

Note: the text on this page was almost integrally written by Niall Douglas, the original author of the patch, and placed on nedprod.com. This is basically a local mirror (especially useful because the external website now appears to be down).

Why is the new C++ visibility support so useful?

Put simply, it hides most of the ELF symbols which would have previously (and unnecessarily) been public. This means:

- *It very substantially improves load times of your DSO (Dynamic Shared Object).* For example, a huge C++ template-based library which was tested (the TnFOX Boost.Python bindings library) now loads in eight seconds rather than over six minutes!
- *It lets the optimiser produce better code.* PLT indirections (when a function call or variable access must be looked up via the Global Offset Table such as in PIC code) can be completely avoided, thus substantially avoiding pipeline stalls on modern processors and thus much faster code. Furthermore when most of the symbols are bound locally, they can be safely elided (removed) completely through the entire DSO. This gives greater latitude especially to the inliner which no longer needs to keep an entry point around "just in case".
- *It reduces the size of your DSO by 5-20%.* ELF's exported symbol table format is quite a space hog, giving the complete mangled symbol name which with heavy template usage can average around 1000 bytes. C++ templates spew out a huge amount of symbols and a typical C++ library can easily surpass 30,000 symbols which is around 5-6Mb! Therefore if you cut out the 60-80% of unnecessary symbols, your DSO can be megabytes smaller!
- *Much lower chance of symbol collision.* The old woe of two libraries internally using the same symbol for different things is finally behind us with this patch. Hallelujah!

Although the library quoted above is an extreme case, the new visibility support reduced the exported symbol table from > 200,000 symbols to less than 18,000. Some 21Mb was knocked off the binary size as well!

Some people may suggest that GNU linker version scripts can do just as well. Perhaps for C programs this is true, but for C++ it cannot be true - unless you laboriously specify each and every symbol to make public (and the complex mangled name of it), you must use wildcards which tend to let a lot of spurious symbols through. And you

have to update the linker script if you decide to change names to the classes or the functions. In the case of the library above, the author couldn't get the symbol table below 40,000 symbols using version scripts. Furthermore, using linker version scripts doesn't permit GCC to better optimise the code.

Windows compatibility

For anyone who has worked on any sizeable portable application on both Windows and POSIX, you'll know the sense of frustration that non-Windows builds of GCC don't offer an equivalent to `__declspec(dllexport)` i.e. the ability to mark your C/C++ interface as being that of the shared library. Frustration because good DSO interface design is just as important for healthy coding as good class design, or correctly opaquing internal data structures.

While the semantics can't be the same with Windows DLL's and ELF DSO's, almost all Windows-based code uses a macro to compile-time select whether `dllimport` or `dllexport` is being used. This mechanism can be easily reused with this patch so adding support to anything already able to be compiled as a Windows DLL is literally a five minute operation.

Note: The semantics are not the same between Windows and this GCC feature - for example, `__declspec(dllexport) void (*foo)(void)` and `void (__declspec(dllexport) *foo)(void)` mean quite different things whereas this generates a warning about not being able to apply attributes to non-types on GCC.

Still not convinced?

A further reading on the subject of good DSO design is [this article](#) by Ulrich Drepper (lead maintainer of GNU glibc).

How to use the new C++ visibility support

In your header files, wherever you want an interface or API made public outside the current DSO, place `__attribute__((visibility("default")))` in struct, class and function declarations you wish to make public (it's easier if you define a macro as this). You don't need to specify it in the definition. Then, alter your make system to pass `-fvisibility=hidden` to each call of GCC compiling a source file. If you are throwing exceptions across shared object boundaries see the section "Problems with C++ exceptions" below. Use `nm -C -D` on the outputted DSO to compare before and after to see the difference it makes.

Some examples of the syntax:

```
#if defined _WIN32 || defined __CYGWIN__
#ifdef BUILDING_DLL
#ifdef __GNUC__
#define DLL_PUBLIC __attribute__((dllexport))
#else
#define DLL_PUBLIC __declspec(dllexport) // Note: actually gcc
seems to also supports this syntax.
#endif
#else
#ifdef __GNUC__
#define DLL_PUBLIC __attribute__((dllimport))
#else
#define DLL_PUBLIC __declspec(dllimport) // Note: actually gcc
seems to also supports this syntax.
#endif
#endif
#define DLL_LOCAL
#else
#if __GNUC__ >= 4
#define DLL_PUBLIC __attribute__((visibility ("default")))
#define DLL_LOCAL __attribute__((visibility ("hidden")))
#else
#define DLL_PUBLIC
#define DLL_LOCAL
#endif
#endif
#endif

extern "C" DLL_PUBLIC void function(int a);
class DLL_PUBLIC SomeClass
{
    int c;
    DLL_LOCAL void privateMethod(); // Only for use within this DSO
public:
    Person(int _c) : c(_c) { }
    static void foo(int a);
};
```

This also helps producing more optimised code: when you declare something defined outside the current compilation unit, GCC cannot know if that symbol resides inside or outside the DSO in which the current compilation unit will eventually end up; so, GCC must assume the worst and route everything through the GOT (Global Offset Table) which carries overhead both in code space and extra (costly) relocations for the dynamic linker to perform. To tell GCC a class, struct, function or variable is defined within the current DSO you must specify hidden visibility manually within its header file declaration (using the example above, you declare such things with `DLLLOCAL`). This causes GCC to generate optimal code.

But this is of course cumbersome: this is why `-fvisibility` was added. With

`-fvisibility=hidden`, you are telling GCC that every declaration not explicitly marked with a visibility attribute has a hidden visibility. And like in the example above, even for classes marked as visible (exported from the DSO), you may still want to mark e.g. private members as hidden, so that optimal code will be produced when calling them (from within the DSO).

To aid you converting old code to use the new system, GCC now supports also a `#pragma GCC visibility` command:

```
extern void foo(int);  
#pragma GCC visibility push(hidden)  
extern void someprivatefunc(int);  
#pragma GCC visibility pop
```

`#pragma GCC visibility` is stronger than `-fvisibility`; it affects extern declarations as well. `-fvisibility` only affects definitions, so that existing code can be recompiled with minimal changes. This is more true for C than C++; C++ interfaces tend use classes, which are affected by `-fvisibility`.

Lastly, there's one other new command line switch: `-fvisibility-inlines-hidden`. This causes all inlined class member functions to have hidden visibility, causing significant export symbol table size & binary size reductions though not as much as using `-fvisibility=hidden`. However, `-fvisibility-inlines-hidden` can be used with no source alterations, unless you need to override it for inlines where address identity is important either for the function itself or any function local static data.

Problems with C++ exceptions (please read!)

Exception catching of a user defined type in a binary other than the one which threw the exception requires a typeinfo lookup. **Go back and read that last statement again.** When exceptions start mysteriously malfunctioning, the cause is exactly this one!


Just like functions and variables, types that are thrown between multiple shared objects are public interfaces and must have default visibility. The obvious first step is to mark all types throwable across shared object boundaries always as default visibility. You must do this because even if (e.g.) the exception type's implementation code lives in DLL A, when DLL B throws an instance of that type, the catch handler in DLL C will look for the typeinfo in DLL B.

However, this isn't the full story - it gets harder. Symbol visibility is "default" by default but if the linker encounters just one definition with it hidden - just one - that

typeinfo symbol becomes permanently hidden (remember the C++ standard's ODR - one definition rule). This is true for all symbols, but is more likely to affect you with typeinfos; typeinfo symbols for classes without a vtable are defined on demand within each object file that uses the class for EH and are defined weakly so the definitions get merged at link time into one copy.

The upshot of this is that if you forget your preprocessor defines in just one object file, or if at any time a throwable type is not declared explicitly public, the `-fvisibility=hidden` will cause it to be marked hidden in that object file, which overrides all the other definitions with default visibility and causes the typeinfo to vanish in the outputted binary (which then causes any throws of that type to cause `terminate()` to be called in the catching binary). Your binaries will link perfectly and appear to work correctly, even though they don't.

While it would be lovely to have a warning for this, there are plenty of legitimate reasons to keep throwable types out of public view. And until whole program optimisation is added to GCC, the compiler can't know which throws are caught locally.

The same issue can arise with other  vague linkage entities such as static data members of a class template. If the class has hidden visibility, the data member can be instantiated in multiple DSOs and referenced separately, causing havoc.

This issue also shows up with classes used as the operand of `dynamic_cast`. Make sure to export all such classes.

Step-by-step guide

The following instructions are how to add full support to your library, yielding the highest quality code with the greatest reductions in binary size, load times and link times. All new code should have this support from the beginning! And it's worth your while especially in speed critical libraries to spend the few days required to implement it fully - it's a once off investment of time with nothing but good resulting forever more. You can however add basic support to your library in far less time though it is not recommended that you do so.

- Place something along the lines of the following code in your master header file (or a specific header that you will include everywhere). This code is taken from the aforementioned TnFOX library:

```
// Generic helper definitions for shared library support
#if defined _WIN32 || defined __CYGWIN__
```

```

#define FOX_HELPER_DLL_IMPORT __declspec(dllimport)
#define FOX_HELPER_DLL_EXPORT __declspec(dllexport)
#define FOX_HELPER_DLL_LOCAL
#else
    #if __GNUC__ >= 4
        #define FOX_HELPER_DLL_IMPORT __attribute__((visibility
("default")))
        #define FOX_HELPER_DLL_EXPORT __attribute__((visibility
("default")))
        #define FOX_HELPER_DLL_LOCAL __attribute__((visibility
("hidden")))
    #else
        #define FOX_HELPER_DLL_IMPORT
        #define FOX_HELPER_DLL_EXPORT
        #define FOX_HELPER_DLL_LOCAL
    #endif
#endif

// Now we use the generic helper definitions above to define FOX_API
and FOX_LOCAL.
// FOX_API is used for the public API symbols. It either DLL imports
or DLL exports (or does nothing for static build)
// FOX_LOCAL is used for non-api symbols.

#ifndef FOX_DLL // defined if FOX is compiled as a DLL
    #ifndef FOX_DLL_EXPORTS // defined if we are building the FOX DLL
(instead of using it)
        #define FOX_API FOX_HELPER_DLL_EXPORT
    #else
        #define FOX_API FOX_HELPER_DLL_IMPORT
    #endif // FOX_DLL_EXPORTS
    #define FOX_LOCAL FOX_HELPER_DLL_LOCAL
#else // FOX_DLL is not defined: this means FOX is a static lib.
    #define FOX_API
    #define FOX_LOCAL
#endif // FOX_DLL

```

Obviously, you may wish to replace the FOX with a prefix suiting your library, and for projects also supporting Win32, you'll find a lot of the above familiar (you can reuse most of your Win32 macro machinery to also support GCC). To explain:

- If `_WIN32` is defined (as is automatic when building for Windows, even for 64-bit systems):
- If `FOX_DLL_EXPORTS` is defined, we are building our library and symbols should be exported. So you would define `FOX_DLL_EXPORTS` in the build system that builds the FOX DLL. Something ending with `_EXPORTS` is defined by MSVC by default in all IDE projects (ditto CMake default, see [CMake Wiki BuildingWinDLL](#)).
- If `FOX_DLL_EXPORTS` is not defined (as is the case for clients using the library),

we are importing the library and symbols should be imported.

- If `_WIN32` is not defined (as is the case when building for Unix with GCC):
- If `__GNUC__ >= 4` is true, then it means the compiler is GCC version 4.0 or later, and hence supports the new features.
- For every non-templated non-static function definition in your library (both headers and source files), decide if it is publicly used or internally used:
- If it is publicly used, mark with `FOX_API` like this:
`extern FOX_API PublicFunc()`
- If it is only internally used, mark with `FOX_LOCAL` like this:
`extern FOX_LOCAL PublicFunc()` Remember, static functions need no demarcation, nor does anything which is templated.
- For every non-templated class definition in your library (both headers and source files), decide if it is publicly used or internally used:
- If it is publicly used, mark with `FOX_API` like this: `class FOX_API PublicClass`
- If it is only internally used, mark with `FOX_LOCAL` like this:
`class FOX_LOCAL PublicClass`
Individual member functions of an exported class that are not part of the interface, in particular ones which are private, and are not used by friend code, should be marked individually with `FOX_LOCAL`.
- In your build system (Makefile etc), you will probably wish to add the `-fvisibility=hidden` and `-fvisibility-inlines-hidden` options to the command line arguments of every GCC invocation. Remember to test your library thoroughly afterwards, including that all exceptions correctly traverse shared object boundaries.

If you want to see before and after results, use the command `nm -C -D <library>.so` which lists all exported symbols in demangled form.

None: Visibility (2016-10-12 23:28:26由CarlosODonell编辑)