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

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

- Consider overloading on rvalue reference.

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);
}
```

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);
```

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

t is an rvalue 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);
```

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

t is an rvalue reference to int

```
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();
A j;
g(i, j, 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);
}
```

is const A&

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

```
T is const A&
T&& is const 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);
}
```

U&& is A&

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

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());
```

U is 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));
}
```

is A&&

```
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());
```

Perfect!
And easy!

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

Summary

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