

Doc No.: D2288R0 DRAFT 1

Project: Programming Language - C++ (WG21)

Author: Andrew Tomazos <[andrewtomazos@gmail.com](mailto:andrewtomazos@gmail.com)>

Audience: EWG

Target: C++23

Date: 2021-01-21

## Design Notes for C++23 Named Parameters

We present a partially complete and approximate, but concrete suggestion for the syntax and semantics of a named parameters feature for C++23. It is intended only at this stage to provoke thought and discussion.

### Arguments

In a function call expression or initializer list, each argument shall be either a *positional argument* or a *labelled argument*. [Rationale: Not aware of any alternatives, end Rationale]

A labelled argument can use one of two syntaxes that have identical semantics:

```
f(3) // a positional argument

f(x: 3) // a labelled argument

f(.x = 3) // also a labelled argument,
          // equivalent semantics to previous
```

[Rationale: The `.x=3` syntax is useful for uniformity and backward-compatibility with existing code that uses C / C++20 designated initializer lists, allowing smooth migration from an old aggregate class type to a new version that is a non-aggregate class type with a constructor that has labelled parameters.

For example after migrating:

```
struct Point { int x, y; };
```

to

```
struct Point { Point(int. x, int. y); /*...*/ };
```

the following program continues to work unmodified:

```
int main() {
    Point p1{2, 3};
}
```

```
Point p2{.x = 2, .y = 3};  
}
```

The ``x: 3`` labelled argument syntax is uniform with labelled statements and is the more natural, terser and preferred syntax, and doesn't inappropriately appeal to the member access operator as ``x=3`` does.

A syntax of ``x=3`` is not possible due to ambiguity with assignment statement.

This is a similar design decision to the introduction of the largely equivalent keyword ``class`` in C++ despite the existence of the keyword ``struct`` in C.

end Rationale]

## Parameters

A function type has:

1. Zero or more positional parameters
2. Zero or more label-allowed parameters
3. Zero or more label-required parameters

A positional parameter has no label declarator.

A label-allowed parameter has a ``.`` label declarator.

A label-required parameter has a ``:`` label declarator.

[Example:

```
void f(  
    int  a, // positional parameter  
    int. b, // label-allowed parameter  
    int: c // label-required parameter  
);
```

end Example]

Collectively positional parameters and label-allowed parameters are called *position-sensitive parameters*.

Collectively label-allowed parameters and label-required parameters are called *labelled parameters*.

A positional argument may only match to a position-sensitive parameter.

A labelled argument may only match to a labelled parameter.

In the declaration of a function type, position-sensitive parameters must precede label-required parameters parameters:

[Example:

```
void f(int a, int. b, int c, int :d, int :e); // OK

void f(int a, int. b, int c, int :d, int .e); // ERROR:
// The position-sensitive parameter e must precede
// the label-required parameter d
```

end example]

[Rationale: As label-required arguments do not participate in positional argument matching, it would be confusing and error-prone to have them interspersed with position-sensitive parameters]

All labelled parameters must have parameter names, and they must be unique within the enclosing function type.

## Function Types

For the purposes of determining if two function types are the same, the label-required parameters of a function-type are sorted in lexicographical order of their parameter name (that is, the label-required parameters are not order-sensitive). After this transformation, two function types are the same using the usual C++20 rules with the addition that each parameter must match in kind (positional, label-allowed or label-required) and their labelled parameters must match in parameter name.

[Example:

The following function types are all the SAME:

```
void(int, int);
void(int, int b);
void(int a, int b);
void(int x, int y);
```

The following function types are DIFFERENT:

```
void(int: a, int: b);  
void(int: x, int: y);
```

The following function types are the SAME:

```
void(int: a, float: b);  
void(float: b, int: a);
```

The following function types are DIFFERENT:

```
void(int. a, float. b);  
void(float. b, int. a);
```

The following function types are all DIFFERENT:

```
void(int a, int b);  
void(int a, int. b);  
void(int. a, int b);  
void(int. a, int. b);  
void(int a, int: b);  
void(int. a, int: b);  
void(int: a, int: b);
```

end Examples]

An object of function pointer/reference type F is convertible to another function pointer/reference of type G if they are the same type except for G having some positional parameters that differ only in kind (but not type) with some label-allowed parameters of F.

[Example:

```
std::function<void(int. widgets)> f = /*...*/;  
std::function<void(int)> g = f; // OK
```

whereas:

```
std::function<void(int: widgets)> f = /*...*/;  
std::function<void(int)> g = f; // ERROR
```

end Example]

## Forwarding

It should be possible to forward a mixed set of positional and labelled arguments in a generic fashion.

[Example:

```
class WidgetFactory {
public:
    WidgetFactory(
        std::string. name,
        int: height,
        int: width,
        WidgetLayout: layout
    );
}

int main() {
    auto factory = std::make_unique<WidgetFactory>(
        "MyWidgetFactory",
        height: 420,
        width: 380,
        layout: WidgetLayout::CoolBananas
    );
}
```

end Example]

## Overriding

In the same fashion as function types are convertible between label-allowed parameters to positional parameters, it should be possible to override along that same conversion:

[Example:

```
struct Base {
    virtual void f(int) = 0;
};

struct Derived1 {
    void f(int. x) override; // OK
};

struct Derived2 {
    void f(int: x) override; // ERROR: does not override
```

```
};
```

## Default Arguments

The rules regarding default argument order do not apply to any label-required parameters. Each label-required parameter may or may not have a default argument:

[Example:

```
void f(  
    int  x1,  
    int. x2,  
    int. x3,  
    int. X4 = 40,  
    int  x5 = 50,  
    int: x5,          // <---OK NO DEFAULT  
    int: x6 = 70,  
    int: x8           // <---OK NO DEFAULT  
);
```

end Example]

## Overload Resolution

As usual, the rules about when two function types are not the same are used for the purposes of forming a function overload set:

```
void f(int  a, int. b) { /*1*/ }  
void f(int. a, int  b) { /*2*/ }  
void f(int. a, int. b) { /*3*/ }  
void f(int  a, int: b) { /*4*/ }  
void f(int. a, int: b) { /*5*/ }  
void f(int: a, int: b) { /*6*/ }  
void f(int: x, int: y) { /*7*/ }  
void f(int: a, int: y) { /*8*/ }
```

Based on the arguments to a function call expression `f(/\*...\*/)` function overload resolution proceeds roughly as follows:

1. First, functions from the overload set that are not viable are eliminated.
2. If no viable functions remain, it is an error
3. Remaining viable functions are compared according to a ranking algorithm
4. If there is no clear one best match, then it is an error (ambiguous)

5. The best match is returned

This is specified more precisely in **Clause 16 Overloading** of the C++ spec.

Adjusting this algorithm for named parameters thus involves:

Specifying when a given argument list is viable for a given parameter list, where either the argument list has a labelled argument or the parameter list has a labelled parameter (or both).

Then specifying how the viable function ranking algorithm is affected by the presence of labelled arguments or labelled parameters (or both).

We're going to hand-wave this for now, short of saying that as previously specified:

1. A positional argument is only a viable match to a position-sensitive parameter
2. A labelled argument is only a viable match to a labelled parameter