# Static and Dynamic Libraries

## Libraries Motivation :

- Use the same code in some projects
- Divide the project into small independent parts
- Reduce memory usage while running many programs which running mutual code.
- Deployment and maintenance

## Archive (static library)

- An archive (or static library) is simply a collection of object files stored as a single file.
- When you provide an archive to the linker, the linker searches the archive for the object files it needs, extracts and links them into the program.
- The linker uses only the objects which are needed.

## Static Library Advantages :

- Portable - all in one file
- Simple to implement
- Loads quicker

## Static Library Disadvantages :

- Bigger executable
- Longer linking
- No ability to share resources

# How to build a static library :

You can create an archive using the ar command.

Archive files traditionally use a “.a” extension.

Example :

```
ar cqv libheap.a heap.o vector.o
```

c = create, q = quick add, v = verbose

Display library content :

```
ar t libheap.a
```

Will output :

```
heap.o
vector.o
```

## Linking with a static library

- If libheap.a is placed in standard library path (like /usr/local/lib), link with -lheap (omitting the lib) gcc test_heap.c –lheap
- Otherwise using linking flag -L **<library-path>** If in located in current directory, use ' . ' gcc test_heap.c -lheap -L.
- Or explicitly : gcc test_heap.c libheap.a

## How does ar works ?

- The linker searches the archive for all definitions of symbols (functions and variables) that are referenced from the object files that it has already processed but are not yet defined.
- The object files that define those symbols are extracted from the archive and included in the final executable.

## Libraries usage in the link

- The linker searches the archive when it is encountered on the command line.
- If the archive appears before the object which needs it, no object from the archive might be used.
- Therefore, the archive should appear last
  gcc test_heap.c -lheap -L. (good)
  gcc -lheap test_heap.c -L. (bad)

## Vector makefile

```
LIB_OBJS=vector.o
OBJS=$(LIB_OBJS) vector_test.o
LIB_DIR=../lib
LIB=$(LIB_DIR)/libDS.a

lib: $(LIB)

$(LIB): $(LIB_OBJS)
	mkdir -p $(LIB_DIR)
	ar rcs $(LIB) $(LIB_OBJS)
```

## Heap makefile

```
OBJS=heap.o heap_test.o
LIB_DIR=../lib
CFLAGS+=-I../vector
LDFLAGS=-L$(LIB_DIR) -lDS

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJS) $(LDFLAGS)
```

## Main makefile 1/2

```
# Directories to use for the library creation:
DIRS=vector list
# Object files to use for the library creation:
LIB_OBJS=vector/vector.o list/list.o list/list_iter.o
# Path to the target library:
STATIC_LIB=lib/libDS.a

# Create the static library as archive file:
$(STATIC_LIB): dirs_make $(DIRS)
	ar -rcs $(STATIC_LIB) $(LIB_OBJS)
```

## Main makefile 2/2

```
# Make all folders by target lib:
dirs_make:
	$(shell for dir in $(DIRS); do \
		echo "make lib -C $$dir;"; \
	done)

clean:
	$(shell for dir in $(DIRS); do \
		echo "make clean -C $$dir;"; \
	done)
```

## DS library from vector and list

The makefiles of the specific data structures should not create the library. Just compile the C files.
Leave in vector makefile, only that:

```
LIB_OBJS=vector.o
OBJS=$(LIB_OBJS) vector_test.o

lib: $(LIB_OBJS)
```

## Shared libraries

- A shared library is similar to a archive in that it is a grouping of object files.
- Also called shared object or dynamically linked library.
- When a shared library is linked into a program, the final executable does not actually contain the code that is present in the shared library.
- The executable references to the shared library

## Shared libraries basics

- A shared library is an ELF file.
- If several programs on the system are linked against the same shared library, they will all reference the same library.
- Only one instance of the library exists in memory.
- When the last of the processes that reference it will terminate, the library will be released.
- The object files that compose the shared library are combined into a single object file (the shared object).
- A program that links against a shared library always loads all of the code in the library, rather than just those portions that are needed.

## PIC - Position-Independent Code

- Shared objects, can be loaded at different addresses in different processes.
- Addresses in the shared object must be relative.
- With position independent code you have to load the address of your function and then jump to it.
- Global offset tables hold absolute addresses of PIC in private data which maps PIC to absolute locations.

## Creating a shared library

- Object files in the shared library must be compiled using the -fPIC compiler flag :

  ```
  gcc -c -fPIC vector.c $(CFLAGS)
  ```

  PIC stands for “Position Independent Code", which means that the addresses are not absolute.

After compiling the files (using -fPIC), we link them :

```
gcc -shared -fPIC -o libDS.so $(OBJS)
```

The -shared option tells the linker to produce a shared library rather than an executable.

## Programs using a shared library

- Shared libraries use the extension .so , which stands for **S**hared **O**bject.
- Like static archives, the name always begins with lib to indicate that the file is a library.
- A program that was linked with a shared library specifies the library to be loaded.

## Linking with a shared library

- Linking with a shared library is similar to linking with a static archive
- For example, the following line will link with libDS.so , if it is in the current directory, or one of the standard library search folders on the system :

```
gcc -o application.out $(OBJS) -L. –lDS
```

## Linker libraries seeking

- Suppose that both libDS.a and libDS.so are available.
- The linker searches each directory :
  First those specified with -L options, and
  Then those in the standard directories
- The linker stops after finding libDS.a or libDS.so.

## Linker libraries preferences

- If only one of the two variants is present in the directory, the linker chooses that variant.
- Otherwise, the linker chooses the shared library version, unless explicitly instructed otherwise.
- -static flag will force choosing static archives.

```
gcc -static -o application.out $(OBJS) -L. –lDS
```

## Shared object loading problem

- The linker places only the name of the shared library in the program which is linking with it.
- When the program is actually ran, the system searches for the shared library and loads it.
- The system searches only the system libraries by default ( /lib and /usr/lib ).
- What if the shared object is located elsewhere?

## Determine shared object position

- Run-time search path ( rpath)
- The linker can save the rpath in the executable.

```
LDFLAGS+=-Wl,-rpath=$(CURDIR)/lib/
```

- -L is used by the linker in order to resolve symbols.
- The program uses rpath to locate library in runtime.

## LD_LIBRARY_PATH

- Environment variable which extends the system's shared objects search path.
- Script example :

```
export LD_LIBRARY_PATH=$PWD/lib
./executable.out
```