

+ 21

Dynamically Loaded Libraries Outside the Standard

ZHIHAO YUAN



20
21



Dynamically Loaded Libraries Outside the Standard

Zhihao Yuan <zhihao.yuan@broadcom.com>

2021/10/29

Are we talking about...

- Dynamic-link library (.dll)
- Dynamic shared object (.so)
- Mach-O dynamic library (.dylib)

Background



Dynamic linking ≠ Dynamic loading

Dynamic linking

- As opposed to **static linking**
- Form a physical aspect of a program
- Relocation is done at load time

Dynamic loading

- Ask for **additional** functionalities
- Often involve library discovery
- Relocation is done at run time
- May be capable of “**unloading**”

Example: NSBundle

- Older versions of Mac OS X did not ship `dlopen()`
- Loadable modules, called *bundles* in Mac OS X, are not *.dylib*
- Usually have *.so* or *.bundle* filename extension
- Do **not** support linking by passing `-lname` to `ld`
- Some deprecated APIs can load, link, and unload bundles at runtime
- `dlopen()` APIs works with both dynamic libraries and bundles
- Prior to Mac OS X 10.5, `dlclose()` does not unload dynamic libraries

In case you don't know...

- **P**ortable **E**xecutable format (PE) – exe, dll, etc. on Windows
- **E**xecutable and **L**inkable **F**ormat (ELF) – for UNIX and UNIX-like OS
- **M**ach **O**bject file format (Mach-O) – executable, dylib, bundle, etc.

Customized file formats for dynamic loading

- Apache httpd modules (DSO modules)
 - `LoadModule` directive
- C extension for CPython (*.pyd* on Windows)
 - `import module_name`
- Java Native Interface (JNI)
 - `System.loadLibrary(name)`

Sample loadable library

```
#include "repromath_export.h"
```

```
namespace repromath
```

```
{
```

```
    REPROMATH_EXPORT auto ddot(int n, double const *x, double const *y) -> double;
```

```
    REPROMATH_EXPORT auto dsum(int n, double const *x) -> double;
```

```
}
```

CMake

```
add_library(repromath MODULE)
generate_export_header(repromath)
```

APIs: Win32

- **LoadLibraryEx**("mylib.dll", *nullptr*, *flags*)
 - open and get a handle to the library
- **LoadLibrary**("mlib.dll")
 - ditto, but searches default DLL directories
- **GetProcAddress**(*handle*, "function_or_variable_name")
 - get addresses to entities
- **FreeLibrary**(*handle*)
 - unload the library
- **GetLastError**()
 - get error code if any call failed

Checkout what symbols are exported

- `dumpbin /exports repromath.dll`

Section contains the following exports for repromath.dll

```
00000000 characteristics
FFFFFFFF time date stamp
    0.00 version
        1 ordinal base
        2 number of functions
        2 number of names
```

ordinal	hint	RVA	name
---------	------	-----	------

1	0	00001203	?ddot@repromath@@YANHPEBN0@Z = @ILT+510(?ddot@repromath@@YANHPEBN0@Z)
2	1	00001118	?dsum@repromath@@YANHPEBN@Z = @ILT+275(?dsum@repromath@@YANHPEBN@Z)

Example: Win32

```
double x[] = {1., 2., 3.};
double y[] = {4., -5., 6.};
auto lib = ::LoadLibraryW(L"repromath.dll");
/* ...handle errors */
typedef auto ddot_t(int, double const *, double const *) -> double;
auto ddot = (ddot_t *)::GetProcAddress(lib, "?ddot@repromath@@YANHPEBN0@Z");

printf("result = %g\n", ddot(3, x, y));
::FreeLibrary(lib);
```

APIs: POSIX

- `dlopen("mylib.so")`
 - open and get a handle to the library
- `dlsym(handle, "symbol_name")`
 - get addresses to entities
- `dlclose(handle)`
 - close symbol table handle
 - may unload the library; varies across implementations
- `dlerror()`
 - get a descriptive string of the last error in dl

Checkout what symbols are exported

- `objdump -T librepromath.so`

```
librepromath.so:      file format elf64-x86-64
```

DYNAMIC SYMBOL TABLE:

0000000000000000	w	DF	*UND*	0000000000000000	GLIBC_2.2.5	__cxa_finalize
0000000000000000	w	D	*UND*	0000000000000000		_ITM_deregisterTMCloneTable
0000000000000000	w	D	*UND*	0000000000000000		__gmon_start__
0000000000000000	w	D	*UND*	0000000000000000		_ITM_registerTMCloneTable
0000000000001160	g	DF	.text	0000000000000033	Base	_ZN9repromath4dsunEiPKd
0000000000001120	g	DF	.text	000000000000003b	Base	_ZN9repromath4ddotEiPKdS1_
0000000000001210	w	DF	.text	000000000000004b	Base	_ZSt10accumulateIPKddET0_T_S3_S2_
00000000000011a0	w	DF	.text	0000000000000069	Base	_ZSt13inner_productIPKdS1_dET1_T_S3_T0_S2_

Example: POSIX

```
double x[] = {1., 2., 3.};
double y[] = {4., -5., 6.};
auto lib = ::dlopen("./librepromath.so", RTLD_LOCAL | RTLD_NOW);
/* ...handle errors */
typedef auto ddot_t(int, double const *, double const *) -> double;
auto ddot = (ddot_t *)::dlsym(lib, "_ZN9repromath4ddotEiPKdS1_");

printf("result = %g\n", ddot(3, x, y));
::dlclose(lib);
```

Loaded libraries are reference counted

- `dlopen`, `LoadLibrary`, and `LoadLibraryEx` increment the count
- `dlclose` and `FreeLibrary` decrement the count
- `GetModuleHandleEx` can increment the reference count of a loaded module



Different Levels of Dynamic



How dynamic is dynamic?

- Level 0: Dependent libraries

Show library dependencies

- `dumpbin /dependents pe-filename`
- `ldd elf-filename`
- `otool -L mach-o-filename`

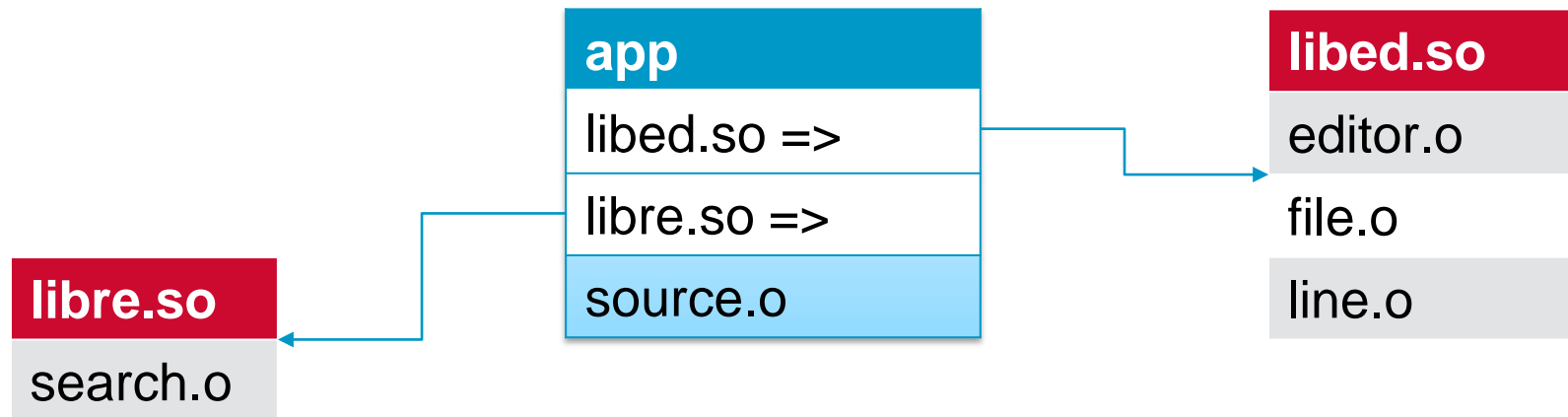
Static linking

app
editor.o
file.o
line.o
search.o
source.o

Static linking static libraries



Dynamic linking



Dependent libraries

- Decompose an application's material for shipping
- Dependency established at build time
- C++ abstract machine can be implemented on top of dependent libraries
 - But not all that dependent libraries can do fit into the language

How dynamic is dynamic?

- Level 0: Dependent libraries
- Level 1: Delay loading

Delay loading

- Load a dependent library only when referencing the first name in it
- More dynamic in terms of timing
- Useful when improving application startup time

Cross-platform delay loading?

- Windows: helper library *delayimp.lib* + linker option `/DELAYLOAD:mylib.dll`
- Solaris: linker option `-z lazyload -lmylib`
- Linux: DIY solution: *Implib.so* (DLL-like import library for POSIX)

MSVC /DELAYLOAD

```
printf("address prior to use: %p\n", repromath::ddot);  
printf("result = %g\n", repromath::ddot(3, x, y));  
printf("address after using:  %p\n", repromath::ddot);
```

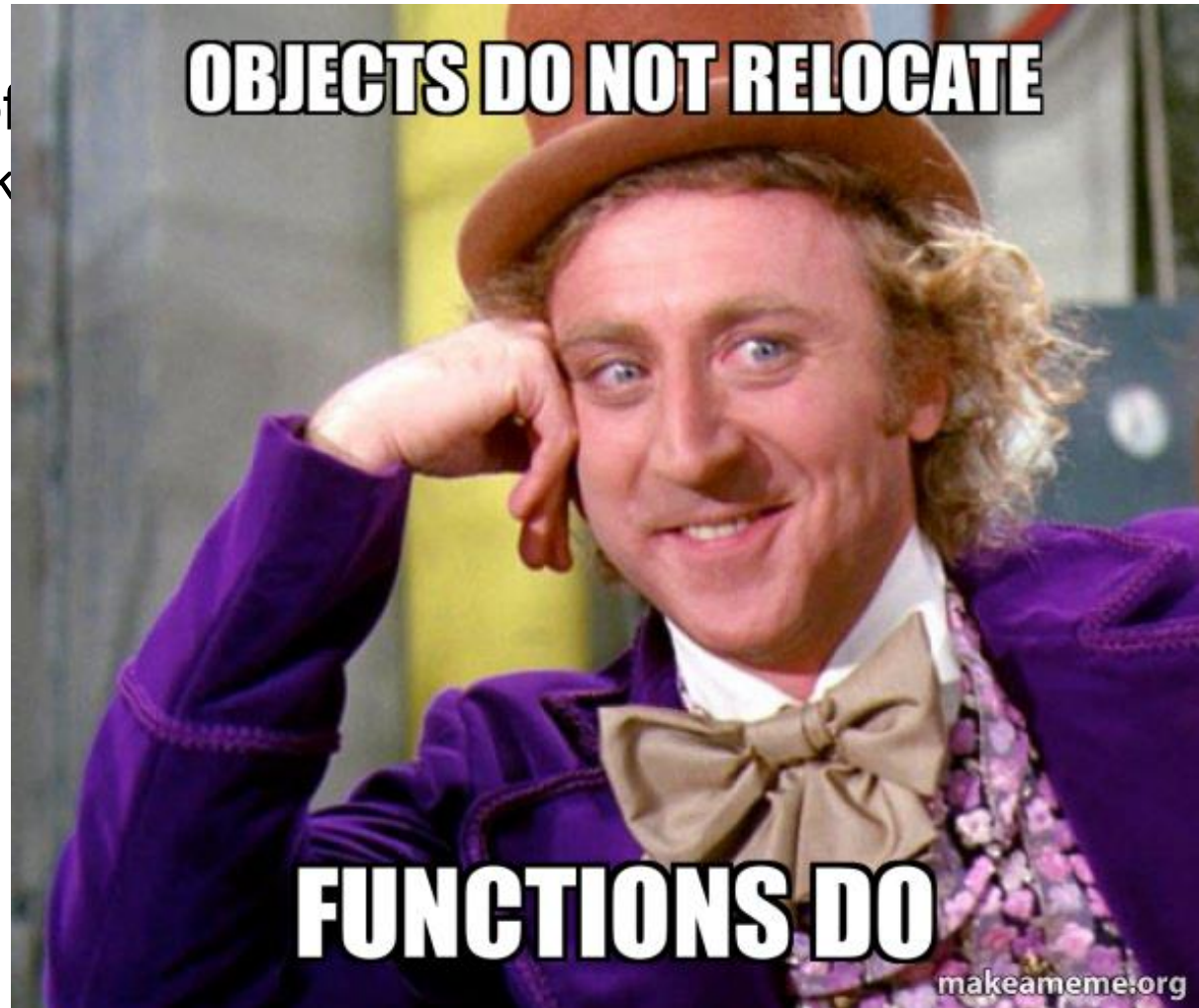
```
address prior to use: 00007FF6FD6F1819  
result = 12  
address after using:  00007FFF627A1203
```

Surprise of delay loading

- The addresses of functions may change at runtime
 - address of thunk \neq address of the actual function

Surprise of delay loading

- The addresses of
 - address of thunk



How dynamic is dynamic?

- Level 0: Dependent libraries
- Level 1: Delay loading
- Level 2: Foreign linkage modules

Let's take a closer look at the dynamic loading APIs

```
void *dlsym(void *handle, char const *symbol);
```

Let's take a closer look at the dynamic loading APIs

```
(ddot_t *)::dlsym(lib, "_ZN9repromath4ddotEiPKdS1_");
```

Can a pointer of type `void *` be
casted to a pointer to function?

Short answer

- Converting a function pointer to an object pointer type or vice versa is conditionally-supported
- POSIX requires this conversion to work correctly on conforming implementations

What about Win32?

```
FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName);
```

```
typedef INT_PTR (FAR WINAPI *FARPROC)();
```

GetProcAddress

- The return type is a pointer to function
- A pointer to which function?



Surprise of the dynamic entities

The results of resolving symbols at runtime point to

- functions that are foreign to the program, or
- objects that are foreign to the object model

How foreign is foreign?

- Loadable module being used in a different language
- Accessed through a **Foreign Function Interface (FFI)**

Example: ctypes

```
from ctypes import CDLL, c_double
```

```
x = (c_double * 3)(1., 2., 3.)
```

```
y = (c_double * 3)(4., -5., 6.)
```

```
repmath = CDLL('repmath.dll')
```

```
ddot = repmath['?ddot@repmath@@YANHPEBN0@Z']
```

```
ddot.restype = c_double
```

You'll get garbage if omitting this

```
print("result = {}".format(ddot(3, x, y)));
```

FFI want to be strongly-typed as well

- **pybind11**: You write C++ code that sets up the module in Python
- **JNA** (Java Native Access): You declare C functions using Java grammar
- **pydffi** (**DragonFFI** for Python): You declare C or C++ functions in... C++

```
import pydffi
pydffi.dlopen("/path/to/libarchive.so")
CU = pydffi.FFI().cdef("#include <archive.h>")
a = funcs.archive_read_new()
```

Foreign linkage (hypothetical)

```
#include <repromath.h>
```

```
int main()
```

```
{
```

```
    double x[] = {1., 2., 3.};
```

```
    double y[] = {4., -5., 6.};
```

```
    auto lib = dlopen("repromath");
```

```
    auto ddot = __magic<repromath::ddot>(lib);
```

```
    printf("result = %g\n", ddot(3, x, y));
```

```
}
```


Resolving dynamic entities using declarations

- Declaration decides the symbol of the entity
- The idea also presents in *Plugins in C++* (wg21.link/n2015), 2006

How dynamic is dynamic?

- Level 0: Dependent libraries
- Level 1: Delay loading
- Level 2: Foreign linkage modules
- Level 3: Plugin systems

What happens if two loadable libraries defined the same entity?

- If we view the whole program in memory as a “C++ program,” ODR violation
 - Symptoms may vary
- In real world, all mainstream platforms made effort to mitigate this risk
 - But there are always intriguing ways to be hit by this issue


Symbol conflicts are accidental to entities with foreign linkage


- What OpenSSL + LibreSSL can do anything good in the same process?
- “ABI compatible” implies that we want the functionality to be substitutable


What if we use the same ABI to programmatically get extra functionality?


Case: GEGL


- **Generic Graphics Library (GEGL)** from GIMP
- Adding-removing functionalities by dragging & dropping files


 exr-load.dll


 jp2-load.dll


 npy-save.dll


 pixbuf-save.dll


 ppm-load.dll


 rgbe-load.dll


 tiff-load.dll


 webp-save.dll


 exr-save.dll


 jpg-load.dll


 pdf-load.dll


 png-load.dll


 ppm-save.dll


 rgbe-save.dll


 tiff-save.dll


 gif-load.dll


 jpg-save.dll

 pixbuf-load.dll

 png-save.dll

 raw-load.dll

 svg-load.dll

 webp-load.dll

A typical plugin architecture in C

```
PLUGINAPP_API LPPLUGINSTRUCT plugin_app_create_plugin(void);
PLUGINAPP_API void plugin_app_destroy_plugin(LPPLUGINSTRUCT);
PLUGINAPP_API const gchar* plugin_app_get_plugin_name(void);
PLUGINAPP_API const gchar* plugin_app_get_plugin_provider(void);
PLUGINAPP_API const gchar* plugin_app_get_menu_name(void);
PLUGINAPP_API const gchar* plugin_app_get_menu_category(void);
PLUGINAPP_API void plugin_app_run_proc(void);
```

Example modified from
<https://www.codeproject.com/Articles/389667/Simple-Plug-in-Architecture-in-Plain-C>

Plugin1 implements...

```
LPPLUGINSTRUCT plugin_app_create_plugin()
{
    g_debug("PluginDialog1::plugin_app_create_plugin");
    /* ... */
    return PLS;
}

const gchar* plugin_app_get_plugin_name()
{
    g_debug("PluginDialog1::plugin_app_get_plugin_name");
    return "Dialog1 Plugin";
}
```

Plugin2 implements...

```
LPPLUGINSTRUCT plugin_app_create_plugin()
{
    g_debug("...");
    /* ... */
    return PLS;
}
```

```
const gchar* plugin_app_get_plugin_name()
{
    g_debug("...");
    return "Dialog2 Plugin";
}
```


Multiple definitions to the same entity

```
LPPLUGINSTRUCT plugin_app_create_plugin()  
{  
    g_debug("Some thing");  
    /* ... */  
    return PLS;  
}  
  
const gchar* plugin_app_get_plugin_name()  
{  
    g_debug("...");  
    return "Dialog1 Plugin";  
}
```

```
LPPLUGINSTRUCT plugin_app_create_plugin()  
{  
    g_debug("Some other thing");  
    /* ... */  
    return PLS;  
}  
  
const gchar* plugin_app_get_plugin_name()  
{  
    g_debug("...");  
    return "Dialog2 Plugin";  
}
```

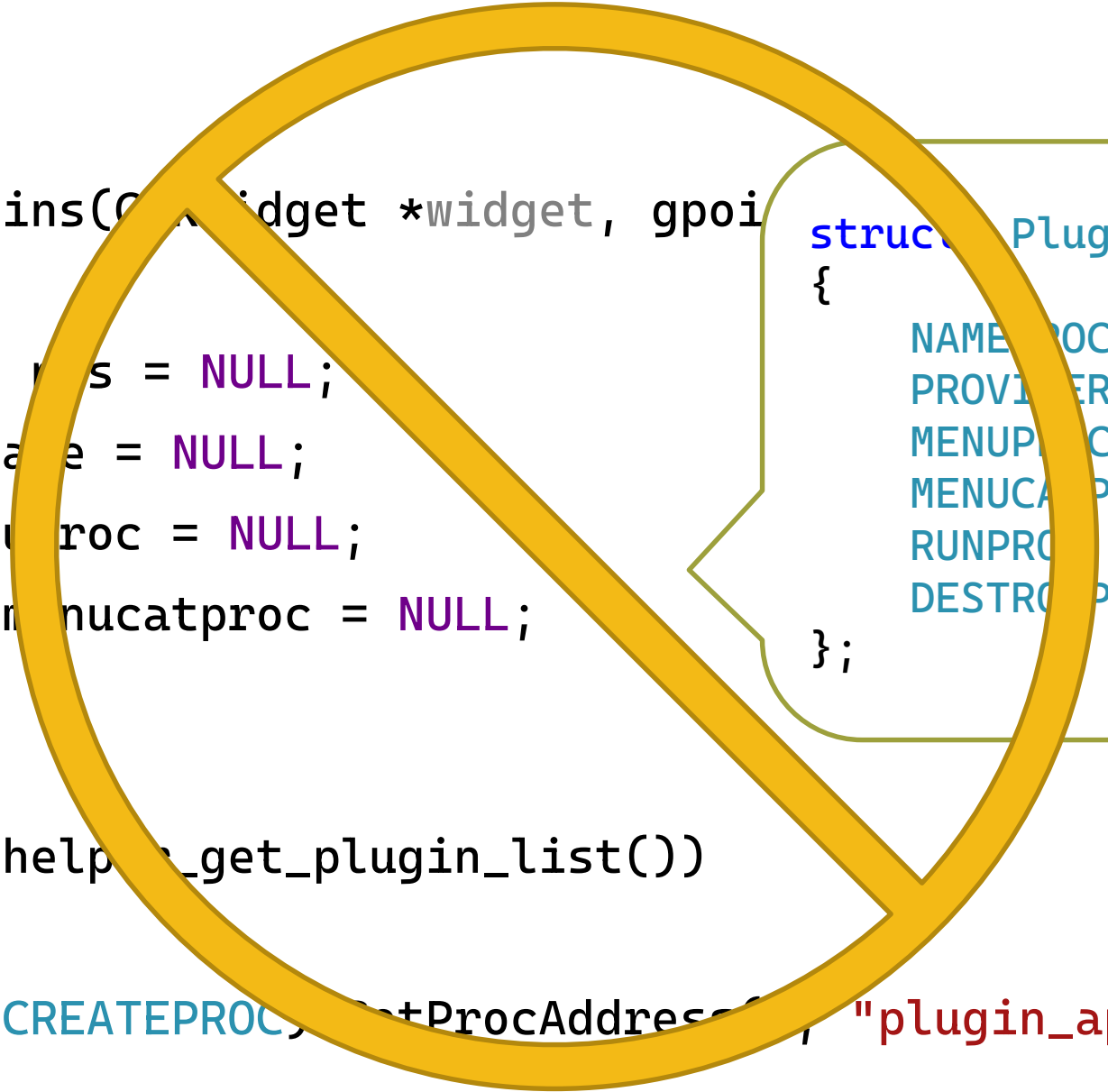
Prepare the types for GetProcAddress

```
typedef LPPLUGINSTRUCT (*CREATEPROC) (void);  
typedef void           (*DESTROYPROC) (LPPLUGINSTRUCT);  
typedef const gchar*   (*NAMEPROC)    (void);  
typedef const gchar*   (*PROVIDERPROC)(void);  
typedef const gchar*   (*MENUPROC)    (void);  
typedef const gchar*   (*MENUCATPROC) (void);  
typedef void           (*RUNPROC)     (void);
```

How to use

```
void load_all_plugins(CppWidget *widget, gpoi
{
    LPPLUGINSTRUCT p = NULL;
    CREATEPROC create = NULL;
    MENUPROC menuProc = NULL;
    MENUCATPROC menucatproc = NULL;
    /* ... */

    while (plugin_helper_get_plugin_list())
    {
        create = (CREATEPROC) GetProcAddress("plugin_app_create_plugin");
```

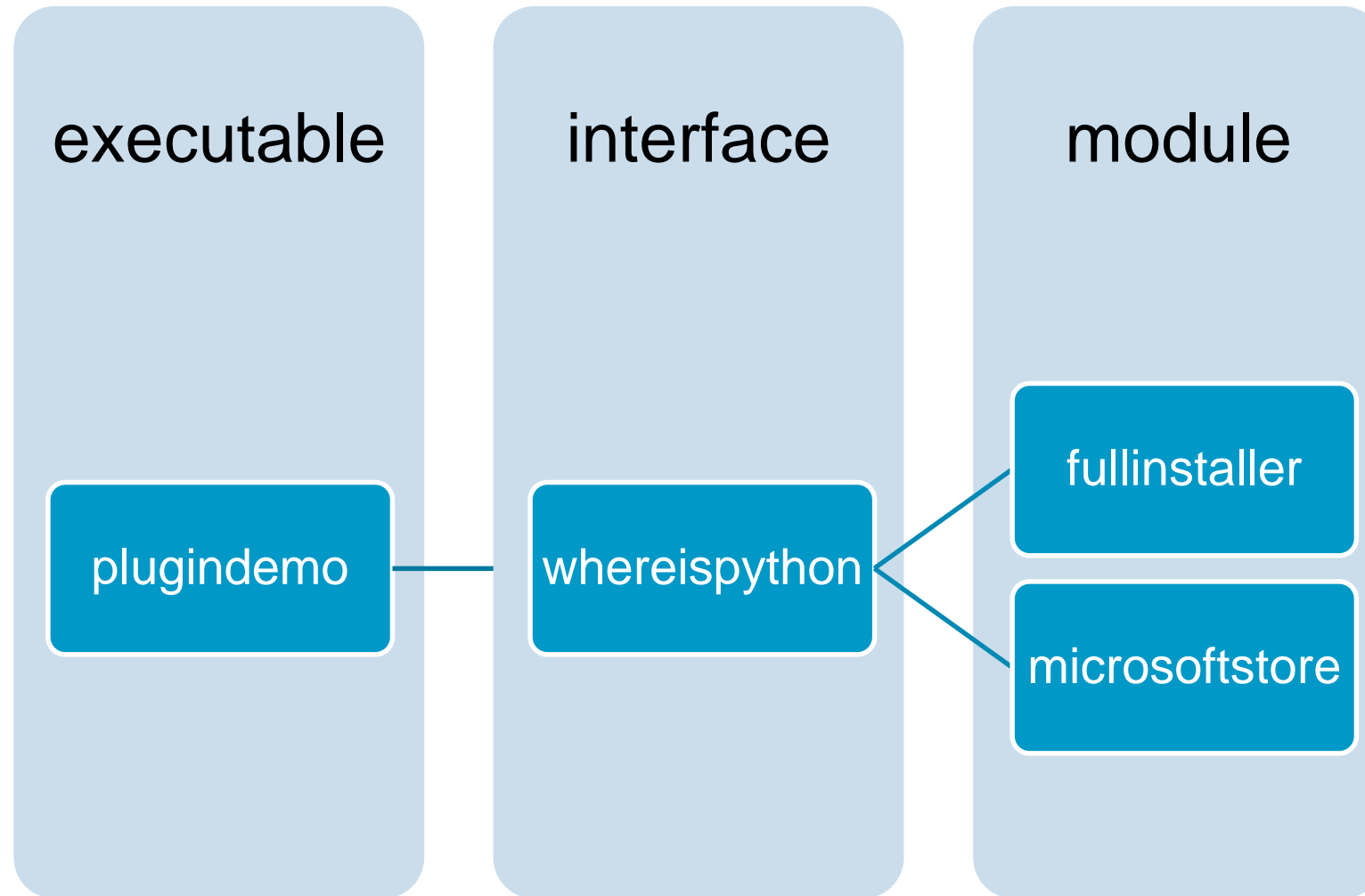


```
struct PluginStruct
{
    NAMEPROC nameProc;
    PROVIDERPROC providerProc;
    MENUPROC menuProc;
    MENUCATPROC menuCatProc;
    RUNPROC runProc;
    DESTROYPROC destProc;
};
```

Vtable

- Provide different implementations to the same set of functions

Example: Plugin system in C++



whereispython: header & namespace

```
class installation
{
    public:
        virtual auto executable() -> std::filesystem::path = 0;
        virtual auto windowed_executable() -> std::filesystem::path = 0;
        virtual ~installation() = default;
};

class factory
{
    public:
        virtual auto lookup(char const *ver) -> std::unique_ptr<installation> = 0;
};
```

fullinstaller: dll and namespace

```
class fullinstaller : public installation
{
    std::unique_ptr<HKEY, hkey_deleter> hkey_;

public:
    explicit fullinstaller(char const *version);

    auto executable() -> std::filesystem::path override
    { return string_value(L"ExecutablePath"); }

    auto windowed_executable() -> std::filesystem::path override
    { return string_value(L"WindowedExecutablePath"); }

};
```

fullinstaller: dll and namespace

```
class fullinstaller_factory : public factory
{
public:
    virtual auto lookup(char const *version) -> std::unique_ptr<installation>
    {
        try
        { return std::make_unique<fullinstaller>(version); }
        catch (std::exception &)
        { return nullptr; }
    }
};
```


fullinstaller: dll & global namespace

```
#pragma comment(linker, "/export:instance=?instance@@3Vfullinstaller_factory@whereispython@@A")
```

```
whereispython::fullinstaller_factory instance;
```

Section contains the following exports for fullinstaller.dll

```
00000000 characteristics
FFFFFFFF time date stamp
0.00 version
1 ordinal base
1 number of functions
1 number of names
```

ordinal	hint	RVA	name
---------	------	-----	------

1	0	00015000	instance = ?instance@@3Vfullinstaller_factory@whereispython@@A
---	---	----------	--

microsoftstore: dll and namespace

```
class microsoftstore : public installation
{
    std::filesystem::path install_location_;

public:
    explicit microsoftstore(char const *version)
    {
        auto shell = PowerShell::Create()
            ->AddCommand("Get-AppxPackage")

        ...
    }

    ...
}

class microsoftstore_factory : public factory
{
    ...
}
```

microsoftstore: dll & global namespace

```
#pragma comment(linker, "/export:instance=?instance@@3Vmicrosoftstore_factory@whereispython@@A")
```

```
whereispython::microsoftstore_factory instance;
```

You can also use a DEF file

```
Section contains the following exports for microsoftstore.dll

00000000 characteristics
FFFFFFFF time date stamp
    0.00 version
        1 ordinal base
        1 number of functions
        1 number of names

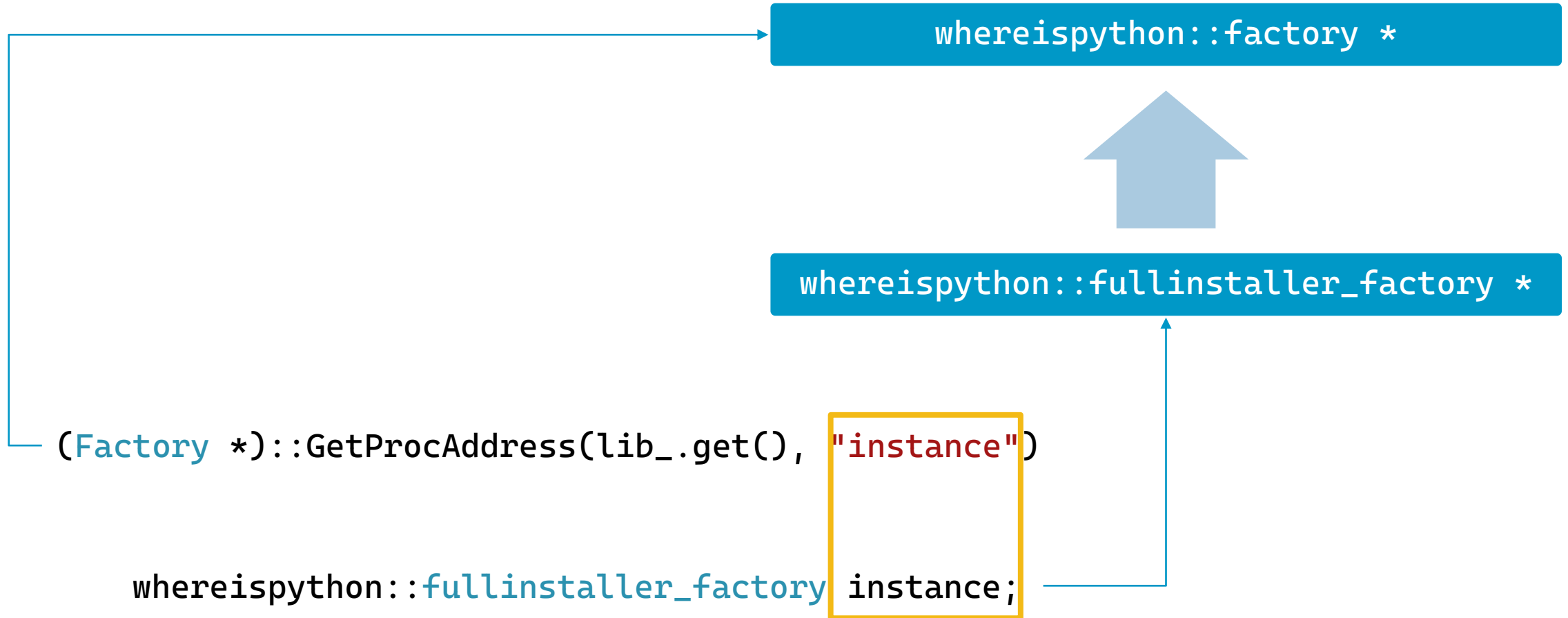
ordinal hint RVA      name
          1      0 0003F010 instance = ?instance@@3Vmicrosoftstore_fa
tstore_factory instance)
```

plugin.h namespace plugindemo

```
template <class Factory> class plugin
{
    std::unique_ptr<HMODULE, library_deleter> lib_;
    Factory *obj_;

public:
    explicit plugin(std::filesystem::path const &dll)
        : lib_([&] { /* ... */ }()),
          obj_([this] {
                if (auto pinst = (Factory *)::GetProcAddress(lib_.get(), "instance"))
                    return pinst;
                /* ... */
            })()
    {}
};
```

plugin<whereispython::factory>



openplugin.h: namespace plugin demo

- Load a list of plugins under a directory

```
auto openplugins(std::filesystem::path dir) ->  
    std::vector<plugin<whereispython::factory>>;
```

plugindemo: main function

```
for (auto &plugin : plugindemo::openplugins(fs::current_path()))
{
    if (auto python = plugin->lookup(argv[1]))
    {
        if (nonempty)
            std::cout << std::endl;
        std::cout << python->executable() << '\n';
        std::cout << python->windowed_executable() << '\n';
        nonempty = true;
    }
}
```

Demo

- Deleting fullinstaller.dll, loses the first set of answers; deleting microsoftstore.dll, loses the second set of answers
- Plug in by dragging & dropping files

```
"C:\\Program Files (x86)\\Microsoft Visual Studio\\Shared\\Python37_64\\python.exe"
```

```
"C:\\Program Files (x86)\\Microsoft Visual Studio\\Shared\\Python37_64\\pythonw.exe"
```

```
"C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.7_3.7.2544.0_x64__qbz5n2kfra8p0\\python.exe"
```

```
"C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.7_3.7.2544.0_x64__qbz5n2kfra8p0\\pythonw.exe"
```

Looking up version "3.7"

Plugins want to violate One Definition Rule

- In principle, but not necessarily on functions
- Having control over aliases exported for loading purposes would help

GCC & Clang


```
int foo asm("myfoo") = 2;
```


How dynamic is dynamic?


- Level 0: Dependent libraries
- Level 1: Delay loading
- Level 2: Foreign linkage modules
- Level 3: Plugin systems
- Level 4: Live update


Recall a typical plugin systems


- You don't want to unload any of these when GIMP is running [↩](#)


 exr-load.dll


 jp2-load.dll


 npy-save.dll


 pixbuf-save.dll


 ppm-load.dll


 rgbe-load.dll


 tiff-load.dll


 webp-save.dll


 exr-save.dll


 jpg-load.dll


 pdf-load.dll


 png-load.dll


 ppm-save.dll


 rgbe-save.dll


 tiff-save.dll


 gif-load.dll


 jpg-save.dll

 pixbuf-load.dll

 png-save.dll

 raw-load.dll

 svg-load.dll

 webp-load.dll

But you may want to unload... old code

- Calling a function in an unloaded library incurs access violation

But you may want

- Calling a function

tion

Now you come to me

**And tell me that a function
has lifetime**

Unloading is often avoided

- Apache & Nginx modules cannot be unloaded
- Python's module system does not unload Python C or C++ extensions
 - `importlib.reload` does not reload extensions
- `musl libc's dlclose` is a no-op

Relaunching the process
solves all problems

Complicating thread-local
storage (TLS) implementation
if library may unload

Better than violating the language rule, if live
objects created from extensions do not keep
the extensions alive

Can objects from a library outlive the library?

Nested lifetime

- No problem ✓

Objects can escape

- Let every object hold a strong reference to the factory object ❗
- Let the library track all objects ❗
 - You may give the library full control over the timing of unloading itself by allowing it to increment its dynamic loading reference count by 1 (see also `GetModuleHandleEx`)
- If a thread may outlive the library, read `FreeLibraryAndExitThread`
- Very difficult to debug



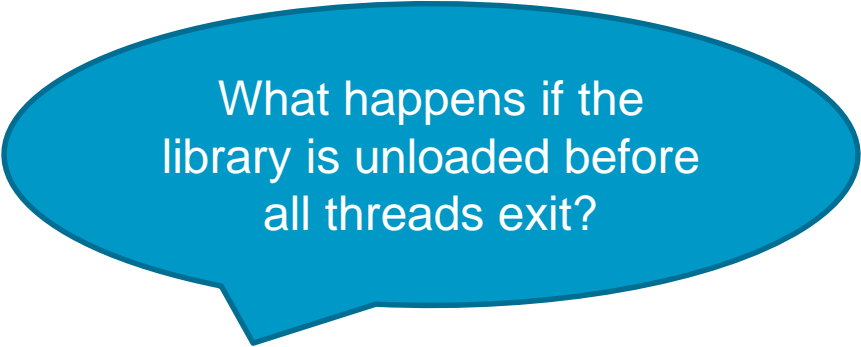
Library lifetime realized in C++

- When loading a library, initialize objects with static storage duration at namespace scope
- When unloading a library, destruct objects with static storage duration

Reminder: loaded libraries are
reference counted

Interaction with `thread_local`

- When a thread starts, initialize objects with thread storage duration at namespace scope
- When a thread exits, destruct objects with thread storage duration



What happens if the library is unloaded before all threads exit?

If destructor code is unloaded from the process address space, how to run destructors?

Let's run a quick test

```
class logger
{
    public:
        virtual ~logger() = default;
};
```

```
class singleton
{
    public:
        virtual auto get() -> logger & = 0;
};
```

Instances of the logger will be thread-specific

```
class memory_logger : public logger
{
    std::unique_ptr<char[]> buf_ = std::make_unique<char[]>(1024);

public:
    memory_logger() { /* log thread id */ }
    ~memory_logger() { /* log thread id */ }
};
```

Return such an instance

```
class memory_logger_singleton : public singleton
{
    public:
        virtual auto get() -> logger & override
        {
            thread_local memory_logger inst;
            return inst;
        }
};
```

```
EXPORT_UNDNAME tslogger::memory_logger_singleton instance;
```

Platform-specific tricks to export a variable without mangling

```
#if defined(_MSC_VER)
#pragma comment(linker, "/export:instance=?instance@@3Vmemory_logger_singleton@tslogger@@A")
#define EXPORT_UNDNAME
#else
#define EXPORT_UNDNAME __attribute__((visibility("default")))
#endif
```

Executable's main function

```
auto load = [] { return plugin<singleton>(fs::current_path() / libname); };  
auto inst = load();  
  
std::thread th[] = { /* next slide */ };  
  
for (auto &thr : th)  
    thr.join();
```

Threads using the library

th[0]

```
std::thread([&] {  
    inst->get();  
    /* after th[1] unload lib */  
})
```

th[1]

```
std::thread([&] {  
    /* after th[0] init TLS */  
    inst->get();  
    inst.unload();  
})
```


Threads using the library

th[0]

```
.....  
std::thread([&] {  
    inst->get(); ❶  
    /* after th[1] unload lib */  
})
```

th[1]

```
.....  
std::thread([&] {  
    /* after th[0] init TLS */  
    ❷ inst->get();  
    ❸ inst.unload();  
})
```

Recall what our thread-specific logger does

```
class memory_logger : public logger
{
    ...

    memory_logger()
    {
        std::cout << " + thread (" << std::this_thread::get_id() << ") attached\n";
    }

    ~memory_logger()
    {
        std::cout << " - thread (" << std::this_thread::get_id() << ") detached\n";
    }
}
```

MSVC on Windows 10

```
+ thread (7908) attached  
+ thread (18280) attached  
- thread (18280) detached
```

GCC + glibc on Linux

```
+ thread (140553905006336) attached  
+ thread (140553896613632) attached  
- thread (140553896613632) detached  
- thread (140553905006336) detached
```

Too good to be real?

Let's log static object's activities as well

```
class memory_logger_singleton : public singleton
{
    ...

    memory_logger_singleton() { std::cout << " + process attached\n"; }
    ~memory_logger_singleton() { std::cout << " + process detached\n"; }
};
```

And load it one more time after all threads exits

```
auto load = [] { return plugin<singleton>(fs::current_path() / libname); };  
auto inst = load();  
  
std::thread th[] = { /* ... */ };  
  
for (auto &thr : th)  
    thr.join();  
  
load();
```

MSVC on Windows 10

- Leaking but has the right semantics

```
+ process attached  
+ thread (15556) attached  
+ thread (10404) attached  
- thread (10404) detached  
+ process detached  
+ process attached  
+ process detached
```

GCC + glibc on Linux

- ...static variable not reinitialized?

```
+ process attached
+ thread (140172707596032) attached
+ thread (140172699203328) attached
- thread (140172699203328) detached
- thread (140172707596032) detached
+ process detached
```


- **RTLD_NODELETE**
 - Do not unload the shared object during `dlopen()`.
 - Consequently, the object's static and global variables are not reinitialized if the object is reloaded with `dlopen()` at a later time.
- **DF_1_NODELETE (elf.h)**
 - Set flag on the DSO until all `thread_local` objects defined in the DSO are destroyed
 - After the flag being cleared, a subsequent `dlopen()` unloads the DSO

`dlopen()` in the middle of destructing
`thread_local` objects is a no-op

Surprises of unloading

- Functions may have lifetime
- Implementations need to prevent objects with thread storage duration from outliving their destructors

Tools to diagnose issues in DLL

- Application Verifier
 - turn on runtime checks on executables to flag issues when dll unloads
 - the executables then can stop the debugger
- WinDbg (or WinDbg Preview)
 - look into the causes

File Edit View Help



To use AppVerifier select an application to test (Ctrl+A), click the checkboxes next to the tests, and execute the code. To view the application's properties and logs. To stop testing an application click Ctrl+D. For more detailed information see the Help.

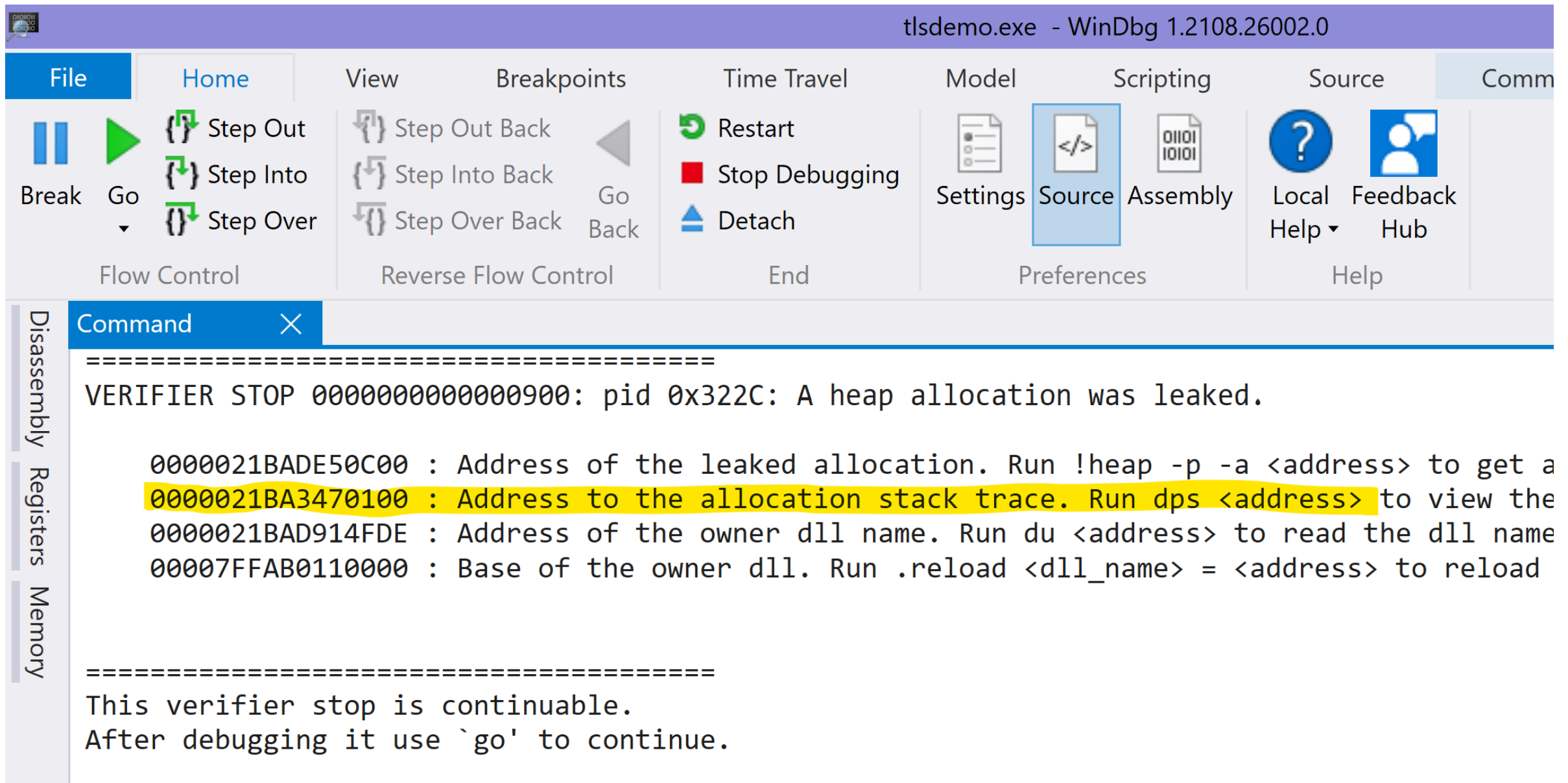
Applications

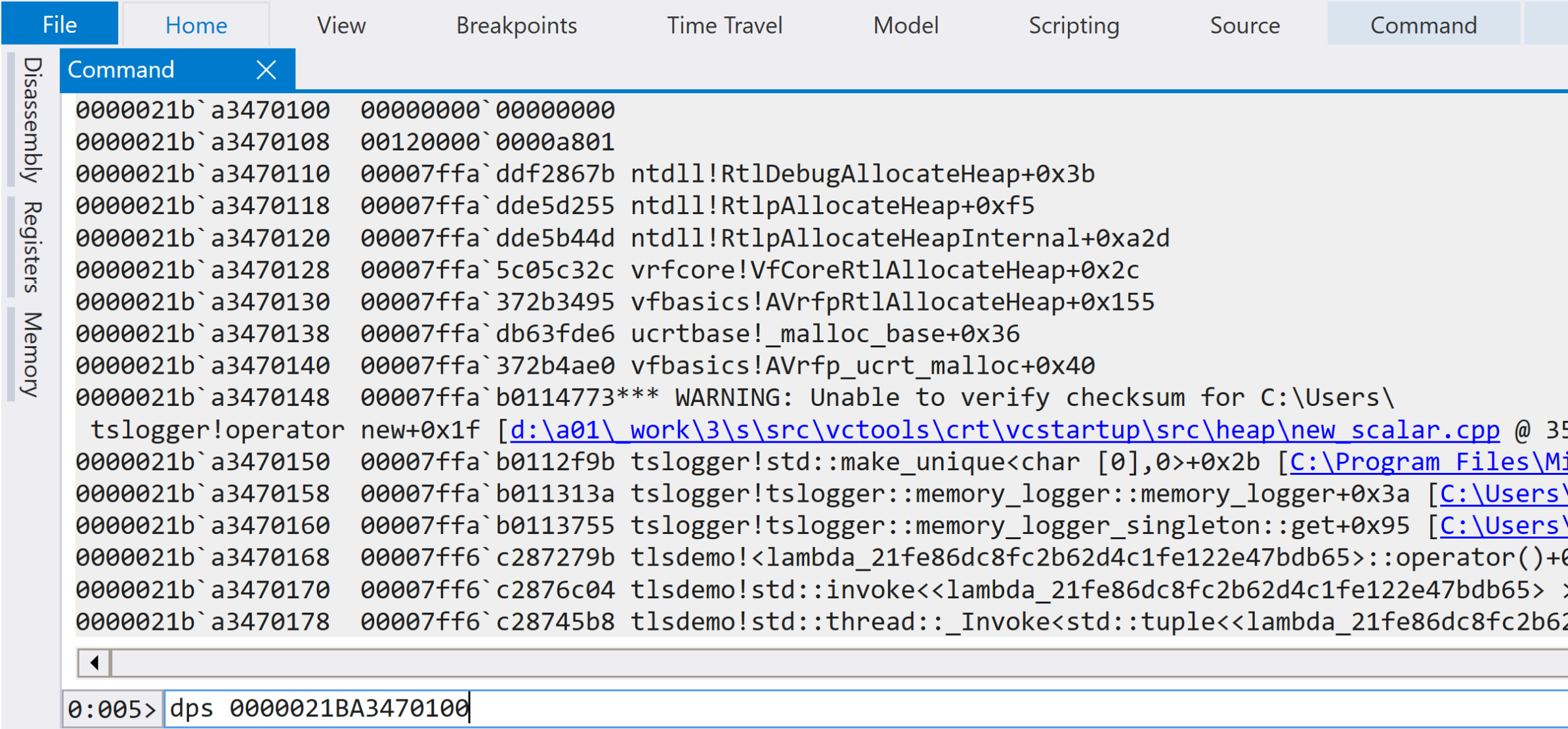
Image Name

tsdemo.exe

Tests

- ☒ Basics
- ☐ Compatibility
 - ☐ Cuzz
 - ☐ Low Resource Simulation
 - ☐ LuaPriv
- ☐ Miscellaneous
- ☐ Networking
- ☐ NTLM
- ☐ Printing
- ☐ Webservices





tlsdemo.exe - WinDbg 1.2108.26002.0

File Home View Breakpoints Time Travel Model Scripting Source Command

tslogger.cpp X

Disassembly Registers Memory

```

10 #else
11 #define EXPORT_UNDNAME __attribute__((visibility(
12 #endif
13
14 namespace tslogger
15 {
16
17 class memory_logger : public logger
18 {
19     std::unique_ptr<char[]> buf_ = std::make_un
20
21 public:
22     memory_logger()
23     {
24         std::cout << " + thread (" << std::this
25     }
26
27     ~memory_logger()

```

Command X

0000021b`a3470110	00007ffa`ddf2867b	ntdll
0000021b`a3470118	00007ffa`dde5d255	ntdll
0000021b`a3470120	00007ffa`dde5b44d	ntdll
0000021b`a3470128	00007ffa`5c05c32c	vrfco
0000021b`a3470130	00007ffa`372b3495	vfbas
0000021b`a3470138	00007ffa`db63fde6	ucrtb
0000021b`a3470140	00007ffa`372b4ae0	vfbas
0000021b`a3470148	00007ffa`b0114773***	WA
tslogger!operator new+0x1f [d:\a01\work\		
0000021b`a3470150	00007ffa`b0112f9b	tslog
0000021b`a3470158	00007ffa`b011313a	tslog
0000021b`a3470160	00007ffa`b0113755	tslog
0000021b`a3470168	00007ff6`c287279b	tlsde
0000021b`a3470170	00007ff6`c2876c04	tlsde
0000021b`a3470178	00007ff6`c28745b8	tlsde

windbg> .open -a 7ffab011313a

0:005>

Summary



Summary

- Dynamically loaded libraries are outside the standard, but useful

More dynamic, more outliers

Delay loading

- The addresses of functions may change at runtime



Foreign linkage (explicit loading)

- Functions and objects may be foreign to the program or object model, respectively



Plugins (loading multiple defs)

- Violates ODR



Live update (unloading)

- Functions may have lifetime
- Destructors with lifetime challenge TLS implementation

Summary

- Dynamically loaded libraries are outside the standard, but useful
- It's practical to create usable abstractions with them on major platforms
- Some standardization would add type safety and portability to the use cases

Questions?

 zhihaoy/dl-examples

 @lichray

