

Clean C++ without SOAP

IPC with C++ templates

*If all your code is in C++,
what use is language neutrality?*

Motivation ?

- Clean, modern, native C++
- Lightweight
- No intermediate language
- Efficient
- Supports composition
- Portable between C++ compilers

Existing solutions

- DCE RPC
- COM
- Rendezvous
- XML-RPC
- SOAP

DCE RPC

- Ideally suited to C
- Intermediate language IDL
- Intermediate proxy generators
- Can be bound to supported transports
- Widely supported

COM

- Ideally suited to Visual Basic, less than natural to C++
- Intermediate language IDL
- Intermediate proxy generators (sometimes)
- Microsoft specific

Rendezvous tm

- Multi language
- Message based, not function based
- Runtime checked
- Multiplatform
- Proprietary (vendor: Tibco)

XML RPC and SOAP

- Language independent
- Intermediate language WDL
- Intermediate proxy generators (function driven)
- Can be message driven
- Multiplatform
- Transport independent
- Expensive decode

marshal::

- Very C++ specific, very natural to C++
- Function call semantics (not message based)
- No intermediate language
- No external proxy generators
- Portable across C++ compilers
- Lightweight
- Can be bound to different transports
- Support for different encoding schemes
- As platform portable and as efficient, or not, as you want

Ease of use

// Server process:

```
vector<string> get_titles( const string & author )  
{  
    // your implementation  
}
```

// Client process:

```
vector<string> titles = proxy->get_titles( author );
```

- Think about how to implement this using RPC, COM, SOAP, etc.

Steps

- serialize [in] arguments
- transport to server (callee)
- reconstruct arguments in the server process
- dispatch the call to the target function
- serialize [out] arguments and return value
- transport back to client (caller)
- update client's (caller) arguments

Direction selection

How to discriminate **[in]** and **[out]** arguments?

- **[in]** T, const T &, const T *
- **[in/out]** T &, T *
- **[out]** T *using the return value*

```
void get_name(string & given, string & family);
```

[in/out] vs [out]

```
void get_name(string & given, string & family);
```

- C++ does not have a truly [out-only] parameter type. Passing a reference on the stack is cheap.
- Assignment to parameters occurs within the function being called. Thus the function must receive an object upon which it can call the assignment operator.
- Typically the input arguments will be empty strings, but they might contain old values to be overwritten by the assignment operator.
- Sending redundant data to the server can be very costly.

The need for [out] arguments

```
string given;  
string family;  
for (unsigned i = 0; i < n ; ++i)  
{  
    proxy->get_name(given, family);  
    ...
```

- compare with -

```
for (unsigned i = 0; i < n ; ++i)  
{  
    string given;  
    string family;  
    proxy->get_name(given, family);  
    ...
```

marshal::out< >

```
void get_name( marshal::out<string &> given,  
              marshal::out<string &> family );
```

[in]

```
T, const T &, const T *,  
marshal::in<T>, marshal::in<const T &>,  
marshal::in<const T *>
```

[in/out]

```
T &, T *,  
marshal::in_out<T &>, marshal::in_out<T *>
```

[out]

```
T using the return value  
marshal::out<T &>, marshal::out<T *>
```

marshal::traits< >

```
namespace marshal
{
    template <typename T> struct traits
    {
        typedef ... param_type;
        typedef ... Value_type;
        static param_type to_arg(value_type & val)
        {
            return val;
        }

        enum { in = 1, out = 0,
              returnable = 1, indirect = 0 };
    };
};
```

marshal::traits< >

param_type

- The most efficient way to pass an object as a parameter to a function. This is important as the arguments are passed to various internal functions.

value_type

- A type that can own the resources when the data arrives at the server (callee). For example, a string class is needed to own the storage for a `const char *` to be marshaled.
- The value type is a type with the same wire-representation, that can be received by the callee.

marshal::traits< >

`static param_type to_arg(value_type &)`

- A conversion from `value_type &` to the type expected by the server (callee).
- When `value_type` is `T`, `to_arg` converts `T&` to `T&` and would be optimized away, if it was evaluated.

marshal::traits< >

i n

The caller will serialize the parameter, and send it to the callee which will need to deserialize the parameter.

o u t

The callee will serialize the parameter, and send it back to the caller which will need to deserialize the parameter.

r e t u r n a b l e

Whether the type can be returned. Only value types and handle types can be returned. Handle types are `void*` and `const void*`. An attempt to marshal a function returning `char*` will result in a compilation error.

Meta information

- No meta information regarding the functions need to be transmitted or parameters.
- All the information is included in the function prototype.
- The prototype is visible to the compiler of the caller (client) and the callee (server).
- The traits are used to decide what can be marshaled and in which direction.

C support

This is a C++ library, with limited support for C idioms.

```
char * fgets(char * str, int n, FILE * stream);
```

- Compilation error, return type is not a value type
- Compilation error `char *` is not permitted (`const char *` is okay)
generally arrays are not supported, use `std::vector< >` instead

```
bool fgets(file_t file_handle, string & str);
```

```
bool fgets(file_t file_handle, marshal::out<string &> str);
```

Encoding schemes

```
stream & operator<<(stream &, const T &)  
stream & operator>>(stream &, T &)
```

Two stream types are provided:

```
raw:      raw_ostream, raw_istream, raw_iostream  
lexical: lex_ostream, lex_istream, lex_iostream
```

Any object that is inserted with a `operator<<` must be fully reconstructed using `operator>>`.

```
stream & operator>>(stream &, marshal::traits<T>::value_type &)
```

Encoding schemes

- **raw_stream**
Like **memcpy**, not portable across platforms, but highly suitable for IPC within a single machine. Very efficient, assumes **sizeof** built in types is the same for caller and callee.
- **lex_stream**
Encodes to human readable character stream. No character set is assumed, byte order of wide characters is not defined (subject to change). Relatively portable.
- **null_stream**
Used for debugging.
- Other possible streams
ACE_stream, **ASN1_stream**, **little_endian_stream**

Transport

- Pipes
`marshal::handle_buffer_t`
Suitable for local machine or LAN, for Win32 or POSIX.
- HTTP/S
`marshal::WinHTTP_buffer_t, Win32`
`marshal::WinINET_HTTP_buffer_t, Win32`
`marshal::ISAPI_HTTP_buffer_t, Win32`
`marshal::CGI_buffer_t Win32 or POSIX`
- Other possible transports
raw sockets: ACE socket, netxx socket

Interface definition

```
int strlen(const char * str);  
std::vector<std::string> get_titles( const std::string & author);  
std::string get_website();
```

```
.  
.
```

```
BEGIN_MARSHAL_PROXY( proxy_t )  
    MARSHAL_1ARG( strlen )  
    MARSHAL_1ARG( get_titles )  
    MARSHAL_0ARG( get_website )  
END_MARSHAL_PROXY()
```

Dispatch

```
int message_id = get_id( pmessage );

switch ( message_id )
{
    case MSG_ID_X: do_x( pmessage );
                  break;
    case MSG_ID_Y: do_y( pmessage );
                  break;
}
```

Automatic dispatch

```
struct proxy_t // generated by BEGIN_MASHAL_PROXY on line 100 in source
{
    struct proxy_impl : marshal::detail::proxy_base_t
    {
        enum { line_begin = 100 };

        struct lookup_101 // generated by macro on line 101 in source
        {
            lookup_101() { set_invoker(101 - line_begin, invoke); }
            void invoke(...) // invoke function using dispatch id of 1
        } m_lookup_101;

        struct lookup_102 // generated by macro on line 102 in source
        {
            lookup_102() { set_invoker(102 - line_begin, invoke); }
            void invoke(...) // invoke function using dispatch id of 2
        } m_lookup_102;
    };
};
```

Automatic dispatch

```
struct proxy_t // generated by BEGIN_MASHAL_PROXY on line 100 in source
{
    struct proxy_impl : marshal::detail::proxy_base_t
    {

        struct lookup_101 // generated by macro on line 101 in source
        {
            lookup_101() { set_invoker(offsetof(proxy_impl, m_lookup_101), invoke);}
            void invoke(...) // invoke function using dispatch id of 1
        } m_lookup_101;

        struct lookup_102 // generated by macro on line 102 in source
        {
            lookup_102() { set_invoker(offsetof(proxy_impl, m_lookup_102), invoke);}
            void invoke(...) // invoke function using dispatch id of 2
        } m_lookup_102;
```

Automatic dispatch

- Using `offsetof` is really a little more complex:

```
#include <iostream>

int main()
{
    struct S { int marker1; int marker2; int lookup1; int lookup2; };

#define DISPATCH_ID(S, lookup) ((offsetof(S, lookup) - offsetof(S, marker2)) \
                                / (offsetof(S, marker2) - offsetof(S, marker1)))

    std::cout << DISPATCH_ID(S, lookup1) << ", " << DISPATCH_ID(S, lookup2);

    return 0;
}
```

Automatic dispatch issues

- Using the preprocessor means that inserting vertical white space changes the dispatch ids.
- Using `offsetof` is strictly speaking for POD only, and may generate warnings.
- Using member initialization order has a slight runtime overhead.

Composition

- Normally there are two streams, input and output.
- The output encoding scheme and transport of the caller must match the input scheme and transport of the callee.
- The input encoding scheme and transport of the caller must match the output scheme and transport of the callee.
- A third debug stream can be supplied for logging. An example can be downloaded that binds the debug stream to `std::clog` (normally `null_stream` is used).

Usage - caller

```
int main()
{
    HANDLE os_handle = open_os_handle("\\\\. \\pipe\\p1");

    marshal::handle_buffer_t two_way_transport(os_handle);
    marshal::raw_ostream encoder(two_way_transport);

    // Configure with a debugging stream
    proxy_t<marshal::caller, marshal::raw_ostream,
           marshal::raw_istream, std::ostream> proxy;

    proxy.bind(encoder, encoder, std::clog);

    // Start marking remote function calls
    int retval = proxy->strlen("hello");
    std::vector<std::string> titles = proxy->get_titles("Koenig");
    std::string home_page = proxy->get_website();

    return EXIT_SUCCESS;
}
```

Usage - callee

```
int main()
{
    HANDLE os_handle = create_named_pipe("\\\\. \\pipe\\p1");

    marshal::handle_buffer_t transport(os_handle);
    marshal::raw_ostream encoder(transport);

    proxy_t<marshal::callee, marshal::raw_ostream> proxy;

    proxy.bind(encoder); // Bind the proxy object to encoding stream(s)

    // wait for incoming calls, dispatch calls until link is closed
    while (proxy.dispatch_call()) {}

    return EXIT_SUCCESS;
}
```

Usage - callee

- No special support required in the callee's business code:

```
std::vector<std::string> get_titles(const std::string & author)
{
    std::vector<std::string> retval;
    if (author == "Koenig")
    {
        retval.push_back("Accelerated C++");
        retval.push_back("C Traps and Pitfalls");
        retval.push_back("Ruminations on C++");
    }
    else if (author == "Alexandrescu")
    {
        retval.push_back("Modern C++ Design");
    }
    else if (author == "Jagger")
    {
        retval.push_back("Microsoft Visual C# .NET Step by Step");
    }
    return retval;
}
```

Future directions

- Support for timeouts for caller and callee
- Byte ordered `wchar_t` for `lex_stream`
- More encoders and transports
- Chainable configurable encoders, zip, encrypt
- Removal of the global namespace requirement
- User definable handle types (not just `void*`)
- bjam build

- Checkout <http://www.bugbrowser.com>,
also C/C++ Users Journal <http://www.cuj.com>