

P2668R0



Rule based parameter passing

Bengt Gustafsson

Kona - 2022

# Presentation contents

- Rationale
- Credits
- Proposal contents

# Rationale

- C++ parameter passing is very complicated.
- Non-template and template parameters work differently.
- The implicit object reference works in yet another way.
- Writing overloads for optimal performance is tedious.
- Non-optimality does not generate compiler warnings.
- Optimal calling convention differs by type (P2666).

# Credits

- Thanks to Herb Sutter who wrote D0708 to highlight this as a problematic area and created statistics on the learning issues around this.
- Thanks to Barry Revzin who wrote P2481, requesting solutions to the automatic CRTP of deducing this.

# Proposal contents

- *Type sets* are used to generate declarations.
- Type sets can be templates.
- First template parameter injected as for concepts.
- Type sets inhibit universal references.
- Type set templates can be applied to functions.
- Standard type sets included in `std::`

# Basic syntax

- Type sets are enclosed in angle brackets.

`<float, double, const long double>`

- Function parameters can be type sets.

`auto sin(<float, double, const long double> a);`

- **type\_set** is a new keyword to name a type set.

`type_set floats = <float, double, const long double>;`

`auto cos(floats a);`

*Generates:*

`auto cos(float a);`

`auto cos(double a);`

`auto cos(const long double& a);`

# Usually type sets are templates

- **type\_set** declarations can be templates

```
template<typename T> type_set  fwd = <T, const T&, T&&>;
```

- **type\_set** templates are substituted like concepts

```
std::vector<T>::push_back( fwd T value) { ... }
```

*Generates:*

```
std::vector<T>::push_back(T value) { ... }  
std::vector<T>::push_back(const T& value) { ... }  
std::vector<T>::push_back(T&& value) { ... }
```

```
void myFunction( fwd auto x);           // Works with auto
```

```
void myConstrained( fwd myConcept auto x); // Works with onstrained auto.
```

# Type sets inhibit universal references

- Parameter uncvref type is first deduced as usual
- This type is substituted into the type set template
- The resulting types are never universal references

```
template<typename T> type_set fwd = <T, const T&, T&&>;
```

```
template<typename T> void func(fwd T x);
```

*Generates:*

```
template<typename T> void func(T x);
```

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

```
template<typename T> void func(T&& x);           // Not a universal reference!
```

```
template<typename T> void append(<T&> container, int value); // Not a universal reference!
```



# Type set templates can be applied to functions

- A type set template name can replace trailing cvref

// P2481 optional transform example.

```
template<typename T> struct optional {  
    template<typename F> constexpr auto transform(this fwd optional, F&&);  
};
```

// This proposal: No deducing this required.

```
template<typename T> struct optional {  
    template<typename F> constexpr auto transform(F&&) fwd;  
};
```

# Subtle problem solved with deducing this

**\*this** is const in a const qualified member function.

```
template<typename T> type_set dual = <const T&, T&>;  
template<typename T> class MyClass {  
    auto& get() dual { return m_member; }  
};
```

But **\*this** is **lvalue** in a rvalue qualified member function.

```
template<typename T> type_set fwd = <const T&, T&&>;  
template<typename T> class MyClass {  
    auto&& get() fwd { return m_member; } // Impossible  
};  
  
template<typename T> class MyClass {  
    auto&& get(this fwd MyClass self) { return std::forward<decltype(self)>(self).m_member; }  
};
```

# Standard type sets in std::

- Standard type sets in namespace nested in std

```
inline namespace std::type_set_templates {  
    template<typename T> type_set in = <const T, const T&>;  
    template<typename T> type_set ref = <T&>; // Name clash!  
    template<typename T> type_set dual = <const T&, T&>;  
    template<typename T> type_set fwd = <T, const T&, T&&>;  
    template<typename T> type_set mv = <T, T&&>;  
}  
using namespace std::type_set_templates;
```

- The names thus usable without qualification
- If shadowed they can be qualified with **std::** only.

# Issues treated in the proposal

- Should type sets be usable in all declarations?
- Overloading between regular and generated functions.
- Can type set templates be chained?
- Can type sets be declared in class and/or block scope?
- How about dependent type sets?
- Alternate spellings (including reusing **typedef**).
- The effects of <Type> as an inlined type alias.

# Open issues

- Implementation in .cpp files?
- Name mangling: Same or different?
- Can trailing type set template make `*this` an rvalue?
- Can a type set used in a parameter declaration cause arrays to be passed by value?
- Reassessing using **using** to declare type sets.

# Trailing type set template makes `*this` an rvalue

- To reduce confusion and increase WYSIWYG:  
`*this` is **rvalue** in a rvalue qualified member function.

```
template<typename T> type_set fwd = <const T&, T&&>;
```

```
template<typename T> class MyClass {  
    auto&& get() fwd { return m_member; }    // Thanks in part to P2666 "Last use optimization"  
};
```

```
template<typename T> class MyClass {  
    auto&& get(this fwd MyClass self) { return self.m_member; }    // Does not need a special rule  
};
```

# Passing arrays by value

- To reduce confusion and increase WYSIWYG:

```
void myFuntion(<int[10]> arr);
```

```
// Force by value using <>
```

```
void optimalCall(in int arr[10]);
```

```
// Let compiler select from in overloads
```

*Generates:*

```
void optimalCall(const int[10] arr);  
void optimalCall(const int (&arr)[10]);
```

# Reassessing using **using** to declare type sets.

- The reason for a new keyword was to be able to disambiguate a dependent name as a **type\_set**.
- But: Parsing does not need to know if a dependent name is a type or a type set.
- We could drop **<>**, at the cost of **loosing in-line type set feature**:

```
using floats = float, double, const double&;
```

```
template<typename T> using fwd = T, const T&, T&&;
```



# Relation to other proposals

- P2665: Overload selection:  
*Let compiler select by value or by reference.*
- P2666: Last use optimization:  
*Simplifies finding rvalues when calling functions.*
- Coming up later:  
*Labelled types provides named parameters.*  
*Simplified declarators with all of the type first.*