Role based parameter passing

Document: P2668R0

Date: 2022-10-14

Project: Programming language C++

Audience: EWG(I)

Reply-to: Bengt Gustafsson, bengt.gustafsson@beamways.com

Introduction

This proposal introduces a way to get a role based parameter passing style similar to Herb Sutter's ideas in D0708. The basic idea is to be able to name closed sets of types and use them to create closed sets of function declarations. Here are some examples introducing the principles. Step 7 and 8 are the typical uses of the feature.

```
// 1. Declare three sin functions using a type set.
auto sin(<float, double, const long double&> x) { ... }

// 2. Declare a type_set using a new keyword
type_set floats = <float, double, const long double&>;

// 3. Use it for some functions
auto cos(floats x) { ... }
auto tan(floats x) { ... }

// 4. Declare nine functions at once.
auto floating_multiply(floats a, floats b) { return a * b; }

// 5. Overload floating_multiply for one parameter type combination
double floating_multiply(float a, float b) { return double(a) * b; }

// 6. Declare a type_set template
template<typename T> type_set fwd = <const T&, T&&>;

// 7. Declare a pair of functions that forward a std::string parameter
void set_name(fwd std::string name) { m_name = std::forward<decltype(name)>
(name); }

// 8. Declare a pair of template functions, (not three, i.e. T&& does not
implicitly mean T& too)
template<typename T> void use_value(fwd T value);
void use_value2(fwd auto value);  // Shorthand syntax

// 9. Constraining the type with concepts is supported
template<std::regular T> void use_value3(fwd T value);
void use_value4(fwd std::regular auto value);  // Shorthand syntax

// 10. A type set of one can be used to disable the universal reference aspect of
templated parameters
template<typename T> void only_rvalue(<T&&> rvalue);
template<typename T> void only_reference(<T&> lvalue);

class Test {
    // 11. A templated type_set can be placed after a non-static member function
declaration, but
    //     this requires additional help.
    void give_contents(Taker& taker) fwd {
taker.take(std::forward<decltype(m_contents)>(m_contents)); }

    // 12. Deducing this can now easily avoid CRTP.
    void give_contents2(this fwd Test self, Taker& taker) {
        taker.take(std::forward<decltype(self.m_contents)>(self.m_contents));
```

```
    }
};
```

1. The syntax for a type set when used directly in a declaration uses angle brackets, which is easily parsed as `<` can never start an expression or declaration today.
2. A new reserved word (bike-sheddable) is introduced to allow naming sets of types for later use.
3. When used as a parameter type a type_set must be recognized by the compiler during parsing, just like a concept-name has to be recognized when parsing a template parameter list today.
4. When more than one parameter is declared using a type set functions with all combinations of parameter types are declared.
5. Function declarations with type sets can be overloaded with regular functions.
6. type_set declarations can be templates where the first template parameter must be a type. The template arguments are substituted into the type set elements.
7. When a templated type_set is used in a function declaration the first template parameter is taken from the succeeding type, in a similar fashion to how concepts substitute the type to test into the first template parameter. But note that we are still declaring an overload set, in this case consisting of `set_name(const std::string& name)` and `set_name(std::string&& name)`. Inside the function body the type of `name` differs between the overloads which share the definition, so forwarding is required.
8. When combined with a template parameter a set of template functions is created, which have the universal reference mechanism disabled for parameters that are specified using type sets.
9. When types are constrained by concepts the argument type is subjected to the concept check and if it passes all the function specializations generated from the declaration are viable.
10. The function signatures generated using type sets, even if templated, do not automatically expand to universal references like regular template parameters. This can be used to avoid undesired function overloads from being formed.
11. When applied to a function the type of the this pointer is affected, but this code always copies as members are lvalues in rvalue qualified functions, which means that there is no way to forward m_contents correctly for both generated function definitions.
12. "Deducing this" comes to the rescue, and the example also shows how to avoid the "automatic CRTP" problem of deducing this.

Note that with P2666 (last use optimization) the std::forward calls in 7, 11, 12 are unnecessary, as these are all last uses of their respective parameters:

```
// 7. Declare a pair of functions that forward a std::string parameter
void set_name(fwd std::string name) { m_name = name; }

 class Test {
    // 11. A templated type_set can be placed after a non-static member function
declaration
    void give_contents(Taker& taker) fwd { taker.take(m_contents); }

    // 12. Deducing this can now easily avoid CRTP. This works exactly as 11, but
is more verbose.
    void give_contents2(this fwd Test self, Taker& taker) {
        taker.take(self.m_contents);
    }
};
```

Scope and motivation

The motivation for this proposal is to simplify creating optimal overload sets without having to think so much about it. P0708 contains an analysis based on the amount of rules and conventions in style guides and education material devoted to function declarations and function calling. In contrast with P0708 this proposal contains a general mechanism rather than a fixed set of named calling conventions. This said, a small set of useful type set templates in the namespace std::type_set_templates are also included in the proposal.

The reason for choosing a general mechanism is partly that this is the C++ way: Build general abstractions instead of special purpose features. But also that this actually provides additional useful functionality such as the `<T&>` construct.

This proposal works best in concert with P2665 which allows an overload set to contain both `T` and `const T&`, and P2666 which greatly reduces the need for using `std::forward` and `std::move`. Example 7 and 11 in the introduction clearly shows the advantages of P2666.

Impact on the standard

The primary impact on parsing is rules for function parameters, as well as the new type set declarations. The second large area affected is rules to synthesize the individual declarations and how overloading works.

The type set clause

Parsing a type-set-clause is similar to parsing a template-argument-list. Such parsing is started when a `<` is seen where a type-id is allowed. type-set-clauses are only allowed when a parameter is declared and when a type-set is defined.

```
type-set-clause:
    < type-id-list >

type-id-list:
    type-id
    type-id-list , type-id
```

type set declarations

The new keyword **type_set** is used only as a means to create named type sets, with or without template-introducer. With this proposal it is only valid in namespace scope, like concepts are today.

In contrast with concepts typeset declarations can be non-templated, although this is not a very important feature. The absolute majority of cases there will have a single type parameter but in it seems that it could be useful to allow additional template parameters, in a similar vein as for concepts. For now no examples of this are in this proposal, this needs further exploration. As template-declaration consists of a template introducer followed by *declaration* this is where the new type-set-declaration production must go.

```
type-set-declaration:
    type_set ts-identifier = type-set-clause ;

declaration:
    --- current cases ---
    type-set-declaration
```

Usage of a type set

In the grammar excerpt below a new parameter-decl-specifier production has been introduced to reflect the fact that in addition to the decl-specifier-seq case there are now cases strting with a ts-name. As a decl-specifier-seq may start with the name of a user defined type the parser must check the symbol table to see if an identifier denotes a type or a type-set before parsing can commence. To see if a ts-name starts a type-set or a ts-template clause the parser must again refer to the symbol table.

```
ts-name:
    type-set-name
    nested-name-specifier type-set-name

type-set:
    type-set-clause:
    ts-name

ts-template:
    ts-name
    ts-name < template-argument-list >

parameter-decl-specifier:
    decl-specifier-seq
    type-set
    ts-template decl-specifier-seq

parameter-declaration:
    parameter-decl-specifier declarator
    parameter-decl-specifier declarator = initializer
    parameter-decl-specifier abstract-declarator
    parameter-decl-specifier abstract-declarator = initalizer
```

Note: *For clarity this grammar does not show the attribute-specifiers, the leading this or that some elements of parameter-declaration cases are optional.*

type-set includes non-templated ts-names only.

ts-template includes ts-names of type set templates with one type parameter and type sets with more than one template parameter curried to one parameter using a template-argument-list.

If a type set has more than one template parameter a template-argument-list containing values from template parameter 2 and up must be added, just as for concepts.

While parsing after a ts-template continues as usual for a parameter-declaration the resulting type must not be a reference type or top level cv-qualified, as type-set templates take over the responsibility to add such qualifications.

type-set after parameter list of member functions

A new case for parameters-and-qualifiers is added to allow type set templates to be used to qualify the implicit object reference of a non-static member function. This replaces both the cv-qualifier-seq and ref-cualifier as such qualifications must be inside the type-list elements.

Note that there are restrictions on the type-set elements: They must refer to a cvref qualified type derived from the class of the member function being declared.

```
parameters-and-qualifiers:
    --- pre-existing case ---
    (parameter-declaration-clause) cv-qualifier-seq ref-qualifier exception-
specification

    --- new case ---
    (parameter-declaration-clause) ts-template exception-specification
```

Examples of more complicated parameter types

Here are some examples of how type-set template parameter declarations interact with the complexities of C++ declarations. Basically the type-set template comes first and then a regular declararation, except that it can't declare a reference or a top-level const (const closest to the declared name).

```
// p2 is ok as the const is not top-level, it just indicates that the pointee is
const.
void ok_ptrs(in int* p1, ref const float* p2);

// These have top level const or references, which is not allowed.
void bad_ptrs(in int*& p1, ref const float* const p2);

// A ref to an array is ok, but you can not assign to it anyway, unless arrays
are made regular.
void ok_arrs(in int[10], ref const float[10]);

// Function pointers
void ok_funcs(in int(*p1)(int, bool), ref const float(*p1)(float, bool));
void bad_funcs(in int(*&p1)(int, bool), ref const float(const* o2)(float,
bool));
```

```
// Method pointers
void ok_methods(in int (MyClass::* p1)(int, bool), ref const float(MyClass::*
p2)(float, bool));
void bad_methods(in int (MyClass::*& p1)(int, bool), ref const float(const
MyClass::* p2)(float, bool));
```

Note: This could be our chance to make arrays regular. The main blocker for this today is that we can't pass arrays by value anyway. With type-sets, which already modify behaviour of template parameters, we could normalize array parameters to allow passing arrays by value using the syntax `f(<int[10]> x)`. With a predefined type-set template **val** we could write `f(val int x[10])`. This is not proposed in R0 but can be added within the scope of this proposal, but not separately. Having a **val** type set template is probably not needed for other types, as while it does convey intent "I need a copy of this because I'm going to change it locally" this intent is not interesting for callers of the function, and programmers can continue using just `T` in this case. On the other hand organizations will soon set up rules that all parameters *shall* use type sets so the standard needs to offer a complete enough set.

Subnote: The reason for having to write the array size inside the type-set list when declaring it in situ but after the parameter name in the **val** case is that as val is a type-set template the type of the rest of the parameter-declarator is determined and then substituted into the type-set template. So after this substitution int[10] is in the type-set anyway.

Parameter packs declared with a type set

When a parameter pack is declared with a type set template this works separately for each argument pack element. This just means that the compiler makes the selection between the elements of the type set individually for each pack element, much like it selects rvalue or lvalue individually today when a template parameter is declared `T&&...`

Note: It would be possible to specify that a non-template type set can be followed by an ellipsis to declare a pack of equally typed elements, but this is probably best left to a separate proposal.

Semantics of a function declaration with type sets

The easiest way of thinking about function declarations where one or more parameters are specified using type sets is that it *generates* a set of function declarations with all combinations generated when selecting one type-id in the type set of each parameter. If one or more parameters are of template type (longhand or shorthand) each of the generated function declarations is a function template declaration. However, even in the template case the parameter types of these function templates are never universal references.

In the case of function definitions, all of the generated definitions have the same function body, but the semantics may vary due to the varying parameter cv-qualifiactions and value categories (or even types).

Overloading rules

Rules for function declarations

As a function declaration with type sets generates multiple individual function we call this a *generating* declaration in the following discussion. The individual resulting declarations are called *generated* declarations. A function declaration with no type set parameters is called a *plain* function declaration.

With overloaded generating functions partially overlapping sets of generated declarations can be created. This is not allowed, except that a plain declaration can be identical to a generated function declaration.

```
// Make two definitions which both generate the math(double, double) function
definition.
auto math(floats a, double b) { return a + b; }
auto math(double a, floats b) { return a + b; }  // Error!
```

Both generating and plain function declarations can be repeated.

Rules for function definitions

Function definitions are function declarations, so abide by the rules stated above. This ensures that there is only one function body for each generated function definition, except for the case of a plain definition which can replace one of the generated function definitions.

To make this work with separate compilation a declaration (or definition) of the plain function must have been seen when the definition of the generating function is seen. Having seen this declaration prevents code generation for the generated function it replaces.

If these rules are violated for a non-inline function a compile time or link time error is encountered, with a few exceptions. The basic problem is that if the definition of a generating function is encountered it will generate code for the generated function that has been overloaded by a plain function too, which should never be used. The best case scenario is that the plain function declaration or definition is encountered later in the same translation unit. This error can easily be detected.

However, the hard cases relate to separate compilation of these function pairs. As both function definitions are code generated into separate object modules this error may or may not get detected by the linker depending on whether one or both are placed in library (Linux: archive or shared object) files. This can be viewed as a type of ODR violation.

inline functions

For inline function definitions the situation is worse, but inline function declarations should not be put in implementation files so the only way a problem can occur is that the generating function and plain function definition are in different headers included by different translation units. This is very similar to how the wrong function overload can be selected if not all headers containing overloads of the function are included.

Guaranteed same behaviour

To prevent that there is ambiguity about which function body should be code generated a modified ODR rule applies. This means that this example will fail as indicated, which is very similar to how explicitly specializing a template after it has been implicitly specialized is handled:

```
// Remember floats from the first example:
type_set floats = <float, double, const long double&>;

// Make two definitions which both generate the math(double, double) function
definition.
auto math(floats a, floats b) { return a + b; }

auto x = math(1.0, 2.0);    // return 3

auto math(double a, double b) { return a - b; }  // Error: math(double, double)
has already been used.
```

The same rules apply to template functions. If there are both template and non-template (generated) functions in the resulting overload set this is resolved using the current rules.Risk assessment

Name mangling

The rules presented above are needed to make sure that the same name mangling rules that we have today can be used. The mangled name is the same for plain functions and generated functions. This is true also for function template specializations.

This means that as long as no generated declarations are removed programmers are free to rewrite declarations using type sets, without having to recompile all code. So for instance `std::vector::push_back` can now be written as one function, and with the help of P2665 and P2666 it can be defined like this, and get better optimization than today:

```
template<typename T> type_set fwd = <T, const T&, T&&>;     // P2665: T
overloads with const T&.

void push_back(fwd T value) {
    reserve(size() + 1);
    traits_type::construct(m_end++, value);     // P2666: Last use of value
doesn't need move/forward
    m_end++;
}
```

This proposal allows one push_back to generate three function definitions, including both `push_back(T)` and `push_back(const T&)`. The compiler selects which of these to call according to P2665. In the `construct` call `value` is passed as an rvalue in the `push_back(T)` and `push_back(T&&)` definitions without any move or forward wrapping thanks to P2666. This is fortunate as the value category of value is only available in `decltype(value)` in this type of shared function definitions.

Note: An alternative would be to mangle named of generated functions differently from plain functions with the same parameter types. This could solve some of the ODR problems noted above but the author thinks that the backwards compatibility issues this creates would be detrimental to the migration of code bases to using type set based parameters.

Standard type set templates

This proposal also defines a number of standard type set templates. The proposal is to put these in a inline namespace in std and use that namespace, so that unless you have redefined a name you don't have to provide any namespace when using them. As these identifiers will soon be used in all function declarations their names are kept short and close to what exists in other languages.

Note that *out*, as specified in P0708, that the argument is constructed in place by the called function, is not possible to attain without other fairly far reaching changed to the language.

```
namespace std

inline namespace type_set_templates {

template<typename T> type_set in = <const T, const T&>;
template<typename T> type_set ref = <T&>;
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&&>;

}

// This allows the use of the declard names except if hidden by other
declarations, in which case they are instead reachable as std::fwd etc.
using namespace std::type_set_templates;
```

This set of calling conventions covers almost all parameter passing needs. For completeness an `obs = <const T&>` could be added, but this is actually superfluous as `const T&` is not a universal reference for templates anyway. Here is a rundown of the uses of each of these standard calling conventions:

in

The **in** convention allows the compiler to select whether to call the function with a by value or by const reference parameter (given P2665). As the compiler may select to call using `const T&` there is no way for the function body to know if it can modify the parameter value, so the by value alternative is `const T`, enforcing that the parameter value can never be modified.

ref

ref parameters typically refer to objects like containers which are modified by the function. The advantage of using **ref** rather than `T&` is that for templated parameters the function is not callable with const data, avoiding the deeply nested errors we se, at the point of mutation today, if the function template called with a const argument.

dual

dual is usable when the return type of a function depends on its input. Often this is combined with *deducing this* to prevent automatic CRTP from happening:

```
class Base {
public:
    // C++20
    auto& get20() const { return s; }
```

```
    auto& get20() { return s; }

    // With deducing this you can write this, but beware if a subclass has its
own s member!
    template<typename T> auto& get23(T&& this self) { return self.s; }

    // With this proposal we're back to safety and gained some characters.
    auto& get26(this dual Base self) { return self.s; }

    // We can also bypass deducing this and get this simpler definition
    auto& get26() dual { return s; }

    std::string s;
};
```

There are other use cases for **dual** but they are not that common.

I had a hard time coming up with a name that is short and conveys the "pair of const and non-const" nature of this. dual only conveys that there are two types in the set, which is *not* the primary feature of **dual**. Maybe the best would be to come up with a ridiculous abbreviation like **canc** for "const and non-const", but better ideas are appreciated.

fwd

The **fwd** calling convention is useful mainly for non-templates as `T&&` works for templates. However, even for templates the inclusion of `T` in the type set offers the compiler to pass the parameter by value and if the type is easy to copy this is likely going to be selected regardless of if the argument is a lvalue or rvalue, reducing the number of specializations.

mv

The **mv** calling convention is useful for templates where `T&&` does not work as a *move only* parameter. This helps avoiding bugs of this kind:

```
template<typename T> std::vector<T> dataVectors;

template<typename T> void addToGlobal(T&& data)
{
    dataVectors<T>::push_back(std::move(data));
}

MyType lvalue;
addToGlobal(lvalue);

// Oops: lvalue now in a moved from state!
```

The problem is that the `T` of `addToGlobal` is bound to `T&` by the call site, and then `std::move` happily converts that to `T&&` and push_back steals the value. The bug is that `std::move` should have been `std::forward`.

Notes on standard type set qualification

Placing the standard type set templates in a inline nested namespace inside std seems like the best possible solution to issues regarding name clashes and reachability of both the standard type set templates themselves and other uses of their names.

The `using namespace std::type_set_templates;` directive brings these names into the root namespace thus making like-named entities in the root namespace ambiguous with them. Fortunately, as they are actually declared in an inline namespace in std they can still be reached as std::fwd etc. At the same time the global name they clashed with are available as ::fwd and as the ambiguity is signalled by the compiler it is fairly easy to fix.

If a standard type set template name clashes with a name inside a namespace it is instead shadowed by that name. This means that to use the standard type set template you have to qualify them with std:: while to use the other conflicting name you don't have to do anything. The problem with this is that if the clashing name is a type there will be surprises when you start declaring functions with unnamed parameters, as their signature are transformed to a declaration of a named parameter, which works unless the original parameter type is a primitive type.

```
#include <vector>

#include <type_sets>   // I want to use standard type sets

namespace my_namespace {
    class other{};

    void f1(fwd other);     // unnamed parameter of type other
    void f3(fwd other x);   // parameter of type other, named x
    void f4(fwd int);       // unnamed parameter of type int

    class fwd{};

    void f4(fwd other);     // parameter named other of type my_namespace::fwd
    void f5(fwd other x);   // Error: Junk after parameter name
    void f6(fwd int);       // Error: tried to name a parameter int
}
```

One problem that seems hard to avoid is when a parameter has the same name as a standard type set template. This will probably happen relatively often for the `in` name. So when migrating code to start using standard type set templates in declarations some clashes with parameter names will occur. These clashes will all be loud, as the newly declared parameter name is never a type that could be legally the type of the next parameter:

```
void my_two_input_function(in float in, in float extra);
```

Here the first parameter declaration introduces a parameter name which the second parameter tries to use as the standard type set template. This fails of course, but can be disambiguated with `std::in`.

Note that shadowing type set template names by local variables is no problem as type set template names are not useful inside function bodies anyway. As local variable names tend to be short while externally visible names tend to be longer most name clashes will probably be with local variables, and thus inconsequential.

When is the using directive introduced

Assuming that standard library implementers wants to start using the standard type set templates when needed inside the standard headers a header file containing them will have to be by all standard headers. The name of this header is not standardized, only that the standard type set names are available qualified by std:: if any standard header is included.

A special header containing only an include of the non-standardized header and a `namespace std::type_set_templates;` declaration is standardized, with the name `<type_sets>`.

This special header adds clarity and by being an opt-in feature simplifies migrating libraries to using them. However, as soon as a header includes `<type_sets>` all files including *that* header file are affected. The author still think this is a nice and clean approach.

Syntax alternatives considered

Repurpose typedef

One rather appealing syntax alternative for type-set-declaration is to recirculate the typedef keyword which is currently hopelessly out of vouge. Unfortunately it is still allowed in its previous capacity, but it would probably be possible to differentiate anyway:

```
typedef int* intp;      // C style typedef.

// Should be possible: Even if floats is a type name in an outer scope the equals
sign gives
// this away as a type_set declaration
typedef floats = <float, double, const long double&>;

// We have no typedef templates, so this can't be a problem.
template<typename T> typedef fwd = <const T&, T&&>;
```

This seems possible, but does *typedef* carry the right connotations? What if people starts talking of type_sets as typedefs, while listeners (of age) think that a type alias is being referred to?

A magical class template

Specifying a magical std::type_set template was considered:

```
using floats = std::type_set<float, double, const long double&>;
template<typename T> using fwd = std::type_set<const T&, T&&>;
```

Note that fwd inherits the magic of std::type_set to allow use in parameter declarations.

The in situ type lists would have to be written:

```
auto sin(std::type_set<float, double, const long double&> x)-
>std::remove_reference_t<decltype(x)>;
```

While ugly and misleading this is maybe acceptable as it will be seldom used.

The main drawback with this is that if a future direction is taken to allow named type sets in class scope these could end up being dependent, and while you can then disambiguate them using typename there is no way to tell the compiler that it is an instance of the magical `std::type_set` template unless a new keyword is introduced anyway. Example:

```
template<typename T> MyClass {
    using MyTypeSet = typename T::the_type_set;
    template<typename T> using MyTplTypeset = T:template the_tpl_typeset<T>;

    void method(MyTypeSet x);              // Compiler does not know if
MyTypeSet is a type or a type set.
    void otherMethod(MyTplTypeset int y);  // Compiler must guess that
MyTplTypeset is a type set template.
};
```

A magical variable template

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

A magical variable template works just about the same as a magical class template, but syntactically it is more like a typedef.

The main probem is the same, but maybe worse. As the dependent name is conceptually a value it does not need to be disambiguated, which makes a method declaration look like a method call.

Technical specification

No more formal specification than above has been started yet.

Future extensions (Not proposed)

There are more uses of type sets and type set templates that have not been fully explored, and are thus not proposed now. There are some other extensions to keep UTPs universal and allow all scopes, not proposed.

type sets for variable declarations

This proposal is limited to function parameters but it would be possible to extend to variable declarations, much in the same way that concepts can be used as a means of checking that function return types model the concept. Variables would then be declared to have one of a closed set of types. Which one is selected is determined by the conversion rules for the function return type.

If the type_set is templated `remove_cvref` of the function return type is substituted into the type_set and the best match is selected. As all variables have names it is possible to allow eliding the `auto` which makes it possible to declare let and mut as type_sets to provide convenient ways to declare variables:

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

```
float some_func();
float& ref_func();
const float& cref_func();

let a = some_func();      // auto elided
let auto b = some_func(); // Same as a
let int c = some_func();
let d = ref_func();
let int e = ref_func();  // Error? Or is e a const int by value?

mut f = some_func();
mut int g = some_func(); // int initiated from the float
mut h = ref_func();
mut i = cref_Func();     // Error? float by value?
mut int i = ref_func();  // Error? int by value?
```

The best rules for functions returning references and type_sets containing references are still not well understood, so this is not proposed at this point. The potential seems real though.

type sets for return types

What about return types? Do type sets have a place there too?

Using in-situ type sets of one to clear up declarations

Declaring references to arrays of function pointers is not trivial in C++. Using angle brackets as parentheses may clear up this a bit.

```
void f(<<<int(float, bool)>*>[10]>& x);

// Today we can do this:
using ftype = int(float, bool);
using pftype = ftype*;
using apftype = pftype[10];
void f(apftype& x);
```

To achieve this we must be more clever about where in the grammar the type-set-clause is introduced, or the nesting won't work.

Implicitly declared type set templates

What about allowing `func(<auto, const auto&> c);` that is, create a type set template on the spot.

Secondary type sets

It would be possible to allow creating type sets by binding some or all of the template parameters. This could prove useful but seems maybe not that useful as it should be as easy to start from scratch. A possible syntax would be:

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

typeset in_int = in<int>;
```

Chaining type sets

It seems possible to allow using multiple type sets in a chained fashion to separate concerns, for instance between the calling convention and a closed set of types. All type sets except the last must be templates. This can be expanded to a Cobol like syntax with single element type sets adding pointers, making arrays etc. At this point we don't think this adds enough value to be proposed, but here is an example anyway:

```
type_set widgets = <Canvas, Entry, Toolbar>;

void placeWidget(fwd widgets x);   // Two chained type sets, the first is a
template. 6 functions.

template<typename T> type_set pointer_to = <T*>;

void setInt(in pointer_to int p) { *p = 3; }

// Hello Cobol-land!
template<typename T, size_t N> type_set array_of = <T[N]>;
template<typename T, typename... Ps> type_set function_taking = <T(Ps...)>;
void f(mv array_of<10> function_taking<int, float> int x); // int(*(&&x)[10])
(int, float), give or take
```

At first blush there seems to be no big issues, except that the committee will be drowned in more or less exotic proposals of additions to the standard type sets.

Type sets as template parameters and in class- and block scopes

It should be possible to allow declaring type sets in class and block scope, and as template parameters. Allowing type sets in class scope opens up for dependent names that need to be disambiguated as type sets. If type sets is a kind that can be carried by UTPs it is hard to differentiate between disambiguation using a prefix type_set identifier and declaring a new type_set of the same name as the UTP. But if an equals sign follows I guess it must be a new type_set being declared as there is no syntax to just name an existing type_set followed by `=` (as variables are always named).

Acknowledgements

Thanks to my employer ContextVision AB for supporting the author attending standardization meetings.