Accessing Private Parts

Without Getting Into Trouble

Private Parts in C++

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

Private Parts 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
// which can achieve this!
```

First Attempt

The `#define` Version

```
#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 `#define` Version

```
#define class struct
#define private public
#define protected public
```

- Applies to all classes/structs
- Applies via nested includes, including standard headers/types
- May change the layout of classes
- May cause ODR issues when not applied consistently

The `#define` Version

```
#define class struct
#define private public
#define protected public
```

This is not a solution!

Never do this!

Solution(s)

Two Major Solutions

- http://bloglitb.blogspot.com/2010/07/access-to-private-members-thats-easy.html
  - More well known/found by Google, but won't be discussed here
- https://github.com/facebook/folly/blob/master/folly/memory/UninitializedMemoryHacks.h
  - IMHO more direct and scaleable, but hard to understand at first
  - We'll present a cleaned up version on the following slides

Solution

```
// declare overload set for your types
void resize_uninitialized_proxy( std::string& v, const std::size_t n );
void resize_uninitialized_proxy( std::basic_string<…>& v, const std::size_t n );

// generic top-level logic
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_proxy( v, n );
   }
}
```

Solution (MSVC)

```
// define via friend, with access to proxy's template parameters
template< typename T, void (T::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_proxy( 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 >;
```

Solution (libc++)

```
template< typename T, void (T::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_proxy( 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 >;
// …
```

Solution (libstdc++, C++11 ABI)

```
template< typename T, void (T::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_proxy( 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 >;
// …
```

Solution (libstdc++, old ABI)

```
template< typename T, typename R, R* (T::*G)(), void (R::*F)( std::size_t ) >
struct proxy
{
   friend void resize_uninitialized_proxy( 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 >;
// …
```

Thank You!

https://github.com/taocpp/taopq

Questions?

https://github.com/taocpp/taopq