Change Logs

2016.06.09 - Initial discussion with CMake team. Scikit-build issue #37 created.

2016.06.28 - Finalized initial version of opadron/weak-linking-demo

2016.06.29 - Draft rev1 presented to CMake team

2016.06.29 - Based on Brad K. comments, we are revisiting our initial approach to use generator expression like $<LINK_WEAK:somelib>

2016.06.30 - Re-organized document adding “Proposal rev2” section

2016.07.25 - Added “Proposal rev3” section + send email to CMake core dev

Proposal rev3

Instead of adding new option to target_link_libraries, a dedicated CMake module (based on prior work of Bradley Lowekamp) has been created.

See https://github.com/scikit-build/scikit-build/blob/master/skbuild/resources/cmake/targetLinkLibrariesWithDynamicLookup.cmake

Proposal rev2

TBD

weak-linking-demo could be updated to output the command for the different scenario. That would address the comment "*proposal should show examples of how the actual link command lines will look in each case (and for each type of platform support).*”

CMake sources related to TargetLinkLibraries:

- Documentation
  - https://github.com/Kitware/CMake/blob/master/Tests/RunCMake/set_property/LINK_LIBRARIES.cmake
  - https://github.com/Kitware/CMake/blob/master/Help/command/link_libraries.rst

- Platform description (e.g Linux-GNU-CXX)
  - https://github.com/Kitware/CMake/blob/master/Modules/Platform/Linux-GNU-CXX.cmake
  - https://github.com/Kitware/CMake/blob/master/Modules/Platform/Linux-GNU.cmake
  - https://github.com/Kitware/CMake/blob/master/Modules/Platform/GNU.cmake
- Implementation
  - cmComputeLinkInformation.cxx
  - cmTarget.cxx
  - cmTargetLinkLibrariesCommand.cxx
  - cmGeneratorExpressionDAGChecker.cxx
  - cmGeneratorTarget.cxx

>

Proposal rev1

To support weak-linking in CMake, we propose:

- To introduce a new property for binary targets, "SYMBOL_RESOLUTION", that would indicate whether undefined symbols are allowed to be resolved at load time (SYMBOL_RESOLUTION=DYNAMIC). Default: "LINK".

- To introduce another new property for binary targets, "WEAK_LIBS", that would maintain the set of all dependency libraries that are not meant to actually be included in the linker command.

- To introduce a new keyword to the target_link_libraries CMake command, "WEAK", that would apply to every library specified after it in the argument list. Internally, this keyword would have the effect of amending the target's "WEAK_LIBS" property with any specified dependency libraries that are not already there -- as well as setting the target's "SYMBOL_RESOLUTION" property to "DYNAMIC".

- To introduce a new system introspection routine (i.e.: try_compile/try_run) that would determine if the linker supported weak linking. (See weak-linking-demo for a reference implementation.)

- To introduce another new system introspection routine (i.e.: try_compile/try_run) that would determine if at run-time, the loader could de-duplicate identical symbols that have been duplicated across link boundaries (e.g.: Linux). (See weak-linking-demo for a reference implementation.)

    - In the case where the platform linker does not support weak-linking, but the run-time loader can de-dupe (e.g.: Linux), silently treat the "weak-linking" operation as a normal, proper linking. This would be equivalent to internally setting the target's SYMBOL_RESOLUTION property back to "LINK", changing the set of library dependencies to UNION(dependencies, WEAK_LIB), and clearing the target's WEAK_LIB property.

- Extend the current logic that is carried out when building targets to include an additional post-build step that resembles the following:

```
CHECK(the target's SYMBOL_RESOLUTION property)
IF(SYMBOL_RESOLUTION is "DYNAMIC")
    INSPECT(the produced binary for undefined symbols)
    FOREACH(undefined symbol, S)
        SEARCH(through every dependency library and
               RECURSIVELY through their dependencies
               (including "WEAK_LIBS"))
        IF(a definition for S can not be found)
            ADD(S to a list of problematic symbols)
IF(problem symbols list is non-empty)
    ISSUE(warning)
```

- Introduce a global property and target-specific property that can be used to suppress or escalate warnings due to undefined symbols for which the CMake post-build step could not find a matching library dependency. Perhaps something like "WARN_MISSING_WEAK_SYMBOLS"; "WARN" for default

behavior, "SUPPRESS" to suppress the warnings, or "FATAL" to promote them to FATAL errors.

Examples

Reference implementation

https://github.com/opadron/weak-linking-demo

Hypothetical application that requires weak-linking for proper execution.

"weaklib.c"

"stronglib.cpp"

"module.cpp"

- Shared library (aka module) loaded at runtime (i.e.: "MODULE"), weakly linked against "weaklib". Strongly linked against "stronglib".

"main.cpp"

- Strongly linked against external library: "lib"

```
add_library(weaklib SHARED weaklib.c)
add_library(stronglib SHARED stronglib.cpp)

add_library(module MODULE module.cpp)
if(SIMPLE_VERSION)
  target_link_libraries(module stronglib WEAK weaklib)
else() # FINER-GRAINED CONTROL
  target_link_libraries(module stronglib)
  set_target_properties(module PROPERTIES SYMBOL_RESOLUTION DYNAMIC)
  set_target_properties(module PROPERTIES WEAK_LIBS weaklib)
endif()
```

```
    add_executable(main main.cpp)
    target_link_libraries(main lib)
```

Changes to SimpleITK

BEFORE

```
    sitk_target_link_libraries_with_dynamic_lookup(
        ${SWIG_MODULE_SimpleITKPython_TARGET_NAME}
        ${PYTHON_LIBRARIES})
```

AFTER

```
    target_link_libraries(
        ${SWIG_MODULE_SimpleITKPython_TARGET_NAME}
        WEAK ${PYTHON_LIBRARIES})
```

Discussion

Email chain as of this writing:

2016.06.09 - Initial discussion

On 06/09/2016 03:20 PM, Jean-Christophe Fillion-Robin wrote:
> To allow weak linking against libraries like "libpython', we now have two
> separate modules in ITK and VTK allowing to do weak linking:
>
> vtkTargetLinkLibrariesWithDynamicLookup.cmake
<https://gitlab.kitware.com/vtk/vtk/merge_requests/1511>
> itkTargetLinkLibrariesWithDynamicLookup.cmake <http://review.source.kitware.com/#/c/21091/>

The meat of those appears to be the following macro along with
the check for allowed undefined symbols:

```
macro( vtk_target_link_libraries_with_dynamic_lookup target )
  if ( ${CMAKE_SYSTEM_NAME} MATCHES "Darwin" )
    set_target_properties( ${target} PROPERTIES LINK_FLAGS "-undefined dynamic_lookup" )
  elseif(VTK_UNDEFINED_SYMBOLS_ALLOWED)
    # linker allows undefined symbols, let's just not link
  else()
    target_link_libraries ( ${target} ${ARGN}  )
```

```
  endif()
endmacro()
```

IIUC the idea is to use this when ${target} comes from add_library()
with the MODULE option, and the goal is to not actually link to
anything but instead tolerate undefined symbols by putting in a note
for the dynamic loader to link them at runtime. I see options for
OS X above. Do we know of equivalents on other platforms?
…

> What is the path forward regarding these modules:
>
> (1) Add a module named "CMakeTargetLinkLibrariesWithDynamicLookup.cmake"

Probably not.

> (2) Improve "target_link_libraries"

Maybe. Certainly there can be a first-class solution. We just need to
figure out what the right interface is to do this.

What do you propose for the "Improve target_link_libraries" approach?

```
add_library(lib1 MODULE source0.cxx source1.cxx ...)
target_link_libraries(lib1 WEAK dep_lib0 dep_lib1 ...)

add_executable(main main.cxx)
# NOT USING WEAK, HERE (SOMEONE HAS TO ACTUALLY LINK)
target_link_libraries(main dep_lib0 dep_lib1 ...)
```

2016.06.29 - Brad K. comments:

On Wed, Jun 29, 2016 at 11:18 AM, Brad King <brad.king@kitware.com> wrote:

There was a lot of work done to make the LINK_LIBRARIES target
property the authoritative reference on direct link dependencies.
The WEAK_LIBS target property splits some of the information out
elsewhere and introduces the possibility of inconsistent values.
Instead one could use a generator expression inside the value of
the LINK_LIBRARIES property to indicate weak linking, e.g.

  $<LINK_WEAK:somelib>

One will also have to think about what this means when placed in
INTERFACE_LINK_LIBRARIES for consumption by dependents. I didn't
notice anything in the proposal about how WEAK_LIBS would be
treated with respect to such usage requirements.

Also, the proposal should show examples of how the actual link
command lines will look in each case (and for each type of platform
support).

What "Weak Linking" Really Means

What's really happening is that you're asking the linker to tolerate undefined symbols while producing a binary (i.e.: executable or library). Normally, the object files that constitute the binary in question would provide definitions for those symbols, or at link time, the linker would be given a list of libraries to search for definitions that aren't found in the source code (i.e.: "normal" linking).

When "weak linking", you're effectively asking the linker to delay the resolution of undefined symbols until the very last possible moment: when the symbols are actually used. In exchange, you're making a "promise" that the

symbol definitions "will be there" when that time comes, since by then, it would be too late for the linker to save you from your own lack of prudence. :) Libraries are the usual mechanism for introducing symbols that were not initially defined at link time (i.e.: "importing"), but they are not the only one. You could load individual object files, synthesize new symbols at runtime with a JIT compiler, or just reassemble the jump tables on-the-fly, if you had the inclination and the right lines of assembly.

So, I mention all of this to help you understand why the question of whether weak linking applies to particular libraries or all libraries doesn't make sense. The answer to that exact question is always "it depends on what you mean".

If you gave me a binary that references symbols from three libraries, A, B, and C, and you've asked the linker to tolerate undefined symbols, I still could not tell you which libraries were being "weakly linked" - there's not enough information. If you actually specify all three libraries when linking, the answer is "none of them -- there are no undefined symbols, so the whole thing is moot". If you leave out one of the libraries (say, B), then you could say that the binary was "weakly linked" against it, but then again, you could say the same thing about any library that exports a symbol by the same name as one that your binary would need from B at run time. What about a fourth library, D? In a sense, your binary would be "weakly linked" against it ... assuming it actually refers to symbols exported by D that are otherwise undefined. If not, then there's nothing actually "linking" the binary to D. You could go out of your way to import D's symbols somehow and the only thing you'd accomplish is wasting memory (and possibly colliding with symbols defined elsewhere).

On the matter of building a weak-linking model in CMake based on individual symbols, I'll say it's something we can do, but it would be impractical for large numbers of symbols; and I think it would be of little use since the only symbols that matter are those that are undefined anywhere else. The only value might be to keep you from accidentally leaving a symbol undefined, but you're just as likely to accidentally *define* a symbol from another library and a program crash + backtrace should work well enough to figure out when you've made a mistake.

Also, I'll add this other thought I just had on the problem of undefined symbols:

The only scenarios in which leaving a symbol undefined is problematic in practice is when there is a violation of the API boundary, which is against best practice, anyway.
If your program uses a symbol that is part of a library's public API, and you upgrade to a version that breaks backwards compatibility by removing this symbol, then your program would likely crash. But in this scenario, your program is likely to crash for any number of reasons that have nothing to do with how you were resolving symbols -- that's why you have a test suite. Now, something we can and *absolute should* do is inspect the produced binary and its dependencies and at least make sure that every undefined symbol has a definition *somewhere*, or emit a warning otherwise. If you weakly-link your library against some other library, and you refer to symbols that are not explicitly advertised by your library's documentation as being part of the API, well then the onus really should be on you to keep your own ducks in a row.

Interesting read:

- https://wiki.python.org/moin/boost.python/CrossExtensionModuleDependencies
    - Very interesting. I must confess that when I was considering how multiple copies of a symbol might cause problems, I was only thinking with my C programmer hat. It doesn't surprise me that static symbols and function pointers require extra care, but I didn't even think about how this issue compounds with C++ semantics! -- Omar

-

Target properties

- SYMBOL_RESOLUTION with values:
    - LINK: Default
    - DYNAMIC: indicate whether undefined symbols are allowed to be resolved at load time
- WEAK_LIBS: set of all dependency libraries that are not meant to actually be included in the linker command

Interesting reads:

- https://www.akkadia.org/drepper/goodpractice.pdf
- http://www.linuxjournal.com/article/6463
- https://www.akkadia.org/drepper/dsohowto.pdf