Variadic Friends

Contents

1 Introduction 1

2 Motivation 2

2.1 The Passkey Idiom 2

2.2 CRTP Access to Derived 3

3 Other Implications 4

4 Wording 4

5 Acknowledgements 5

1 Introduction

This paper proposes support for granting friendship to all classes in a parameter pack. Several existing idioms are implemented by providing friendship to a class via template parameter. However, these patterns can only be used with a single template parameter, because friendship cannot be currently granted to a pack of types.

The following code snippet provides a general summary of the current issue.

```cpp
template <typename T>
class Foo1
{
    friend T; // Supported
public:
    // ...
};

template <typename... Ts>
class Foo2
{
    friend Ts...; // ERROR
public:
    // ...
};
```
2 Motivation

This section documents two common idioms, and how they could be expanded with support for pack expansion in a friend declaration.

2.1 The Passkey Idiom

Granting general friendship has wide implications. In the following example, Blarg has been granted friendship by Foo.

```cpp
class Blarg;
class Foo
{
 friend Blarg;
 // ...
};
```

Thus, Blarg can access all private members of Foo, even if friendship were only needed in order to access a single non-public member. Wide friendship has a number of issues, and is not generally desired.

With the Passkey idiom, limited access can be granted on a per-member function basis. In the following example, the Foo::do_something member function is public, but it must be given an instance of Passkey as a function argument.

```cpp
class Foo
{
 public:
 // Only callable from Blarg
 void do_something(Passkey);
};
```

If Passkey has a non-public constructor, but grants friendship to Blarg, then Blarg can call Foo::do_something by passing an instance of Passkey.

```cpp
class Passkey
{
 friend Blarg;
 Passkey() = default;
};
```

Furthermore, the only entities that can call Foo::do_something are those who have been granted friendship with Passkey, since friendship with Passkey is required in order to create an instance of Passkey.

This technique can be generalized with the following idiom.

```cpp
template <typename T>
class Passkey
{
 friend T;
 Passkey() = default;
};
class Foo
{
 public:
 // Only callable from Blarg
```
void do_something(Passkey<Blarg>);
};

The above code has the same effect, where Foo::do_something can only be called if the caller can create an instance of Passkey<Blarg>.

We would like to expand this idiom, and grant access to Foo::do_something by more than one class.

template<typename... Ts>
class Passkey
{
    friend Ts...;
    Passkey() = default;
};

class Foo
{
public:
    // Only callable from Blarg, Blip, and Baz
    void do_something(Passkey<Blarg, Blip, Baz>);
};

However, C++ does not allow the variadic friend declarations, so the above code will not compile.

## 2.2 CRTP Access to Derived

Another common pattern is to inherit from some class template, passing the type of the derived class as a template parameter to the base class.

There may be parts of the derived class API which are needed in the base class, but only the base class, so they are private, and friendship is granted to the base class.

template<typename DispatcherT, typename MsgT>
class Receiver
{
    // Can static_cast to DispatcherT and access private stuff
};

template<typename MsgT>
class Dispatcher :
    public Receiver<Dispatcher<MsgT>, MsgT>
{
    friend Receiver<Dispatcher, MsgT>;

    // stuff that we want Receiver to access
    // ... 

    public:
        using Receiver<Dispatcher, MsgT>::Receiver;
};
If we want to support multiple base classes, we would like to write the following.

```cpp
template <typename DispatcherT, typename MsgT>
class Receiver
{
};

template <typename... MsgTs>
class Dispatcher
: public Receiver<Dispatcher<MsgTs...>, MsgTs>... // OK
{
    friend Receiver<Dispatcher, MsgTs>...; // ERROR

    // stuff that we want any Receiver to access
    // ...

public:
    using Receiver<Dispatcher, MsgTs>::Receiver...; // OK
};
```

Unfortunately, this results in a compiler error, because pack expansion is not supported for friend declarations. Note, however, that both inheritance and using support pack expansion.

### 3 Other Implications

The following would be implicitly supported as a result of supporting pack expansion for friend declarations.

```cpp
class Phoebe
{
    friend Rachel, Monica, Ross, Chandler, Joey;
};

template <typename T, typename U>
class Foo
{
    friend T, U;
};

template <typename... Ts, typename... Us>
class Bar<mp_list<Ts...>, mp_list<Us...>>
{
    friend Ts..., Us...;
};
```

### 4 Wording

These changes are relative to C++ draft n4944.

Extend the grammar for a friend declaration in 11.8.4 [class.friend]/3 and extend the example:

3 A friend declaration that does not declare a function shall have one of the following forms:
friend typename-specifier-list:
+ typename-specifier ...opt
+ friend typename-specifier-list, typename-specifier ...opt

friend elaborated-type-specifier;
friend simple-type-specifier;
- friend typename-specifier;
+ friend typename-specifier-list;

And extend the example following that paragraph:

class C;
typedef C Ct;

class X1 {
    friend C; // OK, class C is a friend
};

class X2 {
    friend Ct; // OK, class C is a friend
    friend D; // error: D not found
    friend class D; // OK, elaborated-type-specifier declares new class
};
template <typename T> class R {
    friend T;
};
+ template <typename... Ts> class V {
    + friend Ts...;
    +
};
R<C> rc; // class C is a friend of R<C>
R<int> Ri; // OK, "friend int;" is ignored
+ V<C, X1> vc; // both C and X1 are friends of V<C, X1>

Insert a new stipulation in 13.7.4 [temp.variadic]/5:

5 A pack expansion consists of a pattern and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

5 Acknowledgements

I had been sitting on this since posting to the mailing list about it in January 2020. This paper was finally written during C++Now 2023, Feature in a Week. Thus, it is very appropriate to say that it would not exist without that program. Thanks to Jeff Garland, Marshall Clow, Barry Revzin, and JF Bastien for running that program, and all the attendees who helped discuss and review the proposal. Special thanks to Barry Revzin for lots of help with the document itself.