G++/GCC Command

```
g++ -g --std=c++0x –fomit–frame–pointer
        # x86-64 : –fomit–frame–pointer  it doesn’t prevent debugging.
        # -g: debug info
gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c#create PIC object files
gcc -g -shared -o libfoo.so mod1.o mod2.o mod3.o  #create shared library
gcc -g -fPIC -Wall mod1.c mod2.c mod3.c -shared -o libfoo.so  #create & compile share library
gcc -g -shared -Wl,-soname,libbar.so -o libfoo.so mod1.o mod2.o mod3.o
        #embedded SONAME for libfoo.so

gcc -g -Wall -Wl,-rpath,/home/mtk/pdir -o prog prog.c libdemo.so
        # –rpath : insert library path into the ELF for runtime lookup

gcc -g -Wall -o prog prog.c -Wl,--enable-new-dtags -Wl,-rpath,/home/mtk/pdir/d1 \
-L/home/mtk/pdir/d1 -lx1  #enable DT_RUNPATH in ELF

gcc -Wl,-rpath,'$ORIGIN'/lib
        #to locate runtime library based on the application location

gcc –Wl,–znodelete # load and never delete, like dlopen's flag = RTLD_NODELETE

gcc -g -shared -Wl,-Bsymbolic -o libfoo.so foo.o
        #To let the symbol reference to the symbol in the same library, use -Bsymbolic
        # like dlopen's flag = RTLD_DEEPBIND

gcc -Wl,--export-dynamic main.c
gcc -export-dynamic main.c
        #Let the library use symbols in the main program:

gcc -Wl,--version-script,myscriptfile.map …

gcc -g -c -fPIC -Wall sv_lib_v2.c
gcc -g -shared -o libsv.so sv_lib_v2.o -Wl,--version-script,sv_v2.map
```

AR Command

```
ar r libdemo.a mod1.o mod2.o mod3.o        #update
ar tv libdemo.a          #list table of content
ar d libdemo.a mod3.o          #delete file
```

NM Command

```
nm #list symbols of object files
nm mod1.o | grep _GLOBAL_OFFSET_TABLE_ #check if compiled with –fPIC option

nm -A /usr/lib/lib*.so 2> /dev/null | grep ' crypt$'
```

–A option to nm specifies that the library name should be listed at the start of each line displaying a symbol.

readelf Command

```
readelf #displays information about ELF files.
readelf -s mod1.o | grep _GLOBAL_OFFSET_TABLE_  # check if it’s dynamic link file, which
                                                  has GOT
readelf -d libfoo.so | grep TEXTREL
```

The string TEXTREL indicates the presence of an object module whose text segment contains a reference that requires run-time relocation.

```
readelf -d libfoo.so | grep SONAME

readelf —dynamic  #(or, equivalently, readelf –d) command.

readelf --syms --use-dynamic vis.so | grep vis_  #read version script name
```

objdump Command

```
objdump         #display information from object files

objdump --all-headers libfoo.so | grep TEXTREL
readelf -d libfoo.so | grep TEXTREL

objdump -p libfoo.so | grep SONAME
readelf -d libfoo.so | grep SONAME

objdump -p prog | grep PATH # to check rpath in the obj file
readelf —dynamic  #(or, equivalently, readelf –d) command.

objdump -t p1 # display the symbal tables
```

ldconfig

```
$ /sbin/ldconfig -nv . #create soname link
.:
libdemo.so.1 -> libdemo.so.1.0.1
```

ELF file format

DT_NEEDED #library name needed
DT_SONAME # so name injected during the compile time
--
If a shared library has a soname, then, during static linking, the soname is embedded in the executable file instead of the real name, and subsequently used by the dynamic linker when searching for the library at run time.

--
Embedding the name of the library inside the executable happens automatically when we link our program with a shared library.
--
The
/lib/ld-linux.so.2 #dynamic linker
need to know the location of the .so files.

The dynamic linker(**runtime**) looks in these places to find the library: (From book Linkers and Loaders)

- If the dynamic segment contains an entry called **DT_RPATH**, it's a colon-separated list of directories to search for libraries. This entry is added by a command line switch or environment variable to the regular (not dynamic) linker at the time a program is linked. It's mostly used for subsystems like databases that load a collection of programs and supporting libraries into a single directory.
- If there's an environment symbol **LD_LIBRARY_PATH**, it's treated as a colon-separated list of directories in which the linker looks for the library. This lets a developer build a new version of a library, put it in the LD_LIBRARY_PATH and use it with existing linked programs either to test the new library, or equally well to instrument the behavior of the program. (It skips this step if the program is set-uid, for security reasons.)
- The linker looks in the library cache file **/etc/ld.so.conf** which contains a list of library names and paths. If the library name is present, it uses the corresponding path. This is the usual way that most libraries are found. (The file name at the end of the path need not be exactly the same as the library name, see the section on library versions, below.)
- If all else fails, it looks in the **default directory /usr/lib**, and if the library's still not found, displays an error message and exits.

**** LIBRARY_PATH is for linking time to search for the library, like -L option for linker!**

Static linking is sometimes also referred to as link editing, and a static linker such as ld is sometimes referred to as a link editor
----

DYNAMIC Section:

NEEDED: the name of a library this file needs. (Always in programs, sometimes in libraries when one library is dependend on another, can occur more than once.)

SONAME: "shared object name", the name of the file the linker uses. (Libraries.)

SYMTAB, STRTAB, HASH, SYMENT, STRSZ,: point to the symbol table, associated string and hash tables, size of a symbol table entry, size of string table. (Both.)

PLTGOT: points to the GOT, or on some architectures to the PLT (Both.)

REL, RELSZ, and RELENT or RELA, RELASZ, and RELAENT: pointer to, number of, and size of relocation entries. REL entries don't contain addends, RELA entries do. (Both.)

JMPREL, PLTRELSZ, and PLTREL: pointer to, size, and format (REL or RELA) of relocation table for data referred to by the PLT. (Both.)

I **NIT and FINI**: pointer to initializer and finalizer routines to be called at program startup and finish. (Optional but usual in both.)

Generate share library

Generate soname inside the library:

```
gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
gcc -g -shared -Wl,-soname,libbar.so -o libfoo.so mod1.o mod2.o mod3.o
```

Generate soname with link:

```
ln -s libfoo.so libbar.so
```

Because any program that link with libfoo.so during the runtime will look for soname : libbar.so

Determine the soname in the library:

```
objdump -p libfoo.so | grep SONAME
readelf -d libfoo.so | grep SONAME
```

Check while library the process is using now:

/proc/PID/maps

Share Library Naming Convention

Real names, sonames, and linker names:

real name : libname.so.major-id.minor-id

soname : ibname.so.major-id #link to real name, maintain by ldconfig

linker name : libname.so #link to soname

Static Link

Specify the pathname of the static library (including the .a extension) on the gcc command line.

- Specify the –static option to gcc.
- Use the gcc options **–Wl,–Bstatic** and **–Wl,–Bdynamic** to explicitly toggle the linker’s choice between static and shared libraries. These options can be intermingled with –l options on the gcc command line. The linker processes the options *in the order* in which they are specified.

Runtime Library Lookup inside the ELF file

```
gcc -g -Wall -Wl,-rpath,/home/mtk/pdir -o prog prog.c libdemo.so
        –rpath : insert library path into the ELF for runtime lookup
```

environment variable:

```
        LD_RUN_PATH # used only if -rpath is NOT specified.
```

**** -rpath is runtime library location. -L is link time library location.**

```
objdump -p prog | grep PATH # to check rpath in the obj file
readelf —dynamic  #(or, equivalently, readelf –d) command.
```

At run time, search for library: *(precedence)*

DT_RPATH	//OLD in ELF
LD_LIBRARY_PATH	//env variable
DT_RUNPATH	//NEW in ELF

```
gcc -Wl,-rpath,'$ORIGIN'/lib
        # to locate runtime library based on the application location
```

**** If the executable is a set-user-ID or set-group-ID program, then LD_LIBRARY_PATH is ignored.**

```
gcc –Wl,–znodelete # load and never delete, like dlopen's flag = RTLD_NODELETE
```

Symbol Lookup Rule

1. A definition of a global symbol in the main program overrides a definition in a library.

2. If a global symbol is defined in multiple libraries, then a reference to that symbol is bound to the **first definition found by scanning libraries in the left-to-right** order in which they were listed on the static link command line.

```
gcc -g -shared -Wl,-Bsymbolic -o libfoo.so foo.o
        #To let the symbol reference to the symbol in the same library, use -Bsymbolic
        # like dlopen's flag = RTLD_DEEPBIND
```

Dynamic Load during the running process

dlopen API on Linux, we must specify the **–ldl**

void *dlopen(const char *libfilename, int flags); **#this function call has reference count**

flags:

RTLD_LAZY

RTLD_NOW

environment variable:

LD_BIND_NOW # set this overrides dlopen's flag RTLD_NOW

RTLD_GLOBAL

Symbols in this library and its dependency tree are made available for resolving references in other libraries loaded by this process and also for lookups via dlsym().

RTLD_LOCAL (Linux default)

This is the converse of RTLD_GLOBAL and the default if neither constant is specified. It specifies that symbols in this library and its dependency tree are not available to resolve references in subsequently loaded libraries.

RTLD_NODELETE (since glibc 2.2)

Don’t unload the library during a dlclose(), even if the reference count falls to 0. This means that the library’s static variables are not reinitialized if the library is later reloaded by dlopen(). (We can achieve a similar effect for libraries loaded automatically by the dynamic linker by specifying the gcc –Wl,**–znodelete** option when creating the library.)

RTLD_NOLOAD (since glibc 2.2)

Don’t load the library. This serves two purposes. First, we can use this flag to check if a particular library is currently loaded as part of the process's address space. If it is, dlopen() returns the library's handle; if it is not, dlopen() returns NULL. Second, we can use this flag to "promote" the flags of an already loaded library. For example, we can specify RTLD_NOLOAD | RTLD_GLOBAL in flags when using dlopen() on a library previously opened with RTLD_LOCAL.

RTLD_DEEPBIND (since glibc 2.3.4)

When resolving symbol references made by this library, search for definitions in the library before searching for definitions in libraries that have already been loaded. This allows a library to be self-contained, using its own symbol definitions in preference to global symbols with the same name defined in other shared libraries that have already been loaded. (This is similar to the effect of the –Bsymbolic linker option)

If we receive an error return from dlopen() or one of the other functions in the dlopen API, we can use **dlerror**() to obtain a pointer to a string that indicates the causeof the error.

```
const char *dlerror(void);
```

```
void *dlsym(void *handle, char *symbol);
```

pseudohandles

RTLD_DEFAULT

Search for symbol starting with the main program, and then proceeding in order through the list of all shared libraries loaded, including those libraries dynamically loaded by dlopen() with the RTLD_GLOBAL flag. This corresponds to the default search model employed by the dynamic linker.

RTLD_NEXT

Search for symbol in shared libraries loaded after the one invoking dlsym(). This is useful when creating a wrapper function with the same name as a function defined elsewhere. For example, in our main program, we may define our own version of malloc() (which perhaps does some bookkeeping of memory allocation), and this function can invoke the real malloc() by first obtaining its address via the call func = dlsym(RTLD_NEXT, "malloc").

To let the dynamic loaded library to call the function inside main program instead of the function inside the library itself.

```
gcc -Wl,--export-dynamic main.c (plus further options and arguments)
```

or

```
gcc -export-dynamic main.c
        #Let the library use symbols in the main program:
```

The following techniques can be used to control the export of symbols:

- In a C program, we can use the static keyword to make a symbol private to a source-code module, thus rendering it unavailable for binding by other object files. (Anonymous namespace in C++)

> As well as making a symbol private to a source-code module, the static keyword also has a converse effect. If a symbol is marked as static, then all references to the symbol in the same source file will be bound to that definition of the symbol. Consequently, these references won't be subject to run-time interposition by definitions from other shared libraries. This effect of the static keyword is similar to the **–Bsymbolic** linker option described in Section 41.12, with the difference that the static keyword affects a single symbol within a single source file.

- Version scripts can be used to precisely control symbol visibility and to select the version of a symbol to which a reference is bound.
- When dynamically loading a shared library ,the dlopen() RTLD_GLOBAL flag can be used to specify that the symbols defined by the library should be made available for binding by subsequently loaded libraries, and the —–export–dynamic linker option can be used to make the global symbols of the main program available to dynamically loaded libraries.

Linker Version Scripts

```
gcc -Wl,--version-script,vis.map ...
```

```
readelf --syms --use-dynamic vis.so | grep vis_
```

Version script format:

```
VER_1 {
global:
vis_f1;
vis_f2;
local:
*;
};
```

Sample Code:

sv_lib_v2.c:

```
#include <stdio.h>
__asm__(".symver xyz_old,xyz@VER_1");
__asm__(".symver xyz_new,xyz@@VER_2");  //@@ means default when statically linked against this shared library.
void xyz_old(void) { printf("v1 xyz\n"); }
void xyz_new(void) { printf("v2 xyz\n"); }
void pqr(void) { printf("v2 pqr\n"); }
```

sv_v2.map:

```
VER_1 {
global: xyz;
local: *; # Hide all other symbols
};
VER_2 {
global: pqr;
} VER_1;
```

```
gcc -g -c -fPIC -Wall sv_lib_v2.c
gcc -g -shared -o libsv.so sv_lib_v2.o -Wl,--version-script,sv_v2.map
```

```
objdump -t p1 # display the symbol tables
```

Initialization and Finalization Functions for Library

```
void __attribute__ ((constructor)) some_name_load(void)
{
/* Initialization code */
}

void __attribute__ ((destructor)) some_name_unload(void)
{
```

/* Finalization code */
}

Link Env Variable

LD_PRELOAD=libalt.so ./program
The LD_PRELOAD environment variable controls preloading on a **per-process** basis.

/etc/ld.so.preload #system wide basis.

LD_PRELOAD >> /etc/ld.so.preload (precedence)
set-user-ID and set-group-ID programs ignore LD_PRELOAD.
LD_DEBUG

Dynamic ELF GOT (GLOBAL_OFFSET_TABLE)

R_386_GOT32: The relative location of the slot in the GOT
where the linker has placed a pointer to the given symbol. Used
for indirectly referenced global data. 即GOT頭到此SLOT的offset 此slot存放symbol的位置

R_386_GOTOFF: The distance from the base of the GOT to the
given symbol or address. Used to address static data relative to the
GOT. 即GOT根部到symbol的距離

R_386_RELATIVE: Used to mark data addresses in a PIC shared
library that need to be relocated at load time. share library內symbol的位置 需被relocate at load time

uselib()

    #system call. uselib() is Linux-specific, and should not be used in programs intended to be portable.

Procedure Linkage Table (PLT)

References:
[Linkers and Loaders]
[Linking libstdc++ statically]
[The GCC low-level runtime library]
[C++ ABI Summary]
[Linux static linking is dead?]
[Statically-linking libstdc++ on AIX]

Telling gcc directly to link a library statically
Program Library HOWTO
Working with libraries and the linker