# Static libraries

# Overview

- Shared libraries
- First simple static library
- Static library within shared library
- Rules for symbol inclusion
- Order of static libs on command line when linking
  - Circular references
- Hiding symbols when linked into shared libraries

# Shared libraries

```
$ gcc -fPIC -c so01.c -o so01.o
$ gcc -shared so01.o -o so01.so
$ gcc -c main.c -o main.o
$ gcc main.o so01.so -o test01
$ ./test01
$ ./test01: error while loading shared libraries: so01.so: cannot open
shared object file: No such file or directory
$ gcc main.o so01.so -Wl,-rpath,$PWD -o test02
$ ./test02
In func01 in so01.so
```

Important to note:
- Code in .so-files need to be compiled position independent (PIC)
- GCC needs to be told the output file is a shared object file
- .so-files can be passed on command line directly (“-L” isn’t necessary)
- “-Wl,…” allows for passing argument to linker invoked by gcc
- Runtime linker only looks in default locations, you can set rpath during linking, or use LD_LIBRARY_PATH variable during runtime

# Static libraries

```
$ gcc -c a01.c -o a01.o
$ rm -f a01.a
$ ar qc a01.a a01.o
$ ranlib a01.a
$ gcc -c main.c -o main.o
$ gcc main.o a01.a -o test01
$ ./test01
In func01 in a01.a
```

Differences from shared libraries:

- Instead of gcc, we use ar (“archive”) to create libraries.
- Argument “q” means “quick append”, no files are replaced
- Argument “c” means “create”, otherwise it will issue a warning when output file doesn’t exist yet
- Since “q” doesn’t check if file is already there “rm -f” will allow us to start with a fresh archive
- “ranlib” will prepare archive for use as static lib: it will create an index of all symbols in library

# Static libraries in shared libraries

```
$ cat main.c
void func01();
int main() {
   func01();
}

$ cat a01.c
#include <stdio.h>
void func01() {
   printf("In func01 in a01.a\n");
}
```

Test setup:
- app containing “main()” will link to “so01.so”, that only contains “a01.a”, that only contains “a01.c”:

test01(main()) —> so01.so () —> a01.a(func01())

# Static libraries in shared libraries

```
$ gcc -fPIC -c a01.c -o a01.o
$ ar qc a01.a a01.o
$ ranlib a01.a
$ gcc -shared a01.a -o so01.so
$ gcc -c main.c -o main.o
$ gcc main.o so01.so -Wl,-rpath,$PWD -o test01
main.o: In function `main':
main.c:(.text+0xa): undefined reference to `func01'
collect2: error: ld returned 1 exit status
$ gcc -shared -Wl,--whole-archive a01.a -Wl,--no-whole-archive -o so01.so
$ gcc main.o so01.so -Wl,-rpath,$PWD -o test01
$ ./test01
In func01 in a01.a
```

Explanation:

- Files need to be compiled with PICode, when used (indirectly) in .so
- When linking static library, only object files needed are linked in
- “--whole-archive” overrides this behaviour: all object files are
- Still need to pass “--no-whole-archive”, so rest of (hidden) command line isn’t messed up

# When shared lib only references part of static lib

```
$ cat a.c
#include <stdio.h>
void funca1() {
   printf("In funca1 in a01.a\n");
}
void funca2() {
   printf("In funca2 in a01.a\n");
}
$ cat b.c
#include <stdio.h>
void funcb1() {
   printf("In funcb1 in a01.a\n");
}
$ cat so01.c
void soFunc() {
   funca1();
}
```

Test setup:
- “so01.so” will consist of “a.a”, “b.a”, and “so01.c”
- “so01.c” only references “funca1”, in “a.c”, in “a.a”, but not “funcb1”

# When shared lib only references part of static lib

```
$ gcc -fPIC -c a.c -o a.o
$ gcc -fPIC -c b.c -o b.o
$ ar qc a.a a.o b.o
$ ranlib a.a
$ gcc -fPIC -c so01.c -o so01.o
$ gcc -shared so01.o a.a -o so01.so
$ nm -D so01.so | grep func
0000000000000721 T funca1
0000000000000734 T funca2
```

Explanation:
- Whenever an .a-file is linked in, only object files containing required symbols are linked in
- So not only the symbol itself, but everything of that object file
- Object files not containing missing symbols are ignored

# Order of static libs when linking

```
$ cat inner01.c
#include <stdio.h>
void inner01() {
   printf("In inner01\n");
}
$ cat outer01.c
#include <stdio.h>
void outer01() {
   printf("In outer01\n");
   inner01();
}
$ cat main.c
#include <stdio.h>
int main() {
   printf("In main\n");
   outer01();
   return 0;
}
```

Test setup:

- We will create a simple executable that contains one object file and two static libs that reference each other like this:

  main.o —> outer01.a —> inner01.a

# Order of static libs when linking

```
$ ls *.a main.o
inner01.a  main.o  outer01.a
$ gcc inner01.a outer01.a main.o -o test05
main.o: In function `main':
main.c:(.text+0x14): undefined reference to `outer01'
collect2: error: ld returned 1 exit status
$ gcc inner01.a main.o outer01.a -o test05
outer01.a(outer01.o): In function `outer01':
outer01.c:(.text+0x14): undefined reference to `inner01'
collect2: error: ld returned 1 exit status
$ gcc main.o inner01.a outer01.a -o test05
outer01.a(outer01.o): In function `outer01':
outer01.c:(.text+0x14): undefined reference to `inner01'
collect2: error: ld returned 1 exit status
$ gcc main.o outer01.a inner01.a -o test05
$ ./test05
In main
In outer01
In inner01
```

Explanation:

- The linker goes left-to-right through files, adding all symbols from object files, and adding object files containing missing symbols from static libs
- Order matters: every file is visited only once, symbols needed later on aren’t collected

# Circular references

```
$ cat inner01.c
#include <stdio.h>
void inner01() {
   printf("In inner01\n");
   outer02();
}
$ cat outer02.c
#include <stdio.h>
void outer02() {
   printf("In outer02\n");
}
```

Test setup:
- “outer.a” still references “inner01.a” from one of its object files.
- “inner01.a” references a different object file from “outer.a”

main.o —> outer.a —> inner01.a —> outer.a

# Circular references

```
$ ls *.a main.o
inner01.a  main.o  outer.a
$ gcc main.o outer.a inner01.a -o test06
inner01.a(inner01.o): In function `inner01':
inner01.c:(.text+0x14): undefined reference to `outer02'
collect2: error: ld returned 1 exit status
$ gcc main.o outer.a inner01.a outer.a -o test06
$ echo $?
0
$ gcc main.o -Wl,-\( outer.a inner01.a -Wl,-\) -o test06
$ echo $?
0
```

Explanation:
- Once a static lib has been processed, it won't be visited again for missing symbols
- You are free to repeat an earlier passed static lib
  - There is a risk you will have to do that several times to resolve linking errors
- The linker recognises "-(" and "-)" to group static libs to resolve circular dependencies
  - We need to add "-Wl," to tell gcc that it is for the linker stage
  - We need to escape "(" and ")" so bash won't treat it as a subshell

# Hiding functions

```
$ cat secret01.c
#include <string.h>
void GetSuperSecretPassword(char buffer[]) {
   strcpy(buffer, "the_password");
}
$ cat so01.c
#include <stdio.h>
void DoSomething() {
   char buffer[32];
   GetSuperSecretPassword(buffer);
   printf("Got the password: \"%s\"\n", buffer);
}
```

Test setup:
- We want to hide the “GetSuperSecretPassword” function to the outside, but make it so it can be used from within the .so-file
- Application: “GetDrmId(…)”, “GetDeviceId(…)” inside “libplayready.so” and Netflix library

# Hiding functions

```
$ ls *.a so01.o
secret01.a  so01.o
$ gcc -shared so01.o secret01.a -o so01.so
$ nm -D so01.so | grep GetSuperSecretPassword
00000000000007b2 T GetSuperSecretPassword
$ strip so01.so
$ nm -D so01.so | grep GetSuperSecretPassword
00000000000007b2 T GetSuperSecretPassword
```

Explanation:

- We didn’t tell the linker to include all symbols from secret01.a
- It had to be included because so01.o references it
- Once it is included, it is made available
- Stripping won’t help either, since it is a public symbol

# Hidden visibility attribute

```
$ cat secret01.h
__attribute__((visibility ("hidden"))) void GetSuperSecretPassword(char
buffer[]);
$ cat secret01.c
#include "secret01.h"
#include <string.h>
void GetSuperSecretPassword(char buffer[]) {
   strcpy(buffer, "the_password");
}
```

“__attribute__((visibility (“hidden”)))":
- It tells linker not to make symbol publicly available
- Is used to speed up runtime linking
- We use it to hide functions

# Hidden visibility attribute

```
$ ls *.a so01.o
secret01.a  so01.o
$ gcc -shared so01.o secret01.a -o so01.so
$ nm -D so01.so | grep GetSuperSecretPassword
$ strings so01.so | grep GetSuperSecretPassword
GetSuperSecretPassword
$ readelf --symbols so01.so | grep GetSuperSecretPassword
42: 0000000000000762    39 FUNC    LOCAL  DEFAULT   12 GetSuperSecretPassword
$ strip so01.so
$ readelf --symbols so01.so | grep GetSuperSecretPassword
$
```

Explanation:

- “__attribute__((visibility (“hidden”)))” indeed makes a function invisible
- Yet it is still in the .so-file as a “LOCAL” function
- Stripping the .so-file gets rid of it