Template Meta-Programming

A real-world use case

PostgreSQL from C++

```
struct connection
{
    template< typename... Args >
    result execute( const char* statement, const Args&... args )
    {
        // call libpq C-style API
    }
};

// conn.execute( “SELECT * FROM tbl WHERE id = $1”, 42 );
```

PostgreSQL from C++

```
struct connection
{
    template< typename... Args >
    result execute( const char* statement, const Args&... args )
    {
        // call libpq C-style API
    }
};

// user_t u;
// conn.execute( “INSERT INTO user VALUES ( $1, $2, $3, $4 )”, u );
```

PostgreSQL from C++

```
struct connection
{
   template< typename... Args >
   result execute( const char* statement, const Args&... args )
   {
      return result( PQexecParams( m_pgconn,        // PGconn*
                                   statement,
                                   n_params,        // number of parameters
                                   param_types,     // const Oid[]
                                   param_values,    // const char* const[]
                                   param_lengths,   // const int[]
                                   param_formats,   // const int[]
                                   0 ) );
   }
};
```

PostgreSQL from C++

```
struct connection
{
    template< typename... Args >
    result execute( const char* statement, const Args&... args )
    {
        return result( PQexecParams( m_pgconn,        // PGconn*
                                     statement,
                                     n_params,        // number of parameters
                                     param_types,     // const Oid[]
                                     param_values,    // const char* const[]
                                     param_lengths,   // const int[]
                                     param_formats,   // const int[]
                                     0 ) );
    }
};
```

```
class connection
{
private:
   result execute_c( const char* statement,
                     int n_params,
                     const Oid types[],
                     const char* const values[],
                     const int lengths[],
                     const int formats[] );

public:
   template< typename... Args >
   result execute( const char* statement, const Args&... args )
   {
      // call execute_c()
   }
};
```

```
template< typename... Ts >
result execute_traits( const char* statement, const Ts&... ts )
{
   // call execute_c()
}

template< typename... Args >
result execute( const char* statement, const Args&... args )
{
   return execute_traits( statement, traits< Args >( args )... );
}
```

```
template< typename T >  // specialised for each type T
struct traits
{
   explicit traits( const T& v ) { … }

   static constexpr std::size_t columns = …;

   template< std::size_t I > static constexpr Oid type() { … }
   template< std::size_t I > const char* value() const { … }
   template< std::size_t I > static constexpr int length() { … }
   template< std::size_t I > static constexpr int format() { … }
};
```

```
template< typename T >  // specialised for each type T
struct traits
{
   explicit traits( const T& v ) { … }

   static constexpr std::size_t columns = …;

   template< std::size_t I > static constexpr Oid type() { … }
   template< std::size_t I > const char* value() const { … }
   template< std::size_t I > static constexpr int length() { … }
   template< std::size_t I > static constexpr int format() { … }
};
```

```
template< typename T >  // specialised for each type T
struct traits
{
   explicit traits( const T& v ) { … }

   static constexpr std::size_t columns = …;

   template< std::size_t I > static constexpr Oid type() { … }
   template< std::size_t I > const char* value() const { … }
   template< std::size_t I > const int length() const { … }
   template< std::size_t I > static constexpr int format() { … }
};
```

```
template< std::size_t... Os, std::size_t... Is, typename... Ts >
result execute_idx( const char* statement, const std::tuple< Ts... >& tuple )
{
   const Oid types[] = { std::get< Os >( tuple ).type< Is >()... };
   const char* const values[] = { std::get< Os >( tuple ).value< Is >()... };
   const int lengths[] = { std::get< Os >( tuple ).length< Is >()... };
   const int formats[] = { std::get< Os >( tuple ).format< Is >()... };
   return execute_c( statement, sizeof...( Os ), types, values, lengths, formats );
}

template< typename... Ts >
result execute_traits( const char* statement, const Ts&... ts )
{
   return execute_idx( statement, std::tie( ts... ) );
}
```

```
template< std::size_t... Os, std::size_t... Is, typename... Ts >
result execute_idx( const char* statement, const std::tuple< Ts... >& tuple )

// wait... what are the Os and Is, and where are they coming from??

{
   const Oid types[] = { std::get< Os >( tuple ).type< Is >()... };
   const char* const values[] = { std::get< Os >( tuple ).value< Is >()... };
   const int lengths[] = { std::get< Os >( tuple ).length< Is >()... };
   const int formats[] = { std::get< Os >( tuple ).format< Is >()... };
   return execute_c( statement, sizeof...( Os ), types, values, lengths, formats );
}

template< typename... Ts >
result execute_traits( const char* statement, const Ts&... ts )
{
   return execute_idx( statement, std::tie( ts... ) );
}
```

```
template< std::size_t... Os, std::size_t... Is, typename... Ts >
result execute_idx( const char* statement, const std::tuple< Ts... >& tuple )
   // What we need:
   // Os...: 0 1 2 2 2 3 3 4
   // Is...: 0 0 0 1 2 0 1 0
{
   const Oid types[] = { std::get< Os >( tuple ).type< Is >()... };
   const char* const values[] = { std::get< Os >( tuple ).value< Is >()... };
   const int lengths[] = { std::get< Os >( tuple ).length< Is >()... };
   const int formats[] = { std::get< Os >( tuple ).format< Is >()... };
   return execute_c( statement, sizeof...( Os ), types, values, lengths, formats );
}

template< typename... Ts >
result execute_traits( const char* statement, const Ts&... ts )
{
   // Ts::columns...: 1 1 3 2 1
   return execute_idx( statement, std::tie( ts... ) );
}
```

```
template< std::size_t... Os, std::size_t... Is, typename... Ts >
result execute_idx( const char* statement,
                    std::index_sequence< Os... >,
                    std::index_sequence< Is... >,
                    const std::tuple< Ts... >& tuple )
{
   const Oid types[] = { std::get< Os >( tuple ).type< Is >()... };
   const char* const values[] = { std::get< Os >( tuple ).value< Is >()... };
   const int lengths[] = { std::get< Os >( tuple ).length< Is >()... };
   const int formats[] = { std::get< Os >( tuple ).format< Is >()... };
   return execute_c( statement, sizeof...( Os ), types, values, lengths, formats );
}

template< typename... Ts >
result execute_traits( const char* statement, const Ts&... ts )
{
   using gen = internal::gen< Ts::columns... >;
   return execute_idx( statement, gen::outer(), gen::inner(), std::tie( ts... ) );
}
```

```
namespace internal
{
    template< std::size_t... Ns >
    using gen = make< std::make_index_sequence< ( 0 + ... + Ns ) >,
                      std::make_index_sequence< sizeof...( Ns ) >,
                      exclusive_scan_t< std::index_sequence< Ns... > > >;
}
```

```
template< typename S, typename = std::make_index_sequence< S::size() > >
struct exclusive_scan;

template< std::size_t... Ns, std::size_t... Is >
struct exclusive_scan< std::index_sequence< Ns... >,
                       std::index_sequence< Is... > >
{
   template< std::size_t I >
   static constexpr std::size_t partial_sum = ( 0 + ... + ( ( Is < I ) ? Ns : 0 ) );

   using type = std::index_sequence< partial_sum< Is >... >;
};

template< typename S >
using exclusive_scan_t = typename exclusive_scan< S >::type;
```

```cpp
template< typename S, typename = std::make_index_sequence< S::size() > >
struct exclusive_scan;

template< std::size_t... Ns, std::size_t... Is >
struct exclusive_scan< std::index_sequence< Ns... >,
                       std::index_sequence< Is... > >
{
   template< std::size_t I >
   static constexpr std::size_t partial_sum = ( 0 + ... + ( ( Is < I ) ? Ns : 0 ) );

   using type = std::index_sequence< partial_sum< Is >... >;
};

template< typename S >
using exclusive_scan_t = typename exclusive_scan< S >::type;

// Ns: 1 1 3 2 1

// exclusive_scan_t: 0 1 2 5 7
```

```cpp
template< typename S, typename = std::make_index_sequence< S::size() > >
struct exclusive_scan;

template< std::size_t... Ns, std::size_t... Is >
struct exclusive_scan< std::index_sequence< Ns... >,
                       std::index_sequence< Is... > >
{
   template< std::size_t I >
   static constexpr std::size_t partial_sum = ( 0 + ... + ( ( Is < I ) ? Ns : 0 ) );

   using type = std::index_sequence< partial_sum< Is >... >;
};

template< typename S >
using exclusive_scan_t = typename exclusive_scan< S >::type;

// Ns: 1 1 3 2 1

// exclusive_scan_t: 0 1 2 5 7
```

```
template< typename S, typename = std::make_index_sequence< S::size() > >
struct exclusive_scan;

template< std::size_t... Ns, std::size_t... Is >
struct exclusive_scan< std::index_sequence< Ns... >,
                       std::index_sequence< Is... > >
{
   template< std::size_t I >
   static constexpr std::size_t partial_sum = ( 0 + ... + ( ( Is < I ) ? Ns : 0 ) );

   using type = std::index_sequence< partial_sum< Is >... >;
};

template< typename S >
using exclusive_scan_t = typename exclusive_scan< S >::type;

// Ns: 1 1 3 2 1

// exclusive_scan_t: 0 1 2 5 7
```

```cpp
template< typename S, typename = std::make_index_sequence< S::size() > >
struct exclusive_scan;

template< std::size_t... Ns, std::size_t... Is >
struct exclusive_scan< std::index_sequence< Ns... >,
                       std::index_sequence< Is... > >
{
   template< std::size_t I >
   static constexpr std::size_t partial_sum = ( 0 + ... + ( ( Is < I ) ? Ns : 0 ) );

   using type = std::index_sequence< partial_sum< Is >... >;
};

template< typename S >
using exclusive_scan_t = typename exclusive_scan< S >::type;

// Ns: 1 1 3 2 1

// exclusive_scan_t: 0 1 2 5 7
```

```
template< typename S, typename = std::make_index_sequence< S::size() > >
struct exclusive_scan;

template< std::size_t... Ns, std::size_t... Is >
struct exclusive_scan< std::index_sequence< Ns... >,
                       std::index_sequence< Is... > >
{
   template< std::size_t I >
   static constexpr std::size_t partial_sum = ( 0 + ... + ( ( Is < I ) ? Ns : 0 ) );

   using type = std::index_sequence< partial_sum< Is >... >;
};

template< typename S >
using exclusive_scan_t = typename exclusive_scan< S >::type;

// Ns: 1 1 3 2 1

// exclusive_scan_t: 0 1 2 5 7
```

```
namespace internal
{
    template< std::size_t... Ns >
    using gen = make< std::make_index_sequence< ( 0 + ... + Ns ) >,
                      std::make_index_sequence< sizeof...( Ns ) >,
                      exclusive_scan_t< std::index_sequence< Ns... > > >;
}

// Ns: 1 1 3 2 1

// sum: 8

// first: 0 1 2 3 4 5 6 7
// second: 0 1 2 3 4
// exclusive_scan_t: 0 1 2 5 7
```

```
template< typename, typename, typename >
struct make;

template< std::size_t... Is,
          std::size_t... Js,
          std::size_t... Ns >
struct make< std::index_sequence< Is... >,
             std::index_sequence< Js... >,
             std::index_sequence< Ns... > >
{
   template< std::size_t I >
   static constexpr std::size_t count = ( 0 + ... + ( ( Ns <= I ) ? 1 : 0 ) ) - 1;

   template< std::size_t J >
   static constexpr std::size_t select = ( 0 + ... + ( ( Js == J ) ? Ns : 0 ) );

   using outer = std::index_sequence< count< Is >... >;
   using inner = std::index_sequence< ( Is - select< count< Is > > )... >;
};
```

```
template< std::size_t... Is, std::size_t... Js, std::size_t... Ns >
struct make
{
   template< std::size_t I >
   static constexpr std::size_t count = ( 0 + ... + ( ( Ns <= I ) ? 1 : 0 ) ) - 1;

   template< std::size_t J >
   static constexpr std::size_t select = ( 0 + ... + ( ( Js == J ) ? Ns : 0 ) );

   using outer = std::index_sequence< count< Is >... >;
   using inner = std::index_sequence< ( Is - select< count< Is > > )... >;
};

// Js: 0 1 2 3 4
// Ns: 0 1 2 5 7

// Is:    0 1 2 3 4 5 6 7 ← count: How many Ns are less or equal each I? (minus 1)
// outer: 0 1 2 2 2 3 3 4
```

```
template< std::size_t... Is, std::size_t... Js, std::size_t... Ns >
struct make
{
   template< std::size_t I >
   static constexpr std::size_t count = ( 0 + ... + ( ( Ns <= I ) ? 1 : 0 ) ) - 1;

   template< std::size_t J >
   static constexpr std::size_t select = ( 0 + ... + ( ( Js == J ) ? Ns : 0 ) );

   using outer = std::index_sequence< count< Is >... >;
   using inner = std::index_sequence< ( Is - select< count< Is > > )... >;
};

// Js: 0 1 2 3 4
// Ns: 0 1 2 5 7

// Is:    0 1 2 3 4 5 6 7
// outer: 0 1 2 2 2 3 3 4 ← select: Map outer to Ns (via Js)
// tmp:   0 1 2 2 2 5 5 7
```

```
template< std::size_t... Is, std::size_t... Js, std::size_t... Ns >
struct make
{
   template< std::size_t I >
   static constexpr std::size_t count = ( 0 + ... + ( ( Ns <= I ) ? 1 : 0 ) ) - 1;

   template< std::size_t J >
   static constexpr std::size_t select = ( 0 + ... + ( ( Js == J ) ? Ns : 0 ) );

   using outer = std::index_sequence< count< Is >... >;
   using inner = std::index_sequence< ( Is - select< count< Is > > )... >;
};

// Js: 0 1 2 3 4
// Ns: 0 1 2 5 7

// Is:    0 1 2 3 4 5 6 7
// tmp:   0 1 2 2 2 5 5 7 ← Subtract tmp from Is
// inner: 0 0 0 1 2 0 1 0
```

```
template< std::size_t... Is, std::size_t... Js, std::size_t... Ns >
struct make
{
   template< std::size_t I >
   static constexpr std::size_t count = ( 0 + ... + ( ( Ns <= I ) ? 1 : 0 ) ) - 1;

   template< std::size_t J >
   static constexpr std::size_t select = ( 0 + ... + ( ( Js == J ) ? Ns : 0 ) );

   using outer = std::index_sequence< count< Is >... >;
   using inner = std::index_sequence< ( Is - select< count< Is > > )... >;
};

// Is: 0 1 2 3 4 5 6 7
// Js: 0 1 2 3 4
// Ns: 0 1 2 5 7

// outer: 0 1 2 2 2 3 3 4
// inner: 0 0 0 1 2 0 1 0
```

```
template< std::size_t... Os, std::size_t... Is, typename... Ts >
result execute_idx( const char* statement, …, const std::tuple< Ts... >& tuple )
{
   const Oid types[] = { std::get< Os >( tuple ).type< Is >()... };
   const char* const values[] = { std::get< Os >( tuple ).value< Is >()... };
   const int lengths[] = { std::get< Os >( tuple ).length< Is >()... };
   const int formats[] = { std::get< Os >( tuple ).format< Is >()... };
   return execute_c( statement, sizeof...( Os ), types, values, lengths, formats );
}

template< typename... Ts >
result execute_traits( const char* statement, const Ts&... ts )
{
   using gen = internal::gen< Ts::columns... >;
   return execute_idx( statement, gen::outer(), gen::inner(), std::tie( ts... ) );
}

template< typename... Args >
result execute( const char* statement, const Args&... Args )
{
   return execute_traits( statement, traits< Args >( args )... );
}
```

Thank You!

https://github.com/taocpp/taopq

Questions?

https://github.com/taocpp/taopq