - Destructor runs on moved-from object

Move-Enable Your Classes?

- When you care about performance

- When your types benefit from move
  - Aggregate movable types
  - Have external (movable) state

Some `std::` Types That Move

- `pair`
- `tuple`
- `basic_string`
- `vector`
- `deque`
- `list`
- `set`
- `map`
- `unordered_set`
- `unordered_map`
- `array`
- `basic_regex`
- `istream`
- `ostream`
- `fstream`
- `basic_stringstream`
- `unique_ptr`
- `future`
- `thread`
- `promise`

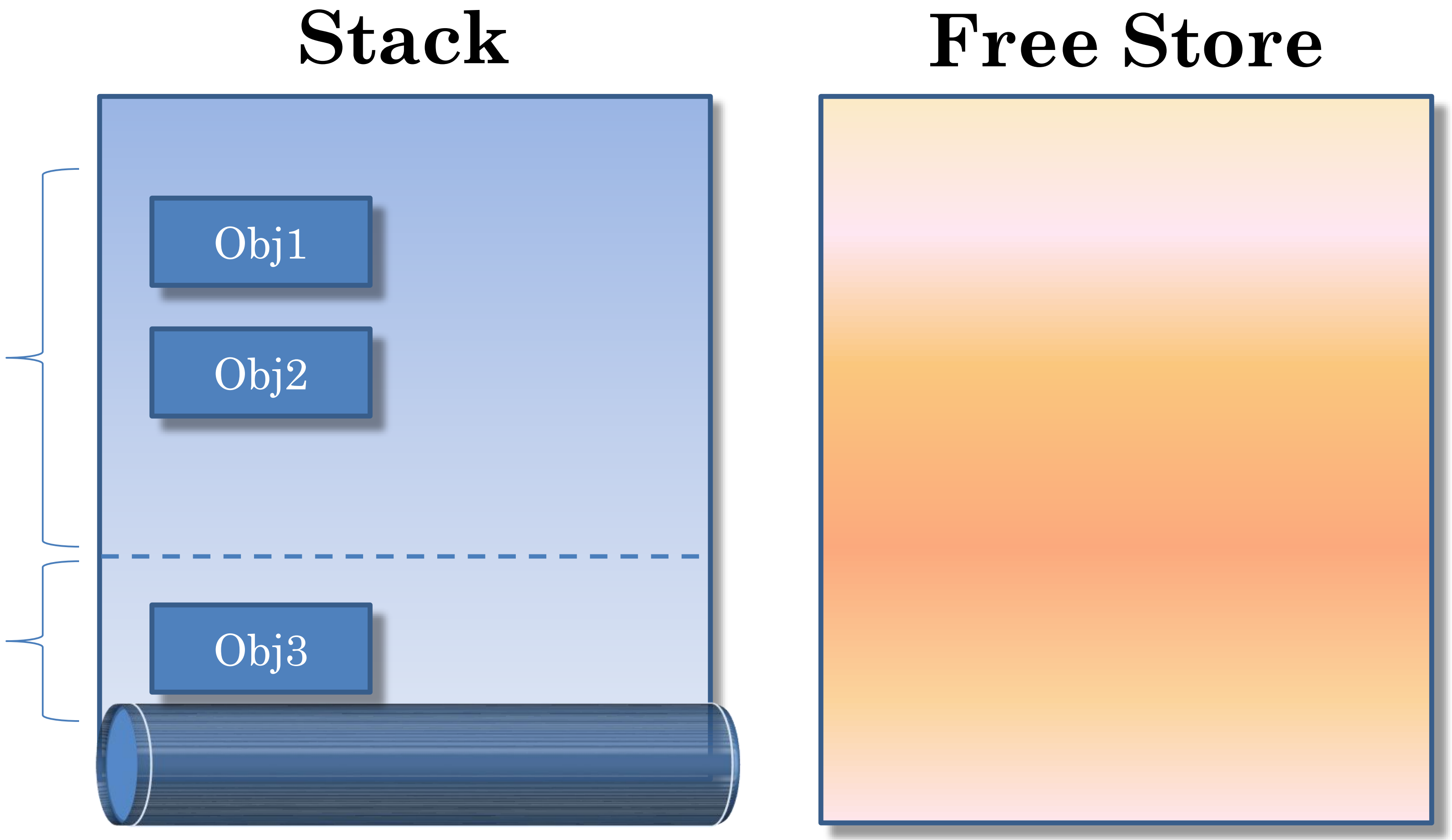
std:: Move-Only Types

- pair
- tuple
- basic_string
- vector
- deque
- list
- set
- map
- unordered_set
- unordered_map
- array
- basic_regex
- istream
- ostream
- fstream
- basic_stringstream
- unique_ptr
- future
- thread
- promise

Moving Is A Euphemism

- We’re **stealing** one or more parts of an object
- The parts we steal **do not move**.
  - The *address* of free store allocations is copied
  - The *FILE** or *handle* of a file is copied
  - The *handle* of a thread is copied

Not All Objects Can Move



What Can We Move From?

- Goals when adding move semantics
  - Improve speed
  - Don't break old programs
- What can be moved from safely?
  - Stealing from most objects will break the program
  - What category of objects can we steal from?
- **Temporaries**

What is a Temporary?

- Anything that isn't an *lvalue*
- "*An* lvalue *is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator.*" – Thomas Becker, C++ Rvalue References Explained
- An *rvalue*

Rvalue Examples

```
std::string a {"Hi Mom!"};   // a is an lvalue
std::string b {"Hi Dad!"};   // b is an lvalue

// Temporaries in complex expressions are rvalues
auto abc = a + b + "c";

// Constructed-in-place parameters are rvalues
void takesAString(const std::string& arg);
takesAString("d");       // "d" as a string is an rvalue

// An object returned by a function is an rvalue
std::vector<int> returnsIntVector(); // returns rvalue
auto v = returnsIntVector();         // v is an lvalue
```

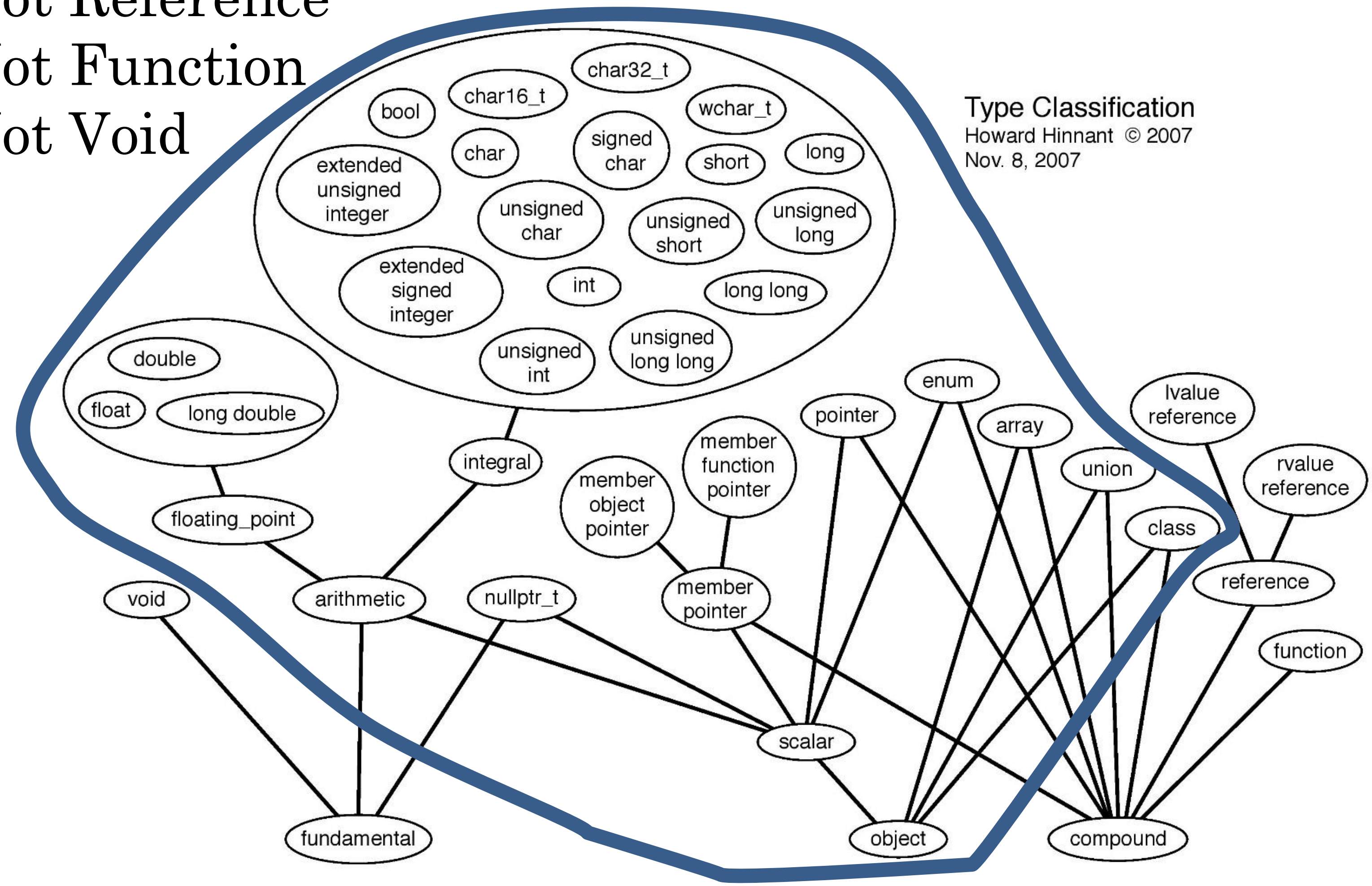
Returning Rvalues

- An *object type* returned by a function is an rvalue

```
int  getInt()    // Returns an rvalue
{
    return 2;
}
int& getIntRef() // Returns an lvalue
{
    static int i = 2;
    return i;
}
```

Object Types

- Not Reference
- Not Function
- Not Void



Used with Permission

Identifying Rvalues pt 1

- New C++11 type: rvalue reference (&&)
- C++98 reference (&) now lvalue reference
- Used for overloading

```
struct Movable1
{
    Movable1(const Movable1& rhs);                  //Copy
    Movable1(Movable1&& rhs) noexcept;              //Move
    Movable1& operator=(const Movable1& rhs);    //Copy
    Movable1& operator=(Movable1&& rhs) noexcept;//Move
};
```

Overload Resolution

- Rvalues…
  - *Prefer* binding to `&&`
  - But also bind to `const &`
- Lvalues…
  - *Never* bind to `&&`
  - But bind to `&`
- If only `const &` is provided everything still works… at C++98 speeds

Always Make && Non-Const!

- Why?

- Because we're going to modify the referred-to object – **steal** from it

Caution!

- && is used primarily in signatures (and occasionally casts, auto and decltype)
- Any other use is suspect

```
std::string&& rRStr = rRefStr();  // Suspect
```

- That compiles and links
- There's probably a problem or misunderstanding

Foreshadowing

- Beware: `&&` may be more than an rvalue reference

- More later

Moving from Lvalues

- Lvalues can be moved from too!
- Appropriate when lvalue not used again before destruction
- As usual, for improved efficiency
- Common in containers
- Common in mutating algorithms

How to Move from an Lvalue

- Remove any current references
- `static_cast` to an rvalue reference

```
static_cast<typename std::remove_reference<T>::type &&>(arg)
```

- Leave this to your standard library!
- Call `std::move()`

`std::move()` Moves Nothing

- `std::move()` is just a cast to an rvalue reference.

```
template<typename _Tp>
  constexpr typename std::remove_reference<_Tp>::type&&
  move(_Tp&& __t) noexcept
  { return static_cast<typename
      std::remove_reference<_Tp>::type&&>(__t); }
```

- Other code must do the actual moving

`noexcept` Move Is Best

Move construction and move assignment are best if they are `noexcept`

- `swap()` is `noexcept` if the move ctor and move assignment are `noexcept`
- If the move ctor for a type in a `vector` is **not** `noexcept` then `vector` uses the copy ctor rather than the move ctor

Don't Return `const` Objects

```
const std::string cantBeMovedFrom() {
    return "Can't move from const";
}
```

- This was fine in C++98
- Less efficient in C++11
- Can't move from const returned objects
- Returning `const` lvalue references is still useful
- But you can't move from `const` refs

Never Return Refs to Locals

```
std::string& bad1() {
    std::string local{ "Local" };
    return local;               // Don't return ref to local
}

std::string&& justAsBad1() {
    std::string local { "Local" };
    return std::move(local); // Don't return ref to local
}
```

- The stack giveth and…
- The stack taketh away

Don't `move()` Return Values

```
std::string lessEfficient() {
    std::string local { "Local" };
    return std::move(local); // Less efficient!
}
```

- Prevents Return Value Optimization
- RVO is more efficient than moving

Topics

- Move Motivation and Background
- **Implementing Move**
- Universal References and Perfect Forwarding
- Overloading With Universal References
- Summary

Implementing Mooove

Ways To Implement Move

- **With compiler-provided move constructors and move assignment**
- Using movable types
- Hand coding

C++11 Special Member Functions

The compiler may silently provide:

- Default constructor
- Copy constructor
- Copy assignment
- Destructor
- Move constructor
- Move assignment

New with C++11

Implicit Move

```
class Mov1 {   // Has an implicit move constructor!
public:
    Mov1()
    : s_ { "Avoid the small string optimization" }
    { }
    bool empty() const { return s_.empty(); }
private:
    std::string s_;
};
STATIC_ASSERT(
std::is_nothrow_move_constructible<Mov1>::value == true);
STATIC_ASSERT(
std::is_nothrow_move_assignable<Mov1>::value == true);
```

Compiler-Provided Move and `noexcept`

If all non-static data members have `noexcept` move the compiler generates `noexcept` move

- Move constructor
- Move assignment operator

Test Implicit Move Ctor

```
void mov1_test() {
    {
        Mov1 mA;                    // Default construct mA
        auto mB(mA);                // Copy construct mB
        assert(mA.empty() == false);
    }
    {
        Mov1 mC;                    // Default construct mC
        auto mD(std::move(mC));     // Move construct mD
        assert(mC.empty() == true);
        // mC is empty() 'cuz it's moved from
    }
}
```

Moving from `string`

- The standard does not guarantee a moved-from `std::string` is `empty`

- When move constructing, a long enough moved-from `string` is *typically* `empty`

- The moved-from `string` is in a valid state with an unspecified value

Losing An Implicit Move Ctor pt 1

No implicit move constructor if you declare any of…

- Copy constructor
- Copy assignment operator
- Move assignment operator
- Destructor

Losing An Implicit Move Ctor pt 2

No implicit move constructor if class:

- Has non-static data members
  - With no move constructor and
  - Non-trivially copyable
- Has a base class
  - With no move constructor and
  - Non-trivially copyable
- Has a base class with a deleted or inaccessible destructor
- Is a union with a variant member with a non-trivial copy constructor

Caution!

If you rely on the presence of an implicit move constructor or move assignment operator…

- **Guarantee** it with a static_assert
- Or explicitly request the default implementation

Implicit Move-Only

```
class MovOnly1 {   // Implicitly move-only
public:
    MovOnly1() : up_ { new std::string {"Mooo"} } { }
    bool empty() const { return !up_; }
private:
    std::unique_ptr<std::string> up_;
};
STATIC_ASSERT(
std::is_copy_constructible<MovOnly1>::value == false);
STATIC_ASSERT(
std::is_nothrow_move_constructible<MovOnly1>::value == true);
STATIC_ASSERT(
std::is_copy_assignable<MovOnly1>::value == false);
STATIC_ASSERT(
std::is_nothrow_move_assignable<MovOnly1>::value == true);
```

Implicit Move-Only Test

```
void movOnly1_test() {
    {
        MovOnly1 mA;          // Default construct mA
//      auto mB(mA);          // Doesn't compile.  No copy.
    }
    {
        MovOnly1 mC;                   // Default construct mC
        assert(mC.empty() == false);
        auto mD(std::move(mC));  // Move construct mD
        assert(mC.empty() == true);
    }
}
```

Explicit Default Move Ctor

```
class Mov2a {   // Explicit default move constructor
public:
    Mov2a()
    : s_{ "Avoid the small string optimization" } { }
    Mov2a(Mov2a&& rhs) noexcept = default;
    bool empty() const { return s_.empty(); }
private:
    std::string s_;
};
STATIC_ASSERT(
std::is_nothrow_move_constructible<Mov2a>::value == true);
```

Test Explicit Default Move

```
void mov2a_test() {
    Mov2a mA;                    // Default construct mA
    auto mB(mA);                 // Can’t copy construct mB
    assert(mA.empty() == false);

    Mov2a mC;                    // Default construct mC
    auto mD(std::move(mC));      // Move construct mC
    assert(mC.empty() == true);
}
```

Compile Error!

```
error: use of deleted function 'Mov2a::Mov2a(const
Mov2a&)'
     auto mB(mA);
```

Implicit Functions Bite Again

Explicitly declaring the move constructor…

Deleted the implicit copy constructor

Suggestion:

- If you `default` or `delete` any special member functions…
- Explicitly `default` or `delete` any others you don't implement.

Explicit Defaults

```
class Mov2 {   // Explicit default move constructor
public:
    Mov2()
    : s_{ "Avoid the small string optimization" } { }
    Mov2(const Mov2& rhs) = default;
    Mov2(Mov2&& rhs) noexcept = default;
    Mov2& operator=(const Mov2& rhs) = default;
    Mov2& operator=(Mov2&& rhs) noexcept = default;
    ~Mov2() noexcept = default;
    bool empty() const { return s_.empty(); }
private:
    std::string s_;
};
```

Check Your Work

Write six `static_asserts` to verify that the special member functions are exactly what you expect

No run-time cost!

Traits for `static_assert`

```
STATIC_ASSERT(
std::is_default_constructible<Mov2>::value == true);
STATIC_ASSERT(
std::is_copy_constructible<Mov2>::value == true);
STATIC_ASSERT(
std::is_nothrow_move_constructible<Mov2>::value == true);
STATIC_ASSERT(
std::is_copy_assignable<Mov2>::value == true);
STATIC_ASSERT(
std::is_nothrow_move_assignable<Mov2>::value == true);
STATIC_ASSERT(
std::is_nothrow_destructible<Mov2>::value == true);
```

Test Explicit Defaults

```
void mov2_test() {
    Mov2 mA;                         // Default construct mA
    auto mB(mA);                     // Copy construct mB
    assert(mA.empty() == false);

    Mov2 mC;                         // Default construct mC
    auto mD(std::move(mC));  // Move construct mC
    assert(mC.empty() == true);
}
```

Ta dah!

Ways To Implement Move

- With compiler-provided move constructors and move assignment

- **Using movable types**

- Hand coding

Moving Movable Types Bug

```
class Mov3a {   // Explicit move constructor
public:
    Mov3a()
    : s_ { "Avoid the small string optimization" } { }
    Mov3a(const Mov3a& rhs) = default;

    Mov3a(Mov3a&& rhs) noexcept : s_(rhs.s_) { }  // Bug!

    Mov3a& operator=(const Mov3a& rhs) = default;
    Mov3a& operator=(Mov3a&& rhs) noexcept = default;
    ~Mov3a() noexcept = default;
    ...
```

The Move Did Not Move

```
void mov3a_copy_test() {
    Mov3a mA;                    // Default construct mA
    auto mB(mA);                 // Copy construct mB
    assert(mA.empty() == false);

    Mov3a mC;                    // Default construct mC
    auto mD(std::move(mC));      // Move construct mC
    assert(mC.empty() == true);  // Fails!
}
```

Runtime Error!

```
Assertion failed!
Expression: mC.empty() == true
```

Moving Lvalues

```
...
Mov3a(Mov3a&& rhs) noexcept : s_(rhs.s_) { }  // Bug!
...
```

- Argument `rhs` has a name
- **A named rvalue ref is an lvalue**!
- We have implemented **copy**, not move

The Fix

```
...
Mov3b(Mov3b&& rhs) noexcept : s_(std::move(rhs.s_)) { }
...
```

- Always call `std::move()` when moving
- It's occasionally not needed but it's never wrong and there's no run-time overhead

Moving Movable Types

```
class Mov3 {   // Explicit moves
public:
    Mov3() :s_{"Avoid the small string optimization"} { }
    Mov3(const std::string& rhs) : s_ { rhs } { }
    Mov3(const Mov3& rhs) : s_{rhs.s_} { }
    Mov3(Mov3&& rhs) noexcept : s_{std::move(rhs.s_)} { }
    Mov3& operator=(const Mov3& rhs)
        { s_ = rhs.s_; return *this; }
    Mov3& operator=(Mov3&& rhs) noexcept
        { s_ = std::move(rhs.s_); return *this; }
    ~Mov3() noexcept { }
    ...
```

Now The Move Works

```
void mov3a_test() {
    Mov3a mA;                    // Default construct mA
    auto mB(mA);                 // Copy construct mB
    assert(mA.empty() == false);

    Mov3a mC;                    // Default construct mC
    auto mD(std::move(mC));      // Move construct mC
    assert(mC.empty() == true);
}
```

Move Assignment?

```
void mov3_assign_bad_test() {
    Mov3 mA, mB;                // Default construct mA
    mB = std::move(mA);         // Assignment
    assert(mA.empty() == true);  // May fail
}
```

Runtime Error!

```
Assertion failed!
Expression: mA.empty() == true
```

What Went Wrong?

- `std::string` move assignment may be implemented with `swap()`

- We assume the moved-from object will be destroyed soon
- Not only may we steal from it…
- We may load it up with our garbage

Testing Moved-From Objects

- Tests of moved-from objects are dubious
- The destructor is guaranteed to work
- Classes that support assignment are assignable after being moved from
- Everything else is implementation (or standard) dependent
- But it may be the only way to know your moves are working

Ways To Implement Move

- With compiler-provided move constructors and move assignment

- Using movable types

- **Hand coding**

Minimize Hand Coding

- Put free store objects in `unique_ptr<T>`
- Put free store arrays in `unique_ptr<T[]>`
- Let `unique_ptr` do the moving

unique_ptr Example

```
class MovHeap {
    static std::string* dup(const MovHeap& m)
    {return m.up_ ? new std::string {*m.up_} : nullptr;}
public:
    MovHeap() : up_ { new std::string {"Moooooo"} } { }
    MovHeap(const MovHeap& rhs) : up_ { dup(rhs) } { }
    MovHeap(MovHeap&& rhs) noexcept = default;
    MovHeap& operator=(const MovHeap& rhs)
        { up_.reset(dup(rhs)); return *this; }
    MovHeap& operator=(MovHeap&& rhs) noexcept = default;
    bool empty() const { return !up_; }
private:
    std::unique_ptr<std::string> up_;
};
```

Let **ME** Do It!

```
class MovByHand {
    std::string* p_;
public:
    // ...
    MovByHand(MovByHand&& rhs) noexcept
      : p_ { rhs.p_ } { rhs.p_ = nullptr; }
    MovByHand& operator=(MovByHand&& rhs) noexcept {
            if (this != &rhs) {
                delete(p_);
                p_ = rhs.p_;
                rhs.p_ = nullptr;
            }
            return *this;
        }
    // ...
};
```

Move Assignment To Self

- `std::string` is guaranteed no-op
- `std::unique_ptr` in GCC 4.8.2 is no-op
- `std::unique_ptr` is **not** guaranteed to be no-op by the standard

- In general move assignment to self may leave an empty object

Non-Free Store Moves

For non-memory moves…

- Prefer immediate delete…
  - Like `std::unique_ptr`
- Over `swap()` like behavior
  - Like some `std::string` implementations

- `swap()` like behavior postpones releasing the resource out of the immediate scope

Mooving Setter

Moving Setter / Non-Copy Ctor

```
class MyFile {
public:
    MyFile() noexcept { }
    MyFile(const std::string& name) :
        name_ { name } { }
    MyFile(std::string&& name) noexcept :
        name_ { std::move(name) } { }
    void SetName(const std::string& name)
        { name_ = name; }
    void SetName(std::string&& name)
        { name_ = std::move(name); }
    bool empty() const { return name_.empty(); }
private:
    std::string name_;
};
```

Multi-Argument Ctor

```
struct PathAndFile1 {
    using string = std::string;
    PathAndFile1(const string&  p, const string&  f)
        : p_ { p },            f_ { f }            { }

    PathAndFile1(      string&& p, const string&  f)
        : p_ { std::move(p) }, f_ { f }            { }

    PathAndFile1(const string&  p,       string&& f)
        : p_ { p },            f_ { std::move(f) } { }

    PathAndFile1(      string&& p,       string&& f)
        : p_ { std::move(p) }, f_ { std::move(f) } { }
private:
    string p_, f_;
};
```

Topics

- Move Motivation and Background
- Implementing Move
- **Universal References and Perfect Forwarding**
- Overloading With Universal References
- Summary

Universal References and Perfect Forwarding

The Trouble With Tuples

- C++11 std::tuple has a ctor that takes N arguments.
- Any of those arguments may be rvalues.
- What to do?

- Universal references
- Perfect forwarding

Universal References

```
template <class T> void f(T&& v);
```

- Scott Meyers calls this use of `&&` a *universal reference*
- `T&&` binds to almost anything
  - lvalues
  - rvalues
- No const or volatile in `T&&`
- `T` must be locally deduced

Universal Reference Rules

Universal reference type decode:

- `T&  -> T&`
- `T&& -> T`

- `const / volatile` preserved

Universal Reference Test

```
template<typename EXPECT, typename T>
void urefDecode(T&& t)
{ STATIC_ASSERT(std::is_same< EXPECT, T >::value); }

void urefDecode_test()
{
    using string = std::string;
    string s { "str" };
    const volatile string cv_s { "cv_str" };
    urefDecode<string&>(s);
    urefDecode<const volatile string&>(cv_s);
    urefDecode<string>(std::move(s));
    urefDecode<const volatile string>(std::move(cv_s));
}
```

C++11 Reference Collapsing

- Behavior taking a ref of ref?

- `T&   &   ->  T&`
- `T&&  &   ->  T&`
- `T&   &&  ->  T&`
- `T&&  &&  ->  T&&`

Perfect Forwarding Test

```
template<typename EXPECT, typename T>
void fwdDecode(T&& t)
{ STATIC_ASSERT(std::is_same< EXPECT, T&& >::value); }

void fwdDecode_test()
{
    using string = std::string;
    string s { "str" };
    const volatile string cv_s { "cv_str" };
    fwdDecode<string&>(s);
    fwdDecode<const volatile string&>(cv_s);
    fwdDecode<string&&>(std::move(s));
    fwdDecode<const volatile string&&>(std::move(cv_s));
}
```

Use `std::forward<>()`

Don't do the casting yourself

- `std::forward` covers more cases
- `std::forward` is easier to read
- `std::forward` shows intent

Locally Deduced Type

```
template<typename T>
struct tClass
{
    // Not a universal ref.  'T' comes from class.
    template<typename EXPECT>
    static void notURef(T&& t)
    { STATIC_ASSERT(std::is_same<EXPECT, T&&>::value); }

    // Is a universal ref.  'U' is local to function.
    template<typename EXPECT, typename U>
    static void yesURef(U&& u)
    { STATIC_ASSERT(std::is_same<EXPECT, U&&>::value); }
};
```

Locally Deduced Test

```
void deduce_test()
{
    using string = std::string;
    string s { "str" };

//  tClass<string>::notURef<string&>(s); // Compile error
    tClass<string>::notURef<string&&>(std::move(s));

    tClass<string>::yesURef<string&>(s);
    tClass<string>::yesURef<string&&>(std::move(s));
}
```

Multi-Argument Ctor Revisited

```
struct PathAndFile {

    // Four ctors become one templated ctor
    template<typename P, typename F>
    PathAndFile(P&& p, F&& f)
        : p_ { std::forward<P>(p) }
        , f_ { std::forward<F>(f) } { }
private:
    std::string p_, f_;
};
```

Multi-Argument Ctor Test

```
void pathAndFile_test()
{
    using string = std::string;
    string pA { "pathA" };
    string fA { "fileA" };
    PathAndFile(pA, fA);
    assert((pA.empty() == false) && (fA.empty() == false));

    PathAndFile(std::move(pA), std::move(fA));
    assert((pA.empty() == true ) && (fA.empty() == true ));
}
```

- One multi-argument ctor copies or moves

A Better Multi-Argument Ctor

```
struct PathAndFile {
    // Four ctors become one templated ctor
    template<typename P, typename F,
        typename = typename std::enable_if<
            std::is_constructible<std::string, P>::value
         && std::is_constructible<std::string, F>::value,
            void>::type
    >
    PathAndFile2(P&&  p, F&& f)
        : p_ { std::forward<P>(p) }
        , f_ { std::forward<F>(f) } { }
private:
    std::string p_, f_;
}
```

Nearly Perfect Forwarding

We can't perfectly forward…

- 0 as null pointer constant (use `nullptr`),
- Braced initializer lists,
- Template names (e.g., `std::endl`),
- Non-const lvalue bitfields,
- Other odd cases.

Nearly perfect forwarding is still pretty good

Don't `move()` a Universal Ref

```
struct BadPathAndFile {

    // Four ctors become one templated ctor
    template<typename P, typename F>
    BadPathAndFile(P&&  p, F&& f)
        : p_ { std::move(p) }      // bug
        , f_ { std::move(f) } { }  // bug
private:
    string p_, f_;
};
```

Moving a Universal Ref

```
void badPathAndFile_test()
{
    using string = std::string;
    string pA { "pathA" };
    string fA { "fileA" };
    BadPathAndFile(pA, fA);
    assert((pA.empty() == false) && (fA.empty() == false));
}
```

```
Assertion failed!
Expression: (pA.empty() == false) && (fA.empty() ==
false)
```

`std::move` moves both rvalues and lvalues

Know the Difference!

- Universal Reference:

```
template <typename T>                  // T locally type deduced
void f(T&& arg);                       // No qualifiers
```

- Any other use of unary && is an rvalue reference.

Important!

- Always use `std::forward<>()` with universal references.

- Always use `std::move()` with rvalue references.

Topics

- Move Motivation and Background
- Implementing Move
- Universal References and Perfect Forwarding
- **Overloading With Universal References**
- Summary

Overloading With Universal References

An Easy Mistaik

```
class BadMov1 {
public:
    BadMov1() : s_ { "Moooove over..." } { }
    BadMov1(const BadMov1& s) = default;
    BadMov1(BadMov1&& s) noexcept = default;
    template<typename S>
    BadMov1(const S& s) : s_ { s } { }
    template<typename S>
    BadMov1(S&& s) : s_ { std::move(s) } { }  // Bug!
    bool empty() const { return s_.empty(); }
private:
    std::string s_;
};
```

Test For Easy Mistaik

```
void badMov1_test()
{
    std::string badMovStr { "bmA" };  // non-const lvalue
    BadMov1 bmA { badMovStr };    // uref moved badMovStr
    assert(badMovStr.empty() == false);
}
```

```
Assertion failed!
Expression: badMovStr.empty() == false
```

The Easy Mistaik?

- Recognize universal references.

- Always use `std::forward` in a universal reference.

The Next Mistaque

```
class BadMov2 {
public:
    BadMov2() : s_ { "Moooove over..." } { }
    BadMov2(const BadMov2& s) = default;
    BadMov2(BadMov2&& s) noexcept = default;
    template<typename S>
    BadMov2(S&& s) : s_ { std::forward(s) } { }  // Bug!
    bool empty() const { return s_.empty(); }
private:
    std::string s_;
};
```

Test For Next Mistaque

```
void badMov2_test()
{
    BadMov2 bmA;            // non-const BadMov2 lvalue
    BadMov2 bmB(bmA);       // binds to universal reference
}
```

```
URef.h: In instantiation of 'BadMov2::BadMov2(S&&)
[with S = BadMov2&]':

error: no matching function for call to
'forward(BadMov2&)‘
BadMov2(S&& s) : s_ { std::forward(s) } { } // Bug!
```

The Next Mistaque?

- Universal references bind to anything that other overloads don't explicitly specify.
- Avoid overloads on universal references.

Another Misteak

```
class BadMov3 {
public:
   BadMov3() : s_ { "Moooove over..." } { }
   BadMov3(const BadMov3& m) = delete;              // Bug!
   BadMov3(BadMov3&& m) noexcept = delete;          // Bug!
   template<typename T>
   BadMov3(T&& t) : BadMov3 { std::forward<T>(t),
       typename std::is_same<BadMov3,
           typename std::remove_cv<
               typename std::remove_reference<T>
                   ::type>::type>::type() } { }
   bool empty() const { return s_.empty(); }
private:
   BadMov3(const BadMov3& m, std::true_type) : s_ { m.s_ } { }
   BadMov3(BadMov3&& m, std::true_type) : s_ { std::move(m.s_) } { }
   template<typename S>
   BadMov3(S&& s, std::false_type) : s_(std::forward<S>(s)) { }
   std::string s_;
};
```

Test For Another Misteak

```
void badMov3_test()
{
    BadMov3 bmA;
    BadMov3 bmB(bmA);
}
```

```
error: use of deleted function
'BadMov3::BadMov3(const BadMov3&)'
     BadMov3 bmB(bmA);
                    ^
URefOverload.h:96:5: error: declared here
     BadMov3(const BadMov3& m) = delete;   // Bug!
```

Another Misteak?

- You may not refer to a deleted function

- N3485 Section 8.4.3 paragraph 2:
  "A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed.“

- We deleted our copy and move ctors

A Way Out

```
class OkayMov {
public:
   OkayMov() : s_ { "Moooove over..." } { }
   OkayMov(const OkayMov& m) : OkayMov { m, std::true_type() } { }
   OkayMov(OkayMov&& m) noexcept
       : OkayMov {std::move(m), std::true_type()} { }
   template<typename T>
   OkayMov(T&& t) : OkayMov { std::forward<T>(t),
       typename std::is_same<OkayMov,
           typename std::remove_cv<
               typename std::remove_reference<T>
                   ::type>::type>::type() } { }
   bool empty() const { return s_.empty(); }
private:
   OkayMov(const OkayMov& m, std::true_type) : s_ { m.s_ } { }
   OkayMov(OkayMov&& m, std::true_type) : s_ { std::move(m.s_) } { }
   template<typename S>
   OkayMov(S&& s, std::false_type) : s_(std::forward<S>(s)) { }
   std::string s_;
};
```

A Shorter Way Out

```
class OkMov {
public:
    OkMov() : s_ { "Moooove over..." } { }
    OkMov(const OkMov& m) = default;
    OkMov(OkMov&& m) noexcept = default;
    template<typename T,
        typename = typename std::enable_if<
        std::is_constructible<std::string, T>::value,
        void>::type
    >
    OkMov(T&& s) : s_{ std::forward<T>(s) } { }
    bool empty() const { return s_.empty(); }
private:
    std::string s_;
};
```

A Way Out?

- `OkayMov` and `OkMov` survived my tests
- How badly do you want that universal reference overload anyway?

Topics

- Move Motivation and Background
- Implementing Move
- Universal References and Perfect Forwarding
- Overloading With Universal References
- **Summary**

The End Of The Trail

Guidance

1. Don't return `const` objects
2. Use `&&` only in signatures (and maybe casts, auto, and decltype)
3. Don't use `const` in `&&` signatures
4. Prefer `noexcept` in move ctors and move assignment
5. Don't `std::move()` return values; it disables the Return Value Optimization

More Guidance

6. Use `static_assert`s to validate special member functions
7. Learn to recognize the difference between universal refs and ordinary rvalue refs
8. Use `std::move()` with rvalue refs
9. Use `std::forward<>()` with universal refs
10. Take care when overloading on universal references
11. **Test your work**

Should I Stop Worrying And Return By Value?

- Free store access hurts efficiency
- Move semantics are not a panacea
- The most efficient designs reuse storage

Consider Eric Niebler's article:

Out Parameters, Move Semantics, and Stateful Algorithms.

http://ericniebler.com/2013/10/13/out-parameters-vs-move-semantics/

Sources pt 1

- Hinnant, Abrahams, and Dimov *N1377: A Proposal to Add Move Semantics Support to the C++ Language* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm#std::move
- Thomas Becker *C++ Rvalue References Explained* http://thbecker.net/articles/rvalue_references/section_01.html
- Scott Meyers *Overview of the New C++ (C++11)*
- Scott Meyers *Adventures In Perfect Forwarding* http://aristeia.com/TalkNotes/Facebook2012_PerfectForwarding.pdf
- Scott Meyers *The Universal Reference/Overloading Collision Conundrum* http://www.youtube.com/watch?v=T5swP3dr190
- Dan Saks *Lvalues and Rvalues* http://www.embedded.com/electronics-blogs/programming-pointers/4023341/Lvalues-and-Rvalues

Sources pt 2

- Howard Hinnant *Moving Forward with C++11* parts 1 and 2 *https://github.com/boostcon/cppnow_presentations_2012*
- *http://en.cppreference.com/w/cpp/language/move_constructor* Rules for implicit copy constructors
- *http://en.cppreference.com/w/cpp/language/move_operator* Rules for implicit move assignment operators.
- John Ahlgren *Limits of Named Return Value Optimization* http://john-ahlgren.blogspot.com/2012/04/limits-of-named-return-value.html

Huge thanks to my reviewers, Rob Stewart and Howard Hinnant.

Thanks to Stephan T. Lavavej for the MinGW g++ distro.

All mistakes, errors, and flubs belong to Scott Schurr exclusively.

Questions?