Accessing Private Members

…the Right Way

Private Members in C++

```
// something.hpp
class something
{
private:
    int variable;
    void function();
    using type = int;
};
```

Private Members in C++

```
#include “something.hpp”

int main()
{
    something s;
    s.variable = 5;  // error
    s.function();  // error
    using T = something::type;  // error
}
```

Why would you even want to access private members?

Motivation

```
// this pattern occurs in https://github.com/taocpp/taopq
std::string read( const std::size_t max )
{
   std::string buffer;
   buffer.resize( max );
   const std::size_t delivered = read_from_database( buffer.data(), max );
   buffer.resize( delivered );
   return buffer;
}
```

Motivation

```
// this pattern occurs in https://github.com/taocpp/taopq
std::string read( const std::size_t max )
{
   std::string buffer;
   buffer.resize( max );  // slow
   const std::size_t delivered = read_from_database( buffer.data(), max );
   buffer.resize( delivered );
   return buffer;
}

// often ‘max’ is large (e.g. read up to 1MB), while ‘delivered’ is often small,
// leading to the first ‘resize’ to be slow, as the string will always be
// initialised, i.e. filled with ‘\0’.
```

Motivation

```
// this pattern occurs in https://github.com/taocpp/taopq
std::string read( const std::size_t max )
{
   std::string buffer;
   buffer.resize( max );  // slow
   const std::size_t delivered = read_from_database( buffer.data(), max );
   buffer.resize( delivered );
   return buffer;
}

// often ‘max’ is large (e.g. read up to 1MB), while ‘delivered’ is often small,
// leading to the first ‘resize’ to be slow, as the string will always be
// initialised, i.e. filled with ‘\0’.

// if only there were a way to resize a std::string without initialising its
// buffer (which in this example will be overwritten or discarded anyways!)
```

Motivation

```
// this pattern occurs in https://github.com/taocpp/taopq
std::string read( const std::size_t max )
{
   std::string buffer;
   buffer.resize( max );  // slow
   const std::size_t delivered = read_from_database( buffer.data(), max );
   buffer.resize( delivered );
   return buffer;
}

// luckily, all major standard libraries have
// (private) member functions to achieve this!
```

The Five Candidates

- The Pickpocket
- The Cheat
- The Liar
- The Robber
- The Master Thief

The Pickpocket

http://www.gotw.ca/gotw/076.htm

The Pickpocket

```
#define class struct
#define private public
#define protected public
#include “something.hpp”

int main()
{
    something s;
    s.variable = 5;  // OK
    s.function();  // OK
    using T = something::type;  // OK
}
```

The Pickpocket

- Applies to all classes/structs
- Applies via nested includes, including standard headers/types
- Redefines a reserved word, not allowed by the standard
- ODR violation, as the same class now has two different definitions
- May change the layout of classes
- May cause actual ODR issues when not applied consistently

The Cheat

http://www.gotw.ca/gotw/076.htm

The Cheat

```
#include “something.hpp”

struct something_compatible  // hope for layout “compatibility”
{
   int variable;
};

int main()
{
   something s;
   reinterpret_cast< something_compatible& >( s ).variable = 5;  // OK?
}
```

The Cheat

- Only “works” for member variables
- Invokes undefined behaviour (`reinterpret_cast`)
- Dangerous when `something` changes

The Liar

http://www.gotw.ca/gotw/076.htm

The Liar

```
// instead of: #include “something.hpp”
class something;
void hijack( something& );

// manually duplicate something’s definition…
class something
{
   // …and sneakily add a new friend
   friend void hijack( something& );
};

void hijack( something& s )
{
   s.variable = 5;  // OK
   s.function();  // OK
   using T = something::type;  // OK
}
```

The Liar

- ODR violation, as the same class now has two different definitions
- Dangerous when `something` changes

Two Actual Solutions

- “The Robber”
  - http://bloglitb.blogspot.com/2010/07/access-to-private-members-thats-easy.html
- “The Master Thief”
  - https://github.com/facebook/folly/blob/master/folly/memory/UninitializedMemoryHacks.h
  - IMHO more direct and scaleable, but hard to understand from the link

Top-Level Wrapper

```
// generic top-level wrapper
template< typename T >
void resize_uninitialized( std::basic_string<T>& v, const std::size_t n )
{
   if( n <= v.size() )
      v.resize( n );
   else {
      if( n > v.capacity() )
         v.reserve( n );
      resize_uninitialized_impl( v, n );
   }
}
```

The Robber

http://bloglitb.blogspot.com/2010/07/access-to-private-members-thats-easy.html

The Robber

Disclaimer:

- Don’t worry if you don’t understand all the details
- I deliberately skip over “The Robber” quickly
- It is more complicated and doesn’t scale as well as “The Master Thief”, which will be explained afterwards

The Robber

```
// generic helper I
template< typename Tag >
struct result
{
   using type = typename Tag::type;
   static type ptr;
};

template< typename Tag >
typename result< Tag >::type result< Tag >::ptr;
```

The Robber

```
// generic helper II
template< typename Tag, typename Tag::type Ptr >
struct rob
{
   struct filler_t
   {
      filler_t()
      {
         result< Tag >::ptr = Ptr;
      }
   };
   static filler_t filler;
};

template< typename Tag, typename Tag::type Ptr >
typename rob< Tag, Ptr >::filler_t rob< Tag, Ptr >::filler;
```

…for (MSVC)

```
// tag class for a member functions with signature void (T::)( std::size_t )
template< typename T >
struct tag
{
   using type = void (T::*)( std::size_t );
};

template< typename T >
void resize_uninitialized_impl( std::basic_string<T>& v, const std::size_t n )
{
   // v._Eos( n );
   (v.*result< tag< std::basic_string<T> > >::ptr)( n );
}

// explicit instantiation bypasses access checks!
template struct rob< tag< std::string >, &std::string::_Eos >;
template struct rob< tag< std::basic_string<…> >, &std::basic_string<…>::_Eos >;
```

…for (libc++)

```
template< typename T >
struct tag
{
   using type = void (T::*)( std::size_t );
};

template< typename T >
void resize_uninitialized_impl( std::basic_string<T>& v, const std::size_t n )
{
   // v.__set_size( n );
   (v.*result< tag< std::basic_string<T> > >::ptr)( n );
   v[ v.size() ] = typename T::value_type( 0 );
}

template struct rob< tag< std::string >, &std::string::__set_size >;
// …
```

…for (libstdc++, C++11 ABI)

```
template< typename T >
struct tag
{
   using type = void (T::*)( std::size_t );
};

template< typename T >
void resize_uninitialized_impl( std::basic_string<T>& v, const std::size_t n )
{
   // v._M_set_length( n );
   (v.*result< tag< std::basic_string<T> > >::ptr)( n );
}

template struct rob< tag< std::string >, &std::string::_M_set_length >;
// …
```

…for (libstdc++, old ABI)

```
template< typename T >
struct tag
{
   using type = void (T::*)( std::size_t );
};

template< typename T >
void resize_uninitialized_impl( std::basic_string<T>& v, const std::size_t n )
{
   // v._M_rep()->_M_set_length_and_sharable( n );

   // wait… what?
}

template struct rob< tag< std::string >, …???… >;
// …
```

…for (libstdc++, old ABI)

```
template< typename T, typename R > struct tag1 { using type = R* (T::*)(); };
template< typename R > struct tag2 { using type = void (R::*)( std::size_t ); };

template< typename T > using rep_t = typename std::basic_string<T>::_Rep;

template< typename T >
void resize_uninitialized_impl( std::basic_string<T>& v, const std::size_t n )
{
   // v._M_rep()->_M_set_length_and_sharable( n );
   auto* rep = (v.*result< tag1< std::basic_string<T>, rep_t<T> > >::ptr)();
   (rep.*result< tag2< rep_t<T> > >::ptr)( n );
}

template struct rob< tag1< std::string, rep_t<char> >, &std::string::_M_rep >;
template struct rob< tag2< rep_t<char> >,
                     &rep_t<char>::_M_set_length_and_sharable >;
// …
```

…for (libstdc++, old ABI)

```
template< typename T, typename R > struct tag1 { using type = R* (T::*)(); };
template< typename R > struct tag2 { using type = void (R::*)( std::size_t ); };

// Oops, this is illegal and results in an error, as _Rep is private
template< typename T > using rep_t = typename std::basic_string<T>::_Rep;

template< typename T >
void resize_uninitialized_impl( std::basic_string<T>& v, const std::size_t n )
{
   // v._M_rep()->_M_set_length_and_sharable( n );
   auto* rep = (v.*result< tag1< std::basic_string<T>, rep_t<T> > >::ptr)( n );
   (rep.*result< tag2< rep_t<T> > >::ptr)( n );
}

template struct rob< tag1< std::string, rep_t<char> >, &std::string::_M_rep >;
template struct rob< tag2< rep_t<char> >,
                     &rep_t<char>::_M_set_length_and_sharable >;
```

The Master Thief

https://github.com/facebook/folly/blob/master/folly/memory/UninitializedMemoryHacks.h

The Master Thief

```
// declare for the required types
void resize_uninitialized_impl( std::string& v, const std::size_t n );
void resize_uninitialized_impl( std::basic_string<…>& v, const std::size_t n );
```

…for (MSVC)

```
// define as friend, with access to proxy's template parameters
template< typename T, void (T::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_impl( T& v, const std::size_t n )
   {
      // v._Eos( n );
      (v.*F)( n );
   }
};

// explicit instantiation bypasses access checks!
template struct proxy< std::string, &std::string::_Eos >;
template struct proxy< std::basic_string<…>, &std::basic_string<…>::_Eos >;
```

…for (libc++)

```
template< typename T, void (T::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_impl( T& v, const std::size_t n )
   {
      // v.__set_size( n );
      (v.*F)( n );
      v[ v.size() ] = typename T::value_type( 0 );
   }
};

template struct proxy< std::string, &std::string::__set_size >;
// …
```

…for (libstdc++, C++11 ABI)

```
template< typename T, void (T::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_impl( T& v, const std::size_t n )
   {
      // v._M_set_length( n );
      (v.*F)( n );
   }
};

template struct proxy< std::string, &std::string::_M_set_length >;
// …
```

…for (libstdc++, old ABI)

```
template< typename T, typename R, R* (T::*G)(), void (R::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_impl( T& v, const std::size_t n )
   {
      // v._M_rep()->_M_set_length_and_sharable( n );
      R* rep = (v.*G)();
      (rep->*F)( n );
   }
};

template struct proxy< std::string,
                       std::string::_Rep,
                       &std::string::_M_rep,
                       &std::string::_Rep::_M_set_length_and_sharable >;
// …
```

Other Use Cases

- Good for writing unit tests:
  - Set internal state
  - Verify internal state after the test
  - Call internal functions if needed
  - No need to extend the public interface

Thank You!

https://github.com/taocpp/taopq

Questions?

https://github.com/taocpp/taopq