Back To Basics
Forwarding References

MATEUSZ PUSZ

Cppcon
The C++ Conference

2023
October 01 - 06

Workshopy Style

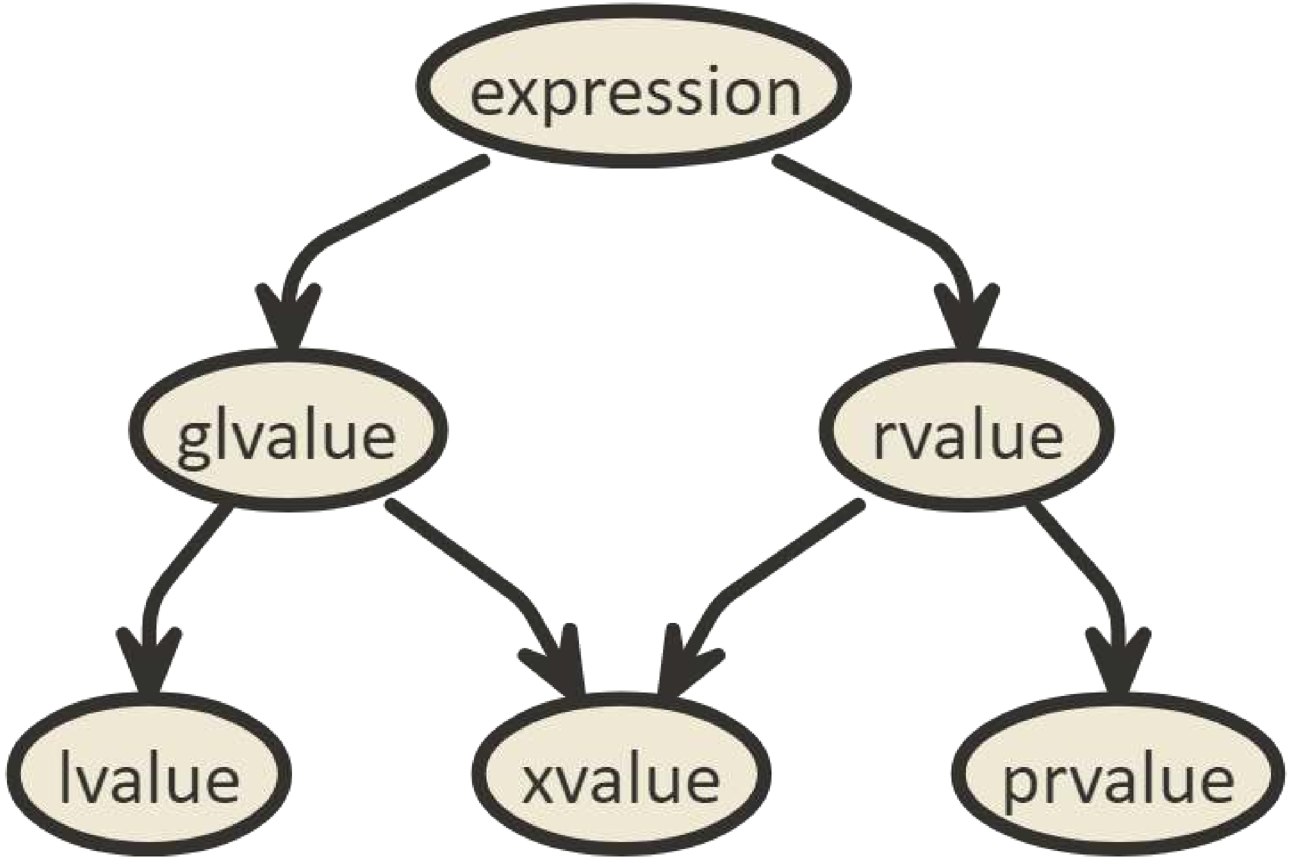
Workshopy Style

- Provide rationale
- Facilitate discussion
  - force the audience to think
  - not just a lecture
- Describe
  - pitfalls
  - corner cases
- Provide recommendations
- ~~Lot's of coding~~

https://ahaslides.com/FWDREF

Thank You Scott Meyers!

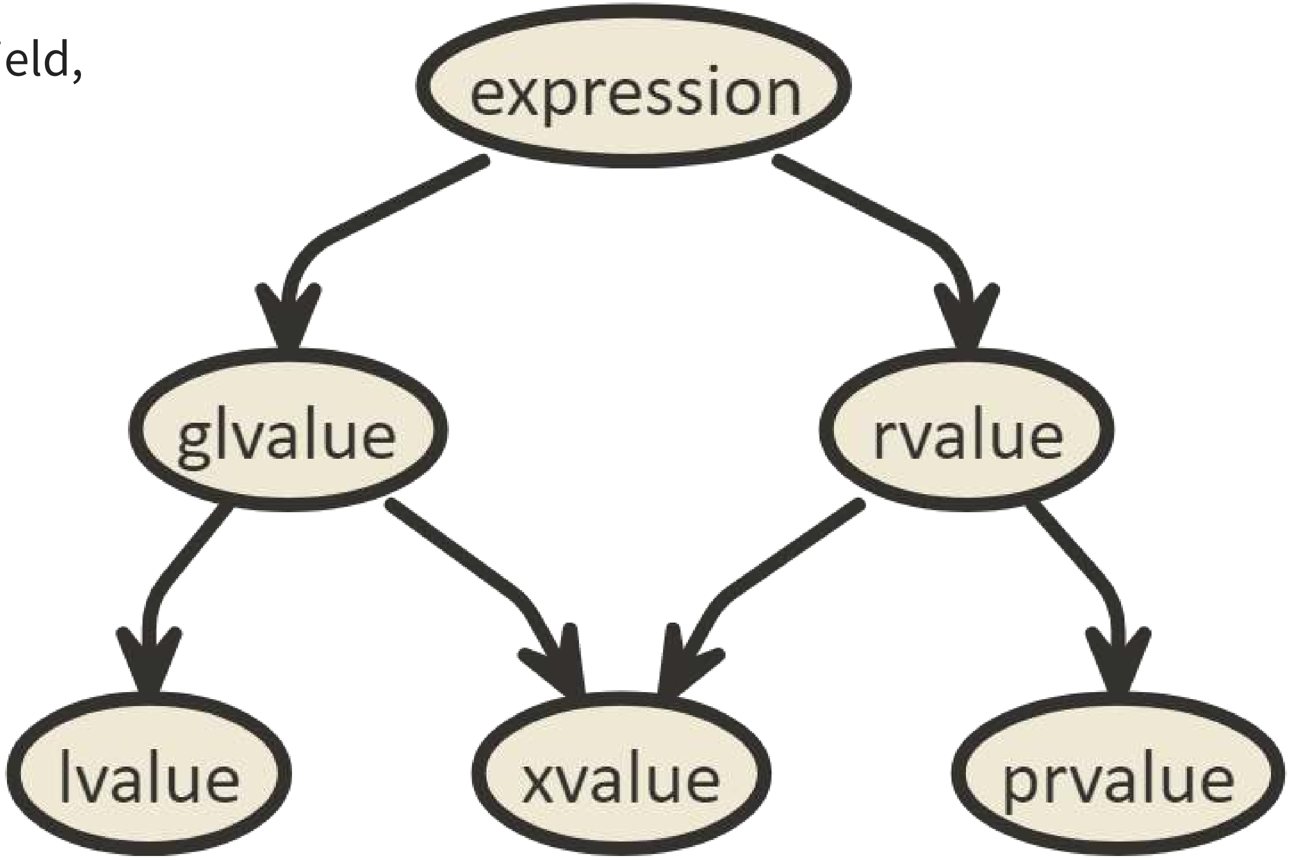
Value categories



Value categories

“GENERALIZED” LVALUES

Determines *the identity* of an object, bit-field, or function



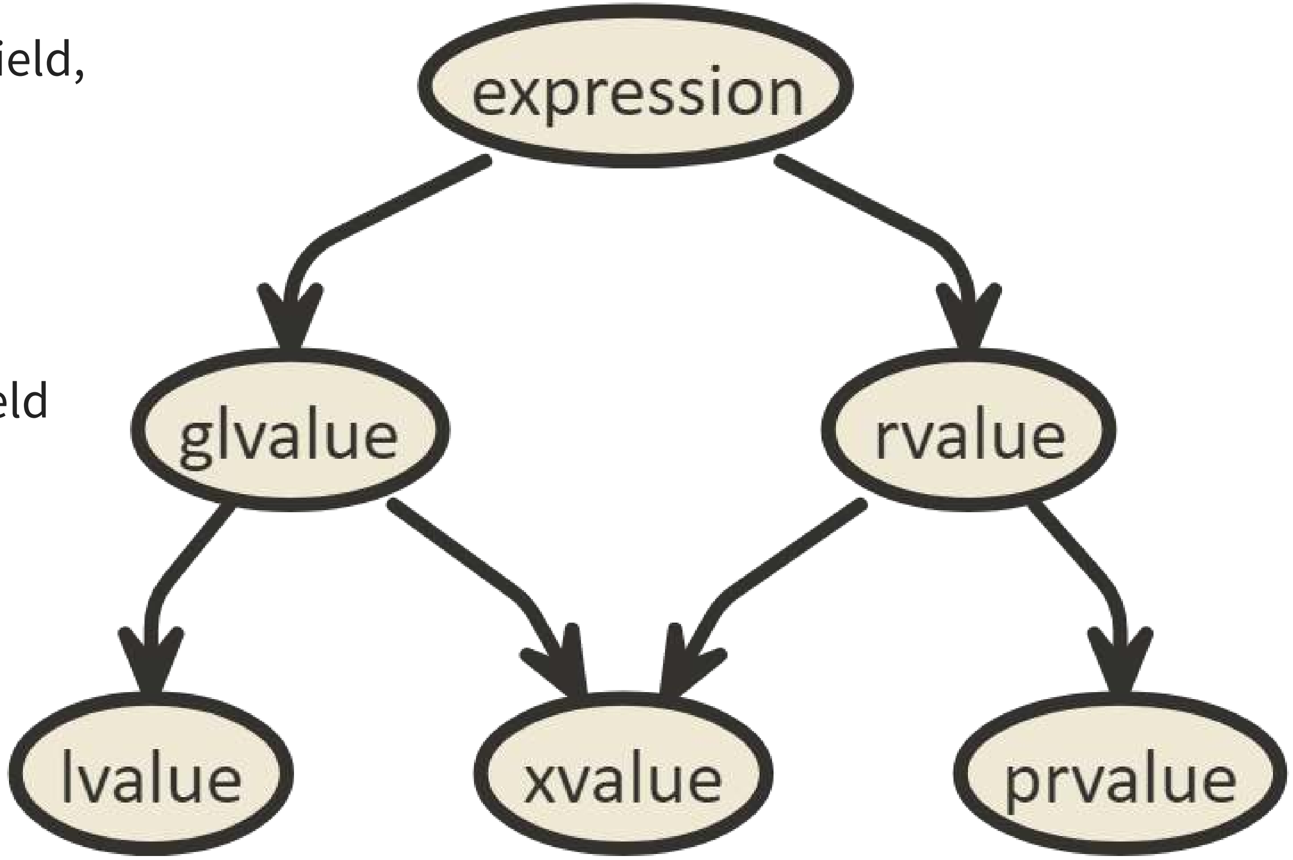
Value categories

“GENERALIZED” LVALUES

Determines *the identity* of an object, bit-field, or function

"PURE" RVALUES

Computes the value of the operand of an operator or *initializes* an object or a bit-field



Value categories

“GENERALIZED” LVALUES

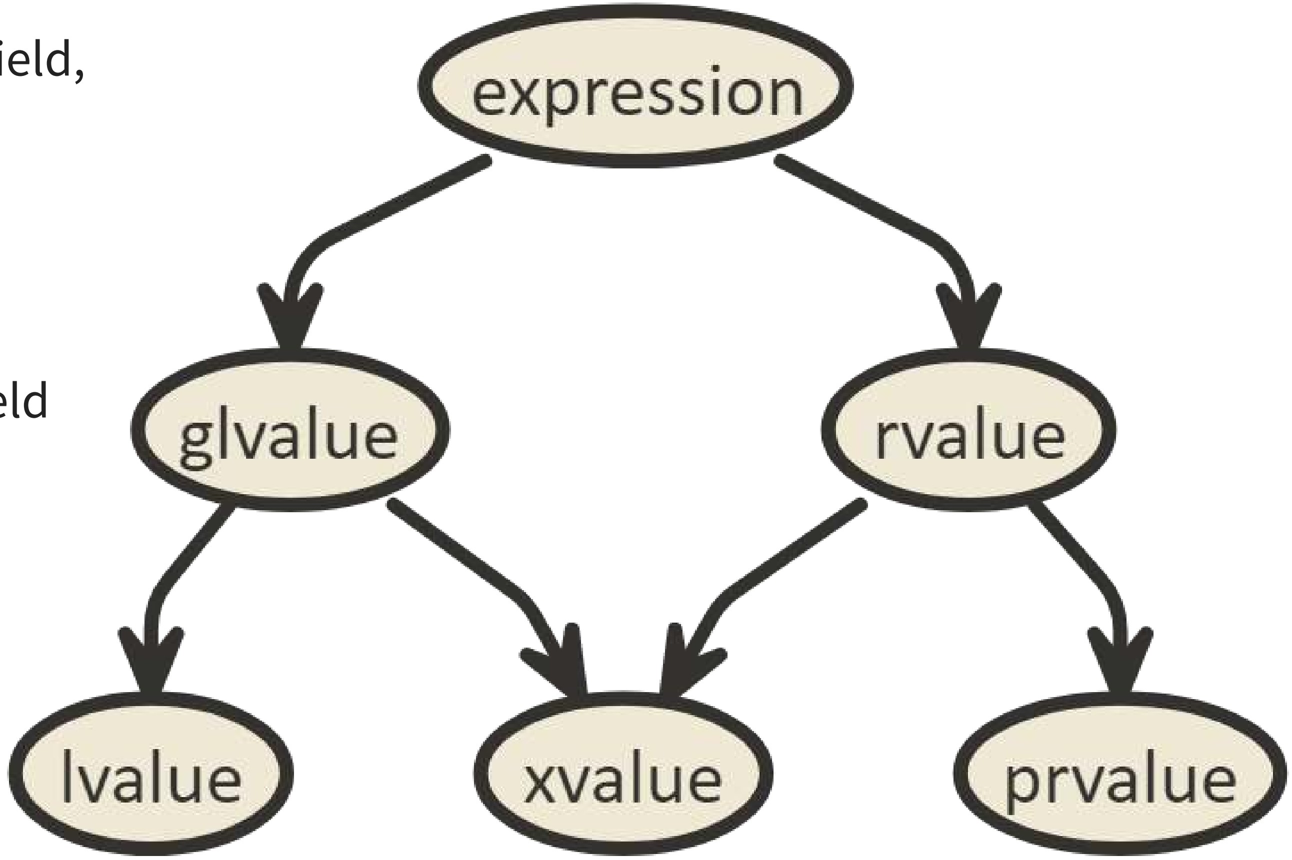
Determines *the identity* of an object, bit-field, or function

"PURE" RVALUES

Computes the value of the operand of an operator or *initializes* an object or a bit-field

"EXPIRING" LVALUES

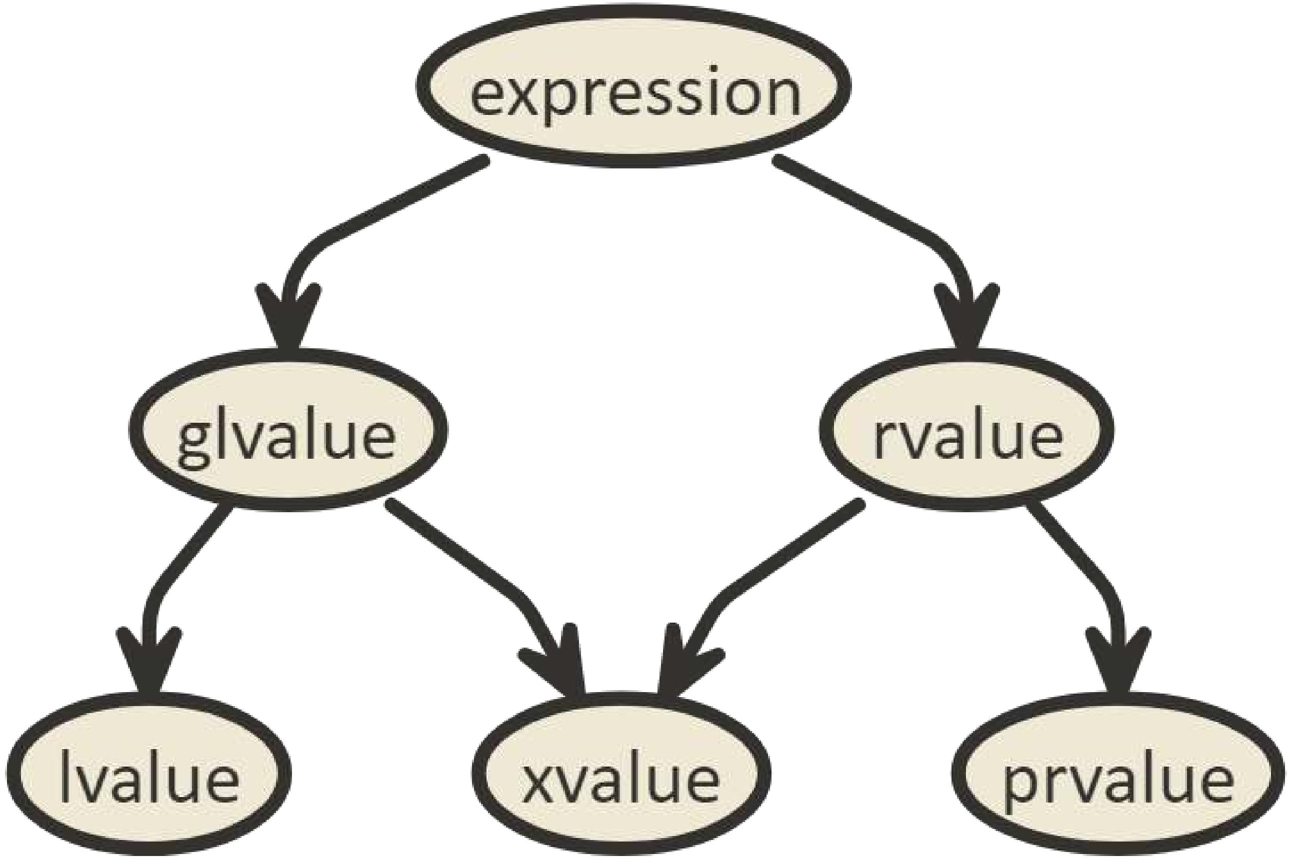
glvalue that denotes an object or bit-field whose *resources can be reused*



Value categories

LVALUE

glvalue that is not an *xvalue*



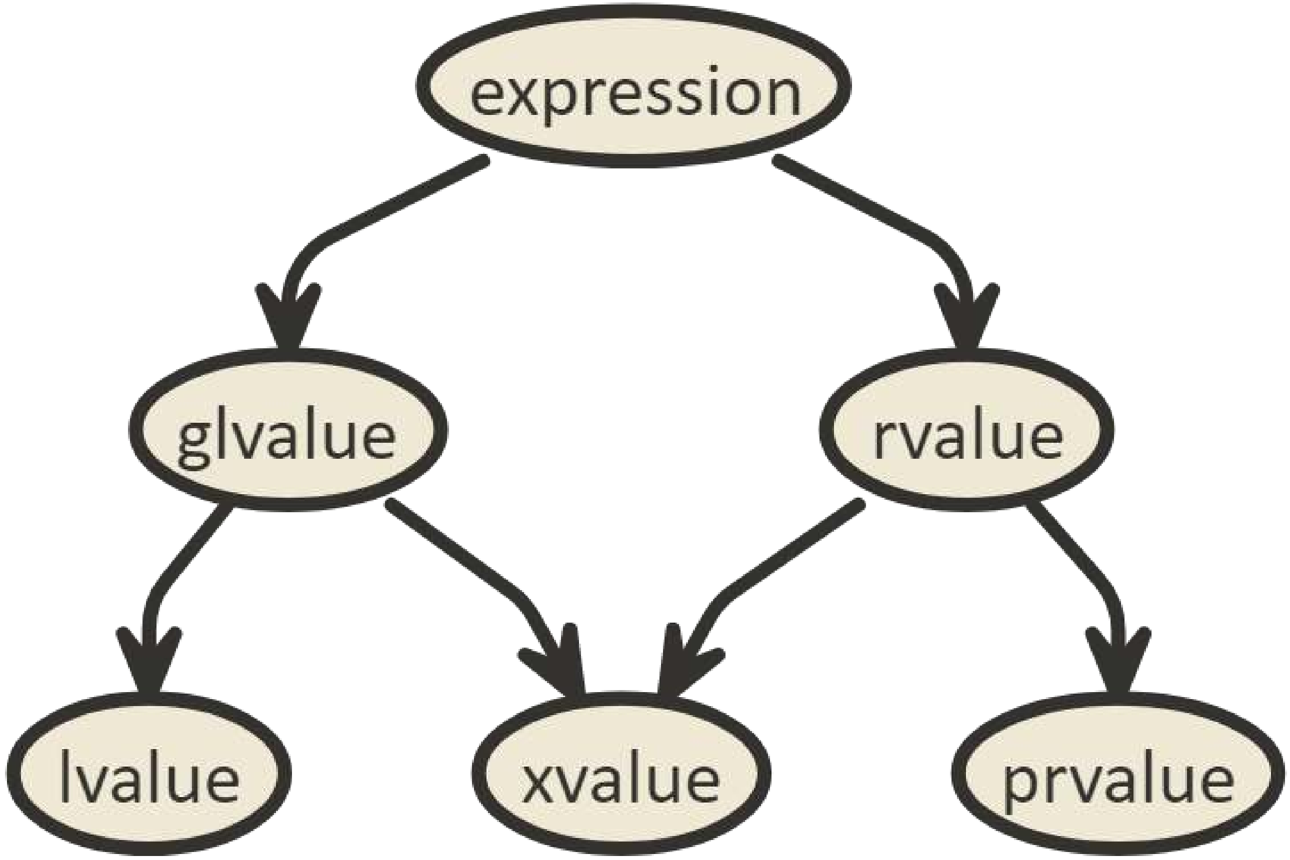
Value categories

LVALUE

glvalue that is not an *xvalue*

RVALUE

prvalue or an *xvalue*



Quiz

https://ahaslides.com/FWDREF

Quiz: Pointers

```cpp
void f(int* ptr);
void f(const int* ptr);
```

```cpp
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```cpp
f(i);                  // ???
f(ci);                 // ???
f(std::move(i));       // ???
f(std::move(ci));      // ???
f(42);                 // ???
f(make_int());         // ???
```

Pointers

```
void f(int* ptr);
void f(const int* ptr);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);              // ERROR
f(ci);             // ERROR
f(std::move(i));   // ERROR
f(std::move(ci));  // ERROR
f(42);             // ERROR
f(make_int());     // ERROR
```

In order to pass a variable to a function taking a pointer we need to pass its address.

Quiz: Pointers

```
void f(int* ptr);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(&i);                 // ???
f(&ci)                 // ???
f(&std::move(i));      // ???
f(&std::move(ci));     // ???
f(&42);                // ???
f(&make_int());        // ???
```

Pointers

```
void f(int* ptr);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(&i);                 // OK
f(&ci)                 // ERROR
f(&std::move(i));      // ERROR
f(&std::move(ci));     // ERROR
f(&42);                // ERROR
f(&make_int());        // ERROR
```

Quiz: Pointers

```
void f(const int* ptr);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(&i);                  // ???
f(&ci)                  // ???
f(&std::move(i));       // ???
f(&std::move(ci));      // ???
f(&42);                 // ???
f(&make_int());         // ???
```

Pointers

```
void f(const int* ptr);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(&i);                // OK
f(&ci)                // OK
f(&std::move(i));     // ERROR
f(&std::move(ci));    // ERROR
f(&42);               // ERROR
f(&make_int());       // ERROR
```

Quiz: References

```
void f(int& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);               // ???
f(ci);              // ???
f(std::move(i));    // ???
f(std::move(ci));   // ???
f(42);              // ???
f(make_int());      // ???
```

References

```
void f(int& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);                 // OK
f(ci);                // ERROR
f(std::move(i));      // ERROR
f(std::move(ci));     // ERROR
f(42);                // ERROR
f(make_int());        // ERROR
```

Quiz: References

```
void f(const int& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);                  // ???
f(ci);                 // ???
f(std::move(i));       // ???
f(std::move(ci));      // ???
f(42);                 // ???
f(make_int());         // ???
```

References

```
void f(const int& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);                  // OK
f(ci);                 // OK
f(std::move(i));       // OK
f(std::move(ci));      // OK
f(42);                 // OK
f(make_int());         // OK
```

Pointers vs references

POINTER	REFERENCE
Objects	Alias (not an object)
Always occupy memory	May not occupy storage
Arrays of pointers are legal	No arrays of references
Pointers to pointers legal	References or pointers to references not allowed
Pointers to `void` legal	No references to `void`
May be uninitialized	Must be initialized
Can be reassigned after initialization	Immutable
Can be cv-qualified	Can't be cv-qualified

Pointers vs references

> Use references when you can, and pointers when you have to.
>
> *-- C++ FAQ*

Quiz: rvalue references

```
void f(int&& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);                // ???
f(ci);               // ???
f(std::move(i));     // ???
f(std::move(ci));    // ???
f(42);               // ???
f(make_int());       // ???
```

rvalue references

```
void f(int&& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);                 // ERROR
f(ci);                // ERROR
f(std::move(i));      // OK
f(std::move(ci));     // ERROR
f(42);                // OK
f(make_int());        // OK
```

Quiz: rvalue references

```
void f(const int&& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);                 // ???
f(ci);                // ???
f(std::move(i));      // ???
f(std::move(ci));     // ???
f(42);                // ???
f(make_int());        // ???
```

rvalue references

```
void f(const int&& ref);
```

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);                 // ERROR
f(ci);                // ERROR
f(std::move(i));      // OK
f(std::move(ci));     // OK
f(42);                // OK
f(make_int());        // OK
```

Passing temporary arguments

```
void foo(const int& x)
{
  int temp = x;
  x = 0;   // ERROR
}

foo(make_int());
```

- lvalue references to **const** CAN'T be mutated
- **True even if we pass a temporary (rvalue)**

Passing temporary arguments

```
void foo(const int& x)
{
  int temp = x;
  x = 0;   // ERROR
}

foo(make_int());
```

- lvalue references to **const** CAN'T be mutated
- True even if we pass a temporary (rvalue)

```
void foo(int&& x)
{
  int temp = x;
  x = 0;   // OK
}

foo(make_int());
```

- rvalue references CAN be mutated
- Enable Move Semantics

Passing temporary arguments

```
void foo(const int& x)
{
  int temp = x;
  x = 0;   // ERROR
}

foo(make_int());
```

- lvalue references to **const** CAN'T be mutated
- True even if we pass a temporary (rvalue)

```
void foo(int&& x)
{
  int temp = x;
  x = 0;   // OK
}

foo(make_int());
```

- rvalue references CAN be mutated
- Enable Move Semantics

rvalue references to **const** can't be mutated. They are similar to lvalue references to **const** except they bind only to rvalues.

Objects of rvalue reference type

rvalue reference variables are lvalues when used in expressions.

Objects of rvalue reference type

rvalue reference variables are lvalues when used in expressions.

```
void boo(int&& x);

void foo(int&& x)
{
  boo(x); // ERROR
}
```

Objects of rvalue reference type

rvalue reference variables are lvalues when used in expressions.

```
void boo(int&& x);

void foo(int&& x)
{
  boo(x); // ERROR
}
```

```
void boo(int&& x);

void foo(int&& x)
{
  boo(std::move(x)); // OK
}
```

`std::move()`

`std::move` does not move! It is just a cast to an rvalue reference type.

`std::move()`

`std::move` does not move! It is just a cast to an rvalue reference type.

All move semantics related logic is provided by the function that is being called with the argument of an rvalue reference type.

Forwarding reference

- Deduced function template parameter declared as rvalue reference to cv-unqualified type template parameter

```
template<typename T> void f(T&& x);
```

```
void f(auto&& x);
```

Forwarding reference

- **Deduced function template parameter** declared as **rvalue reference** to **cv-unqualified type template parameter**

```
template<typename T> void f(T&& x);
```

```
void f(auto&& x);
```

- Special kind of reference that **binds to any function argument**

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
f(i);               // OK
f(ci);              // OK
f(std::move(i));    // OK
f(std::move(ci));   // OK
f(42);              // OK
f(make_int());      // OK
```

NOT A Forwarding reference

```
template<class T>
int g(const T&& y);
```

NOT A Forwarding reference

```
template<class T>
int g(const T&& y);
```

```
template<class T>
struct A {
  template<class U, class Z = T>
  A(T&& t, U&& u, Z&& z); // only 'u' and 'z' are forwarding references
};
```

NOT A Forwarding reference

```
template<class T>
int g(const T&& y);
```

```
template<class T>
struct A {
  template<class U, class Z = T>
  A(T&& t, U&& u, Z&& z); // only 'u' and 'z' are forwarding references
};
```

```
template<typename T>
void foo(std::vector<T>&& v);
```

NOT A Forwarding reference

```
template<class T>
int g(const T&& y);
```

```
template<class T>
struct A {
  template<class U, class Z = T>
  A(T&& t, U&& u, Z&& z); // only 'u' and 'z' are forwarding references
};
```

```
template<typename T>
void foo(std::vector<T>&& v);
```

```
auto&& z = {1, 2, 3}; // special case for initializer lists
```

A Universal Reference?

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
template<typename T>
void f(T&& x);
```

```
f(i);               // f<int&>(int&)
f(ci);              // f<const int&>(const int&)
f(std::move(i));    // f<int>(int&&)
f(std::move(ci));   // f<const int>(const int&&)
f(42);              // f<int>(int&&)
f(make_int());      // f<int>(int&&)
```

A Universal Reference?

```
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```
template<typename T>
void f(T&& x);
```

```
f(i);              // f<int&>(int&)
f(ci);             // f<const int&>(const int&)
f(std::move(i));   // f<int>(int&&)
f(std::move(ci));  // f<const int>(const int&&)
f(42);             // f<int>(int&&)
f(make_int());     // f<int>(int&&)
```

Special rule: When **T&&** is used as a function template parameter and an lvalue is being passed the deduced type is **T&** instead of **T**.

A Universal Reference?

```cpp
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```cpp
template<typename T>
void f(T&& x);
```

```cpp
f(i);                // f<int&>(int&)
f(ci);               // f<const int&>(const int&)
f(std::move(i));     // f<int>(int&&)
f(std::move(ci));    // f<const int>(const int&&)
f(42);               // f<int>(int&&)
f(make_int());       // f<int>(int&&)
```

```cpp
template<typename T>
void f(const T& x);
```

```cpp
f(i);                // f<int>(const int&)
f(ci);               // f<int>(const int&)
f(std::move(i));     // f<int>(const int&)
f(std::move(ci));    // f<int>(const int&)
f(42);               // f<int>(const int&)
f(make_int());       // f<int>(const int&)
```

A Universal Reference?

```cpp
int i = 42;
const int ci = 42;
int make_int() { return 42; }
```

```cpp
template<typename T>
void f(T&& x);
```

```cpp
f(i);              // f<int&>(int&)
f(ci);             // f<const int&>(const int&)
f(std::move(i));   // f<int>(int&&)
f(std::move(ci));  // f<const int>(const int&&)
f(42);             // f<int>(int&&)
f(make_int());     // f<int>(int&&)
```

```cpp
template<typename T>
void f(const T& x);
```

```cpp
f(i);              // f<int>(const int&)
f(ci);             // f<int>(const int&)
f(std::move(i));   // f<int>(const int&)
f(std::move(ci));  // f<int>(const int&)
f(42);             // f<int>(const int&)
f(make_int());     // f<int>(const int&)
```

Both **forwarding references** and **const lvalue references** bind to everything.
However, the former might be more expensive.

Argument binding summary

ARGUMENT BINDS TO	`int*`	`const int*`	`int&`	`const int&`	`int&&`	`const int&&`	`T&&`
lvalue	Yes	Yes	Yes	Yes			Yes
lvalue to **`const`**		Yes		Yes			Yes
xvalue				Yes	Yes	Yes	Yes
xvalue to **`const`**				Yes		Yes	Yes
prvalue				Yes	Yes	Yes	Yes

Recommendation

If you need to implement a function that binds to everything and "just" needs to read data, you probably should choose <u>**const lvalue reference**</u> as an argument.

Forwarding function parameters

```
void f(int&)          { std::cout << "lvalue\n"; }
void f(const int&)    { std::cout << "const lvalue\n"; }
void f(int&&)         { std::cout << "rvalue\n"; }
```

Forwarding function parameters

```
void f(int&)          { std::cout << "lvalue\n"; }
void f(const int&)    { std::cout << "const lvalue\n"; }
void f(int&&)         { std::cout << "rvalue\n"; }
```

```
void wrapper(int& v)       { do_something(); f(v); }
void wrapper(const int& v) { do_something(); f(v); }
void wrapper(int&& v)      { do_something(); f(std::move(v)); }
```

Forwarding function parameters

```
void f(int&)          { std::cout << "lvalue\n"; }
void f(const int&)    { std::cout << "const lvalue\n"; }
void f(int&&)         { std::cout << "rvalue\n"; }
```

```
void wrapper(int& v)        { do_something(); f(v); }
void wrapper(const int& v) { do_something(); f(v); }
void wrapper(int&& v)       { do_something(); f(std::move(v)); }
```

Even more expensive in generic code for multiple function parameters.

Perfect Forwarding

Forwarding references <u>**preserve the value category**</u> of a function argument, making it possible to forward it by means of **`std::forward`**.

Perfect Forwarding

Forwarding references <u>**preserve the value category**</u> of a function argument, making it possible to forward it by means of **`std::forward`**.

```
void f(int&)        { std::cout << "lvalue\n"; }
void f(const int&)  { std::cout << "const lvalue\n"; }
void f(int&&)       { std::cout << "rvalue\n"; }
```

Perfect Forwarding

Forwarding references **preserve the value category** of a function argument, making it possible to forward it by means of **`std::forward`**.

```
void f(int&)         { std::cout << "lvalue\n"; }
void f(const int&)   { std::cout << "const lvalue\n"; }
void f(int&&)        { std::cout << "rvalue\n"; }
```

```
template<typename T> void wrapper(T&& v) { do_something(); f(std::forward<T>(v)); }
```

Perfect Forwarding

Forwarding references **preserve the value category** of a function argument, making it possible to forward it by means of **`std::forward`**.

```
void f(int&)          { std::cout << "lvalue\n"; }
void f(const int&)    { std::cout << "const lvalue\n"; }
void f(int&&)         { std::cout << "rvalue\n"; }
```

```
template<typename T> void wrapper(T&& v) { do_something(); f(std::forward<T>(v)); }
```

`std::forward<T>(t)` converts **`t`** to rvalue reference only if rvalue.

std::forward Implementation

```
template<typename T>
[[nodiscard]] constexpr T&& forward(typename std::type_identity_t<T>& param)
{
  return static_cast<std::type_identity_t<T>&&>(param);
}
```

std::forward Implementation

```
template<typename T>
[[nodiscard]] constexpr T&& forward(typename std::type_identity_t<T>& param)
{
  return static_cast<std::type_identity_t<T>&&>(param);
}
```

REFERENCE COLLAPSING

It is permitted to form references to references through type manipulations in templates or typedefs

- *rvalue reference to rvalue reference* collapses to *rvalue reference*
- *all other* combinations form *lvalue reference*

```
using lref = int&;
using rref = int&&;
int n;
```

```
lref&  r1 = n;    // type of r1 is int&
lref&& r2 = n;    // type of r2 is int&
rref&  r3 = n;    // type of r3 is int&
rref&& r4 = 1;    // type of r4 is int&&
```

Recommendation

Prefer forwarding references when you need to forward function parameters to other functions **while preserving their value category**.

Sinks: The most efficient overload set

```
class MyClass {
  std::string txt_;
public:
  explicit MyClass(const std::string& txt):
    txt_(txt)
  {}
  explicit MyClass(std::string&& txt):
    txt_(std::move(txt))
  {}
};
```

Sinks: Scaling problems

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  MyClass(const std::string& txt1, const std::string& txt2):
    txt1_(txt1), txt1_(txt1)
  {}
  MyClass(std::string&& txt1, std::string&& txt2):
    txt1_(std::move(txt1)), txt2_(std::move(txt2))
  {}
  MyClass(const std::string& txt1, std::string&& txt2):
    txt1_(txt1), txt1_(std::move(txt1))
  {}
  MyClass(std::string&& txt1, const std::string& txt2):
    txt1_(std::move(txt1)), txt2_(txt2)
  {}
};
```

Sinks: Good compromise

```cpp
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  MyClass(std::string txt1, std::string txt2):
    txt1_(std::move(txt1)), txt2_(std::move(txt2))
  {}
};
```

Sinks: Good compromise

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  MyClass(std::string txt1, std::string txt2):
    txt1_(std::move(txt1)), txt2_(std::move(txt2))
  {}
};
```

- Adds one *move construction* and *destruction of a moved from object* operations for **lvalues** and **xvalues**

Sinks: Good compromise

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  MyClass(std::string txt1, std::string txt2):
    txt1_(std::move(txt1)), txt2_(std::move(txt2))
  {}
};
```

- Adds one *move construction* and *destruction of a moved from object* operations for **lvalues** and **xvalues**

Scales well and still quite fast.

Sinks: Perfect Forwarding to the rescue

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<typename T, typename U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

Sinks: Perfect Forwarding to the rescue

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<typename T, typename U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```
MyClass c1(42, 3.14);
```

```
In instantiation of 'MyClass::MyClass(T&&, U&&) [with T = int; U = double]':
error: no matching function for call to 'std::__cxx11::basic_string<char>::basic_string(int)'
    9 |     txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Sinks: Perfect Forwarding to the rescue

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<typename T, typename U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```
MyClass c1(42, 3.14);
```

```
In instantiation of 'MyClass::MyClass(T&&, U&&) [with T = int; U = double]':
error: no matching function for call to 'std::__cxx11::basic_string<char>::basic_string(int)'
    9 |     txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Takes everything as arguments 😢

Sinks: Constraining a forwarding reference (naive approach)

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<std::same_as<std::string> T, std::same_as<std::string> U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

Sinks: Constraining a forwarding reference (naive approach)

```cpp
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<std::same_as<std::string> T, std::same_as<std::string> U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```cpp
MyClass c1(42, 3.14);                 // ERROR
```

Sinks: Constraining a forwarding reference (naive approach)

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<std::same_as<std::string> T, std::same_as<std::string> U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```
MyClass c1(42, 3.14);              // ERROR
```

```
MyClass c2("abc"s, "def"s);        // OK
```

Sinks: Constraining a forwarding reference (naive approach)

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<std::same_as<std::string> T, std::same_as<std::string> U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```
MyClass c1(42, 3.14);           // ERROR
```

```
MyClass c2("abc"s, "def"s);     // OK
```

```
std::string txt = "abc";
MyClass c3(txt, txt);           // ERROR
```

Sinks: Constraining a forwarding reference

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<typename T, typename U>
    requires std::same_as<std::remove_cvref_t<T>, std::string> &&
             std::same_as<std::remove_cvref_t<U>, std::string>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

Sinks: Constraining a forwarding reference

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<typename T, typename U>
    requires std::same_as<std::remove_cvref_t<T>, std::string> &&
             std::same_as<std::remove_cvref_t<U>, std::string>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```
MyClass c1(42, 3.14);           // ERROR
MyClass c2("abc"s, "def"s);     // OK
std::string txt = "abc";
MyClass c3(txt, txt);           // OK
```

Sinks: Constraining a forwarding reference

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<typename T, typename U>
    requires std::same_as<std::remove_cvref_t<T>, std::string> &&
             std::same_as<std::remove_cvref_t<U>, std::string>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```
MyClass c1(42, 3.14);           // ERROR
MyClass c2("abc"s, "def"s);     // OK
std::string txt = "abc";
MyClass c3(txt, txt);           // OK
```

```
MyClass c4("abc", "def");       // ERROR
```

Sinks: Constraining a forwarding reference

```cpp
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<std::convertible_to<std::string> T, std::convertible_to<std::string> U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

Sinks: Constraining a forwarding reference

```
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<std::convertible_to<std::string> T, std::convertible_to<std::string> U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```
MyClass c1(42, 3.14);              // ERROR
MyClass c2("abc"s, "def"s);        // OK
std::string txt = "abc";
MyClass c3(txt, txt);              // OK
MyClass c4("abc", "def");          // OK
```

Sinks: Constraining a forwarding reference

```cpp
class MyClass {
  std::string txt1_;
  std::string txt2_;
public:
  template<std::convertible_to<std::string> T, std::convertible_to<std::string> U>
  MyClass(T&& txt1, U&& txt2):
    txt1_(std::forward<T>(txt1)), txt2_(std::forward<U>(txt2))
  {}
};
```

```cpp
MyClass c1(42, 3.14);              // ERROR
MyClass c2("abc"s, "def"s);        // OK
std::string txt = "abc";
MyClass c3(txt, txt);              // OK
MyClass c4("abc", "def");          // OK
```

Use **`std::constructible_from`** to allow explicit conversions as well.

Question: Tell me about the code?

```
class int_or_empty {
  int value_ = 0;
  bool empty_ = true;
public:
  int_or_empty() = default;

  template<std::convertible_to<int> T>
  constexpr int_or_empty(T&& v) :
    value_{std::forward<T>(v)}, empty_{false}
  {
  }

  constexpr bool empty() const
  { return empty_; }

  constexpr operator int() const
  { return value_; }
};
```

Question: Tell me about the code?

```
class int_or_empty {
  int value_ = 0;
  bool empty_ = true;
public:
  int_or_empty() = default;

  template<std::convertible_to<int> T>
  constexpr int_or_empty(T&& v) :
    value_{std::forward<T>(v)}, empty_{false}
  {
  }

  constexpr bool empty() const
  { return empty_; }

  constexpr operator int() const
  { return value_; }
};
```

```
int_or_empty empty;
assert(empty.empty());
```

Question: Tell me about the code?

```cpp
class int_or_empty {
  int value_ = 0;
  bool empty_ = true;
public:
  int_or_empty() = default;

  template<std::convertible_to<int> T>
  constexpr int_or_empty(T&& v) :
    value_{std::forward<T>(v)}, empty_{false}
  {
  }

  constexpr bool empty() const
  { return empty_; }

  constexpr operator int() const
  { return value_; }
};
```

```cpp
int_or_empty empty;
assert(empty.empty());
```

```cpp
int_or_empty one{1};
assert(!one.empty());
assert(one == 1);
```

Question: Tell me about the code?

```cpp
class int_or_empty {
  int value_ = 0;
  bool empty_ = true;
public:
  int_or_empty() = default;

  template<std::convertible_to<int> T>
  constexpr int_or_empty(T&& v) :
    value_{std::forward<T>(v)}, empty_{false}
  {
  }

  constexpr bool empty() const
  { return empty_; }

  constexpr operator int() const
  { return value_; }
};
```

```cpp
int_or_empty empty;
assert(empty.empty());
```

```cpp
int_or_empty one{1};
assert(!one.empty());
assert(one == 1);
```

```cpp
int_or_empty empty2{empty};
assert(empty2.empty());
```

Question: Tell me about the code?

```
class int_or_empty {
  int value_ = 0;
  bool empty_ = true;
public:
  int_or_empty() = default;

  template<std::convertible_to<int> T>
  constexpr int_or_empty(T&& v) :
    value_{std::forward<T>(v)}, empty_{false}
  {
  }

  constexpr bool empty() const
  { return empty_; }

  constexpr operator int() const
  { return value_; }
};
```

```
int_or_empty empty;
assert(empty.empty());
```

```
int_or_empty one{1};
assert(!one.empty());
assert(one == 1);
```

```
int_or_empty empty2{empty};
assert(empty2.empty());
```

```
Assertion failed: empty2.empty()
```

Too perfect forwarding

```
class int_or_empty {
  int value_ = 0;
  bool empty_ = true;
public:
  int_or_empty() = default;

  template<std::convertible_to<int> T>
    requires (!std::same_as<std::remove_cvref_t<T>, int_or_empty>)
  constexpr int_or_empty(T&& v) :
    value_{std::forward<T>(v)}, empty_{false}
  {
  }

  constexpr bool empty() const
  { return empty_; }

  constexpr operator int() const
  { return value_; }
};
```

Recommendation

Beware of <u>constructors taking a single template parameter or a template parameter pack</u> of a forwarding reference type. <u>They will hijack your copy-constructor</u>.

Move Semantics with T&&

```
template<typename T>
void foo(T&&);
```

```
std::vector obj = {1, 2, 3};
const std::vector cobj = {1, 2, 3};
foo(obj);                      // lvalue -> should not compile
foo(cobj);                     // lvalue -> should not compile
foo(std::move(obj));           // rvalue -> should compile
foo(std::move(cobj));          // rvalue -> should compile
foo(std::vector{1, 2, 3});     // rvalue -> should compile
```

Move Semantics with T&&

```
template<typename T>
void foo(T&&);
```

```
std::vector obj = {1, 2, 3};
const std::vector cobj = {1, 2, 3};
foo(obj);                      // foo<std::vector<int>&>(std::vector<int>&)
foo(cobj);                     // foo<const std::vector<int>&>(const std::vector<int>&)
foo(std::move(obj));           // foo<std::vector<int>>(std::vector<int>&&)
foo(std::move(cobj));          // foo<const std::vector<int>>(const std::vector<int>&&)
foo(std::vector{1, 2, 3});     // foo<std::vector<int>>(std::vector<int>&&)
```

Move Semantics with T&&

```
template<typename T>
  requires (!std::is_lvalue_reference_v<T>)
void foo(T&&);
```

```
std::vector obj = {1, 2, 3};
const std::vector cobj = {1, 2, 3};
foo(obj);                      // ERROR
foo(cobj);                     // ERROR
foo(std::move(obj));           // OK
foo(std::move(cobj));          // OK
foo(std::vector{1, 2, 3});     // OK
```

Move Semantics with T&&

```
template<typename T>
  requires (!std::is_lvalue_reference_v<T>)
void foo(T&&);
```

```
std::vector obj = {1, 2, 3};
const std::vector cobj = {1, 2, 3};
foo(obj);                      // ERROR
foo(cobj);                     // ERROR
foo(std::move(obj));           // OK
foo(std::move(cobj));          // OK
foo(std::vector{1, 2, 3});     // OK
```

`std::is_reference_v<T>` is not enough here as someone may call the function with an explicit template parameter and not depend on a function template parameter deduction (i.e. **`foo<std::vector<int>&&>(v)`**).

Capturing the argument

Coroutine state stores coroutine parameters

- by-value parameters are **moved or copied**
- by-reference parameters remain **references**

Capturing the argument

Coroutine state stores coroutine parameters

- by-value parameters are **moved or copied**
- by-reference parameters remain **references**

```
template<typename T>
task<void> foo(T&& t) { /* ... */ }

template<typename U>
task<void> boo(U&& u)
{
  return foo(std::forward<U>(u));
}
```

```
int i = 123;
co_await boo(i);  // T -> int&; t -> int&
co_await boo(42); // T -> int; t -> int&&
```

Capturing the argument

Coroutine state stores coroutine parameters

- by-value parameters are moved or copied
- by-reference parameters remain references

```
template<typename T>
task<void> foo(T&& t) { /* ... */ }

template<typename U>
task<void> boo(U&& u)
{
  return foo(std::forward<U>(u));
}
```

```
int i = 123;
co_await boo(i);  // T -> int&; t -> int&
co_await boo(42); // T -> int; t -> int&&
```

```
template<typename T>
task<void> foo(T t) { /* ... */ }

template<typename U>
task<void> boo(U&& u)
{
  return foo<U>(std::forward<U>(u));
}
```

```
int i = 123;
co_await boo(i);  // T -> int&; t -> int&
co_await boo(42); // T -> int; t -> int
```

Perfect Returning

```
template<typename T>
auto wrapper(T&& v)
{
  return foo(std::forward<T>(v));
}
```

Perfect Returning

```
template<typename T>
auto wrapper(T&& v)
{
  return foo(std::forward<T>(v));
}
```

- **auto** always returns **by value**

Perfect Returning

```
template<typename T>
auto&& wrapper(T&& v)
{
  return foo(std::forward<T>(v));
}
```

Perfect Returning

```
template<typename T>
auto&& wrapper(T&& v)
{
  return foo(std::forward<T>(v));
}
```

- **auto&&** will always return a reference (i.e. to local object)

Perfect Returning

```
template<typename T>
decltype(auto) wrapper(T&& v)
{
  return foo(std::forward<T>(v));
}
```

Perfect Returning

```
template<typename T>
decltype(auto) wrapper(T&& v)
{
  return foo(std::forward<T>(v));
}
```

- `decltype(auto)` perfectly returns
  - temporary by value
  - reference by reference

Perfect Forwarding of the returned value

```
template<typename T>
decltype(auto) wrapper(T&& v)
{
  auto&& ret = foo1(std::forward<T>(v));
  // ...
  return foo2(std::forward<decltype(ret)>(ret));
}
```

Perfect Forwarding of the returned value

```
template<typename T>
decltype(auto) wrapper(T&& v)
{
  auto&& ret = foo1(std::forward<T>(v));
  // ...
  return foo2(std::forward<decltype(ret)>(ret));
}
```

Similarly to *lvalue references to const* and *rvalue references*, **auto&&** can be used to **extend the lifetimes of temporary objects**.

Universal Reference?

```
for(auto&& x : f()) {
  // ...
}
```

Universal Reference?

```
for(auto&& x : f()) {
  // ...
}
```

Usage of **auto&&** is the <u>safest way to use range for loops in a generic context</u> where we do not know which value category will be returned from a function (even though we do not forward the object anywhere).

Summary

Summary

- Use `const MyType&` when a **read access** is only needed
  - *binds to everything* and *cheap* to instantiate

Summary

- Use **`const MyType&`** when a **read access** is only needed
  - *binds to everything* and *cheap* to instantiate
- Use both **`const MyType&`** and **`MyType&&`** for **sinks** where it is not too expensive
  - consider **`MyType`** for *expensive scenarios*

Summary

- Use **`const MyType&`** when a **read access** is only needed
  - *binds to everything* and *cheap* to instantiate
- Use both **`const MyType&`** and **`MyType&&`** for **sinks** where it is not too expensive
  - consider **`MyType`** for *expensive scenarios*
- Use **`T&&`** to **forward** the argument
  - *beware of single-argument constructors*
  - remember that it *binds to everything* not just the type you need
  - remember that it is *not a rvalue reference*

Summary

- Use `const MyType&` when a **read access** is only needed
  - *binds to everything* and *cheap* to instantiate
- Use both `const MyType&` and `MyType&&` for **sinks** where it is not too expensive
  - consider `MyType` for *expensive scenarios*
- Use `T&&` to **forward** the argument
  - *beware of single-argument constructors*
  - remember that it *binds to everything* not just the type you need
  - remember that it is *not a rvalue reference*
- Use `decltype(auto)` as a function return type to **perfectly return** the result of another function's invocation

Summary

- Use **`const MyType&`** when a **read access** is only needed
  - *binds to everything* and *cheap* to instantiate
- Use both **`const MyType&`** and **`MyType&&`** for **sinks** where it is not too expensive
  - consider **`MyType`** for *expensive scenarios*
- Use **`T&&`** to **forward** the argument
  - *beware of single-argument constructors*
  - remember that it *binds to everything* not just the type you need
  - remember that it is *not a rvalue reference*
- Use **`decltype(auto)`** as a function return type to **perfectly return** the result of another function's invocation
- Use **`auto&&`** to **store a forwarding reference for later forwarding**
  - also useful in *range-for loops in generic context*

CAUTION
Programming
is addictive
(and too much fun)