Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

An Introduction to Pantheios: Type-safe, Efficient, Extensible, Generic Logging

Matthew Wilson & Garth Lancaster

[To readers of this version: this document has been prepared in stages over the last 15months as the Pantheios library has been refined (and production tested). In no way does this document represent either the final structure or tone/quality of the eventual articles(s), but rather is a sketch of the main ideas, and the general thrust of the discussion. In other words, it's a taster for now, and we're definitely up for comment, but we expect you to be moved more by the quality of Pantheios than of this (version of) this write up. ☺

Note: bits of this discussion come from Imperfect C++, and also from my next two books, "*Extended STL, volume 1*" (which is nearly ready) and "*Breaking up the Monolith*" (which is nearly ready to be proposed). This may partly explain the somewhat patchy flow.]

[Note: the performance stuff is stuck on the end. That'll be done more comprehensively and neatened up prior to the release of Pantheios.]

[Note: It's important to realise that Pantheios is primarily a logging API library, rather than a logging library. It is as an API that we believe it can justly claim to be the "sweetspot". If that's not made clear in the article, please let us know.]

Structure

Introduction

Logging is an important requirement of most non-trivial software systems, particularly daemons and other unattended processes. In addition to being an invaluable aid to debugging such systems, it can also be an important mechanism for the dissemination of systems management information.

Logging has an uneasy balance of costs and benefits. Logging sub-systems have a cost in binary object size. Logging operations have non-negligible runtime costs, both in the formatting of the logging statements – which may be complex – and in the packing and transport of formatted log messages.

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

Traditionally, logging APIs have addressed well only a subset of the issues:

- Variadic functions – `fprintf()`, `syslog()`, and friends – usually have reasonable performance characteristics, but are not type-safe.
- The IOStreams are type-safe, but are (sometimes very) slow. Worse, their output is not atomic: in multithreaded processes, elements from two or more statements executed in concurrent threads may be interleaved in the output.
- Compile time measures to eliminate logging of particular classes of output often rely on Machiavellian tricks with the pre-processor, or on use of "stub streams" – e.g. `logout << "this won't be in a release build" << endl;` – that still incur some runtime costs. They also make a permanent decision about whether such elided information would be useful in production, which doesn't always turn out right.
- Runtime measures to eliminate logging of particular classes of output either result in pollution of the client code with block scoping conditionals – `if(AreWeLoggingNow(LOG_INFO)) { very_expensive_log_statement(); }` – or incur part of the full cost of the call by testing whether the message is to be sent *after* some/all of its payload has been prepared.

One of the most popular logging infrastructure is SysLog, which incorporates the SysLog protocol and, on UNIX systems, the `syslog()` system call. The SysLog protocol originated in BSD UNIX for the dissemination of distributed logging information on a connectionless basis. Essentially SysLog messages comprise a severity level, a facility, a timestamp, and a text payload. (The severity levels are **DEBUG** (7), **INFORMATIONAL** (6), **NOTICE** (5), **WARNING** (4), **ERROR** (3), **CRITICAL** (2), **ALERT** (1) and **FATAL** (0).) SysLog is a great protocol, and its popularity is due in no small measure to its simplicity, platform-independence and specification of transport broadcast facilities.

The `syslog()` API, however, is just a glorified `printf()`, with all the inherent type-unsafety that that implies. It should be noted that, even for those of us who still cling on to the usability of `printf()` in "normal" life, use of variadic functions in logging is hazardous precisely because it is the case that the most important log statements may be those exercised (and therefore tested) the least often.

[NOTE TO SELF: Need to bring in discussion of log4XXX here, and note the problems/inefficiencies/lack of type safety, etc. There's more than just SysLog.]

In summary:

- Logging is important.
- Logging has a confounding mix of costs and benefits, seemingly without a sweet spot.
- Logging APIs are generally a suboptimal mix of type-safety, atomicity, extensibility, efficiency, and so on.
- SysLog is a great transport protocol, but it is not the be-all and end-all of logging.

In this article we will introduce the Pantheios logging API that, we believe, provides an optimal solution to the various pros and cons of traditional logging APIs, and which is fully extensible to work with SysLog and/or other logging transport mechanisms: It is the logging API sweet spot![1]

Introduction to Pantheios

In working together over the last few years, we have experienced cross-pollination of ideas from our very different backgrounds on a number of issues. In one project we had occasion to marry Garth's experiences of enterprise logging with Matthew's concepts of Shims and Type Tunneling. The result is Pantheios. Pantheios provides the following advantages over existing logging APIs:

- You only pay for what you use
- You only pay once for anything you use
- Highly efficient

[1] We wish to make clear that we believe that Pantheios is the best available logging API library, not the best available logging library. It deliberately does not incorporate the rich feature set of, say, the log4xxx family of libraries, precisely because those libraries can be used in combination with Pantheios (via custom front-end/back-end).

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

- 100% type-safety in C++
- Thread-safe, in both debug and release modes
- Atomic and independent (lock-free) log statement processing
- Automatic library initialization
- Simple to use
- Generic
- Extensible "loggable" application types
- Extensible front-end message filtering, via simple C-APIs
- Extensible back-end transports, via simple C-APIs
- Portable, including Mac OS-X, UNIX and Windows operating systems
- Compiler-independent, currently working with Borland (5.6+), Comeau (4.3.3+), Digital Mars (8.45+), GCC (3.2+), Intel (6+), Metrowerks (8+), Microsoft Visual C++ (5.0+), and should work with any reasonably modern compiler.
- Comes with several stock front and back-end implementations "out of the box", including transports based on UNIX SysLog, a custom Win32 SysLog implementation, ACE, Win32 debugger~~ and~~, `fprintf()`, Win32 Event log, COM Error Object, Win32 (colour-coding) Console, and more.

C++ application code sees a very simple view of the Pantheios libraries, via the application layer components that are primarily composed of eight sets of functions, one for each of the SysLog severity levels: `log_DEBUG()`, `log_INFORMATIONAL()`, `log_NOTICE()`, `log_WARNING()`, `log_ERROR()`, `log_CRITICAL()`, `log_ALERT()`, and `log_EMERGENCY()`, and a set of severity-explicit `log()` functions. These functions take between 1 and 32 parameters that together form a single log statement. (The application layer components are auto-generated via Ruby script, so this range can be easily extended by the user.) The parameters are strings, or types that could be strings (which we'll explain shortly). Thus, we might see simple statements involving purely string types:

```
log(PANTHEIOS_SEV_pantheios::informational, "Server::CloseDown()");
```

or

```
void Server::ReadConfig(char const *configName)
{
  if(! . . .
  {
    log_CRITICAL("Could not open config ", configName, ": ", strerror(errno));
    . . .
```

through to those involving a heterogenous mix of non-string types:

```
try
{
  . . .
}
catch(std::exception &x)
{
  log(PANTHEIOS_SEV_ALERT, "Unexpected exception: ", x);
}
```

and

```
string_t  cmdLine = . . .
EXECPP_RC rc = execpp::exec(cmdLine.c_str(), 0, &results
                 , execpp_fn, NULL, NULL, NULL, &retcode);
if(EXECPP_RC_SUCCESS != rc)
```

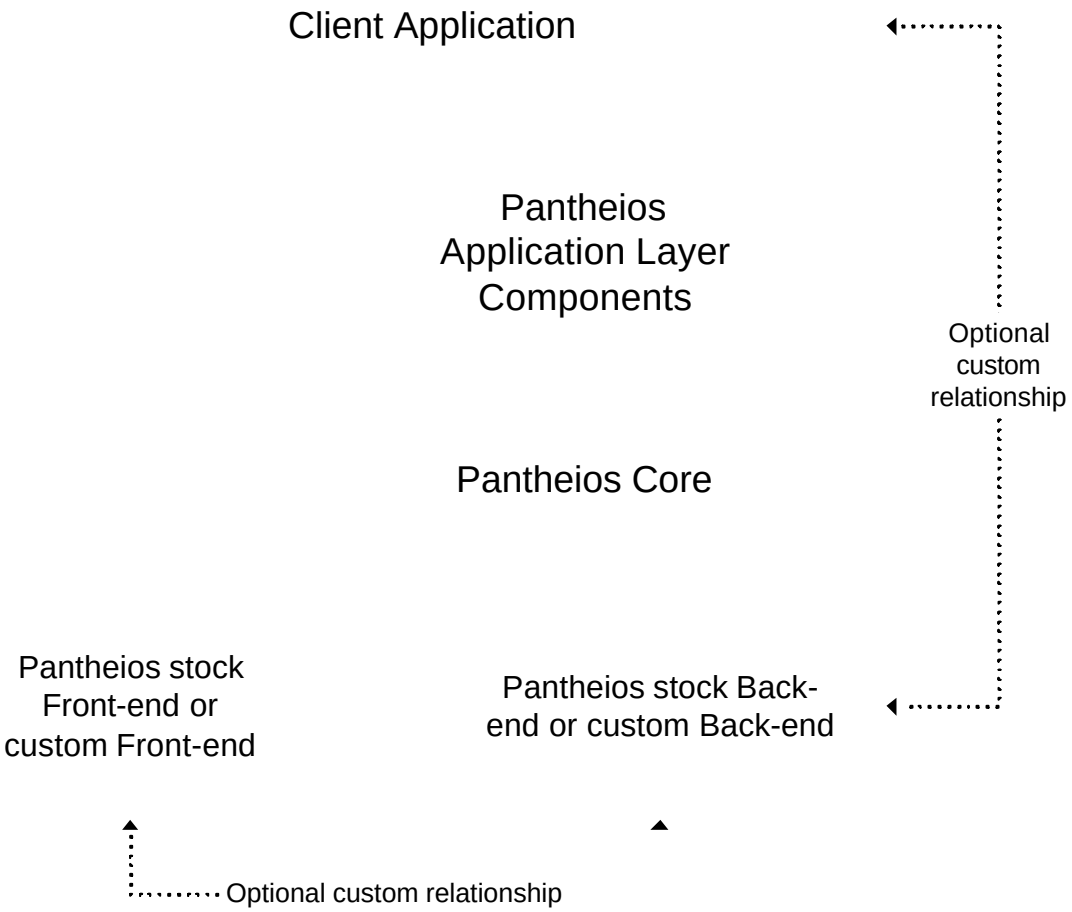
Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

```
{
  // Log cmdline, execpp error, and Win32 error string
  log_ALERT("Failed to execute [", cmdLine, "]: ",
          rc, " (", stlsoft::error_desc(errno), ")");
}
```

in which the `EXECPP_RC` enumeration instance `rc`, the `~~winstl~~stlsoft::error_desc` temporary and the `std::exception` reference `x` are all 'magically' incorporated – via Type Tunnelling (see Sidebar) – into the output without having to explicitly convert them to strings.

Pantheios Architecture

Pantheios has a four-part architecture, comprising application layer, core, front-end and back-~~end, that~~end, which interact with a client application as shown in Figure 1:



The application layer components are the functions and classes that application code invokes to generate log messages. The front-end determines whether a given message will be emitted, based on its severity level. If the message is to be emitted the core efficiently prepares it and dispatches it to the back-end. The back-end transports the message.

Application Layer

The application layer is comprised of several sets of auto-generated functions, for both C and C++ client code, as follows:

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

- 1 set of severity-level explicit `log()` function template overloads (C++ only)
- 8 sets of severity-level implicit function template overloads: `log_DEBUG()`, `log_INFORMATIONAL()`, and so on (C++ only)
- 1 set of `pantheios_log_N()` functions: `pantheios_log_1()`, `pantheios_log_2()`, and so on. (C and C++)

Each function set consists of 32-functions, catering for between 1 and 32 parameters. All told, there are 384 functions in the various function sets, and the task of writing such things is most certainly not for a human: they are auto-generated by a Ruby script, which means they can be regenerated by the user according to requirements, to restrict or expand the number of parameters catered for.

The function template overloads all work in the same way. Consider the 3-parameter version of `log()`:

```
template< typename T0
        , typename T1
        , typename T2
        >
inline void log(int       severity
              , T0 const  &v0
              , T1 const  &v1
              , T2 const  &v2)
{
  if(pantheios_isSeverityLogged(severity))
  {
    log_dispatch_3( PANTHEIOS_SEV_EMERGENCY
                  , stlsoft::c_str_len_a(v0)
                  , stlsoft::c_str_data_a(v0)
                  , stlsoft::c_str_len_a(v1)
                  , stlsoft::c_str_data_a(v1)
                  , stlsoft::c_str_len_a(v2)
                  , stlsoft::c_str_data_a(v2));
  }
}
```

The function's responsibilities are split into two discrete steps. First, the core function `pantheios_isSeverityLogged()` is called to determine whether or not the back end (or one of the back-ends; see section Local/Remote Back-end Splitting) is currently emitting messages of the given severity level. If it is, then the string access shim functions `c_str_len_a()` and `c_str_data_a()` are invoked on the three parameters `v0`, `v1` and `v2`. The results are passed to the core function `log_dispatch_3()`, which takes them into the core.

For those unfamiliar with String Access Shims [GENSTR, IC++], `c_str_data_a()` returns a pointer to a not-necessarily-nul-terminated array of characters (`char const*`) or an instance of a type that is implicitly convertible to such, and `c_str_len_a()` returns the corresponding length of that array (`size_t`). The rules of C++ require that any temporary variables (see Type Tunneling sidebar) returned by the string access shims stay alive until the `log_dispatch_3()` returns, so it's quite safe.

The definition of `log()` ably demonstrates how Pantheios is able to offer all that it does and yet retain the spirit of C: You only pay for what you use. If a given log statement is not emitted then the conversion of its sub-expressions to string form and concatenation of the expressions into a log payload is not carried out. The negligible "cost" in such a case involves the pushing of references to the arguments onto the stack plus a call to `pantheios_isSeverityLogged()`.

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

Further, since the message parameters are converted here into the only form – `char const*` + `size_t` – acceptable to the core, no further conversion is necessary, nor is there any need to perform any further examination of the contents of any of the message parameters, i.e. there are no cycle-consuming `strlen()` calls down the line.

It should be equally clear how the claimed 100% type-safety is inherent in the use of the shims. If you pass an instance of a type for which they're not defined compilation fails. This is anything but the case with variadic logging functions, where a bug can easily get past the compiler and lurk in the executable until just the wrong moment. (This is also true for ostensibly type-safe APIs like IOStreams, where the implicit acceptance of `void const*` can lead to many an unhappy trail through the code to try and work out why a particular log statement element has been displayed incorrectly. In an emergency/alert code path that is travelled all but never, this activity can happen a *long* time after coding.)

For C client code, the application layer provides the set of `pantheios_log_N()` functions, as in:

```
int pantheios_log_1(pan_sev_t   severity
                  , char const  *ptr0
                  , int         size0); /* = -1 */
int pantheios_log_2(pan_sev_t   severity
                  , char const  *ptr0
                  , int         size0
                  , char const  *ptr1
                  , int         size1); /* = -1 */
```

These take pairs of string pointer + length. If the length is –1, then the string is assumed to be nul-terminated, and the length determined with `strlen()`. Use of these functions looks like:

```
int   numUsers  = 1000000;
char  szNumUsers[101];

pantheios_log_3(PANTHEIOS_SEV_ALERT
              , "We're sure there're likely to be >", -1
              , szNumUsers, sprintf(&szNumUsers[0], "%020d", numUsers)
              , " satisfied users of Pantheios", -1);
```

Naturally, C++ client code is much nicer than the C equivalent.

Finally, honesty requires us to mention that the core API also includes the `printf()`-like `pantheios_printf()` and `pantheios_vprintf()` functions.

```
int pantheios_printf( pan_sev_t   severity
                    , char const  *format
                    , ...);
int pantheios_vprintf(pan_sev_t   severity
                    , char const  *format
                    , va_list     args);
```

But these are not type-safe, so we keep them in a cardboard box in the scullery, and deplore their use.

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

sidebar: String Access Shims

The technology underpinning the powerful mix of genericity, efficiency and type-safety in Pantheios is *String Access Shims*, described in the article "*Generalised String Manipulation: Access Shims and Type Tunneling*" (CUJ, August 2003) and in *Imperfect C++* [IC++]. These are a near-ubiquitous feature of the STLSoft libraries.

Shims represent a mechanism for generalisation between (aspects of) conceptually related by physically unrelated types.

Shims afford moderate to high cohesion with minimal (and often zero) coupling.

A shim is an unbounded suite of functions with the following characteristics (NICO):

- Name – the namespace and function name, e.g. stlsoft::c_str_ptr
- Intent – the common purpose of all functions constituting the shim, e.g. return a pointer to nul-terminated non-`NULL` character string representing the instance of the types for which the shim is defined
- Category – the shim category, i.e. Attribute, Access, Composite, Control, Conversion, Logical. The category describes the naming, the behaviour and the constraints on use of the shim. For example, the return values of Conversion and Access shim instances must be used only within the statement within which the shim is invoked. (See later in this article for further discussion.)
- Ostensible Return Type – the type which is returned by the shim functions, or to which the return type of (some of) the shim functions must be implicitly convertible. (See later in this article for further discussion.)

For example, the following are

Essentially, a shim is unbounded suite of overloaded functions that manipulate logically related but physically unrelated types – e.g. `std::string` and `char const*` - in a generic manner. There are Attribute Shims, Logical Shims, Conversion Shims, and Control Shims. Access Shims are a composite of Attribute Shims and Conversion Shims that elicit a common property of the types to which they are applied and which may involve conversions via the return of temporary variables that are themselves implicitly convertible to the target type.

By utilising the **c_str_data_a** and **c_str_len** access shims, the Pantheios application is compatible with *any* type for which these shims are defined, and are thereby both generic and infinitely extensible without requiring any changes to the Pantheios code base.

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

sidebar: Type Tunneling

The *Type Tunneling* pattern describes the situation where a (usually lightweight) generic conversion layer (called the *Tunneling Layer*) effects compatibility between arbitrary types in application code and the limited, fixed type (or types) acceptable to an API. *Type Tunneling* is usually, though not always, associated with the use of Shims. In such cases, the shims "[allow] an external type to be *tunneled* through an interface and presented to the internal type in a recognised and compatible form" [IC++].

In the case of Pantheios, it is the string access shims **c_str_data_a** and **c_str_len_a** which abstract each log statement component into a string form, thereby 'tunneling' them through the Pantheios application layer functions – `log()`, `log_DEBUG()`, and so on – to the Pantheios core API, where they are manipulated in the "recognised and compatible form" of the `pan_slice_t` type.

```
struct pan_slice_t
{
  size_t      len;
  char const  *ptr;
};
```

Consider the following fragment:

```
#include <comstl/string_access.hpp>
#include <winstl/string_access.hpp>

HWND    wnd = . . .;
VARIANT var = . . .;

pantheios::log_DEBUG("v=", var, "; w=", wnd);
```

This 4-part statement results in a log entry consisting of the two prefix string literals concatenated with the string forms of the given `HWND` and `VARIANT` variables. The WinSTL `c_str_len_a(HWND)` function establishes the window text length using `::GetWindowTextLength()`, and elicits the window text in `c_str_data_a(HWND)`, returning a temporary instance of a class – `c_str_ptr_HWND_proxy` – that provides an implicit conversion to `char const*`. Analogous behaviour is provided in the COMSTL shim functions for the COM `VARIANT` type. The temporaries are held "alive" until the statement they are in is complete, by which time the log statement has been prepared and emitted.

The construction of these types and the elicitation of the string forms have some cost, but this cost would have to be borne *somewhere* in the application, whatever the logging API being used. And remember, the `log_DEBUG()` function will not even get to the business of invoking the string access shims if the DEBUG severity level is not being logged. In that case, the cost of manipulating the parameters is just the cost of putting four references to the stack, which is not exactly going to break the bank.

Core Layer

The responsibility of the core is to receive the message parameters and concatenate them into a contiguous nul-terminated string, and then emit them to the back-end(s). Because the application layer components pass down generic types, the core is entirely independent of, and oblivious to, the formulation of particular log statements: it just concatenates them and sends them ~~to the back-end(s)~~on.

```
struct pan_slice_t
{
```

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

```
  size_t      len;
  char const  *ptr;
};

int pantheios_init(void);
void pantheios_uninit(void);

int pantheios_isSeverityLogged(pan_sev_t severity);
char const *pantheios_severityString(pan_sev_t severity);
int pantheios_log_n(pan_sev_t         severity
                  , size_t            numSlices
                  , pan_slice_t const *slices);
```

Pantheios is a reference-counted API [IC++]. In other words, the first successful call to `pantheios_init()` initialises the library, including the front and back-ends, and each subsequent call increases the reference-count. Conversely, the library is only uninitialised when a matching number of calls to `pantheios_uninit()` are made. Thankfully you don't have to care about the initialisation logic, since the Pantheios root C++ header, pantheios/pantheios.hpp, includes pantheios/cpp/initialiser.hpp, which uses Schwarz counters ([IC++]) to ensure that the library is automatically initialised and ready for use, irrespective of which C++ compilation unit happens to have its non-local static objects (i.e. globals) introduce first. You can read up on these techniques in Part 2 of *Imperfect C++*, or just take our word for it. This can be suppressed by definition of the `PANTHEIOS_NO_AUTO_INIT` pre-processor symbol.

(Note that, by default, the Schwarz counter initialisation is not performed in dynamic libraries, in order to allow for control over the time of initialisation and (slightly more) graceful handling of initialisation failure. It can be forced by definition of `PANTHEIOS_FORCE_AUTO_INIT`.)

a. Atomic. Because the subexpressions of a log-statement are concatenated and emitted as a single string, there is no possibility of one thread emitting a message while another is halfway through (as long as a user hasn't plugged in a non-thread-safe back-end in a multi-threaded program, of course).

Concatenation occurs within a stack instance of `stlsoft::auto_buffer` (See "Efficient Variable Automatic Buffers"), whose internal size (configurable at compile-time) is 2K. Thus, any messages whose total length is <= this size can be constructed and emitted without a single memory allocation.

Front-end Layer

The role of the front-end is to define the process identity and to arbitrate log requests, based on severity level. Each time a log statement is executed, the core consults the front-end, via the `pantheios_fe_isSeverityLogged()` function, as to whether the given severity is to be logged. If it is, then the Pantheios code running in the application layer prepares the entry arguments as string slices, and passes them to the core (via `pantheios_log_n()`).

```
int pantheios_fe_init(int reserved, void **ptoken);
void pantheios_fe_uninit(void *token);
char const *pantheios_fe_processIdentity(void *token);
int pantheios_fe_isSeverityLogged(void *token, int severity, int backEndId);
```

`pantheios_fe_init()` is called by the core when it is initialising, and returns >=0 to indicate success. `pantheios_fe_uninit()` is called by the core when it is uninitialising (or when another part of the core has failed to initialise after the Front-end). `pantheios_fe_processIdentity()` is called during initialisation and returns the identity of the process, to be used by the back-end when formatting log messages for output.

The Front-end implementation can, via the `ptoken` parameter of `pantheios_fe_init()`, optionally return a context token that the core will maintain on its behalf and give back to it on each call to the other functions. This can be a class instance, or anything else appropriate to the implementation. Front-end implementations must set `*ptoken`, however, even if it's to `NULL`.

Each of these functions is called at most once, and always in the main thread[2], so need not have any special measures for thread-safety. The remaining function, `pantheios_fe_isSeverityLogged()`, is used to arbitrate message preparation and dispatch. It receives the putative message's severity level, and a back-end identifier. This latter parameter may be 0, to indicate any/all output, or another value to identify a particular back-end in the case where Local/Remote Back-end Splitting (see section **be.lrsplit**) is being used. This function can be called at any time, from any thread, so must be thread-safe and should be efficient and non-blocking. Experience has shown that this is readily achieved, even in cases of sophisticated filtering.

Back-end Layer

The role of the back-end is to provide transport of the message emitted from the core, along with, optionally, any additional formatting as may be appropriate. The back-end API consists of three simple functions:

```
int pantheios_be_init(char const  *processIdentity
                    , int         reserved
                    , void        **ptoken);
void pantheios_be_uninit(void *token);
int pantheios_be_logEntry(void        *feToken
                        , void        *beToken
                        , int         severity
                        , char const  *entry
                        , size_t      cchEntry);
```

As with the Front-end API, the initialisation and uninitialisation functions are called, at most, once per process, in the process main thread, and provide the facility of associating a core-context with the back end. The third function, `pantheios_be_logEntry()`, is responsible to receiving the message emitted from the core. It takes the front-end and back-end tokens derived during initialisation, the message severity, the nul-terminated message payload, and the length (excluding nul-terminator) of the payload. The front-end token facilitates back-end splitting (where a Composite back-end passed off output to several concrete back-ends; see later). Specifying the severity allows back-ends to dispatch to different streams, and, as is the case with the SysLog protocol, to incorporate the severity in the log message. Proving both nul-termination and length removes the need for back-ends to apply one or determine the other to suit the underlying transport APIs. Naturally, if the process is multi-threaded, `pantheios_be_logEntry()` must be safely callable from multiple concurrent threads.

Handling Built-in and User-defined Types

The Application layer functions implicitly handle all types that are strings or that can be expressed, via String Access Shims, as such, including `char const*`, `char[]`, `std::string`, `stlsoft::string_view`, `VARIANT`, `ACE_String_Base`, `HWND`, `struct tm`, `FILETIME`, `std::exception`. However, there are many types, including built-in and user-defined types, which are not handled automatically.

Built-in Types

For built-in types, Pantheios provides the converter classes `integer`, `real` and `pointer`, which are used as follows:

```
short                   s     = 123;
int                     i     = 456;
```

[2] Or whatever thread makes the first call to pantheios_init(), in the case where auto-initialisation is suppressed.)

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

```
  long                    l     = 789;
  float                   f     = 0.123;
  double                  d     = 0.456;
  long double             ld    = 0.789;
  void                    *p    = &l;
  using namespace pantheios;

  log_INFORMATIONAL(
    "Integers (", integer(s, 4 | fmt::zeroPadded), ", ", integer(i), ", ", integer(l), "); "
  , "floating points (", real(f), ", ", real(d), ", ", real(ld), "); "
  , "and a pointer (", pointer(p, fmt::hex | fmt::zeroXPrefix), ")");
```

which outputs

"Integers (0123, 456, 789); floating points (0.123, 0.456, 0.789); and a pointer (0x12ff28)"

These inserter classes maintain a pointer/reference to the variable and any formatting information, and only effect conversion on demand, when (the first of) their `data()` or `length()` methods are invoked.

Other Stock Inserters

Two other stock inserters are also supplied. The `blob` inserter class represents a set of bytes as a sequence of hexadecimal tokens, which may be grouped in powers of 2 (0, 1, 2, 4, …32), and groups and/or lines, separated by user-defined sequences. For example, the following code:

```
  int         ar[2] = { 0x00112233, 0x44556677 };
  char        s[]   = "abc";
  std::string str("def");

  pantheios::log(pantheios::notice, "s=", s, ", blob=", pantheios::blob(ar, sizeof(ar), 2, "-"),
", str=", str);
```

produces the output:

```
   s=abc, blob=2233-0011-6677-4455, str=def
```

The other stock inserter is b64, which converts a set of bytes to a base-64 string. This uses the b64 library (http://synesis.com.au/software/b64.html), however, and so is not linked in to the Pantheios core. It must be linked separately, along with the requisite b64 library.

User-defined Types

User-defined types may be handled in three ways:

1. By writing explicit conversion code in your application and passing the result as a parameter in a log statement, or
2. By writing explicit conversion functions that return an instance of a type, such as `std::string`, that is handled automatically, or
3. By writing a custom inserter class
4. By defining string access shim functions for your type.

1.1.1 Option 1: Writing explicit conversion code

Consider the following code:

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

```
void ChangeWindowSize(RECT const &rc)
{
  if(pantheios::isSeverityLogged(pantheios::debug))
  {
    char  s[200];
    sprintf("%d, %d, %d, %d", rc.left, rc.top, rc.right, rc.bottom");
    log_DEBUG("Changing window size to: ", s);
  }
  . . .
```

This is a bad choice for several reasons. It's hard coding, which is both annoying to have to do, and likely to suffer from copy-pastitis if you want to log another `RECT` somewhere else. It's verbose, whereas logging needs to impinge minimally on application code for it to be attractive to developers, and to remain effective throughout the lifetime of a product. And it's inefficient, unless, as shown above, you make the explicit test for the severity level before effecting the conversion(s).

Option 1 is really what Pantheios is designed to avoid.

1.1.2 Option 1: Write a Conversion Function to a Loggable type

This is a big improvement over option 1, but it still has problems. First, the log statement is more cluttered, though that's a minor problem. A little more serious is that to use it one must know its name, and once a large library of such things develop, it becomes an effort to remember what they're called and where they are.

```
std::string format(RECT const &rc)
{
  char s[200];
  n = sprintf( . . . // conversion as above
  return std::string(s, n);
}

void ChangeWindowSize(RECT const &rc)
{
  log_DEBUG("Changing window size to: ", format(rc));
  . . .
```

But the main objection is that it too is inefficient, because the conversion from `RECT` to `std::string` occurs prior to the severity test.

1.1.3 Option 4: Write a Custom Inserter class

Using the stock inserters as an example, one can write a converter class, as in:

```
class rect
{
public:
  rect(RECT const &rect)
    : m_len(-1)
    , m_value(value)
  {}

public:
```

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

```
  char const *data() const
  {
    if(-1 == m_len)
    {
      convert_();
    }
    return m_buff;
  }
  size_t size() const
  {
    if(-1 == m_len)
    {
      convert_();
    }
    return static_cast<size_t>(m_len);
  }

private:
  void convert_() const; // const_cast<>, and then invoke . . .
  void convert_();       // . . . this one, which does what format() above does

private:
  char       m_buff[200];
  int        m_len;
  RECT const &m_value;
};

namespace stlsoft
{
  inline char const *c_str_data_a(rect const &r)
  {
    return r.data();
  }
  inline size_t c_str_len_a(rect const &r)
  {
    return r.length();
  }
}


void ChangeWindowSize(RECT const &rc)
{
  log_DEBUG("Changing window size to: ", rect(rc));
  . . .
```

This is efficient, and reusable. It is a lot of code to write, however, so you'd only want to do this for regularly used types.

1.1.4 Option 1: Define String Access Shims For Your Type

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

This option requires the definition of two shim functions (and their inclusion into the `stlsoft` namespace), but it leads to more elegant statements, and the adapted type is always automatically compatible so long as the shim functions are included in the compilation unit.

```
namespace stlsoft
{
  CString c_str_data_a(RECT const &rc);
  size_t  c_str_len(RECT const &rc);
} // namespace stlsoft

void ChangeWindowSize(RECT const &rc)
{
  log_DEBUG("Changing window size to: ", rc);
  . . .
```

Further, it is possible with many types for the length of the string to be calculated exactly without performing a string conversion, further increasing efficiency.

Stock Front and Back-ends

Pantheios comes bundled with a number of stock front and back-ends, which cover all the most common logging scenarios on the UNIX and Windows operating systems. No changes to client code are required: for example, you can change the transport mechanism for you application merely by re-linking to a different back-end library.

fe.simple

The stock front end is called **fe.simple**, and it provides a default definition of the Front-end functions. As you can see from Listing X, it requires nothing more of the application code than the definition of a process identity string

```
extern "C" const char FE_SIMPLE_PROCESS_IDENTITY[];
```

Listing X

```
// fe_simple.c
#include <pantheios/pantheios.h>
#include <pantheios/frontend.h>

extern const char FE_SIMPLE_PROCESS_IDENTITY[];

int pantheios_fe_init(int   reserved
                    , void  **ptoken)
{
  *ptoken = NULL;
  return 0;
}
void pantheios_fe_uninit(void *token)
{}
char const* pantheios_fe_processIdentity(void *token)
{
  return FE_SIMPLE_PROCESS_IDENTITY;
}
int pantheios_fe_isSeverityLogged(void  *token
```

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

```
                                , int   severity
                                , int   backEndId)
{
#ifdef NDEBUG
  return severity < PANTHEIOS_SEV_DEBUG;
#else /* ? NDEBUG */
  return 1;
#endif /* NDEBUG */
}
```

Several stock back-ends are supplied, catering for a number of popular log "streams", including:

a. ACE (Adaptive Communications Environment) logger
b. `fprintf()`, to `stderr` (WARNING => EMERGENCY) or `stdout` (DEBUG, INFORMATIONAL, NOTICE)
c. SysLog (UNIX-only), using the `syslog()` API
d. Win32syslog. This effects the equivalent to UNIX's `syslog()` for Pantheios on the Win32 operating system
e. Win32 Debugger. Uses the `OutputDebugString()` Win32 API function
f. L/R split. This is a pseudo back-end that actually splits the output into local and remote back-end streams.
g. Win32 Event Log
h. COM Error Object. Log statements of INFORMATIONAL=>EMERGENCY are used to set the thread's COM error object information.

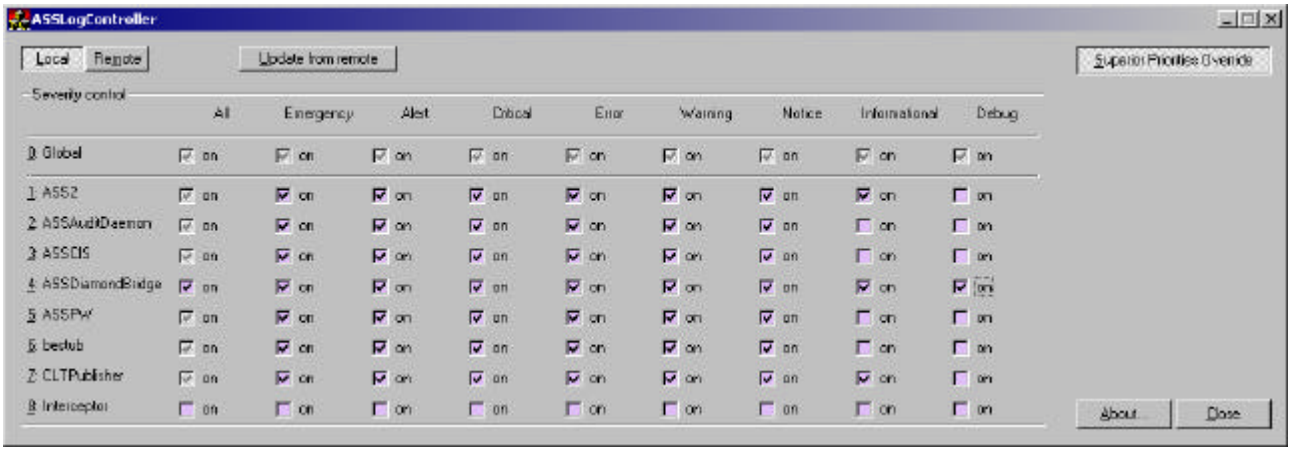
Local/Remote Back-end Splitting

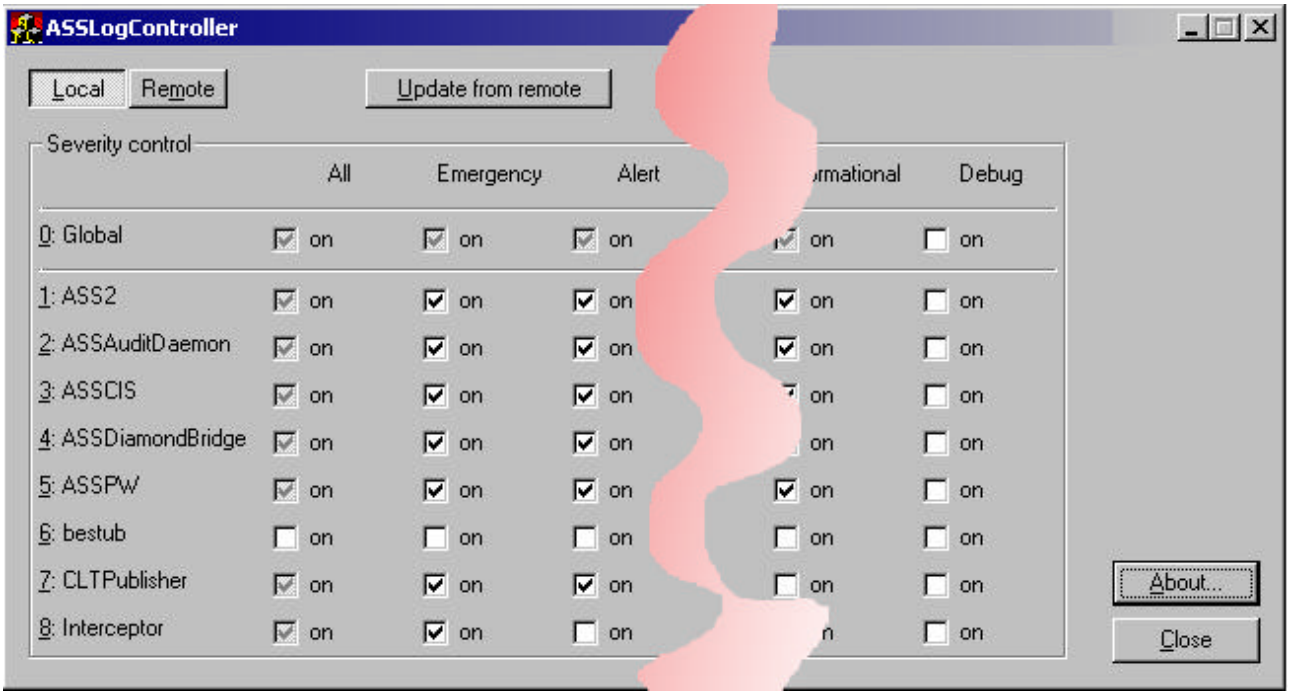
There are cases where having a single output stream is not sufficient. Rather than presenting users of the libraries with extra effort, we have provided the Local/Remote Back-end Splitting library, **be.lrsplit**, which supports differentiated local/remote output. The **be.lrsplit** library implements the back-end API, in terms of its own output API, consisting of `pantheios_be_local_init()`, `pantheios_be_local_uninit()`, `pantheios_be_remote_init()`, `pantheios_be_remote_uninit()`, `pantheios_be_local_logEntry()`, and `pantheios_be_remote_logEntry()`, whose signatures ape those of the back-end API.

Extending Pantheios via Custom Front/Back-ends

If these stock libraries do not satisfy your needs, you can easily implement your own custom variant according to the APIs prescribed, using the stock implementations as an example. And you have the ability to effect interactions between the application and front-end/back-end. For example, in one of our commercial projects – the Auto-claims Switching Service (ASS), which carries all real-time medical insurance claims for continental Australia – we use **be.lrsplit**, coupled with the **Win32Console** and **Win32syslog** back-ends, to provide local console output and/or network SysLog monitoring. We use a custom front-end that communicates, via shared memory, with a runtime process/stream selection GUI application (see Figure X), to enable us to select at runtime which message severity of the eight severity levels from which server processes will be emitted locally and/or remotely. This product suite is currently serving several million claims messages daily, and the cost of Pantheios logging (when Debug and Informational levels are switched off, of course) on performance is virtually undetectable. Naturally, when switching on Debug, one gets all contents of every message on every channel, and that can slow things a little. But that's logging.

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au





Front-end – Back-end cooperation

The four-part architecture ensures a clean separation between the various roles and responsibilities. In particular, the front-end deals with message filtering, and the back-end(s) handle transport. Any additional front-end <-> back-end functionality is handled outside the library. For example, we might want to associate an `fprintf()`-based logging with a specific stream. This can be handled by the front-end and back-end code being mutually aware, such that when Pantheios calls down to `pantheios_be_init()`, it can create an object (to be returned as the `token`) which will have a hook into something in the front-end, such that it actually sends calls to `pantheios_be_logEntry()` 'up' to the front-end library, which can pass calls off to the appropriate destination.

Note: the core and the application layer are written primarily in C++ – fairly advanced C++, to be sure – but the front and back-end APIs are pure C and most of the stock front and back-end libraries are written in C. Thus, C/C++ programmers who are more comfortable in C can easily extend Pantheios.

Drawbacks

There are three minor drawbacks to using Pantheios:

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

First, in the case where an argument is of a type not convertible to string form via a string access shim (or that string access shim is not visible to the given compilation unit), the compilation error messages that result are not completely obvious. However, most compilers make mention of the string access shim functions `c_str_data_a()` and/or `c_str_len()`, ~~so~~ and this usually suffices to turn on the mental light bulb.

Second, There is no facility for, say, specifying the format for all floating-point numbers in a single statement, a la IOStreams' manipulators. This is partly because Pantheios statements are processed atomically – a good thing in a logging API! – and partly because it would introduce a degree of complexity that, so far, we have not considered worthwhile. We've found that the frequency of floating-point statements is sufficiently low that the `real` converter class eminently suffices.

Third, using Pantheios means using STLSoft. Some open-source users require that each library is entirely independent, which is a perfectly reasonable point of view when one considers just how many problems one gets bogged down with when mixing many open-source libraries. However, due to the open-for-extension nature of shims, it simply doesn't make sense for users of Pantheios to have their own shims defined, and therefore miss out on the large (and growing) range of string access shims available as part of STLSoft. (It's also worth noting that STLSoft is 100% header-only, and attempts to exist at the lowest level above the API and it thus a highly lightweight proposition in most cases.)

A more significant practical problem with Pantheios is that selecting and specifying the back-end libraries with which one wishes to link can be quite a verbose activity. For example, one of the test applications that ships with the Pantheios distribution links to.

```
pantheios.core.$(COMP_TAG).mt.debug.lib             // core
pantheios.fe.simple.$(COMP_TAG).mt.debug.lib        // stock front-end: fe.simple
pantheios.be.lrsplit.$(COMP_TAG).mt.debug.lib       // stock back-end:  be.lrsplit
pantheios.bel.Win32Console.$(COMP_TAG).mt.debug.lib // back-end (local): Win32 Console
pantheios.bec.Win32Console.$(COMP_TAG).mt.debug.lib // back-end (common): Win32 Console
pantheios.ber.Win32syslog.$(COMP_TAG).mt.debug.lib  // back-end (remote): Win32 syslog
pantheios.bec.Win32syslog.$(COMP_TAG).mt.debug.lib  // back-end (common): Win32 syslog
```

(where `$(COMP_TAG)` is the compiler identity tag, e.g. `dm` for Digital Mars, `vc71` for Visual C++ 7.1). And that's just for the multithreaded debug build. Each additional configuration (e.g. a single-threaded release build) would have an equivalent list of library names. To be sure, this is a worst-case scenario, but even more "normal" cases can see four or five elements. Thus, for those compilers that support implicit linking, users can simply include the requisite Pantheios implicit-link headers in one of the compilation units of their application, as in:

```
#include <pantheios/implicit_link/core.h>
#include <pantheios/implicit_link/fe_simple.h>
#include <pantheios/implicit_link/be_lrsplit.h>
#include <pantheios/implicit_link/bel_Win32Console.h>
#include <pantheios/implicit_link/ber_Win32Console.h>
#include <pantheios/implicit_link/bel_Win32syslog.h>
#include <pantheios/implicit_link/ber_Win32syslog.h>
```

However, we'd argue that even in cases where you must use explicit linking, this is something readily taken care of in your makefiles or other project files, and can easily be automated. We also provide the Win32 library selector tool (Figure 3) with which you can select the required permutation and format, copy it to the clipboard, and paste it into your makefiles and/or IDE dialogs.

Figure X. Using the library selector for Explicit Linking

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

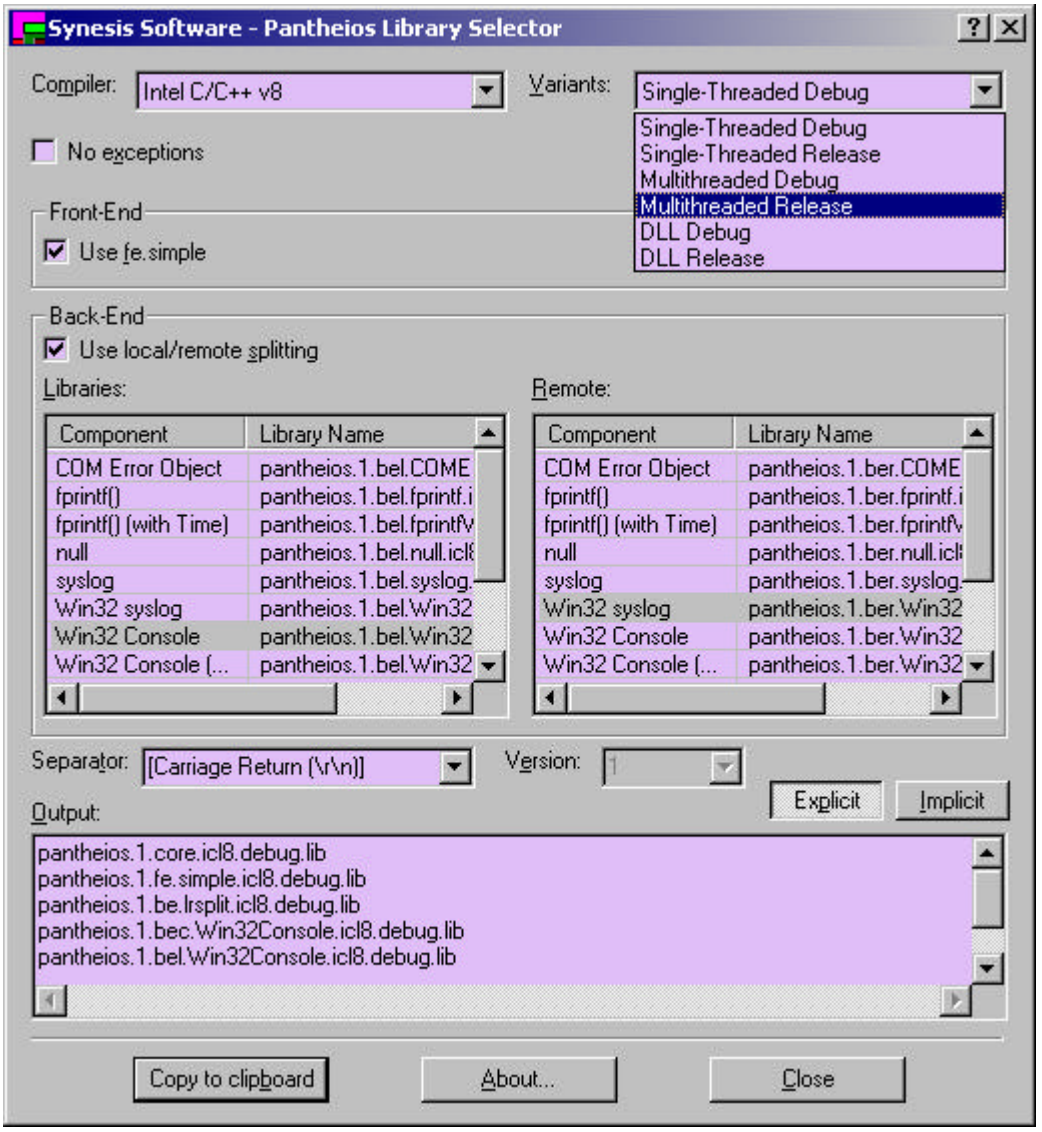
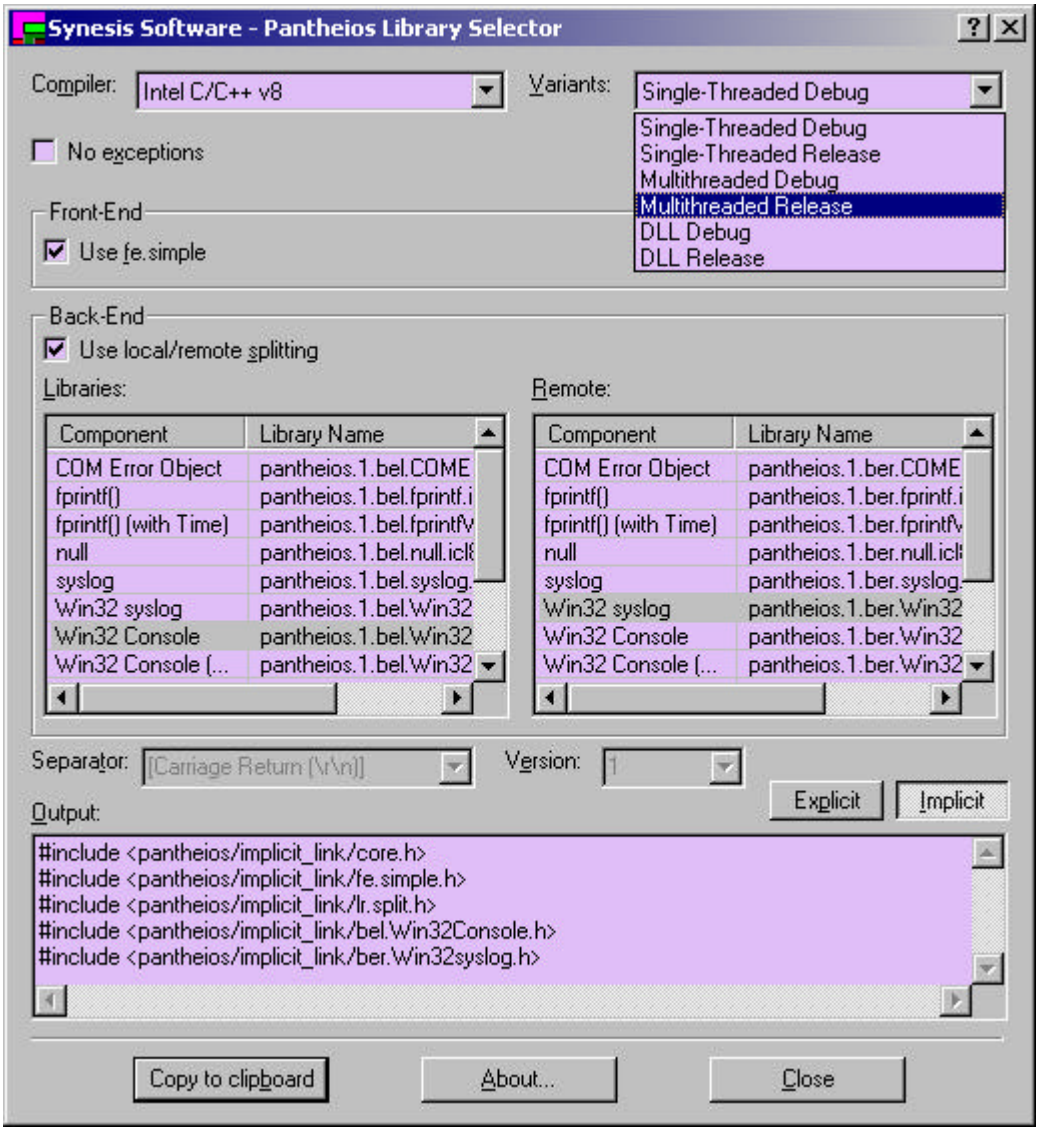


Figure X. Using the library selector for Implicit Linking

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au



Efficiency

The efficiency gains touted earlier in respect of the type-tunnelling, one-time conversion, and at-most one allocation [afforded us by the use of auto_buffer (UP)] are not mere thought experiment. We have conducted tests that demonstrate dramatic performance advantages over the IOStreams, and even over `fprintf()`-family log APIs. We intend to present the details of a thorough analysis at a later time. [As a taster, the initial perf results are tagged on at the end of the article. They'll be expounded soon.]

There are three ways in which this is efficient.

1. The test is before conversion.
2. Uses auto-buffer and manual (albeit algorithmic) string concatenation to avoid memory allocation inside the API unless total string length is > 2048. (This figure can be configured at compile-time.) What this means is that even lengthy and seemingly complex expressions can actually be logged without a single trip to the heap!
3. Use of the string access shims means that the length for each log statement element is calculated exactly once, and for those types (e.g. `std::string`) that already know their length this "calculation" is simply an invocation of the requisite constant-time accessor (i.e. `size()`).

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

4. Since each element's length is known prior to the preparation of the output statement, concatenation of the strings is much faster than any formatting scheme (e.g. `printf()`, IOStreams) where elements are treated heterogeneously by the formatting function.
5. Statement elements do not have to be nul-terminated, or expressed as nul-terminated strings.
6. Back-ends do not need to calculate length or append a nul-terminating character, as both are (known and) provided by the core.
~~6.~~7. The Pantheios core is lock-free, not even using atomic integer operations. Notwithstanding the characteristics of particular back-ends, the output from one thread is essentially independent that of any other.

always know how long each element is

b. Efficiency. In addition to the avoidance of unnecessary costs, when logging *is* carried out, it is extremely efficient. By using the STLSoft `auto_buffer` class template [EVAB] *at most one memory allocation* is carried out by the core; in all practical (i.e non-test) cases we've encountered in our use of Pantheios the use of the heap has been avoided completely. Further, the core passes the prepared log message in the form of a nul-terminated c-style string *and* its length to the back-end API, which means that those transports that need to know length don't have to call `strlen()`. Finally, once initialised, the core is entirely context and lock free, not even requiring calls to atomic integer operations.

Future direction

We believe that we have a solid architecture, and that the balance of efficiency, genericity and type-safety afforded by the use of Type Tunnelling is a long-term winner. However, there are areas where we consider that there may be room for future improvement, and we will continue to explore them. Further, we look forward to feedback from readers on the libraries.

We are currently considering a version 2 of the libraries, which would use advanced template meta-programming (TMP) techniques, such as Type Detection and Type Selection [IIA], to infer the type – string, string-able, integer, floating-point or other – and select the appropriate converter automatically. Naturally, this would have the disadvantages of a reduced compatible compiler population and nastier compiler messages in the cases of TMP errors, but would it make ~~the~~ application code even more succinct and intuitive than it is already.

Cater for wchar_t encoding.

~~We are also considering incorporating a "verbosity" or "facility" parameter into the functions, to enable a two-dimensional filtration of messages. (This may be incorporated by utilising the other 24 bits of the `pan_sev_t` type.)~~ This is now catered for. The ostensible constants, e.g. `pantheios::debug`, `pantheios::alert`, are actually instances of types that have both implicit conversion (to `int`) and a function call operator. The latter allows for succinct use of the other 24-bits of the integer type carrying the severity. What this means is that the following statement:

```
pantheios::log(pantheios::alert, "message");
```

emits a message `"message"` at severity level alert (1), and the following

```
pantheios::log(pantheios::alert(55), "message");
```

emits a message `"message"` at severity level 0x3701. The given argument value is shifted up 8 bits, and may be used by front-end and/or back-end as the user requires. (All stock back-ends are implemented to ignore all but the low 8-bits, which means they're compatible with such uses as any user may wish to put the upper 24-bits.)

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

Pantheios.COM – even before its public release, Pantheios has spawned an adjunct project. Pantheios.COM (which will be available from pantheios.org along with the main project on its release) is a COM logging component that follows the same principle of severity level-test prior to statement aggregation for efficiency. Although an entirely self-contained component – i.e. a single DLL with no non-system dependencies – Pantheios.COM uses several back-end implementations from Pantheios in its logging coclasses.

Summary

We have presented the Pantheios logging API library, and demonstrated how it represents an optimal mix of expressiveness, succinctness, genericity and extensibility, flexibility, efficiency and type-safety. ~~We have shown how this combination of features is a singular~~

We have introduced the stock front and back-end libraries that come with the Pantheios distribution, and shown how Pantheios may be

Because log statement parameters do not undergo conversion until *after* ~~the~~ it is determined whether to emit a message, there is no need on performance grounds for compile-time measures to elide certain message classes. This means that when your system is in production, and something goes wrong, you will still be able to turn on the most verbose output levels~~, and you'll save your job in the process~~.

We have shown how Pantheios can make UNIX-`syslog()` logging a virtual type-safe and efficiency no-brainer. Further, we have shown that Pantheios can be applied to arbitrary logging streams, via stock or custom back-ends.

Pantheios is open-source (available from http://pantheios.sourceforge.net) under the BSD license~~, and, we invite you to use the project and contribute.~~ It's only dependency, ~~on facilities other than~~ over and above the C and C++ standard libraries and ~~the APIs~~ native ~~to UNIX and Win32~~ operating system~~s~~ APIs, is on the STLSoft libraries (http://stlsoft.org, version 1.9.1 beta 18 or later), which provide the `auto_buffer`, String Access Shims, iterator adaptors, and sundry other components. We invite you to use the project and contribute.

Since its development by Synesis Software in 2003/4, it has now become our logging standard. All common dynamic libraries, COM components, Shell extensions (see http://shellext.com), system tools, development tools, plug-ins, and all C++/COM products developed for clients use Pantheios, even (or is that especially) for performance-sensitive servers. It has proven itself in terms of simplicity of use, effectiveness and efficiency many times over in that time. By the time you read this article, the http://pantheios.org website will contain the 1.0.1. distribution, or, at worst, the latest 1.0.1. beta.

About the authors

Matthew Wilson

Matthew is a development consultant for Synesis Software, and creator of the STLSoft libraries. He is author of *Imperfect C++* (Addison-Wesley, 2004), and is currently working on his modicum opus, *Extended STL, volume 1* (to be published in 2006); the internals of Pantheios feature large in volume 1. Matthew can be contacted via http://imperfectcplusplus.com/.

Garth Lancaster

Production manager and all-round technoscenti at MBF, one of Australia's largest medical insurers, Garth is an avid connoisseur of other people's libraries, resorts to writing his own only when it's *really* worthwhile. He can be contacted at Garth.Lancaster@mbf.com.au.

Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au

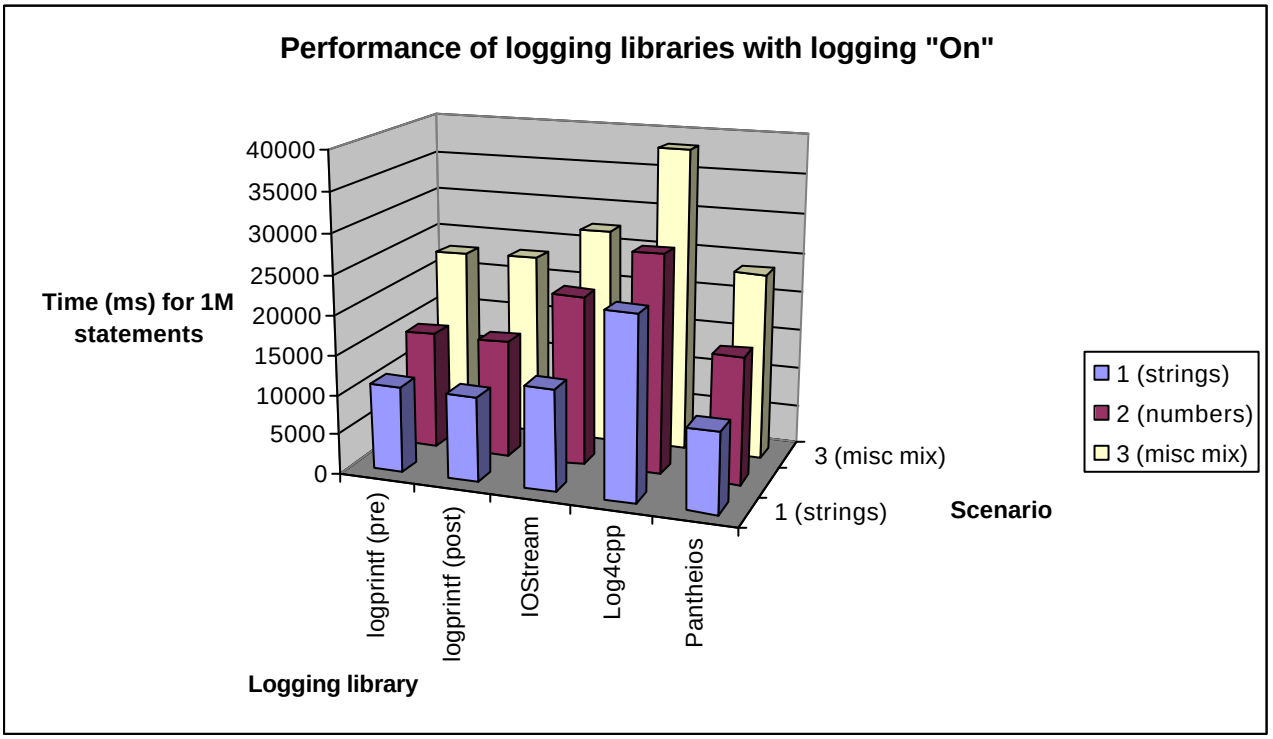
Notes and references

[IC++]
[EVAB] auto_buffer
[String Access Shims] "Generalised String Manipulation"
"Efficient Variable Automatic Buffers"

Performance

Logging Off	logprintf (pre)	logprintf (post)	IOStream	Log4cpp	Pantheios	log4cxx	Boost.Log
1 (strings)	58	1705	3505	30	9		
2 (numbers)	7	5063	10431	26	17		
3 (misc mix)	6903	10199	14578	6946	18		

Logging On	logprintf (pre)	logprintf (post)	IOStream	Log4cpp	Pantheios
1 (strings)	10934	10760	12913	22982	10158
2 (numbers)	14914	14802	21545	27593	15945
3 (misc mix)	22957	23254	27501	38395	23597



Matthew Wilson, Synesis Software,
matthew@synesis.com.au
Garth Lancaster, MBF,
garth.Lancaster@mbf.com.au