## Creating Libraries – Class Notes

Executables can be categorized into two:

- **Static Executables**: They contain fully resolved library functions that are physically linked into executable image during built.

- **Dynamic Executable**: They contain symbolic references to library functions used and these references are fully resolved either during application load time or run time.

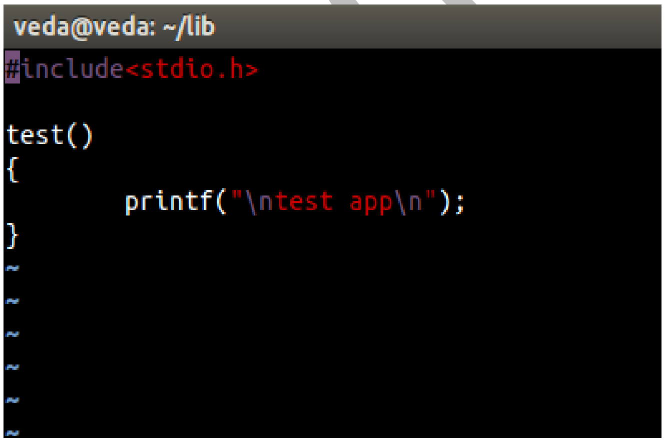
There are mainly two types of libraries:

- Static Libraries **(.a)**
- Dynamically linked libraries **(.so)**

**Steps for creating Static Libraries:**

1) Implement library sources
   eg.:

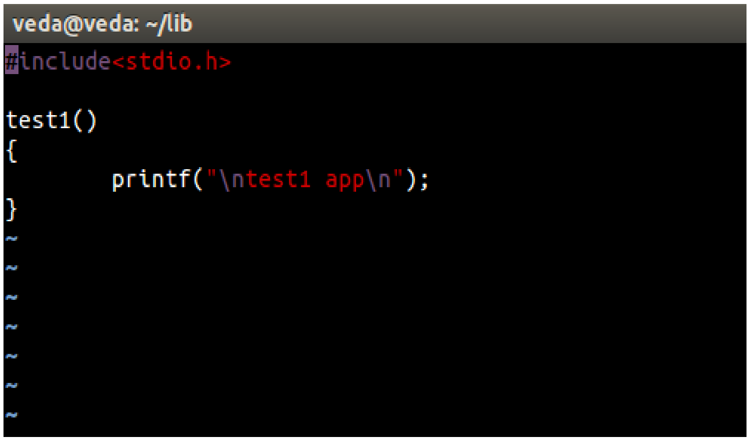
**one.c**



```
veda@veda: ~/lib
#include<stdio.h>

test()
{
        printf("\ntest app\n");
}
~
~
~
~
~
~
```

**two.c**



```
veda@veda: ~/lib
#include<stdio.h>

test1()
{
        printf("\ntest1 app\n");
}
~
~
~
~
~
~
~
```

2) Compile sources into relocatables

```
$gcc  -c  one.c  -o one.o
$gcc  -c  two.c  -o  two.o
```

3) Using Unix static library tool archive [**ar**] create static library

```
$ar  rcs  libtest.a  one.o  two.o
```

- To check the number of files in libtest.a and libtest.so, we uses the commands

  ```
  $ar  -t  libtest.a
  ```

  - In our case it will show one.o and two.o.

  ```
  $nm  -s  libtest.a
  ```

  - It will list all object files also all the functions inside the object file.

Thus created a static library **libtest.a** and in order to check this library let's write **a C** program **test.c** which calls the two functions **test()** and **test1()** in **one.c** and **two.c.**

```
$vim test.c
```

```
#include<stdio.h>
int main()
{
test();
test1();
}
```

Then we need to compile **test.c** with the static library **libtest.a** . Generally linker checks the default path only, but our library is in our working directory therefore while compiling we have to specify the path of our library in order to avoid undefined reference error.

```
$gcc  test.c  -o  teststat  ./libtest.a
```

This will create an executable **teststat** which is statically linked

**Steps for creating Dynamic libraries:**

Generally relocatables are of two types

- **Position dependent:** Relocatables which cannot be shared.
- **Position independent:** Relocatables which can be shared.

- First create two **.c** files **one.c** and **two.c** as above.
- Compile the sources into position independent relocatables

```
$gcc  -c  -fpic  one.c  -o  one.o

$gcc  -c  -fpic  two.c  -o  two.o
```

**-fpic** is a flag which is used to tell the linker as it is position independent.

Using dynamic linker creating shared library

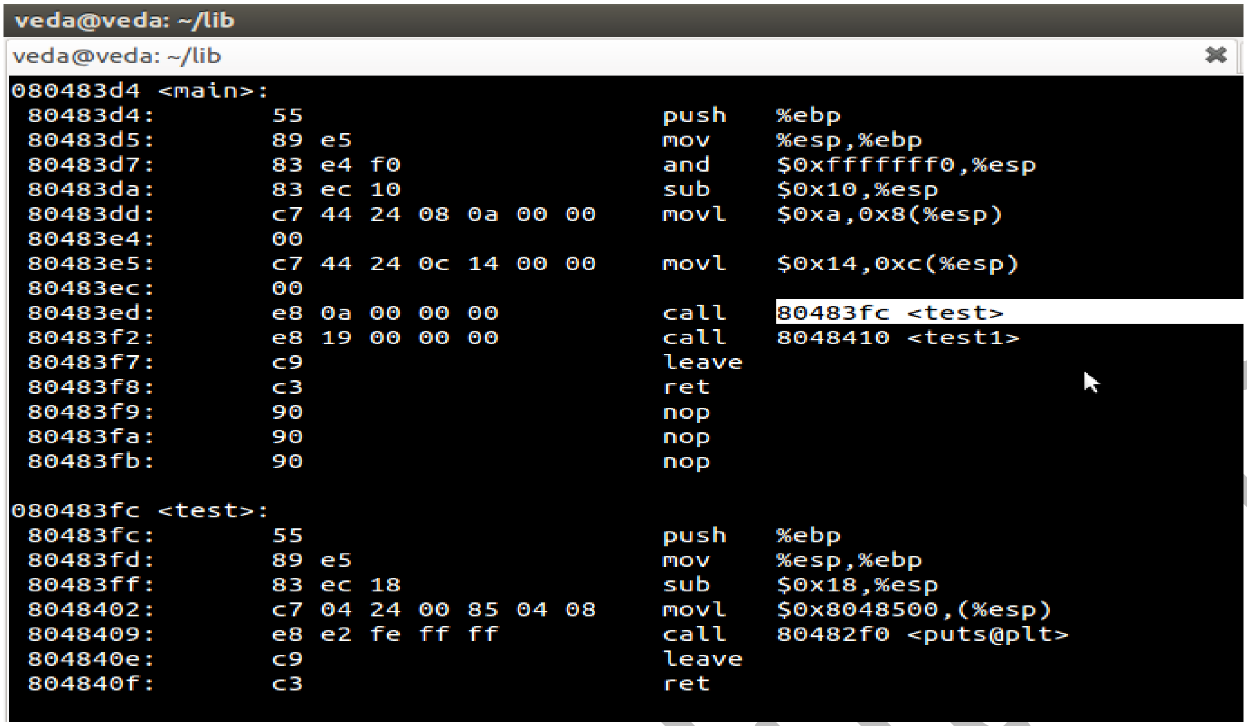
```
$gcc  -shared  -o  libtest.so  one.o  two.o
```

- Then compile the above **test.c** using dynamic library **libtest.so**.

```
$gcc  test.c  -o  testdyn  ./libtest.so
```

In both the cases output will be same. In order to understand the major differences between static and dynamic linking, we will analyze the executables using objdump tool.

```
$ objdump –D  teststat      [static executable]
```

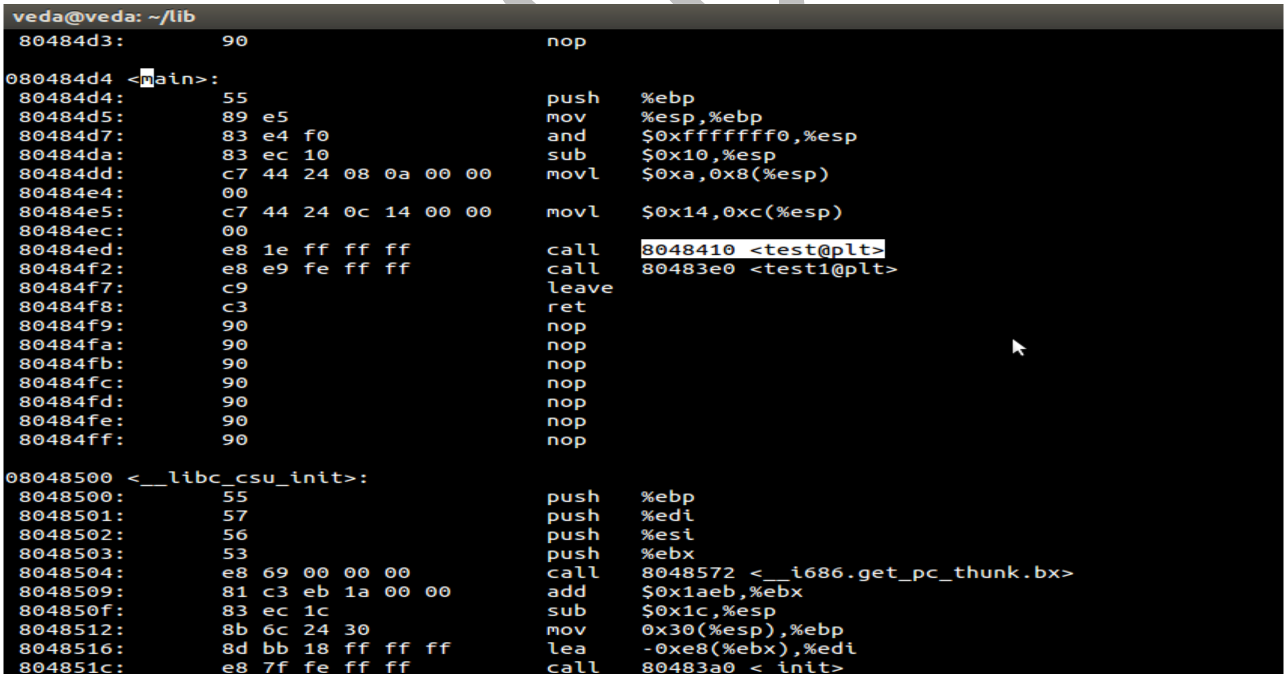


```
080483d4 <main>:
 80483d4:       55                      push   %ebp
 80483d5:       89 e5                   mov    %esp,%ebp
 80483d7:       83 e4 f0                and    $0xfffffff0,%esp
 80483da:       83 ec 10                sub    $0x10,%esp
 80483dd:       c7 44 24 08 0a 00 00    movl   $0xa,0x8(%esp)
 80483e4:       00
 80483e5:       c7 44 24 0c 14 00 00    movl   $0x14,0xc(%esp)
 80483ec:       00
 80483ed:       e8 0a 00 00 00          call   80483fc <test>
 80483f2:       e8 19 00 00 00          call   8048410 <test1>
 80483f7:       c9                      leave
 80483f8:       c3                      ret
 80483f9:       90                      nop
 80483fa:       90                      nop
 80483fb:       90                      nop

080483fc <test>:
 80483fc:       55                      push   %ebp
 80483fd:       89 e5                   mov    %esp,%ebp
 80483ff:       83 ec 18                sub    $0x18,%esp
 8048402:       c7 04 24 00 85 04 08    movl   $0x8048500,(%esp)
 8048409:       e8 e2 fe ff ff          call   80482f0 <puts@plt>
 804840e:       c9                      leave
 804840f:       c3                      ret
```

If we analyze the main function in objdump file we could see that test function base address is given and it is called directly.

$ **objdump –D testdyn** [dynamic executable]

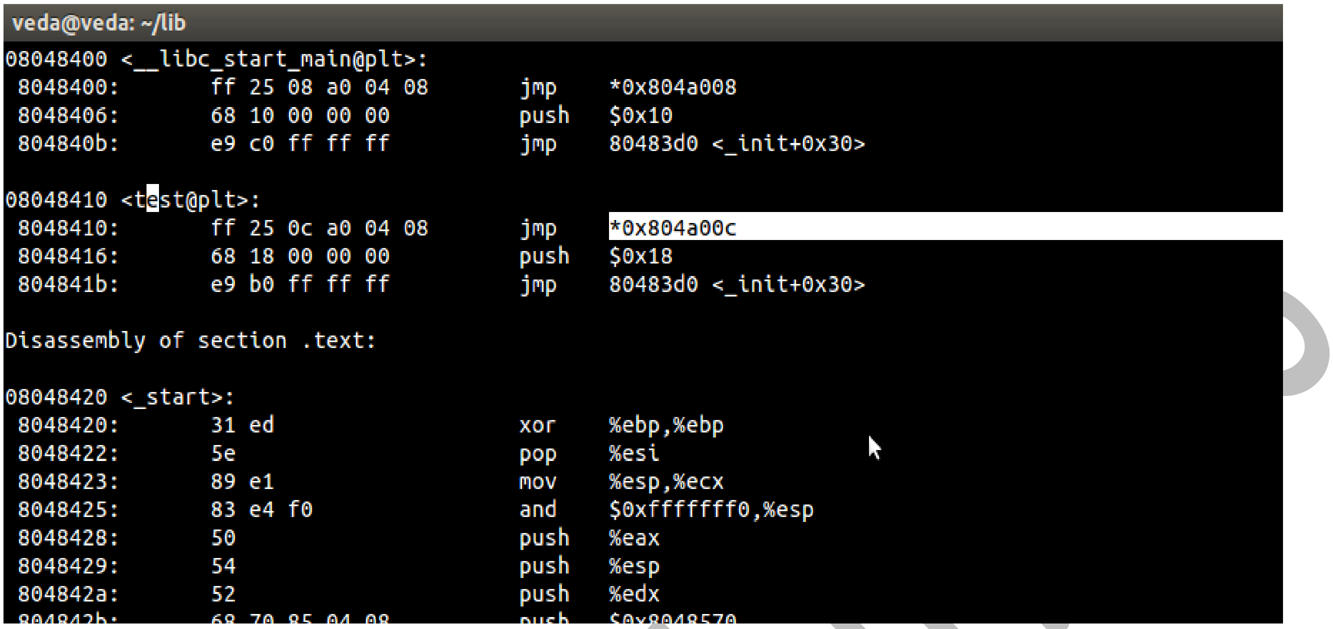


```
 80484d3:       90                      nop

080484d4 <main>:
 80484d4:       55                      push   %ebp
 80484d5:       89 e5                   mov    %esp,%ebp
 80484d7:       83 e4 f0                and    $0xfffffff0,%esp
 80484da:       83 ec 10                sub    $0x10,%esp
 80484dd:       c7 44 24 08 0a 00 00    movl   $0xa,0x8(%esp)
 80484e4:       00
 80484e5:       c7 44 24 0c 14 00 00    movl   $0x14,0xc(%esp)
 80484ec:       00
 80484ed:       e8 1e ff ff ff          call   8048410 <test@plt>
 80484f2:       e8 e9 fe ff ff          call   80483e0 <test1@plt>
 80484f7:       c9                      leave
 80484f8:       c3                      ret
 80484f9:       90                      nop
 80484fa:       90                      nop
 80484fb:       90                      nop
 80484fc:       90                      nop
 80484fd:       90                      nop
 80484fe:       90                      nop
 80484ff:       90                      nop

08048500 <__libc_csu_init>:
 8048500:       55                      push   %ebp
 8048501:       57                      push   %edi
 8048502:       56                      push   %esi
 8048503:       53                      push   %ebx
 8048504:       e8 69 00 00 00          call   8048572 <__i686.get_pc_thunk.bx>
 8048509:       81 c3 eb 1a 00 00       add    $0x1aeb,%ebx
 804850f:       83 ec 1c                sub    $0x1c,%esp
 8048512:       8b 6c 24 30             mov    0x30(%esp),%ebp
 8048516:       8d bb 18 ff ff ff       lea    -0xe8(%ebx),%edi
 804851c:       e8 7f fe ff ff          call   80483a0 <_init>
```

Here in dynamic executables main function, it is calling test@plt , (plt= procedure linkage table). Plt table is generated by linker and contains information about dynamic linking.



```
08048400 <__libc_start_main@plt>:
 8048400:       ff 25 08 a0 04 08       jmp    *0x804a008
 8048406:       68 10 00 00 00          push   $0x10
 804840b:       e9 c0 ff ff ff          jmp    80483d0 <_init+0x30>

08048410 <test@plt>:
 8048410:       ff 25 0c a0 04 08       jmp    *0x804a00c
 8048416:       68 18 00 00 00          push   $0x18
 804841b:       e9 b0 ff ff ff          jmp    80483d0 <_init+0x30>

Disassembly of section .text:

08048420 <_start>:
 8048420:       31 ed                   xor    %ebp,%ebp
 8048422:       5e                      pop    %esi
 8048423:       89 e1                   mov    %esp,%ecx
 8048425:       83 e4 f0                and    $0xfffffff0,%esp
 8048428:       50                      push   %eax
 8048429:       54                      push   %esp
 804842a:       52                      push   %edx
 804842b:       68 70 85 04 08          push   $0x8048570
```

Above figure shows the plt table. In that table we could find a **function pointer**, which derefers to our particular function **test**.

## USE CASES

- Static executables occupies more disk space but it has zero initialization time.
- Dynamic executables consumes less disk space but it consumes n amount of cpu cycles for initialization.
- Static builds are preferred if executables are being built for a specific use and will be used in a resource constrained environment where initialization delays are not tolerable.
- Dynamic builds are preferred for easier customization, maintenance and extensions for an application.