

# Keep alias syntax extendable

Tomasz Kamiński

Principal Software Developer, Sabre

29 października 2017

# Problem statement

## Result of the P06340

Alias declaration syntax is restricted to type aliases only:

**using** *id* = *other-id*;

## Proposal

Revert P06340 changes to alias-declaration syntax, and thus unblock further extensions.

# Use case 1: Importing a function

## Goal

Make invocation of the function `Y::foo` equivalent to `X::foo`.

## Solution

```
namespace Y
{
    using X::foo ;
}
```

## Use case 2: Importing a function

### Goal

Make invocation of the function `Y::bar` equivalent to `X::foo`.

### Solution 1

```
namespace Y
{
    template<typename... Args>
    decltype(auto) bar(Args&&... args)
    { return X::foo(std::forward<Args>(args)...); }
}
```

# Use case: Importing a function

## Goal

Make invocation of the function `Y::bar` equivalent to `X::foo`.

## Solution 2

```
namespace Y
{
    template<typename... Args>
    auto bar(Args&&... args)
        noexcept(X::foo(std::forward<Args>(args)...) )
    -> decltype(X::foo(std::forward<Args>(args)...) )
    { return X::foo(std::forward<Args>(args)...) ; }
}
```

# Use case: Importing a function

## Goal

Make invocation of the function `Y::bar` equivalent to `X::foo`.

## Solution 2: Using P0573R2

```
namespace Y
{
    template<typename... Args>
    auto bar(Args&&... args)
    => decltype(X::foo(std::forward<Args>(args)...))
}
```

# Use case: Importing a constrained function

## Goal

Make invocation of the function `Y::bar` equivalent to `X::foo`.

## Declaration

```
namespace X
{
    template<C1 T1, C2 T1>
    void foo(T1 t1, T2 t2);
}
```

## Solution 3

```
namespace Y
{
    template<C1 T1, C2 T1>
    void bar(T1 t1, T2 t2)
    { return X::foo(std::move(t1), std::move(t2)); }
}
```

# Use case: Non-movable argument

## Goal

Make invocation of the function `Y::bar` equivalent to `X::foo`.

## Declaration

```
namespace X
{
    void foo(NonMovable nv);
}
```

## Solution 4

```
namespace Y
{
    void bar(NonMovable nv)
    { return X::foo(std::move(nv)); }
    // compilation error
}
```



# Use case: Example extension

## Goal

Make invocation of the function `Y::bar` equivalent to `X::foo`.

## Extension

```
namespace Y
{
    using bar = X::foo;
}
```

# Discussion

## Statement

Programmer should be able to differentiate what kind of entity is imported - syntax should be different for templates, functions, variables and types.

## Counterargument

Syntax of importing should not differ if `X::foo` is:

- ▶ normal function
- ▶ function template
- ▶ set of overloaded functions
- ▶ set of overloaded function templates
- ▶ set of overloaded functions and function templates
- ▶ global functor variable

# Discussion

## Statement

Programmer should be able to differentiate what kind of entity is imported - syntax should be different for templates, functions, variables and types.

## Counterargument

**typename** keyword already makes intent clear:

**using** *type* = **typename** *other-type*;

# Discussion

## Statement

Programmer should be able to differentiate what kind of entity is imported - syntax should be different for templates, functions, variables and types.

### Proposal: Expression alias

```
namespace Y
{
    using bar = (X::foo);
}
```

## Counterargument

Does not disambiguate between function, templates and variables.  
Parens required to alias anything other than type.