Dynamic libraries explained

as seen by a low-level programmer

I.Zhirkov

2017

Exemplary environment

- Intel 64 aka AMD64 aka x86_64.
- GNU/Linux
- Object file format: ELF files.
- Languages: C, Assembly (NASM)

Sources

- ELF format specification
- System V Application Binary Interface
- U.Drepper – “How to write shared libraries”
- `http://amir.rachum.com/blog/2016/09/17/shared-libraries/`
- I.Zhirkov – “Low-level programming: C, Assembly and Program Execution”

Contents

Prerequisites

Assembly (NASM)

- Registers: `rax`, `rbx`, `rcx`, ...; `rip`
- Sections: code (.text) separated from data (.data)
- `label: instruction ; comment`
- Addressing modes:
- `db 1` | `dw 2` | `dd 4` | `dq 8`

```
var: dq 42, 43, 44
mov rax, 42             ; imm
mov rax, var            ; address
mov rax, [var]          ; value (42)
mov rax, [var + 8]      ; value (43)
mov rax, [var + 8*rcx]  ; value (43)
```

Preface

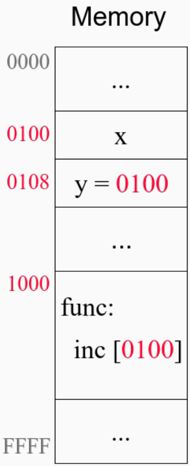
Relocation

- Compiling is not trivial.
- We have Random Access Memory, linear addresses.
- Challenge: carefully placing code and data in memory.

Example

```c
int x;
int* y = &x;

void f() {
    x = x + 1;
}
```



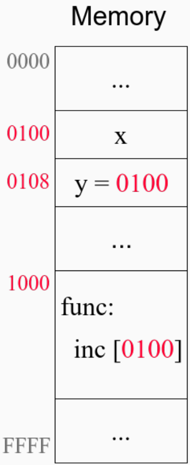
red – depends on positioning

Example

```
int x;
int* y = &x;

void f() {
    x = x + 1;
}
```

- Where to place `x` and `y`?
- Code and data require knowing addresses.
- *Once an address is picked, it is difficult to change.*



Solution: linking stage

- Last stage of compilation: linking.
- Defer placement until linking.
  - Instructions generated, we know all functions and global variables.
- **Symbol** – program entities which are useful for linking.
  - Anything with an absolute address
    - Global variables.
    - Functions.
  - Utility symbols.

Symbols

For each symbol we know its:

- Name.
- Address (if assigned).
- Locations where referenced.

Assigning addresses to symbols is called **relocation**.

```c
int x;            // symbol 'x'

void func() {     // symbol 'f'
    x = x + 1;    // symbol 'x' referenced
}
```

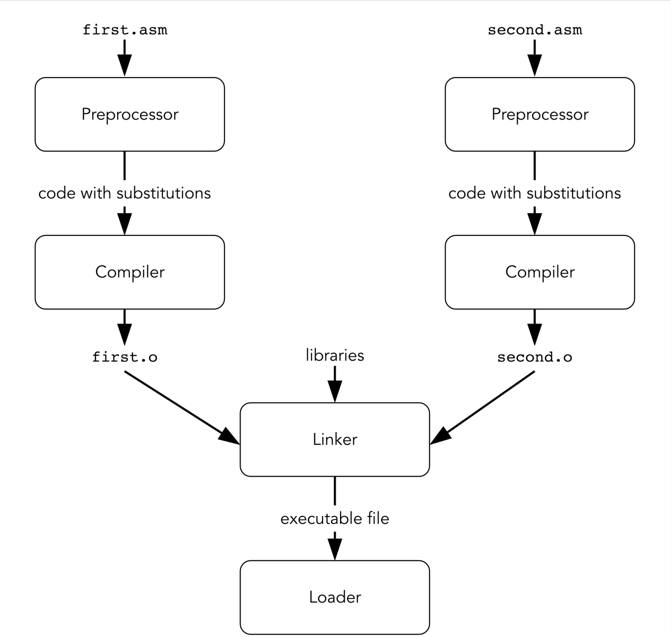
Linker

Linker is a program that assigns addresses to symbols and finalizes compilation.

Allows for separate compilation, which we need:

- Programs too big for one file (split into modules).
- Using already compiled code (libraries).
- Fast debugging (each change invokes recompilation).
  - Some programs take *hours* to compile from scratch.

Compilation pipeline



Modules and objects

Compiler works with atomic code entities called **modules**.

In C and assembly, a module corresponds to a `.c` or `.asm` file.

Modules are transformed into **object files**.

Object files are structured and contain translated instructions.

ELF object files

ELF – Executable and Linkable Format, typical for *nix systems.

Can be:

1. **Relocatable**

   `.o`-files, produced by compiler, not linked.

   Same as **static libraries**.

2. **Executable**

   program after linking, ready-to-run.

3. **Shared**

   `.so`, dynamic libraries, to be linked in runtime.

Static linker transforms 1 into 2 or 3.

Dynamic linker prepares 3 for execution.

Static linking

Toolset

Tools to examine object files:

- `readelf` – ELF meta-information
- `objdump` – meta-information of any format, disassembler
- `nm` – only symbols.

What we use:

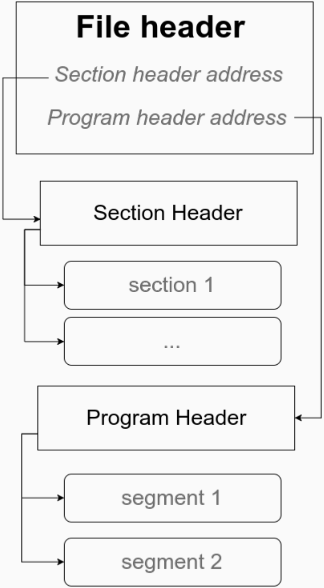
- `objdump` usually, less specific
- `readelf` for verbose ELF structure

ELF file structure

Three headers:

- File header
  - General info.
  - Links to Program and Section headers.
- Section header
  - Information about **sections**.
  - Each section stores code or meta-information.
  - Needed for linking.
- Program header
  - Instructions on how to create process image.
  - Information about **segments**.
  - Segment is a virtual memory region; contains some sections.
  - Needed for execution.

Memory region – consecutive memory pages with same permissions.



Typical sections:

- **.data**
- **.text** – compiled instructions.
- **.rodata** – read only.
- **.bss** – zero-initialized data (only size is stored).
- **.line** – line numbers in source code.
- **.symtab** – symbol table.
- ...

Exemplary program

```
section .data             ; global variables:
x: dq 148842              ; int  x = 148842
y: dq x                   ; int* y = &x

extern somewhere          ; an external symbol
global _start             ; visible to other modules

section .text             ; code:  {
  _start:
    mov rax, x            ; rax := &x
    mov rdx, y            ; rdx := &y

  jmp _start              ; } while (true);
```

ELF File Header

```
> nasm -f elf64 -o symb.o symb.asm  # compile
> ld -o symb symb.o                 # link
> readelf -h  symb                # view file header
```

Class: ELF64

Type: EXEC (Executable file)

Entry point address: 0x4000c0

Start of program headers: 56 (bytes into file)

Start of section headers: 584 (bytes into file)

Number of program headers: 2

Number of section headers: 6

Sections – before linking

```
> objdump -h symb.o
```

No	Name	Size	Address	Offset	Align
1	.data	0x10	0	0x240	4
		CONTENTS, ALLOC, LOAD, RELOC, DATA			
2	.text	0x16	0	0x250	16
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE				

Sections – before linking

```
> objdump -h symb.o
```

No	Name	Size	Address	Offset	Align
1	.data	0x10	0	0x240	4
		CONTENTS, ALLOC, LOAD, RELOC, DATA			
2	.text	0x16	0	0x250	16
		CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE			

- Stubs for addresses.
- Offset from file start.
- `objdump` omits excess information.

Sections – before linking

`readelf` is more verbose.

```
> readelf -S symb.o
```

Name	Type	Address	Offset	Size	EntSize	Flags	Link	Info	Align
.data	PROGBITS	0	240	10	0	WA	0	0	4
.text	PROGBITS	0	250	16	0	AX	0	0	16
.shstrtab	STRTAB	0	270	3d	0	0	0	0	1
.symtab	SYMTAB	0	2b	c0	18	0	5	6	4
.strtab	STRTAB	0	370	2d	0	0	0	0	1
.rela.data	RELA	0	3a0	18	18	0	4	1	4
.rela.text	RELA	0	3c0	30	18	0	4	2	4

`.symtab` stores the symbol table.

`.rela.<section-name>` store relocations.

Symbol table before linking

```
> objdump -tf symb.o
start address 0x0
```

Address	Type	Section	Name
0x0	`l df`	*ABS*	symb.asm
0x0	`l d`	.data	.data
0x0	`l d`	.text	.text
0x0	`l`	.data	x
0x8	`l`	.data	y
0x0		*UND*	somewhere
0x0	g	.text	_start

`l` – local

g – global (visible to other object files)

`d` – debug symbol

`f` – file name

Symbol table before linking

- Addresses are relative to section starts
- Utility symbols are marked as debug
- External symbols are in $*\text{UND}*$ section
- $*\text{ABS}*$ as “unrelated to sections”.

Stubs are also in code

Source

```
_start:
  mov rax, x
  mov rdx, y
jmp _start
```

Disassembly

```
> objdump -d -mintel-mnemonic symb.o

0000000000000000 <_start>:
   0:   48 b8 00 00 00 00 00    mov rax, 0x0
   7:   00 00 00
   a:   48 ba 00 00 00 00 00    mov rdx, 0x0
  11:   00 00 00
```

Relocations

- We need to keep track of the stubs.
- Sections of interest: **.data**, **.rodata**, **.text**.

```
> objdump -r symb.o
```

Relocations in .data:

Offset	Type	Value
0x8	R_X86_64_64	.data

```
x: dq 148842
y: dq x
```

Relocations in .text:

Offset	Type	Value
0x2	R_X86_64_64	.data
0xc	R_X86_64_64	.data+0x8

```
mov rax, x
mov rdx, y
```

Sections in linked file

```
> ld -o symb symb.o
```

No	Name	Size	Address	Offset	Align
1	.text	0x16	0x4000b0	0xb0	16
		CONTENTS, ALLOC, LOAD, READONLY, CODE			
2	.data	0x10	0x6000c8	0xc8	4
			CONTENTS, ALLOC, LOAD, DATA		

Sections in linked file

```
> ld -o symb symb.o
```

No	Name	Size	Address	Offset	Align
1	.text	0x16	0x4000b0	0xb0	16
		CONTENTS, ALLOC, LOAD, READONLY, CODE			
2	.data	0x10	0x6000c8	0xc8	4
			CONTENTS, ALLOC, LOAD, DATA		

- Addresses are chosen.
- No more RELOC mark.

Symbol table after linkage

```
> objdump -tf symb
Flags: EXEC_P, HAS_SYMS, D_PAGED
start address 0x4000b0
```

Address	Type	Section	Name
0x4000b0	l d	.text	.text
0x6000c8	l d	.data	.data
0x000000	l df	*ABS*	symb.asm
0x6000c8	l	.data	x
0x6000d0	l	.data	y
0x000000		*UND*	somewhere
0x4000b0	g	.text	_start
0x6000d8	g	.data	__bss_start
0x6000d8	g	.data	_edata
0x6000d8	g	.data	_end

Program Headers

- Created after static linking (shared / executable object files).
- Each entry is a segment or other information for execution.

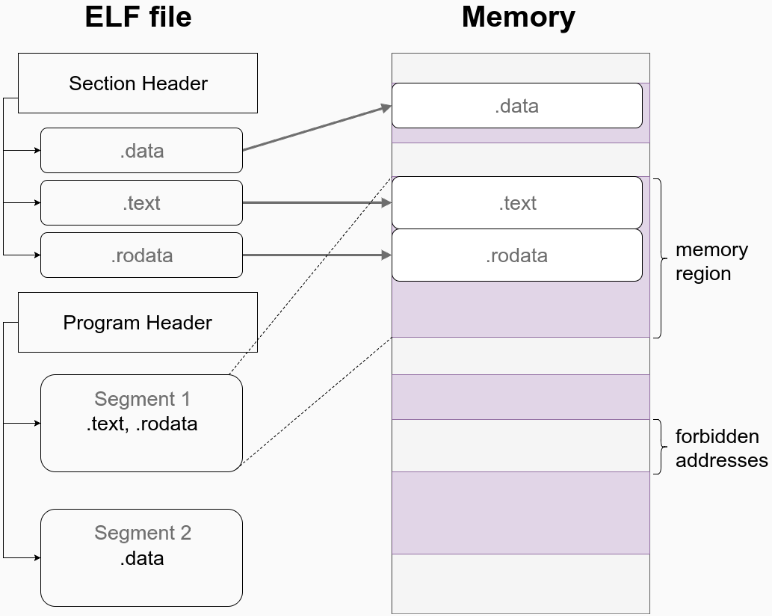
Segments

Confusion: everything is called "segment"

- Real/Protected mode segments.
- ELF segments
- "Segmentation fault"
- ...

ELF segment maps some sections to a region of memory.

Mapping sections into memory



Program Headers

```
> objdump -l symb
```

Type	Offset	VirtAddr	FileSize	MemSize	Flags	Align
LOAD	0	0x400000	0xc6	0xc6	r-x	0x200000
LOAD	0xc8	0x6000c8	0x10	0x10	rw-	0x200000

```
Section to Segment mapping:
 Segment Sections...
  00      .text
  01      .data
```

Loader uses flags to set up permissions in page tables.

With more sections

```
global _start

section .text
  _start: jmp _start
section .data
db 10

section .rodata
db 1

section .bss
resq 1024
```

With more sections

```
> objdump -l symb
```

Type	Offset	VirtAddr	FileSize	MemSize	Flags	Align
LOAD	0	0x400000	0xb8	0xb8	r-x	0x200000
LOAD	0xb8	0x6000b8	0x01	0x2008	rw-	0x200000

```
Section to Segment mapping:
 Segment Sections...
  00     .text .rodata
  01     .data .bss
```

With more sections

```
> objdump -l symb
```

Type	Offset	VirtAddr	FileSize	MemSize	Flags	Align
LOAD	0	0x400000	0xb8	0xb8	r-x	0x200000
LOAD	0xb8	0x6000b8	0x01	0x2008	rw-	0x200000

```
 Section to Segment mapping:
  Segment Sections...
   00      .text .rodata
   01      .data .bss
```

- .bss spares space in file
- .rodata has execution permissions but it seems to be fine.

Static linking: summary

Compiler

- Generates code with stubs for absolute addresses.
- Generates symbol table.
- Generates relocation tables for sections in need.

Static linker

- Relocates pieces of code and data.
- Fills in stubs found in relocation tables.
- Creates program headers.

Notes on C

- All symbols are global by default.
- `static` makes symbol local.
- Long string literals and constants are likely to be in .rodata.
- Zero initialized data in .bss (makes file smaller).

Shared libraries

What are shared libraries?

- Third type of ELF files.
- Separate file, after linking.
  - `.dll`, `.so`
- Can be updated separately.
- Exposes some of global variables and functions.
- Relocation is partially performed.
- **Reusable by other running processes**.
- Spares memory, but has additional costs when using.

Executable files use many libraries.

Relocations in a shared library

Kinds:

- Links to locations in the same library (resolved by static linker).
- Symbol dependencies (usually in the different object) – performed by dynamic linker at runtime.

Dynamic linker's job

1. Find and load dependencies.
2. Perform relocation.
3. Initialize the application and its dependencies
4. Pass the control to the application.

How to find which libraries we need?

- Search locations:
  - rpath – to be found in section .dynamic
  - `LD_LIBRARY_PATH` environment variable.
  - runpath – to be found in section .dynamic
  - List in the file /etc/ld.so.conf.
  - Standards such as `/lib`
- Depth-first-search order, dependencies and their dependencies.
- Remember, there is an order on dependencies!
- Does not load the same library twice.

How to select a symbol?

As in static linking, we search by name through the symbol tables.

Symbol can be defined in multiple objects, only one will exist in runtime.

Depending on a set of existing objects, its location may change.

Lookup scope of an object file an ordered list of a subset of the loaded objects.

Lookup scopes

Last to first priority.

- Global: the executable and all its dependencies recursively, in a *breadth-first search* order.
  Starts with the executable.
- Legacy: look in metadata if `DF_SYMBOLIC` flag is set. If yes, local definitions are preferred.
- Everything opened by **`dlopen`** call have a common additional separated scope. Not searched for normal lookups.

`LD_PRELOAD` allows to add a library to global scope right after the executable itself.

How to perform relocations?

- Relocations in runtime lead to patching addresses.
- Modified pages can not be shared, hence the advantages of shared libraries pale.

Can we write in a manner we can execute the code no matter the loading address?

Sharing library between processes

- .data and .bss can not be shared anyway (each process should have its own global variables).
- .text can be shared if consists of **position independent code (PIC)**.
- .rodata can be shared if it has no relocations (e.g. an address of a variable).

RIP-relative addressing

RIP – register holding the address of current instruction (Program Counter)

Intel 64 supports RIP-relative addressing out-of-the-box.

Can we just change all addressing to RIP-relative?

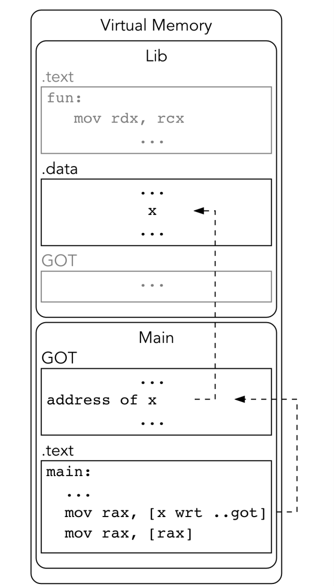
- Works for addresses of *local* variables and functions: we know the offsets between current position in code and everything from the same object file.
- **Not for exported or imported symbols**: we do not know which object will provide them.

Solution: add level of indirection using **Global Offset Table**.

Global Offset Table

- A table storing addresses of global variables and functions.
- Unique per object file.
- Dynamic linker fills it with correct addresses.

Global Offset Table



Accessing a global variable x

`rip`-relative addressing

NASM uses the `rel` keyword to achieve `rip`-relative addressing.
This does not involve GOT nor PLT.

Relocation types

All are processor-specific.

Tells how specifically should we alter an instruction operand.

`R_X86_64_64` for “replace with immediate address”, but others are more complex.

Relocation types for Intel 64

R_X86_64_NONE	No reloc
R_X86_64_64	Direct 64 bit
R_X86_64_PC32	PC relative 32 bit signed
R_X86_64_GOT32	32 bit GOT entry
R_X86_64_PLT32	32 bit PLT address
R_X86_64_COPY	Copy symbol at runtime
R_X86_64_GLOB_DAT	Create GOT entry
R_X86_64_JUMP_SLOT	Create PLT entry
R_X86_64_RELATIVE	Adjust by program base
R_X86_64_GOTPCREL	32 bit signed pc rel offset to GOT
R_X86_64_32	Direct 32 bit zero extended
R_X86_64_32S	Direct 32 bit sign extended
R_X86_64_16	Direct 16 bit zero extended
R_X86_64_PC16	16 bit sign extended pc relative
R_X86_64_8	Direct 8 bit sign extended
R_X86_64_PC8	8 bit sign extended pc relative

Procedure Linkage Table

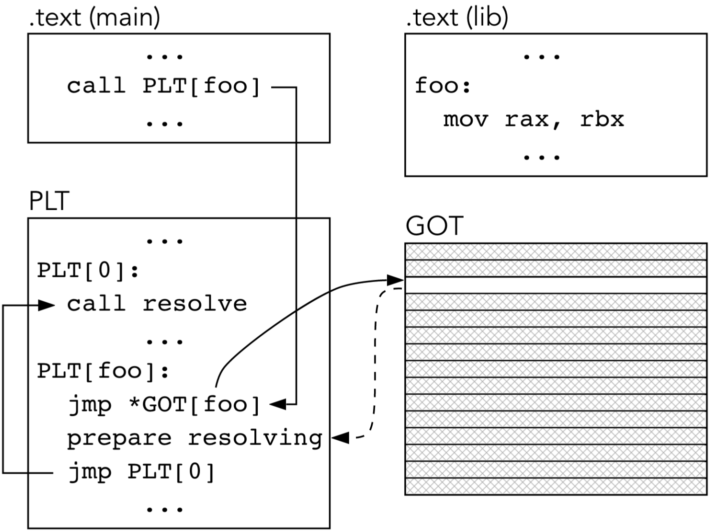
Calling globally available procedures is based on an augmented GOT.

There are many procedures exported, few used.

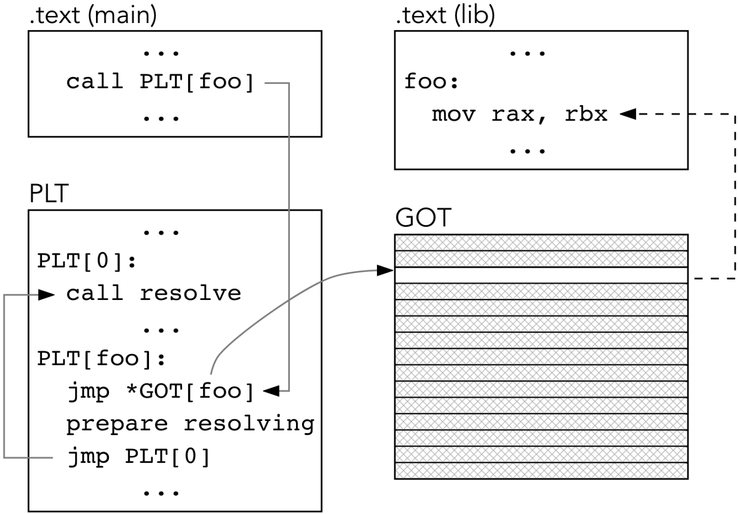
We do not want the linker to resolve them until needed.

So, implement a lazy binding using **Procedure Linkage Table**

PLT – before resolving



PLT – after resolving



PLT

- Resolving happens on first call.
- Later calls have penalty of an extra jump.
- Can be forced on load (see `LD_BIND_NOW` env var).
- PLT can be omitted with `-f-no-plt` (GCC 6).

Symbol addressing summary

Assuming the main executable file is PIC.

The symbol can be defined in an object and be:

- exported (process-global):
  Requires GOT/PLT
- local to the object:
  `rip`-relative addressing (for data) or relative offsets (for function calls).

Symbol addressing summary

Assuming the main executable file is **not** PIC.

The symbol can be:

- **Local to main program**: absolute addresses.
- **Local to PIC shared library**: `rip`-relative addressing (for data) or relative offsets (for function calls).
- **Defined in executable, used globally.**
  Accessed directly from the executable, requires GOT/PLT entry in libraries.
- **Defined in dynamic library and used globally.**
  MAY BE accessed directly in executable (optimization), requires GOT/PLT entry in libraries. **Usually ends up in main program during execution, because the executable is always first in lookup scope**. See example `var-moved-to-main`.

Coding time

- Minimal assembly example (`ex1`); explore with readelf.
- Note a problem with hardcoded linker name.
- Using `ldd`
- Simple library in C (`ex5`).
- Writing and exploring a shared library in Assembly (`ex2`, (`ex3`).
- Interfacing an assembly library with C (`ex4`).

Constructors and destructors

```
void
__attribute ((constructor))
init_function (void)
{ ... }

void
__attribute ((destructor))
fini_function (void)
{ ... }
```

Order of execution: topologically sorted objects w.r.t. dependencies.

Slow, repeat for destructor execution order.

Do not override `_fini` and `_init`, they are used!.

Visibility control

How to make global-global and local-global symbols?

GCC has four visibility types.

Control with:

```
> gcc -fvisibility=hidden ... # hide all symbols
```

Explicitly visible:

```
int __attribute__(( visibility( "default" ) ))
func(int x) { return 42; }
```

Other possibilities

- Export maps (linker scripts).
  - Careful with C++ because of mangling.
- Enclose in:

```
#pragma GCC visibility push(hidden)
...
#pragma GCC visibility pop
```

Optimization

Visibility control

Performance boosters already inside ELF files.

- Hash map: symbols $\mapsto$ addresses.
  `readelf -I`
- GNU extension: add Bloom filter (does this object even define a symbol x?).

-fPIC vs -fpic

On Intel 64 they are the same.

There exist architectures where `-fpic` produces smaller and faster code (but there are architecture-dependent limitations).

General advice

- ALWAYS use `-fpic` or `-fPIC` on libraries. Otherwise certain optimizations can screw your code semantics.
- Get rid of relocations in .data and .rodata.
- Hide everything you can (visibility).
- Less objects (initializers/finalizers, loading, lookup scope...)
- Symbol table length might be not optimal.
- Comparing long strings is costly (C++ names are very long).
- Place everything you can in .rodata

Relocations and data types – 1

```
char* hello = "hey";
```

OR

```
const char hello[] = "hey";
```

?

Relocations and data types – 2

```
static const char *msgs[] = {
    [ERR1] = "message for err1",
    [ERR2] = "message for err2",
    [ERR3] = "message for err3"
};
```

has 3 relocations.

```
static const char msgs[][17] = {
    [ERR1] = "message for err1",
    [ERR2] = "message for err2",
    [ERR3] = "message for err3"
};
```

has none (but wastes memory if different elements length OR ERR1..3 are not consecutive or far from zero).

Relocation and data types – 2.1

```
static const char msgstr[] =
    "message for err1\0"
    "message for err2\0"
    "message for err3";

static const size_t msgidx[] = {
    0,
    sizeof ("message for err1"),
    sizeof ("message for err1")
    + sizeof ("message for err2")
};

const char *errstr (int nr) {
    return msgstr + msgidx[nr];
}
```

Getting rid of PLT

PLT is not used if a symbol is local-global.

- Adapters;
- Aliases (see example `ex-alias`).

Overriding symbols

Using LD_PRELOAD

LD_PRELOAD is an environment variable.

Stores path to libraries to load before anything else.

These have the highest priority (after the executable).

Example: overriding glibc functions

`glibc` is usually dynamically linked.

Let us override a function! (see `override` example).

Code models

What is a code model

In our case, **code model** is a mode of generating instructions related to addressing.

RIP-relative addresses are limited by offsets of 32 bits. What if we need more?

Usually it is enough; no need to waste more bytes clogging the instruction cache.

Otherwise we use multiple instructions, like:

```
mov rax, 0x12345
mov rcx, [rax]
```

gcc option: `-mcmodel=small/medium/large`

Besides kernel-related, there are six code models:

- PIC or Non-PIC modifier;
    - Small – what we are used to
    - Large – all offsets are now 64 bit, performance is hit.
    - Medium – like small, but selected huge arrays are addressed like in large.

Examples follow.