Library Tutorial

1. Introduction

This tutorial intends to help you understand how to make and use static and shared libraries in the Linux environment. This tutorial specifically puts its emphasis on the difference between static library and shared library.

2. Using a library

As an example, below is a simple program that finds the solutions of a given quadratic equation. It accepts the three coefficients, i.e., a, b, and c, from the user, then calculates the real solutions, if any. Note that this program uses a math function sqrt, which is defined in math.h.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
    double a, b, c, d;
    double root1, root2;

    printf("ax^2 + bx + c\n");
    printf("Enter a b c: ");
    scanf("%lf %lf %lf", &a, &b, &c);

    d = b * b - 4 * a * c;

    if (d < 0) {
        printf("No real root\n");
    }
    else {
        root1 = (-b - sqrt(d)) / (2 * a);
        root2 = (-b + sqrt(d)) / (2 * a);
        printf("%f %f\n", root1, root2);
    }
    return 0;
}
```

solver.c

Below is the first trial to compile test.c that turns out to be an error. By investigating the error messages, we can find that the linker (ld) could not find the sqrt function in given source files.

```
$ gcc solver.c
/tmp/cckRFVmy.o: In function `main':
test.c:(.text+0xae): undefined reference to `sqrt'
test.c:(.text+0xdf): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

To solve this problem, all you have to do is to append –lm option as below, which tells gcc that this program requires an additional library whose name is “m”. Then gcc tries to find the library named “m” in several pre-defined directories. Finally, gcc finds the file “libm-2.19.so” and its corresponding symbolic link “libm.so.6” in “/lib/x86_64-linux-gnu” directory. Note that the actual filenames have a prefix “lib” before the library name “m.”

```
$ gcc solver.c -lm
```

Also, you had better pay extra care about the location of –l options. Gcc expects the caller of the library goes before the callee library, thus dragging the –lm option to the left-hand side of solver.c will result in a failure.

```
$ gcc -lm solver.c
/tmp/ccvOondC.o: In function `main':
solver.c:(.text+0xae): undefined reference to `sqrt'
solver.c:(.text+0xdf): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

3. Making a static library

As an example, let us use the following source codes.

```
#include <stdio.h>
#include <stdlib.h>
#include <arithmetic.h>
int main(int argc, char *argv[])
{
   int a = atoi(argv[1]);
   int b = atoi(argv[2]);

   printf("%d + %d = %d\n", a, b, add(a, b));
   printf("%d - %d = %d\n", a, b, sub(a, b));
   printf("%d * %d = %d\n", a, b, mul(a, b));
   printf("%d / %d = %d\n", a, b, dur(a, b));

   return 0;
}
```

cal_main.c

```
#ifndef __ARITHMETIC_H__
#define __ARITHMETIC_H__

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int dur(int a, int b);

#endif
```

arithmetic.h

```
#include <arithmetic.h>

int add(int a, int b)
{
    return a+b;
}
```

add.c

```
#include <arithmetic.h>

int sub(int a, int b)
{
    return a-b;
}
```

sub.c

```
#include <arithmetic.h>

int mul(int a, int b)
{
    return a*b;
}
```

mul.c

```
#include <arithmetic.h>

int dur(int a, int b)
{
    return a/b;
}
```

dur.c

The following is a normal way of building the above example program:

```
$ gcc –I. –c cal_main.c
$ gcc –I. –c add.c
$ gcc –I. –c sub.c
$ gcc –I. –c mul.c
$ gcc –I. –c dur.c
$ gcc –o prog cal_main.o add.o sub.o mul.o dur.o
```

Let's assume that your friend finds that the add, sub, mul, and dur functions are also useful in his/her project. In such case, one way is to give all the source codes to your friend. However, you want to hide the secrets of your functions since they are your precious intellectual properties, then you can give away the object files (i.e., add.o, sub.o, mul.o, and dur.o) alongside with arithmetic.h, keeping your secret source codes. Unfortunately, after receiving the object files, your friend complains that there are too many object files. Now let's consolidate the four object files into a single archival library file for the ease of delivery.

Using the ar rcs command as below, the four object files are simply archived into the libarithmetic.a file, whose postfix "a" represents archive in this case. Also the command ar t simply shows the contents of the given archive file.

```
$ ar rcs libarithmetic.a add.o sub.o mul.o dur.o

$ ls -l libarithmetic.a
-rw-rw-r-- 1 auto auto 5256 Sep 30 00:08 libarithmetic.a

$ ar t libarithmetic.a
add.o
sub.o
mul.o
div.o
```

Then you can deliver a single archive file instead of a bunch of object files. Your friend can now use your library file in two ways. The first method is handling the archive file just like it is an object file.

```
$ gcc –o prog –I. cal_main.c libarithmetic.a
```

The second way is using the –L and –l option together. –L tells the location of library file and –l tells the name of the library file. Thus the below options make gcc search the current directory (.) and link the library file libarithmetic.a into the executable. Please note the library file's prefix "lib" and postfix ".a."

```
$ gcc –o prog –I. cal_main.c –L. -larithmetic
```

4. Making a shared library

Let's try another way of making and using a library. By investigating the structure of the static library mechanism, we can find several limitations as follows:

- If there are plenty of executable files based on a same static library, there might be too many duplications of the static library eating up the capacity of your hard disks.
- If you find a bug in your library, all the related executable files should be re-linked with the new static library file. This would be a huge cost.

Shared library is invented to solve the above problems. Unlike the static libraries which increase the executable's size since it becomes a part of the executable, shared libraries do not increase the size of the executable file. Instead, using shared libraries, gcc just lists the library's name in the executable without adding any code. At a later time when the program actually starts, the listed libraries are dynamically linked to the executable.

To make a shared library, you have to use a special option –fPIC when compiling each source file. This option makes gcc generate a position independent code, which is necessary because the shared library does not have a fixed location when loaded into an executable's address space. Thus the generated code should not have any absolute address value inside it, which is usual for normal compile options.

```
$ gcc -fPIC -I. -c add.c
$ gcc -fPIC -I. -c sub.c
$ gcc -fPIC -I. -c mul.c
$ gcc -fPIC -I. -c dur.c
```

Then you can generate a shared library file libarithmetic.so using the –shared option. Unlike static library files with the .a postfix, shared libraries have the .so postfix, which means shared objects. Note that the shared library file is a little larger than the static library file for the same object files. This is because usually a position independent code is generated longer than its position dependent counterpart.

```
$ gcc –shared –o libarithmetic.so add.o sub.o mul.o dur.o

$ ls -l libarithmetic.so
-rwxrwxr-x 1 auto auto 8019 Sep 30 00:47 libarithmetic.so
```

When building the executable, we use the same options as the static library case.

```
$ gcc –o prog_shared –I. cal_main.c –L. -larithmetic
```

Now try to run the new executable file with the shared library. However, it fails loading the shared library.

```
$ ./prog_shared 1 2
./prog_shared: error while loading shared libraries:
libarithmetic.so: cannot open shared object file: No such file
or directory
```

What is missing is to let the system know where it can find the shared library file. To do so, use the LD_LIBRARY_PATH environment variable, which is a colon-seperated list of directories where the

system searches to find the necessary shared library files.

```
$ export LD_LIBRARY_PATH=/home/auto/makefile:$LD_LIBRARY_PATH
$ echo $LD_LIBRARY_PATH
/home/auto/makefile:
$ ./prog_shared 1 2
1 + 2 = 3
1 - 2 = -1
1 * 2 = 2
1 / 2 = 0
```

Now try to modify the shared library then see the modification is immediately applied even though you do not compile the executable file. This is because the shared libraries are dynamically linked when the executable starts.

There are other ways of letting the system know about the shared library's location. I'll skip the detail here. Interested readers are referred to [1], which contains more formal description of shared libraries.

[1] Program Library HOWTO. http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html