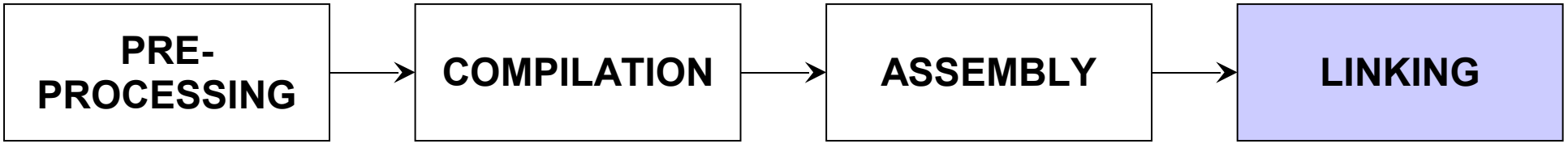
Linking

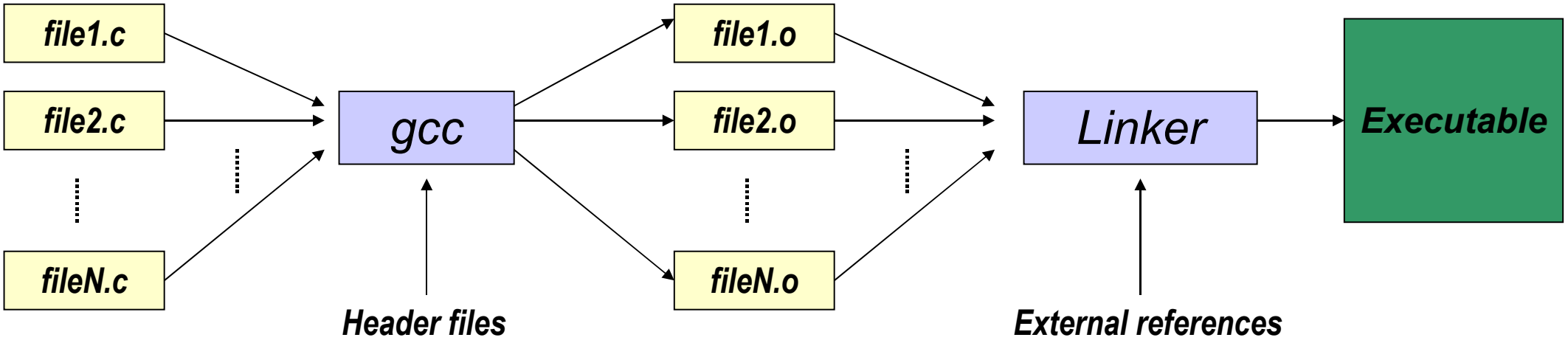
- Last stage in building a program



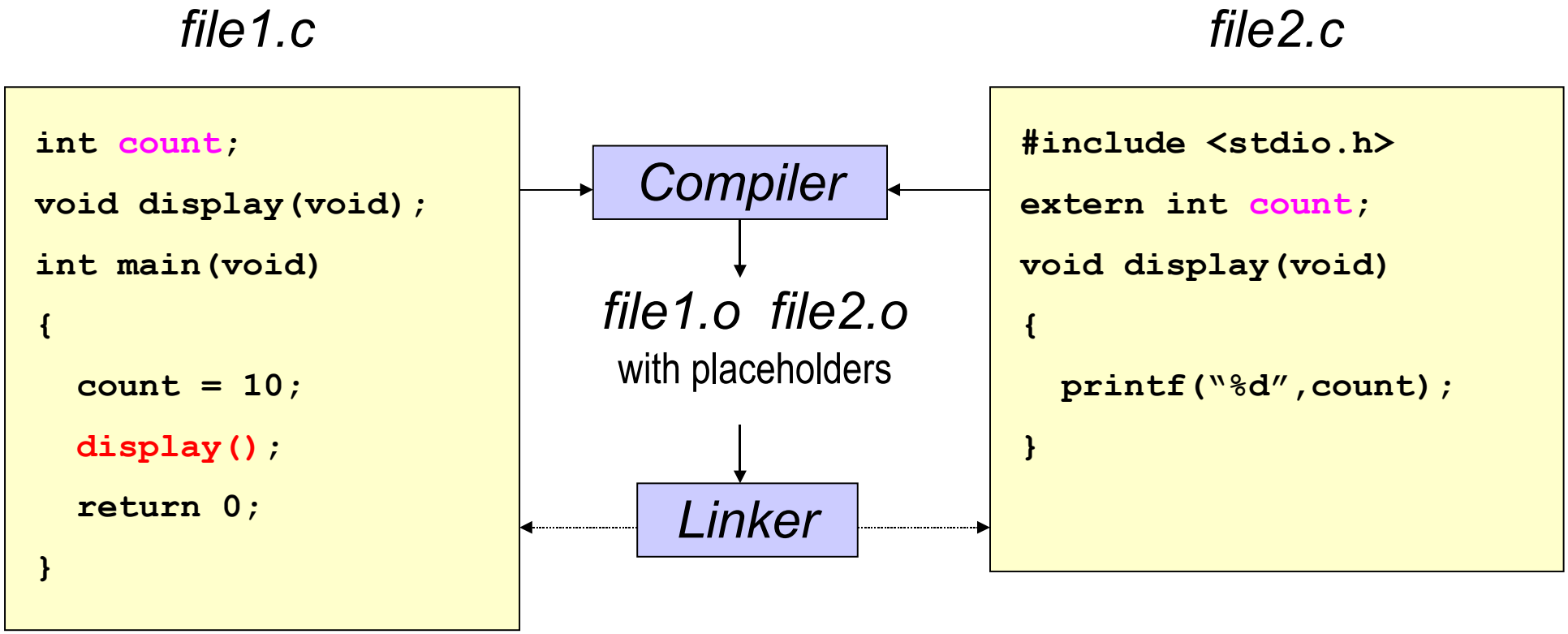
- Combining separate code into one executable
- Linking done by the Linker
  - ld in Unix
  - a.k.a. “link-editor” or “loader”
- Often transparent (gcc can do it all for you)

Linking involves...

- Combining several object modules (the `.o` files corresponding to `.c` files) into one file
- Resolving external references to variables and functions
- Producing an executable file (if no errors)



Linking with External References



- **`file1.o`** has placeholder for **`display()`**
- **`file2.o`** has placeholder for **`count`**
- object modules are relocatable
  - addresses are relative offsets from top of file

Libraries

- Definition:
  - a file containing functions that can be referenced externally by a C program
- Purpose:
  - easy access to functions used repeatedly
  - promote code modularity and re-use
  - reduce source and executable file size

Libraries

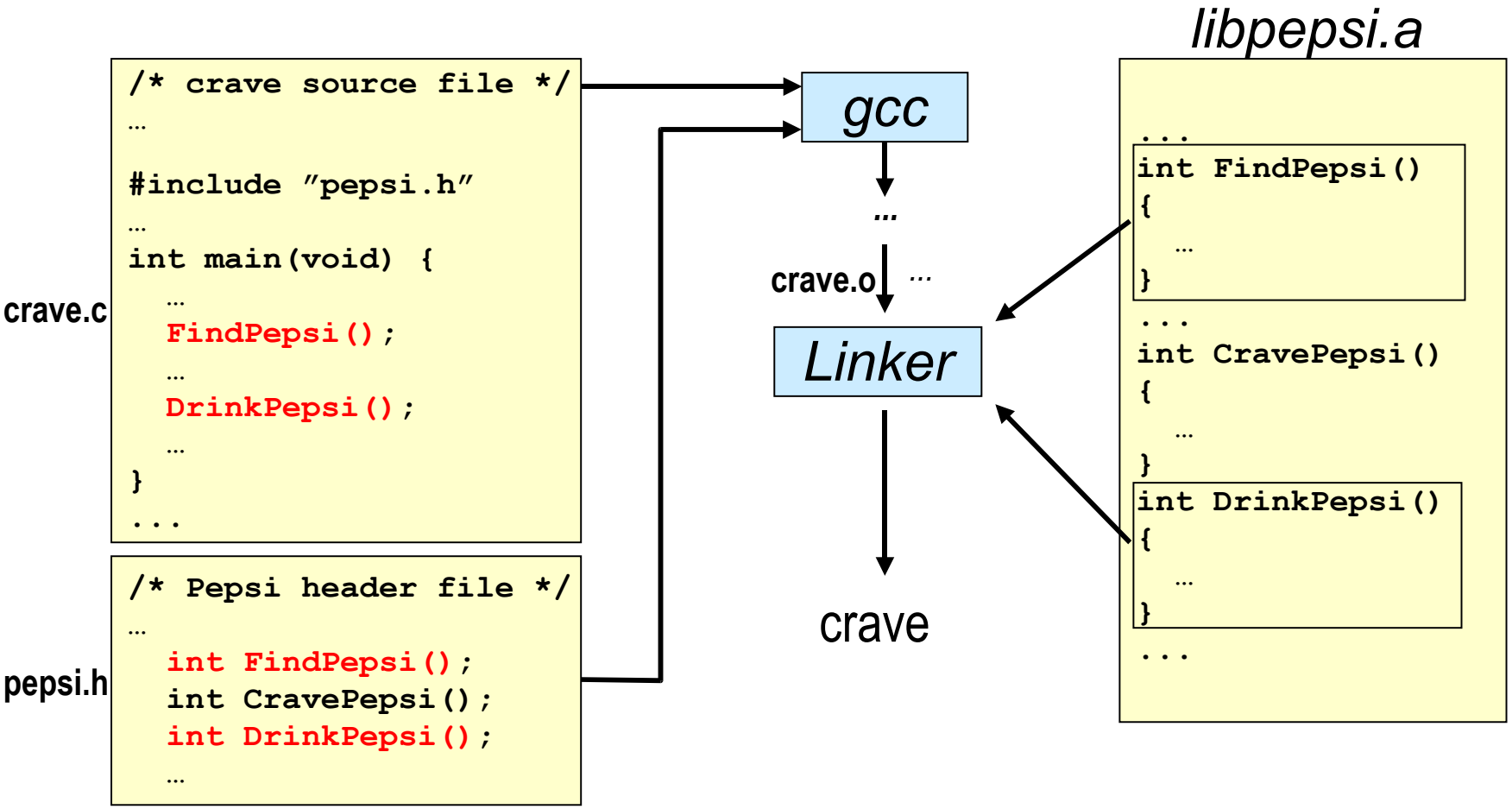
- Static (*Archive*)
  - **`libname.a`** on Unix; **`name.lib`** on DOS/Windows
  - Only modules with referenced code linked when compiling
    - unlike **`.o`** files
  - Linker copies function from library into executable file
  - Update to library requires recompiling program

Libraries

- Dynamic (*Shared Object* or *Dynamic Link Library*)
  - **`libname.so`** on Unix; **`name.dll`** on DOS/Windows
  - Referenced code not copied into executable
    - Loaded in memory at run time
  - Smaller executable size
  - Can update library without recompiling program
  - Drawback: slightly slower program startup

Libraries

- Linking a static library



Standard libraries

- ANSI C standard: set of functions which must be supported by ANSI-standard compilers
- Standard ANSI headers contain library function prototypes: stdio.h, stdlib.h, string.h, time.h, signal.h, math.h, …
- Standard libraries used by default when compiling
- Compilers can provide additional non-ANSI functions (e.g. graphics)
- Unix: Standard libs often found in /lib and /usr/lib (Solaris)

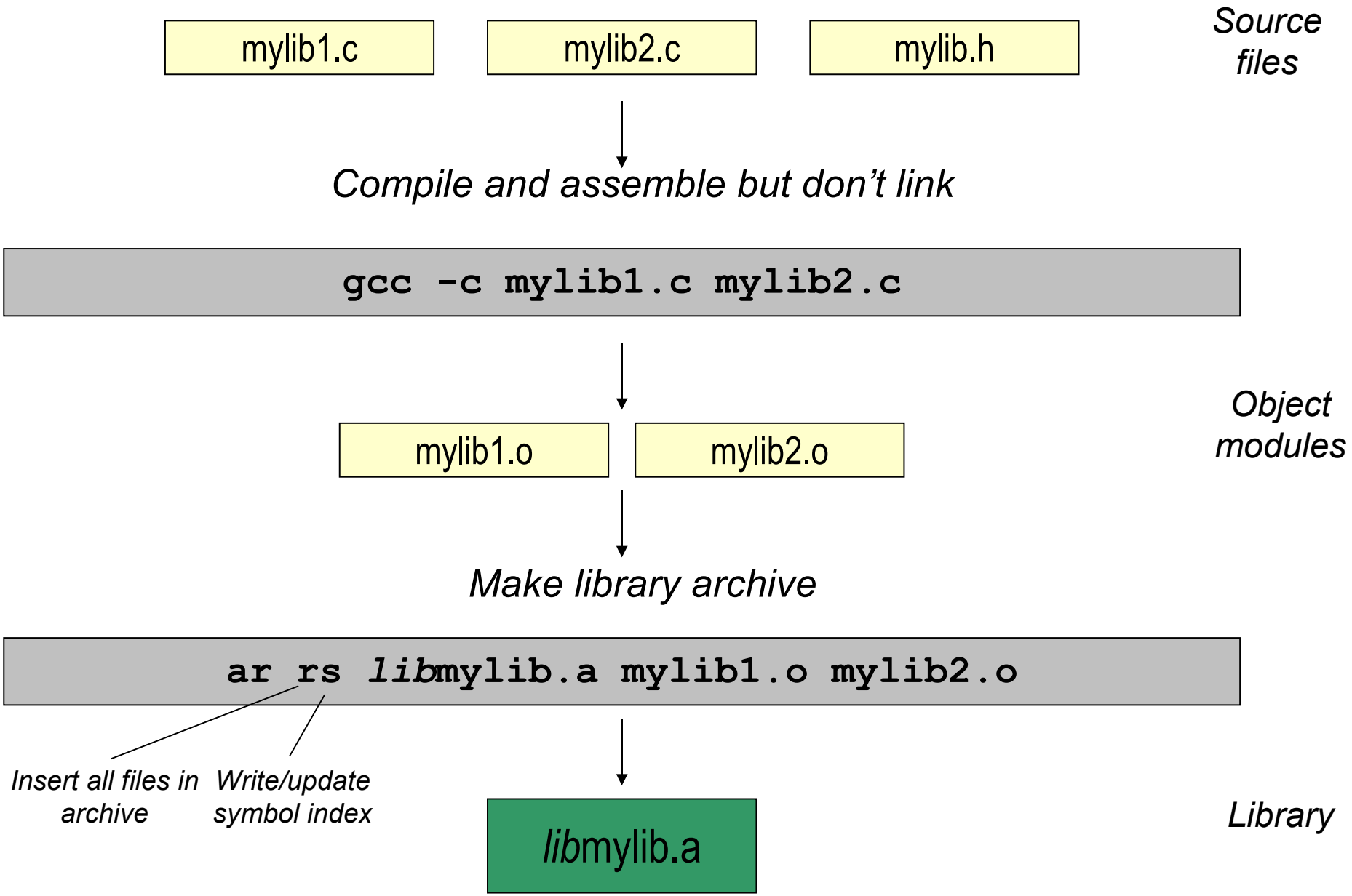
Creating your own library

- Why:
  - If you have functions that are used a lot by different programs
  - If you want to share your functions with others

Creating your own library

- How
  - collect functions into one or more .c files
  - must not have **`main()`** function
  - put function prototypes in header file
  - if you must use globals, make them static
  - create the library using **`ar`** command (Unix)
- **`ar`**
  - tool to create, modify, extract from an archive
  - archive = indexed collection of object files
  - keeps an index of symbols (functions, variables) defined in object files for easy retrieval

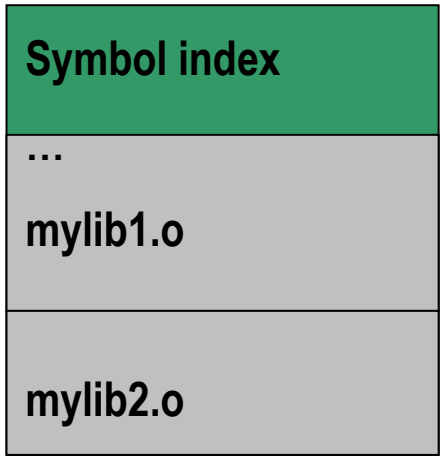
Using ar


mylib1.c
mylib2.c
mylib.h
Source files
Compile and assemble but don't link
gcc -c mylib1.c mylib2.c
mylib1.o
mylib2.o
Object modules
Make library archive
ar rs libmylib.a mylib1.o mylib2.o
Insert all files in archive
Write/update symbol index
libmylib.a
Library

Using `ar`

- **`ar operation[modifier] archive [list of files]`**
  (e.g. **`ar rs libmylib.a mylib1.o mylib2.o`**)
- Common operations
  - **`d`** : delete modules
  - **`p [list]`**: print specified modules to stdout
  - **`r`** : insert with replacement
  - **`t`** : print table of modules
  - **`x`** : extract modules
- Additional modifiers used with above
  - e.g. **`rs`** : insert + update index
  - **`tv`** : include timestamp, owner, etc.
  - **`rsu`** : insert only updated modules and update index

libmylib.a



gcc's library options

- **`-lname`**
  - use library archive **`libname.a`**
    - e.g.: **`gcc -o plot main.o plot_line.o -lm`**

"use libm.a"

  - order of **`.o`**'s and **`-l`**'s important!
  - e.g.: **`gcc file1.c -lmylib file2.c`**
    may cause linker error: **`Undefined symbol [file2.o]`**
    (Outstanding references resolved only when library searched)

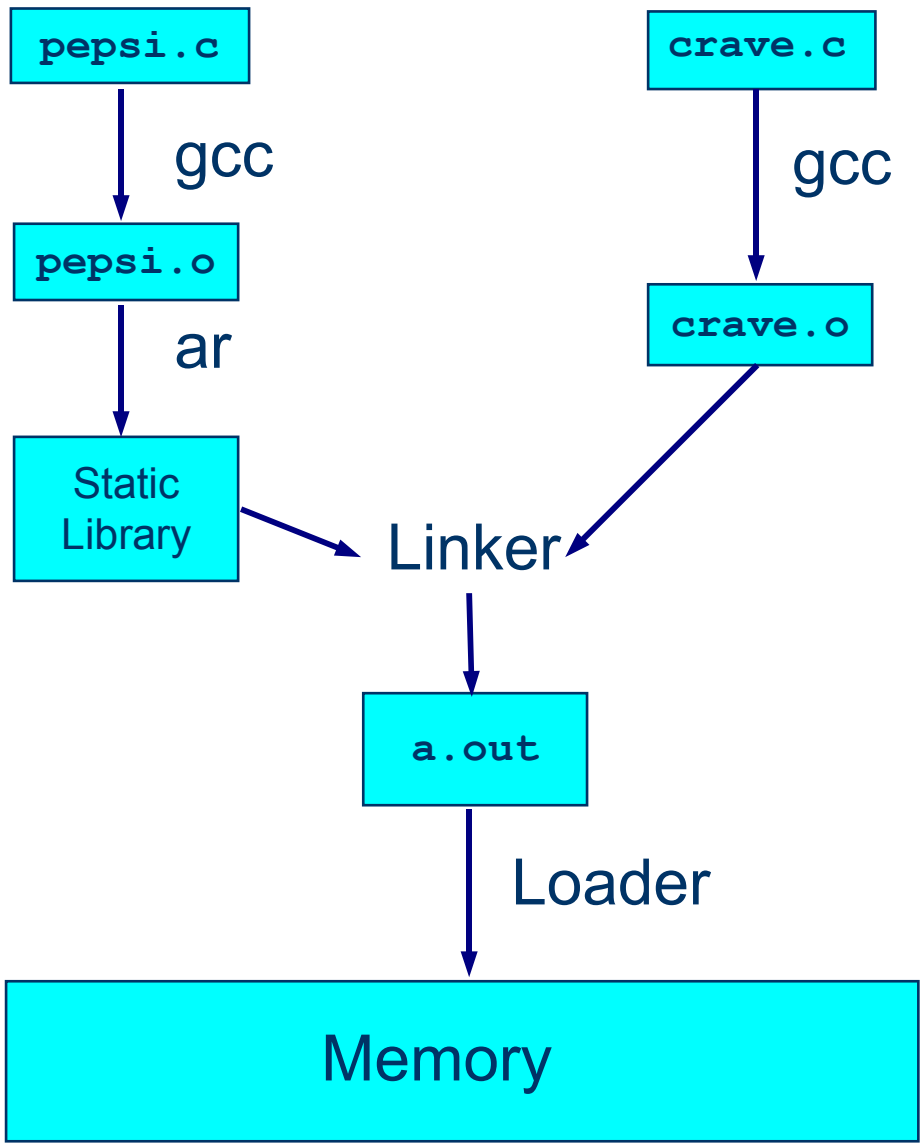
  - e.g.: **`gcc file1.c file2.c -lstop -lwalk`**
    may cause similar error if **`libwalk.a`** references functions in **`libstop.a`**

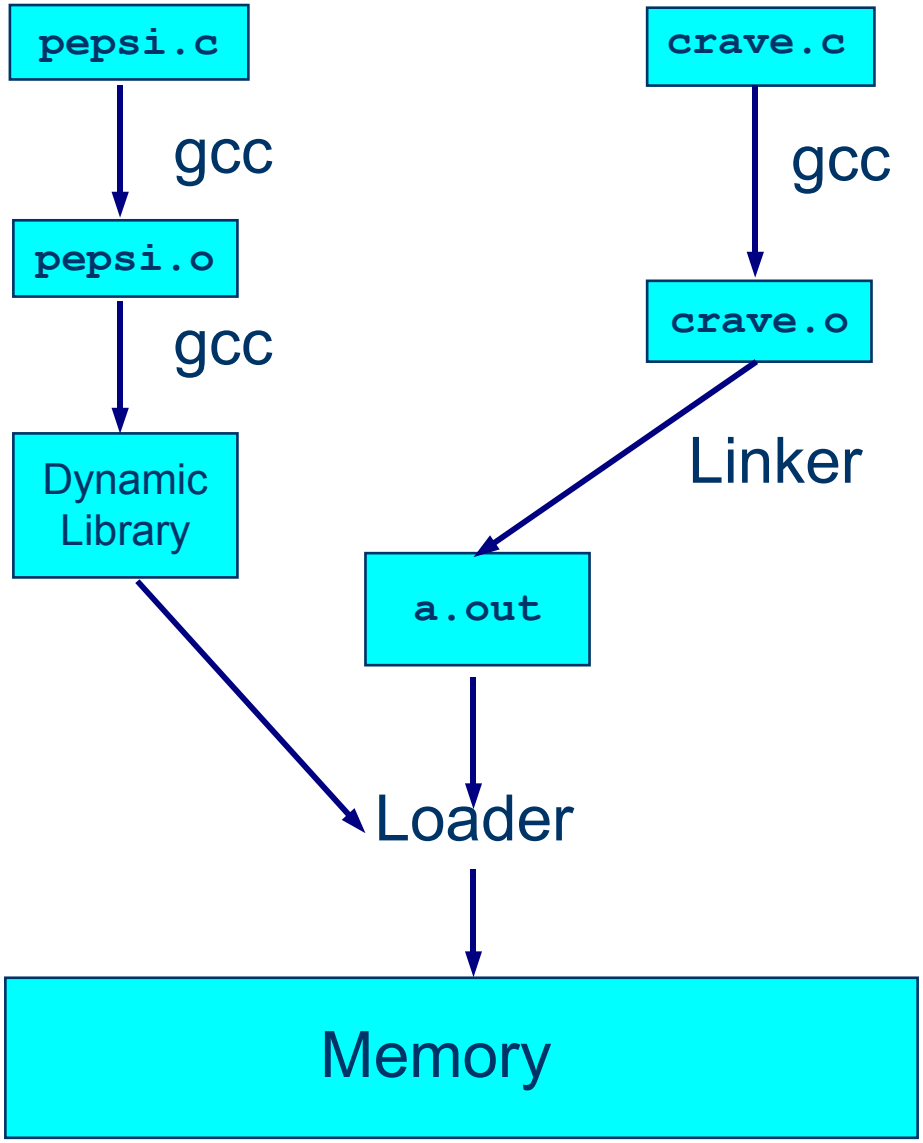
gcc's library options

- **`-Ldir`**
  - add dir to list of search directories
  - e.g.: **`gcc -o crave -L/home/libs crave.c -lpepsi`**
  - Linker searches **`/home/libs`** first, then standard directories (**`/usr/lib`**, **`/lib`** on Linux)
  - Can override standard libraries with local versions

  - **`-nostdlib`** : don't use standard libraries
  - **`-static`** : link only static libraries
  - **`-shared`** : use shared libraries when possible
  - **`-u`*`symbol`*** : pretend symbol undefined to force loading of module

Static vs. Dynamic Linking





Creating Dynamic Library

- Creating the library
  - `gcc -fPIC -Wall -pedantic -c findpepsi.c`
  - `gcc -fPIC -Wall -pedantic -c drinkpepsi.c`
  - `gcc -fPIC -Wall -pedantic -c cravepepsi.c`
  - `gcc -shared -Wl,-soname,libpepsi.so -o libpepsi.so findpepsi.o drinkpepsi.o cravepepsi.o`
- Using the library
  - Linker searches for shared libraries in the predefined system directories
  - To search for a shared library in a different directory, set **`LD_LIBRARY_PATH`** environment variable to point to this directory
    - e.g. **`export LD_LIBRARY_PATH=.`**

Summary (linking + libraries)

- Linking
  - combines code from multiple files into a single executable
  - resolves external references
- Libraries
  - provide “services” to programs when needed
- Together, linking and libraries promote modularity and reuse of code