Exploring Deducing `this`

(and what else I learned at CppCon 2022)

Richard Powell, rmpowell77@me.com, v1.0

- 5 day C++ conference
  - 600 in person Attendees, and more than 900 Online
- Over 100 Sessions, 5 tracks, Workshops, Lightning Talks and Open Sessions
- 5 Keynotes
  - Bjarne Stroustrup: “C++ in Constrained Environments”
  - Daniela Engert: “Contemporary C++ in Action”
  - Erik Rainey: “Using C++14 in an Embedded ‘SuperLoop’ Firmware”
  - Herb Sutter: “Can C++ be 10x Simplier & Safer”
  - Timur Doumler: “How C++23 Changes the Way We Write Code”

Timur Doumler: “How C++23 Changes the Way We Write Code”

Timur Doumler: “How C++23 Changes the Way We Write Code”

- The 4 big new things C++23:
  - deducing `this`
  - `std::expected`
  - `std::mdspan`
  - `std::print`

Functions Argument Deduction

```
void foo(Type <Value Category> arg);
```

```
foo(<expr>);
```

- A function argument specifies two things: the Type of the argument, and the Value Category (ref-qualifiers and CV-qualifiers).
- When calling a function, the resolution of the Expression determines which function, what type, and what Value Categories to use.

Vocab lesson

- L-Value: generally things you can take the address of
  - Named objects
- R-Value: generally things you can't take the address of.
  - Unnamed temporary objects
- Historical naming from their locations in an assignment:

```
VALUE = FUNCTION();
```

L-Value R-Value

Lvalue References

`Foo&`

- A reference to an “named” variable. Like an “alias” to another value.
- Can be non-`const` (mutable) or `const`.
- **Must** reference a value, cannot be reassigned to a different value.

Rvalue References

`Foo&&`

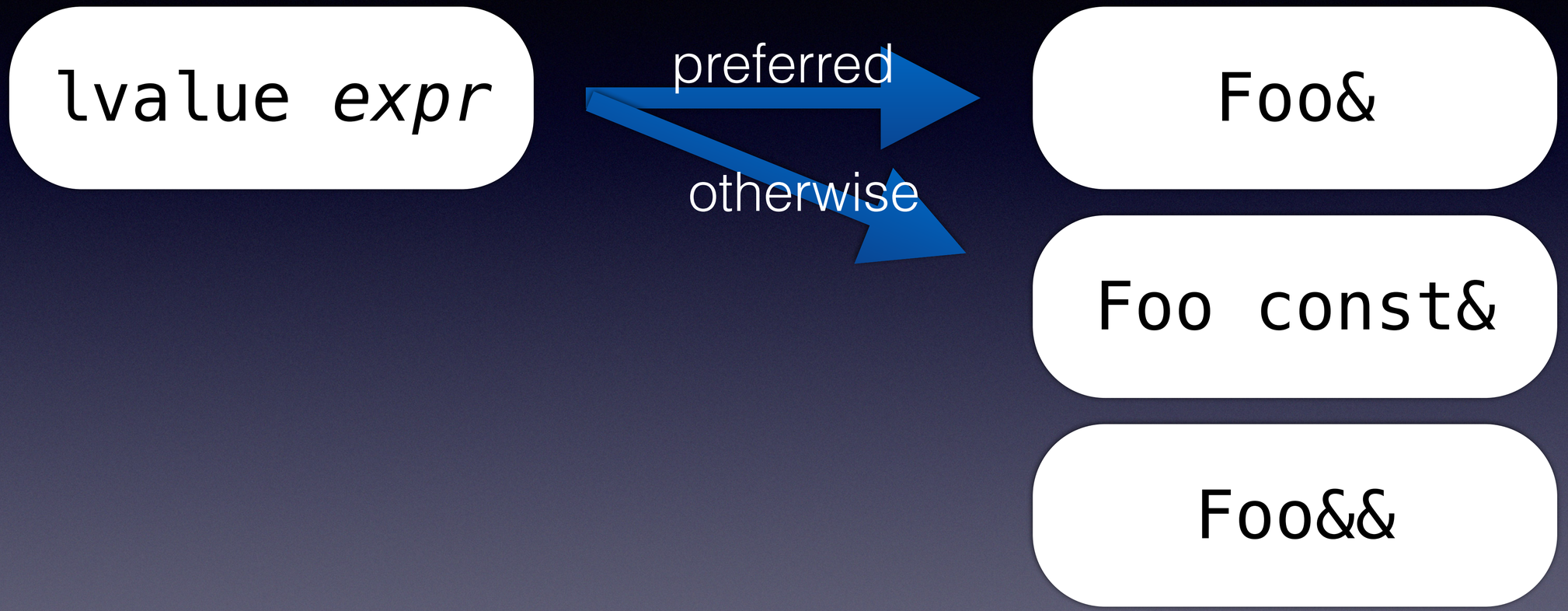
- A reference to an “unnamed” value.
- Like a lvalue reference, must reference a value, cannot be reassigned to a different value.
- Rarely `const`.
- **Rvalue references identify objects that may be moved from.**

Overload Functions by Value Category

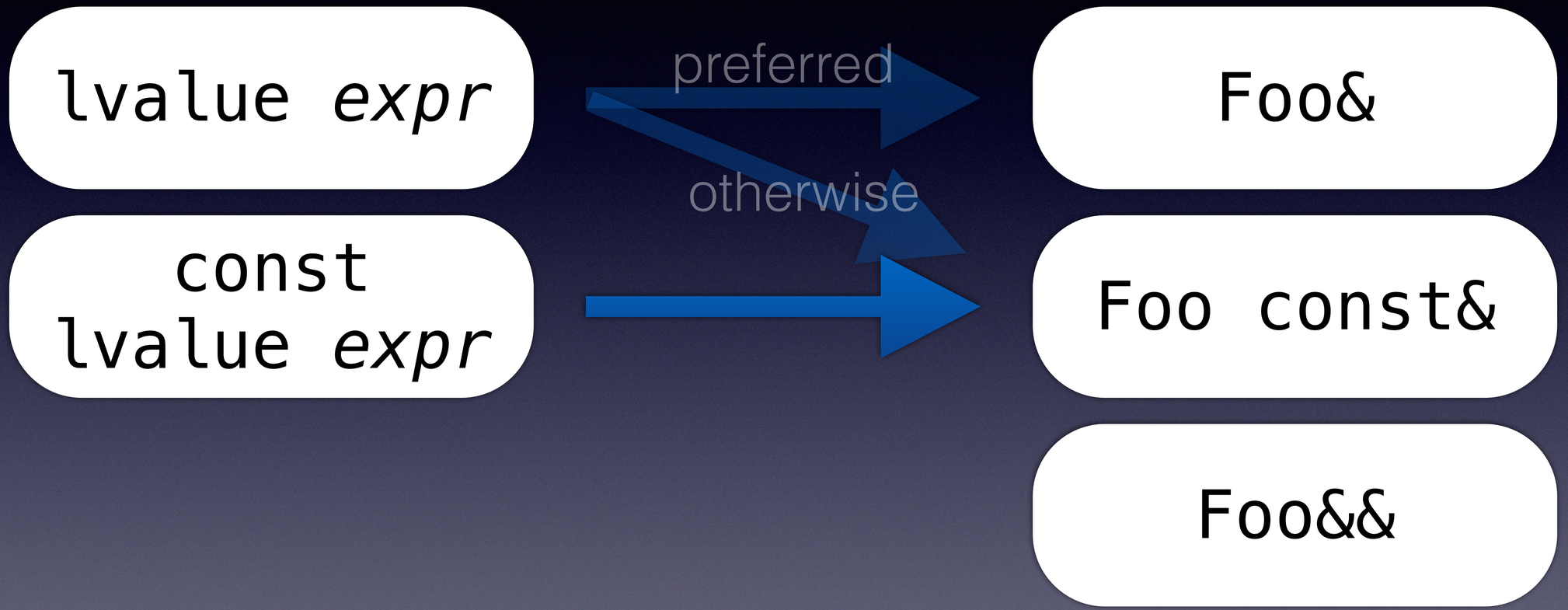
```
// by lvalue reference
// argument must be an lvalue
void foo(std::string& arg);

// by lvalue reference to const
// argument can be lvalue or rvalue
void foo(std::string const& arg);

// by rvalue reference
// argument must be rvalue
void foo(std::string&& arg);
```

- Overload functions to change which one gets called depending on the argument expression.
- We use Value Categories to express how the argument is used.

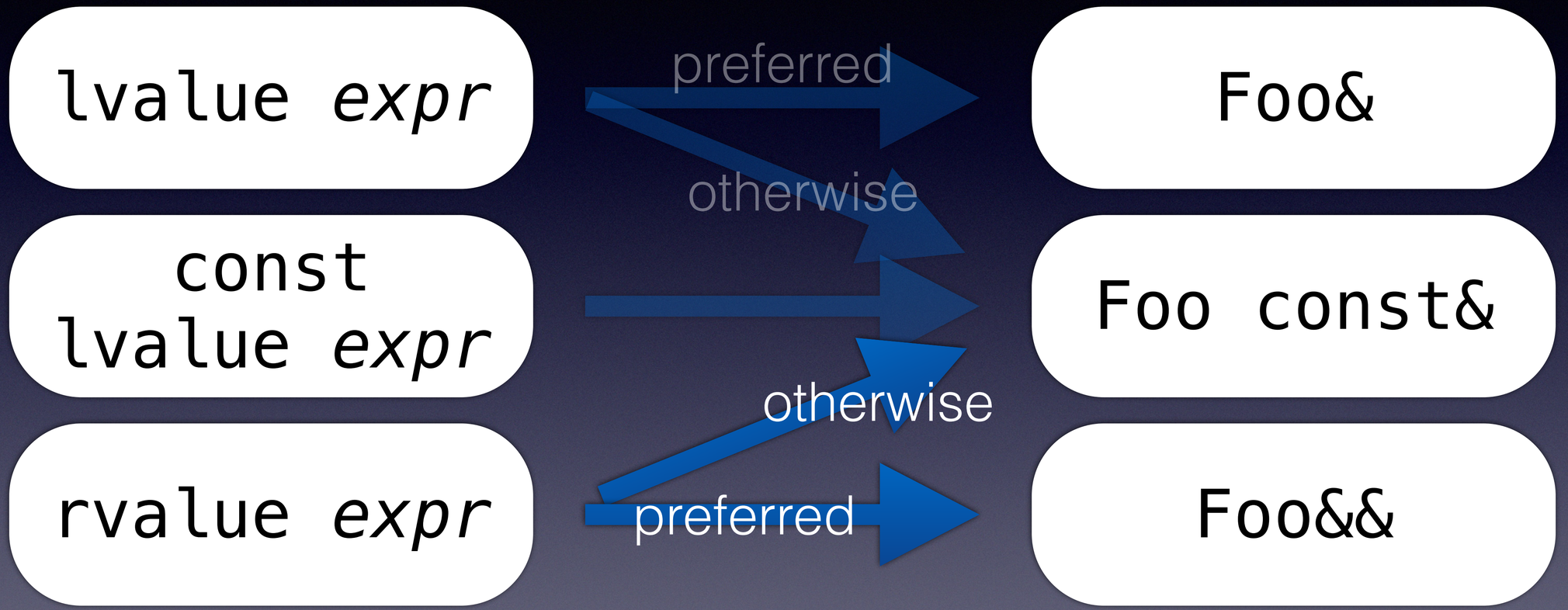
Binding rules



Binding rules

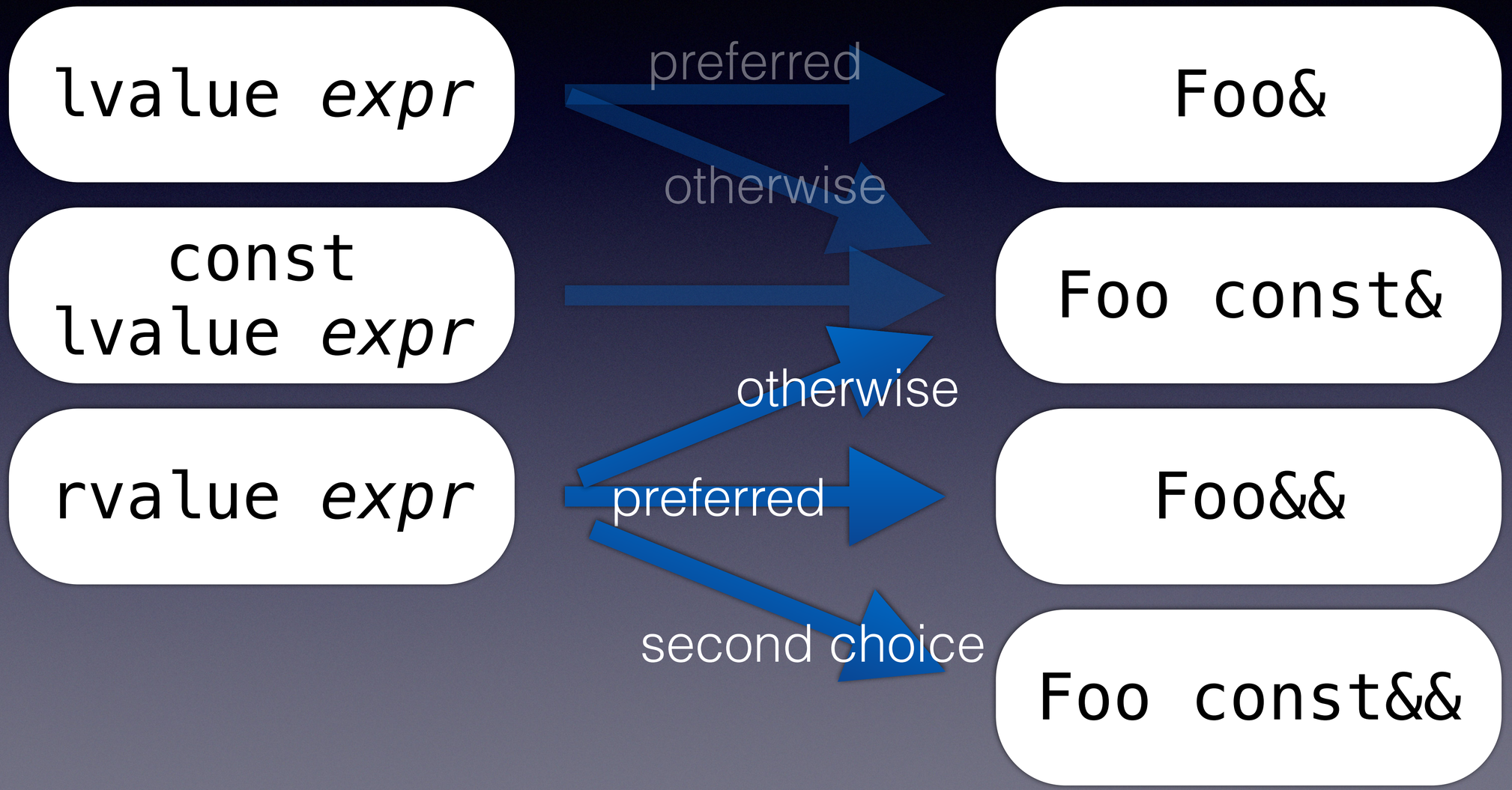


Binding rules



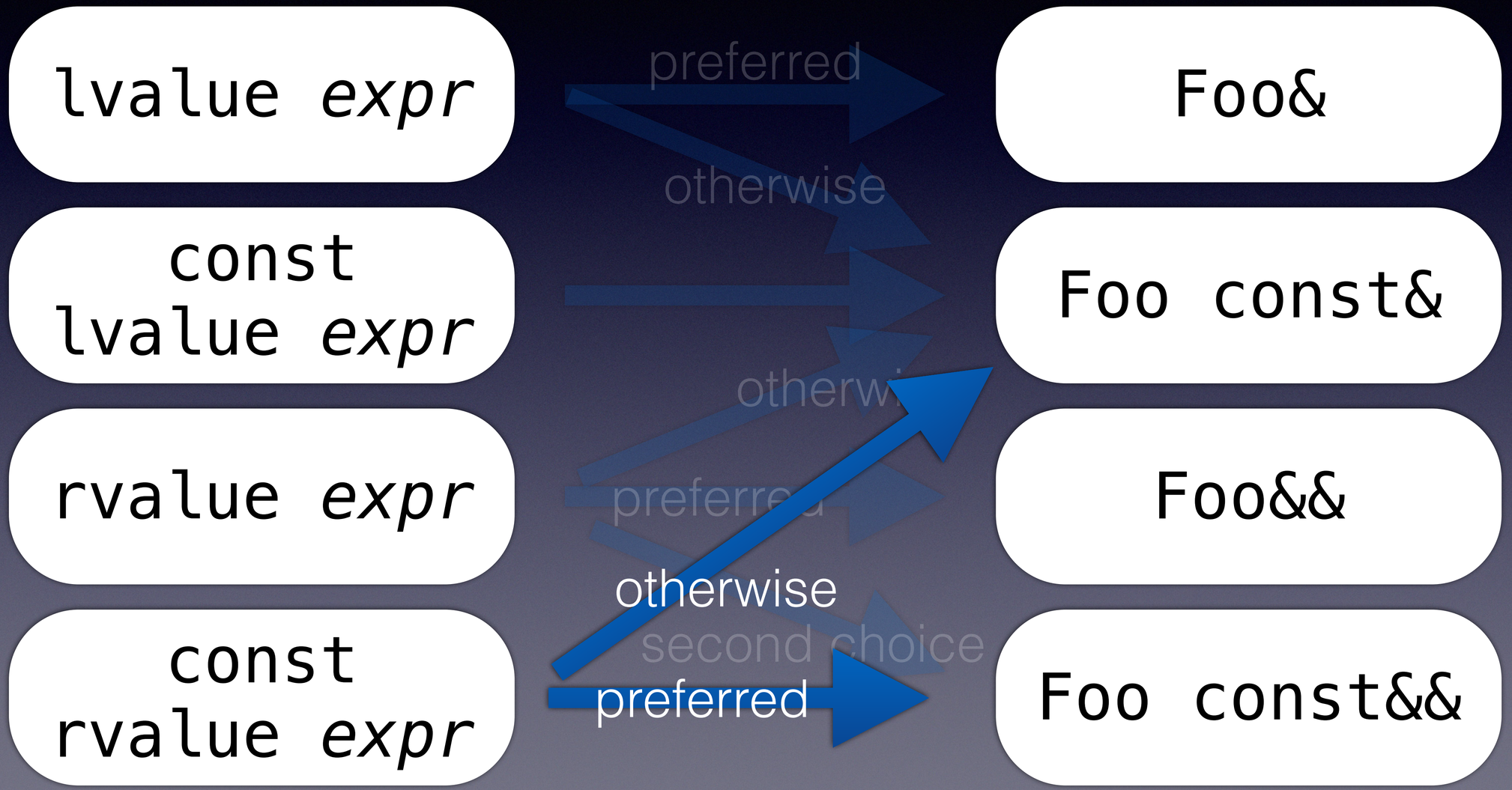
ADVANCED TOPIC

Binding rules

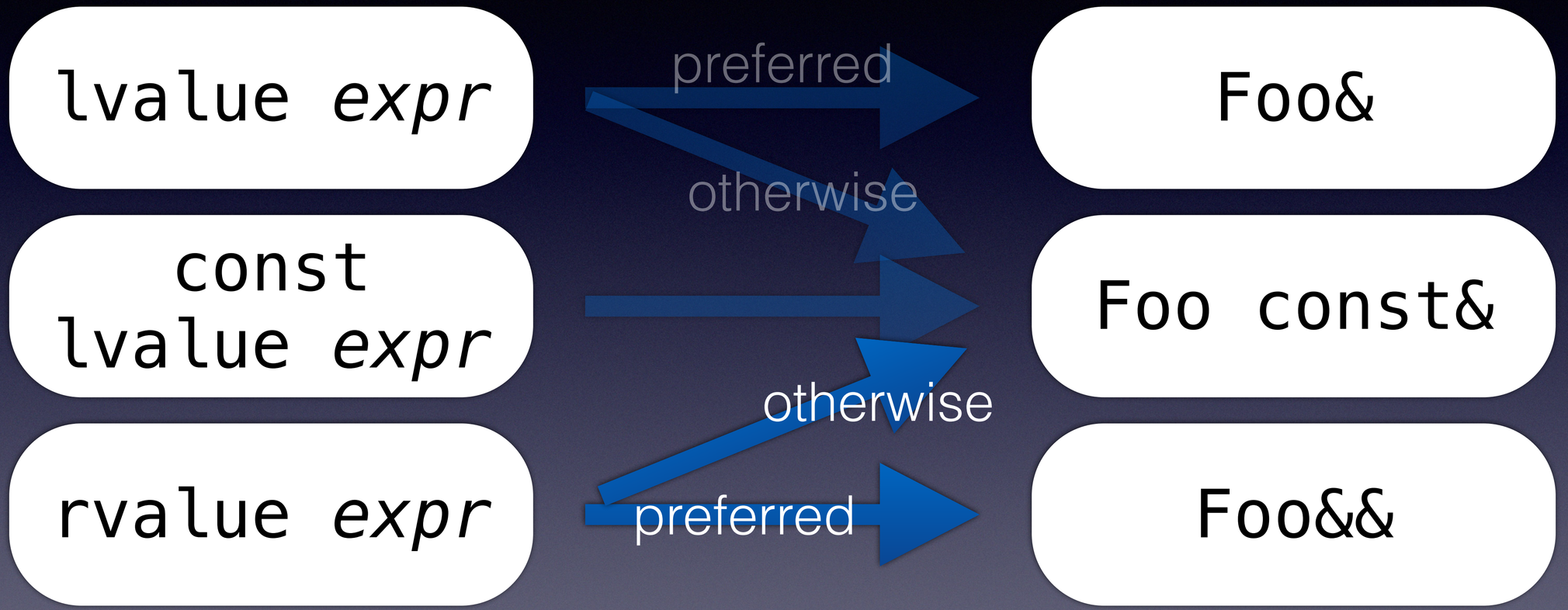


ADVANCED TOPIC

Binding rules



Binding rules



```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	1	2	3
Line A	☐	☐	☐
Line B	☐	☐	☐
Line C	☐	☐	☐
Line D	☐	☐	☐
Line E	☐	☐	☐

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☐	☐
Line C	☐	☐	☐
Line D	☐	☐	☐
Line E	☐	☐	☐

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☐	☒
Line C	☐	☐	☐
Line D	☐	☐	☐
Line E	☐	☐	☐

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☐	☒
Line C	☐	☒	☐
Line D	☐	☐	☐
Line E	☐	☐	☐

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☐	☒
Line C	☐	☒	☐
Line D	☐	☐	☒
Line E	☐	☐	☐

```cpp
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☐	☒
Line C	☐	☒	☐
Line D	☐	☐	☒
Line E	☐	☐	☒

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                       // F

    char hello[] { "hello" };
    foo(hello);                     // G
}
```

	1	2	3
Line F	☐	☐	☐
Line G	☐	☐	☐

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                       // F

    char hello[] { "hello" };
    foo(hello);                     // G
}
```

	1	2	3
Line F	☐	☐	☐
Line G	☐	☐	☐

Remember: rvalues are **unnamed** temporary values. `tmp` has a name

- lvalueness/rvalueness orthogonal to type
- rvalue: unnamed temporary objects
- lvalue: not rvalue
- An rvalue reference variable has a name; the variable itself is NOT an rvalue.

```
int main()
{
    std::string&& tmp = getString();
    foo(tmp);
}
```

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                       // F

    char hello[] { "hello" };
    foo(hello);                     // G
}
```

	1	2	3
Line F	☐	☐	☐
Line G	☐	☐	☐

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                       // F

    char hello[] { "hello" };
    foo(hello);                     // G
}
```

	1	2	3
Line F	☒	☐	☐
Line G	☐	☐	☐

```
void foo(std::string& arg);         // 1
void foo(std::string const& arg); // 2
void foo(std::string&& arg);        // 3

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                       // F

    char hello[] { "hello" };
    foo(hello);                     // G
}
```

	1	2	3
Line F	☒	☐	☐
Line G	☐	☐	☒

Overload member functions

- What you can do with an argument, you can also do with `this`.

```
struct Foo {
    // this is lvalue
    void func() &;

    // this is const lvalue
    void func() const&;

    // this is rvalue
    void func() &&;
};
```

```
struct Foo {
    void func() &;       // 1
    void func() const&; // 2
    void func() &&;      // 3
};

Foo getFoo() { return Foo{}; }

int main()
{
    Foo f = Foo{};
    f.func();                   // A

    Foo const b = Foo{};
    b.func();                   // B

    getFoo().func();            // C

    Foo&& d = getFoo();
    d.func();                   // D
}
```

	1	2	3
Line A	☐	☐	☐
Line B	☐	☐	☐
Line C	☐	☐	☐
Line D	☐	☐	☐

```
struct Foo {
    void func() &;       // 1
    void func() const&; // 2
    void func() &&;      // 3
};

Foo getFoo() { return Foo{}; }

int main()
{
    Foo f = Foo{};
    f.func();                    // A

    Foo const b = Foo{};
    b.func();                    // B

    getFoo().func();             // C

    Foo&& d = getFoo();
    d.func();                    // D
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☐	☐
Line C	☐	☐	☐
Line D	☐	☐	☐

```
struct Foo {
    void func() &;       // 1
    void func() const&; // 2
    void func() &&;      // 3
};

Foo getFoo() { return Foo{}; }

int main()
{
    Foo f = Foo{};
    f.func();                    // A

    Foo const b = Foo{};
    b.func();                    // B

    getFoo().func();             // C

    Foo&& d = getFoo();
    d.func();                    // D
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☒	☐
Line C	☐	☐	☐
Line D	☐	☐	☐

```
struct Foo {
    void func() &;       // 1
    void func() const&; // 2
    void func() &&;      // 3
};

Foo getFoo() { return Foo{}; }

int main()
{
    Foo f = Foo{};
    f.func();                    // A

    Foo const b = Foo{};
    b.func();                    // B

    getFoo().func();             // C

    Foo&& d = getFoo();
    d.func();                    // D
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☒	☐
Line C	☐	☐	☒
Line D	☐	☐	☐

```
struct Foo {
    void func() &;       // 1
    void func() const&; // 2
    void func() &&;      // 3
};

Foo getFoo() { return Foo{}; }

int main()
{
    Foo f = Foo{};
    f.func();                    // A

    Foo const b = Foo{};
    b.func();                    // B

    getFoo().func();             // C

    Foo&& d = getFoo();
    d.func();                    // D
}
```

	1	2	3
Line A	☒	☐	☐
Line B	☐	☒	☐
Line C	☐	☐	☒
Line D	☒	☐	☐

ADVANCED TOPIC

Member functions without Ref

```
struct Foo {
    // this is lvalue or rvalue
    void func();

    // this is const lvalue or const rvalue
    void func() const;
};
```

- Rules are slightly modified if you do not supply a ref-qualifier.
- Good practice to supply one.

Forwarding References

```
template <typename T>
void func(T&& t);
```

- A reference intended to “forward” along. For writing “passthrough” functions.
- T&& is either an rvalue reference or an lvalue reference.

```
template <typename T>
void func(ParamType t);

func(expr);
```

- If *`expr`* is an lvalue, both `T` and `ParamType` are deduced to be lvalue references.
- If *`expr`* is an rvalue, ignore the reference (&&) part and pattern-match *`expr`*'s type against `ParamType` to determine `T`.

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

T *ParamType*

Line A

Line B

Line C

Line D

Line E

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	T	*ParamType*
Line A	`string&`	
Line B		
Line C		
Line D		
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B		
Line C		
Line D		
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	
Line C		
Line D		
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	string&&
Line C		
Line D		
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                        // A
    foo(hi + "bye");                // B

    std::string const bye = "bye";
    foo(bye);                       // C
    foo(bye + "bye");               // D

    foo(getString());               // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	string&&
Line C	string const&	
Line D		
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	string&&
Line C	string const&	string const&
Line D		
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	string&&
Line C	string const&	string const&
Line D	string	
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	string&&
Line C	string const&	string const&
Line D	string	string&&
Line E		

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	string&&
Line C	string const&	string const&
Line D	string	string&&
Line E	string	

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

std::string getString();

int main()
{
    std::string hi = "hi";
    foo(hi);                          // A
    foo(hi + "bye");                  // B

    std::string const bye = "bye";
    foo(bye);                         // C
    foo(bye + "bye");                 // D

    foo(getString());                 // E
}
```

	T	*ParamType*
Line A	string&	string&
Line B	string	string&&
Line C	string const&	string const&
Line D	string	string&&
Line E	string	string&&

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

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                          // F

    char hello[] { "hello" };
    foo(hello);                        // G
}
```

T *ParamType*

Line F

Line G

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

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                          // F

    char hello[] { "hello" };
    foo(hello);                        // G
}
```

	T	*ParamType*
Line F	`string&`	
Line G		

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

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                          // F

    char hello[] { "hello" };
    foo(hello);                        // G
}
```

	T	*ParamType*
Line F	`string&`	`string&`
Line G		

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

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                          // F

    char hello[] { "hello" };
    foo(hello);                        // G
}
```

	T	*ParamType*
Line F	`string&`	`string&`
Line G	`char(&)[6]`	

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

std::string getString();

int main()
{
    std::string&& tmp = getString();
    foo(tmp);                          // F

    char hello[] { "hello" };
    foo(hello);                        // G
}
```

	T	*ParamType*
Line F	`string&`	`string&`
Line G	`char(&)[6]`	`char(&)[6]`

Putting it all together!

- You've been tasked with making sure that when certain object in your code base are access, an Authentication check needs to be run first.
- You decide that this could be done with a generic Gate class.

```
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```
int main() {
    auto a = DataEngine("hello");
    std::cout<<"a value = "<<a.doCalculation()<<"\n";
    a.updateEngine(" world");
    std::cout<<"a value = "<<a.doCalculation()<<"\n";
}
```

```
a value = 5
a value = 11
```

```
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```
int main() {
    auto a = make_gated<DataEngine>("hello");
    std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
    a.access().updateEngine(" world");
    std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
}
```

```
a value = 5
a value = 11
```

```
template <typename T>
class Gated {
private:
    T thing;
};
```

```
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

private:
    T thing;
};
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move<T>(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
int main() {
  ✅ auto a = make_gated<DataEngine>("hello");
  ❌ std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
  ❌ a.access().updateEngine(" world");
  ❌ std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
}
```

```
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
int main() {
  ✅ auto a = make_gated<DataEngine>("hello");
  ✅ std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
  ✅ a.access().updateEngine(" world");
  ✅ std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
}
```

```
a value = 5
a value = 5
```

Warning, bug in code

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T& access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
int main() {
  ✅ auto a = make_gated<DataEngine>("hello");
  ✅ std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
  ✅ a.access().updateEngine(" world");
  ✅ std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
}
```

```
a value = 5
a value = 11
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T& access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T& access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
int main() {
    auto b = make_gated<DataEngine>("constant");
    std::cout<<"b value = "<<b.access().doCalculation()<<"\n";
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T& access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
int main() {
  ✅ auto const b = make_gated<DataEngine>("constant");
  ❌ std::cout<<"b value = "<<b.access().doCalculation()<<"\n";
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
int main() {
  ✅ auto const b = make_gated<DataEngine>("constant");
  ✅ std::cout<<"b value = "<<b.access().doCalculation()<<"\n";
}
```

```
b value = 8
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }

    std::string s;
};
```

```cpp
int main() {
  ✅ auto const b = make_gated<DataEngine>("hello");
  ✅ std::cout<<"b value = "<<b.access().doCalculation()<<"\n";
  ❌ b.access().updateEngine(" world");
}
```

THIS IS A GOOD THING!!!

```
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        return s.size();
    }
    std::string s;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        std::cout<<"slow\n";
        return s.size();
    }
    auto doCalculation() && {
        std::cout<<"fast\n";
        return s.size();
    }
    std::string s;
};
```

```cpp
auto CreateDataEngine() { return DataEngine("Create"); }

int main() {
    auto c = CreateDataEngine().doCalculation();
    std::cout<<"c value = "<<c<<"\n";
}
```

```
fast
c value = 6
```

```
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) & {
        s = s.append(newS);
        return *this;
    }

    auto doCalculation() const& {
        std::cout<<"slow\n";
        return s.size();
    }
    auto doCalculation() && {
        std::cout<<"fast\n";
        return s.size();
    }
    std::string s;
};
```

```
auto CreateDataEngine() { return make_gated<DataEngine>("Create"); }

int main() {
    auto c = CreateDataEngine().access().doCalculation();
    std::cout<<"c value = "<<c<<"\n";
}
```

```
slow
c value = 6
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

    T&& access() && {
        // Do access check
        return std::move(thing);
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

    T&& access() && {
        // Do access check
        return std::move(thing);
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) &;

    auto doCalculation() const & {
        std::cout<<"slow\n";
        return s.size();
    }

    auto doCalculation() && {
        std::cout<<"fast\n";
        return s.size();
    }

    std::string s;
};
```

```cpp
auto CreateDataEngine() { return make_gated<DataEngine>("Create"); }

int main() {
    auto c = CreateDataEngine().access().doCalculation();
    std::cout<<"c value = "<<c<<"\n";
}
```

```
fast
c value = 6
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

    T&& access() && {
        // Do access check
        return std::move(thing);
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

    T&& access() && {
        // Do access check
        return std::move(thing);
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

Introducing Deducing `this`*!*

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    T& access() & {
        // Do access check
        return thing;
    }

    T const& access() const& {
        // Do access check
        return thing;
    }

    T&& access() && {
        // Do access check
        return std::move(thing);
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

Introducing Deducing `this`*!*

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template<typename... Args>
    Gated(Args&&... args)
        : thing(std::forward<Args>(args)...) {}

    template <typename Self>
    auto&& access(this Self&& self) {
        // Do access check
        return std::forward<Self>(self).thing;
    }

private:
    T thing;
};

template<typename T, typename... Args>
static auto make_gated(Args&&... args) {
    return Gated<T>(std::forward<Args>(args)...);
}
```

```
template <typename Self>
auto&& access(this Self&& self) {
    // Do access check
    return std::forward<Self>(self).thing;
}
```

- Explicit Object Parameters
- Make Explicit the Implicit `this` parameter.
- Opens all the existing rules of Type deduction!

```
template <typename Self>
auto&& access(this Self&& self) {
    // Do access check
    return std::forward<Self>(self).thing;
}
```

- If *`expr`* is an lvalue, both `Self` and `self` are deduced to be lvalue references.
- If *`expr`* is an rvalue, ignore the reference (&&) part and pattern-match *`expr`*'s type against `self` to determine `Self`.

```
template <typename Self>
auto&& access(this Self&& self) {
    // Do access check
    return std::forward<Self>(self).thing;
}
```

```cpp
template <typename T>
class Gated {
public:
    Gated(T&& t) : thing(std::move(t)) {}

    template <typename Self>
    auto&& access(this Self&& self) {
        // Do access check
        return std::forward<Self>(self).thing;
    }

private:
    T thing;
};
```

```cpp
struct DataEngine {
    DataEngine(std::string_view s) : s(s) {}

    DataEngine& updateEngine(std::string_view newS) &;

    auto doCalculation() const & {
        return s.size();
    }

    auto doCalculation() && {
        std::cout<<"fast\n";
        return s.size();
    }

    std::string s;
};
```

```cpp
auto CreateDataEngine() { return make_gated<DataEngine>("Create"); }
int main() {
    auto a = make_gated<DataEngine>("hello");
    std::cout<<"a value = "<<a.access().doCalculation()<<"\n";
    a.access().updateEngine(" world");
    std::cout<<"a value = "<<a.access().doCalculation()<<"\n";

    auto const b = make_gated<DataEngine>("constant");
    std::cout<<"b value = "<<b.access().doCalculation()<<"\n";

    auto c = CreateDataEngine().access().doCalculation();
    std::cout<<"c value = "<<c<<"\n";
}
```

```
a value = 5
a value = 11
b value = 8
fast
c value = 6
```

```
template <typename Self>
auto&& access(this Self&& self) {
    // Do access check
    return std::forward<Self>(self).thing;
}
```

- Additional Feature enabled:
  - Recursive lambdas
  - A new approach to mixins, a CRTP without the CRT
  - Move-or-copy-into-parameter support for member functions
  - Efficiency by avoiding double indirection with invocation
  - Perfect, sfinae-friendly call wrappers

```
template <typename Self>
auto&& access(this Self&& self) {
    // Do access check
    return std::forward<Self>(self).thing;
}
```

- This will change the way we program in C++23.

Questions?

- Timur Doumler: https://cppcon.digital-medium.co.uk/session/2022/how-c23-changes-the-way-we-write-code/
- “Effective Modern C++”, Scott Meyer
- Deducing this paper: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0847r6.html
- Ben Deane: Deducing this, CppCon2021: https://www.youtube.com/watch?v=jXf--bazhJw