Appendix

C++17 Improvements Made Retroactive to C++11/14

The subsections that follow describe the subtle bugs that came with the previous specification, both for completeness and to give a better understanding of what to expect on very old compilers, though none fully implemented the original specification as written.

Inheriting constructors declared with a C-style ellipsis Forwarding arguments from a constructor declared using a C-style ellipsis cannot be performed correctly. Arguments passed through the ellipsis are not available as named arguments but must instead be accessed through the `va_arg` family of macros. Without named arguments, no easily supported way is available to call the base-class constructor with the additional arguments:

```
struct Base
{
    Base(int x, ...) { }  // constructor taking C-style variadic args
};

struct Derived : Base
{
    using Base::Base;  // Error: Prior to C++17 fixes, standard wording
                       // does not allow forwarding C-style variadic args.
};
```

This problem is sidestepped in C++17 because the base-class constructor becomes available just like any other base-class function made available through a **`using`** declaration in the derived class.

Inheriting constructors that rely on friendship to declare function parameters When a constructor depends on access to a **`private`** member of a class (e.g., a **`typedef`**), an inheriting constructor does not implicitly grant friendship that the base class might have that makes the constructor valid. For example, consider the following class template, which grants friendship to class `B`:

```
template <typename T>
struct S
{
private:
    typedef int X;
    friend struct B;
};
```

Then, we can create a class with a constructor that relies on that friendship. In this case, we consider a constructor template using the dependent member `X`, assuming that, in the normal case, `X` would be publicly accessible:

```
struct B
{
    template <typename T>
    B(T, typename T::X);
};
```

Now consider class `D` derived from `B` and inheriting its constructors:

```
struct D : B
{
    using B::B;
};
```

Without friendship, we cannot construct a `D` from an `S`, but we can construct a `B` from an `S`, suggesting something is wrong with the inheritance. Note that the SFINAE rules for templates mean that the inheriting constructor is a problem only if we try to construct an `S` with the problem type and does not cause a hard error without that use case. The following example illustrates the problematic usage:

```
S<int> s;   // full specialization of S for type int
B b(s, 2);  // OK, thanks to friendship
D d(s, 2);  // Error: Prior to C++17 fixes, friendship is not inherited.
```

As C++17 redefines the semantics of the inheriting constructor as if the base class's constructors were merely exposed in the derived one, friendship is evaluated within the scope of the base class.

Inheriting constructor templates would be ill formed for a local class Local classes have many restrictions, one of which is that they cannot declare member templates. If we inherit constructors from a base class with constructor templates, even **`private`** ones, the implicit declaration of a constructor template to forward arguments to the base-class constructor would be **ill formed**:

```
struct Base
{
    template <typename T>
    Base(T);
};

void f()
{
    class Local : Base
    {
        using Base::Base;  // Error: Prior to C++17 fixes, we cannot redeclare
                           // the constructor template in local class.
    };
}
```

C++17 resolves this by directly exposing the base-class constructors, rather than defining new constructors to forward arguments.

SFINAE evaluation context with default function arguments Constructors that employ SFINAE tricks in default function arguments perform SFINAE checks in the wrong context and therefore inherit ill-formed constructors. No such issues occur when these SFINAE tricks are performed on default template arguments instead. As an example, consider a class template `Wrap` that has a template constructor with a SFINAE constraint:

```
#include <iostream>    // std::cout
#include <type_traits> // std::enable_if, std::is_constructible

struct S { };

template <typename T>
struct Wrap
{
    template <typename U>
    Wrap(U, typename std::enable_if<
        std::is_constructible<T, U>::value>::type* = nullptr)
        // This constructor is enabled only if T is constructible from U.
    {
        std::cout << "SFINAE ctor\n";
    }

    Wrap(S)
    {
        std::cout << "S ctor\n";
    }
};
```

If we derive from `Wrap` and inherit its constructors, we would expect the SFINAE constraint to behave exactly as in the base class, i.e., the template constructor overload would be silently discarded if `std::is_constructible<T, U>::value` evaluates to **`false`**:

```
template <typename T>
struct Derived : Wrap<T>
{
    using Wrap<T>::Wrap;
};
```

However, prior to C++17's retroactive fixes, SFINAE was triggered only for `Wrap`, not for `Derived`:

```
void f()
{
    S s;
    Wrap<int> w(s);     // prints "S ctor"
    Derived<int> d(s);  // error prior to fixes; prints "S ctor" afterward
}
```

Suppression of constructors in the presence of default arguments A constructor having one or more default arguments in the derived class does not suppress any corresponding constructors matching only the nondefaulted arguments in the base class, leading to ambiguities:

```
#include <iostream>  // std::cout

struct B          // base class
{
    B(int, int);  // value constructor with two (required) int parameters
};

struct D : B
{
    using B::B;
    D(int, int, int = 0);  // doesn't suppress D(int, int) from B(int, int)
};
```

In the code example above, the original defective behavior was that there would be two overloaded constructors in `D`; attempting to construct a `D` from two integers became ambiguous. In the corrected behavior, the inheriting `D(int, int)` from the base-class constructor `B(int, int)`, whose domain is fully subsumed by the derived class's explicitly specified constructor `D(int, int, int = 0)`, is suppressed.

Suprising behavior with unary constructor templates Because inherited constructors are redeclarations within the derived class and expect to forward properly to the corresponding base-class constructors, constructor templates may do very surprising things. In particular, a gregarious, templated constructor can appear to cause inheritance of a base-class copy constructor. Consider the following class with a constructor template:

```
struct A
{
    A() = default;
    A(const A&) { std::cout << "copy\n"; }

    template <typename T>
    A(T) { std::cout << "convert\n"; }
};
```

This simple class can convert from any type and prints those of its constructors that were called. Now consider we want to make a **strong typedef** for `A`:

```
struct B : A
{
    using A::A;  // inherited base class A's constructors
};
```

The problem is that because `A` can convert from anything, when `B` inherits `A`'s constructor template, `B` can then use the inherited constructor to construct an instance of `B` from `A`. Perhaps more surprising, because the definition of the inherited constructor in `B` is to initialize the `A` subobject with its parameters, the nontemplate inherited constructor will be chosen as the best match, not the templated, converting constructor![1]

[1]Note that if the template constructor for `A` were a *copy* or *move* constructor for `A`, then it would be excluded from being an inherited constructor and this odd behavior would be avoided. The by-value parameter of this constructor is also why `"copy"` is output twice in this example.

```
A x;
B y = x;  // Surprise! This compiles, and it prints "copy" twice!
```