D's Import and Export Business

Benjamin Thaut

code@benjamin-thaut.de

The Goal

- Build druntime and phobos into a dll.
- Dlls as close as possible to static libraries.
- easy to use, no surprises.

dllexport / dllimport

Listing 1: C/C++

```
#ifdef DLL_EXPORTS
  #define DLL_API __declspec(dllexport)
#else
  #define DLL_API __declspec(dllimport)
#endif
```

Listing 2: D

```
export
```

dllexport / dllimport

What the C/C++ compiler sees:

a.h + b.h

```
A_API void funcA();
B_API void funcB();
```

Compiling Lib A

```
__declspec(dllexport) void funcA();
__declspec(dllimport) void funcB();
```

Compiling Lib B

```
__declspec(dllimport) void funcA();
__declspec(dllexport) void funcB();
```

Compiling Executable

```
__declspec(dllimport) void funcA();
__declspec(dllimport) void funcB();
```

What the D compiler sees:

a.d

```
module a;
export void funcA() { ... }
```

b.d

```
module b;
export void funcB() { ... }
```

$\rightarrow$ The D compiler does not know if a symbol with the export protection level is imported or exported.

Cross dll function calls

Listing 3: dll.h

```
1 #ifdef DLL_EXPORTS
2   #define DLL_API __declspec(dllexport)
3 #else
4   #define DLL_API __declspec(dllimport)
5 #endif
6 DLL_API int fnDll();
```

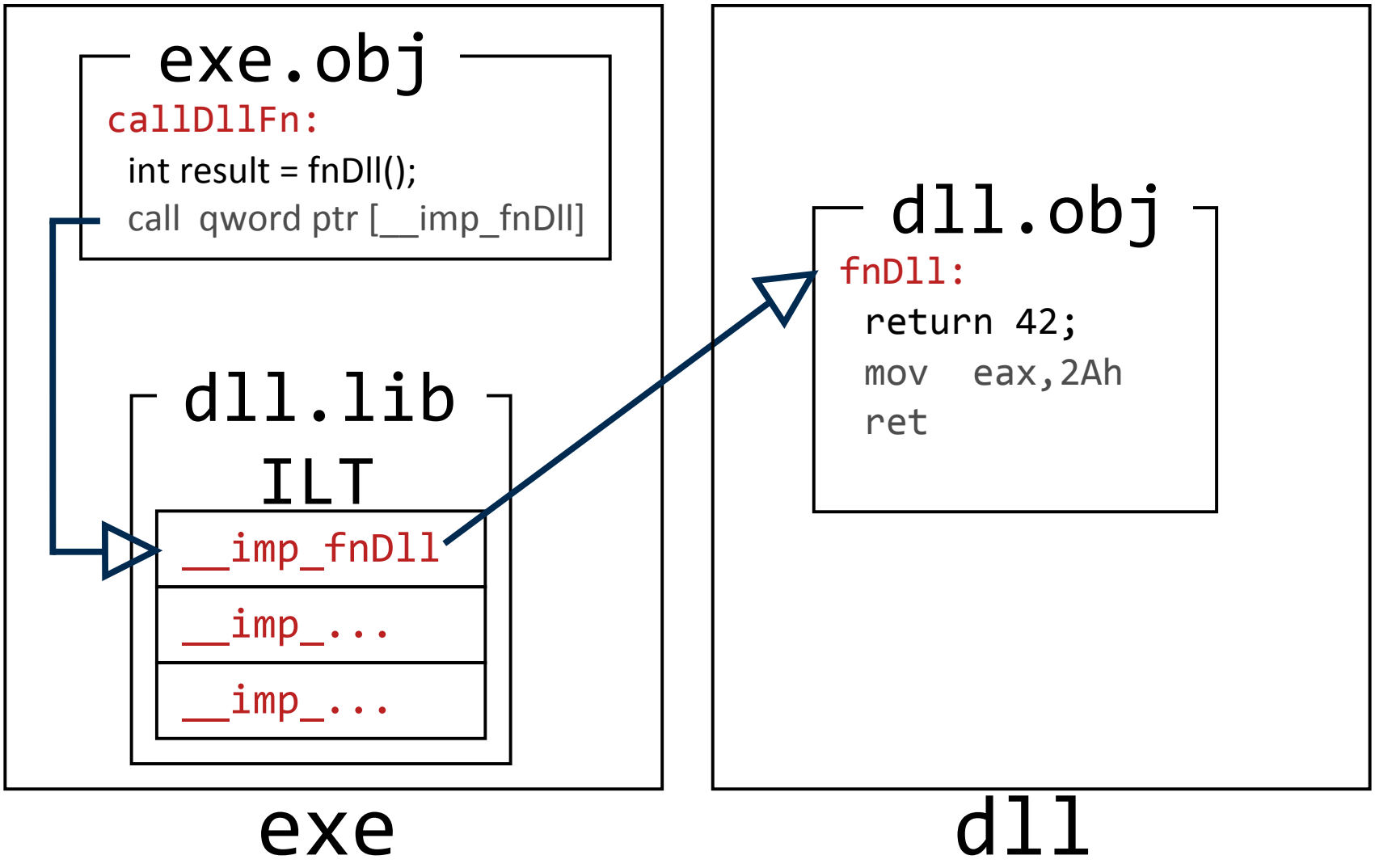
Listing 4: dll.c

```
1 #include "dll.h"
2 DLL_API int fnDll(void) { return 42; }
```

Listing 5: exe.c

```
1 #include "dll.h"
2 void callFnDll() { int result = fnDll(); }
```

Function call with dllimport

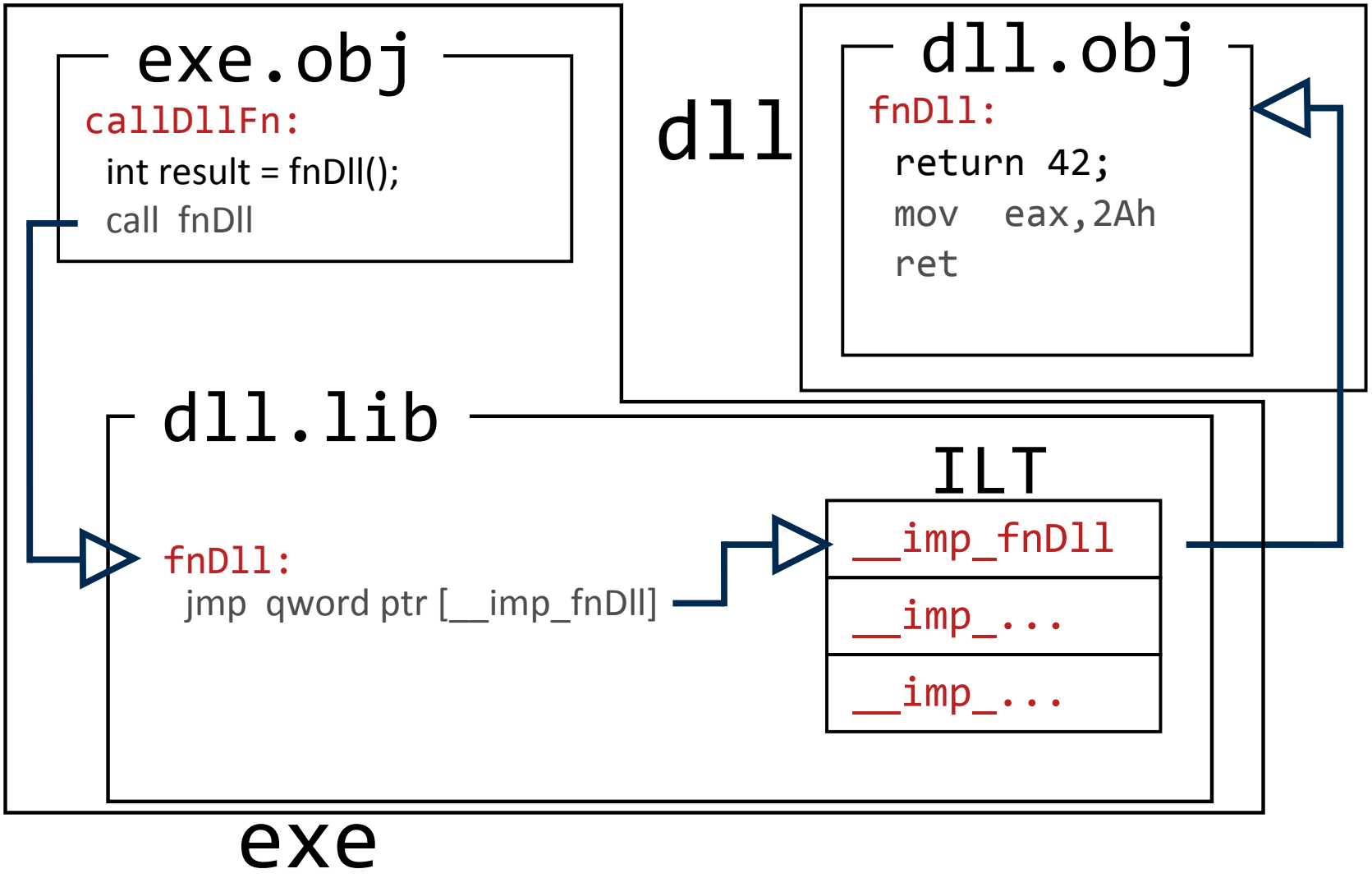


Function call without dllimport

Listing 6: dll.h

```
1 #ifdef DLL_EXPORTS
2   #define DLL_API __declspec(dllexport)
3 #else
4   #define DLL_API
5 #endif
6 DLL_API int fnDll();
```

Function call without dllimport



Cross dll data access

Listing 7: dll.h

```
1 extern DLL_API int nDll;
```

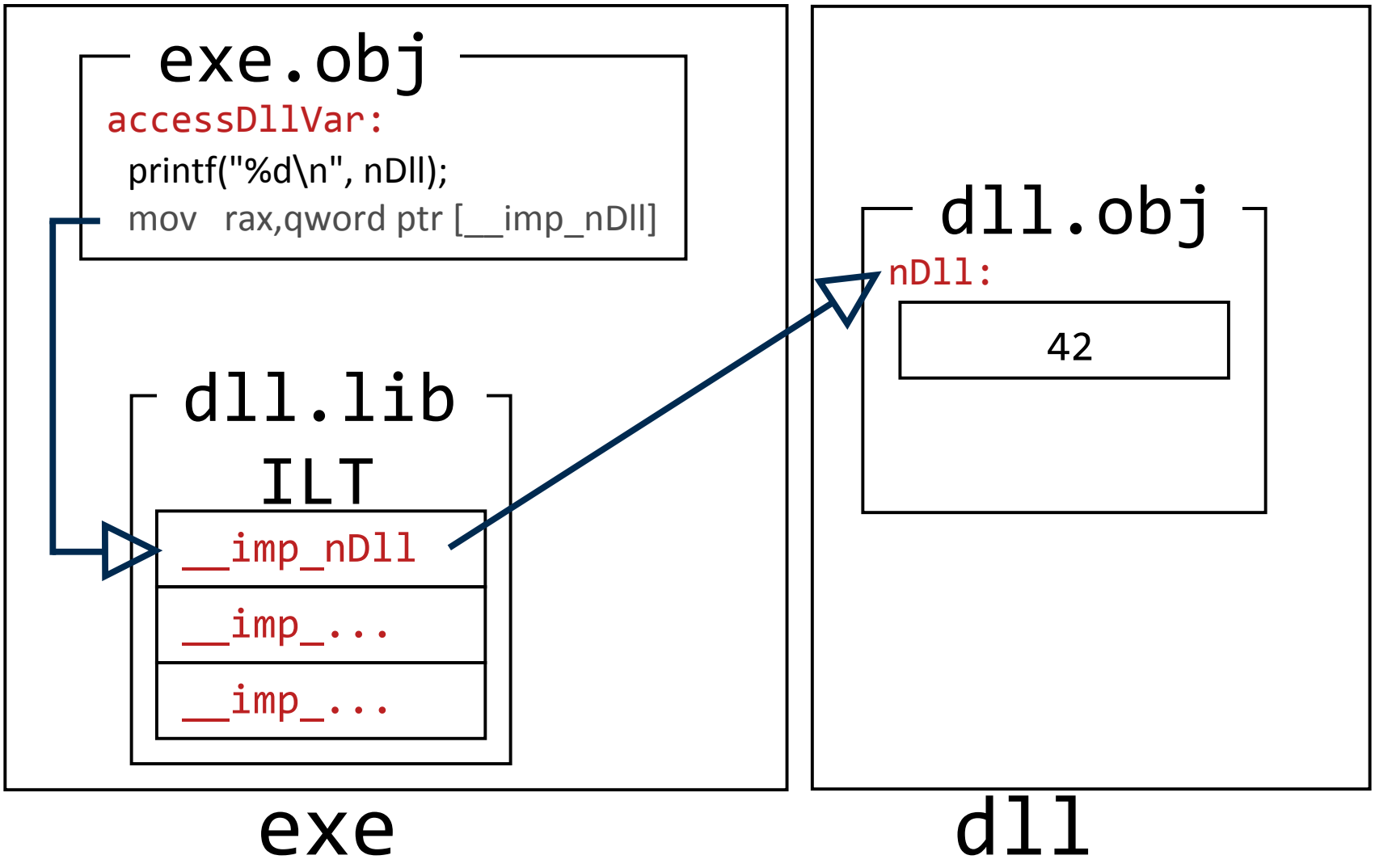
Listing 8: dll.c

```
1 DLL_API int nDll=42;
```

Listing 9: exe.c

```
1 void accessDllVar()
2 {
3   printf("%d\n", nDll);
4 }
```

Data access with dllimport



Data access without dllimport

error LNK2001: unresolved external symbol ””int nDll” (?nDll@@3HA)”.

Data Access Solution

Wanted behaviour:

1. No dlls are involved $\rightarrow$ directly access the symbol (purely static build)
2. The symbol resides in a different dll $\rightarrow$ go through __imp_ symbol
3. The symbol resides in the same binary $\rightarrow$ directly access symbol

Solution:

- For case 1 introduce the -useShared switch. When not given it is assumed that a purely static build is done.
- Always go through the __imp_ symbol. Generate __imp_ symbol manually for case 3.

Case 3 explained

Listing 10: a.d

```
1 module a;
2 import b, std.stdio;
3
4 export __gshared int varA;
5 __gshared int* __imp_varA = &varA; // compiler generated
6
7 void printVarB() { writeln(*__imp_varB); }
```

Listing 11: b.d

```
1 module b;
2 import a, std.stdio;
3
4 export __gshared int varB;
5 __gshared int* __imp_varB = &varB; // compiler generated
6
7 void printVarA() { writeln(*__imp_varA); }
```

Initializers

```
1 module a;
2
3 export __gshared int varA;
```

```
1 module b;
2 import a;
3
4 __gshared int* addrOfA = &varA; // static initializer
```

Initializers needed?

Do we really need the addresses of data symbols in initializers?

Yes we do. Its used in:

- type infos (all of them)
- vtables
- exception handling tables
- module infos

Fix during startup

```
1 module b;
2 import a;
3
4 __gshared int* addrOfA = &__imp_varA; // static initializer
5
6 void beforeDruntimeStartup()
7 {
8     addrOfB = *cast(int**)addrOfA;
9 }
```

Can be done inside DllMain.

More problems

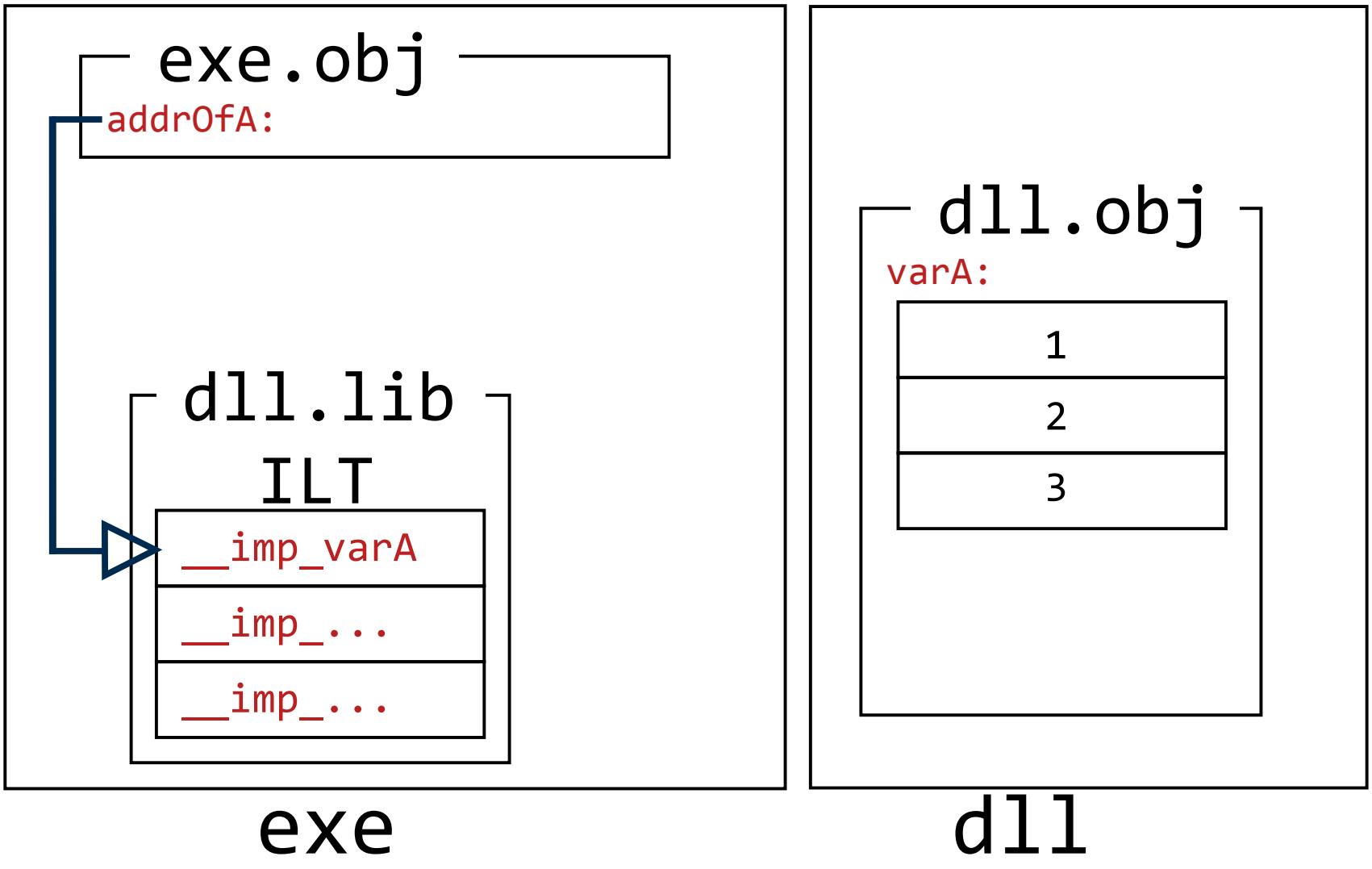
```
1 module a;
2
3 export __gshared int varA[3] = [1, 2, 3];
```

```
1 module b;
2 import a;
3
4 __gshared int* addrOfA = varA.ptr + 2; // static initializer
```

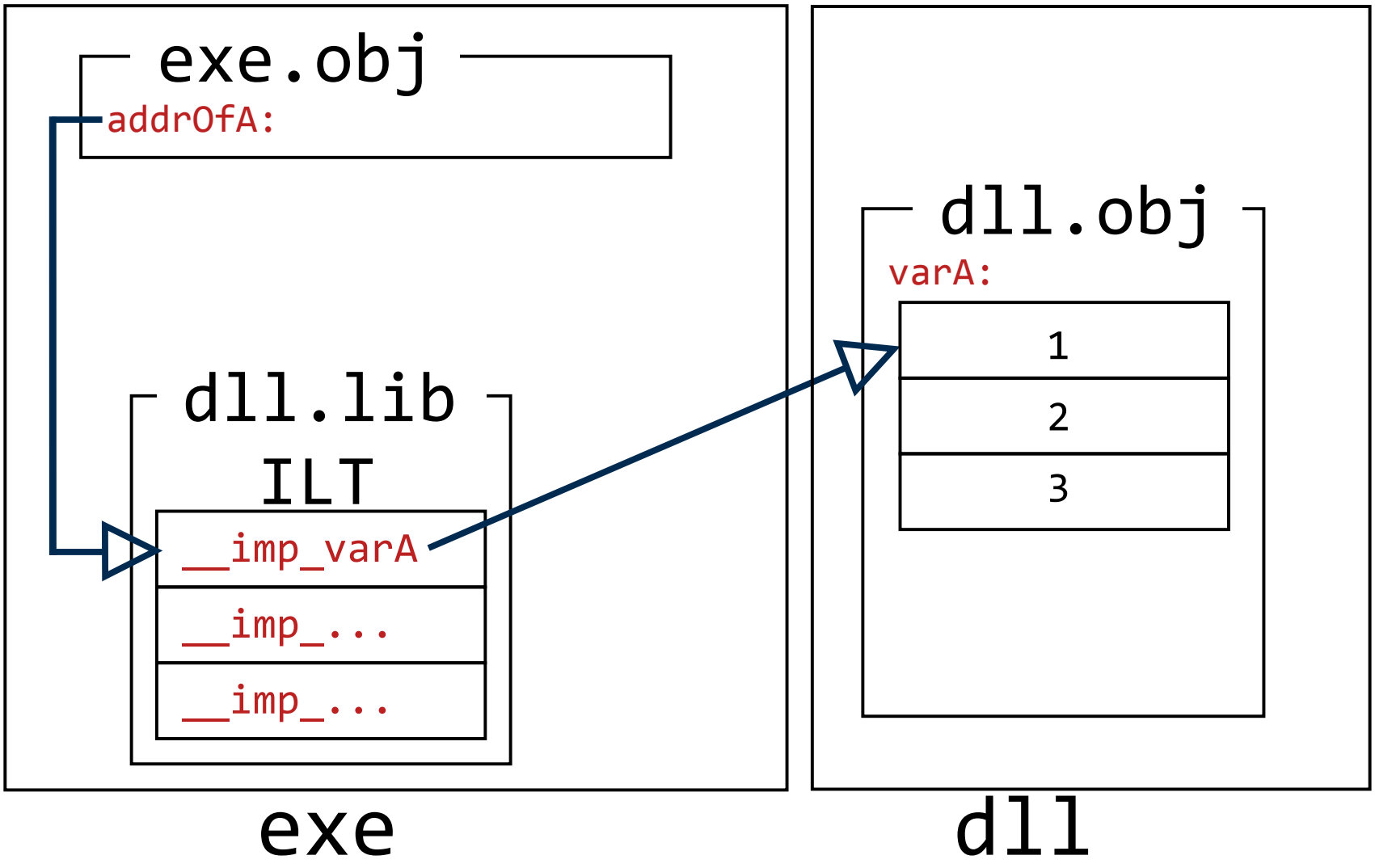
Fix during startup

```
1  module b;
2  import a;
3
4  // __gshared int* addrOfA = varA.ptr + 2;
5  __gshared int* addrOfA = &__imp_varA; // static initializer
6
7  struct FixupEntry
8  {
9    void** pointerToFix;
10   size_t offset;
11 }
12
13 FixupEntry fixupTable[] = [ FixupEntry(&addrOfA, 2 * int.sizeof) ];
14
15 void druntimeStartup() {
16   foreach(ref fixup; fixupTable) {
17     *fixup.pointerToFix = (**cast(void***)fixup.pointerToFix) + fixup.offset;
18   }
19 }
```

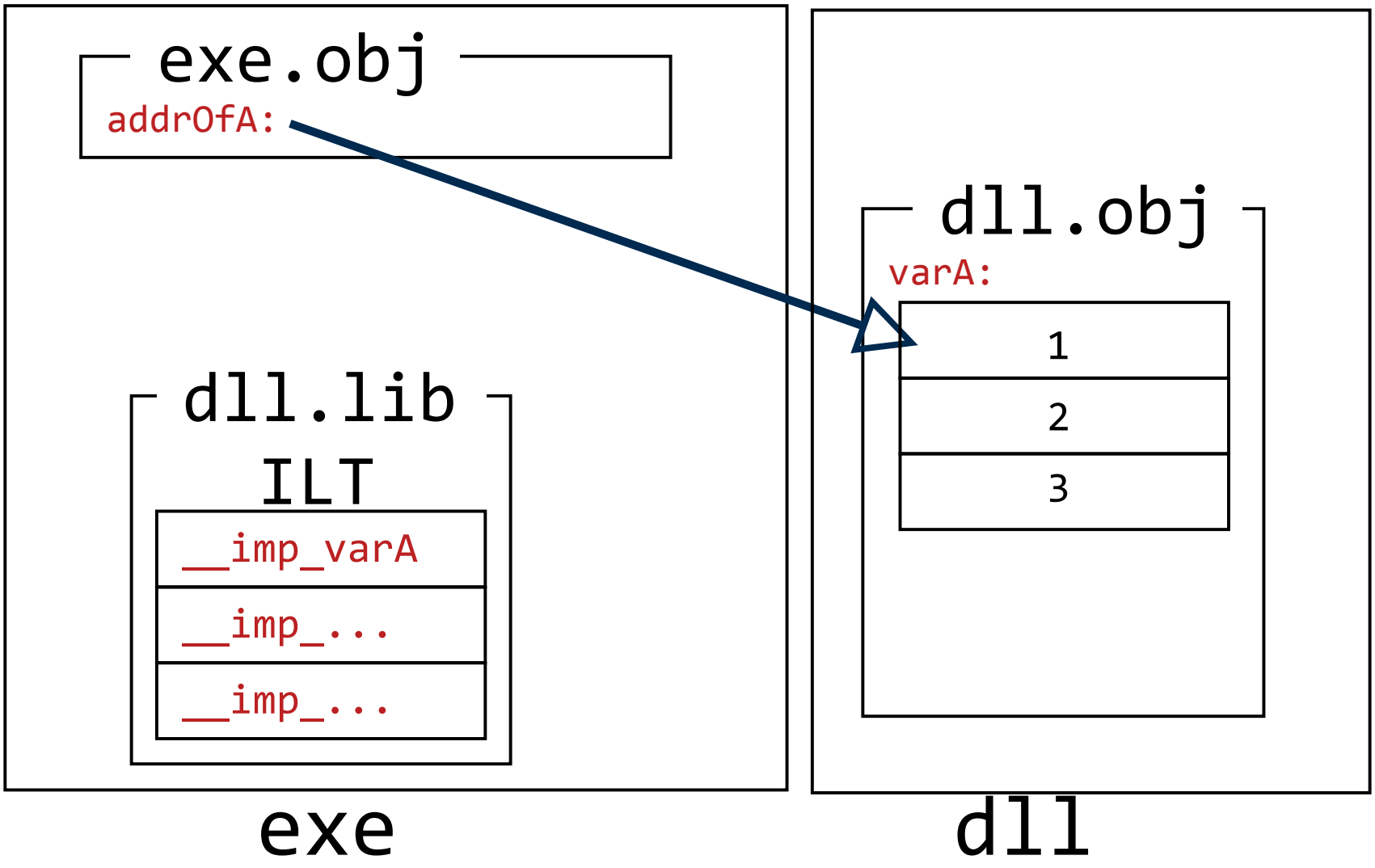
Fix during startup



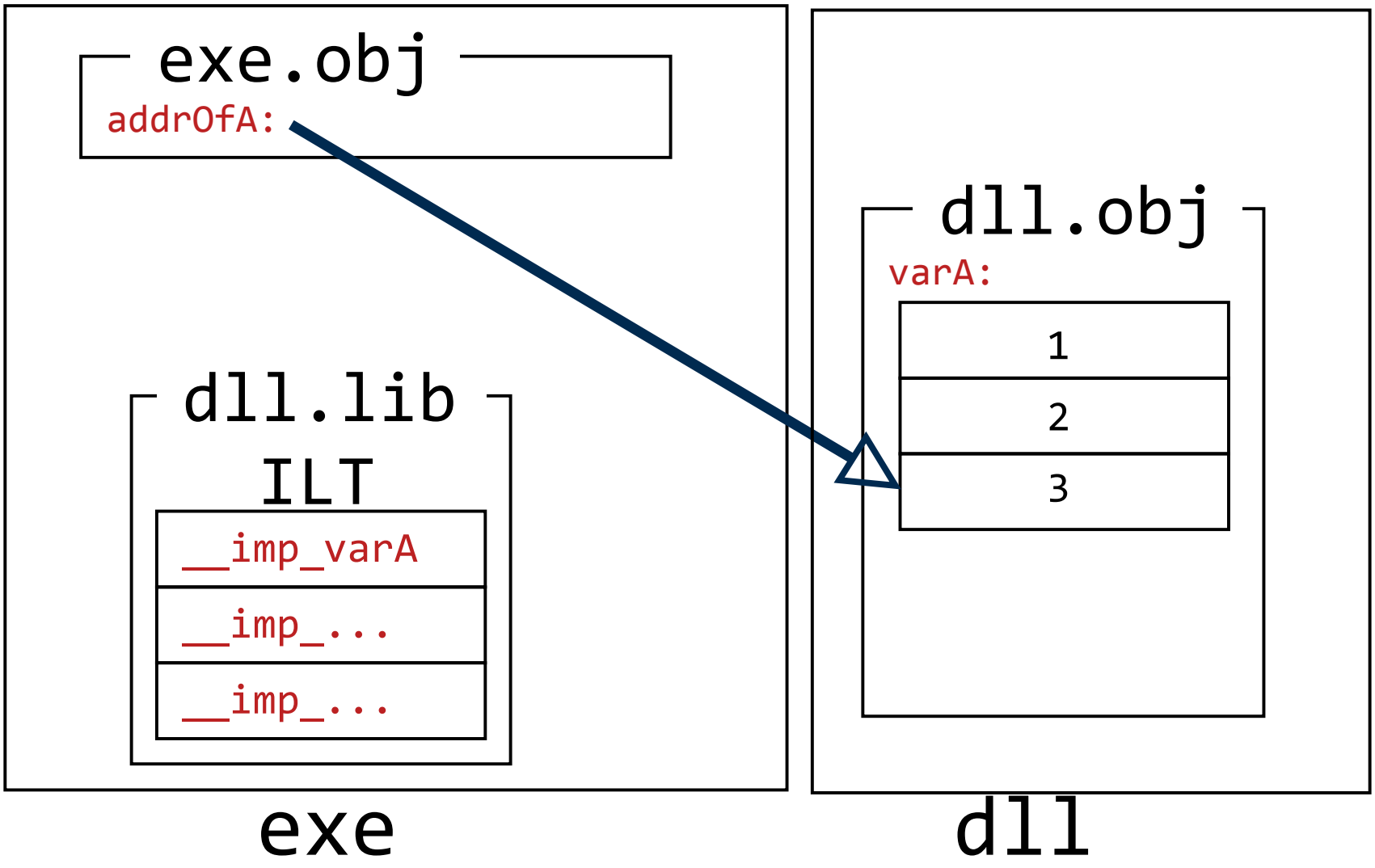
Fix during startup



Fix during startup



Fix during startup



Size Optimizations

- Unused symbols are pulled in.
- Use associated comdats.
- Comdats increase binary size significantly.
- Use relative offsets in 64-bit. (C++ uses code 14-bytes in x64)

What should be exported?

Language Reference

Export means that any code outside the executable can access the member. Export is analogous to exporting definitions from a DLL.

What should be exported: Modules

```
1 module dllModule;
2
3 void someTemplate(T)(T x)
4 {
5   assert(x > 0);
6 }
```

- Module Info
- Module Assert
- Module Unittest
- Module Array Bounds Check

$\rightarrow$ always export module symbols.

What should be exported: Classes

```
1  module dllModule;
2
3  export class SomeClass
4  {
5    public void foo() { fooImpl(); }
6    protected void bar() { ... }
7    private void fooImpl() { ... }
8
9    protected struct Data
10   {
11     public void dataFunc() { ... }
12   }
13 }
```

- Virtual Function Table
- Type Info

What should be exported: Voldemord Types

```
1 module dllModule;
2
3 export auto makeVoldemort()
4 {
5   struct Voldemort { ... }
6   return Voldemort;
7 }
```

What should be exported: Templates

```
1 module dllModule;
2
3 export struct Data(T)
4 {
5   T data;
6 }
```

```
1 module dllModule;
2
3 template Data(T)
4 {
5   export struct Data
6   {
7     T data;
8   }
9 }
```

Export can’t be a protection level

```d
module dllModule;

void write(T)(FILE* file, ref T data)
{
  writeImpl(file, &data, T.sizeof);
}

private void writeImpl(FILE* file, void* data, size_t dataSize)
{
  fwrite(data, dataSize, 1, file);
}
```

Occurs twice in object.d. Over 50 times in phobos (so far).

Contra Argument

All functions / types that must be accessible across a dll boundary should be public because you can access them through GetProcAddress anyway.

Contra Contra Argument

All members of structs / classes should be public because they can be accessed through pointer arithmetic.

```
1
2  struct Data
3  {
4    void setData(int value) { m_data = value; }
5
6    private int m_data;
7  }
8
9  int getData(ref Data d)
10 {
11   return *cast(int*)((cast(void*)&d) + 0);
12 }
```

Pro Argument

Export should be an attribute.

- Does not work at the moment for most cases.
- No projects on GitHub that use it.
- Not even used in VisualD.
- Making it an attribute would only incur very little breakage.
- Would separate visibility concerns.
- This issue is keeping me from doing the pull request.

Unsolved Issue: Testing Dll Libraries

```
1  module dllModule;
2  auto someAlgorithm( ... )
3  {
4    ...
5  }
6
7  unittest
8  {
9    assert(someAlgorithm(someInput) == expectedOutput);
10 }
```

Can't test if all required symbols are exported.

What to do?

- Extract unittests?
- Write new ones?

Unsolved Issue: Template de-duplication

```d
module a;
struct Data(T)
{
  __gshared T g_var = 0;
}
```

```d
module b;
import a, std.stdio;

export void print()
{
  writeln(Data!int.g_var);
}
```

```d
module c;
import a, b, std.stdio;

void main(string[] args)
{
  Data!int.g_var = 5;
  print();
  writeln(Data!int.g_var);
}
```

Unsolved Issue: TLS

```d
module a;

export int g_varA; // ends up in TLS
```

```d
module b;
import a, std.stdio;

void main(string[] args)
{
  writeln("%d", g_varA); // will result in linker error
}
```

Makes static and shared libraries different.

Unsolved Issue: Distribution

DIls actually link against a version of the c-runtime. What do we ship?

- Debug / Release
- VS 2008, VS 2010, VS 2013, VS 2015

Shipping all permutations precompiled would be aprox. 100 MB.

Pull Request - Uphill Battle

- C++ $\rightarrow$ D transition
- Refactoring of dt.c
- Keeping up with changes in druntime / phobos.
- Support for VS 2015.
- I want to get the pull request in so others can help.

https://github.com/Ingrater/dmd/tree/DllSupportD
https://github.com/Ingrater/druntime/tree/DllSupport70
https://github.com/Ingrater/phobos/tree/DllSupport70

Thank you!

Questions?

code@benjamin-thaut.de

Dll Main

Before:

```
1 module dllmain;
2 import core.sys.windows.dll;
3
4 mixin SimpleDllMain;
```

Now:

```
1 module dllmain;
2 import core.sys.windows.dll;
3
4 mixin SimpleDllMain!(DllIsUsedFromC.no);
```

```
1 module dllmain;
2 import core.sys.windows.dll;
3
4 mixin SimpleDllMain!(DllIsUsedFromC.yes);
```

Binary Sizes

- druntime debug: 2.2 MB
- phobos debug: 5 MB
- druntime release: 1.2 MB
- phobos release: 2.7 MB
- size of phobos fixup table: 10 kb
- hello world size: 10 kb (vs. 339 kb)

Example compilation

```
1 dmd -m64 -shared -L/IMPLIB:impa.lib a.d dllmain.d
       -defaultlib="druntime64s.lib" phobos64s.lib
2 dmd -m64 -shared -L/IMPLIB:impb.lib b.d dllmain.d
       -defaultlib="druntime64s.lib" phobos64s.lib
3 dmd -m64 -useShared c.d -defaultlib="druntime64s.lib" phobos64s.lib
       impa.lib impb.lib
```