Dynamic linking in C++

Filip Strömbäck

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Motivation

Why should we know about *linking*?

- Understand and fix errors/warnings
- Motivates the design of some parts of C++
  - Easier to remember “rules” when you know why
- Allows reasoning about optimizations (inlining)
- Utilize the powers of dynamic linking (plugin systems, etc.)
- (Abuse the “strange” corners of dynamic linking)

Scope

(Dynamic) linking is currently **not** standardized. You will soon see why.

- Main focus: UNIX (Linux + elf)
  - `.o`, `.a`, `.so`
- Also: Windows
  - `.obj`, `.lib`, `.dll`
- Some of the examples exhibit undefined behavior, be careful!

Tools

How do we see what happens?

- Compiler (`gcc`)
- Debugger (`gdb`)
- Binutils (`ar`, `readelf`, `objdump`)
- The dynamic linker + tools (`ld.so`, `ldd`)
- Various test programs and a lot of curiosity

LINKÖPING UNIVERSITY

Example

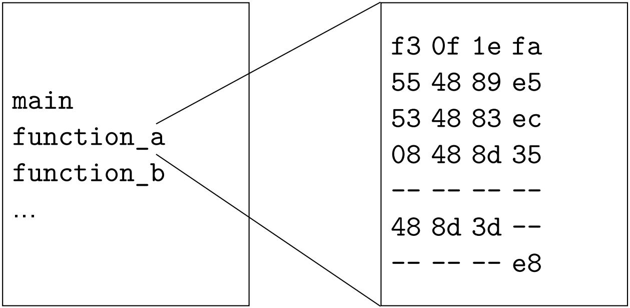
Consider the code in `01_multi-files`

- Why does this work?
- Why can't I modify the parameter to `function_a`?

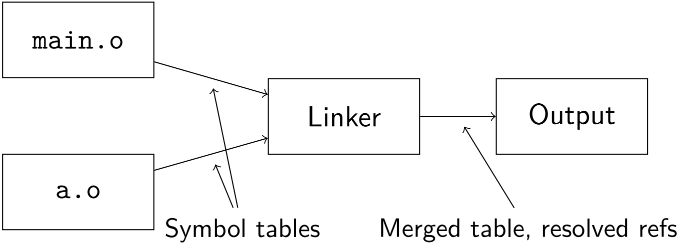
Your program from the linker's perspective

```
main
function_a
function_b
...
```

Your program from the linker's perspective



Your program from the linker's perspective

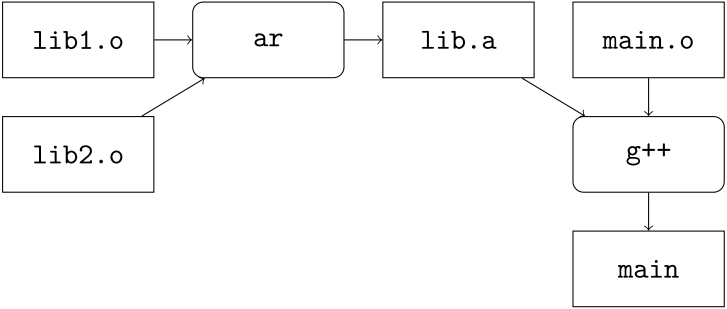


- `objdump -t <file>` – show symbol table
- `objdump -t -C <file>` – show symbol table
- `objdump -d -C <file>` – disassemble code

LINKÖPING UNIVERSITY

Brief history: Static libraries

What if we have a "large" library? Inconvenient to distribute many `.o` files...



What is an .a-file?

Consider the code in `02_static-lib`.

- `ar t <file>` – list members
- `ar x <file>` – extract members

Problems with static linking

Static linking **copies** code into the final binary. This means:

- The final binary becomes larger, both on disk and in RAM
- Fixing a bug in the library requires re-compiling all programs using it

LINKÖPING UNIVERSITY

Dynamic linking

Idea: leave symbols undefined and resolve them when loading the program

We then let the *dynamic linker* handle linking of *shared libraries*

- Avoids copies of code, both on disk and in RAM
- We can easily update the library
- Makes loading code at runtime easier

Practicalities

- Modelled to work like static linking
- This is what the `-l` flag does. Two forms:
  - `-l<x>` ⇒ finds `lib<x>.so`
  - `-l:<x>` ⇒ finds `<x>`
- Default: system's library path, we can use `-L` to modify this
- The dynamic linker also needs to know where to look
  - `rpath` or `runpath`
- Code must be *position independent*: `-fPIC`

Dynamic linking

Consider the code in `03_shared-lib`

- `readelf -h <program>` – show headers
- `readelf -l <program>` – show program headers
- `readelf -d <program>` – inspect dependencies
- `objdump -T <program>` – inspect dynamic symbol table
- `ldd <program>` – inspect behavior of dynamic linker

Multiple dynamic libraries

Consider the code in `04_multi-shared`. We have two libraries, `lib1.so` and `lib2.so` linked to our executable.

- Try uncommenting `lib_name` in `lib1.cpp`
- Try uncommenting `print_greeting` in `lib1.cpp`
- Try uncommenting `check_int` in `main.cpp`

⇒ The symbols in *all* libraries form a *single* namespace!

  - Order based on appearance on command line

- How do we fix this?

Library isolation

Linux (UNIX in general):

- Symbols have *visibility*:
  - default – visible outside the shared object
  - hidden – only visible inside the shared object
  - internal – only called from the same module
  - protected – can not be overridden by another module
- We can set default with `-fvisibility=hidden`
- We can use `static` and anonymous namespaces.

Differences on Windows

Windows takes a different approach:

- Symbols are hidden by default
  - Explicitly export symbols `__declspec(dllexport)`
  - Explicitly import symbols `__declspec(dllimport)`
- Compiling a DLL makes the DLL and a *import library* (`.lib`)
- Link with the *import library* to use the DLL
- Search path typically include executable's path by default

Sidenote: system calls

Linux:

- Has a well-defined interface for system calls
- C library implements a wrapper for these

Windows:

- Exposes system calls through functions in DLLs
- Does not need to define how system calls are performed

Calling the dynamic linker

We can load libraries dynamically by calling the dynamic linker (`-ldl` on Linux):

- `dlopen` or `LoadLibrary` – load a shared library
- `dlclose` or `FreeLibrary` – unload a shared library
- `dlsym` or `GetProcAddress` – get the address of a symbol
  Note: name mangling differs between systems, even for C!

Calling the dynamic linker

Consider the code in `05_dlsym`

- Try running `./main ./lib1.so` and `./main ./lib2.so`
- What happens if we specify `RTLD_NOW`?
- Why do we need `RTLD_GLOBAL`?
- Why don't we get an error when linking `lib2.so`?
  - We can add `-Wl,-z,defs`
- Why can't we add `do_fun_stuff` in main executable?
  - We can link with `-rdynamic`

LINKÖPING UNIVERSITY

What can the compiler assume?

Consider the code in `06_dynamic-rebind`

- Try running `./main`
  and `LD_PRELOAD=inject.so ./main`
- Unless visibility is set, symbols may be overwritten

$\Rightarrow$ Compiler is not able to inline/reason about functions

Patch internal functions

Consider the code in `07_patch`:

- If you know the internals of a library, it is possible to intercept and patch functionality...

Instrument a program

Consider the code in `08_instrument`:

- We can inject our minimal library anywhere we want
- For example: `LD_PRELOAD=./track.so /usr/bin/echo hello`

Windows

- Stronger guarantees by default: compiler is able to reason about the code to a larger extent
- Strong isolation leads to other peculiarities. In particular, we may have multiple copies of the same thing:
  - metadata: must compare *names* of types, rather than pointers
  - globals: sometimes we have multiple *heaps*, must allocate and free from the same DLL
- All symbols must be resolved, more work to make "pluggable" interfaces
- We can still "bad things", but they require more work

Implications for library design

There are many things to consider when writing libraries. Good API design is important, and we need to consider how linking works:

- Consider visibility, especially for internal functions
- Memory allocated by your library might need to be freed by your library
- You might have multiple instances of code and/or data

Filip Strömbäck

www.liu.se

LINKÖPING UNIVERSITY