Table of Contents

1. Overview

1.1. Introduction

This document provides information regarding the General Knowledge lesson.

1.2. References

Table 1-1: References

No.	Documents	Description
1	01_General_Knowledge.pdf	The lesson lecture

1.3. Environment

Table 1-2: Environment

Type	Component	Information
OS	Ubuntu	22.04.5 LTS
Kernel	Linux kernel	6.8.0-65-generic
GNU C Compiler	GCC	11.4.0
Build operation tool	GNU Make	4.3

1.4. Folder structure

Below is the folder tree.

```
<workdir>
    ├── doc                 --- All necessary documents
    │   └── Document.pdf
    ├── include             --- Location for headers
    │   └── dl_strutils.h
    ├── Makefile            --- Makefile
    ├── sample              --- Sample application
    │   └── main.c
    └── src                 --- Library source files
        └── dl_strutils.c
```

Figure 1-1: Folder tree

1.5. Glossary

Table 1-3: Glossary

Abbr.	Description
ELF	Executable and Linkable Format

2. Knowledge

2.1. Makefile

Please refer to section 1 of [1].

2.2. The process of compiling a C program

Please refer to section 2 of [1].

2.3. Static and Shared library

2.3.1. Static library

2.3.1.1. Definition

A static library (known as `archive`) is a collection of object files that are linked into a program (the main program) at compile time. After resolving the various functions references from the main program to the modules in the static library, the linker extracts copies of the required object modules from the library and copies these into the resulting executable file.

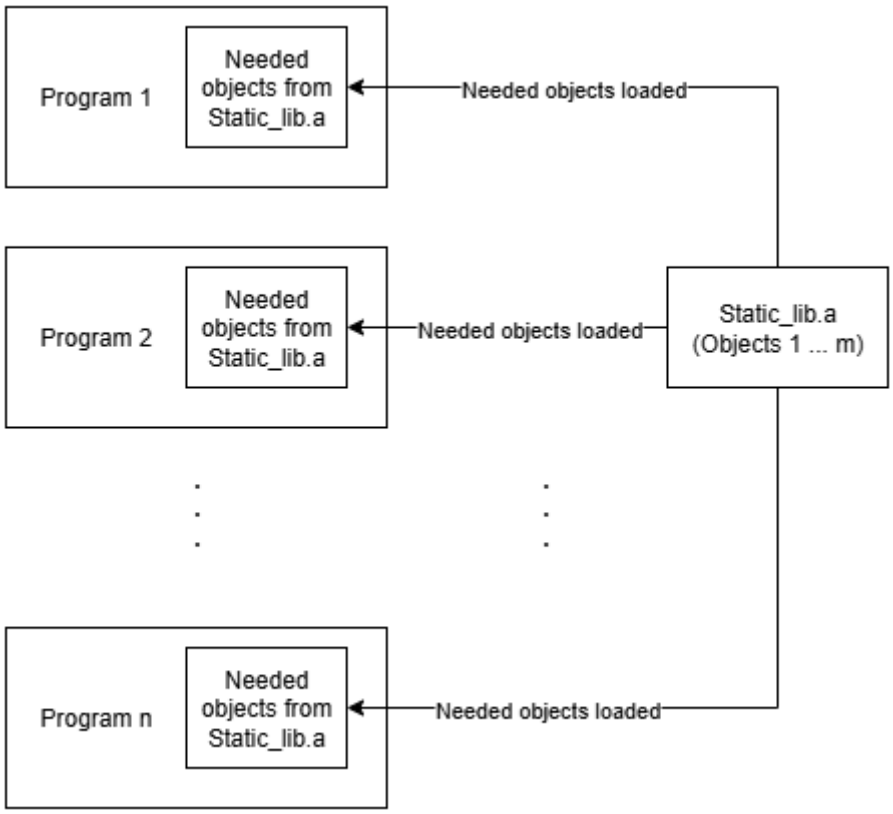

Figure 2-1: Relationship between programs and a static library

2.3.1.2. File format

Static libraries typically have a `.a` file extension in Linux.

2.3.1.3. How to create

Building a static library involves three steps: source (`.c`) → object (`.o`) → archive (`.a`).

Step 1: Create object (`.o`) files:

Please run this command to make the required object file for shared library.

```
gcc <build flags> -c                                \
    <path to lib source 1>/<lib source 1>.c   \
    <path to lib source 2>/<lib source 2>.c   \
                        .
                        .
                        .
    <path to lib source n>/<lib source n>.c   \
    -o <path to lib object>/<lib object>.o
```

Whereas,

- `<build flags>`: Compiler options, such as -Wall, -O2, etc. These options are optional.
- `-c`: A mandatory option which specifies source files.
- `-o`: An option followed by `<path to lib object>/<lib object>.o` to specify both the directory and the name of the generated object file. In the context of this document, using this option is mandatory to ensure the output file's type.

After running that command, you will have the object file `<lib object>.o` in `<path to lib object>`.

Step 2: Create a static library:

You can have multiple object files (`.o`) built in Step 1. Please run this command below in a terminal to create the static library from your object files.

```
ar rcs <path to static library>/<library>.a                \
       <path to object 1>/<object 1>.o   \
       <path to object 2>/<object 2>.o   \
                        .
                        .
                        .
       <path to object n>/<object n>.o
```

Whereas,

- `ar`: Command to make a static library.
- `rcs`: Options for `ar`:
  - `r` (replace): Insert the object files into the archive. If files with the same name already exist

in the archive, they are replaced with the new ones.

- `c` (create): Create the archive if it does not already exist.
- `s` (index): Write an index (also called a "symbol table") into the archive. This index speeds up linking because the linker can quickly find which object files define which symbols.

Then, you will have your static library called `<library>.a` in `<path to static library>`.

2.3.1.4. How to build an application linked with static libraries

Once you have your static library in `<path to static library>`, and your source files `<source file x>` (x = 1, 2, ..., n) in their respective directories `<path to source file x>`, let's just create the executable file linked with the static library.

```
gcc <build flags>                                \
    <path to app source 1>/<app source 1>.c   \
    <path to app source 2>/<app source 2>.c   \
                          .
                          .
                          .
    <path to app source n>/<app source n>.c   \
    -L<path to static library>                    \
    -l<library>                                   \
    -o <path to output executable>/<executable file>
```

Whereas,

- `<build flags>`: Compiler options, such as -Wall, -O2, etc. These options are optional.
- `-L`: An option which specifies a directory for the linker to search for required libraries. If you have more than just one directory, you should use several `-L` to specify them. This option is optional but highly recommended in our case. If it is omitted or the linker fails to find the required library in the directories specified by this option, then the linker will search the standard system library directories (such as `/lib` and `/usr/lib`, etc.).
- `-l`: An option which specifies a library by name. If you need to link multiple libraries, you should use multiple `-l` options, each of which is preceded by the appropriate `-L<directories>` to tell the linker where to search. This option is convenient but optional: if you do not use `-l`, you can still link by explicitly writing the full static library filename `<library>.a`.
- `-o`: An option followed by `<path to output executable>/<executable file>` to specify both the directory and the name of the generated executable file. In the context of this document, using this option is mandatory in order to explicitly control the output location and filename.

After running that command, you will have the executable file `<executable file>` in `<path to output executable>`.

In the directories searched by the linker, if both a static library and a shared library with the same name are present and the full filename is not specified explicitly, the shared library has higher priority and will be linked by default.

You can instruct the linker explicitly with these options:

- `-Wl,-Bstatic -l<library>`: Tell the linker to use the static version

	(`lib<library>.a`). • `-Wl,-Bdynamic -l<library>`: Tell the linker to use the shared library (`lib<library>.so`).

2.3.1.5. How to run the application linked with static libraries

You can simply run the application by executing this command in a terminal.

```
<path to output executable>/<executable file>
```

If you receive a message which says 'Permission denied' then you should change the permission of the application.

```
chmod +x <path to output executable>/<executable file>
```

Then run the application again, it should work.

2.3.1.6. Pros and cons

Static library has some pros and cons mentioned as below.

Pros:

- **Performance**: Since the library is included in the executable, there is no overhead of dynamic linking at runtime. Therefore, better performance is likely to be gained.
- **Portability**: The executable can run on any system without needing the library file.

Cons:

- **Size**: The executable can become larger since it contains all the library code.
- **Updates**: If the library is updated, the executable must be recompiled to use the new version.

2.3.1.7. When to use

With those pros and cons mentioned in Pros and cons, these usages can be considered:

When code is stable and reused across many programs:

If you have a set of utility functions (e.g., math routines, string processing, logging) which rarely change, you can compile them once into a static library (`.a` file) and reuse them across multiple projects.

When you want self-contained executables:

With a static library, the linker copies the required object code into the final executable. This means:

- No need to ship external `.so` (shared library) files.
- The program can run even on systems that do not have the library installed.

This is useful for embedded systems or distributing a single portable binary.

When performance and startup speed matter:

Since the linker resolves everything at compile time:

- The program does not need to resolve symbols at runtime.
- Startup is usually faster compared to dynamically linking against shared libraries.

When library versioning is not a concern:

A static library “freezes” the code inside your executable.

- **Advantage**: your program will not break if someone upgrades or removes a system library.
- **Disadvantage**: if a bug or security issue is fixed in the library, you need to recompile your executable to include the fix.

2.3.2. Shared library

2.3.2.1. Definition

A shared library is a collection of compiled object files that is stored in a single file and loaded into memory at runtime so that multiple programs can use its functions and data simultaneously.

During linking, the linker does not copy the code from the shared library into the executable; instead, it records the library's name in the `DT_NEEDED` entry of the ELF dynamic section. When the executable is loaded into memory, the dynamic linker reads this information, searches for the corresponding shared library file, and maps it into the process address space.

At that point, the unresolved symbols in the executable — such as functions and global variables — are resolved by assigning them actual memory addresses within the loaded shared library, allowing the program to call the library code as if it were part of the executable itself.

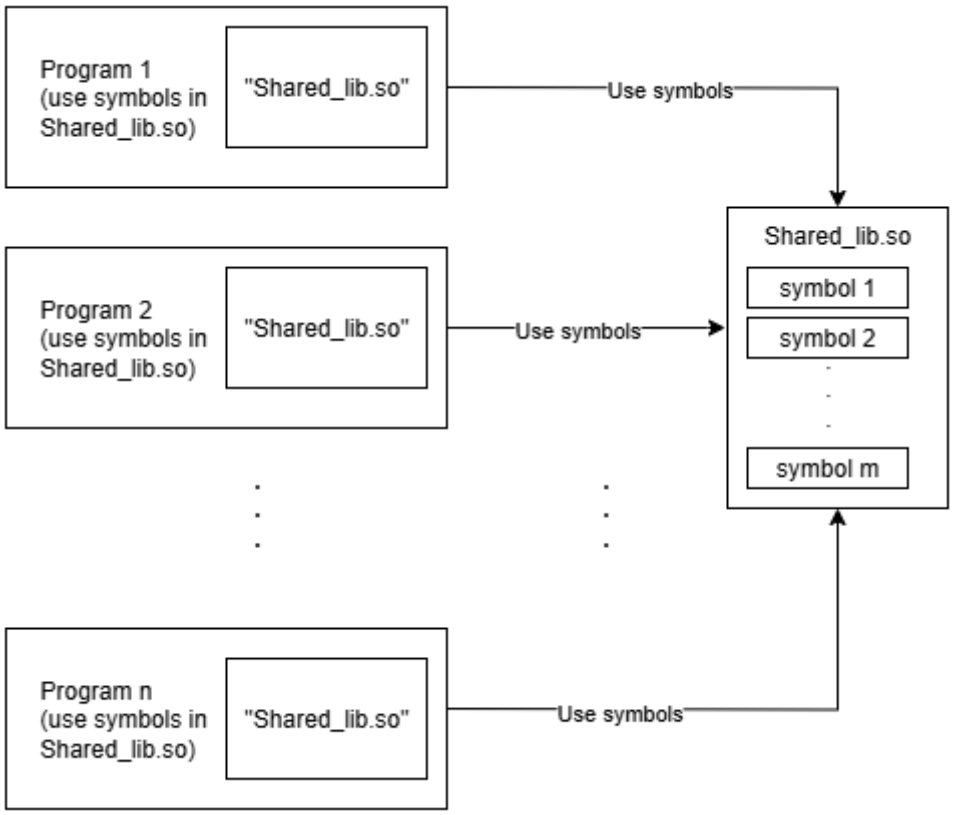

Figure 2-2: Relationship between programs and a shared library

2.3.2.2. File format

Shared libraries typically have a `.so` in their file extension in Linux. For examples:

- `libm.so` (math library)
- `libc.so.6` (GNU C library)
- `libpthread.so.0` (POSIX threads)
- `.so.<version>` like `libabc.so.1`, `libabc.so.1.2.3`, etc.

2.3.2.3. How to create

Building a shared library involves three steps: source (`.c`) → object (`.o`) → shared library.

Step 1: Create object (`.o`) files:

Please run this command to make the required object file for shared library.

```
gcc <build flags> -fPIC -c                      \
    <path to lib source 1>/<lib source 1>.c   \
    <path to lib source 2>/<lib source 2>.c   \
                        .
                        .
                        .
    <path to lib source n>/<lib source n>.c   \
    -o <path to lib object>/<lib object>.o
```

Whereas,

- `<build flags>`: Compiler options, such as -Wall, -O2, etc. These options are optional.
- `-fPIC`: PIC stands for Position-Independent Code. This option tells the compiler to generate a machine code which is independent of fixed memory addresses. This is crucial for creating a shared library.
- For other options, please refer to Create object file for static library.

After running that command, you will have the object file `<lib object>.o` in `<path to lib object>`.

Step 2: Create a shared library:

You can have multiple object files (`.o`) built in Step 1. Please run this command below in a terminal to create the shared library from your object files.

```
gcc -shared -o <path to shared library>/<library>.so      \
       <path to object 1>/<object 1>.o   \
       <path to object 2>/<object 2>.o   \
                        .
                        .
                        .
       <path to object n>/<object n>.o
```

Whereas,

- `-shared`: This option tells the linker to create a shared library rather than an executable file. This option is mandatory.
- `-o`: An option followed by `<path to shared library>/<library>.so` to specify both

the directory and the name of the generated shared library file. In the context of this document, using this option is mandatory in order to explicitly control the output location and filename.

Then, you will have your shared library called `<library>.so` in `<path to shared library>`.

2.3.2.4. How to build an application linked with shared libraries

Once you have your shared library in `<path to shared library>`, and your source files `<source file x>` (x: 1, 2, ..., n) in their respective directories `<path to source file x>`, let's just create the executable file linked with the shared library.

```
gcc <build flags>                                  \
    <path to app source 1>/<app source 1>.c   \
    <path to app source 2>/<app source 2>.c   \
                          .
                          .
                          .
    <path to app source n>/<app source n>.c   \
    -L<path to shared library>                \
    -l<library>                               \
    -o <path to output executable>/<executable file>
```

For all the options, please refer to How to build an application linked with static libraries.

After running that command, you will have the executable file `<executable file>` in `<path to output executable>`.

2.3.2.5. How to run the application linked with shared libraries

At run time, the dynamic linker searches for the shared library in directories whose priorities are listed in descending order, as shown in the table below.

Table 2-1: Searched directories of descending priorities

No.	Directory
1	The actual directories recorded in `DT_NEEDED` in the executable file
2	The directory specified in `DT_RPATH` or `DT_RUNPATH` (if present in the executable file)
3	The environmental variable `LD_LIBRARY_PATH`
4	The system cache `/etc/ld.so.cache` (generated by `ldconfig`)
5	Standard system library directories (such as `/lib`, `/usr/lib`, `/usr/local/lib`, etc.)

For (2) in the Table 2-1, you can add this option `-Wl,-rpath,<path to shared library>` to the build command line in How to build an application linked with shared libraries.

For (3), you have to set the environmental variable `LD_LIBRARY_PATH` before running the application in the same terminal.

```
export LD_LIBRARY_PATH=<path to shared library>:${LD_LIBRARY_PATH}
```

Execute this command to run the application.

```
<path to output executable>/<executable file>
```

If you receive a message which says 'Permission denied' then you should change the permission of the application.

```
chmod +x <path to output executable>/<executable file>
```

Then run the application again, it should work.

2.3.2.6. Pros and cons

Shared library has some pros and cons mentioned as below.

Pros:

- **Size**: The executable is smaller because it does not contain the actual library code, only references to it.
- **Updates**: If the library is updated (e.g., bug fixes, security patches), the executable can immediately benefit without recompilation.
- **Memory efficiency**: Multiple processes can share the same copy of the library in memory, reducing overall memory usage on the system.

Cons:

- **Performance**: There is a small runtime overhead due to dynamic linking and symbol resolution.
- **Dependency**: The executable requires the shared library file to be present on the target system; if the library is missing or has an incompatible version, the program cannot run.
- **Complexity**: Managing library versions (SONAME, symlinks) and ensuring compatibility across different systems can add complexity.

2.3.2.7. When to use

With those pros and cons mentioned in Pros and cons, these usages can be considered:

Multiple programs need the same functionality:

If several executables depend on the same library, using a shared library reduces duplication of code in each binary.

You want to minimize executable size:

Since the executable only contains references to the library, it remains smaller compared to linking the same code statically.

Ease of updates and maintenance:

Updating the shared library (for bug fixes, new features, or security patches) immediately benefits all dependent programs, without the need to recompile them.

Memory efficiency:

When multiple processes run simultaneously, they can share the same copy of the library code in memory, lowering overall memory usage.

3. Exercise

Building and Using static & shared libraries in C.

In C programming, as a project grows, organizing source code into reusable modules is crucial. A library lets us package a set of related functions to share across multiple projects without copying source code.

Static library (`.a`): The library's code is copied and embedded directly into your executable at compile time. The executable is larger but runs independently.

Shared library (`.so` — Shared Object): The library's code is not embedded into the executable. Instead, the OS loads the library into memory at runtime, and multiple programs can share it.

3.1. Exercise 1

Build a String Utilities Library (`strutils`).

3.1.1. Source layout

- `strutils.h`: Declares the functions provided by the library, for example:
  - `str_reverse`: Reverses a string in place.
  - `str_trim`: Removes leading and trailing whitespace from a string.
  - `str_to_int`: Safely converts a string to an integer.
- `bstrutils.c`: Source implementing the functions declared in `strutils.h`.
- `main.c`: A small program to test the `strutils` library functions.

3.1.2. Package the library

- Build a static library: `libstrutils.a`.
- Build a shared library: `libstrutils.so`.

3.1.3. Use the library

Write a main program to test the `strutils` functions. Compile and link:

- With static library:

```
gcc main.c -L. -lstrutils -o main_static
```

- With shared library:

```
gcc main.c -L. -lstrutils -o main_shared
```

To execute main_shared, you must tell the OS where to find the `.so` file (hint: use the `LD_LIBRARY_PATH` environment variable).

3.2. Exercise 2

Automate with a `Makefile`.

Create a `Makefile` to automate all steps above, including these targets:

- `all`: Default target; builds both `main_static` and `main_shared`.
- `static`: Builds only `libstrutils.a` and `main_static`.
- `shared`: Builds only `libstrutils.so` and `main_shared`.
- `clean`: Removes all build artifacts (`.o`, `.a`, `.so`, and executables).

4. Solution notes

This chapter provide notes and information about the solutions to Exercise.

4.1. C Compilation

The following table lists all compiler options used in the solutions.

Table 4-1: C complier options used

Purpose	Option	Description
General Warning and Error Control	-Wall	Enables a common set of important warnings about questionable code
	-Wextra	Enables extra warnings that are not included in -Wall
	-Werror	Treats all warnings as errors, stopping compilation if any warning appears
Pedantic Checks	-Wpedantic	Warns if your code uses non-standard GNU extensions that are not in ISO C
	-pedantic-errors	Like -Wpedantic but treats those warnings as errors
Type Conversion and Shadowing	-Wshadow	Warns if a local variable shadows (hides) another variable with the same name from an outer scope
	-Wconversion	Warns when implicit type conversions may change a value (e.g., float to int)
	-Wsign-conversion	Warns when a value changes sign due to conversion (e.g., unsigned to signed)
	-Wcast-function-type	Warns when casting between incompatible function pointer types (calling such a pointer is undefined behavior)
Precision and Formatting	-Wdouble-promotion	Warns when a float is promoted to a double implicitly
	-Wformat=2	Enables strict format string checks for functions like printf() and scanf()
	-Wfloat-equal	Warns on direct equality/inequality comparisons of floating-point values (fragile due to rounding)
	-Wformat-truncation=2	Warns when bounded printf-style functions (e.g., snprintf) may truncate output (level 2 = stricter)
	-Wformat-overflow=2	Warns when printf-style formatting may overflow the destination buffer (level 2 = stricter)

Purpose	Option	Description
Memory Safety	-Wnull-dereference	Warns when the compiler detects a dereference of a NULL pointer
	-Wcast-align	Warns if a pointer cast results in a stricter alignment requirement
	-Wcast-qual	Warns when casting away const or volatile qualifiers
	-Wcast-align=strict	Like -Wcast-align but warns whenever a cast increases the required alignment (strictest mode)
	-Wstringop-overflow=4	Warns when built-in string ops (e.g., strcpy, memcpy) may overflow the destination (level 4 = most strict)
	-Wstringop-truncation	Warns when string operations may silently truncate the result
	-Walloca	Warns about use of alloca() (stack allocation; easy to misuse and non-portable)
	-Walloc-zero	Warns on zero-size allocations (e.g., malloc(0)), which are implementation-defined and error-prone
Static Analysis	-fanalyzer	Runs GCC's static code analyzer to detect potential runtime bugs (e.g., NULL dereferences, memory leaks)
Optimization and Debugging	-Og	Optimizes for debugging: keeps code easy to debug while still optimizing slightly
	-g	Generates debug information for use with debuggers like gdb

Purpose	Option	Description
Code Safety and Correctness	-Wundef	Warns if an undefined macro is used in #if or #elif without being checked with #ifdef
	-Wstrict-prototypes	In C, warns if a function is declared without specifying argument types
	-Wmissing-prototypes	Warns if a global function is defined without a prior prototype
	-Wpointer-arith	Warns for suspicious pointer arithmetic, like arithmetic on void*
	-Wwrite-strings	Makes string literals have const type to prevent accidental modification
	-Wunreachable-code	Warns about code that will never be executed
	-Wunused	Warns about anything declared but never used
	-Wunused-parameter	Warns when a function parameter is unused
	-Wunused-but-set-variable	Warns when a variable is written to but its value is never read
	-Wlogical-op	Warns about suspicious logical operations (e.g., && vs &, always-true/false tests)
	-Wduplicated-cond	Warns when an if/else if chain repeats the same condition
	-Wduplicated-branches	Warns when different branches contain identical code
	-Wstrict-overflow=5	Warns when the compiler assumes signed overflow is undefined and optimizes based on that (level 5 = most strict)
	-Woverflow	Warns about compile-time constant arithmetic that overflows the destination type
	-Wredundant-decls	Warns when an entity is declared multiple times in the same scope
	-Wnested-externs	Warns on extern declarations placed inside functions (confusing linkage/style)
	-Wmissing-noreturn	Warns when a function that never returns should be marked _Noreturn
	-Wmissing-declarations	Warns when a global function is defined without a prior prototype in a header
	-Winline	Warns when a function marked/expected to inline is not inlined (e.g., too large/complex)

5. Revision history

Version	Date	Chapter	Content
0.01	Aug 23rd, 2025	All	Newly created
0.02	Aug 24th, 2025	Chapter 2.4.1.5	Create a new chapter that provides guidance on running the application
		Chapter 2.4.1.4	Add note about library priority
0.03	Sep 3rd, 2025	Chapter 4	Create a new chapter for the solutions to exercises
		Chapter 2	Remove C compilation flags