Moving Forward to C++11

Howard Hinnant
May 15, 2012

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Outline

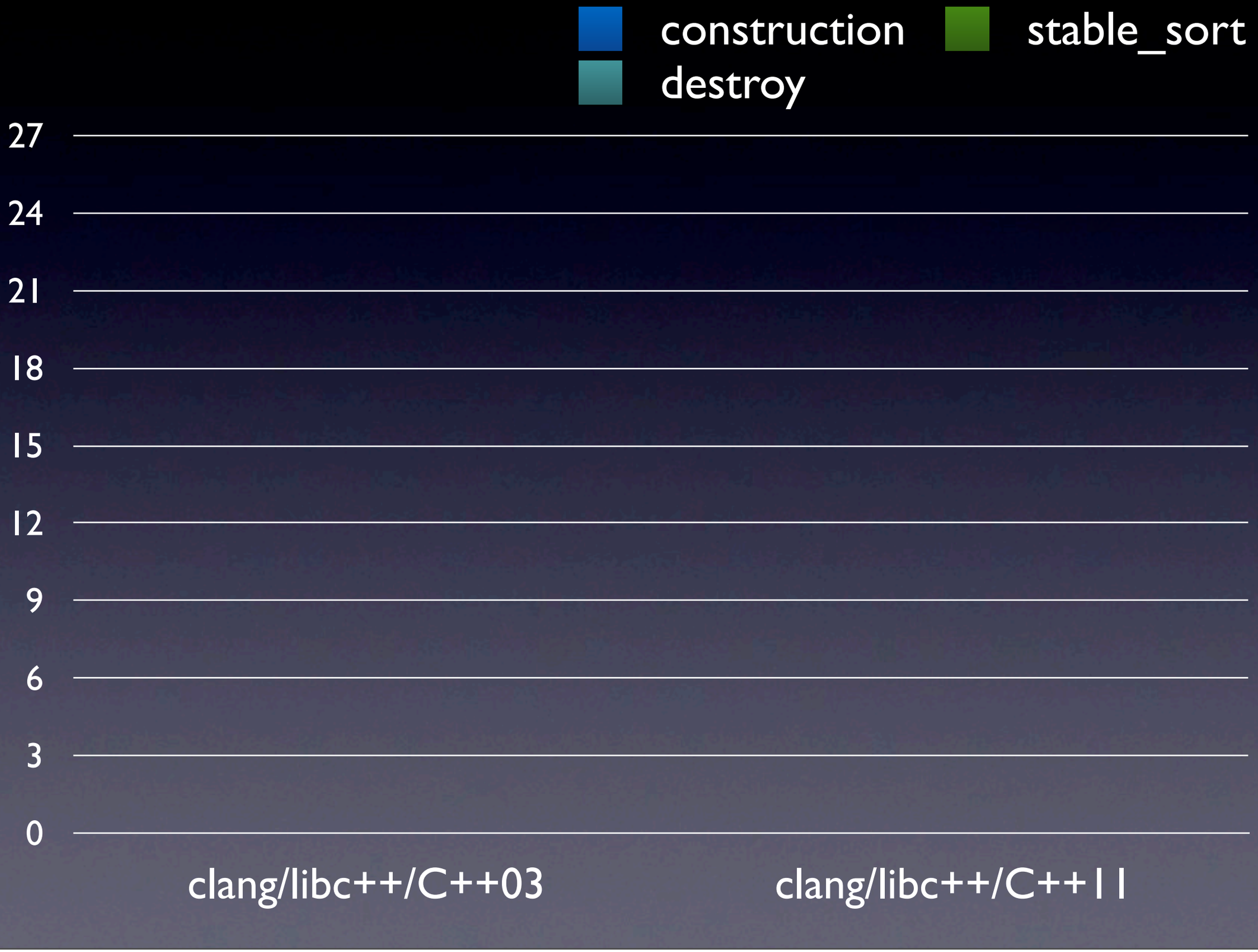
- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Motivation for Move

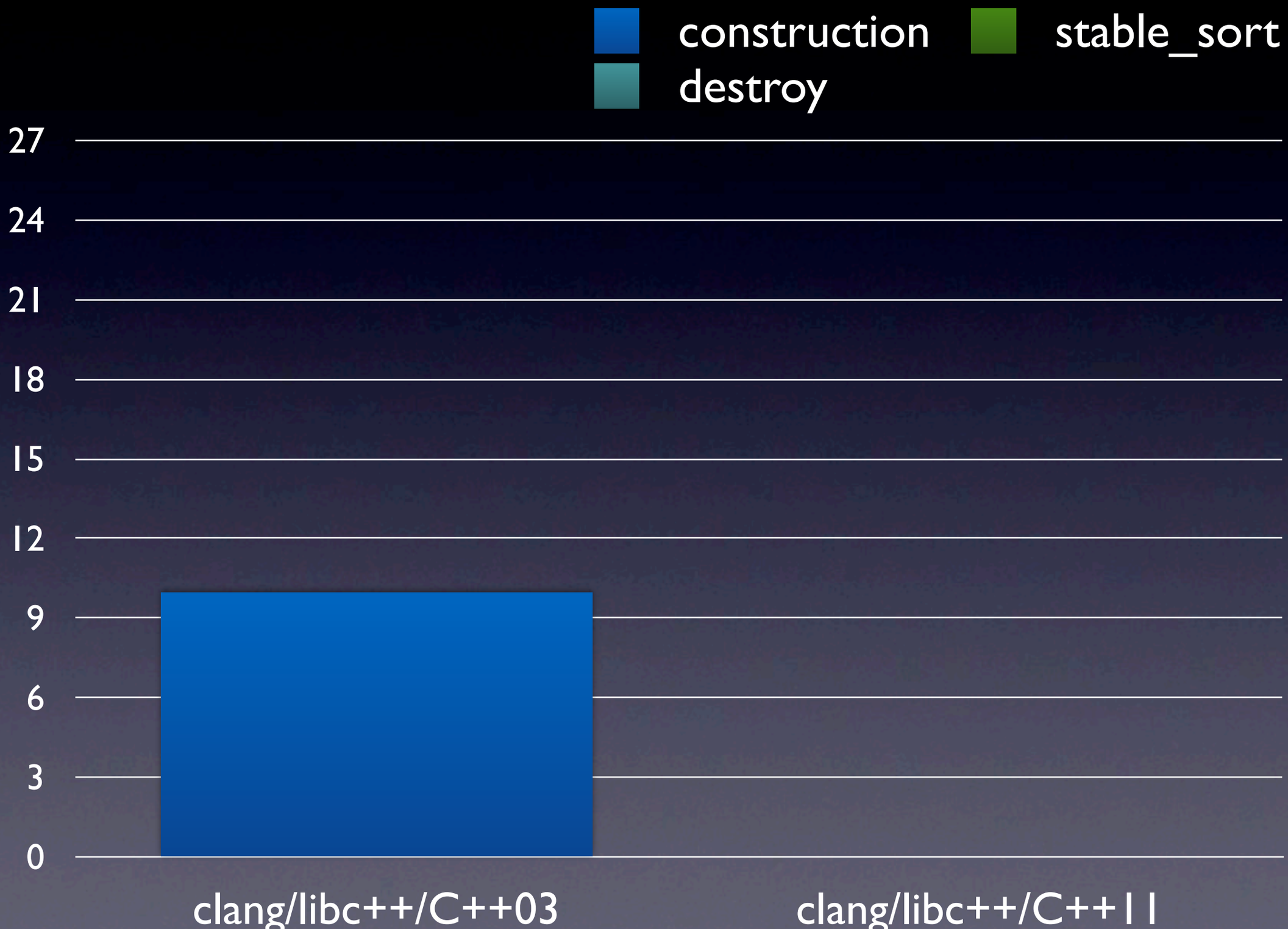
- In 2006, I wrote a benchmark to show off move semantics.
- It manipulated the unlikely data structure vector<set<int>>:
- Return it from factory functions.
- Manipulate it with algorithms.

Motivation for Move

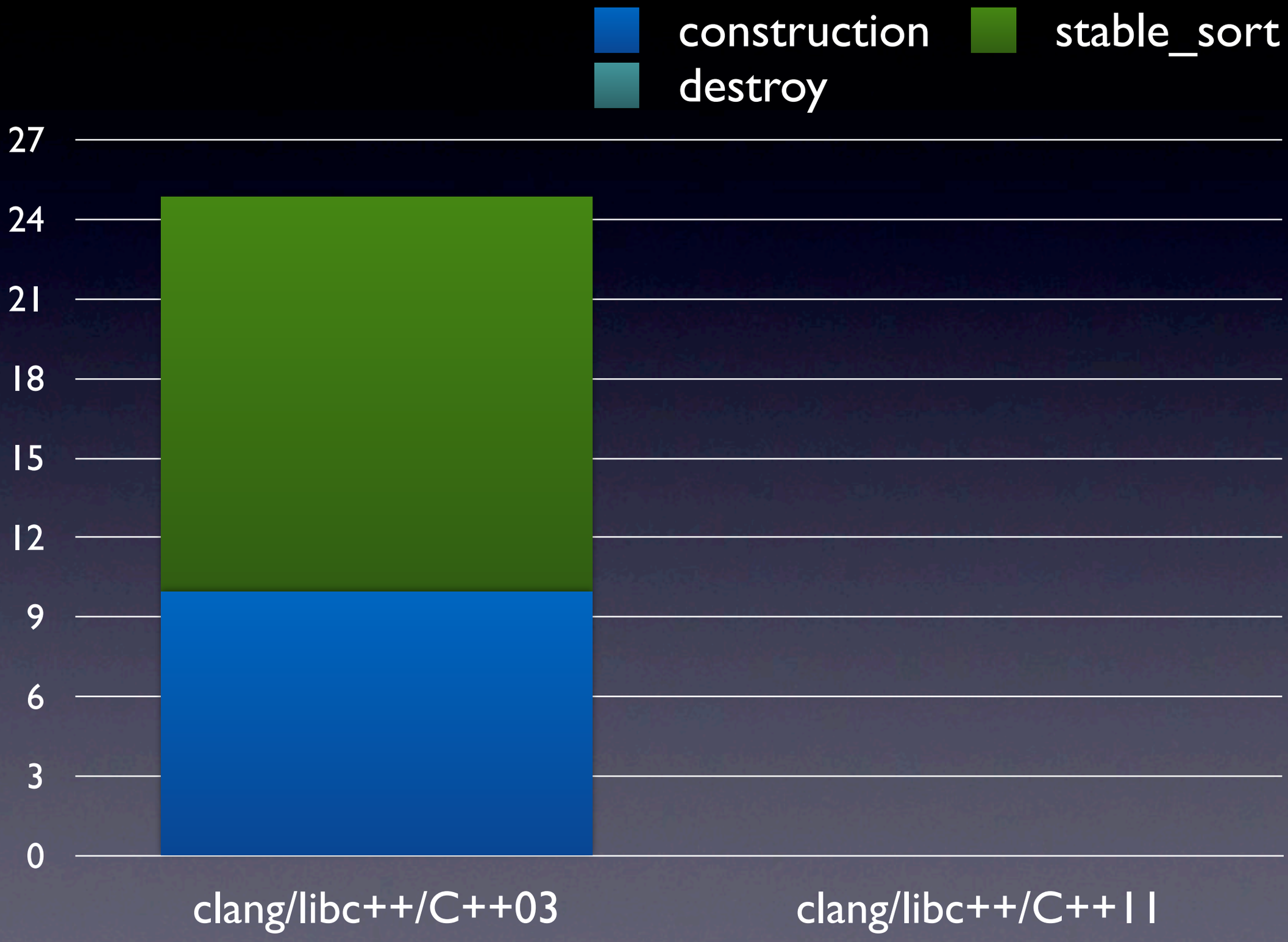
Motivation for Move



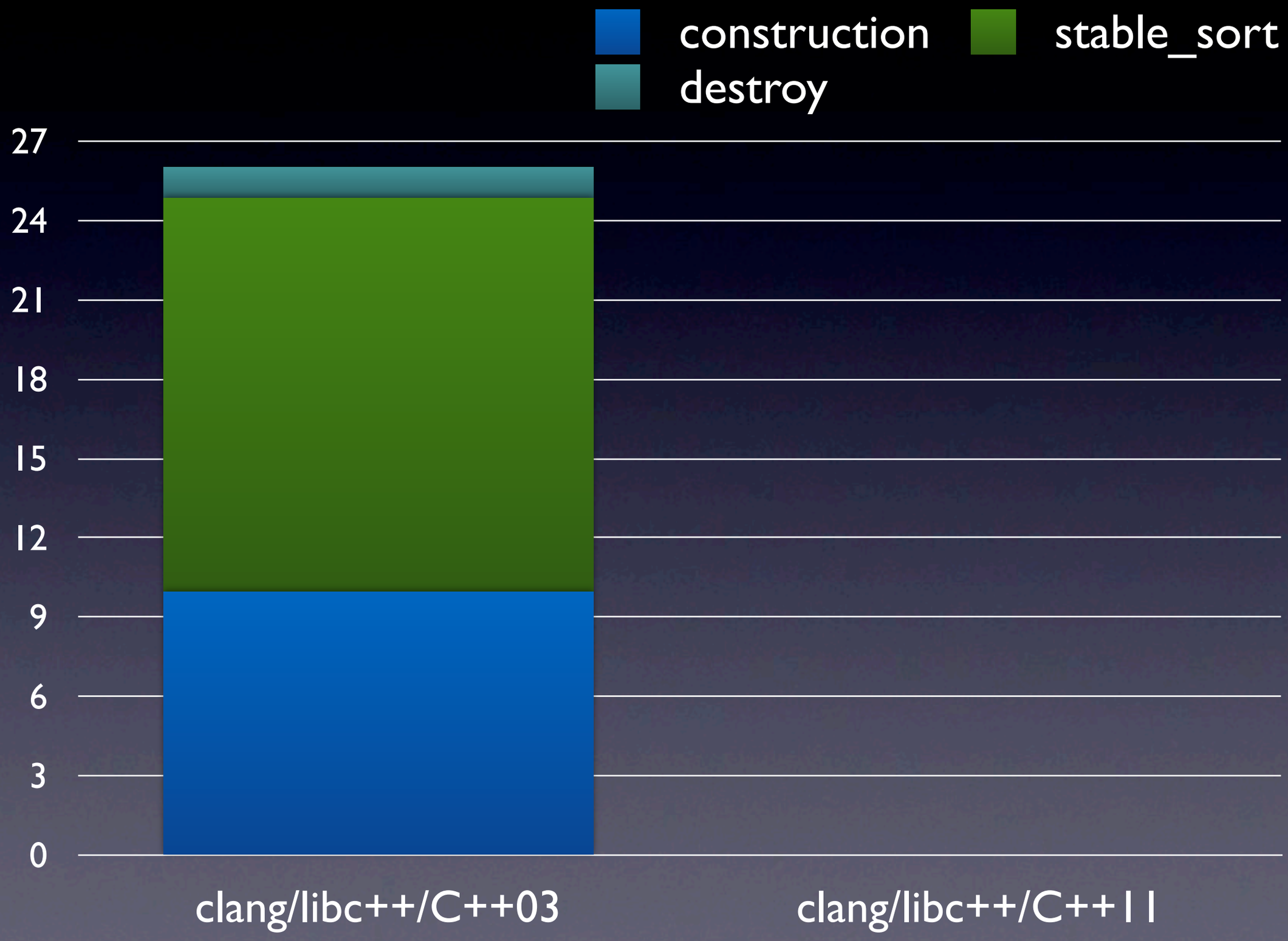
Motivation for Move



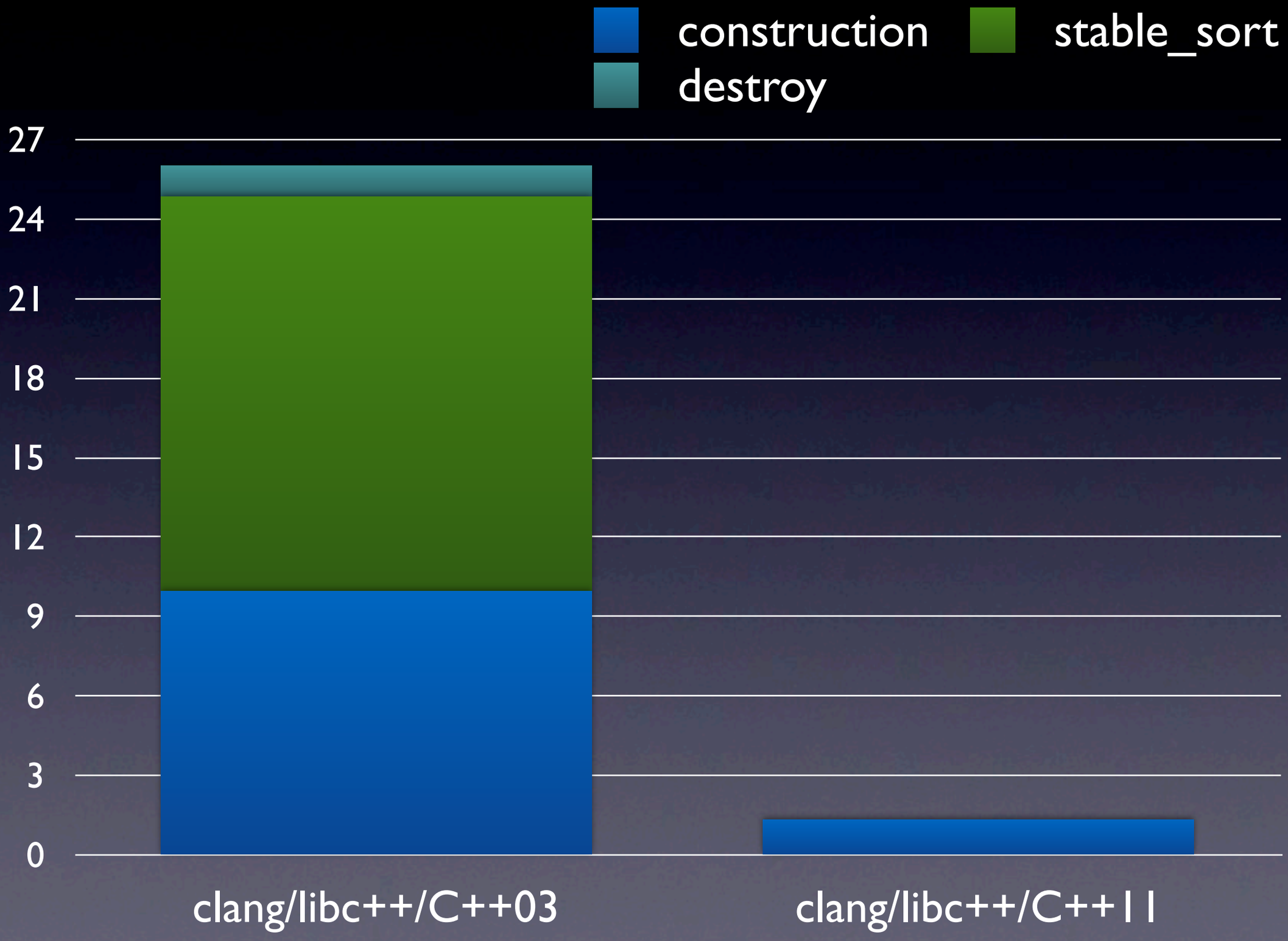
Motivation for Move



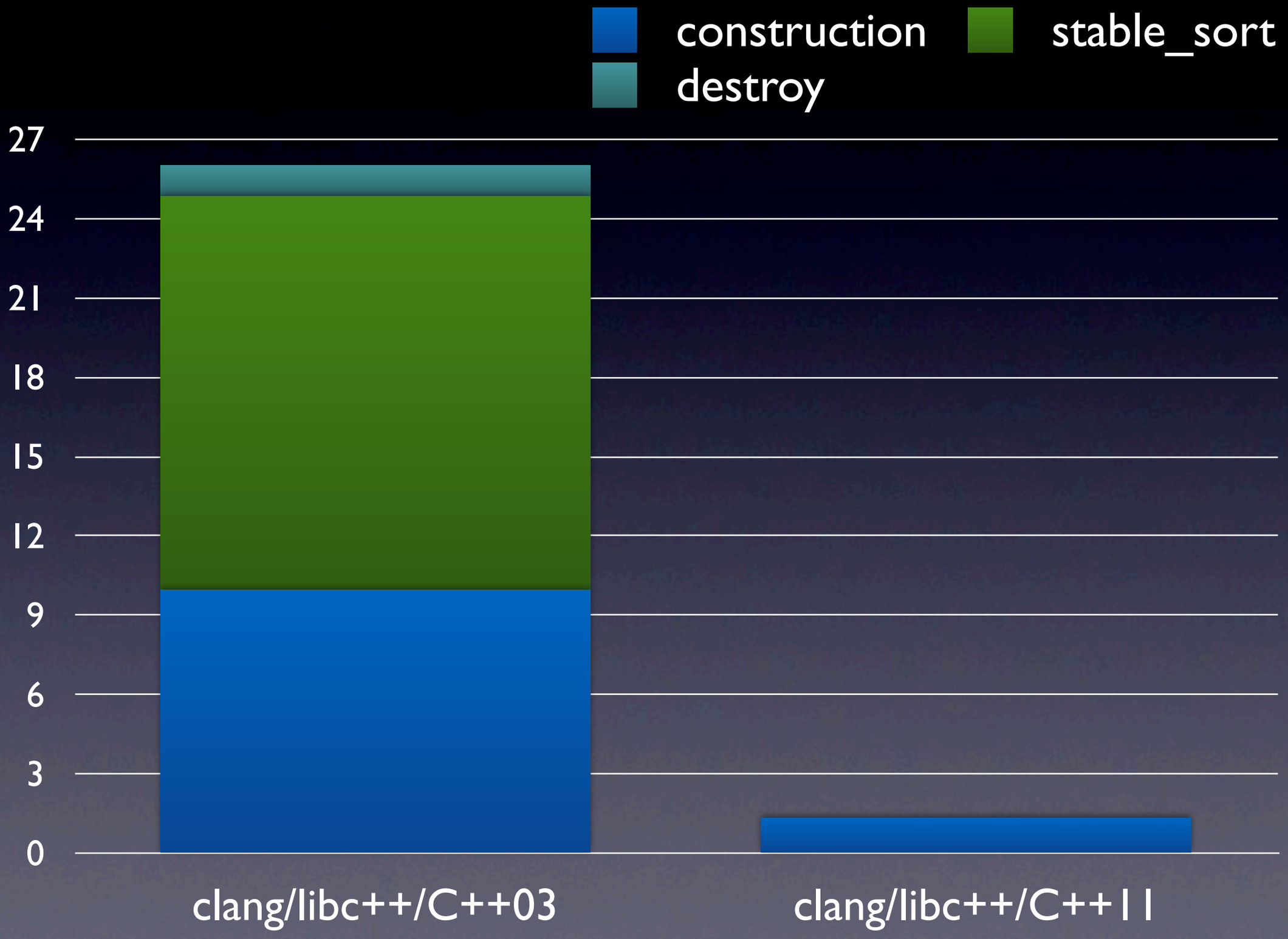
Motivation for Move



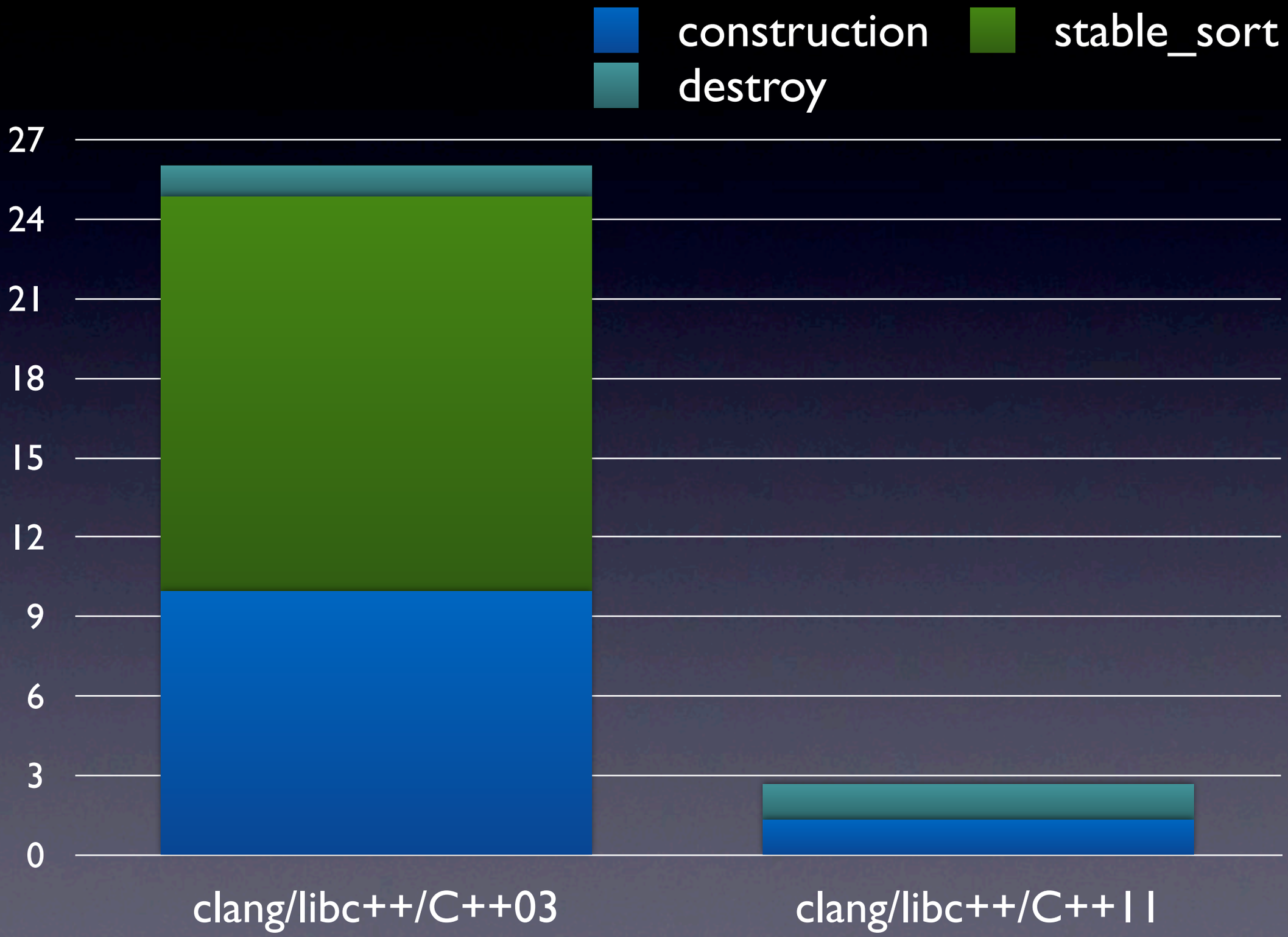
Motivation for Move



Motivation for Move



Motivation for Move



Motivation for Move

Motivation for Move

- With move semantics, vector<set<int>> does not have to be an unlikely data structure.

Motivation for Move

- With move semantics, vector<set<int>> does not have to be an unlikely data structure.
- Containers and algorithms can move around set<int> almost as cheaply as moving around an int.

Motivation for Move

- With move semantics, vector<set<int>> does not have to be an unlikely data structure.
- Containers and algorithms can move around set<int> almost as cheaply as moving around an int.
- And you can install move semantics in your "heavy" data structures.

Rvalue reference syntax

A&

Rvalue reference syntax

A&

- In C++03 we have the reference.

Rvalue reference syntax

A&

Rvalue reference syntax

A&

- In C++11 we renamed “reference” to “lvalue reference.”

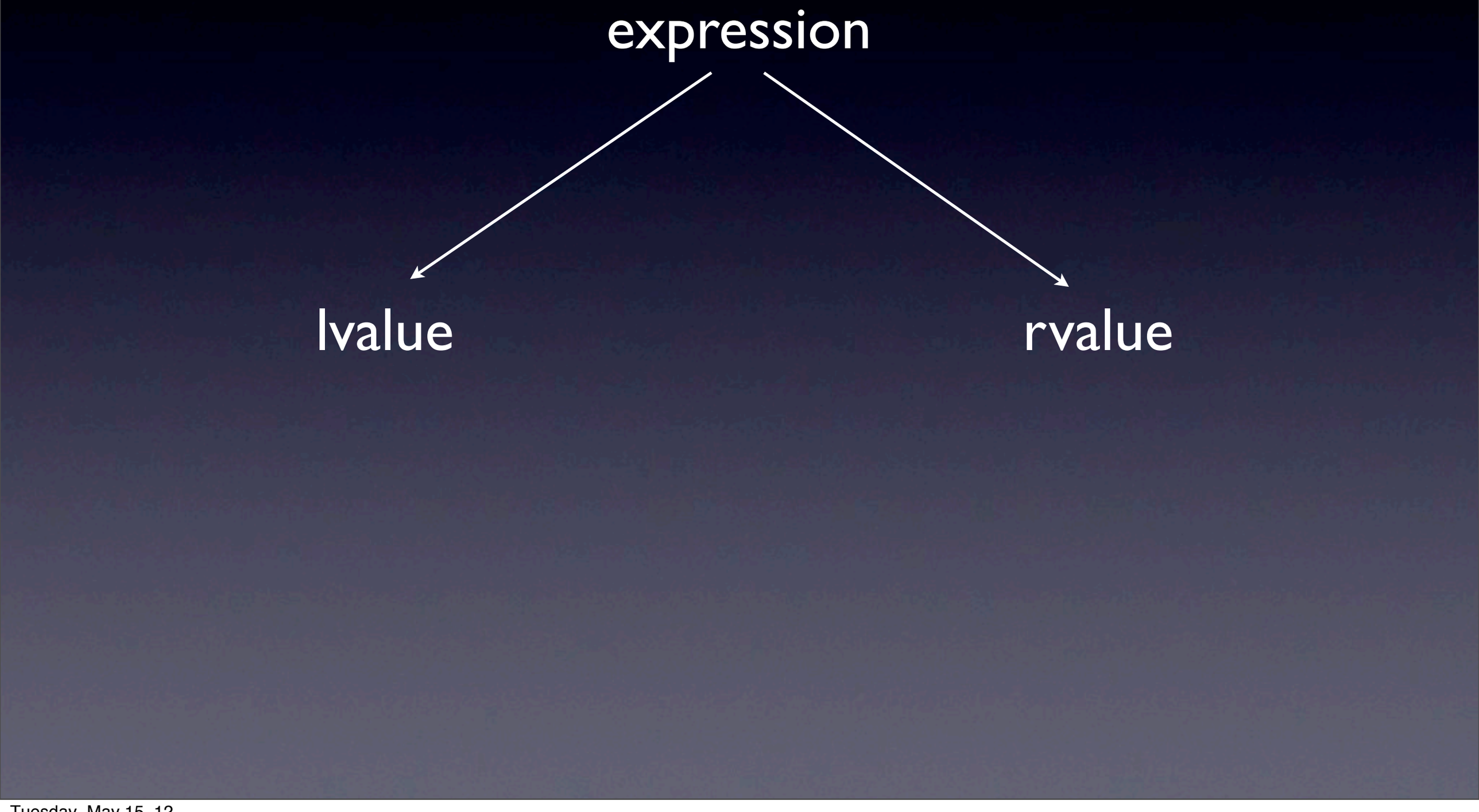
Rvalue reference syntax

A&

A&&

- In C++11 we renamed “reference” to “lvalue reference.”
- And we introduce a new kind of reference called “rvalue reference.”

Expressions

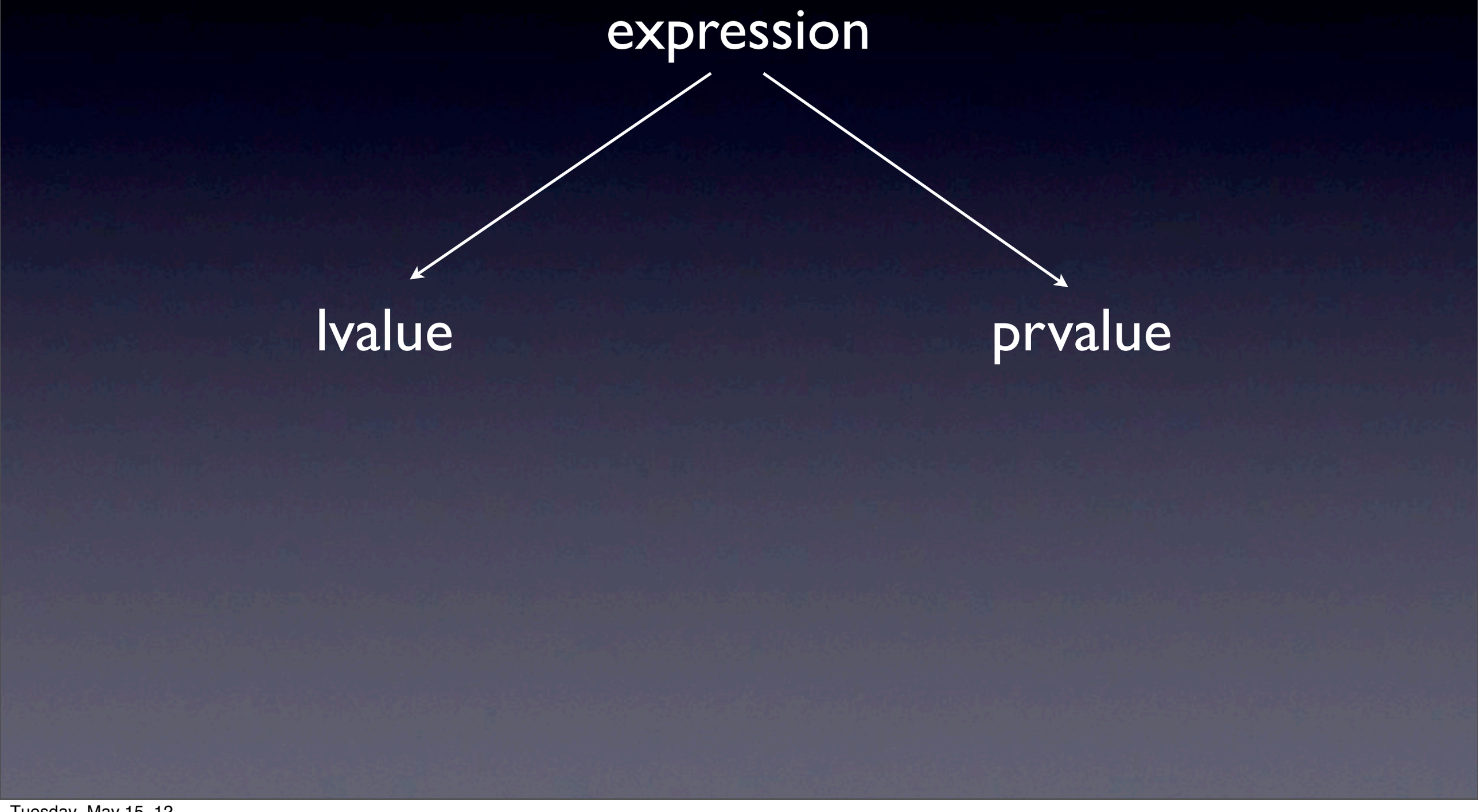


Expressions

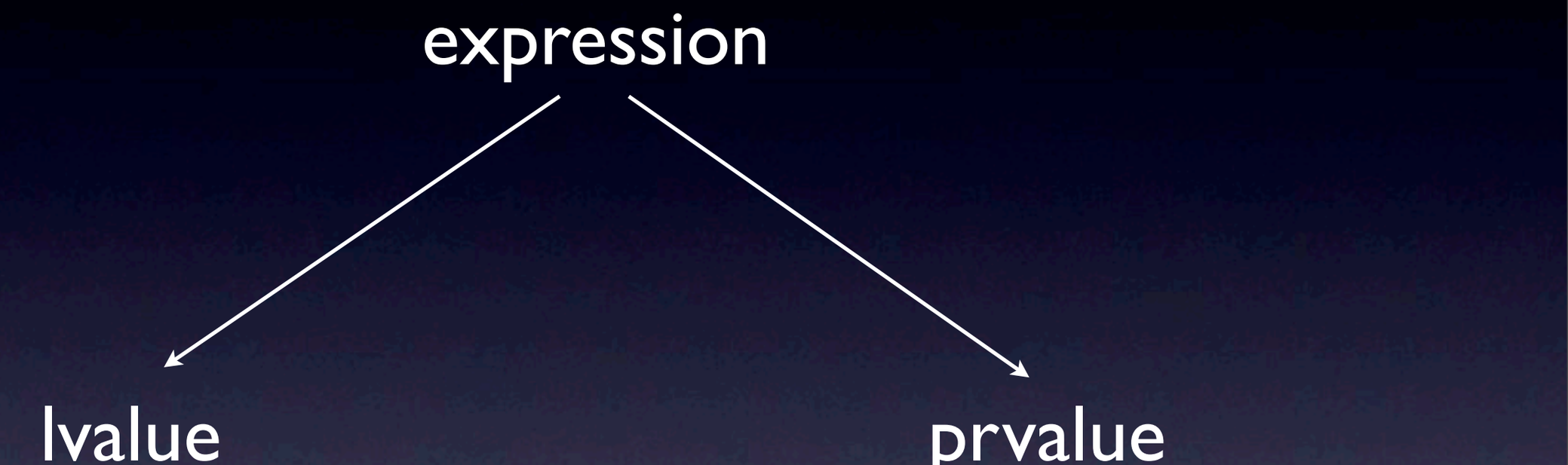
`static_cast<A&>(a)` `static_cast<A>(a)`

- In C++98/03 every expression is lvalue or rvalue.
- Expressions never have reference type.

Expressions

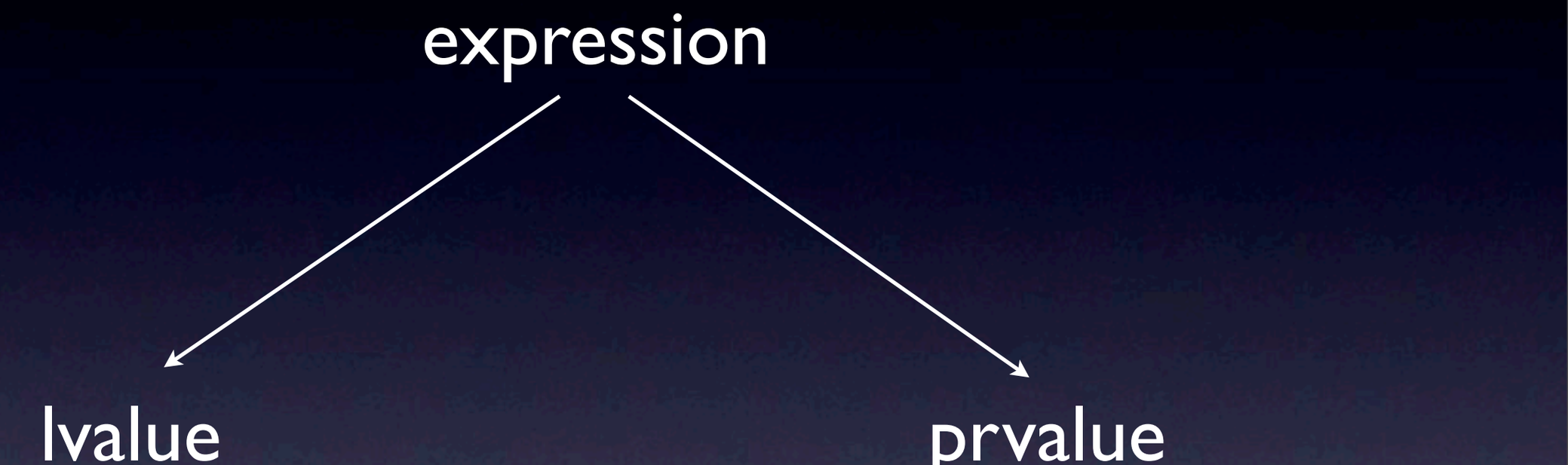


Expressions



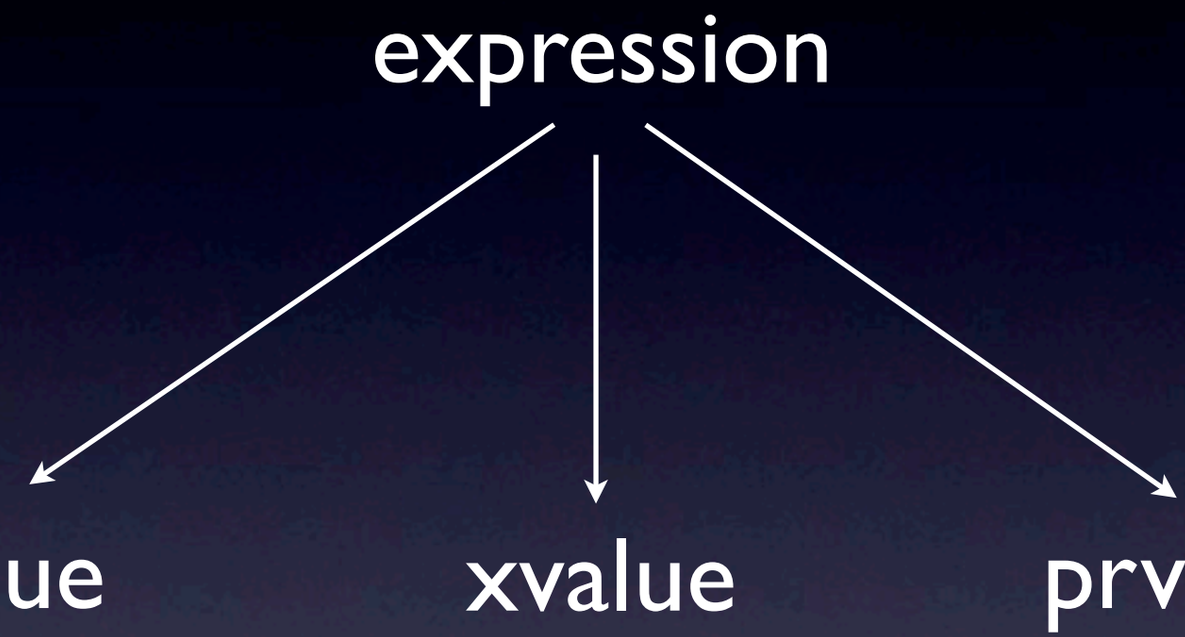
- In C++11 we renamed rvalue to prvalue.

Expressions



- In C++11 we renamed rvalue to prvalue.

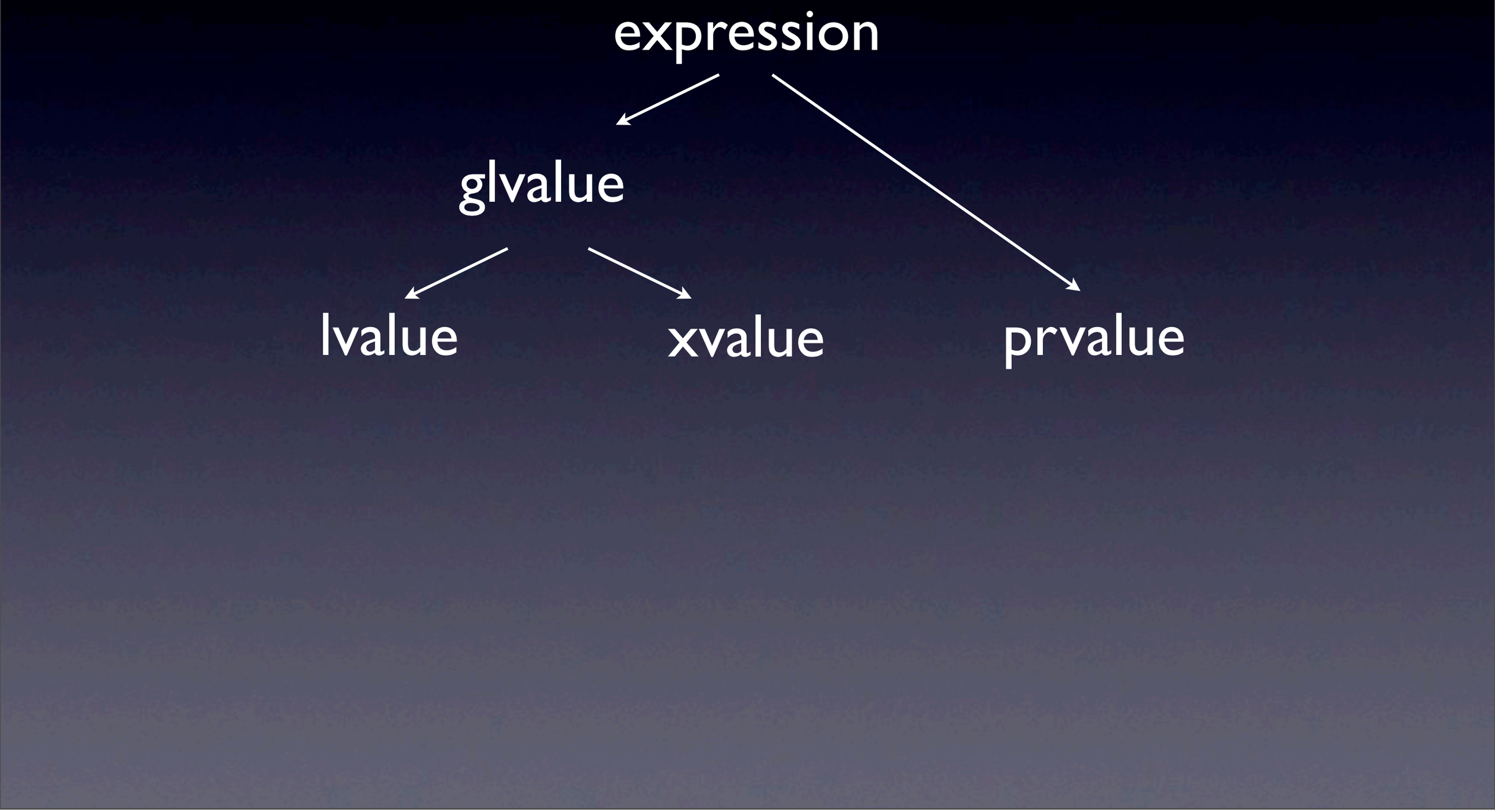
Expressions



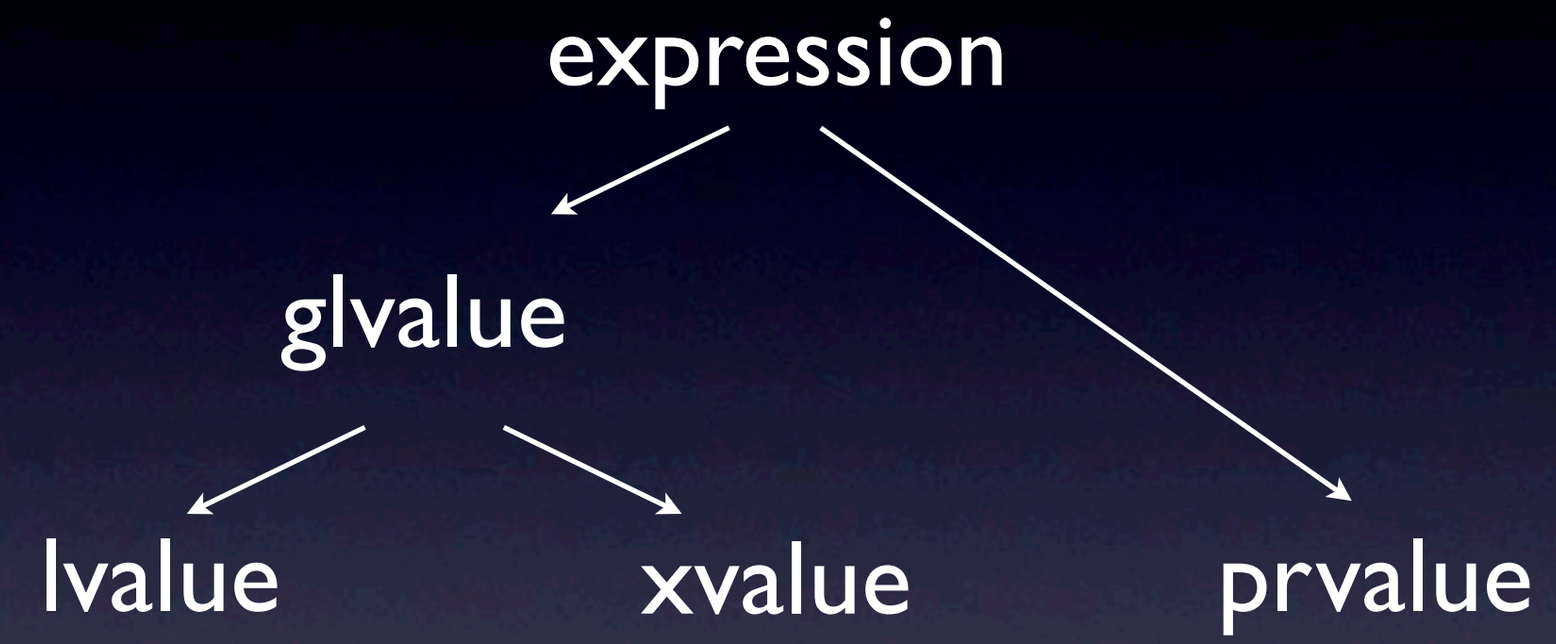
`static_cast<A&&>(a)`

- In C++11 we renamed rvalue to prvalue.
- And we added a new value category: xvalue.

Expressions

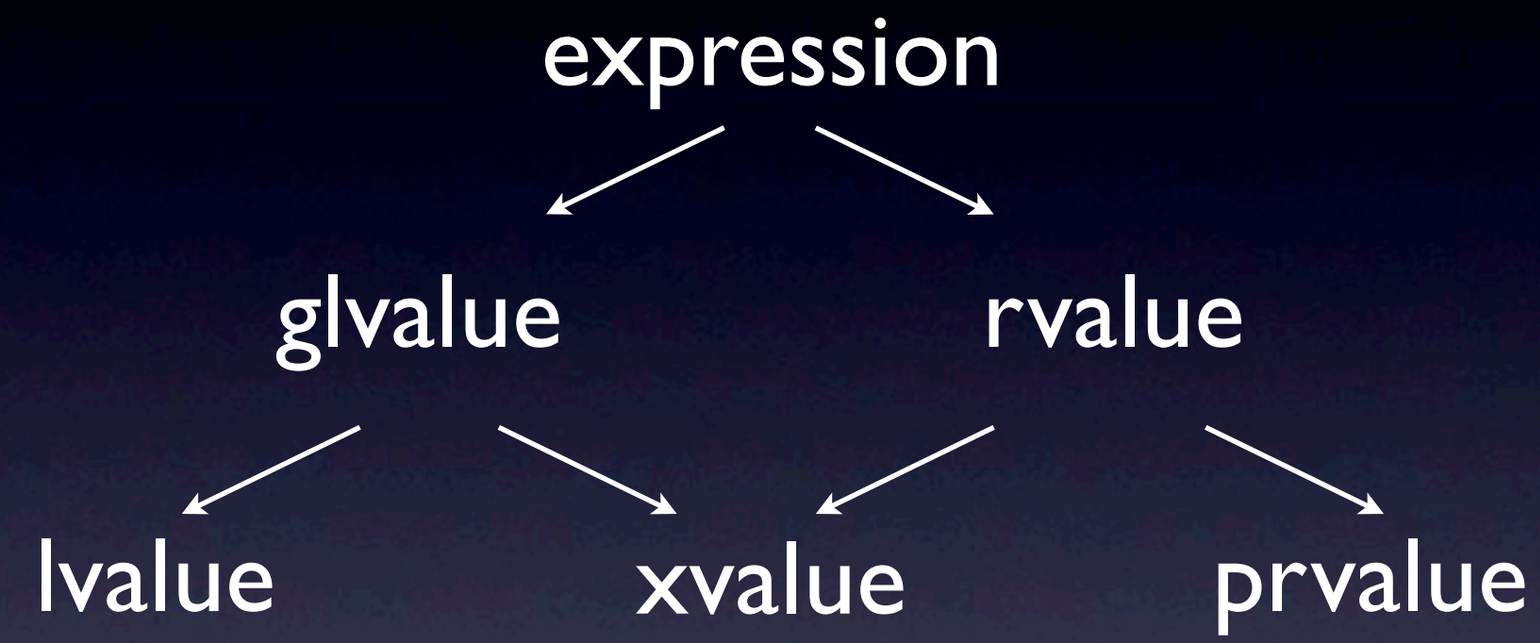


Expressions

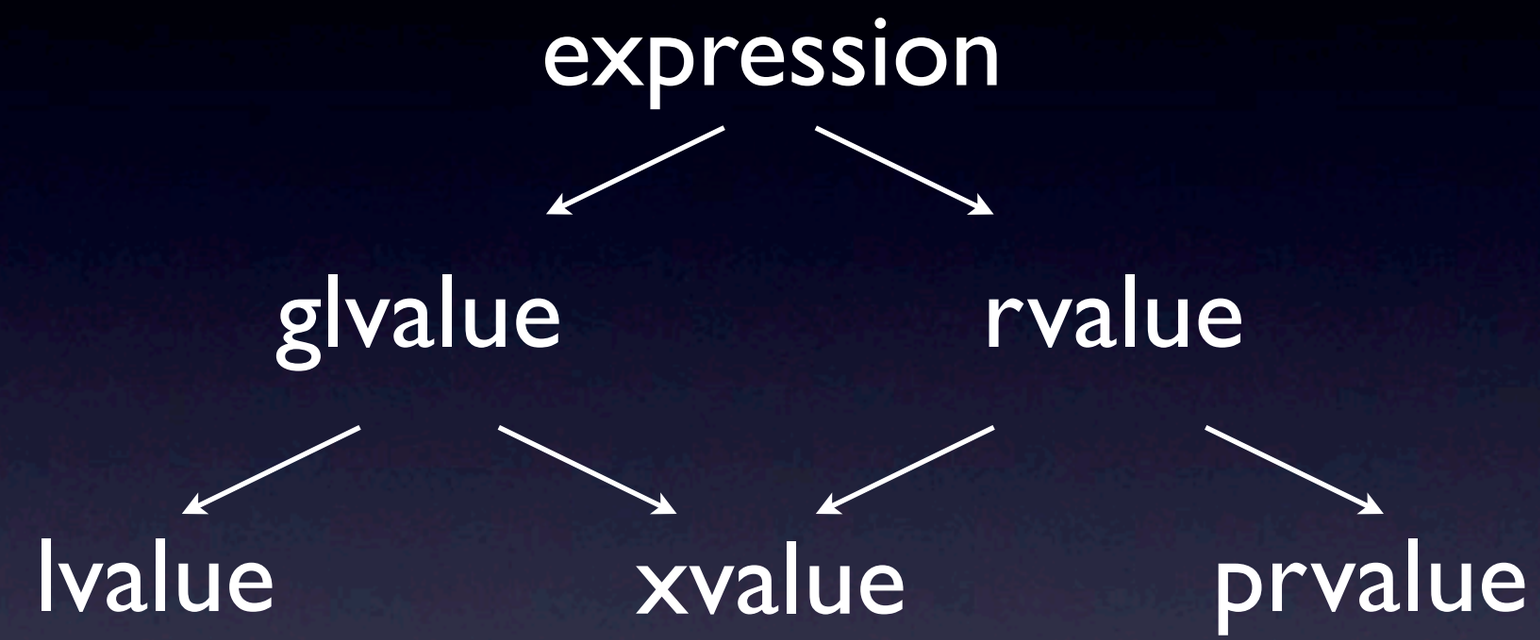


- A glvalue has a distinct address in memory.
- I.e. it has an identity.

Expressions



Expressions



- Only rvalues will bind to an rvalue reference.
- lvalues will not bind to an rvalue reference.

Binding

```
void f(A& i, A j, A&& k);
```

Binding

Binds to lvalues

```
void f(A& i, A j, A&& k);
```

Binding

Binds to lvalues

```
void f(A& i, A j, A&& k);
```

Special case: Will bind to rvalue if const A&

Binding

Binds to lvalues
and rvalues

Binds to lvalues

```
void f(A& i, A j, A&& k);
```

Special case: Will bind to rvalue if const A&

Binding

Binds to lvalues
and rvalues

Binds to lvalues

```
void f(A& i, A j, A&& k);
```

Special case: Will bind to rvalue if const A&

lvalues require copy
rvalues require move
(the move can be elided)

Binding

Binds to lvalues
and rvalues

Binds to lvalues

Binds to rvalues

```
void f(A& i, A j, A&& k);
```

prvalues
and
xvalues

Special case: Will
bind to rvalue if
const A&

lvalues require copy
rvalues require move
(the move can be elided)

Types & Expressions

```
void f(A& i, A j, A&& k)
{
}
```

Types & Expressions

```
void f(A& i, A j, A&& k)
{
}
```

- i is declared as type A&.

Types & Expressions

```
void f(A& i, A j, A&& k)
{
    i; // lvalue A
}
```

- i is declared as type A&.
  - The **expression** i has type A and is an lvalue.

Types & Expressions

```
void f(A& i, A j, A&& k)
{
    i; // lvalue A
}
```

- j is declared as type A.

Types & Expressions

```
void f(A& i, A j, A&& k)
{
    i; // lvalue A
    j; // lvalue A
}
```

- j is declared as type A.
  - The **expression** j has type A and is an lvalue.

Types & Expressions

```
void f(A& i, A j, A&& k)
{
    i; // lvalue A
    j; // lvalue A
}
```

- k is declared as type A&&.

Types & Expressions

```
void f(A& i, A j, A&& k)
{
    i; // lvalue A
    j; // lvalue A
    k; // lvalue A
}
```

- k is declared as type A&&.
  - The **expression** k has type A and is an lvalue.

Types & Expressions

```
void f(A& i, A j, A&& k)
{
}
```

Types & Expressions

```
void g(A&);   // #1
void g(A&&);  // #2

void f(A& i, A j, A&& k)
{

}
```

Types & Expressions

```
void g(A&);   // #1
void g(A&&);  // #2
void f(A& i, A j, A&& k)
{
   g(i);  // calls #1
}
```

Types & Expressions

```
void g(A&);   // #1
void g(A&&);  // #2
void f(A& i, A j, A&& k)
{
    g(i);  // calls #1
    g(j);  // calls #1

}
```

Types & Expressions

```
void g(A&);   // #1
void g(A&&);  // #2
void f(A& i, A j, A&& k)
{
    g(i);  // calls #1
    g(j);  // calls #1
    g(k);  // calls #1
}
```

- The expression k is an lvalue A

Types & Expressions

```
void g(A&);   // #1
void g(A&&);  // #2
void f(A& i, A j, A&& k)
{
    g(static_cast<A&&>(i));  // calls #2
    g(static_cast<A&&>(j));  // calls #2
    g(static_cast<A&&>(k));  // calls #2
}
```

- An lvalue expression can be cast to an rvalue (xvalue) expression

Types & Expressions

```
void g(A&);   // #1
void g(A&&);  // #2
void f(A& i, A j, A&& k)
{
    g(std::move(i));  // calls #2
    g(std::move(j));  // calls #2
    g(std::move(k));  // calls #2
}
```

- Use std::move to perform the cast for better readability.

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Observation

```
void f(A& i, A j, A&& k)
{
  // i is not a unique reference
  // j is a unique reference
  // k is a reference to xvalue or prvalue
}
```

Observation

```
void f(A& i, A j, A&& k)
{
  // i is not a unique reference
  // j is a unique reference
  // k is a reference to xvalue or prvalue
}
```

- f() can do anything it wants to j, as long as the object remains destructible.

Observation

```
void f(A& i, A j, A&& k)
{
  // i is not a unique reference
  // j is a unique reference
  // k is a reference to xvalue or prvalue
}
```

- f() can do anything it wants to j, as long as the object remains destructible.
- f() can do anything it wants to k, as long as k references a prvalue.

Observation

```
void f(A& i, A j, A&& k)
{
  // i is not a unique reference
  // j is a unique reference
  // k is a reference to xvalue or prvalue
}
```

- f() can do anything it wants to j, as long as the object remains destructible.
- f() can do anything it wants to k, as long as k references a prvalue.
- Convention: Do not cast an lvalue to an xvalue unless you want that object to be treated as a prvalue.

The move constructor

```
class A
{
    int* data_;          // heap allocated
public:
    A(const A& a);       // copy constructor
};
```

The move constructor

```
class A
{
    int* data_;          // heap allocated
public:
    A(const A& a);       // copy constructor
};
```

- copy constructor binds to an lvalue and copies resources.

The move constructor

```
class A
{
    int* data_;          // heap allocated
public:
    A(const A& a);       // copy constructor
    A(A&& a) noexcept // move constructor
      : data_(a.data_)
      { a.data_ = nullptr;}
};
```

- copy constructor binds to an lvalue and copies resources.
- move constructor binds to a rvalue and pilfers resources.

The move constructor

```
A make_A();

A a1;
A a2 = a1;                    // Calls copy ctor
```

The move constructor

```
A make_A();

A a1;
A a2 = a1;                  // Calls copy ctor
A a3 = make_A();            // Calls (or elides)
                            //       move ctor
```

The move constructor

```
A make_A();

A a1;
A a2 = a1;                  // Calls copy ctor
A a3 = make_A();            // Calls (or elides)
                            //       move ctor
A a4 = std::move(a1);       // Calls move ctor
```

The move constructor

```
A make_A();

A a1;
A a2 = a1;                  // Calls copy ctor
A a3 = make_A();            // Calls (or elides)
                            //       move ctor
A a4 = std::move(a1);       // Calls move ctor
```

- "Copies" from rvalues are made with the move constructor, which does nothing but trade pointers. *Fast!*

The move constructor

The move constructor

- If a class does not have a move constructor, its copy constructor will be used to copy from rvalues (just as in C++98/03).

The move constructor

- If a class does not have a move constructor, its copy constructor will be used to copy from rvalues (just as in C++98/03).
- Scalars move the same as they copy.

The move constructor

The move constructor

```
struct A
{
    A(const A& a);
    A(A&&) = default;
};
```

- Copy and move constructors can be explicitly defaulted.
  - The default copies/moves each base and data member (unless it is defined as deleted).

The move constructor

The move constructor

```
struct member
{
   member(const member&);

};
struct A
{
     member m_;
     A(A&&) = default; // deleted
};
```

- A defaulted move constructor is defined as deleted if:
  - there is a base or member with no move constructor and it is not trivially copyable.

The move constructor

```
struct member
{
   member(const member&)= default;

};
struct A
{
     member m_;
     A(A&&) = default;
};
```

- A defaulted move constructor is defined as deleted if:
  - there is a base or member with no move constructor and it is not trivially copyable.

The move constructor

```
struct member
{
    member(const member&);
    member(member&&);
};
struct A
{
    member m_;
    A(A&&) = default;
};
```

- A defaulted move constructor is defined as deleted if:
  - there is a base or member with no move constructor and it is not trivially copyable.

The move constructor

```
struct member
{
   member(const member&);

};
struct A
{
     member m_;
     A(A&&) = default;
};
```

The move constructor

```
struct member
{
   member(const member&);

};
struct A
{
     member m_;
     A(A&&) = default;
};
```

- CWG issue 1402 (ready) changes the rules such that the defaulted move members will **not** be implicitly deleted, but instead copy the bases and members.

The move constructor

The move constructor

```
static_assert
(
  std::is_move_constructible<A>::value,
  "A should be move constructible"
);
```

- You can always test at compile time if a complete type is move constructible.

The move constructor

```
static_assert
(
  std::is_move_constructible<A>::value,
  "A should be move constructible"
);
```

- You can always test at compile time if a complete type is move constructible.
- This tests whether or not A is constructible from an rvalue A, not if A has a move constructor.
  - But a type with a deleted move constructor is never move constructible.

The move constructor

The move constructor

```
struct A
{
    A(const A& a) = delete;
    A(A&&)        = default;
};
```

- Copy and move constructors can be explicitly deleted.

The move constructor

```
struct A
{
    A(const A& a) = default;
    A(A&&)        = delete;
};
```

- Copy and move constructors can be explicitly deleted.
  - A deleted move constructor will prohibit copying from rvalues (rarely a good idea). Normally omit rather than delete a move constructor.

The move constructor

```
struct A
{
    A(const A& a) = default;

};
```

- Copy and move constructors can be explicitly deleted.
  - A deleted move constructor will prohibit copying from rvalues (rarely a good idea). Normally omit rather than delete a move constructor.

The move constructor

The move constructor

```
struct A
{
    // A(const A&) = delete;
    // A& operator=(const A&) = delete;
    A(A&&);
};
```

- A user-declared move constructor (defaulted or not) will implicitly create a deleted copy constructor and copy assignment.

Implicit Special Members

```
class A
{
   std::string s_;
public:
   // A() noexcept = default;
   // A(const A&) = default;
   // A& operator=(const A&) = default;
   // A(A&&) noexcept = default;
   // A& operator=(A&&) noexcept = default;
   // ~A() noexcept = default;
};
```

noexcept is extension

- Comments indicate compiler supplied definitions.

Implicit Special Members

```
class A
{
   std::string s_;
public:
   A();
   // A(const A&) = default;
   // A& operator=(const A&) = default;
   // A(A&&) noexcept = default;
   // A& operator=(A&&) noexcept = default;
   // ~A() noexcept = default;
};
```

- Comments indicate compiler supplied definitions.

Implicit Special Members

```
class A
{
   std::string s_;
public:

   A(const A&);
   // A& operator=(const A&) = default;

   // ~A() noexcept = default;
};
```

deprecated

- Comments indicate compiler supplied definitions.

Implicit Special Members

```
class A
{
   std::string s_;
public:
   // A() noexcept = default;
   // A(const A&) = default;
   A& operator=(const A&) = default;

   // ~A() noexcept = default;
};
```

noexcept is extension

deprecated

- Comments indicate compiler supplied definitions.

Implicit Special Members

```
class A
{
   std::string s_;
public:
   // A() noexcept = default;
   // A(const A&) = default;
   // A& operator=(const A&) = default;

   ~A();
};
```

noexcept is extension

deprecated

- Comments indicate compiler supplied definitions.

Implicit Special Members

```
class A
{
   std::string s_;
public:

   // A(const A&) = delete;
   // A& operator=(const A&) = delete;
   A(A&&);

   // ~A() noexcept = default;
};
```

- Comments indicate compiler supplied definitions.

Implicit Special Members

```
class A
{
   std::string s_;
public:
   // A() noexcept = default;
   // A(const A&) = delete;
   // A& operator=(const A&) = delete;

   A& operator=(A&&);
   // ~A() noexcept = default;
};
```

noexcept is extension

- Comments indicate compiler supplied definitions.

Advice

- Put these (or other appropriate) tests right into your release code:

```
struct A
{
    std::string s_;
    std::vector<int> v_;
};
```

Advice

- Put these (or other appropriate) tests right into your release code:

```
struct A
{
    std::string s_;
    std::vector<int> v_;
};

// Howard says put these tests in!
static_assert(std::is_nothrow_default_constructible<A>::value, "");
static_assert(std::is_copy_constructible<A>::value, "");
static_assert(std::is_copy_assignable<A>::value, "");
static_assert(std::is_nothrow_move_constructible<A>::value, "");
static_assert(std::is_nothrow_move_assignable<A>::value, "");
static_assert(std::is_nothrow_destructible<A>::value, "");
```

Advice

- Put these (or other appropriate) tests right into your release code:

```
struct A
{
    std::string s_;
    std::vector<int> v_;
    A(const A&) = default;
};

// Howard says put these tests in!
static_assert(std::is_nothrow_default_constructible<A>::value, "");
static_assert(std::is_copy_constructible<A>::value, "");
static_assert(std::is_copy_assignable<A>::value, "");
static_assert(std::is_nothrow_move_constructible<A>::value, "");
static_assert(std::is_nothrow_move_assignable<A>::value, "");
static_assert(std::is_nothrow_destructible<A>::value, "");
```

Advice

- Put these (or other appropriate) tests right into your release code:

```
struct A
{
    std::string s_;
    std::vector<int> v_;
    A(const A&) = default;
};

// Howard says put these tests in! Or else!!!

static_assert(std::is_copy_constructible<A>::value, "");
static_assert(std::is_copy_assignable<A>::value, "");


static_assert(std::is_nothrow_destructible<A>::value, "");
```

The move assignment operator

The move assignment operator

- Everything that's been said about the move constructor applies to the move assignment operator.

The move assignment operator

```
class A
{
    int* data_;          // heap allocated
public:
    A& operator=(const A& a);    // copy
};
```

The move assignment operator

```
class A
{
    int* data_;          // heap allocated
public:
    A& operator=(const A& a);    // copy
};
```

- copy assignment binds to lvalue rhs and copies resources.

The move assignment operator

```
class A
{
    int* data_;          // heap allocated
public:
    A& operator=(const A& a);    // copy
    A& operator=(A&& a) noexcept // move
    {
       std::swap(data_, a.data_);
       return *this;
    }
};
```

- copy assignment binds to lvalue rhs and copies resources.
- move assignment binds to rvalue rhs and does whatever is fastest to assume value of rhs.

The move assignment operator

The move assignment operator

```
class A
{
    fstream f_;
public:
    A& operator=(A&& a) noexcept
    {
        f_ = std::move(a.f_);
        return *this;
    }
};
```

- If your type holds std::lib components, move assigning those data members will generally do the right thing.

The move assignment operator

```
class A
{
    fstream f_;
public:
    A& operator=(A&& a) = default;
};
```

- If all you need to do is move assign bases and members, consider doing it with “= default”.

The move assignment operator

```
class A
{
    fstream f_;
public:

};
```

- Or doing it implicitly.

The move assignment operator

```
template <class T>
class A {
    T* data_;          // heap allocated
public:
    A& operator=(A&& a) noexcept {
        delete data_;
        data_ = a.data_;
        a.data_ = nullptr;
        return *this;
    }
};
```

- Does the move assignment operator need to check for self-assignment?

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

- Convention: Do not cast an lvalue to an xvalue unless you want that object to be treated as a prvalue.

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

- Convention: Do not cast an lvalue to an xvalue unless you want that object to be treated as a prvalue.
- If ‘a’ refers to a prvalue, then it is not possible for ‘this’ and ‘a’ to refer to the same object.

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}

a = std::move(a);
```

- However if ‘a’ refers to an xvalue, then it is possible for ‘this’ and ‘a’ to refer to the same object.

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}

a = std::move(a);
```

- However if ‘a’ refers to an xvalue, then it is possible for ‘this’ and ‘a’ to refer to the same object.
- But you’ve arguably broken convention.

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

- Self-swap is one place where this can happen.

```
template <class T>
void swap(T& x, T& y) {
  T tmp(std::move(x));
  x = std::move(y);
  y = std::move(tmp);
}

std::swap(a, a);
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

```
template <class T>
void swap(T& x, T& y) {
  T tmp(std::move(x));
  x = std::move(y);
  y = std::move(tmp);
}

  std::swap(a, a);
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

- However in this case, the self-move assignment happens only on a moved-from value.

```
template <class T>
void swap(T& x, T& y) {
  T tmp(std::move(x));
  x = std::move(y);
  y = std::move(tmp);
}

  std::swap(a, a);
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

```
template <class T>
void swap(T& x, T& y) {
  T tmp(std::move(x));
  x = std::move(y);
  y = std::move(tmp);
}

  std::swap(a, a);
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

- Self-move assignment from a moved-from value is most often naturally safe.

```
template <class T>
void swap(T& x, T& y) {
  T tmp(std::move(x));
  x = std::move(y);
  y = std::move(tmp);
}

  std::swap(a, a);
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

The move assignment operator

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

- Indeed, in all permutation rearrangement algorithms (those that do not “remove” elements), the target of a move assignment is always in a “moved-from” state.

The move assignment operator

The move assignment operator

- I treat self-move assignment and self-swap as a performance bug in my code.

The move assignment operator

- I treat self-move assignment and self-swap as a performance bug in my code.
- My move assignment operators are "self-safe" only when in a moved-from state.

The move assignment operator

- I treat self-move assignment and self-swap as a performance bug in my code.
- My move assignment operators are "self-safe" only when in a moved-from state.
- Feel free to check for self-move assignment in your code.

The move assignment operator

- I treat self-move assignment and self-swap as a performance bug in my code.
- My move assignment operators are "self-safe" only when in a moved-from state.
- Feel free to check for self-move assignment in your code.
  - Either with an if - to ignore the bug.

The move assignment operator

- I treat self-move assignment and self-swap as a performance bug in my code.
- My move assignment operators are "self-safe" only when in a moved-from state.
- Feel free to check for self-move assignment in your code.
  - Either with an if - to ignore the bug.
  - Or with an assert - to catch the bug.

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
       swap(a);
       return *this;
    }
};
```

- In C++98/03 it became popular to define assignment using a copy/swap pattern.
- This is very good if you need strong exception safety.

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
        swap(a);
        return *this;
    }
};
```

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
       swap(a);
       return *this;
    }
};
```

- It is very efficient when assigning from rvalues.

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
       swap(a);
       return *this;
    }
};
```

- It is very efficient when assigning from rvalues.
- It is not so efficient when assigning from lvalues.

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
        swap(a);
        return *this;
    }
};
```

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
       swap(a);
       return *this;
    }
};
```

- Strong exception safety is good, but it is not free.

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
       swap(a);
       return *this;
    }
};
```

- Strong exception safety is good, but it is not free.
- Do not pay for it (performance) if you do not need it.

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
        swap(a);
        return *this;
    }
};
```

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
    A& operator=(A a) {
       swap(a);
       return *this;
    }
};
```

- The fatal assumption here is that:

  ```
  A& operator=(const A&) = default;
  ```

  is always about the same speed as:

  ```
  A(const A&) = default;
  ```

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
   A& operator=(const A& a) = default;
   A& operator=(A&& a) = default;

};
```

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
   A& operator=(const A& a) = default;
   A& operator=(A&& a) = default;

};
```

- This copy assignment can be much faster! (2X, 5X, even 7X)

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
   A& operator=(const A& a) = default;
   A& operator=(A&& a) = default;

};
```

- This copy assignment can be much faster! (2X, 5X, even 7X)
- This move assignment is just as fast.

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
   A& operator=(const A& a) = default;
   A& operator=(A&& a) = default;

};
```

Assignment boo boo's

```
class A
{
   std::vector<int> v_;
   std::string s_;
public:
   A& operator=(const A& a) = default;
   A& operator=(A&& a) = default;

};
```

- Prefer defaulted assignment operators.

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Return By Value

```
A
make()
{
   A a;
   // ...
   return a;
}
```

Return By Value

```
A
make()
{
   A a;
   // ...
   return a; // 1. RVO.
}
```

Return By Value

```
A
make()
{
   A a;
   // ...
   return a; // 1. RVO.
}            // 2. Return as if an rvalue.
```

Return By Value

```
A
make()
{
   A a;
   // ...
   return a;  // 1. RVO.
}             // 2. Return as if an rvalue.
              // 3. Return as if an lvalue.
```

- Move semantics makes factory functions efficient.

Return By Value

```
A&&
make()
{
   A a;
   // ...
   return a;
}
```

Return By Value

```
A&&   // Wrong!!
make()
{
   A a;
   // ...
   return a;
}
```

- Do not return a reference to a local object.
  - Not even an rvalue reference.

Return By Value

```
A
make()
{
   A a;
   // ...
   return std::move(a);
}
```

Return By Value

```
A
make()
{
   A a;
   // ...
   return std::move(a);   // Wrong!!
}
```

- Explicitly using std::move will inhibit RVO.

Return By Value

```
template <class ...T>
ifstream
prepare(const string& filename,
        const T& ...t)
{
  create(filename, t...);
  return ifstream(filename);
}
```

- Now you can return “move-only” types from factory functions.

Return a Modified Copy

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

```
A
modify(A a)
{
   a.modify();
   return a;   // 1. Return as if an rvalue.
}              // 2. Return as if an lvalue.
```

- Returning a by-value parameter by-value will also implicitly move.

Return a Modified Copy

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

- There has been a lot of talk lately about which of these designs is “better.”

Return a Modified Copy

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

- There has been a lot of talk lately about which of these designs is “better.”
- Today we will measure and provide quantitative results.

Return a Modified Copy

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

- How many copy constructions?
- How many move constructions?

Return a Modified Copy

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

```
A
modify(const A& a)
{
  A tmp(a);
  tmp.modify();
  return tmp;
}
```

```
A
modify(A a)
{
  a.modify();
  return a;
}
```

- How does the value category (lvalue/rvalue) of the argument impact the results?
- Is the answer different in C++03 than in C++11?

Return a Modified Copy

in C++03

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

Return a Modified Copy

in C++03

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

here

Assuming RVO for all cases

- lvalue: 1 copy construction
- rvalue: 1 copy construction

Return a Modified Copy

C++03	lvalue	rvalue
const A&	1 copy	1 copy
A		

Return a Modified Copy

C++03	lvalue	rvalue
const A&	1 copy	1 copy
A		

- Keep score here

Return a Modified Copy

in C++03

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

in C++03

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

in C++03

```
A
modify(A a) <-------------- here (lvalue only)
{
   a.modify();
   return a; <-------------- here
}
```

- lvalue: 2 copy constructions
- rvalue: 1 copy construction

Return a Modified Copy

C++03	lvalue	rvalue
const A&	1 copy	1 copy
A	2 copies	1 copy

Return a Modified Copy

C++03	lvalue	rvalue
const A&	1 copy	1 copy
A	2 copies	1 copy

- const A& is usually better in C++03

Return a Modified Copy

C++03	lvalue	rvalue
const A&	1 copy	1 copy
A	2 copies	1 copy

- const A& is usually better in C++03
- (perhaps except when A is small and trivial)

Return a Modified Copy

in C++11

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

Return a Modified Copy

in C++11

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

here

- lvalue: 1 copy construction
- xvalue: 1 copy construction
- prvalue: 1 copy construction

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A			

Return a Modified Copy

in C++11

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

in C++11

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

Return a Modified Copy

in C++11

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

copy lvalue here

move here

- lvalue: 1 copy construction + 1 move construction

Return a Modified Copy

in C++11

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

copy lvalue here

move xvalue here

move here

- lvalue: 1 copy construction + 1 move construction
- xvalue: 2 move constructions

Return a Modified Copy

in C++11

```
A
modify(A a)
{
   a.modify();
   return a;
}
```

copy lvalue here

move xvalue here

move here

- lvalue: 1 copy construction + 1 move construction
- xvalue: 2 move constructions
- prvalue: 1 move construction

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A	1 copy 1 move	2 moves	1 move

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A	1 copy 1 move	2 moves	1 move

- Pass by value is better in C++11 if move is much faster than copy, and if the argument is not always an lvalue.

Return a Modified Copy

in C++11

```
A
modify(const A& a)
{
   A tmp(a);
   tmp.modify();
   return tmp;
}
```

Return a Modified Copy

in C++11

```
A                                   A
modify(const A& a)                  modify(A&& a)
{                                   {
   A tmp(a);                           a.modify();
   tmp.modify();                       return std::move(tmp);
   return tmp;                      }
}
```

- Consider overloading on rvalue reference.

Return a Modified Copy

in C++11

```
A                                   A
modify(const A& a)                  modify(A&& a)
{                                   {
   A tmp(a);                           a.modify();
   tmp.modify();                       return std::move(tmp);
   return tmp;                      }
}
```

Return a Modified Copy

in C++11

```
A
modify(const A& a)
{
  A tmp(a);
  tmp.modify();
  return tmp;
}
```

```
A
modify(A&& a)
{
  a.modify();
  return std::move(tmp);
}
```

- lvalue: 1 copy construction
- xvalue: 1 move construction
- prvalue: 1 move construction

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A	1 copy 1 move	2 moves	1 move
const A& + A&&	1 copy	1 move	1 move

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A	1 copy 1 move	2 moves	1 move
const A& + A&&	1 copy	1 move	1 move

- If you pass by reference (and have fast moves), overload on rvalue reference.

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A	1 copy 1 move	2 moves	1 move
const A& + A&&	1 copy	1 move	1 move

- If you pass by reference (and have fast moves), overload on rvalue reference.
- This ideal may or may not be worth overloading.

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A	1 copy 1 move	2 moves	1 move
const A& + A&&	1 copy	1 move	1 move

- Take away: If you hear: "always pass by value" or "never pass by value", you're getting bad information.

Return a Modified Copy

C++11	lvalue	xvalue	prvalue
const A&	1 copy	1 copy	1 copy
A	1 copy 1 move	2 moves	1 move
const A& + A&&	1 copy	1 move	1 move

- Take away: If you hear: "always pass by value" or "never pass by value", you're getting bad information.
- You are not excused from the design process.

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Reference Collapsing

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

Reference Collapsing

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

Consider: `f<int&>(i);` T is int&

Reference Collapsing

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

Consider: `f<int&>(i);` T is int&

This calls: `f<int&>(int& & t);`

Reference Collapsing

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

Consider: `f<int&>(i);` T is int&

This calls: `f<int&>(int& & t);`

Which collapses to: `f<int&>(int& t);`

Reference Collapsing

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

Consider: `f<int&&>(i);` T is int&&

This calls:

Which collapses to:

Reference Collapsing

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

Consider: `f<int&&>(i);` T is int&&

This calls: `f<int&&>(int&& & t);`

Which collapses to:

Reference Collapsing

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

Consider: `f<int&&>(i);` T is int&&

This calls: `f<int&&>(int&& & t);`

Which collapses to: `f<int&&>(int& t);`

Reference Collapsing

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

Consider: `f<int&>(i);` T is int&

This calls:

Which collapses to:

Reference Collapsing

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

Consider: `f<int&>(i);` T is int&

This calls: `f<int&>(int& && t);`

Which collapses to:

Reference Collapsing

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

Consider: `f<int&>(i);` T is int&

This calls: `f<int&>(int& && t);`

Which collapses to: `f<int&>(int& t);`

Reference Collapsing

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

Consider: `f<int&&>(2);` T is int&&

This calls:

Which collapses to:

Reference Collapsing

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

Consider: `f<int&&>(2);` T is int&&

This calls: `f<int&&>(int&& && t);`

Which collapses to:

Reference Collapsing

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

Consider: `f<int&&>(2);` T is int&&

This calls: `f<int&&>(int&& && t);`

Which collapses to: `f<int&&>(int&& t);`

Reference Collapsing

This

Collapses to this

Reference Collapsing

This	Collapses to this
T& &	T&
T& &&	T&
T&& &	T&
T&& &&	T&&

The Set Up

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

- What is the declared type of t?

The Set Up

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

- What is the declared type of t?
  - A reference.

The Set Up

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

- What is the declared type of t?
  - An lvalue reference.

The Set Up

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

- What is the declared type of t?
  - A non-const lvalue reference.

The Set Up

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

- What is the declared type of t?
  - A non-const lvalue reference.

```
void g(const int& i)
{
    f(i); // f<const int>(const int& t);
}
```

The Set Up

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

- What is the declared type of t?
  - A lvalue reference.

Just because it looks like T&, you can not assume it is a non-const lvalue reference.

Special Deduction for rvalue reference

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

Special Deduction for rvalue reference

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

Just because it looks like T&&, you can not assume it is an rvalue reference.

Special Deduction for rvalue reference

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

t is an rvalue reference to int

```
f(3);  // f<int>(int&& t);
```

Just because it looks like T&&, you can not assume it is an rvalue reference.

Special Deduction for rvalue reference

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

t is an rvalue reference to int

```
f(3);   // f<int>(int&& t);

f(i);   // f<int&>(int& && t);
```

Just because it looks like T&&, you can not assume it is an rvalue reference.

Special Deduction for rvalue reference

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

```
f(3);   // f<int>(int&& t);
```

t is an rvalue reference to int

```
f(i);   // f<int&>(int& t);
```

t is an lvalue reference to int

Just because it looks like T&&, you can not assume it is an rvalue reference.

Special Deduction for rvalue reference

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

Special Deduction for rvalue reference

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

- The “T&&” template pattern is special.

Special Deduction for rvalue reference

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

- The “T&&” template pattern is special.
- “const T&&” does not behave this way.

Special Deduction for rvalue reference

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

- The “T&&” template pattern is special.
- “const T&&” does not behave this way.
- If the argument is an lvalue A, T deduces as A&, otherwise T deduces as A.

Special Deduction for rvalue reference

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

- The “T&&” template pattern is special.
- “const T&&” does not behave this way.
- If the argument is an lvalue A, T deduces as A&, otherwise T deduces as A.
- cv-qualifiers will also deduce into T just as in the T& case.

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

Outline

- The rvalue reference
- Move Semantics
- Factory Functions
- More rvalue ref rules
- “Perfect” forwarding

The Forwarding Problem

The Forwarding Problem

- A forwarding function forwards one or more arguments to some other function.

The Forwarding Problem

- A forwarding function forwards one or more arguments to some other function.
- A forwarding function should preserve cv-qualifiers and value category.

The Forwarding Problem

- A forwarding function forwards one or more arguments to some other function.
- A forwarding function should preserve cv-qualifiers and value category.
- If the destination function is overloaded on cv-qualifiers or value category, getting the forwarding wrong can be catastrophic.

The Forwarding Problem

```
template <class T, class U>
void f(T& t, U& u);
```

The Forwarding Problem

```
template <class T, class U>
void f(T& t, U& u);

template <class T, class U>
void g(T& t, U& u)
{f(t,u);}
```

```
int i = 2;
const int j = 3;
g(i, j);
```

Good!

f sees: t is an lvalue int,
u is an lvalue const int

The Forwarding Problem

```
template <class T, class U>
void f(T& t, U& u);

template <class T, class U>
void g(T& t, U& u)
{f(t,u);}
```

```
int i = 2;
const int j = 3;
g(i, j);
```

Good!

f sees: t is an lvalue int,
u is an lvalue const int

But: `g(i, 3);  // Doesn't compile!`

The Forwarding Problem

```
template <class T, class U>
void f(T& t, U& u);
```

```
int i = 2;
const int j = 3;
g(i, j);
```

f sees:

```
g(i, 3);
```

The Forwarding Problem

```
template <class T, class U>
void f(T& t, U& u);

template <class T, class U>
void g(const T& t, const U& u)
{f(t,u);}
```

```
int i = 2;
const int j = 3;
g(i, j);
```

But const unnecessarily added.

f sees:
```
t is an lvalue const int,
u is an lvalue const int
```

Now:
```
g(i, 3);  // Compiles!
```

The Forwarding Problem

The Forwarding Problem

```
template <class T, class U>
void g(T& t, U& u)
{f(t,u);}

template <class T, class U>
void g(const T& t, U& u)
{f(t,u);}

template <class T, class U>
void g(T& t, const U& u)
{f(t,u);}

template <class T, class U>
void g(const T& t, const U& u)
{f(t,u);}
```

The Forwarding Problem

Too Many Overloads!!!

The Forwarding Solution

The Forwarding Solution

```
template <class T, class U, class V>
void f(T&& t, U&& u, V&& v);

template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
    f(t, u, v);
}
```

```
const A i = A();
A j;
g(i, j, A());
```

v should be an rvalue

f sees:

```
t is an lvalue const A
u is an lvalue A
v is an lvalue A
```

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         u,
         v);
}
```

```
const A i = A();
A j;
g(i, j, A());
```

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         u,
         v);
}
```

```
const A i = A();
A j;
g(i, j, A());
```

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         u,
         v);
}
```

```
const A i = A();T is const A&
A j;
g(i, j, A());
```

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
        f(static_cast<T&&>(t),
          u,
          v);
    }
```

```
const A i = A();
A j;
g(i, j, A());
```

```
T is const A&
```

```
T is const A&
```

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         u,
         v);
}
```

```
const A i = A();
A j;
g(i, j, A());
```

T is const A&

T&& is const A&

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         u,
         v);
}
```

```
const A i = A();      T is const A&
A j;                  T&& is const A&
g(i, j, A());
```

f sees: t is an lvalue const A ✓

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         v);
}
```

```
const A i = A();
A j;
g(i, j, A());
```

f sees: t is an lvalue const A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         v);
}
```

```
const A i = A();
A j;
g(i, j, A());
```

f sees: t is an lvalue const A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         v);
}
```

```
const A i = A();
A j;                          U is A&
g(i, j, A());
```

f sees: t is an lvalue const A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         v);
}
```

```
const A i = A();
A j;                    U is A&
g(i, j, A());
```

f sees: t is an lvalue const A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         v);
}
```

```
const A i = A();
A j;                        U is A&
g(i, j, A());               U&& is A&
```

f sees: t is an lvalue const A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         v);
}
```

```
const A i = A();
A j;                        U is A&
g(i, j, A());               U&& is A&
```

f sees: t is an lvalue const A
u is an lvalue A ✓

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         static_cast<V&&>(v));
}
```

```
const A i = A();
A j;
g(i, j, A());
```

f sees: t is an lvalue const A
u is an lvalue A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         static_cast<V&&>(v));
}
```

```
const A i = A();
A j;
g(i, j, A());
```

f sees: t is an lvalue const A
u is an lvalue A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         static_cast<V&&>(v));
}
```

```
const A i = A();
A j;
g(i, j, A());          V is A
```

f sees: t is an lvalue const A
u is an lvalue A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         static_cast<V&&>(v));
}
```

```
const A i = A();
A j;
g(i, j, A());
```

V is A

f sees: t is an lvalue const A
u is an lvalue A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         static_cast<V&&>(v));
}
```

```
const A i = A();
A j;
g(i, j, A());
```

V is A
V&& is A&&

f sees: t is an lvalue const A
u is an lvalue A

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         static_cast<V&&>(v));
}
```

```
const A i = A();
A j;
g(i, j, A());
```

V is A
V&& is A&&

f sees: t is an lvalue const A
u is an lvalue A
v is an rvalue A

✓

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
       f(static_cast<T&&>(t),
         static_cast<U&&>(u),
         static_cast<V&&>(v));
}
```

```
const A i = A();
A j;
g(i, j, A());
```

Perfect!

f sees:
```
t is an lvalue const A
u is an lvalue A
v is an rvalue A
```

The Forwarding Solution

```
const A i = A();
A j;
g(i, j, A());
```

Perfect!

```
f sees:  t is an lvalue const A
         u is an lvalue A
         v is an rvalue A
```

The Forwarding Solution

```
template <class T, class U, class V>
void g(T&& t, U&& u, V&& v) {
    f(std::forward<T>(t),
      std::forward<U>(u),
      std::forward<V>(v));
}
```

Use `std::forward` for better readability.

```
const A i = A();
A j;
g(i, j, A());
```

Perfect!

f sees:
- `t` is an lvalue `const A`
- `u` is an lvalue `A`
- `v` is an rvalue `A`

The Forwarding Solution

```
const A i = A();
A j;
g(i, j, A());
```

Perfect!

```
f sees:  t is an lvalue const A
         u is an lvalue A
         v is an rvalue A
```

The Forwarding Solution

```
template <class ...T>
void g(T&& ...t) {
    f(std::forward<T>(t)...);
}
```

```
const A i = A();
A j;
g(i, j, A());
```

f sees:
```
t is an lvalue const A
u is an lvalue A
v is an rvalue A
```

Perfect!
And easy!

Summary

- The rvalue reference has been introduced to support:
- Move semantics.
- Perfect forwarding.