# PREVENTING RUN-TIME BUGS AT COMPILE-TIME USING C++[*]

R. Neswold[†], Fermilab, Batavia, IL, USA

*Abstract*

In order for a system to be reliable, its software needs to be carefully designed. Despite our best efforts, however, errors occur and we end up having to debug them. Unfortunately, debugging an embedded system changes its dynamics, making it difficult to find and fix concurrency issues. This paper describes techniques, using C++, making it impossible to write code susceptible to certain run-time bugs. A concurrency library, developed at Fermilab, is used in the examples illustrating these techniques.

## INTRODUCTION

The C++ Standard Template Library (STL) is an impressive body of work, giving programmers a high-level library while still being performant. Although C++ templates started as a way to generalize containers across any type, the STL authors have discovered techniques which allow complex decisions to be made at compile time. We would like to use this feature to enforce constraints on how our code can be written.

When a template is used, its parameters are specified and they, in turn, get substituted in the body creating a specialization of the template. More interesting, if the library defines a version of the template where the types have been picked, the compiler will use that template instead of specializing the default one[1]. This is not inheritance; each specialized version of a template could have a different API, if it was useful.

Template parameters can also take values, which then get used by the template body as constant values. This may seem similar to using `#define`, but it's not; each instantiation of a template is a new, unique type. An `Array<10>` has no relation to an `Array<20>`, as far as the compiler is concerned (unless the template derives them from the same base class.) We can use this uniqueness to our advantage.

Over the past several years, we've developed a library of C++ templates for our VxWork-based systems. These templates make writing our embedded software easier and more reliable. Writing our drivers is easier because the templates generate the mundane, boilerplate code and let us focus on the core of the driver. Our reliability improves because our concurrency library makes most concurrency bugs impossible.

## VERIFYING REQUESTS

When a driver in our embedded system receives a request from the network, part of the request consists of a length value, an offset value, and a buffer of data (for readings, the buffer is to be filled and returned; for settings, the buffer contains data.) Since our controls system supports array devices, the offset could be non-zero. Every driver needs to validate the request before continuing which means both the length and offset values need to be a multiple of the base data size of the device. Plus, combined, they cannot exceed the total size the driver expects. Although this isn't rocket science, it is a tedious part of writing drivers. A further complication resides in the data buffer. ACNET has VAX-centric byte-ordering which is different from our PowerPC systems, so developers were responsible for byte-swapping to and from the network. All these details adds one more layer of debugging when developing, so we felt we needed to fix it.

Proxy objects are used in our library to both verify the request parameters and properly exchange data with the buffer associated with the network. If the request parameters are invalid, the proxy's constructor throws an exception and the remote client gets the appropriate bad length or bad offset error status. Otherwise, the proxy object is created and is used to access the raw buffers, returning the data in a native format.

For instance, if a driver expects an unsigned, 16-bit integer as a setting, it would start its handler with

```
SettingProxy<uint16_t> setting(req);
```

This template expands to two tests (`req` is a pointer to the request structure); the offset must be zero and the length must be 2. It also declares a `uint16_t` typecast operator so that using `setting` in an expression returns a properly byte-swapped, 16-bit integer. The template doesn't define an assignment operator, so the compiler generates an error if the programmer tries to write to the setting (which is always incorrect.)

This template works for all simple data types.

But what if we have an array device? We use template specialization to tell the compiler what to do if it sees an array type. If our driver accepted up to four, 32-bit integers, for instance, it would use this

```
SettingProxy<int32_t[4]> setting(req);
```

Now the test makes sure the length and offset are multiples of 4 bytes and they don't exceed the limits of a four element array of integers. In addition, rather than define a typecast operator, this object defines the subscript operator which returns an `int32_t`.

For completeness, specializations were also defined for pointer and reference types. Since it doesn't make sense to pass pointers or references across the network, these specialized template define an empty class so trying to use them results in compile-time errors.

In the case that a driver returns data to a request, a ReadingProxy is used. This proxy template has the same validation tests as the setting and it, too, recognizes array types. But instead of defining operators that access the

[*] Operated by Fermi Research Alliance, LLC, under Contract No. DE-AC02-07CH11359 with the United States Department of Energy

[†] neswold@fnal.gov

data, this template defines assignment operators, making these objects write-only.

## CORRECT CONCURRENCY

Concurrency is a part of modern systems and adds complexity to a project. If data access isn't serialized properly, shared state may become corrupted. If a code path doesn't release a mutex, other threads will block forever. What makes concurrency issues difficult to diagnose is that the errors that arise from these issues don’t indicate the actual cause. Even more troubling, concurrency issues are resistant to debuggers since the very act of starting and stopping threads change the conditions that caused the problem.

The following example shows some design patterns the author used to precisely describe a concurrency environment so the compiler can enforce correct usage. To illustrate some of the concepts, we'll start with a C example for VxWorks and improve it step-by-step.

```
static SEM_ID mtx;
static int data;

void f()
{
  semTake(mtx, WAIT_FOREVER);
  if (ERROR == someFunction(data)) {
    semGive(mtx);
    return;
  }
  data = anotherFunction();
  semGive(mtx);
  someThirdFunction();
}
```

Figure 1: C Implementation.

The thread API for VxWorks (and POSIX threads, for that matter) are for the C language which means every detail is managed by the programmer. The code in Figure 1 is a short example, but shows some of the complexity: The function, `f()`, locks and unlocks the mutex around the code that accesses the shared data. The programmer also needs to handle the situation when the function returns early, like when `someFunction()` returns `ERROR`, to make sure the mutex is unlocked. Some follow the convention of never allowing early returns in a function, to avoid details like this. Unfortunately, the compiler doesn't enforce conventions so you have to hope future maintainers honor them. In addition, forcing all returns to be at the end of a function can actually complicate the code further - some cases require using goto - just to avoid the details of managing resources. Rather than using conventions or describing the requirements in comments, we'd rather have the compiler enforce them.

The first step in accepting the compiler's help is let it manage locking the mutex. The C++ standard defines an object’s lifetime to strictly be when it goes in and out of

```
static Mutex mtx;
static int data;

void f()
{
  {
    Mutex::Lock lock(mtx);

    if (ERROR == someFunction(data))
      return;
    data = anotherFunction();
  }
  someThirdFunction();
}
```

Figure 2: C++ w/RAII.

scope so let the object own the resource while it exists. This pattern is so common, it has an acronym: RAII‡. We can take the approach used by the Boost library and the C++ thread library and use a lock object to control ownership of the mutex. In Figure 2, the VxWorks API is hidden in our Mutex and Lock objects. The Mutex class doesn't publicly expose any lock/unlock methods. It is purely owned through a lock object. Note the nested braces to end the lock's scope before `someThirdFunction()` is called. This matches the scope of mutex ownership in the previous example. Note also that early returns are correctly handled by the compiler which, incidentally, means this example is exception-safe.

At this point, the code properly manages ownership of the mutex, but that's it; a developer could still call `someFunction()` without owning the mutex. He could even manipulate the data without any serialization! We need a further design pattern to limit access to data only when a lock is held.

Rather than directly accessing the data, a small set of functions can be created which manipulate the state. For instance, a stepper motor controller may need two re-

```
static Mutex mtx;
static int data;

STATUS someFunction(Mutex::Lock const&, int);
int anotherFunction(Mutex::Lock const&);

void f()
{
  {
    Mutex::Lock lock(mtx);

    if (ERROR == someFunction(lock))
      return;
    anotherFunction(lock);
  }
  someThirdFunction();
}
```

Figure 3: Requiring locks.

‡ Resource Acquisition Is Initialization

gisters be modified to set the number of steps and direction. Both register accesses can be placed in a single function, forming a primitive that moves the stepper motor.

The set of functions are made public and become the "primitives" which update the state. We can get the compiler's help to ensure the mutex is held by adding a (unused) parameter, as shown in Figure 3. Since the function specifies a lock needs to be passed, the compiler will require a lock to be passed — even though it's unused — which means the mutex is guaranteed to be owned during the function's execution.

This third iteration introduces several improvements. The “primitives" are now composable. They can be called safely in any order because we know the mutex is held the entire time. Also, since these functions know they are executing in an environment with a locked mutex, they don't do any locking of their own so there's no reliance on using "recursive mutexes”.

Is there any further improvement to be made? In the library used at Fermilab, we have made one further improvement. C++ templates take parameters which are, typically, type names. But they can also be constants such as integers and *addresses that can be resolved at link time*. So where the C++ and Boost libraries use the template parameter to specify the type of mutex the lock locks (recursive mutex, semaphore, etc.), we use it to specify *which* mutex to lock.

Figure 4 shows the last improvement. A `typedef` is used to simplify the lock's type. In this final form, the functions not only specify a mutex needs to be held, but it also specifies *which* mutex needs to be held!

It should be mentioned that a third type of constant template parameter is possible, the offset of a field in an object. Our library also supports this form of lock so object methods can require an object’s mutex be held

```
Mutex mtx;
static int data;

typedef Mutex::Lock<&mtx> LockType;

STATUS someFunction(LockType const&, int);
int anotherFunction(LockType const&);

void f()
{
  {
    LockType lock;

    if (ERROR == someFunction(lock))
      return;
    anotherFunction(lock);
  }
  someThirdFunction();
}
```

Figure 4: Enforcing which mutex to use.

## *Benefits*

A typical function is more complicated than the example given here. You could imagine how several threads, using code written in the C-style, could run into concurrency issues. Designing with the style of the final example eliminates nearly all concurrency-related bugs:

- The compiler perfectly tracks object lifetimes, so locks are always freed properly.
- Locks templates only work with the correct mutex, so you can’t lock the wrong mutex
- Functions specifying (unused) lock parameters declaring the environment in which they need to work correctly.

## *Performance*

C++ templates are found in header files, so when using templates, the compiler usually has all the source code to instantiate them. The compiler is, therefore, able to aggressively inline the source code. As a result, the small functions that take the unused lock parameter get completely inlined and there is no run-time penalty for having the extra parameter. On our VxWorks 6.7 compiler (GNU C++ 4.1), the assembly output between the first, C example and the final version was essentially identical. In other words, the extra notation that’s used to verify correct concurrency usage still results in code nearly identical to hand-written C code.

## *Improvements*

We’re continuing to look for new ways to enhance our libraries to make developing easier.

One experimental feature is using templates to associate a variable with a mutex. This would remove the need of functions using a lock as a parameter since the variables themselves would require proof that a lock is held. The current implementation has a heavy syntax, so we’re trying to find ways to clean it up. Also, containers aren’t handled well (someone could use a reference to an element in an array after they released the mutex associated with the array.) Hopefully we’ll be able to fix these deficiencies.

## CONCLUSION

Using templates in C++ is still cumbersome; template syntax isn’t intuitive, experience is required to get comfortable designing libraries with them, and error messages are difficult to understand. However, the benefits are worth the effort. Template specialization allows the compiler to select the best template for the job, allowing optimal code to be generated and the most appropriate API be defined.

We found that, with these techniques, we’re able to tell the compiler what needs to be done, not how to do it.

## REFERENCES

[1] Vandevoorde, David, and Nicolai M. Josuttis. C++ Templates: The Complete Guide. Boston, MA: Addison-Wesley, 2011, pp. 200-203.