# Using "Context Types" in C++

stephan beal <stephan@s11n.net>

14th June 2004

**Abstract**

CVS Info: $Id: context_types.lyx,v 1.4 2004/06/14 14:33:21 stephan Exp $

This paper covers the use of type-based contexts in C++. It is targetted at intermediate-to-advanced C++ programmers.

The entire content of this article is released into the Public Domain.

## 1 Introduction

Most programmers are familiar with the concept of resource handles. To recap: what is a resource handle? It is a unique identifier which identifies some type of resource within a system. The most common example is probably C's classic file handles, which one uses to pass to functions like `fclose()`, so that `fclose()` knows which file it is really supposed to close. Handles come in a variety of flavours, but the are invariably used to define the context within which a given set of operations is performed. e.g., `fclose(myhandle)` instructs `fclose()` to only close the file associated with myhandle, and not another file.

This paper takes the concept of resource handles a step further, defining "type handles", or what we will refer to as Context Types. In short, a context type is an arbitrary type which is used to limit certain operations to a scope - the scope of the given context. Clear as mud? The rest of this paper will try to make clear what exactly context types are, how to implement them, and how they can be used to add flexibility to some otherwise-inflexible code.

C++'s template facilities offer a different type of handle, vaguely related to classic resource handles but based, instead of on runtime-mutable values, on *types*.

## 2 Our first context-dependent function

Jumping right in... here we show a trivial example of how type-based contexts can be used to genericize a very simple function. Consider the following function, who's sole purpose is to dole out a new number each time it is called:

```
size_t next_id() {

    static size_t bob;
    return ++bob;

}
```

Seems straightforward enough, right? Now we're going to make a trivial change to make this function dole out unique IDs based on contexts. Rather than explain what we're going to do, the one-line code difference says it all:

```
template <typename ContextType>
size_t next_id() {
    static size_t bob;
    return ++bob;
}
```

Let's quickly go over the implications of this one-line change:

- Most obviously, the calling syntax of our function now changes from `next_id()` to `next_id<T>()`.
- We no longer have a single `next_id()` function, but, as Alexandrescu has put it, a *family* of `next_id()` functions.
- Every time this function is called with the same context a new number will be generated. That is, calls to `next_id<X>()` and `next_id<Y>()` may very well generate identical values, but call calls to `next_id<X>()` will get unique values (until `size_t` overflows, obviously).
- More subtley, `ContextType` is never instantiated. This means that we can pass arbitrarily large types as a context without any worries about using up system resources or violating constraints such as the Singleton-ness of a given ContextType.

i can almost hear you saying, ”*so what?!?!*” Indeed, so what? This change is so simple and straightforward that it almost seems silly to write a paper about it. The facts are, however:

1. i have never seen this approach used outside of my own projects.
2. i get a tremendous amount of use out of this approach. So much so that i believe other programmers could get some use out of it.

(i say ”approach” because i hate to call it an ”idiom” at this point, though perhaps it could rightfully be called one.)

The above simple example only touches on the potential uses of context types. We'll spend the rest of the paper trying to convince you that context types are useful in a variety of ... well, *contexts*.

We said above that context types are conceptually similar to resource handles. How is that so? Consider: when calling next_id() with different context types we are essentially giving the function a different resource handle, i.e. a different context in which to perform it's work. In doing so we assure that next_id<X>() and next_id<Y>() do not interfere with each other.

## 3 Context-dependent Types

Let's consider this classic example of implementing what are often dubbed Meyers Singletons, after the programming god Scott Meyers, who reportedly first published this approach:

```
template <typename T>
struct MeyersSingleton {
    typedef T singleton_type;
    singleton_type & instance() {
            static singleton_type bob;
            return bob;
        }
    };
```

Pretty straightforward. Now, as we did with the next_id() function, we're now going to make a minor change:

```
    template <typename T, typename ContextType = T>
    struct MeyersSingleton {
        ... same implementation as above ...
    };
```

As before, we haven't had to change the internals of our code, but the implications of this subtle change anything but subtle. We now have what i often call "context singletons":

```
    MyType & m1 = MeyersSingleton<MyType>::instance();
    MyType & m2 = MeyersSingleton<MyType,int>::instance();
```

Those two calls return different objects. Rather than use a POD, such as int, to define a context it is often useful to define no-op marker classes, such as this one:

```
    struct my_sharing_context {};
```

Using this approach we can more easily control which contexts are used (as opposed to inadvertently abused) by other code. Keep in mind that typedefs do not define new types, so the following does not define a new context:

```
    typedef int my_sharing_context;
```

## 4 In closing

This paper has only briefly covered a small number of uses for context types. They have a wide variety of uses not covered here, and i'm quite certain that anyone who has found this paper worth reading will be able to quickly come up with their own ideas for utilizing context types in their projects.

This paper os a more general refinement of a larger (and older) paper on this topic, entitled Context Singletons, available from:

http://s11n.net/papers/

That paper extends the idea of Context Singletons, providing, e.g., a client-agnostic way of populating shared objects upon their first instantiation via a functor.