Dynamic libraries and how to optimize them

C++ Russia 2024

About me

- Yuri “yugr” Gribov
- Compiler engineer
- Gmail: tetra2005
- t.me/the_real_yugr
- https://github.com/yugr
- https://www.linkedin.com/in/yugr/

Plan of the talk

- Dynamic libraries

Plan of the talk

- Dynamic libraries
    - Differences from static libraries
    - Work principles
    - Pros and cons

Plan of the talk

- Dynamic libraries
  - Differences from static libraries
  - Work principles
  - Pros and cons
- Comparison of Linux and Windows implementations

Plan of the talk

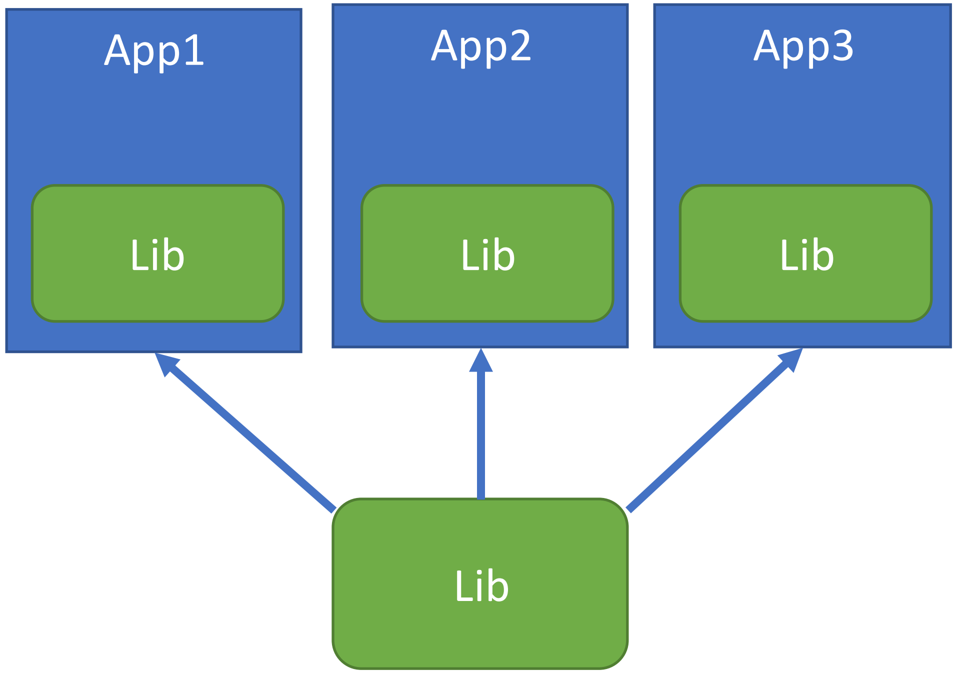
- Dynamic libraries
  - Differences from static libraries
  - Work principles
  - Pros and cons
- Comparison of Linux and Windows implementations
- Speeding up dynamic libraries

Plan of the talk

- Dynamic libraries
  - Differences from static libraries
  - Work principles
  - Pros and cons
- Comparison of Linux and Windows implementations
- Speeding up dynamic libraries
  - Overheads
  - And ways to reduce them

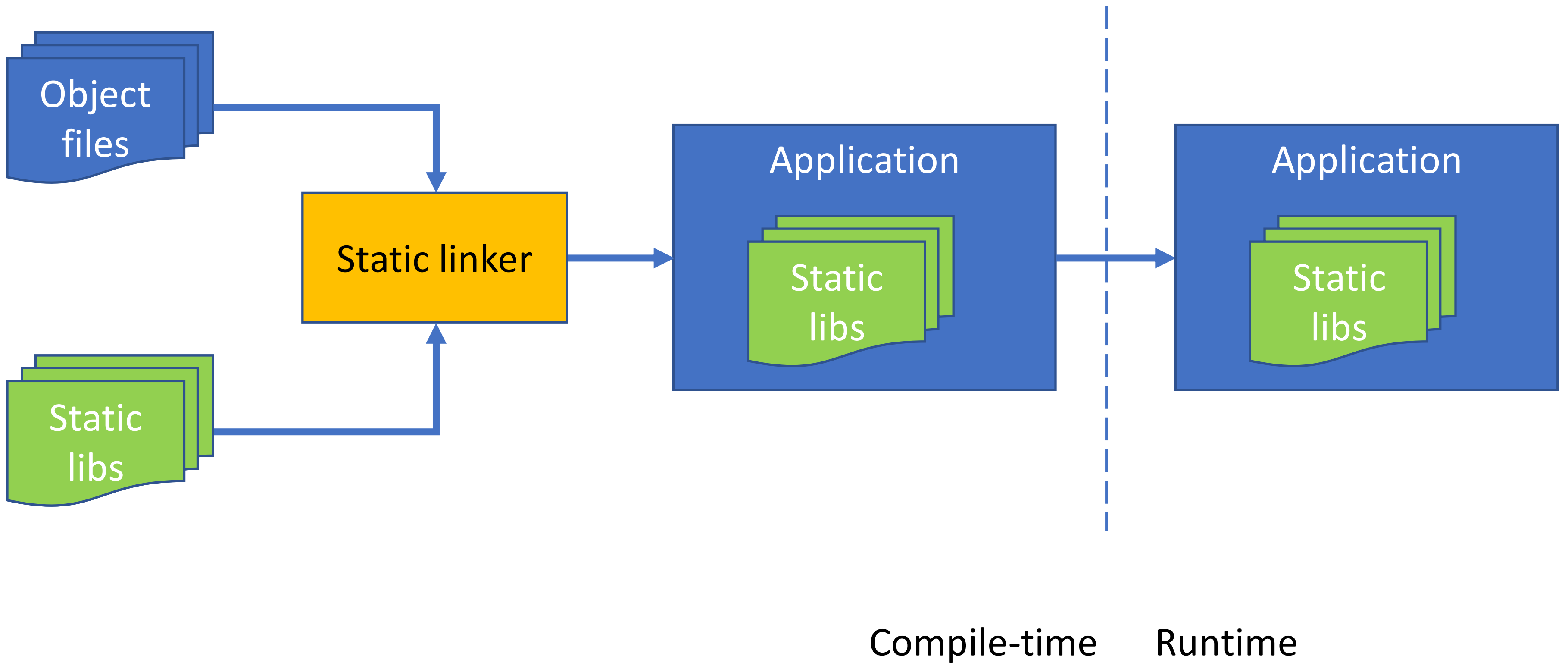
Libraries

- Archives of reusable code
- Can be reused in multiple programs
- Depending on library link time can be
  - Static (.a, .lib)
  - Dynamic (.so, .dll, .dylib)
- All popular platforms support both
  - Windows, Linux, macOS, BSDs



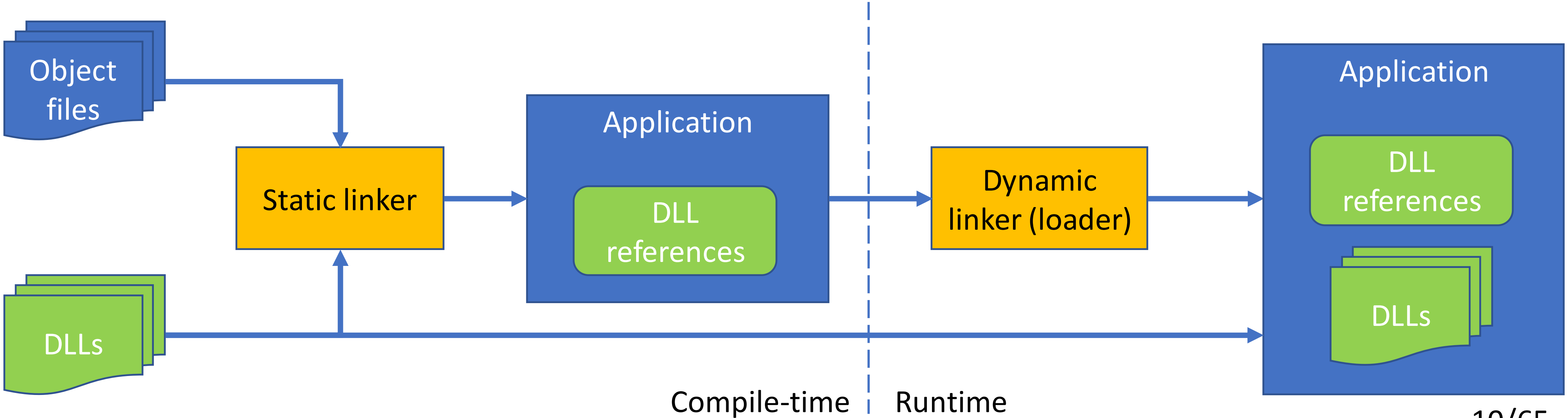
Static libraries

- Become part of executable file at link-time



Dynamic libraries

- Dynamic-link libraries (DLL), shared libraries, shared objects
- Not part of program executable file
- (Usually) loaded at program startup



Using dynamic libraries

- Two main approaches:
  - Traditional, link-time
    ```
    gcc program.o -lgmp
    link.exe program.obj libgmp.lib
    ```
  - Run-time loading (dynamic loading)
    ```
    void *lib = dlopen("libgmp.so", RTLD_LAZY | RTLD_GLOBAL);
    HANDLE lib = LoadLibrary("libgmp.dll");
    ```
- With traditional approach library will be loaded at program startup
- With runtime loading – at any time, in any point in program
  - Enables lazy loading, plugins, etc.

DLL advantages

- RAM and disk savings
  - ~1.1G RAM on my Ubuntu Desktop[1,2] (with running Firefox/KOffice/Thunderbird)
  - ~10G HDD on my Ubuntu Desktop (with Firefox, KOffice, etc.)
- Faster system updates
  - No need to recompile dependent executables on minor library updates
- Support for interesting work scenarios:
  - Lazy loading
  - Extend program functionality with user plugins
  - Load different library versions depending on environment (e.g. on processor capabilities)

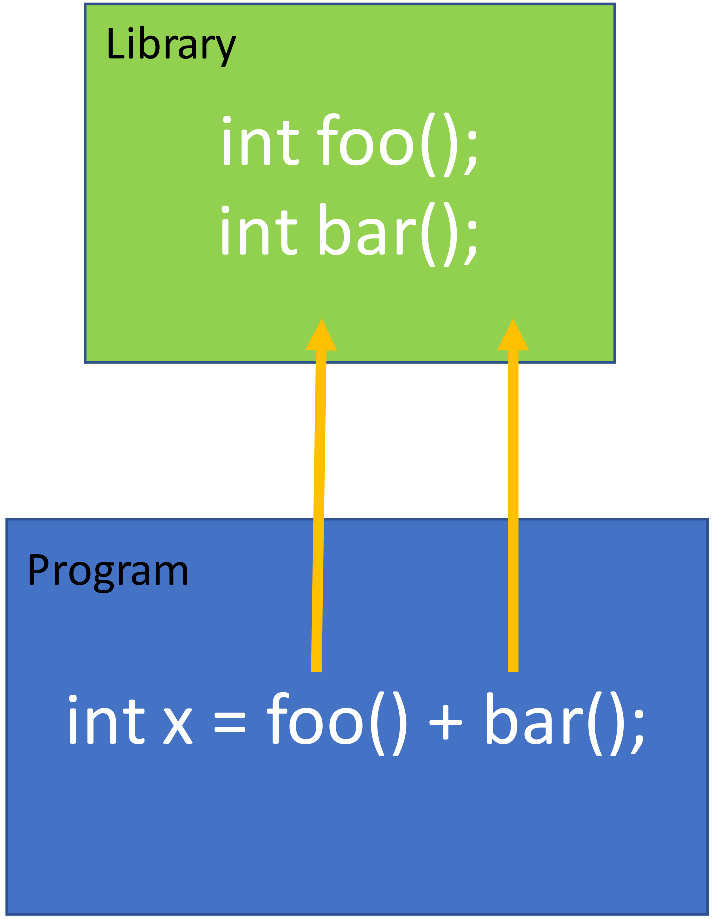
1) Experiment details are available in additional slides at https://github.com/yugr/CppRussia/tree/master/2024

2) Based on https://zvrba.net/articles/solib-memory-savings.html

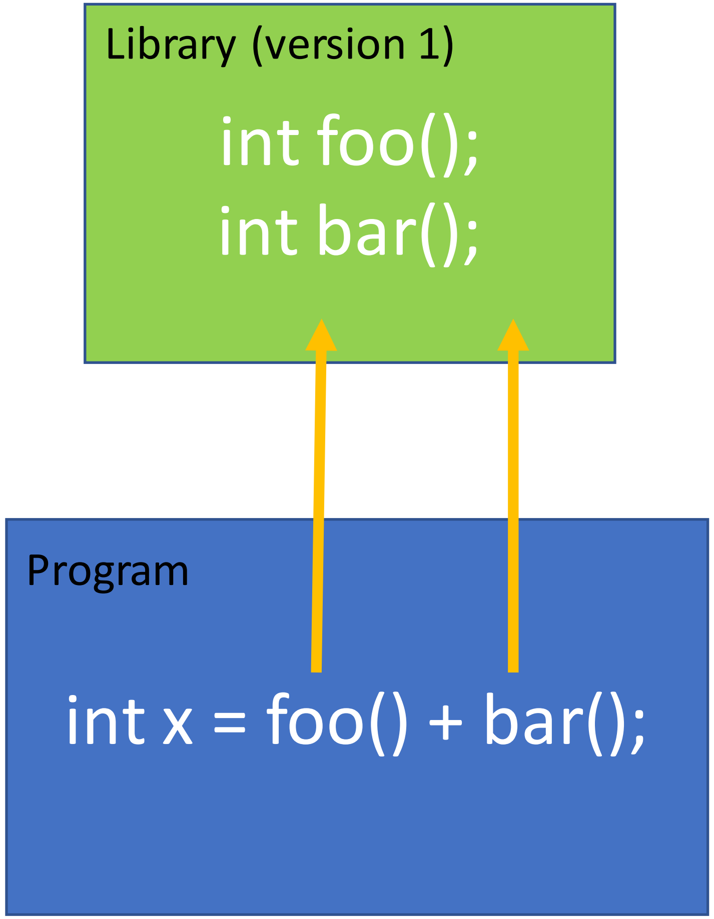
DLL disadvantages

- Performance overhead
  - Program startup (search and load libraries, search for symbols)
  - Calling library functions
- Fragile infrastructure (DLL hell)

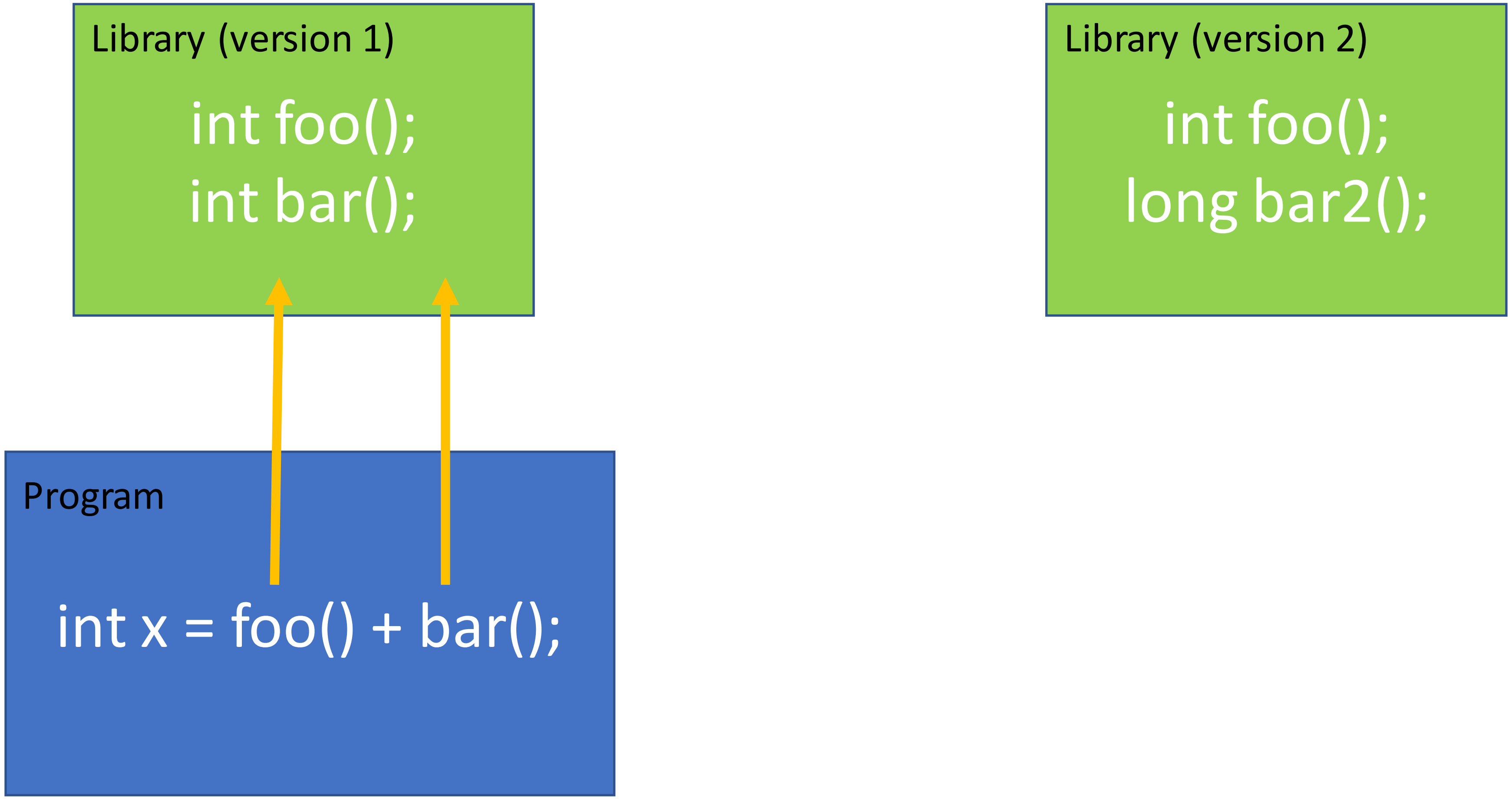
DLL Hell: example



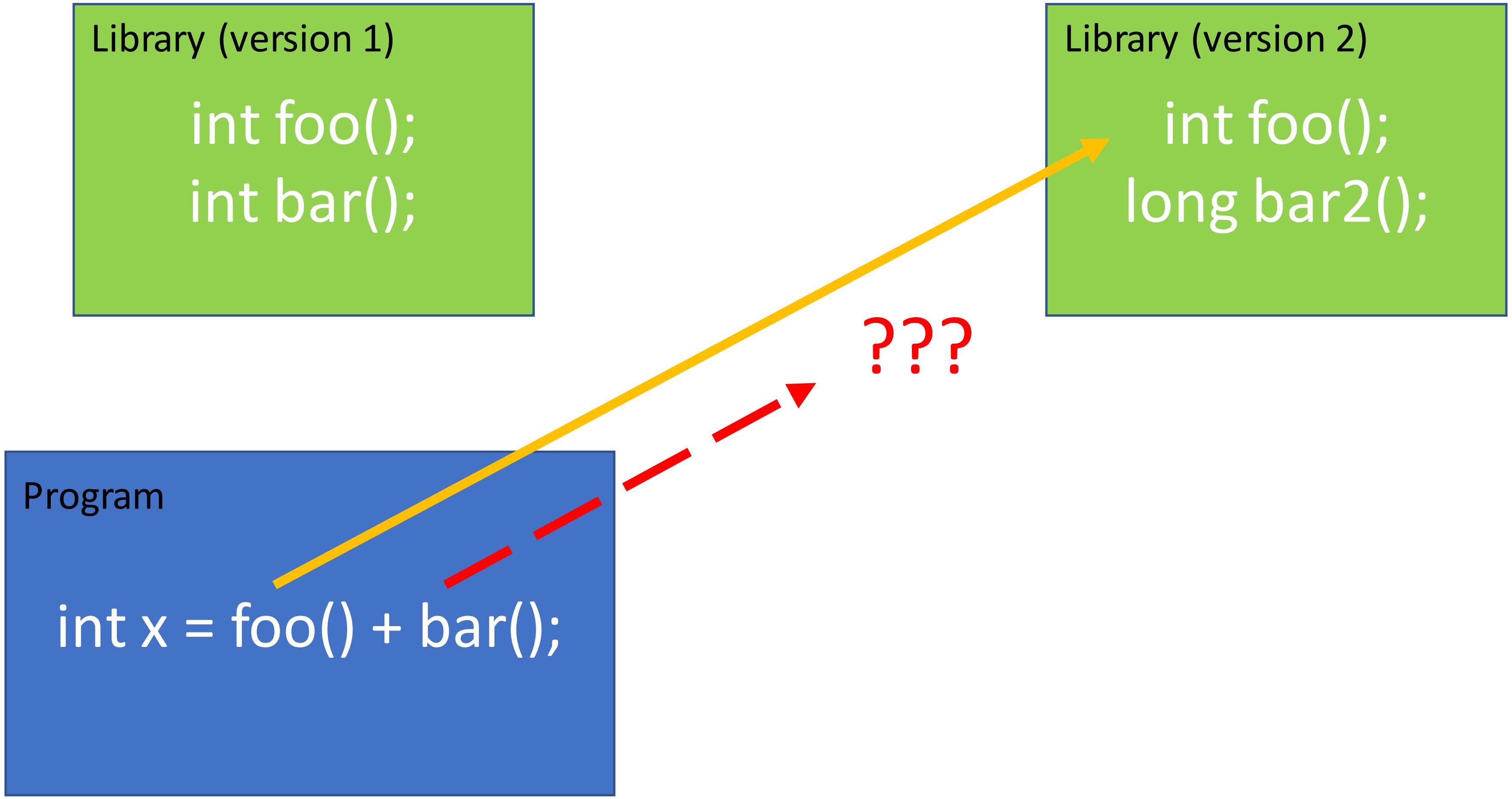
DLL Hell: example



DLL Hell: example



DLL Hell: example



DLL Hell

- It is very easy to introduce *incompatible changes* when developing a library
    - Remove function or change its signature
- Programs which used old version will not be able to work eith new one
    - Will crash at startup or later

DLL Hell: solution

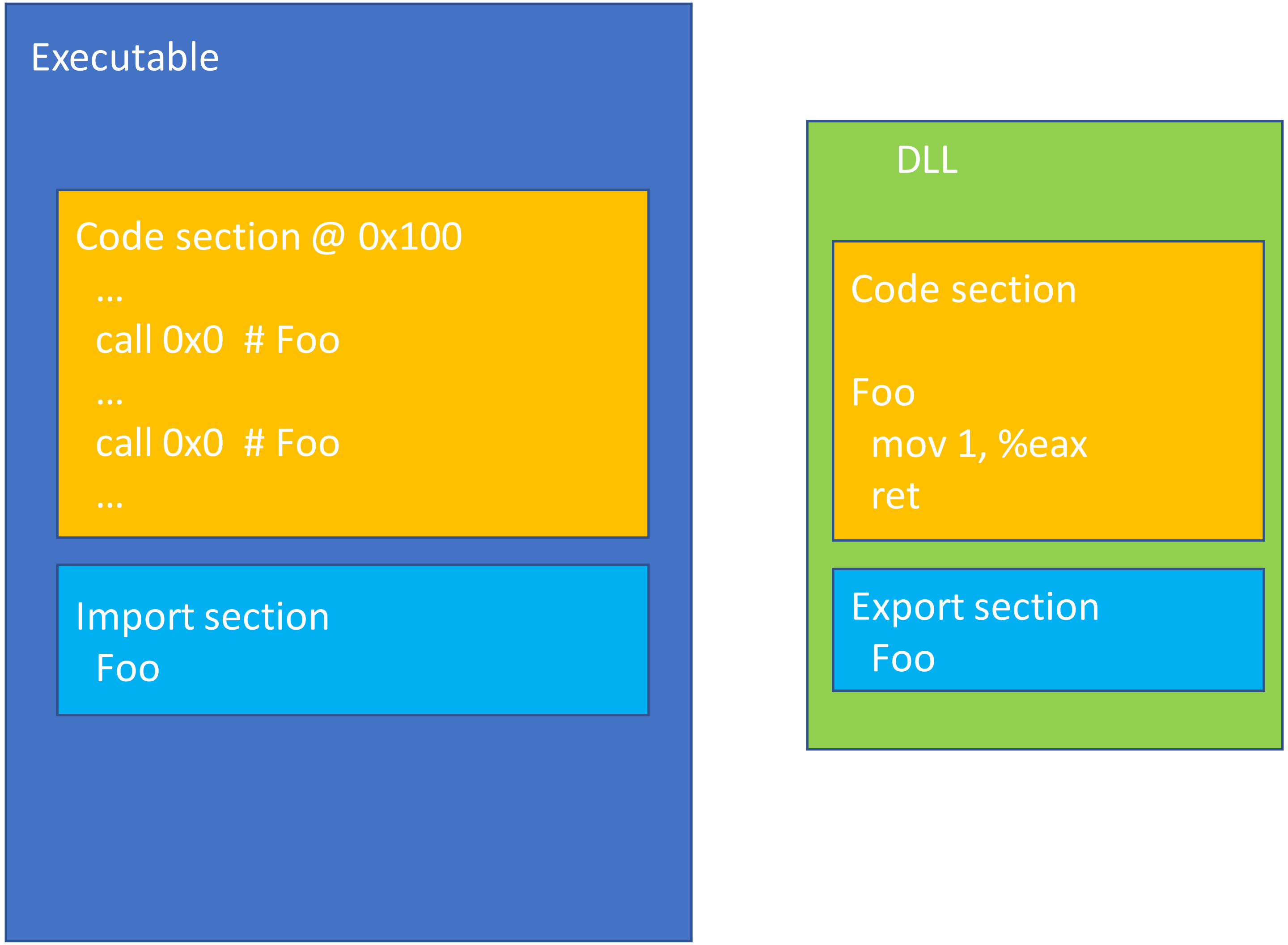
- Library developers should avoid incompatible changes
  - Incompatibility checking can and should be automated (libabigail, ABI Compliance Checker, etc.)
- If such changes are inevitable developer needs to update library version info
  - Embedded in library file
  - SONAME on Linux, DLL manifests on Windows
- This will alow OS to determine which library version is needed for particular program
- Details are OS-specific

DLL working principles

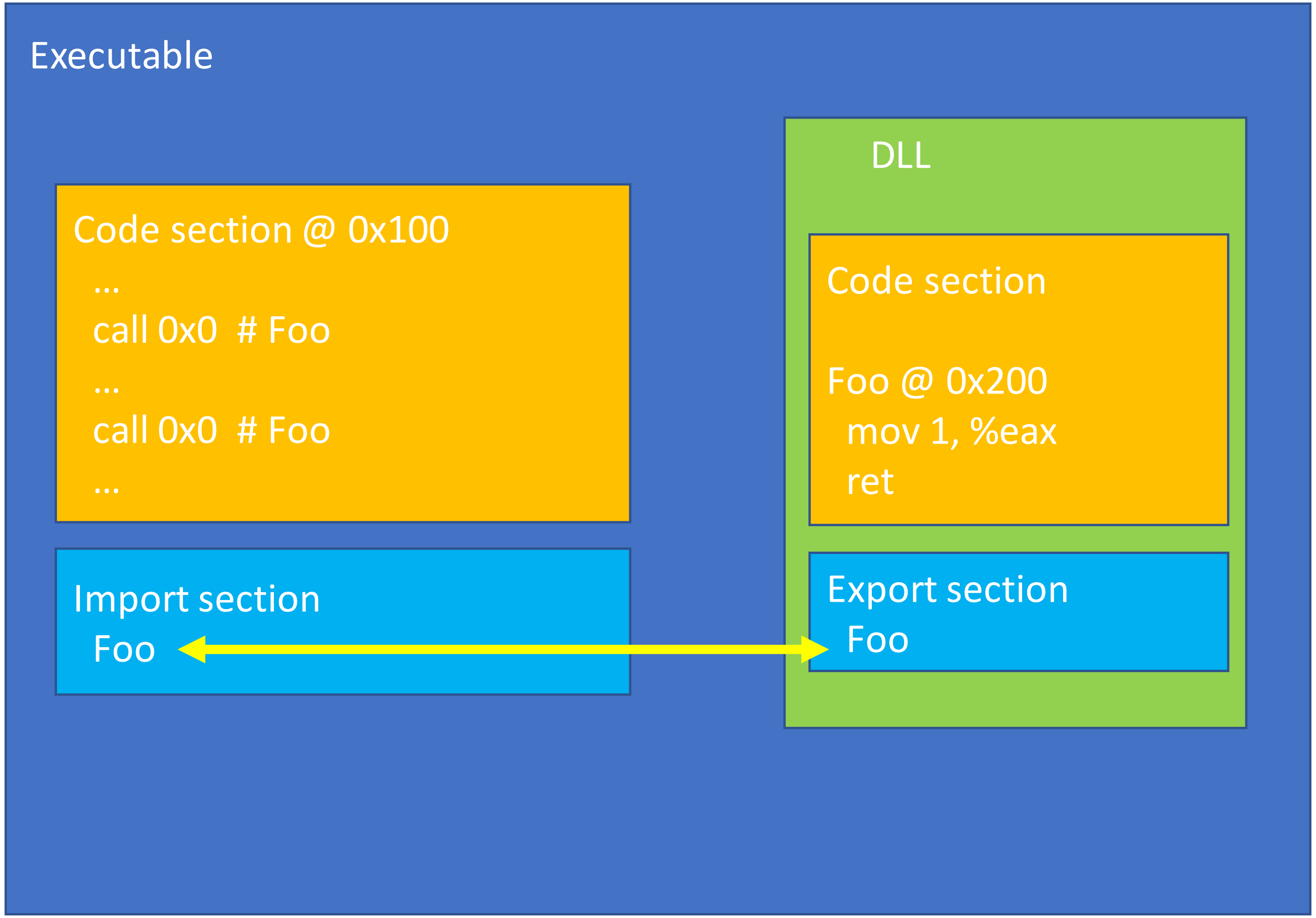
Main principles

- DLLs and executables share the same format
  - PE on Windows, ELF on Linux
- Library keeps its exported symbols in a special table
  - .edata on Windows, .dynsym on Linux
- Executable file keeps the list of needed libraries and imported symbols in another table
  - .idata on Windows, .dynsym/.dynamic on Linux

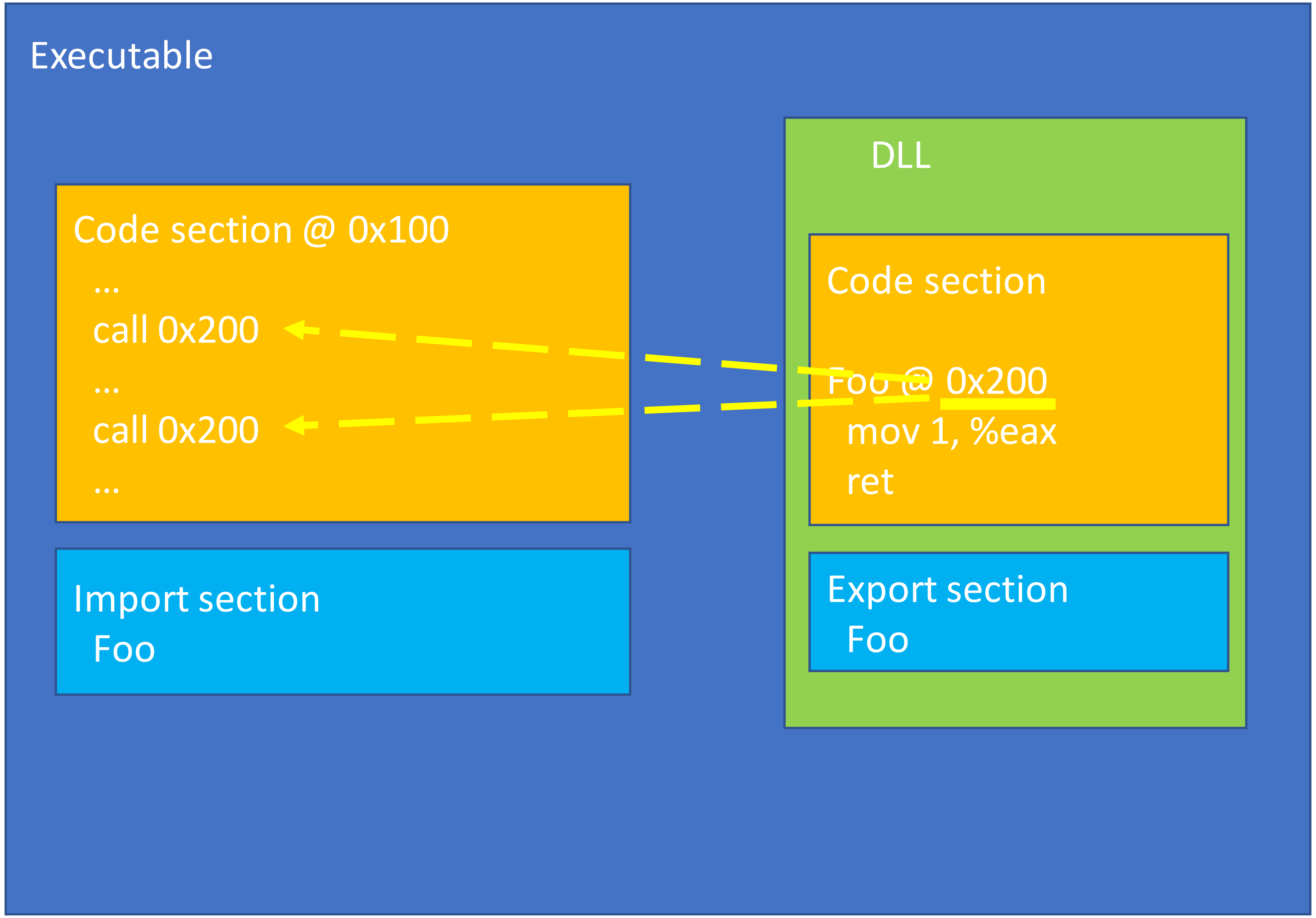
Main principles



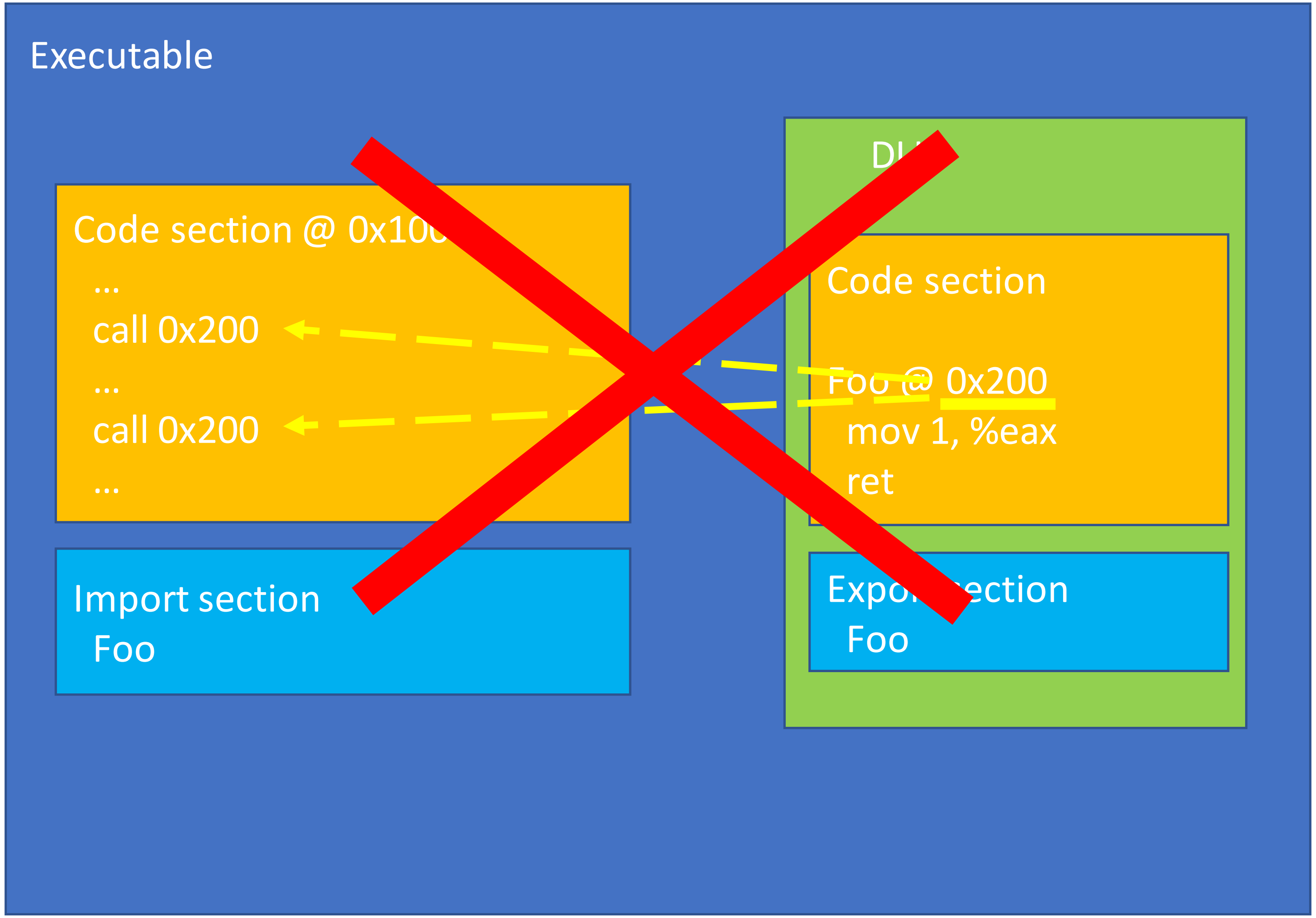
Main principles



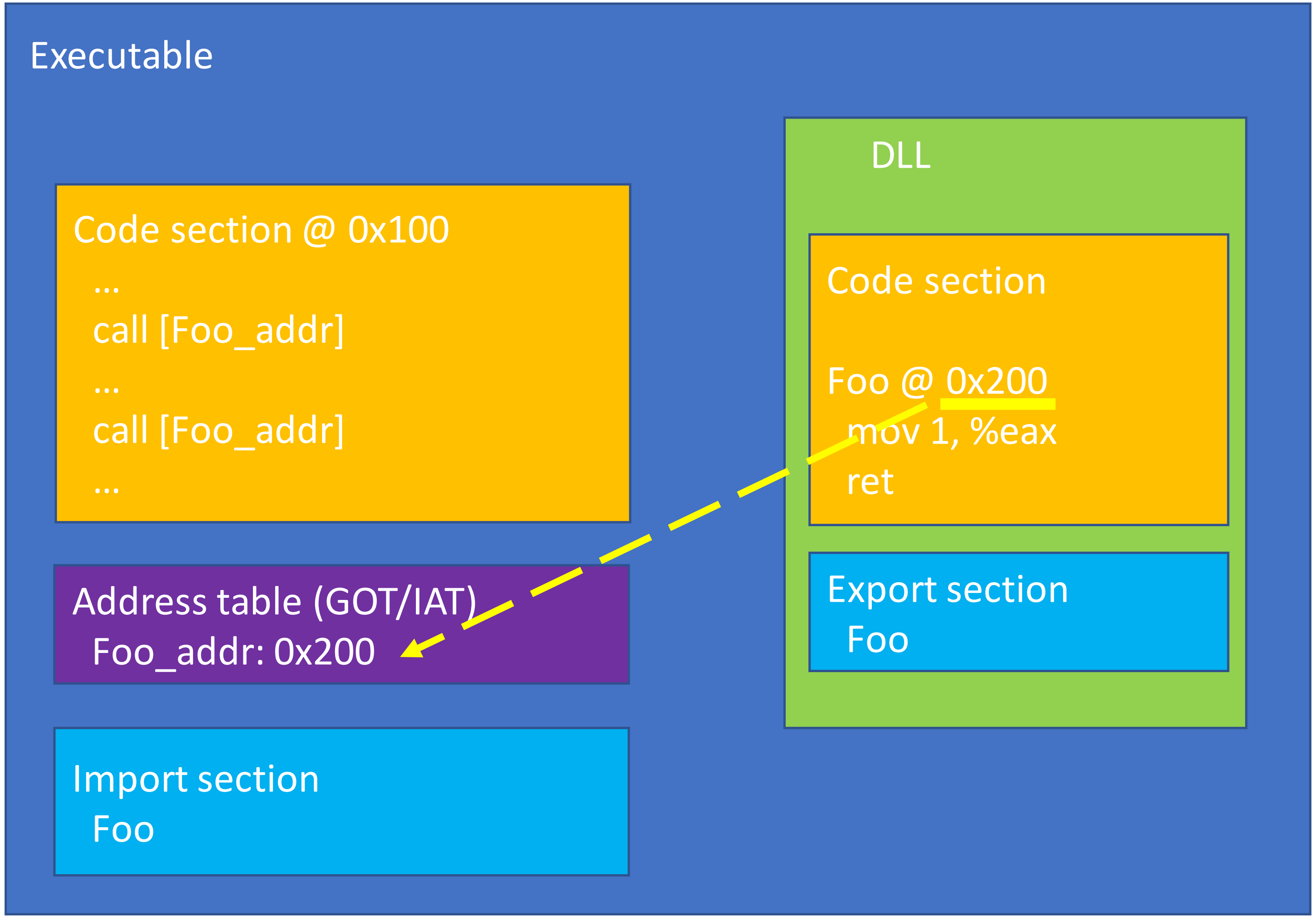
Main principles



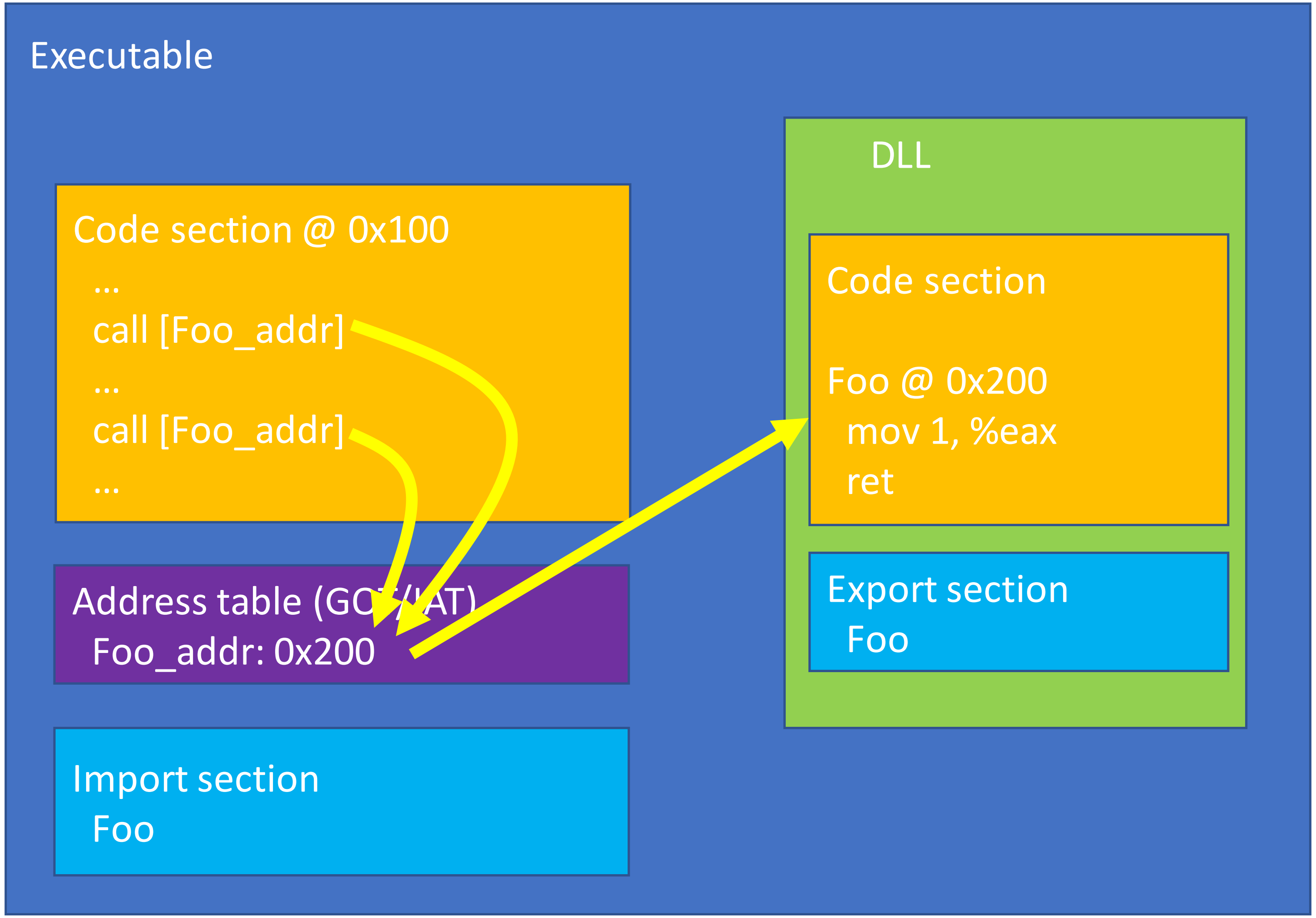
Main principles



Main principles



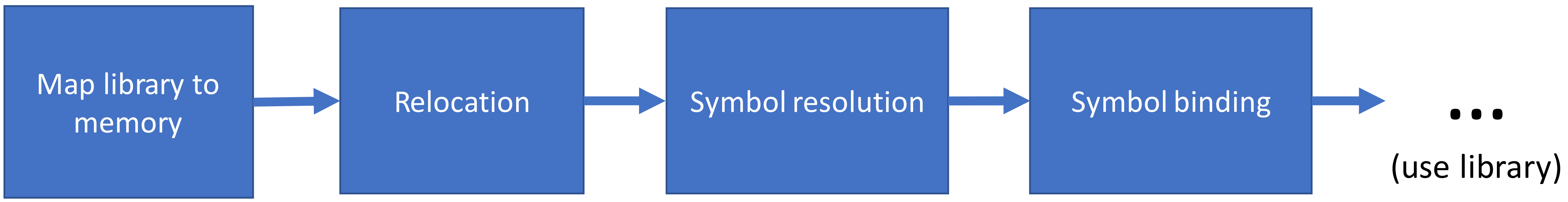
Main principles



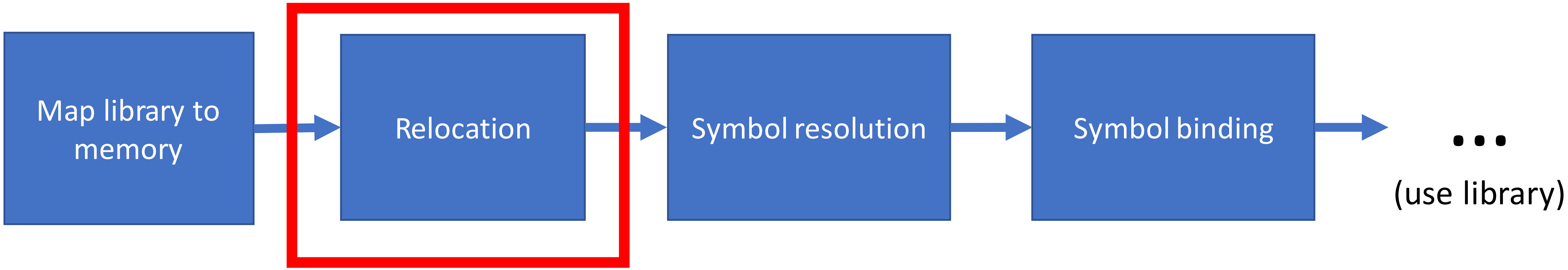
Dynamic loader

- Libraries are imported into running program by dynamic loader
  - /lib64/ld-linux-x86-64.so.2 on Linux
  - Image loader (Ldr) on Windows
- On program startup kernel maps loader into process memory and transfers control to it
- The loader
  - Maps needed libraries in process address space
  - Resolves and binds exported and imported symbols
  - Transfers control to main program

Loading DLL


Map library to memory
Relocation
Symbol resolution
Symbol binding
...
(use library)

Loading DLL


Map library to memory
Relocation
Symbol resolution
Symbol binding
…
(use library)

Relocation: example

```
$ cat lib.c
int x = 0x12;
int *p = &x;

$ gcc -shared -fPIC lib.c
```

Relocation: example

```
$ readelf --dyn-syms a.out
     ...
     5: 0000000000004028     8 OBJECT  GLOBAL DEFAULT   17 p
     6: 0000000000004020     4 OBJECT  GLOBAL DEFAULT   17 x
```

Relocation: example

```
$ readelf --dyn-syms a.out
     ...
     5: 0000000000004028      8 OBJECT  GLOBAL DEFAULT   17 p
     6: 0000000000004020      4 OBJECT  GLOBAL DEFAULT   17 x

$ objdump -s -j .data a.out
 4018 18400000 00000000 12000000 00000000  .@..............
 4028 00000000 00000000                    ........
```

P

X

Relocation

- Function and global variables addresses can only be determined at runtime
  - When library load address is known
- DLL contains special table with addresses of pointers that need to be updated after load
  - .rela.dyn on Linux, .reloc on Windows
- This patching is called *relocation*

Relocation: example

```
$ readelf -r a.out

Relocation section '.rela.dyn' at offset 0x358 contains 8 entries:
...
000000004028  000600000001 R_X86_64_64       0000000000004020 x + 0
```

P

Relocation: position-independent code

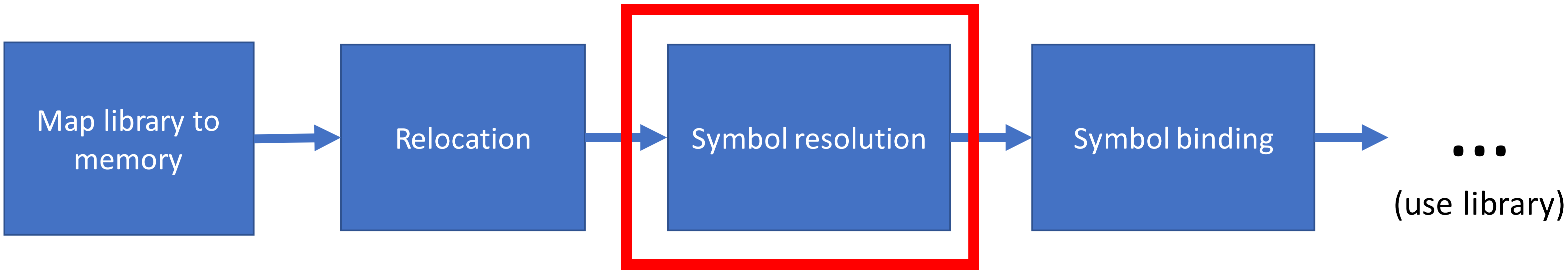
- All libraries are linked in position-independent (RIP/PC-relative) mode
  - Binary code does not use explicit addresses of function or global variables
  - Addresses are specified as offsets from to current instruction's address:

```
mov global_var, %rdi   ⟶   mov global_var(%rip), %rdi
```

- Such code does not need to be relocated at load time
  - Faster library loads
  - Code segment is constant so can be shared by multiple processes
- Data still needs to be relocated (e.g. vtables)
  - But such relocations are few

```
int data;
int *ptr = &data;   // Relocation needed
```

Loading DLL



Map library to memory
Relocation
Symbol resolution
Symbol binding
...
(use library)

Symbol resolution

- Matching exported and imported symbols
- To speed up search symbol information is stored in special hash tables
- Windows and Linux use different approaches:
  - Windows: each imported symbol is bound to particular library at link time (and will only be searched in that library)
  - Linux: imported symbols are searched sequentially in all loaded libraries
    - This enable runtime symbol interposition

Runtime interposition

- We can force loader to find imported symbols in different library than the one it was supposed to come from
- Usually interposition is enabled via LD_PRELOAD environment varible:

```
$ cat prog.c
int main(int argc) { printf("%d\n", argc); }
$ ./prog a b c
4
$ cat lib.c
int printf(char *fmt, ...) { puts("Hello from interceptor\n"); }
$ LD_PRELOAD=./lib.so ./prog a b c
Hello from interceptor
```

- Often used by debug tools like Electric Fence or AddressSanitizer to intercept memory operations (malloc, etc.)

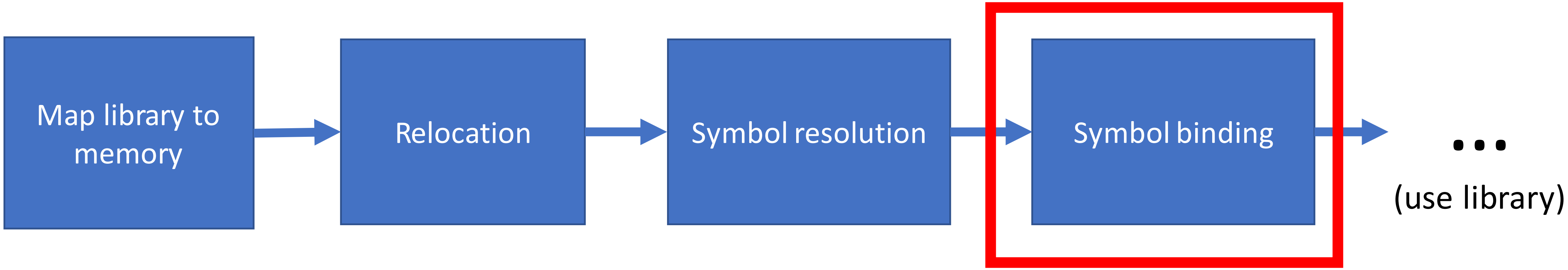
Interposition may hurt optimizations

- Compiler has to limit optimizations due to potential interposition
- E.g. compiler fails to inline due to potential interposition of foo:

```
$ cat mylib.c
void foo() {}
void bar() { foo(); }

$ gcc mylib.c -O3 -fPIC -S -o -
...
bar:
  jmp     foo@PLT
```

Loading DLL


Map library to memory
Relocation
Symbol resolution
Symbol binding
...
(use library)

Symbol binding

- Binding function calls in program with addresses of imported functions that were identified at symbol resolution stage
- Addresses of imported functions are stored in special dispatch table
  - Import Address Table on Windows, Global Offset Table on Linux
  - Initialized by loader at program startup
- Call of imported function is done by loading its address from the table:

```
# Windows
call qword ptr [__imp_foo]

# Linux
call *foo@GOTPCREL(%rip)
```

- Library calls are indirect (like virtual functions)

Lazy binding on Linux

- Loading function address from dispatch table is done by a special stub function (PLT stub)
- PLT stubs are generated by linker
- Delays symbol resolution and binding until first use:

```
    .section .text
    ...
    call foo
    ...
    .section .plt
  foo:   # PLT stub pseudocode
    if (first call)
      GOT[foo] = resolve address of foo
    call GOT[foo]
```

Speeding up dynamic libraries

DLL overheads

- Library load
  - Relocation
  - Symbol resolution and binding
- Library use
  - Indirect calls

DLL overheads

- **Library load**
  - Relocation
  - Symbol resolution and binding
- Library use
  - Indirect calls

DLL speedup: disabling unused libraries

- Often large programs may accidentally link against unused libraries
- Such libraries will slow program even down even if none of their functions are called
- -Wl,--as-needed flag allows linker to identify and ignore such libraries
- Enabled by default in some distros (Ubuntu but not Fedora/RHEL)

DLL speedup: delayed library loading (lazy loading)

- Library may be used in only some rare scenarios
- Instead of loading it at startup we could load it on first use (lazy loading)
- Some platforms support this out-of-the-box:
  - Windows: /DELAYLOAD flag
  - macOS: -Wl,-z,-lazy-l flag (no longer supported)
- No standard solution for Linux but can use Implib.so
  - https://github.com/yugr/Implib.so

Implib.so

- Given a DLL, generates small static library with stub functions (trampolines)
- Instead of DLL we link our program against that library
- At runtime executing a stub function will cause library to be loaded and control passed to it:

```
int foo(type1 arg1, type2 arg2, ...) {  # Stub
  static void *foo_real = NULL;
  if (!foo_real) {
    void *handle = dlopen(...);
    foo_real = dlsym(handle, "foo");
  }
  return foo_real(arg1, arg2, ...);
}
```

Implib.so

- Implements delayed loading for POSIX systems
- Uses runtime loading API (dlopen, dlsym)
- Supports many different targets
  - x86, ARM, AArch64, RISC-V, e2k, etc.
  - Linux (+ part. BSD)

DLL overheads

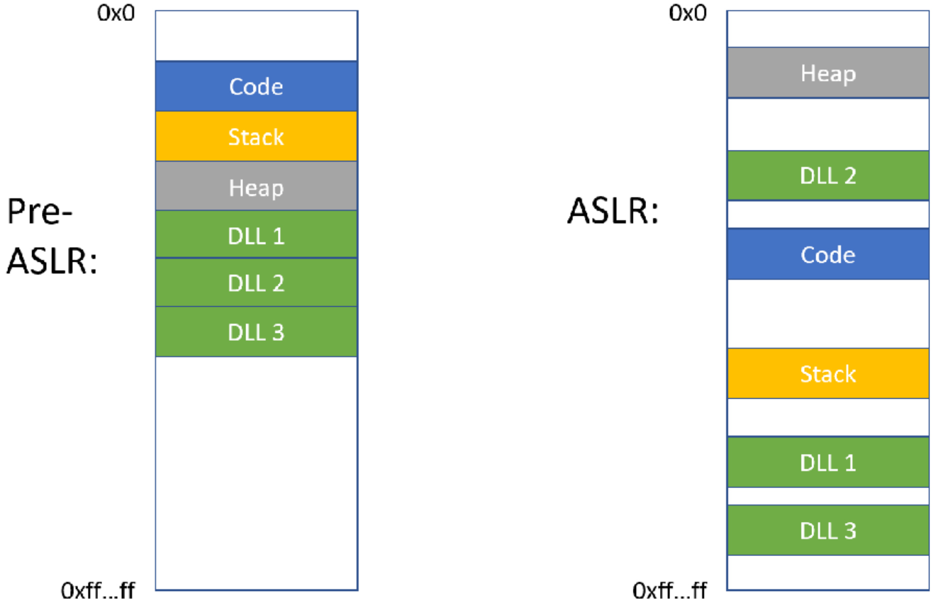
- Library load
  - **Relocation**
  - Symbol resolution and binding
- Library use
  - Indirect calls

DLL speedup: link-time relocation

- Relocation may be avoided if library is linked at address which is guaranteed to be free at load time
- To achieve this we could
  - Scan all installed programs and libraries
  - Statically partition address space between all libraries
  - Link each library at its dedicated address
- Then dynamic loader will be able to avoid library relocation
- Solutions:
  - Windows: preferred load address (/BASE linker parameter)
  - Linux: Prelink

DLL speedup: link-time relocation

- Optimization is no longer relevant due to modern security guidelines
- ASLR requires DLL to be loaded at random locations (to complicate hacker's work)



Relocation: Windows optimization

- Legacy Win32-libraries do not use position-independent code
  - X86 lacks position-independent instructions
- A lot of instructions need to be relocated at load time
- To speed things up same library is loaded at the same address in all running processes
  - Relocation is only needed on first load
  - Works in modern Windows versions

DLL overheads

- Library load
  - Relocation
  - **Symbol resolution and binding**
- Library use
  - Indirect calls

DLL speedup: prelinking

- Dispatch table inside executable file could be statically initialized with precomputed function addresses
- This will only work if library is guaranteed to always be loaded at same fixed address
  - I.e. link-time relocation was performed
- Solutions:
  - Windows: DLL binding
  - Linux: Prelink
- No longer relevant in modern Windows and Linux due to ASLR

DLL speedup: optimizing symbol tables

- During symbol resolution symbols are looked up in hashtables inside dynamic libraries
- On Linux linkers provide some means to control size and format of these hashtables
- The usually recommended set of options:
  - -Wl,--hash-style=both -Wl,-O1
- -Wl,--hash-style=both is turned on by default in all modern distros
- -Wl,-O1 does not improve performance in practice

DLL speedup: disable lazy binding

- Lazy binding on Linux speeds up program startup at the cost of additional function call overhead
- In addition to address load and indirect call we have a PLT stub call:
  - Extra jump
  - Increased cache/branch predictor pressure
  - Address has to be loaded from GOT on each call
- Lazy binding and related overheads may be disabled via -fno-plt compiler flag

DLL speedup: disable dynamic loading

- -fno-plt
  - Speeds up library function calls
  - Reduces pressure on I$ and BTB
  - Slows down program startup (as all addresses now need to be resolved and bound at program startup)
- Modern security guidelines suggest that all addresses are resolved at startup anyway
  - Allows Full Relro (-Wl,-z,relro) protection to avoid unintended GOT modifications during program run
  - Full Relro is enabled by default in RHEL/Fedora and Ubuntu

DLL speedup: disable lazy binding

- Examples:
  - Using -fno-plt in Clang improves performance by up to 10%

DLL overheads

- Library load
  - Relocation
  - Symbol resolution and binding
- Library use
  - **Indirect calls**

Problem with exported symbols

- By default all library functions are exported on Linux
  - For compatibility with static libraries
- Due to potential interposition all function calls inside the library must go through GOT
- Overheads:
  - Unnecessary indirect calls
  - Disabled compiler optimizations (inlining, cloning, etc.)

DLL speedup: disabling function interposition

- Special compiler flags allow compiler to ignore interposition
- -Bsymbolic/-Bsymbolic-functions – replaces indirect calls of library functions inside the library with direct calls
  - Turned on by default in some distributions (Ubuntu but not Debian)
- -fno-semantic-interposition – tells compiler to ignore possibility of interposition
  - Turned on by default in Clang but not GCC
  - Enabled in GCC under -Ofast
- Need both flags for optimal performance

DLL speedup: disabling function interposition

- Examples:
  - Using -Bsymbolic-functions speeds up Clang by up to 10%
  - Using -fno-semantic-interposition when building Python gives up to 30% performance improvement
    - https://fedoraproject.org/wiki/Changes/PythonNoSemanticInterpositionSpeedup

DLL speedup: reducing library interface

- Simple way to improve performance
- Does not require non-standard build flags
- Explicit control over which symbols are exported:

```
$ cat mylib.c
void internal() {}

__attribute__((visibility("default")))
void public() { internal(); }

$ gcc mylib.c -fvisibility=hidden -fPIC -shared
```

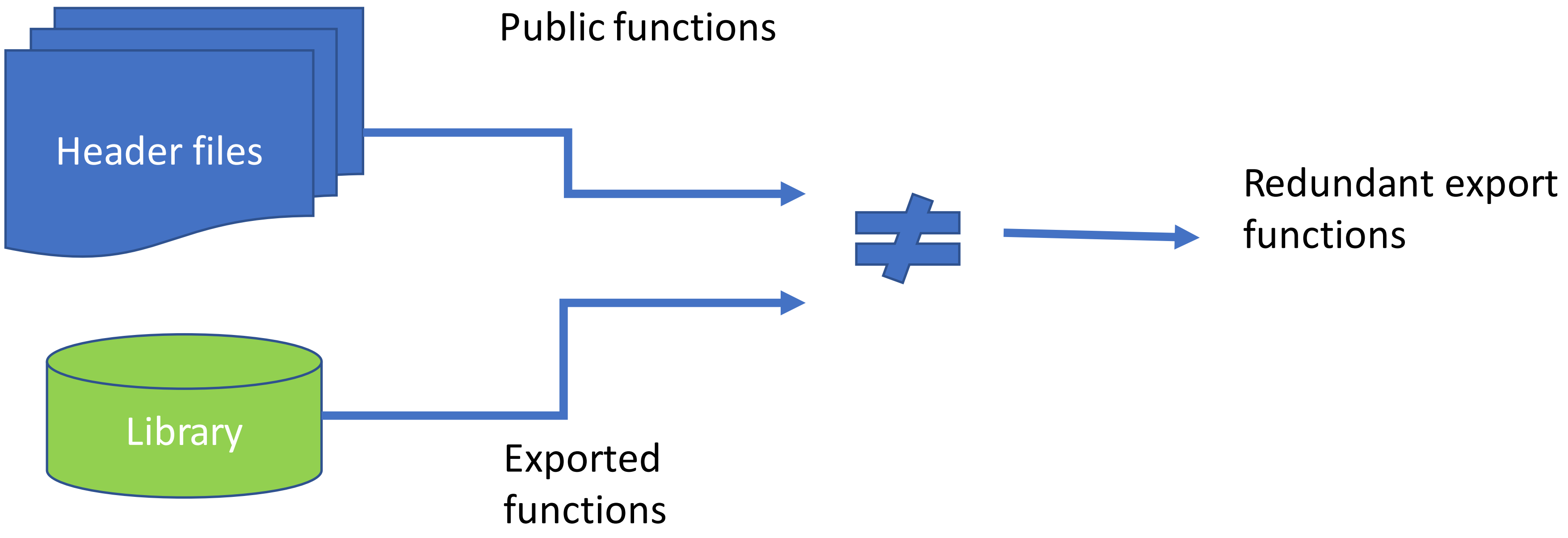
- Which functions to export?
  - Usually functions from public header files
  - Such functions are a tiny fraction of all library functions

DLL speedup: reducing library interface

- For a large code base (e.g. Linux distro) it may be hard to identify libraries with redundant exports
- Search of such libraries may be automated with ShlibVisibilityChecker tool
  - https://github.com/yugr/ShlibVisibilityChecker

ShlibVisibilityChecker

- Analyzes functions in public library header files via libclang
- Compares them against functions actually exported by the library
- Reports redundant exports which need to be hidden



ShlibVisibilityChecker example

```
$ read_header_api --only-args /usr/include/x86_64-linux-gnu/gmp.h > api.txt

$ read_binary_api --permissive /usr/lib/x86_64-linux-gnu/libgmp.so.10.4.1 > abi.txt

$ diff api.txt abi.txt | wc -l
323

$ diff api.txt abi.txt
0a1,10
> __gmp_0
> __gmp_allocate_func
> __gmp_asprintf_final
> __gmp_asprintf_funs
…
```

Conclusions

Conclusions

- Dynamic libraries have some advantages over static ones

Conclusions

- Dynamic libraries have some advantages over static ones
- But add overheads at program startup and program runtime

Conclusions

- Dynamic libraries have some advantages over static ones
- But add overheads at program startup and program runtime
- Modern toolchains provide means to reduce overheads
  - Especially on Linux

Additional reading

- Linkers, Loaders and Shared Libraries in Windows, Linux, and C++ (Ofek Shilon, CppCon 2023)
  - https://www.youtube.com/watch?v=_enXuIxuNV4
  - General overview of DLLs on different platforms
- How to Write Shared Libraries (by Ulrich Drepper)
  - https://www.akkadia.org/drepper/dsohowto.pdf
  - All you need to know about DLLs on Linux
- Everything You Ever Wanted to Know about DLLs (by James McNellis, CppCon 2017)
  - https://www.youtube.com/watch?v=JPQWQfDhICA
  - All you need to know about DLLs on Windows
- MaskRay Blog
  - https://maskray.me/blog
  - Linux system programming blog (GOT, PLT, etc.)

Thanks!

Check RAM savings

- Build scanner
  - gcc -Wall -Wextra scripts/ram-savings.c
- Run under sudo:
  - sudo ./a.out

Check disk savings

- Run
  - scripts/disk-savings.pl
- Script reports upper bound – real savings would be lower
  - With static libs not all functions will be imported by the applications
  - So only parts of the libraries will be included in executables

Check -Wl,-O1

- Build two versions of LLVM:
  - -DBUILD_SHARED_LIBS=ON
  - -DBUILD_SHARED_LIBS=ON -DCMAKE_SHARED_LINKER_FLAGS='-Wl,-O1’
- Compare performance:
  - ./benchmark.pl 10 path/to/clang -h

Check -fno-plt

- Build two versions of LLVM:
  - -DBUILD_SHARED_LIBS=ON
  - -DBUILD_SHARED_LIBS=ON -DCMAKE_CXX_FLAGS='-fno-plt'
- Compare performance:
  - ./benchmark.pl 10 path/to/clang -S -O2 ~/InstCombining.ii

Check -Bsymbolic-functions

- Build two versions of LLVM:
  - -DBUILD_SHARED_LIBS=ON
  - -DBUILD_SHARED_LIBS=ON -DCMAKE_SHARED_LINKER_FLAGS='-Wl,-Bsymbolic-functions’
- Compare performance:
  - ./benchmark.pl 10 path/to/clang -S -O2 ~/InstCombining.ii

Address-space Layout Randomization