Base_From_Member

Daryle Walker

Table of Contents

Rationale

When developing a class, sometimes a base class needs to be initialized with a member of the current class. As a naïve example:

```
#include <streambuf>  /* for std::streambuf */
#include <ostream>    /* for std::ostream */

class fdoutbuf
  : public std::streambuf
{
public:
    explicit fdoutbuf( int fd );
    //...
};

class fdostream
  : public std::ostream
{
protected:
    fdoutbuf buf;
public:
    explicit fdostream( int fd )
      : buf( fd ), std::ostream( &buf ) {}
    //...
};
```

This is undefined because C++'s initialization order mandates that the base class is initialized before the member it uses. R. Samuel Klatchko developed a way around this by using the initialization order in his favor. Base classes are intialized in order of declaration, so moving the desired member to another base class, that is initialized before the desired base class, can ensure proper initialization.

A custom base class can be made for this idiom:

```
#include <streambuf>  /* for std::streambuf */
#include <ostream>    /* for std::ostream */

class fdoutbuf
  : public std::streambuf
{
public:
    explicit fdoutbuf( int fd );
    //...
};

struct fdostream_pbase
{
    fdoutbuf sbuffer;

    explicit fdostream_pbase( int fd )
      : sbuffer( fd ) {}
};

class fdostream
  : private fdostream_pbase
  , public std::ostream
{
    typedef fdostream_pbase  pbase_type;
    typedef std::ostream     base_type;

public:
    explicit fdostream( int fd )
      : pbase_type( fd ), base_type( &sbuffer ) {}
    //...
};
```

Other projects can use similar custom base classes. The technique is basic enough to make a template, with a sample template class in this library. The main template parameter is the type of the enclosed member. The template class has several (explicit) constructor member templates, which implicitly type the constructor arguments and pass them to the member. The template class uses implicit copy construction and assignment, cancelling them if the enclosed member is non-copyable.

Manually coding a base class may be better if the construction and/or copying needs are too complex for the supplied template class, or if the compiler is not advanced enough to use it.

Since base classes are unnamed, a class cannot have multiple (direct) base classes of the same type. The supplied template class has an extra template parameter, an integer, that exists solely to provide type differentiation. This parameter has a default value so a single use of a particular member type does not need to concern itself with the integer.

Synopsis

```
#include <type_traits>  /* exposition only */

#ifndef BOOST_BASE_FROM_MEMBER_MAX_ARITY
#define BOOST_BASE_FROM_MEMBER_MAX_ARITY  10
#endif

template < typename MemberType, int UniqueID = 0 >
class boost::base_from_member
{
protected:
    MemberType  member;

#if C++11 is in use
    template< typename ...T >
    explicit constexpr   base_from_member( T&& ...x )
     noexcept( std::is_nothrow_constructible<MemberType, T...>::value );
#else
    base_from_member();

    template< typename T1 >
    explicit  base_from_member( T1 x1 );

    template< typename T1, typename T2 >
    base_from_member( T1 x1, T2 x2 );

    //...

    template< typename T1, typename T2, typename T3, typename T4,
     typename T5, typename T6, typename T7, typename T8, typename T9,
     typename T10 >
    base_from_member( T1 x1, T2 x2, T3 x3, T4 x4, T5 x5, T6 x6, T7 x7,
     T8 x8, T9 x9, T10 x10 );
#endif
};

template < typename MemberType, int UniqueID >
class base_from_member<MemberType&, UniqueID>
{
protected:
    MemberType& member;

    explicit constexpr base_from_member( MemberType& x )
        noexcept;
};
```

The class template has a first template parameter `MemberType` representing the type of the based-member. It has a last template parameter `UniqueID`, that is an `int`, to differentiate between multiple base classes that use the same based-member type. The last template parameter has a default value of zero if it is omitted. The class template has a protected data member called `member` that the derived class can use for later base classes (or itself).

If the appropriate features of C++11 are present, there will be a single constructor template. It implements *perfect forwarding* to the best constructor call of `member` (if any). The constructor template is marked both `constexpr` and `explicit`. The former will be ignored if the corresponding inner constructor call (of `member`) does not have the marker. The latter binds the other way; always taking effect, even when the inner constructor call does not have the marker. The constructor template propagates the `noexcept` status of the inner constructor call. (The constructor template has a trailing parameter with a default value that disables the template when its signature is too close to the signatures of the automatically-defined non-template copy- and/or move-constructors of `base_from_member`.)

On earlier-standard compilers, there is a default constructor and several constructor member templates. These constructor templates can take as many arguments (currently up to ten) as possible and pass them to a constructor of the data member.

A specialization for member references offers a single constructor taking a `MemberType&`, which is the only way to initialize a reference.

Since C++ does not allow any way to explicitly state the template parameters of a templated constructor, make sure that the arguments are already close as possible to the actual type used in the data member's desired constructor. Explicit conversions may be necessary.

The `BOOST_BASE_FROM_MEMBER_MAX_ARITY` macro constant specifies the maximum argument length for the constructor templates. The constant may be overridden if more (or less) argument configurations are needed. The constant may be read for code that is expandable like the class template and needs to maintain the same maximum size. (Example code would be a class that uses this class template as a base class for a member with a flexible set of constructors.) This constant is ignored when C++11 features are present.

Usage

With the starting example, the `fdoutbuf` sub-object needs to be encapsulated in a base class that is inheirited before `std::ostream`.

```
#include <boost/utility/base_from_member.hpp>

#include <streambuf>  // for std::streambuf
#include <ostream>    // for std::ostream

class fdoutbuf
  : public std::streambuf
{
public:
    explicit fdoutbuf( int fd );
    //...
};

class fdostream
  : private boost::base_from_member<fdoutbuf>
  , public std::ostream
{
    // Helper typedef's
    typedef boost::base_from_member<fdoutbuf>  pbase_type;
    typedef std::ostream                        base_type;

public:
    explicit fdostream( int fd )
      : pbase_type( fd ), base_type( &member ){}
    //...
};
```

The base-from-member idiom is an implementation detail, so it should not be visible to the clients (or any derived classes) of `fdostream`. Due to the initialization order, the `fdoutbuf` sub-object will get initialized before the `std::ostream` sub-object does, making the former sub-object safe to use in the latter sub-object's construction. Since the `fdoutbuf` sub-object of the final type is the only sub-object with the name `member` that name can be used unqualified within the final class.

Example

The base-from-member class templates should commonly involve only one base-from-member sub-object, usually for attaching a stream-buffer to an I/O stream. The next example demonstrates how to use multiple base-from-member sub-objects and the resulting qualification issues.

```
#include <boost/utility/base_from_member.hpp>

#include <cstddef>  /* for NULL */

struct an_int
{
    int  y;

    an_int( float yf );
};

class switcher
{
public:
    switcher();
    switcher( double, int * );
    //...
};

class flow_regulator
{
public:
    flow_regulator( switcher &, switcher & );
    //...
};

template < unsigned Size >
class fan
{
public:
    explicit fan( switcher );
    //...
};

class system
  : private boost::base_from_member<an_int>
  , private boost::base_from_member<switcher>
  , private boost::base_from_member<switcher, 1>
  , private boost::base_from_member<switcher, 2>
  , protected flow_regulator
  , public fan<6>
{
    // Helper typedef's
    typedef boost::base_from_member<an_int>       pbase0_type;
    typedef boost::base_from_member<switcher>     pbase1_type;
    typedef boost::base_from_member<switcher, 1>  pbase2_type;
    typedef boost::base_from_member<switcher, 2>  pbase3_type;

    typedef flow_regulator  base1_type;
    typedef fan<6>          base2_type;

public:
    system( double x );
    //...
};
```

```
system::system( double x )
  : pbase0_type( 0.2 )
  , pbase1_type()
  , pbase2_type( -16, &this->pbase0_type::member.y )
  , pbase3_type( x, static_cast<int *>(NULL) )
  , base1_type( pbase3_type::member, pbase1_type::member )
  , base2_type( pbase2_type::member )
{
    //...
}
```

The final class has multiple sub-objects with the name `member`, so any use of that name needs qualification by a name of the appropriate base type. (Using `typedef`s ease mentioning the base types.) However, the fix introduces a new problem when a pointer is needed. Using the address operator with a sub-object qualified with its class's name results in a pointer-to-member (here, having a type of `an_int boost::base_from_member< an_int, 0> :: *`) instead of a pointer to the member (having a type of `an_int *`). The new problem is fixed by qualifying the sub-object with `this->` and is needed just for pointers, and not for references or values.

There are some argument conversions in the initialization. The constructor argument for `pbase0_type` is converted from `double` to `float`. The first constructor argument for `pbase2_type` is converted from `int` to `double`. The second constructor argument for `pbase3_type` is a special case of necessary conversion; all forms of the null-pointer literal in C++ (except `nullptr` from C++11) also look like compile-time integral expressions, so C++ always interprets such code as an integer when it has overloads that can take either an integer or a pointer. The last conversion is necessary for the compiler to call a constructor form with the exact pointer type used in `switcher`'s constructor. (If C++11's `nullptr` is used, it still needs a conversion if multiple pointer types can be accepted in a constructor call but `std::nullptr_t` cannot.)

Acknowledgments

- Ed Brey suggested some interface changes.
- R. Samuel Klatchko (rsk@moocat.org, rsk@brightmail.com) invented the idiom of how to use a class member for initializing a base class.
- Dietmar Kuehl popularized the base-from-member idiom in his IOStream example classes.
- Jonathan Turkanis supplied an implementation of generating the constructor templates that can be controlled and automated with macros. The implementation uses the Preprocessor library.
- Walker started the library. Contributed the test file base_from_member_test.cpp.