Using Dynamic Libraries and Plugins in C++

Thomas Witt
witt@styleadvisor.com

April 20. 2006

Welcome to the basement of software construction

Dynamic Shared Object

- Various names
  - Dynamic Library
  - Dll
  - Shared Library
  - Plugin
  - Module
- A means of packaging executable code
  - More executable than library

Dynamic Linking

- Defer final program assembly until load time
- Benefits:
  - Shared Libraries
  - Delay loading
  - Explicit loading
  - Easily serviceable libraries (well…)
- Costs:
  - Increased startup time

What the standard says

...

Terminology

- Translation unit (C++03)
  - Preprocessed source file
- Compiled Translation Unit
  - Object file
- Symbol
  - Named entity in an compiled translation unit
- Load unit
  - Executable, dynamic library

Terminology (Cont.)

- Direct dependencies of a load unit
  - Other load units available at static linktime
- Dependencies of a load unit
  - Transitive closure of direct dependencies
- Load set
  - Primary load unit and its dependencies

Compilation Model

- The compiler operates on individual translation units

- The static linker operates on set of compiled translation units and their direct dependencies

---

- The dynamic linker operates on load sets

Linker/Loader

- Operating system facility
- Task: Create a runnable program in memory from compiled translation units
- Linker
  - Symbol binding
  - Relocation
- Loader
  - Program loading
  - Relocation

Object File Format

- Linker meta information
  - Header
  - Relocations
  - Symbols
  - Runtime dependencies
- Code
- Text (data)
- Debugging information

Object File Format

- Common Object File Format (COFF)

  System V, AIX (XCOFF)

- Portable Executable Format (PE)

  Windows, BeOS R3

- Executable and Linking Format (ELF)

  System V R4, Solaris, IRIX, BSD, Linux, BeOS

- Mach object file format (Mach-O)

  Mac OS X, Darwin, NeXTSTEP

Symbol Visibility

- Visibility at load unit level
  - Distinct from linkage that works per translation unit (static anybody?)
- External visibility
  - Symbols are visible and accessible from other load units
- Internal visibility
  - Symbols are neither visible nor accessible from other load units

Symbol Visibility (PE)

- Internal visibility by default
- Export symbol tables
- __declspec(dllexport)
  - Marks symbols as externally visible
- __declspec(dllimport)
  - Improves code generation for imported symbols

Symbol Visibility (ELF)

- External visibility by default
- __attribute__((visibility(option)))
  - default -> external visibility
  - hidden -> internal visibility
  - internal
  - protected
- #pragma
- Compiler switch
- Symbol tables

Symbol Lookup (PE)

- Symbols are bound to load units at static link time
  - Import libraries
  - Load units can forward symbols
- Symbols are resolved on first use
- Lookup by
  - Hint and name
  - Ordinal

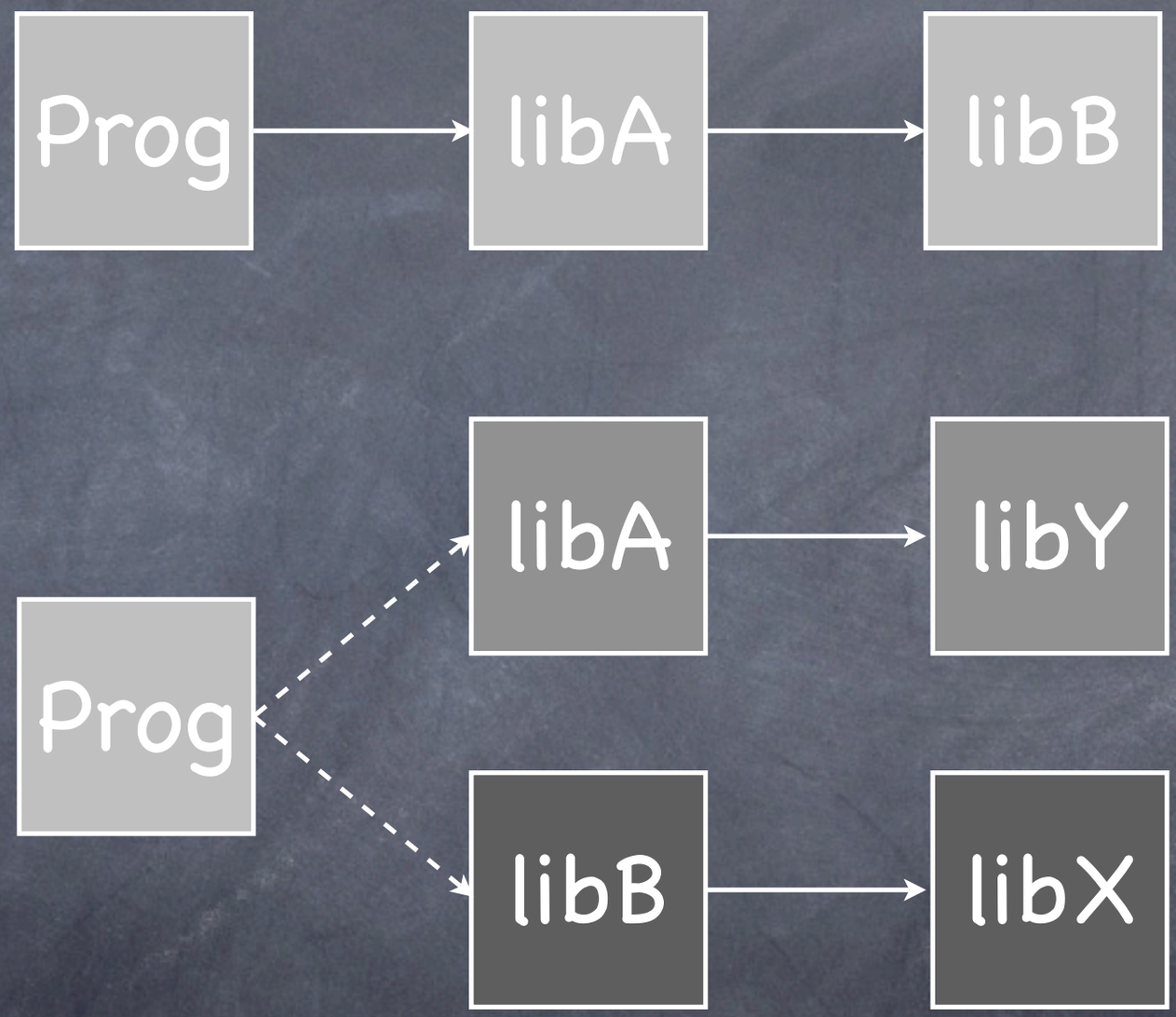
Symbol Lookup (ELF)

- Symbols are bound to load units at dynamic link time by default
  - Symbol interposition
- Symbols can be bound to load units at static link time
  - Direct binding
  - Interposers
- Lookup by name
  - Hash table

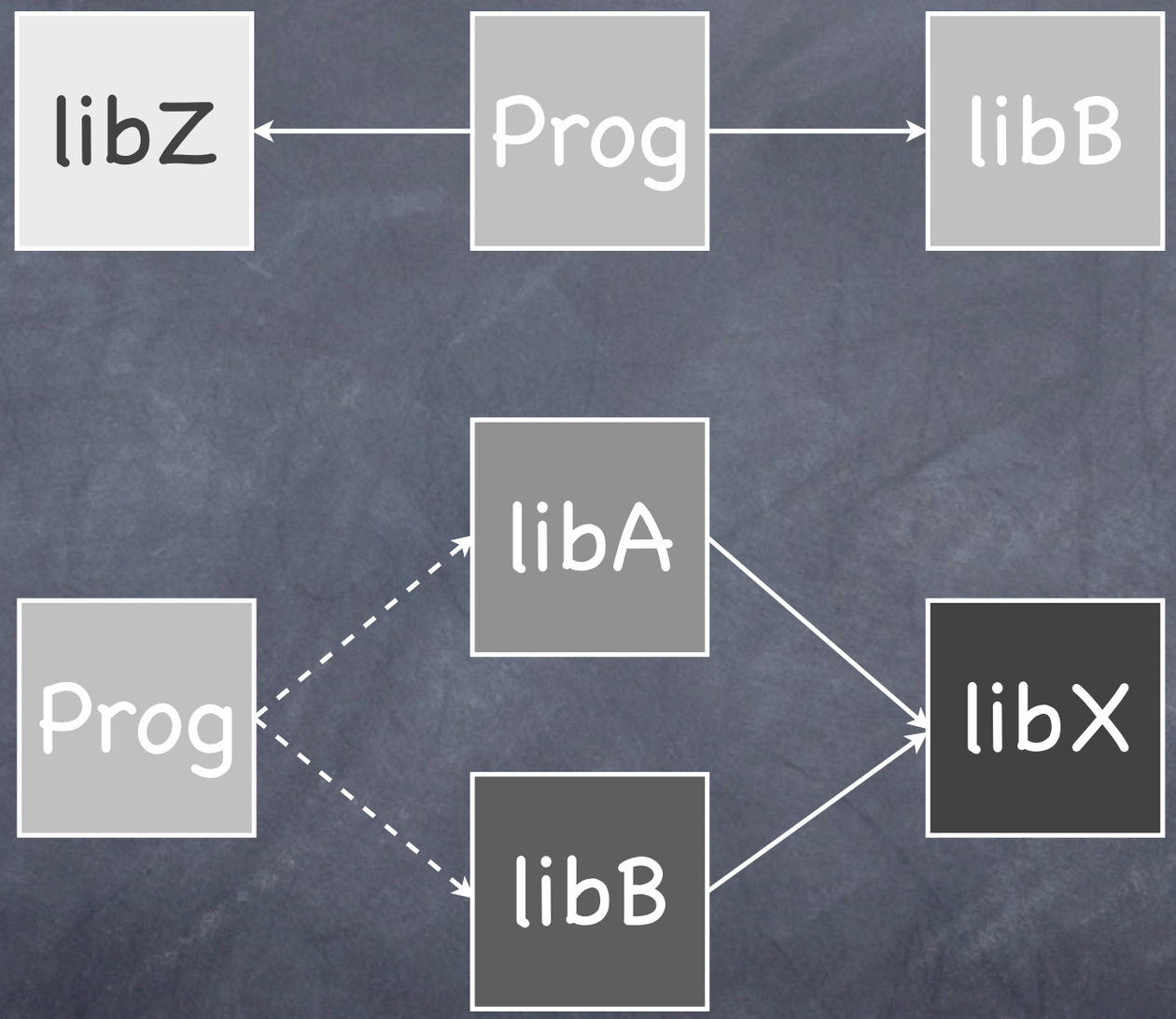
Symbol Lookup (ELF)

- Search scope
  - world
  - group
- Visibility (ELF)
  - world
  - local
- Load units are searched in load order

Symbol Lookup (ELF)



Symbol Lookup (ELF)



Symbol Lookup (Mach-O)

- Symbols are bound to load units at static link time by default
  - Two level namespace
  - Flat Namespace
- Otherwise like ELF

C++

finally

Issues

- DSO compatiblity
  - ABI configuration
  - ABI stability
  - Versioning
- Packaging
  - Encapsulation
  - Symbol resolution

ABI

- Application Binary Interface
  - Interface for compiled object code
- Components
  - Memory layout
  - Function call interface
  - Compiler generated objects
  - Name mangling
- Itanium C++ ABI 58 pages
  - C++ Standard Library not covered

Name Mangling

- Mapping C++ names to symbol names
  - Overloaded functions
  - Templates
  - Namespaces
- namespace bar { int foo(double); }
  - Microsoft

    ?foo@bar@@YAXN@Z
  - GCC 4.0.1
    - _ZN3bar3fooEd

Extern "C"

- extern "C" gives C-Linkage to a declaration
  - Simpler C name mangling rules
  - C calling conventions
- C-Linkage reduces the available feature set.

ABI Stability

- Rules for dynamic library APIs according to Apple (C++ Runtime Environment Programming Guide 2005-08-11)
  - Avoid inheriting from Classes in the standard C++ library
  - In your class definition, avoid member variables whose type is a class defined by the C++ runtime
  - Avoid using standard C++ runtime classes for arguments and return values.

Compiler Generated

- Class members
- Inlines
- Template Instantiation
  - Extern Template
- Run Time Type Information (RTTI)
- Virtual Function Table
- Compiler generated objects do not have a clear binding to a single translation unit
- Collapsing requires load set analysis

RTTI

- Application
  - typeid()
  - Exception handling
  - dynamic_cast (polymorphic types)
- RTTI might be emitted in multiple object files

More Possible Duplicates

- Global operator new
  - Requires symbol interposition
- Static objects
  - Templates
  - Inlines
- Inlines
  - Address of

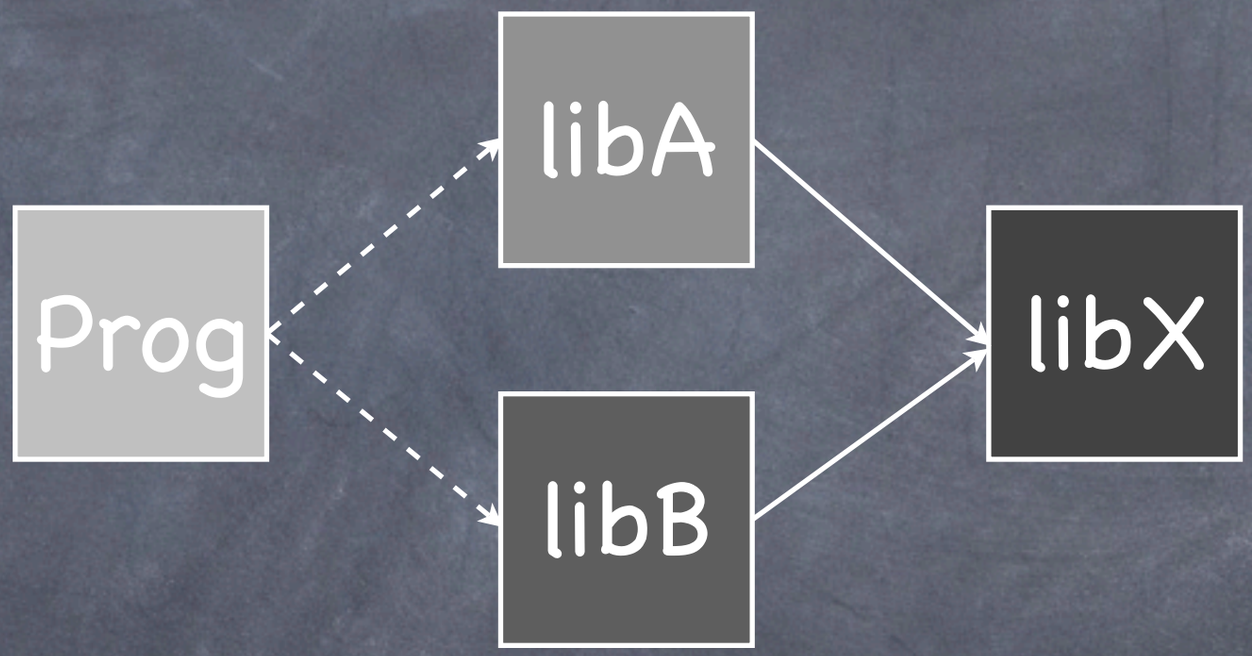
Modules

- Load semantics
  - Thread local storage (PE)
- ABI stability
  - Modules should work across application releases
- Encapsulation
  - Name clashes
  - Memory management

Programmatic Loading

- Loading load units at runtime
  - dlopen, dlsym
    - ELF
    - Mach-O
  - LoadLibrary, GetProcAddress
    - PE
- Modules
- Plug-Ins

Symbol Lookup (ELF)



Modules

- C-Interface
- Control memory allocation
  - Do not expose allocated memory
  - Provide deallocation API
- Link statically
  - Alas, some libraries are dynamic only

Using C++ to implement modules today is unsafe in some scenarios

Application Startup

- Relocations
  - Preferred load address (PE, ELF)
  - Position independent code (ELF, Mach-O)
- Symbol binding
  - Reduce symbol table size
  - Shorten symbol names (ELF, Mach-O ?)

Summary

- Dynamic shared objects form integral part of modern program design
- C++03 is oblivious to DSOs
- C++ is not very well suited for packaging in independently compiled DSOs
- Using DSOs may require intimate plattform knowledge
- Poor portability
- C++0x may (shall ?) address DSOs

References

- John R. Levine
  "Linkers and Loaders"
  Morgan-Kauffman, Oct 1999, ISBN 1-55860-496-0
- Ulrich Drepper
  "How To Write Shared Libraries"
  Jan 22 2005
- "Itanium C++ ABI ($Revision: 1.86$)"
  http://www.codesourcery.com/cxx-abi/abi.html
- "Linker and Libraries Guide"
  Sun Microsystems, Inc., December 2003

References

- Benjamin Kosnik
  "Dynamic Shared Objects: Survey and Issues"
  Nov 2 2005
- Matt Austern
  "N1400 Toward standardization of dynamic libraries"
  Sep 25 2002
- "Mac OS X ABI Mach-O File Format Reference"
  Apple Computer, Inc., Mar 8 2006
- "Mach-O Programming Topics"
  Apple Computer, Inc., Nov 9 2005
- "Dynamic Library Programming Topics"
  Apple Computer, Inc., Feb 7 2006

References

- Russ Osterlund
  "What Goes On Inside Windows 2000: Solving the Mysteries of the Loader"
- Mark Pietrek
  "An In-Depth Look into the Win32 Portable Executable File Format"
- Mark Pietrek
  "An In-Depth Look into the Win32 Portable Executable File Format, Part 2"
- Mark Pietrek
  "Optimizing DLL Load Time Performance"
- Kang Seonghoon
  "Microsoft C++ Name Mangling Scheme"
  Version 1.1, Nov 25 2005