Contents

[Static vs Dynamic Libraries 1](#_Toc499128825)

[Memory Layout of C Programs 29](#_Toc499128826)

[C++ program's major distinct memory areas 34](#_Toc499128827)

[Namespace in C++ 35](#_Toc499128828)

[**Can namespaces be nested in C++?** 40](#_Toc499128829)

[Where are the local, global, static, auto, register, extern, const, volatile variables are stored? 41](#_Toc499128830)

[Internal and External Linkage in C++ 41](#_Toc499128831)

[TL;DR 41](#_Toc499128832)

[Table of Contents 42](#_Toc499128833)

[Basics 42](#_Toc499128834)

[Declaration vs. Definition 42](#_Toc499128835)

[Translation Units 44](#_Toc499128836)

[Linkage 45](#_Toc499128837)

[External Linkage 45](#_Toc499128838)

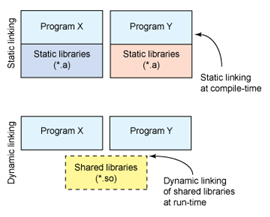
[Internal Linkage 48](#_Toc499128839)

# Static vs Dynamic Libraries

Use dynamic libraries instead of static libraries whenever possible!  
<http://www.bogotobogo.com/cplusplus/libraries.php>

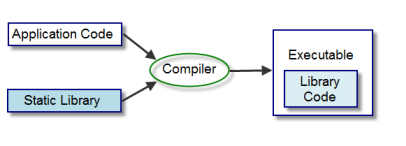
There are two types of libraries we can make. The decision on which one we take can have a significant impact on our clients' applications, such as executable size, load time, etc.

Static Libraries

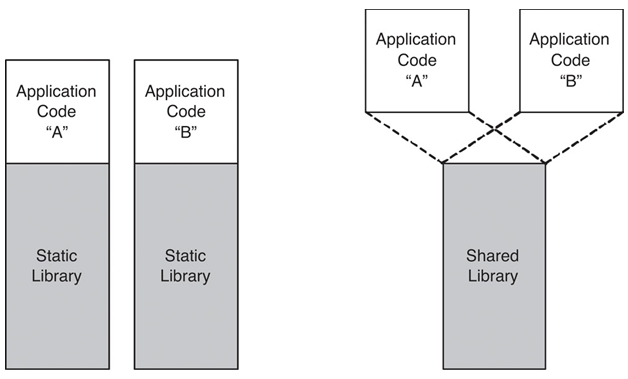


A static library contains object code linked with an end-user application, and then becomes part of that executable. A static library is sometimes called an **archive** since it is just a package of compiled object files. These libraries are in directories such as **/lib**, **/usr/lib** or **/usr/local/lib**.

After resolving the various function references from the main program to the modules in the static library, a linker extracts copies of the required object modules from the library and copies these into the resulting executable file. When linking is done during the creation of an executable, it is known as **static linking** or **early binding**. In this case, the linking is usually done by a linker, but may also be done by the compiler. A static library, also known as an archive, is intended to be statically linked. Originally, only static libraries existed. Static linking must be performed when any modules are recompiled.



All of the modules required by a program are sometimes statically linked and copied into the executable file. This process, and the resulting stand-alone file, is known as a **static build** of the program.



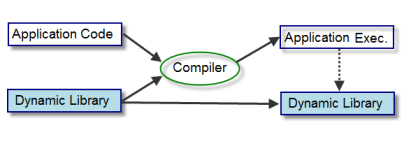
The filenames always start with **lib**, and end with **.a** (archive, static library) on Unix/Linux, and on Windows it's a little bit complicated. Depending on how they are compiled, **\*.LIB** files can be either static libraries or representations of dynamically linkable libraries needed only during compilation, known as **Import Libraries**. Unlike in the UNIX world, where different file extensions are used, when linking against **\*.LIB** file in Windows one must first know if it is a **regular static** library or an **import** library. In the latter case, a **.DLL** file must be present at run time.

Here are implications of distributing our implementation as a static library:

1. A static library is only needed to link an application. It is not needed to run that application because the library code is already embedded inside the application. So, our clients can distribute their applications without any additional run-time dependencies.
2. If our clients want to link our library into multiple executables, each one will embed a copy of our code. If our library is 100MB in size, and our client wishes to link this into three separate programs, then we could be adding up to 300MB to the total size of their product. Notice that **only the object files in the static library that are actually used are copied to the application**. Thus, in reality, the total size of each application could be less than this worst case.
3. Our clients can distribute their applications without any concerns that it will find an incompatible library on the end-user's side or a completely different library with the same name from another vendor.
4. But if our clients want to be able to hot patch their application, in other words, they want to update the version of our library used by their application, they must replace the entire executable to achieve this. If this is done as an internet-based update, the end user may have to download a much larger update and hence wait longer for the update to complete.

Dynamic Libraries

A **shared library** or **shared object** is a file that is intended to be shared by executable files and further shared objects files. Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a single monolithic executable file for the program.



In other words, dynamic libraries are files linked against at compile time to resolve undefined references and then distributed with the end-user application so that the application can load the library code at run time. Usually, this requires use of a dynamic linker on the end user's machine to determine and load all dynamic library dependencies at run time, perform the necessary symbol relocations, and then pass control to the application.

The Linux dynamic linker is called **ld.so** and on the Mac it is called **dyld**. The dynamic linker supports a number of environment variables to modify or debug its behavior.

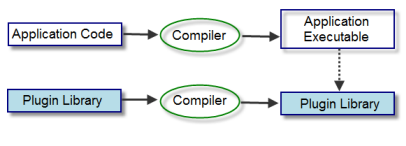
As mentioned earlier, dynamic libraries are sometimes called shared libraries because they can be shared by multiple programs. On Unix-like machine, they have a **.so** (shared object, dynamically linked library) file extension. On Windows, dynamically linkable libraries usually have the suffix **\*.DLL**, although other file name extensions may be used for specific purpose dynamically-linked libraries, e.g. **\*.OCX** for OLE libraries.

Here are implications of using dynamic libraries to distribute our program:

1. Our clients must distribute our dynamic library with their application as well as any dynamic libraries that our library depends on, so that it can be discovered when the application is run.
2. Our clients' applications will nor run if the dynamic library cannot be found. Also, the application may not run if the dynamic library is upgraded.
3. Using dynamic libraries can often more efficient than static libraries in terms of disk space if more than one application needs to use the library. This is because the library code is stored in a single shared file and not duplicated inside each executable. However, even with static library, the executable only needs to include the object code from the static library that is actually used. So, if each application uses only a small fraction of the total static library, the disk space efficiency can still rival that of a single complete dynamic library.
4. Dynamic libraries may also be more efficient in terms of memory. Most modern OS will attempt to only load the dynamic library code into memory once, and share it across all applications that depend upon it. This may also lead to better cache utilization. By comparison, every application that is linked against a static library will load duplicate copies of the library code into memory.
5. If our clients want to hot patch their application with a new (backward compatible) version of our shared library, they can simply drop in the replacement library file and all of their applications will use this new library without having to recompile or relink.

Plugins

Plug-ins enable customizing the functionality of an application. For example, plug-ins are commonly used in web browsers to play video, scan for viruses, and display new file types. Well-known plug-ins examples include Adobe Flash Player, QuickTime, and Microsoft Silverlight.



Dynamic libraries are usually linked against an application and then distributed with that application so that the OS can load the library when the application is launched. However, it is also possible for an application to load a dynamic library on demand without the application having been compiled and linked against that library.

This can be used to create plugin interface, when the application can load additional code at run time that extends the basic capabilities of the program.

This gives us the capability to create extensible code that allow our clients to drop in new functionality that our program will then load and execute.

Building Library

Here are sample codes:

// mylib.h

#ifndef MYLIB\_H

#define MYLIB\_H

double calcSqrt(double);

#endif

// calc.c

#include <math.h>

#include "mylib.h"

double calcSqrt(double d)

{

return sqrt(d);

}

// main.c

#include <stdio.h>

#include "mylib.h"

int main()

{

double d = 100;

printf("sqrt(%3.0f)=%2.0f\n",d,calcSqrt(d));

return 0;

}

Building Library - static

$ gcc -c main.c

$ gcc -c calc.c

$ gcc -o main calc.o main.o -lm or gcc -o main calc.o main.o /usr/lib/libm.a

$ ./main

sqrt(100)=10

Now, let's build a new library using **ar**.

$ ar crv libcalc.a calc.o

Now we can make executable from the library we just built, **libcalc.a**

$ gcc -o main main.o libcalc.a -lm

or

$ gcc -o main main.o -L. -lcalc -lm

$ ./main

sqrt(100)=10

The **-L.** option tells compiler that the library, **libcalc.a** can be found in the current directory.

We can list symbols in the object/library by using **nm** command:

$ nm main

080495a0 d \_DYNAMIC

08049674 d \_GLOBAL\_OFFSET\_TABLE\_

0804855c R \_IO\_stdin\_used

w \_Jv\_RegisterClasses

08049590 d \_\_CTOR\_END\_\_

0804958c d \_\_CTOR\_LIST\_\_

08049598 D \_\_DTOR\_END\_\_

08049594 d \_\_DTOR\_LIST\_\_

08048588 r \_\_FRAME\_END\_\_

0804959c d \_\_JCR\_END\_\_

0804959c d \_\_JCR\_LIST\_\_

08049694 A \_\_bss\_start

08049690 D \_\_data\_start

08048510 t \_\_do\_global\_ctors\_aux

08048380 t \_\_do\_global\_dtors\_aux

08048560 R \_\_dso\_handle

0804958c d \_\_fini\_array\_end

0804958c d \_\_fini\_array\_start

w \_\_gmon\_start\_\_

08048509 T \_\_i686.get\_pc\_thunk.bx

0804958c d \_\_init\_array\_end

0804958c d \_\_init\_array\_start

08048490 T \_\_libc\_csu\_fini

080484a0 T \_\_libc\_csu\_init

U \_\_libc\_start\_main@@GLIBC\_2.0

0804958c d \_\_preinit\_array\_end

0804958c d \_\_preinit\_array\_start

08049694 A \_edata

0804969c A \_end

08048538 T \_fini

08048558 R \_fp\_hw

080482c0 T \_init

08048330 T \_start

08048450 T calcSqrt

08048354 t call\_gmon\_start

08049698 b completed.5791

08049690 W data\_start

08049694 b dtor\_idx.5793

080483e0 t frame\_dummy

08048404 T main

U printf@@GLIBC\_2.0

U sqrt@@GLIBC\_2.0

$ nm libcalc.a

calc.o:

00000000 T calcSqrt

U sqrt

The character in the second column represents the symbol type: **T** refers to a text section symbol that is defined in the library, **U** refers to a symbol that is referenced by the library but is not defined by it. An uppercase letter specifies an external symbol, while a lowercase represents an internal symbol.

The string in the 3rd column provides the mangled symbol name. We can unmangle a symbol using **c++filt** command.

We can list global symbols using **g** option in the **nm** command:

nm -g main

Also, see [Creating Static Libraries on Linux](http://www.bogotobogo.com/cplusplus/libraries.php#linux_static)

Building Library - shared (dynamic)

A typical shared library extension is **so**. For math library, it is **/usr/lib/libm.so**.

When a program uses shared library, the code itself is not included in the program, but it just links the library to refer it at run time. In that way, several program use the library without having it, and can save space. In other words, shared libraries address the disadvantages of static libraries. A shared library is an object module that can be loaded at run time at an arbitrary memory address, and it can be linked to by a program in memory. Shared libraries often are called as shared objects. On most UNIX systems they are denoted with a **.so** suffix and Microsoft refer to them as DLLs (dynamic link libraries).

Also, shared library has addition advantage over static library because it can be updated independently.

Actually, the file, **/usr/lib/libm.so** is a symbolic link to the revision of **libm.so.6**, and it is used at compile time.

$ ls -la libm.so

lrwxrwxrwx 1 root root 19 Feb 6 2008 libm.so -> ../../lib/libm.so.6

When linux application starts, the OS checks the library version requested by the application.

In linux system, **ld.so** (dynamic linker/loader) loads the shared libraries needed by a program, prepares the program to run, and then runs it.

When we try to run a program, but it is complaining that a library is missing though we know the library is there (such as in /usr/local/lib). But most likely /usr/local/lib is not on the default place (/usr/lib) where the program looks for the lib. So, we need to let the system know where to look for. There is a file (**/etc/ld.so.conf**) on our system where all the paths to the libraries are mentioned. To do that, we should run **ldconfig**.

We can check the required shared libraries for a program to run by using **ldd**:

$ **ldd main**

linux-gate.so.1 => (0x00dbb000)

libm.so.6 => /lib/libm.so.6 (0x493c9000)

libc.so.6 => /lib/libc.so.6 (0x4a439000)

/lib/ld-linux.so.2 (0x4a417000)

This our case, the **main** program is using shared libraries for math and Standard C.

To make shared library with calc.o, we do the following:

$ gcc -shared -fPIC -o libcalc.so calc.o

The command tells the compiler driver to generate a shared library, **libcalc.so**, from the object module **calc.o**. The **-fPIC** option tells the compiler to generate **position independent code (PIC)**.

Now, suppose the primary object module is **main.o**, which has dependencies on **calc.o**. In this case, the linker is invoked with:

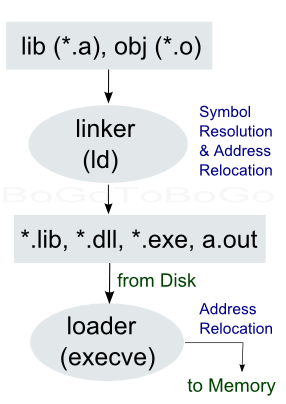
$ gcc main.o ./libcalc.so -lm

The command above makes an executable file, **a.out**, in a form that can be linked to **libcalc.so** at load time. However, **a.out** does not contain the object module **calc.o**, which would have been included had we created a static library instead of a shared library. The executable simply contains some **relocation and symbol table information** that allow references to code and data in **libcalc.so** to be resolved at run time. Thus, **a.out** here is a partially executable file that still has its dependency in **libcalc.so**.

The executable also contains a **.interp** section that has the name of the dynamic linker, which itself is a shared object on Linux systems, **ld-linux.so**. So, when the executable is loaded into memory, the loader passes control to the dynamic linker. The dynamic linker contains some start-up code that maps the shared libraries to the program's address space.

It then does the following:

1. relocates the text and data of **libcalc.so** into memory segment
2. relocates any references in **a.out** to symbols defined by **libcalc.so**



Then, the dynamic linker passes control to the application, and now the location of shared object is fixed in the memory.

Shared libraries can be loaded from applications even in the middle of their executions. An application can request a dynamic linker to load and link shared libraries, even without linking those shared libraries to the executable. Linux provides system calls, such as **dlopen** to load a shared object, **dlsym** to look up a symbol in that shared object, and **dlclose** to close the shared object.

We've already used some of the tools to explore object/executable files, here is the list of tools:

1. **ar**  
   creates static libraries.
2. **ldd**  
   lists the shared libraries on which the object binary is dependent.
3. **nm**  
   lists the symbols defined in the symbol table of an object file or a static library.
4. **objdump**  
   to display all the information in an object binary file.
5. **strings**  
   list all the printable strings in a binary file.
6. **strip**  
   deletes the symbol table information.
7. **c++filt**  
   demangle low-level names into user-level names (unix/linux command).

Linker vs Loader

Unlike some authors who prefer using combined word "link-loading, in the page, I've been using those separated from each other.

1. **Linker**   
   After compilation, the fragments (compilation unit) of a program are glued together by a linker.  
   Combines two or more objects (**relocatable object**) and supplies the information needed to allow references between them.  
   **Relocation** (loader does this as well) and the **resolution of external references** are performed by the linker.
2. **Loader**   
   Loading an **executable object** into **memory** for execution.

Note:

1. **relocatable object**  
   A relocatable object code is acceptable as input to a linker.  
   A relocatable object file has following table information: import table, relocation table, and export table.
2. **executable object**  
   Executable object code is acceptable as an input to a loader, and it can be brought into memory and run.  
   Executable object file is distinguished by the fact that it contains no references to external symbols, and it also defines a starting address for execution.

Static libraries with multiply defined symbols

Using static libraries can hide problems in case we defined the same function several times. Let's look at the following example, where **a.c** and **b.c** both defined **who\_are\_you()** function.

In the first build, we combine the two source files and make it into one archive. At link time, the linker will extract the first version of **who\_are\_you()** that it encounters. However, it will not give any warning regarding the presence of multiple definitions.

In the second build, instead of combining the two sources into one archive, the same code was linked using individual object files. Then, the linker fails to make the executable.

// a.c

#include <stdio.h>

void who\_are\_you()

{

printf("I am A\n");

}

// b.c

#include <stdio.h>

void who\_are\_you()

{

printf("I am B\n");

}

// main.c

#include <stdio.h>

void who\_are\_you();

int main()

{

printf("calling who\_are\_you()\n");

who\_are\_you();

return 0;

}

Build Case I - link against an archive already combined a.o and b.o:

$ gcc -c a.c

$ gcc -c b.c

$ ar -r mylib.a a.o b.o

$ gcc main.c mylib.a

$ ./a.out

calling who\_are\_you()

I am A

$

When we combine them into an archive and link into an executable, the linker extracts one definition out of two. In our case, it selected from **a.c**. However, the builds fails because of the multiply defined symbol if we link the object file directly and make an executable as shown in the example below:

Build Case II - individual link - a.o, b.o:

$ gcc main.c a.o b.o

b.o: In function `who\_are\_you':

b.c:(.text+0x0): multiple definition of `who\_are\_you'

a.o:a.c:(.text+0x0): first defined here

collect2: error: ld returned 1 exit status

$

We get this behavior because lots of compilers do not **cross-file checking (or optimization)**, which means either the code held in the static library does not play a role in cross-file checking or the functions are not inlined from the static library into the executable.

To inspect what's in the library, we can use **nm** on the static library:

$ nm mylib.a

a.o:

U puts

0000000000000000 T who\_are\_you

b.o:

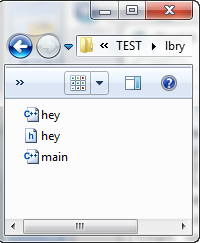
U puts

0000000000000000 T who\_are\_you

Library Porting

Porting - Windows

In this example, we will try to create a small **.dll** file that has a single function, **printHey()**, which is called from the main routine in the **main.cpp** file.



Here are the contents of the files:

// hey.h

#ifdef BUILD\_BOGO\_DLL

#define BOGO\_PRINT\_API \_\_declspec(dllexport)

#else

#define BOGO\_PRINT\_API \_\_declspec(dllimport)

#endif

extern "C" BOGO\_PRINT\_API void printHey();

// hey.cpp

#include <iostream>

#include "hey.h"

void printHey()

{

std::cout << "Hey! Welcome to bogotobogo" << std::endl;

}

// main.cpp

#include "hey.h"

int main()

{

printHey();

return 0;

}

If we want a function to be callable from a DLL on Windows, we must explicitly mark its declaration with the following:

\_\_declspec(dllexport)

For example,

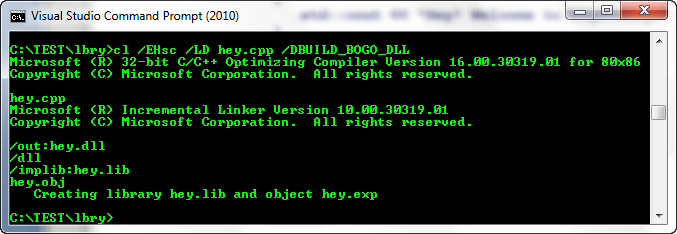
\_\_declspec(dllexport) void MyFunction();

class \_\_declspec(dllexport) MyClass;

In the opposite case, if we want to use an exported DLL function in an application, then we must prefix the function prototype with the following:

\_\_declspec(dllimport)

Therefore, it's common to employ preprocessor macros to use the export declaration when building an API but the import decoration is needed when using the same API.

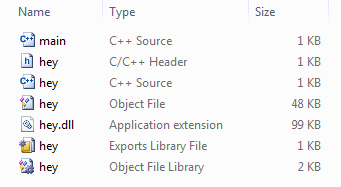
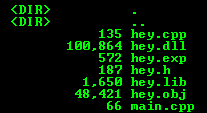


**/LD** instructs cl to create a **.dll** file. (It can be instructed to create other formats such as .exe or .obj.) **/DBUILD\_BOGO\_DLL** defines the **BOGO\_PRINT\_API** macro for this particular building process so that the **printHey symbol** is exported from this DLL. **/EHsc** will make catch clause not to catch asynchronous exceptions.

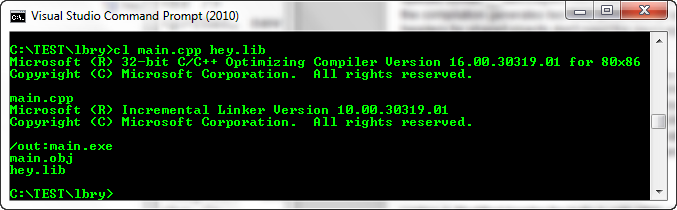
See [make](http://www.bogotobogo.com/cplusplus/make.php).

Let take a quick inspection of the sources and generated output. It reveals two important facts. As mentioned earlier, the Windows-specific syntax, **\_\_declspec(dllexport)**, is needed to export any functions, variables, or classes from a DLL. Likewise, the Windows-specific syntax, **\_\_declspec(dllimport)**, is needed to import any functions, variables, or classes from a DLL. 

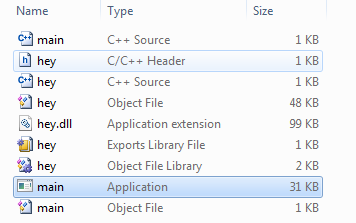
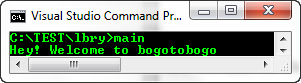
Then, new files have been generated as shown below since the compilation generates the files: **hey.dll** and **hey.lib**. The **hey.lib** is used to link the main sources.

To compile and link the main code, use the following command line:



Now, we have application called **main.exe**:

As a quick summary, we can use a Visual C++ project to create one of three variants (single or multi-threaded) of a project:

1. Dynamic-link library (DLL or .dll)
2. Static library (LIB or .lib)
3. Executable (.exe)

For more complex variants, use a Visual Studio [.NET solution](http://www.bogotobogo.com/CSharp/.netframework.php)which makes it possible to create and manage multiple projects.

For more on DLL, please visit   
<http://www.bogotobogo.com/Win32API/Win32API_DLL.php>.

Porting DLL to Linux/Unix

Modified **hey.h** looks like this:

// hey.h

#if defined (\_\_GNUC\_\_)

#define BOGO\_PRINT\_API \_\_attribute\_\_ ((\_\_visibility\_\_("default")))

#elif defined (WIN32)

#ifdef BUILD\_BOGO\_DLL

#define BOGO\_PRINT\_API \_\_declspec(dllexport)

#else

#define BOGO\_PRINT\_API \_\_declspec(dllimport)

#endif

#endif

extern "C" BOGO\_PRINT\_API void printHey();

Using the GNU C++ compiler, we can simply use the **-shared** linker option to generate a **.so** file instead of an executable. On platforms where it is not the default behavior, we should also specify either the **-fpic** or the **-fPIC** option to instruct the compiler to emit **position-independeet code (PIC)**. This is needed because the code in a shared library may be loaded into a different memory location for different executables. So, it's important to generate PIC code for shared libraries so that user code deos not dependent on the absolute memory address of symbols. A shared library is potentially mapped to a new memory address every time it gets loaded. Therefore, it makes sense to generate the addresses of all variables and functions inside the library in a way that can be easily computed relative to the start address that the library is loaded to. This code is generated by the **-fPIC** option and makes the code relocatable. The **-o** option is used to specify the name of an output file, and the **-shared** option builds a shared library in which unresolved references are allowed.

g++ -c -fPIC hey.cpp -o hey.o

g++ -shared -Wl,-soname,libhey.so.1 -o libhey.so.1.0.1 hey.o

Every shared library has a special name called the **soname**. The soname has the prefix **lib**, the name of the library, the phrase **.so**, followed by a period and a version number that is incremented whenever the interface changes. On a working system a fully-qualified soname is simply a symbolic link to the shared library's **real name**.

Every shared library also has a **real name**, which is the filename containing the actual library code. The real name adds to the soname a period, a minor number, another period, and the release number. The last period and release number are optional. The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed.

The last command (g++ -shared ..), on successful, produces a shared library named **libhey.so.1.0.1**.

In that command, **-W1** passes options to linker. In this example, the options to be passed on to the linker:

1. **-soname libhey.so.1**
2. the name passed with the **-o** option is passed to **g++**.

After those commands, we have the following in the list:

[]$ ls

hey.cpp hey.h hey.o libhey.so.1.0.1 main.cpp

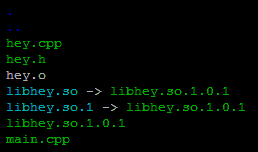
Now we have successfully created a shared library named **libhey.so.1.0.1**. Let us see how to include this shared library in our application.

symbolic link:

ln -sf libhey.so.1.0.1 libhey.so

ln -sf libhey.so.1.0.1 libhey.so.1

1. The link to **./libhey.so** allows the naming convention for the compile flag **-lhey** to work.
2. The link to **./libhey.so.1** allows the run time binding to work.

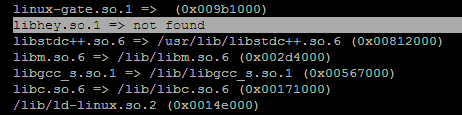


g++ -o main main.cpp -L. -lhey

1. The **-l** option tells the compiler to look for a file named libsomething.so The something is specified by the argument immediately following the **-l**. i.e. **-lhey**
2. The **-L** option tells the compiler where to find the library. The path to the directory containing the shared libraries is followd by **-L**. If no **-L** is specified, the compiler will search the usual locations.
3. **-L.** means looking for the shared libraries in the current directory.

If compilation is successful an executable named **main** is created. We can check if our library is include successfully into the executable by linker using the following command:

ldd main



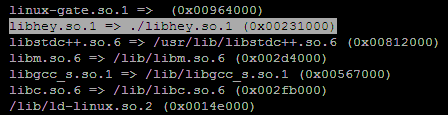
We can see that linker cannot find our shared library **libhey.so**.  
When we run an executable that depends on a dynamic library, linux system will search for the standard library locations, such as **/lib** and **/usr/lib**. If the **.so** file cannot be found in those places, the executable fail to start. The **ldd** commnad, as shown above, can be used to tell us if the system cannot find any dependent dynamic library. There are three main options to resolve this issue:

1. We have to ensure that our library is installed in the standard locations.
2. The **LD\_LIBRARY\_PATH** environment variable can be set to argument the default library search parh with a colon-separated list of directories.
3. We can use the **rpath (run path)** linker option to burn the preferred path to search for dynamic libraries into executable. The following line of compile will produce an executable that will cause the system to search in **/usr/local/lib** for any dynamic libraries:
4. g++ -o main main.cpp -L. -lhey -Wl,-rpath,/usr/local/lib

In our case, we will set the environment variable **LD\_LIBRARY\_PATH** to the directory containing the shared libraries, which is current directory ("."):

export LD\_LIBRARY\_PATH=.

If we do **ldd main** again:



Now, if we issue **./main**, we get:

Hey! Wlecome to bogotobogo

Creating Static Libraries on Linux

On linux, static libraries is an archive of object files. We use **ar** to compile object files into a static library. The command below shows how we compile **hey.cpp** file to **.o** and create a static library, **libhey.a** from the object files:

ar -crs libhey.a hey.o

The **-c** option to **ar** creates an archive, **-r** inserts the supplied **.o** file into that archive, and **–s** creates an index for the archive.

Then, we can link against the library using **-l** option to **ld** or **g++**. The **-L** linker option is used to specify the directory where the library can be found:

$ g++ -o main main.cpp -L. -lhey

$ ./main

Hey! Welcome to bogotobogo

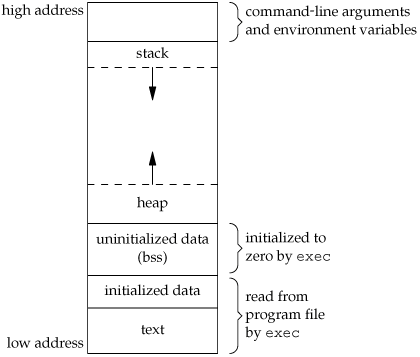
$

Also, see [Building Library - static](http://www.bogotobogo.com/cplusplus/libraries.php#building_libraries_static)

# Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment  
2. Initialized data segment  
3. Uninitialized data segment  
4. Stack  
5. Heap

[](http://www.geeksforgeeks.org/wp-content/uploads/Memory-Layout.gif)  
A typical memory layout of a running process

**1. Text Segment:**  
A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

**2. Initialized Data Segment:**  
Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

**3. Uninitialized Data Segment:**  
Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.  
For instance a global variable declared int j; would be contained in the BSS segment.

**4. Stack:**  
The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

**5. Heap:**  
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Examples.

The **size(1)** command reports the sizes (in bytes) of the text, data, and bss segments. ( for more details please refer man page of size(1) )

1. Check the following simple C program

|  |
| --- |
| #include <stdio.h>    int main(void)  {      return 0;  } |

Run on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ **size memory-layout**

text data bss dec hex filename

960 248 8 1216 4c0 memory-layout

2. Let us add one global variable in program, now check the size of bss (highlighted in red color).

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      return 0;  } |

Run on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **12** 1220 4c4 memory-layout

3. Let us add one static variable which is also stored in bss.

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      static int i; /\* Uninitialized static variable stored in bss \*/      return 0;  } |

Run on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **16** 1224 4c8 memory-layout

4. Let us initialize the static variable which will then be stored in Data Segment (DS)

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      static int i = 100; /\* Initialized static variable stored in DS\*/      return 0;  } |

Run on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **252 12** 1224 4c8 memory-layout

5. Let us initialize the global variable which will then be stored in Data Segment (DS)

|  |
| --- |
| #include <stdio.h>    int global = 10; /\* initialized global variable stored in DS\*/    int main(void)  {      static int i = 100; /\* Initialized static variable stored in DS\*/      return 0;  } |

Run on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **256 8** 1224 4c8 memory-layout

***Reference :***[***http://stackoverflow.com/questions/1350819/c-free-store-vs-heap***](http://stackoverflow.com/questions/1350819/c-free-store-vs-heap)

|  |  |
| --- | --- |
| **Heap** | **Free Store** |
| A dynamic memory area that is allocated/freed by the malloc/free functions. | A dynamic memory area that is allocated/freed by new/delete. |
| malloc and free do not call the constructor and destructor respectively. | new and delete call the constructor and destructor, respectively |

# C++ program's major distinct memory areas

Memory Area Characteristics and Object Lifetimes

-------------- ------------------------------------------------

Const Data The const data area stores string literals and

other data whose values are known at compile

time. No objects of class type can exist in

this area. All data in this area is available

during the entire lifetime of the program.

Further, all of this data is read-only, and the

results of trying to modify it are undefined.

This is in part because even the underlying

storage format is subject to arbitrary

optimization by the implementation. For

example, a particular compiler may store string

literals in overlapping objects if it wants to.

Stack The stack stores automatic variables. Typically

allocation is much faster than for dynamic

storage (heap or free store) because a memory

allocation involves only pointer increment

rather than more complex management. Objects

are constructed immediately after memory is

allocated and destroyed immediately before

memory is deallocated, so there is no

opportunity for programmers to directly

manipulate allocated but uninitialized stack

space (barring willful tampering using explicit

dtors and placement new).

Free Store The free store is one of the two dynamic memory

areas, allocated/freed by new/delete. Object

lifetime can be less than the time the storage

is allocated; that is, free store objects can

have memory allocated without being immediately

initialized, and can be destroyed without the

memory being immediately deallocated. During

the period when the storage is allocated but

outside the object's lifetime, the storage may

be accessed and manipulated through a void\* but

none of the proto-object's nonstatic members or

member functions may be accessed, have their

addresses taken, or be otherwise manipulated.

Heap The heap is the other dynamic memory area,

allocated/freed by malloc/free and their

variants. Note that while the default global

new and delete might be implemented in terms of

malloc and free by a particular compiler, the

heap is not the same as free store and memory

allocated in one area cannot be safely

deallocated in the other. Memory allocated from

the heap can be used for objects of class type

by placement-new construction and explicit

destruction. If so used, the notes about free

store object lifetime apply similarly here.

Global/Static Global or static variables and objects have

their storage allocated at program startup, but

may not be initialized until after the program

has begun executing. For instance, a static

variable in a function is initialized only the

first time program execution passes through its

definition. The order of initialization of

global variables across translation units is not

defined, and special care is needed to manage

dependencies between global objects (including

class statics). As always, uninitialized proto-

objects' storage may be accessed and manipulated

through a void\* but no nonstatic members or

member functions may be used or referenced

outside the object's actual lifetime.

# Namespace in C++

Consider following C++ program.

|  |
| --- |
| // A program to demonstrate need of namespace  int main()  {      int value;      value = 0;      double value; // Error here      value = 0.0;  } |

Run on IDE

Output :

Compiler Error:

'value' has a previous declaration as 'int value'

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

|  |
| --- |
| // Here we can see that more than one variables  // are being used without reporting any error.  // That is because they are declared in the  // different namespaces and scopes.  #include <iostream>  using namespace std;    // Variable created inside namespace  namespace first  {      int val = 500;  }    // Global variable  int val = 100;    int main()  {      // Local variable      int val = 200;        // These variables can be accessed from      // outside the namespace using the scope      // operator ::      cout << first::val << '\n';        return 0;  } |

Run on IDE

**Output:**

500

**Definition and Creation:**

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

Namespace is a feature added in C++ and not present in C.

A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.

Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

namespace namespace\_name

{

int x, y; // code declarations where

// x and y are declared in

// namespace\_name's scope

}

Namespace declarations appear only at global scope.

Namespace declarations can be nested within another namespace.

Namespace declarations don’t have access specifiers. (Public or private)

No need to give semicolon after the closing brace of definition of namespace.

We can split the definition of namespace over several units.

|  |
| --- |
| // Creating namespaces  #include <iostream>  using namespace std;  namespace ns1  {      int value()    { return 5; }  }  namespace ns2  {      const double x = 100;      double value() {  return 2\*x; }  }    int main()  {      // Access value function within ns1      cout << ns1::value() << '\n';        // Access value function within ns2      cout << ns2::value() << '\n';        // Access variable x directly      cout << ns2::x << '\n';        return 0;  } |

Run on IDE

**Output:**

5

200

100

**Classes and Namespace:**

Following is a simple way to create classes in a name space

|  |
| --- |
| // A C++ program to demonstrate use of class  // in a namespace  #include <iostream>  using namespace std;    namespace ns  {      // A Class in a namespace      class geek      {      public:          void display()          {              cout << "ns::geek::display()\n";          }      };  }    int main()  {      // Creating Object of student Class      ns::geek obj;        obj.display();        return 0;  } |

Run on IDE

Output:

ns::geek::display()

**Class can also be declared inside namespace and defined outside namespace** using following syntax

|  |
| --- |
| // A C++ program to demonstrate use of class  // in a namespace  #include <iostream>  using namespace std;    namespace ns  {      // Only declaring class here      class geek;  }    // Defining class outside  class ns::geek  {  public:      void display()      {          cout << "ns::geek::display()\n";      }  };    int main()  {      //Creating Object of student Class      ns::geek obj;      obj.display();      return 0;  } |

Run on IDE

Output:

ns::geek::display()

We can **define methods also outside the namespace**. Following is an example code.

|  |
| --- |
| // A C++ code to demonstrate that we can define  // methods outside namespace.  #include <iostream>  using namespace std;    // Creating a namespace  namespace ns  {      void display();      class geek      {      public:         void display();      };  }    // Defining methods of namespace  void ns::geek::display()  {      cout << "ns::geek::display()\n";  }  void ns::display()  {      cout << "ns::display()\n";  }    // Driver code  int main()  {      ns::geek obj;      ns::display();      obj.display();      return 0;  } |

Run on IDE

**Output:**

ns::display()

ns::geek::display()

Finally, you can introduce only specific members of a namespace using a using-declaration with the syntax

using *namespace\_name*::*thing*;

One trick with namespaces is to use an unnamed namespace to avoid naming conflicts. To do so, simply declare a namespace with the normal syntax, but leave off the identifier; when this is done, you will have

namespace

{

//functions

}

and within the namespace you are assured that no global names will conflict because each namespace's function names take precedence over outside function names.   
  
Now, you might ask, how can you actually use anything in that namespace? When your program is compiled, the "anonymous" namespace you have created will be accessible within the file you created it in. In effect, it's as though an additional "using" clause was included implicitly. This effectively limits the scope of anything in the namespace to the file level (so you can't call the functions in that namespace from another other file). This is comparable to the effect of the [static keyword](https://www.cprogramming.com/tutorial/statickeyword.html). 

#### Renaming namespaces

Finally, if you just don't feel like typing the entire name of namespace, but you're trying to keep to a good style and not use the *using* keyword, you can rename a namespace to reduce the typing:

namespace <new> = <old>

# **Can namespaces be nested in C++?**

In C++, [namespaces](http://www.geeksforgeeks.org/namespace-in-c/) can be nested, and resolution of namespace variables is hierarchical. For example, in the following code, namespace inner is created inside namespace outer, which is inside the global namespace. In the line “int z = x”, x refers to outer::x. If x would not have been in outer then this xwould have referred to x in global namespace.

|  |
| --- |
| #include <iostream>    int x = 20;  namespace outer {    int x = 10;    namespace inner {      int z = x; // this x refers to outer::x    }  }    int main()  {    std::cout<<outer::inner::z; //prints 10    getchar();    return 0;  } |

Run on IDE

Output of the above program is 10.

# [Where are the local, global, static, auto, register, extern, const, volatile variables are stored?](https://stackoverflow.com/questions/3684760/where-are-the-local-global-static-auto-register-extern-const-volatile-var)

* **local** variables can be stored either on the stack or in a data segment depending on whether they are auto or static. (if neither auto or static is explicitly specified, auto is assumed)
* **global** variables are stored in a data segment (unless the compiler can optimize them away, see const) and have visibility from the point of declaration to the end of the compilation unit.
* **static** variables are stored in a data segment (again, unless the compiler can optimize them away) and have visibility from the point of declaration to the end of the enclosing scope. Global variables which are not static are also visible in other compilation units (see extern).
* **auto** variables are always local and are stored on the stack.
* the **register** modifier tells the compiler to do its best to keep the variable in a register if at all possible. Otherwise it is stored on the stack.
* **extern** variables are stored in the data segment. The extern modifier tells the compiler that a different compilation unit is actually declaring the variable, so don't create another instance of it or there will be a name collision at link time.
* **const** variables can be stored either on the stack or a readonly data segment depending on whether they are auto or static. However, if the compiler can determine that they cannot be referenced from a different compilation unit, or that your code is not using the address of the const variable, it is free to optimize it away (each reference can be replaced by the constant value). In that case it's not stored anywhere.
* the **volatile** modifier tells the compiler that the value of a variable may change at anytime from external influences (usually hardware) so it should not try to optimize away any reloads from memory into a register when that variable is referenced. This implies static storage.

# Internal and External Linkage in C++

Ever come across the terms internal and external linkage? Ever wanted to know what the extern keyword is for and what declaring something static does in the global scope? Then this post is for you.

## TL;DR

A translation unit refers to an implementation (.c/.cpp) file and all header (.h/.hpp) files it includes. If an object or function inside such a translation unit has internal linkage, then that specific symbol is only visible to the linker within that translation unit. If an object or function has external linkage, the linker can also see it when processing other translation units. The static keyword, when used in the global namespace, forces a symbol to have internal linkage. The extern keyword results in a symbol having external linkage.

The compiler defaults the linkage of symbols such that:

* Non-const global variables have external linkage by default
* Const global variables have internal linkage by default
* Functions have external linkage by default

## Table of Contents

* [Table of Contents](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#table-of-contents)
* [Basics](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#basics)
  + [Declaration vs. Definition](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#declaration-vs-definition)
  + [Translation Units](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#translation-units)
* [Linkage](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#linkage)
  + [External Linkage](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#external-linkage)
  + [Internal Linkage](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#internal-linkage)
* [References](http://www.goldsborough.me/c/c++/linker/2016/03/30/19-34-25-internal_and_external_linkage_in_c++/#references)

## Basics

Lets first cover two rudimentary concepts that we’ll need to properly discuss linkage:

1. The difference between a declaration and a definition
2. Translation Units

Also, just a quick word on naming: we’ll use the term symbol to refer to any kind of “code entity” that a linker works with, i.e. variables and functions (and also classes/structs, but we won’t talk much about those).

### Declaration vs. Definition

Let’s quickly discuss the difference between declaring a symbol and defining a symbol: A declaration tells the compiler about the existence of a certain symbol and makes it possible to refer to that symbol everywhere where the explicit memory address or required storage of that symbol is not required. A definition tells the compiler what the body of a function contains or how much memory it must allocate for a variable.

Situations where a declaration is not sufficient to the compiler are, for example, when a data member of a class is of reference or value (as in, neither reference nor pointer) type. At the same time, it is always allowed to have pointers to a declared (but not defined) type, because pointers require fixed memory capacity (e.g. 8 bytes on 64-bit systems) and do not depend on the type pointed to. When you dereference that pointer, the definition does become necessary. Also, for function declarations, all parameters (no matter whether taken by value, reference or pointer) and the return type need only be declared and not defined. Definitions of parameter and return value types only become necessary for the function definition.

#### Functions

The difference between the declaration and definition of a function is fairly obvious:

int f(); // declaration

int f() { return 42; } // definition

#### Variables

For variables, it is a bit different. Declaration and definition are usually not explicitly separate. Most importantly, this:

int x;

Does not just declare x, but also define it. In this case, by calling the default constructor of int. (As an aside, in C++, as opposed to Java, the constructor of primitive types (such as int) does not default-initialize (in C++ lingo: value initialize) the variable to 0. The value of x above will be whatever garbage lay at the memory address allocated for it by the compiler.)

You can, however, explicitly separate the declaration of a variable from its definition by using the extern keyword:

extern int x; **// declaration**

int x = 42; **// definition**

However, when extern is prepended to the declaration and an initialization is provided as well, then the expression turns into a definition and the extern keyword essentially becomes useless:

extern int x = 5; // is the same thing as

int x = 5;

#### Forward Declaring

In C++ there exists the concept of forward declaring a symbol. What we mean by this is that we declare the type and name of a symbol so that we can use it where its definition is not required. By doing so, we don’t have to include the full definition of a symbol (usually a header file) when it is not explicitly necessary. This way, we reduce dependency on the file containing the definition. The main advantage of this is that when the file containing the definition changes, the file where we forward declared that symbol does not need to be re-compiled (and therefore, also not all further files including it).

##### Example

Say we have a function declaration (also called prototype) for f, taking an object of type Class by value:

// file.hpp

void f(Class object);

Now, the naïve thing to do would be to include Class’s definition right away. But because we only declare f here, it is sufficient to provide the compiler with a declaration of Class. This way, the compiler can identify the function by its prototype, but we can remove the dependency of file.hpp on the file containing the definition of Class, say class.hpp:

// file.hpp

class Class;

void f(Class object);

Now, say, we include file.hpp in 100 other files. And say we change Class’s definition in class.hpp. If we had included class.hpp in file.hpp, file.hppand all 100 files including it would have to be recompiled. By forward declaring Class, the only files requiring recompilation are class.hpp and file.cpp(assuming that’s where f is defined).

#### Usage Frequency

One very important difference between declarations and definitions is that a symbol may be **declared many times, but defined only once**. For example, you can forward declare a function or class however often you want, but you may only ever have one definition for it. This is called **the**[**one definition rule**](https://en.wikipedia.org/wiki/One_Definition_Rule). Therefore, this is valid C++:

int f();

int f();

int f();

int f();

int f();

int f();

int f() { return 5; }

While this isn’t:

int f() { return 6; }

int f() { return 9; }

### Translation Units

Programmers usually deal with header files and implementation files. Compilers don’t – they deal with translation units (TUs), sometimes referred to as compilation units. The definition of such a **translation unit** is very simple: Any file, fed to the compiler, after it has been pre-processed. In detail, this means that it is the file resulting from the pre-processor expanding macros, conditionally including source code depending on #ifdef and #ifndef statements and copy-pasting any #includeed files.

Given these files:

header.hpp:

**#ifndef HEADER\_HPP**

**#define HEADER\_HPP**

**#define VALUE 5**

**#ifndef VALUE**

struct Foo { private: int ryan; };

**#endif**

int strlen(const char\* string);

**#endif** /\* HEADER\_HPP \*/

program.cpp:

**#include "header.hpp"**

int strlen(const char\* string)

{

int length = 0;

while(string[length]) ++length;

return length + VALUE;

}

The pre-processor will produce the following translation unit, which is then fed to the compiler:

int strlen(const char\* string);

int strlen(const char\* string)

{

int length = 0;

while(string[length]) ++length;

return length + 5;

}

## Linkage

Now that we’ve covered the basics, we can deal with linkage. In general, linkage will refer to the visibility of symbols to the linker when processing files. Linkage can be either internal or external.

### External Linkage

When a symbol (variable or function) has external linkage, that means that that **symbol is visible to the linker from other files**, i.e. it is “globally” visible and can be shared between translation units. In practice, this means that you must define such a symbol in a place where it will end up in one and only one translation unit, typically an implementation file (.c/.cpp), such that it has only one visible definition. If you were to define such a symbol on the spot, along with declaring it, or to place its definition in the same file you declare it, you run the risk of making your linker very angry. As soon as you include that file in more than one implementation file, such that its definition ends up in more than one translation unit, your linker will start crying.

In C and C++, the extern keyword (explicitly) declares a symbol to have external linkage:

extern int x;

extern void f(const std::string& argument);

Both of these symbols have external linkage. **Above it was mentioned that const global variables have internal linkage by default, and non-const global variables have external linkage by default**. That means that int x; is the same as extern int x;, right? Not quite. int x; is actually the same as extern int x{}; (using C++11 uniform/brace initialization syntax to avoid the most vexing parse), as int x; not only declares, but also defines x. Therefore, not prepending extern to int x; in the global scope is just as bad as also defining a variable when declaring it as extern:

int x; // is the same as

extern int x{}; // which will both likely cause linker errors.

extern int x; // while this only declares the integer, which is ok.

#### Example Badness

Let’s declare a function f with external linkage in file.hpp and also define it in the same file:

// file.hpp

#ifndef FILE\_HPP

#define FILE\_HPP

extern int f(int x);

/\* ... \*/

int f(int) { return x + 1; }

/\* ... \*/

#endif /\* FILE\_HPP \*/

Note that prepending extern here is redundant, as all functions are implicitly extern, and separating the declaration from the definition here is also unnecessary. So let’s just quickly rewrite this as:

// file.hpp

#ifndef FILE\_HPP

#define FILE\_HPP

int f(int) { return x + 1; }

#endif /\* FILE\_HPP \*/

This is code one would be inclined to write before reading this article or after reading it but under influence of alcohol or strong drugs (e.g. pop tarts).

So let’s see why this is bad. We’ll now have two implementation files: a.cpp and b.cpp, both including this file.hpp:

// a.cpp

#include "file.hpp"

/\* ... \*/

// b.cpp

#include "file.hpp"

/\* ... \*/

Now let the compiler do its job and generate two translation units for the two implementation files above (remember that #includeing means to literally copy-paste):

// TU A, from a.cpp

int f(int) { return x + 1; }

/\* ... \*/

// TU B, from b.cpp

int f(int) { return x + 1; }

/\* ... \*/

At this point, the linker will step in (linking comes after compilation). The linker will pick up the symbol f and look for definitions. Because it’s the linker’s lucky day, it will even find two! One in TU A and one in TU B. The linker will be so happy, it’ll stop and tell you in a way similar to this:

duplicate symbol \_\_Z1fv in:

/path/to/a.o

/path/to/b.o

The linker found two definitions for the same symbol f. Because it had external linkage, f was visible to the linker when processing both TU A and TU B. Naturally, this violates the One-Definition-Rule, so this causes a linker error. More specifically, this is when you get a duplicate symbol error, which is the one you’ll get most often along with an undefined symbol error (if we had only ever declared, but never defined f).

#### Usage

A common use case for declaring variables explicitly extern are global variables. For example, say you are working on a self-baking cake. There may be certain global system variables connected with self-baking cakes that you need to access in various places throughout your program. Let’s say the clock-rate of the edible chip inside your cake. Such a value would naturally be required in many, many places to make all the chocolate electronics work synchronously. The C (evil) way of declaring such a global variable would be a macro:

#define CLK 1000000

A C++ programmer, naturally despising macros, would rather use real code. So you could do this:

// global.hpp

namespace Global

{

extern unsigned int clock\_rate;

}

// global.cpp

namespace Global

{

unsigned int clock\_rate = 1000000;

}

(As a modern C++ programmer, you might also want to take advantage of (separator literals)[http://www.informit.com/articles/article.aspx?p=2209021]: unsigned int clock\_rate = 1'000'000;)

### Internal Linkage

When a symbol has internal linkage, it will only be visible within the current translation unit. Do not confuse the term visible here with access rights like private. Visibility here means that the linker will only be able to use this symbol when processing the translation unit in which the symbol was declared, and not later (as with symbols with external linkage). In practice, this means that when you declare a symbol to have internal linkage in a header file, each translation unit you include this file in will get its own unique copy of that symbol. I.e. it will be as if you redefined each such symbol in every translation unit. For objects, this means that the compiler will literally allocate an entirely new, unique copy for each translation unit, which can obviously incur high memory costs.

To declare a symbol with internal linkage, C and C++ provide the **static** keyword. Its usage here is entirely separate from its usage in classes or functions (or, generally, any block).

#### Example

Here an example:

header.hpp:

static int variable = 42;

file1.hpp:

void function1();

file2.hpp:

void function2();

file1.cpp:

**#include "header.hpp"**

void function1() { variable = 10; }

file2.cpp:

**#include "header.hpp"**

void function2() { variable = 123; }

main.cpp:

**#include "header.hpp"**

**#include "file1.hpp"**

**#include "file2.hpp"**

**#include <iostream>**

auto main() -> int

{

function1();

function2();

std::cout << variable << std::endl;

}

Because variable has internal linkage, each translation unit that includes header.hpp gets its own unique copy of variable. Here, there are three translation units:

1. file1.cpp
2. file2.cpp
3. main.cpp

When function1 is called, file1.cpp’s copy of variable is set to 10. When function2 is called, file2.cpp’s copy of variable is set to 123. However, the value printed out in main.cpp is variable, unchanged: 42.

#### Anonymous Namespaces

In C++, there exists another way to declare one or more symbols to have internal linkage: anonymous namespaces. Such a namespace ensures that the symbols declared inside it are visible only within the current translation unit. It is, in essence, just a way to declare many symbols as static. In fact, for a while, the static keyword for the use of declaring a symbol to have internal linkage was deprecated in favor of anonymous namespaces. However, it was recently undeprecated, because it is useful to declare a single variable or function to have internal linkage. There are also a few minor differences which I won’t go into here.

In any case, this:

namespace { int variable = 0; }

does (almost) the same thing as this:

static int variable = 0;

#### Usage

So when and why would one make use of internal linkage? For objects, it is probably most often a very bad idea to make use of it. The memory cost can be very high for large objects given that each translation unit gets its own copy. But mainly, it can really just cause odd, unexpected behavior. Imagine you had a singleton (a class of which you instantiate only a single instance), and would suddenly end up having multiple instances of your “singleton” (one for every translation unit).

However, one interesting use case could be to hide translation-unit-local helper functions from the global scope. Imagine you have a helper function fooin your file1.hpp which you use in file1.cpp, but then you also have a helper function foo in your file2.hpp which you use in file2.cpp. The first foodoes something completely different than the second foo, but you cannot think of a better name for them. So, you can declare them both static. Unless you include both file1.hpp and file2.hpp in some same translation unit, this will hide the respective foos from each other. If you don’t declare themstatic, they will implicitly have external linkage and the first foo’s definition will collide with the second foos definition and cause a linker error due to a violation of the one-definition-rule.

# ar(1) - Linux man page

## Name

ar - create, modify, and extract from archives

## Synopsis

ar [**--plugin** *name*] [**-X32\_64**] [**-**]*p*[*mod* [*relpos*] [*count*]] *archive* [*member*...]

## Description

The GNU **ar** program creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called *members* of the archive).

The original files' contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and can be restored on extraction.

GNU **ar** can maintain archives whose members have names of any length; however, depending on how **ar** is configured on your system, a limit on member-name length may be imposed for compatibility with archive formats maintained with other tools. If it exists, the limit is often 15 characters (typical of formats related to a.out) or 16 characters (typical of formats related to coff).

**ar** is considered a binary utility because archives of this sort are most often used as *libraries* holding commonly needed subroutines.

**ar** creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier **s**. Once created, this index is updated in the archive whenever **ar** makes a change to its contents (save for the **q** update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use **nm -s** or **nm --print-armap** to list this index table. If an archive lacks the table, another form of **ar** called **ranlib** can be used to add just the table.

GNU **ar** can optionally create a *thin* archive, which contains a symbol index and references to the original copies of the member files of the archives. Such an archive is useful for building libraries for use within a local build, where the relocatable objects are expected to remain available, and copying the contents of each object would only waste time and space. Thin archives are also *flattened*, so that adding one or more archives to a thin archive will add the elements of the nested archive individually. The paths to the elements of the archive are stored relative to the archive itself.

GNU **ar** is designed to be compatible with two different facilities. You can control its activity using command-line options, like the different varieties of **ar** on Unix systems; or, if you specify the single command-line option **-M**, you can control it with a script supplied via standard input, like the MRI "librarian" program.

## Options

GNU **ar** allows you to mix the operation code *p* and modifier flags *mod* in any order, within the first command-line argument.

If you wish, you may begin the first command-line argument with a dash.

The *p* keyletter specifies what operation to execute; it may be any of the following, but you must specify only one of them:

**d**

*Delete* modules from the archive. Specify the names of modules to be deleted as *member*...; the archive is untouched if you specify no files to delete.

If you specify the **v** modifier, **ar** lists each module as it is deleted.

**m**

Use this operation to *move* members in an archive.

The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member.

If no modifiers are used with "m", any members you name in the *member* arguments are moved to the *end*of the archive; you can use the **a**, **b**, or **i** modifiers to move them to a specified place instead.

**p**

*Print* the specified members of the archive, to the standard output file. If the **v** modifier is specified, show the member name before copying its contents to standard output.

If you specify no *member* arguments, all the files in the archive are printed.

**q**

*Quick append*; Historically, add the files *member*... to the end of *archive*, without checking for replacement.

The modifiers **a**, **b**, and **i** do *not* affect this operation; new members are always placed at the end of the archive.

The modifier **v** makes **ar** list each file as it is appended.

Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed; you can use **ar s** or **ranlib** explicitly to update the symbol table index.

However, too many different systems assume quick append rebuilds the index, so GNU **ar** implements **q** as a synonym for **r**.

**r**

Insert the files *member*... into *archive* (with *replacement*). This operation differs from **q** in that any previously existing members are deleted if their names match those being added.

If one of the files named in *member*... does not exist, **ar** displays an error message, and leaves undisturbed any existing members of the archive matching that name.

By default, new members are added at the end of the file; but you may use one of the modifiers **a**, **b**, or **i**to request placement relative to some existing member.

The modifier **v** used with this operation elicits a line of output for each file inserted, along with one of the letters **a** or **r** to indicate whether the file was appended (no old member deleted) or replaced.

**t**

Display a *table* listing the contents of *archive*, or those of the files listed in *member*... that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the **v** modifier.

If you do not specify a *member*, all files in the archive are listed.

If there is more than one file with the same name (say, **fie**) in an archive (say **b.a**), **ar t b.a fie** lists only the first instance; to see them all, you must ask for a complete listing---in our example, **ar t b.a**.

**x**

*Extract* members (named *member*) from the archive. You can use the **v** modifier with this operation, to request that **ar** list each name as it extracts it.

If you do not specify a *member*, all files in the archive are extracted.

Files cannot be extracted from a thin archive.

A number of modifiers (*mod*) may immediately follow the *p* keyletter, to specify variations on an operation's behavior:

**a**

Add new files *after* an existing member of the archive. If you use the modifier **a**, the name of an existing archive member must be present as the *relpos* argument, before the *archive* specification.

**b**

Add new files *before* an existing member of the archive. If you use the modifier **b**, the name of an existing archive member must be present as the *relpos* argument, before the *archive* specification. (same as **i**).

**c**

*Create* the archive. The specified *archive* is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

**D**

Operate in *deterministic* mode. When adding files and the archive index use zero for UIDs, GIDs, timestamps, and use consistent file modes for all files. When this option is used, if **ar** is used with identical options and identical input files, multiple runs will create identical output files regardless of the input files' owners, groups, file modes, or modification times.

**f**

Truncate names in the archive. GNU **ar** will normally permit file names of any length. This will cause it to create archives which are not compatible with the native **ar** program on some systems. If this is a concern, the **f** modifier may be used to truncate file names when putting them in the archive.

**i**

Insert new files *before* an existing member of the archive. If you use the modifier **i**, the name of an existing archive member must be present as the *relpos* argument, before the *archive* specification. (same as **b**).

**l**

This modifier is accepted but not used.

**N**

Uses the *count* parameter. This is used if there are multiple entries in the archive with the same name. Extract or delete instance *count* of the given name from the archive.

**o**

Preserve the *original* dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.

**P**

Use the full path name when matching names in the archive. GNU **ar** can not create an archive with a full path name (such archives are not POSIX complaint), but other archive creators can. This option will cause GNU **ar** to match file names using a complete path name, which can be convenient when extracting a single file from an archive created by another tool.

**s**

Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running **ar s** on an archive is equivalent to running **ranlib** on it.

**S**

Do not generate an archive symbol table. This can speed up building a large library in several steps. The resulting archive can not be used with the linker. In order to build a symbol table, you must omit the **S**modifier on the last execution of **ar**, or you must run **ranlib** on the archive.

**T**

Make the specified *archive* a *thin* archive. If it already exists and is a regular archive, the existing members must be present in the same directory as *archive*.

**u**

Normally, **ar r**... inserts all files listed into the archive. If you would like to insert *only* those of the files you list that are newer than existing members of the same names, use this modifier. The **u** modifier is allowed only for the operation **r** (replace). In particular, the combination **qu** is not allowed, since checking the timestamps would lose any speed advantage from the operation **q**.

**v**

This modifier requests the *verbose* version of an operation. Many operations display additional information, such as filenames processed, when the modifier **v** is appended.

**V**

This modifier shows the version number of **ar**.

# **UNIX ar Examples: How To Create, View, Extract, Modify C Archive Files (\*.a)**

*by* BALAKRISHNAN MARIYAPPAN *on* AUGUST 11, 2010

ar is an archive tool used to combine objects to create an archive file with .a extension, also known as library.

In this article, let us discuss about how to create an user defined static library in C programming using the “ar” utility. The examples shows how to create, extract, and modify the archives using Linux ar command.

To demonstrate the static library creation, let us create two C programs — addition.c and multiplication.c

Using gcc, the object code for these programs are obtained, and the static library libarith.a is created from these two objects.

### **1. Create Two Sample C Programs**

Create addition.c program as shown below.

int addition(int a,int b)

{

int result;

result = a + b;

return result;

}

Create multiplication.c program as shown below.

int multiplication(int a, int b)

{

int result;

result = a \* b;

return result;

}

A while back we discussed about fundamental of writing C program using [C hello world example](http://www.thegeekstuff.com/2009/09/how-to-write-compile-and-execute-c-program-on-unix-os-with-hello-world-example/).

### **2. Compile the Programs and Get Object Codes**

Use -c option to compile both the c program. Using option -c will create the corresponding .o files.

$ gcc -c addition.c

$ gcc -c multiplication.c

Now, the current working directory contains both the .c and .o files as shown below.

$ ls

addition.c multiplication.c addition.o multiplication.o

### **3. Create the C Program Static Library using ar utility**

Now create the static library “libarith.a” with the addition object file and multiplication object file as follows,

$ ar cr libarith.a addition.o multiplication.o

### **4. Write C program to Use the Library libarith.a**

The library file libarith.a is now ready to usage. Following example indicates how to write a sample C program with the header file to use the libarith.a static library.

Create header.h :

#include <stdio.h>

int addition(int a,int b);

int multiplication(int a,int b);

Create example.c :

#include "header.h"

int main()

{

int result;

result = addition(1,2);

printf("addition result is : %d\n",result);

result = multiplication(3,2);

printf("multiplication result is : %d\n",result);

}

Note: [How to Debug C Program using gdb in 5 Simple Steps](http://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/) provides step-by-step instruction on debugging your C code.

Compile example.c :

$ gcc -Wall example.c -L/home/guest/ -larith -o example

The option -L instructs the compiler to look in the /home/guest directory for library files. From this directory, the compiler takes the libarith library file, compiles it with example.c program.

Another method to Compile example.c :

$ gcc -Wall example.c libarith.a -o example

Execute example executable :

$ ./example

addition result is : 3

multiplication result is : 6

### **5. View Object Files in an Archive Using ar Command, option t**

To list the object files available in the libarith.a:

$ ar t libarith.a

addition.o

multiplication.o

The options in ar command are similar to the [tar command](http://www.thegeekstuff.com/2010/04/unix-tar-command-examples/).

### **6. Extract Object Files from an Archive Using ar Command, option x**

You can extract the object files available in an archive as follows.

$ mkdir object

$ cp libarith.a object/

$ cd object

$ ar x libarith.a

$ ls \*.o

addition.o

multiplication.o

### **7. Add an Object File into the Existing Archive Using ar, option r**

Let assume that you have create another object file called subtraction.o

The following command extends the libarith.a library file, by inserting subtraction.o object as shown below.

$ ar r libarith.a subtraction.o

$ ar t libarith.a

addition.o

multiplication.o

subtraction.o

While inserting a .o file, it it already exists in the archive, it would be replaced. Without checking for replacements the objects can be added to end of the archive by using -q option.

### **8. Delete a Specific Archive Member Using ar, option d**

In order to delete a specific archive member from the library file, do the following.

$ ar d libarith.a addition.o

$ ar t libarith.a

multiplication.o

subtraction.o

# Creating Libraries

* If you have a bunch of files that contain just functions, you can turn these source files into libraries that can be used statically or dynamically by programs. This is good for program modularity, and code re-use. Write Once, Use Many.
* A library is basically just an archive of object files.

## Creating Libraries :: Static Library Setup

* First thing you must do is create your C source files containing any functions that will be used. Your library can contain multiple object files.
* After creating the C source files, compile the files into object files.
* To create a library:
* ar rc libmylib.a objfile1.o objfile2.o objfile3.o

This will create a static library called libname.a. Rename the "mylib" portion of the library to whatever you want.

* Next:
* ranlib libmylib.a

This creates an index inside the library. That should be it! If you plan on copying the library, remember to use the -p option with cp to preserve permissions.

## Creating Libraries :: Static Library Usage

* Remember to prototype your library function calls so that you do not get implicit declaration errors.
* When linking your program to the libraries, make sure you specify where the library can be found:
* gcc -o foo -L. -lmylib foo.o

The -L. piece tells gcc to look in the current directory in addition to the other library directories for finding libmylib.a.

* You can easily **integrate** this into your Makefile (even the Static Library Setup part)!

## Creating Libraries :: Shared Library Setup

* Creating shared or dynamic libraries is simple also. Using the previous example, to create a shared library:
* gcc -fPIC -c objfile1.c
* gcc -fPIC -c objfile2.c
* gcc -fPIC -c objfile3.c
* gcc -shared -o libmylib.so objfile1.o objfile2.o objfile3.o

The -fPIC option is to tell the compiler to create Position Independent Code (create libraries using relative addresses rather than absolute addresses because these libraries can be loaded multiple times). The -shared option is to specify that an architecture-dependent shared library is being created. However, not all platforms support this flag.

* Now we have to compile the actual program using the libraries:
* gcc -o foo -L. -lmylib foo.o

Notice it is exactly the same as creating a static library. Although, it is compiled in the same way, none of the actual library code is inserted into the executable, hence the dynamic/shared library.

* **Note:** You can automate this process using Makefiles!

## Creating Libraries :: Shared Library Usage

* Since programs that use static libraries already have the library code compiled into the program, it can run on its own. Shared libraries dynamically access libraries **at run-time** thus the program needs to know where the shared library is stored. What's the advantage of creating executables using Dynamic Libraries? The executable is much smaller than with static libraries. If it is a standard library that can be installed, there is no need to compile it into the executable at compile time!
* The key to making your program work with dynamic libraries is through the LD\_LIBRARY\_PATH enviornment variable. To display this variable, at a shell:
* echo $LD\_LIBRARY\_PATH

Will display this variable if it is already defined. If it isn't, you can create a wrapper script for your program to set this variable at run-time. Depending on your shell, simply use setenv (tcsh, csh) or export (bash, sh, etc) commands. If you already have LD\_LIBRARY\_PATH defined, make sure you **append** to the variable, not overwrite it! For example:

setenv LD\_LIBRARY\_PATH /path/to/library:${LD\_LIBRARY\_PATH}

would be the command you would use if you had tcsh/csh and already had an existing LD\_LIBRARY\_PATH. If you didn't have it already defined, just remove everything right of the :. An example with bash shells:

export LD\_LIBRARY\_PATH=/path/to/library:${LD\_LIBRARY\_PATH}

Again, remove the stuff right of the : and the : itself if you don't already have an existing LD\_LIBRARY\_PATH.

* If you have administrative rights to your computer, you can install the particular library to the /usr/local/lib directory and permanently add an LD\_LIBRARY\_PATH into your .tcshrc, .cshrc, .bashrc, etc. file.