DDL

… will save you from the darkness of *DLL Hell*

by Tomasz “h3r3tic” Stachowiak

Dynamic Link Libraries

- Everyone knows what they are…
- Widely useful in extensible applications
- Supported to varying degrees by OSes
  - **SO** on Unix is quite nice
  - **DLL** on Windows not so much
    - Most problems discussed here are about Windows DLLs
- Many potential uses
  - But the APIs are archaic

DLL Hell

D’s own DLL hell – circle o

- **API**
- Very raw and low-level interfaces to dynamic libs
  - Nothing beyond simple symbol iteration
  - Usually extern(C) or extern(Windows) must be applied
  - No easy way to access classes

D’s own DLL hell – circle 1

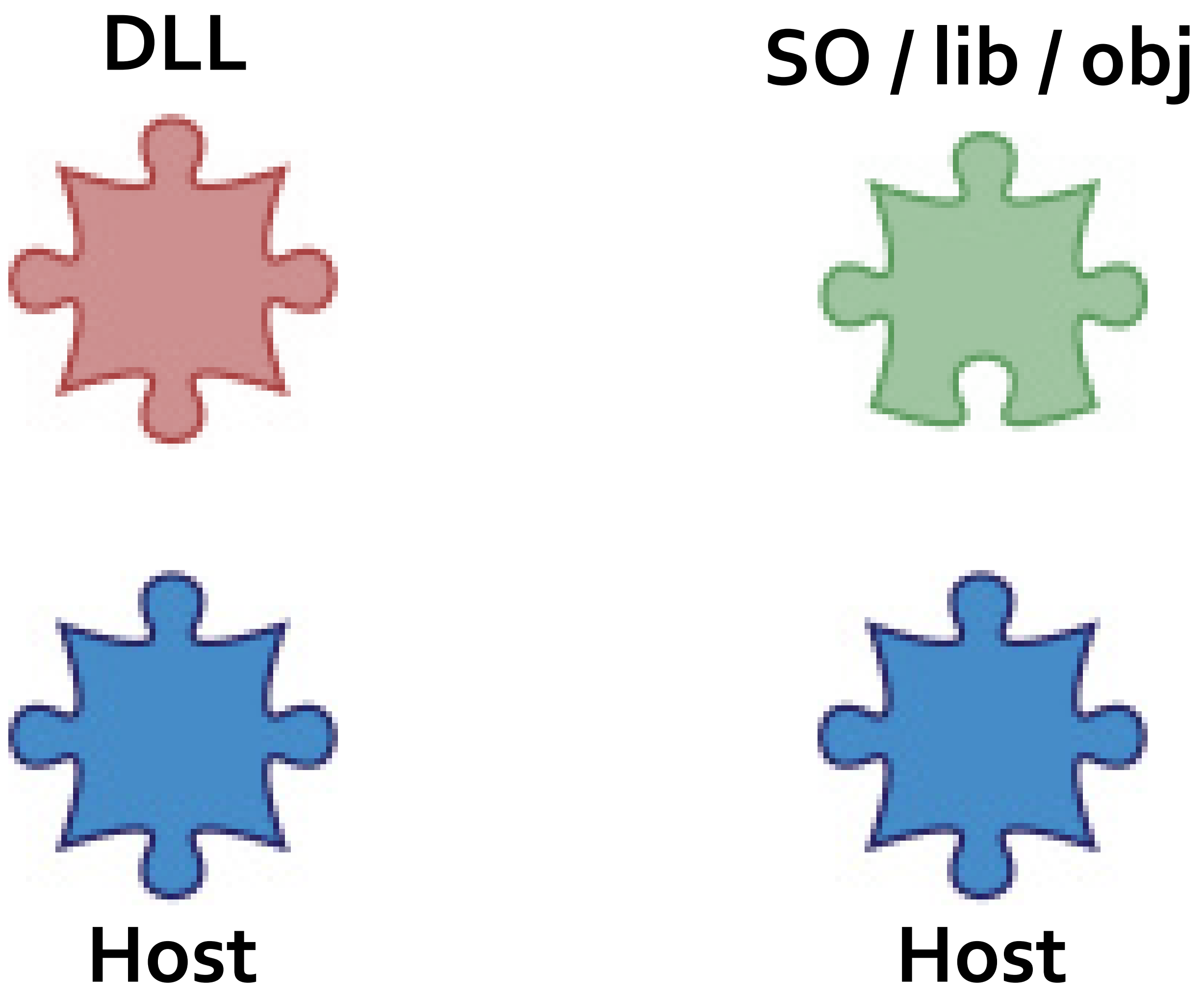
- **Memory** “boundaries”
  - Allocate GC memory on one side, store only on the other -> unexpected garbage collections
  - Allocate memory on one side, free on the other -> crash
    - Partially fixed with a shared GC handle

D’s own DLL hell – circle 2

- **Bloat**
- All symbols have to be strong …
  - Multiple runtimes
  - Multiple globals
    - Singletons that aren’t
  - Cannot reference symbols from the host
    - Cannot use the same ModuleInfo, TypeInfo or ClassInfo
      - … more on that later

// Symbol - A name that represents a code, data or metadata address at runtime

Strong vs weak/unresolved symbols



D’s own DLL hell – circle 3

- **Locking**
- The DLL file is locked while the library is loaded by the application
  - May sound reasonable at first, but…
  - Can’t recompile and quickly reload the DLL
  - The host must resort to complex locking schemes

D’s own DLL hell – circle 4

- **Casting**
- cast doesn’t work anymore…
  - Seriously… cast(Object)dllObject executed inside the host will yield null
  - Apps have to cast though void* and do classinfo.name – based type checking
    - But classinfo.name doesn’t work well for class templates

D’s own DLL hell – circle 5

- **Exceptions** don’t work across DLL boundaries
  - Exception hooks are not shared
  - Even if it could, exception types are detected though ClassInfo …

D’s own DLL hell – circle 6

- **Unloading**
  - The app doesn’t have any idea when it’s safe to unload the DLL
  - Unloading a library whose class instances still exist will yield finalizer calls into nothingness
  - Access Violations on seemingly innocent pointer/reference access
  - … But we can’t hold onto the lib for long, since it’s huge and we need to unload it to unlock the file…

Problems with SO

- Not available on Windows
  - Security, please take out the *nix zealots
- DMD-Linux can’t do SO
- The GC can’t track dependencies in the kernel
- “DLL” has not enough ‘D’ in it, “SO” the worse

The origins of DDL

- A heroic coder, Eric "*pragma*" Anderton went into the deepest levels of DLL hell
- He was looking for a legendary artifact he could use in the DSP project
  - D Server Pages
  - Mixed D/HTML pages compiled on-demand into dynamic libs
- But the artifact could not be found in the depths of DLL. Thus *pragma* crafted his own.

DDL – overview

- Parsers for intermediate files
- Dynamic Modules
- Dynamic Libraries
- Dynamic Library Loaders
- Loader Registries
- Linkers

- Tools

D Dynamic Libraries - structure

- Parsers for intermediate files
  - Only OMF complete at the moment
    - Read: DMD-Win
  - ELF and COFF partially done

- DynamicModule (and subclasses)
  - Wraps data from a Binary
  - Does relocation
  - Interface for symbol, namespace and attrib access

D Dynamic Libraries – structure (2)

- DynamicLibrary
  - May contain multiple DynamicModules
  - Can accelerate symbol lookup by creating a cross-reference
  - May implement custom symbol lookup mechanisms
    - PathLibrary
    - LazyLibrary

D Dynamic Libraries – structure (3)

- LoaderRegistry
  - Matches loaders for specific formats to binary files
  - DefaultRegistry
- Linker
  - Takes multiple libraries / modules and binds them together
  - Runs module ctors
  - Will turn unresolved libs into working binary code

DDL – simple demo

```
char[] helloWorld() {
        return "Hello from DDL";
}
```

```
auto linker = new Linker(new DefaultRegistry);
linker.loadAndRegister("Host.map");
auto plugin = linker.loadAndLink("Plugin.obj");
auto helloWorld = plugin.getDExport!(char[] function(),"Plugin.helloWorld")();
Stdout(helloWorld()).newline;
```

- Notice lack of extern(C/Windows)
- Plugins must be built with –g, host with –L/M
- Symbol sharing is being used already
  - “unresolved *ModuleInfo.__vtbl*” in the plugin

Can it really save us from damnation?

- That was pretty trivial, but DDL has worked on a larger scale…
- … in Deadlock
  - Plugins
    - Acquisition of subclasses
  - Rendering kernels
    - Runtime compilation and loading
  - Stable …
  - … but required a messy build system

Does this hero work alone?

- **insitu**
  - Wraps .map files in an optimized, portable format
- **bless**
  - May contain any DDL-loadable binaries
  - Additionally: attributes
    - e.g., version info
- **ddlinfo**
  - Can tell you everything

ddlinfo

```
> ddlinfo Plugin.obj

filename: 'Plugin.obj'
type: 'OMF'
attributes:
omf.filename - Plugin.obj

Modules (1):

Plugin.d

Symbols (3):
weak char[] Plugin.helloWorld()
unresolved  ModuleInfo.__vtbl
strong  Plugin.__ModuleInfo
```

The light is getting brighter…

- TangoTrace provides stack traces upon program crashes
  - Forked off the Phobos backtrace hack by Shinichiro Hamaji
- DDL + TangoTrace = stack traces within dynamic libs
  - Currently only in my experimental DDL branch
- Demo!

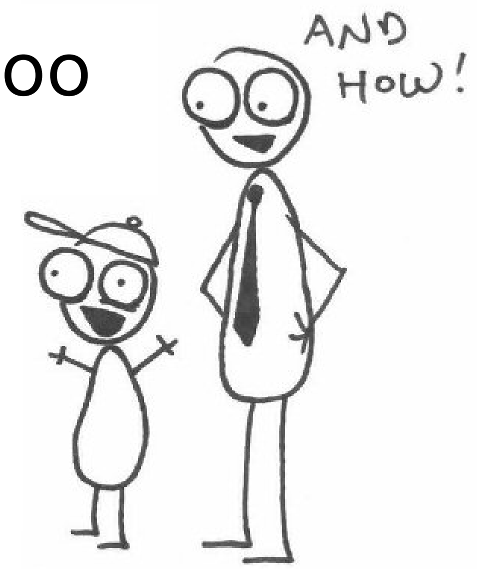
DDL Heaven

DDL heaven

- Loading of D symbols via simple function templates
- Class iteration, constructor acquisition
  - foreach (cl; dynamicLib.**getSubclasses**!(Plugin))
  - cl.**newObject**(foo, bar, mudkip);
- Libraries can have unresolved symbols
  - No more bloat
- Global sharing
  - Singletons that truly are

DDL heaven (2)

- Casting across binary boundaries works again
  - It has to, ClassInfo is shared
- Exception handling works too



- The light… it’s almost blinding me…

DDL heaven (7)

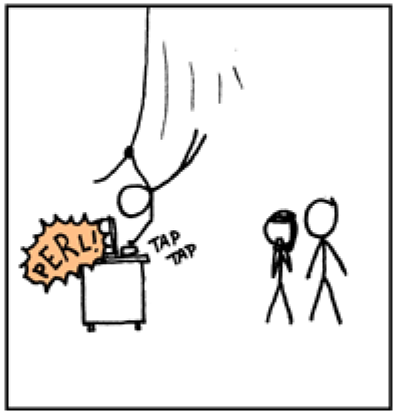
- Libraries can be re-compiled in-place
  - No file locking
- Unloading can be left to the GC
  - Modules will not be unloaded while something is referencing the code within them
- Yet, manual unloading is possible as well

My experimental branch

- Custom linker
- Flexible library “providers”
- On-demand compilation
  - Object (.obj) and dependency caching
  - Dependency management
- User-defined link order
- Config file – driven

Experimental architecture

- Extension over DDL’s binary Loaders:
  - Provider
    - Loads and returns a DynamicLibrary given its path
  - ProviderRegistry
    - Matches Providers by rules
    - type plugin   regex   plugin/.*



- ObjProvider
  - Simply uses DDL’s LoaderRegistry in order to load standard DDL files (.map, .obj, .lib, .ddl, .situ, …)

Experimental architecture (2)

- DProvider
  - Compiles D modules with DMD
  - Uses Bu[il]d’s **–uses** option to find dependencies between modules
  - Caches Object files and dependency info
    - Compares file times
    - Future work: store options used to compile as well
  - Returns one DynamicLibrary per module

Experimental architecture (3)

- LazyLinker
  - Resolves symbols in the user-defined order
  - Doesn’t load libraries unless necessary
  - Recognizes different types of libraries
    - Each may define its own link order
    - Some plugins should not get symbols from other plugins
      - Might result in unloading problems
  - Can be told to unload libraries

I have no idea what you're talking about… so here's a bunny with a pancake on its head

Simple example

```
import xf.linker.DefaultLinker;
import tango.io.Stdout;

void main() {
    auto linker = createDefaultLinker(`Host.link`);
    auto plugin = linker.load("Plugin.d", ".");
    auto helloWorld = plugin.getDExport!(
                    char[]
    function(),"Plugin.helloWorld")();

    Stdout(helloWorld()).newline;
}
```

```
module Plugin;

char[] helloWorld() {
            return "Hello from DDL";
}
```

```
std-include          import

type host            regex     .*\.map
type plugin          regex     .*\.d

order host           self
order plugin         host self

load Host.map        .
```

More interesting example

- Host
  - Creates an OpenGL window
  - Calls the plugin’s rendering function in a loop
  - Checks the plugin’s source file for modifications
    - Unloads, recompiles and reloads the plugin on the fly
  - Leaves the old plugin for the GC
- Plugin
  - Renders a simple scene to an OpenGL texture using ray tracing

The linker inside Nucleus

- Rendering quarks managed by the same mechanism
- We don’t want symbol sharing between quarks
  - Custom link order does the trick
- Quarks may pull symbols from plugins
  - Worst case – reload all quarks
- Everything pulls symbols from the host

Nucleus' linking mechanism

- D code plus extra constructs
  - preprocessing
- D code
  - compilation
- Obj files
  - loading
- Lazy linking

The future of DDL

- ELF support
  - Read: Unix
- Linker enhancements
  - Hopefully influenced by LazyLinker
- Reflection Lib
  - Access classes / methods / fields within the libs and the host app
- Runtime High-Level Assembler
  - Create new functions, objects and data at runtime

References

- http://dsource.org/projects/ddl/
- http://dsource.org/forums/viewforum.php?f=70
- http://teamoxf.com:1024/linker
- http://teamoxf.com:1024/ext -> ddl

- Eric “pragma” Anderton
  - Still alive!
  - Reachable!
  - eric.t.anderton@gmail.com

Thanks for listening!

Questions ?

A wizard has turned you into a mudkip.

Is this awesome (Y/N)?