Supermacros:
powerful, maintainable preprocessor macros in C++

stephan beal <stephan@s11n.net>

22nd August 2004

Abstract

CVS info: $Id: supermacros_cpp.lyx,v 1.2 2004/08/22 19:18:34 sgbeal Exp $

This paper discusses some uses for so-called "supermacros" in C++. Supermacros are header files which are used much in the same ways as traditional preprocessor macros, but have a lot more flexibility and power, and do not suffer from some of the more notorious problems which classical macros do.

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

Change History:

22 Aug 2004: Minor text corrections.

13 Aug 2004: Initial publication.

1 Introduction

During your C or C++ career you've probably heard several different takes on the religious wars surrounding C's preprocessor, in particular the use (and abuse) of macros. While i believe i can safely say that most experienced programmers abhor macros, i think i can also safely say that most programmers understand that macros are, like it or not, basically a necessary part of C and C++ developement.

Granted, nobody really *loves* macros, but when they're useful, they're *really useful*, and we would have much more difficult programming careers without them. Well, that's not 100% true - we'd simply have to start pre-filtering our source files with, e.g., Perl, before compiling them. While that option does exist, is arbitrarily flexible, and is practiced in many cases, it is of course not platform-portable, nor portable across build environments, and is not an in-language approach. Some purists would go so far as to say that any code generation, except possibly purely in-language techniques such as class templates (which, as others wiser than i have pointed out, *are* code generators) has no good place in C++ projects. i won't go quite that far, but i must admit that i shy away from code generation when practical, if for no other reason than the interest of build portability. But that's not what we're here to talk about...

Some would argue that macros themselves are not technically an in-language feature, and they may be correct, but macros are such a part of the C and C++ environment that they can be considered, for our purposes, as being inseparable from C++ development, and therefor are considered here to be an in-language mechanism.

This paper focuses on one of macros' more glaring weaknesses: the difficulty of physically maintaining large macros, particularly those which span more than a few lines. Here we will explore what i call "supermacros", and we will see how to add them to our own projects. They serve the same purpose as classical macros, but have some fundamental differences, some of which we explore in this paper.

1.1 Background

Historically, macros have been used to handle everything from defining types to adding member functions to types to refering to global data to wrapping up function calls (e.g., C's venerable `assert()` macro). Despite the general utility of macros, they have a number of limitations and "gotchas", some of which are:

- They are easy to abuse, used for things which are much better accomplished using more traditional, in-language C++ mechanisms. (IMO, the `assert()` macro is one of these abuses, as it does nothing if `NDEBUG` is set (even if it is set to a logically-false value like 0), and we normally learn this the hard way after realizing that our `asserts()` suddenly don't work any more!)

- They are notoriously difficult to edit once they get more than a couple of lines in length, as each end-of-line in the macro must be backslash-escaped.
- For big macros, getting useful error diagnostics from the compiler is essentially impossible, as macros technically live on a single line of code, no matter whether they are one line or 25. The main implication of this is that error line numbers from the compiler are almost certainly incorrect, and may inadvertently point the user at some *completely* different part of the code. To be clear, i mean "incorrect" in the sense of, "not what the user would expect," not "technically wrong."
- They are difficult to publically document, as some documentation generation systems do not process documentation for macros.

While i would like to add "poor style" to that list, that's largely a matter of taste, and can't really be claimed as a point for getting developers to avoid them.

This particular paper evolved from the libs11n project (http://s11n.net), where supermacros are used to:

- Register types with their classloader.
- Install user-supplied classes or preferences into the framework.
- Create class template specializations from a common code snippet, replacing only certain elements. i.e., a class template template.
- Map type names to stringified forms of their names (something which C++ does not internally support).

Disclaimer:

i cannot claim to have come up with the technique of using supermacros - i am quite certain that many developers have used a similar technique over the years, even if they haven't called them "supermacros." However, i have never personally seen this technique used in source trees other than that of the s11n project, and so this paper is intended to help prod the imagination of other C++ users who might get some use out of this mechanism.

2 Supermacro primer and mini-HOWTO

A supermacro is a header file which is written to work like a C++ macro, which essentially means that it is designed to be included, potentially repeatedly, and "passed parameters."

Potential uses of supermacros include:

- Register types with, e.g., a classloader.
- Install classes within the source tree at compile-time. e.g., libs11n uses supermacros to install user-extended specializations of a number of class templates.
- Set up code which is to be run when static/global data is initialized by the runtime environment.
- Generate "similarly-typed" helper/back-end classes in such a way as to avoid ODR (One Definition Rule) violations.

Supermacros allow some - er... *TONS* - of features which classical macros simply cannot come close to providing. The following list of benefits comes to mind:

- A supermacro can handle anything a normal macro can, plus many more cases, using a single - yet extendable - interface.
- Supermacro arguments can contain characters, like commas, which break conventional macros.
- Supermacros can do arbitrary tasks, like classloader registration.
- Arbitrary new sets of supermacros can be introduced at any time without impacting existing code, which means, for example, client code can use a `#define` to swtich between interfaces by including different registration macros.

- When using macros to generate types, ODR violations can be more easily eliminated with supermacros (in theory, completely). (This was one of the major drivers behind libs11n's move from classical macros to supermacros for classloader registrations.)
- As they are implemented in "real header code", they are completely immune to the editing-related limitations of macros, and are simply *much* easier to maintain.
- Supermacros can be arbitrarily large, wheres macros get very tedious to edit once they are longer than a few lines.
- They are *much, much* easier to debug when something doesn't compile: unlike conventional macros, we even get proper file names and line numbers (*yes!!!!*).
- Supermacros can accept any number of arguments without modifying their interface, and can do some degree of sanity checks on parameters. They can, e.g., provide useful error messages when a user fails to set a specific parameter.
- Parameters passed to supermacros are named, not positional.

2.1 Calling supermacros

Use of a supermacro looks something like this:

```
#define MYARG1 "some string"
#define MYARG2 foo::AType
#include "my_supermacro.hpp"
```

Here we are "passing" two "parameters" or "arguments" to `my_supermacro.hpp`, named `MYARG1` and `MYARG2`.

By convention, and for client convenience, the supermacro is responsible for unsetting any arguments it uses after it is done with them (even if it does not use them on a given invocation), so client code may repeatedly call the macro without `#undef`'ing them.

Sample:

```
#define S11N_TYPE MyType
#define S11N_TYPE_NAME "MyType"
#define S11N_SERIALIZE_FUNCTOR MyType_s11n
#include <s11n.net/s11n/reg_serializable_traits.hpp>
#define S11N_TYPE MyOtherType
#define S11N_TYPE_NAME "MyOtherType"
#define S11N_SERIALIZE_FUNCTOR MyOtherType_s11n
#include <s11n.net/s11n/reg_serializable_traits.hpp>
```

2.2 Sample Supermacro

Here we show a small supermacro. Keep in mind that supermacros can do anything which normal header files can do, and thus can be of arbitrary size and complexity. One particular aspect of Supermacros which differs from standard header files is that they typically do not have a so-called "include guard", as they are intended to be included multiple times. Because of this, care must be taken to ensure that all required parameters "passed" to a supermacro are set up before the macro is used and undefined before the macro ends (i.e., before the end of the file).

For this example we are going to steal some code from the s11n project. In that project we have a mechanism for mapping types to their type *names*, something which C++, very unfortunately, does not provide. (Don't *even* start on about typeid::name(), because it *officially* provides undefined behaviour.) To this end we have a class, which is not part of a supermacro, which looks like the following:

```
namespace { // anon namespace required for this particular case
```

```
    template <class T> struct class_name
    {
        typedef T value_type;
        static const char * name()
        {
            static const std::string tid = typeid((value_type *)0).name();
            return tid.c_str();
        }
    };
} // end anonymous namespace
```

This class acts as our default implementation, tucked away in it's own header file, but doesn't provide a behaviour we can rely on, so clients are expected to specialize it for their types. To do this, we use a supermacro, which is used like so:

```
#define NAME_TYPE FooT
#define TYPE_NAME "FooT"
#include <s11n.net/name_type/name_type.hpp>
```

The `name_type.hpp` supermacro installs various specializations of the `class_name<>` type. It looks something like this:

```
#ifndef NAME_TYPE
# error "You must set both NAME_TYPE and TYPE_NAME before including this supermacro."
#endif // NAME_TYPE
#ifndef TYPE_NAME
# error "You must set both NAME_TYPE and TYPE_NAME before including this supermacro."
#endif // TYPE_NAME
#include "class_name.hpp" // import base class_name<> implementation
namespace { // anon namespace important for our particular case
    template <> struct class_name< NAME_TYPE >
    {
        static const char * name() { return TYPE_NAME; }
    };
    template <> struct class_name< NAME_TYPE * >
    {
        static const char * name() { return TYPE_NAME; }
    };
} // anonymous namespace
#undef NAME_TYPE
#undef TYPE_NAME
```

The above example is fairly short and straightforward, but supermacros can be arbitrarily complex.

Things to note about the supermacro:

- There is no "include guard", as is normaly used in header files. Sometimes *parts* of a supermacro require one, however.
- Supermacros make use of traditional macros for "passing of parameters", as it were, and these parameters are named, not positional.

- The supermacro checks it's "parameters" before it starts, to avoid *really* weird (and potentially *really* long) error messages when a user fails to set a specific parameter. This can also be used to, e.g., support backwards compatibility by re-assigning macros as needed. libs11n uses this, e.g., to set the `S11N_TYPE_NAME` macro if the user has used it's older (deprecated) variant, `S11N_TYPE`.
- The undefining of all parameters at the end of the macro, to allow easy multiple-inclusion by clients.
- Supermacros can call each other, to arbitrarily deep levels. When a given supermacro is made obsolete, it is often feasible to re-implement it to forward the "call" to another supermacro (presumably the obsolete one's replacement). This affords clients some degree of backwards compatibility, for example.
- It's "logical interface" can often be changed without changing existing client-side calls to the macro.
- Can effectively act as an "overloaded" macro, taking a variable number of arguments, using default values for those which a user does not pass in.
- For reasons too complex to go into here, anonymous namespaces are often useful for isolating types and preventing ODR violations. This is not specifically a trait of supermacros, but in my experience anonymous namespaces are very useful in many supermacros, especially when type generation is needed and ODR violations are a problem. This is especially true, over classic macros, because supermacros can selectively guard sections of their code from multiple inclusion.

3 Conclusion

That's about all there is to it! So where's the catch? There is none, really. Granted, supermacros have a more verbose calling convention than normal macros, but that is, as far as i'm concerned, the only down-side to using them over their more primitive cousins. Given their *much* greater flexibility, the verbosity is a cost which i, as a developer, happily pay.

Supermacros do not completely replace traditional macros and, as we've seen here, makes use of them to do it's own work. They are not suitable for many cases which standard macros are, such as acting as function call wrappers. Thus i will not for a moment propose that C++ coders should (or could) eliminate the use of macros altogether. i will, however, suggest that supermacros can fit many roles much better than standard macros, and that they are worthy of consideration whenever non-trivial, macro-driven functionality is needed.

The s11n library uses supermacros which create behind-the-scenes template specializations for performing tasks like registering types with their classloaders and registering i/o handlers with the framework. Readers wishing to examine those are referred to the files `src/s11n/reg_*.hpp` in the s11n source tree, available from that project's download page: `http://s11n.net/download/`

Thanks for taking the time to read this paper. If you enjoyed it, or would like to feed back on it, please feel free to contact the author via the address at the top of this article. (It's always nice to get a mail saying someone's read what one has written. :)