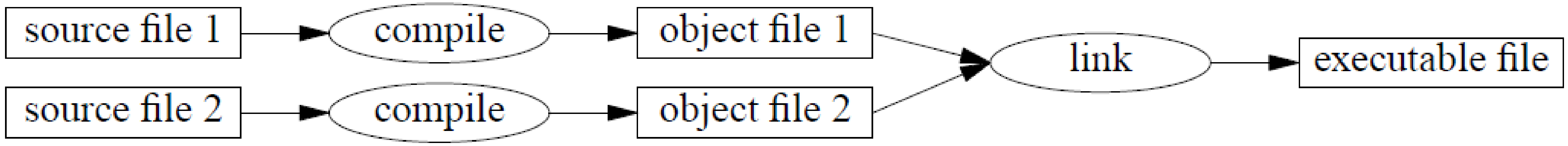
תכנות מתקדם
מצגת 7

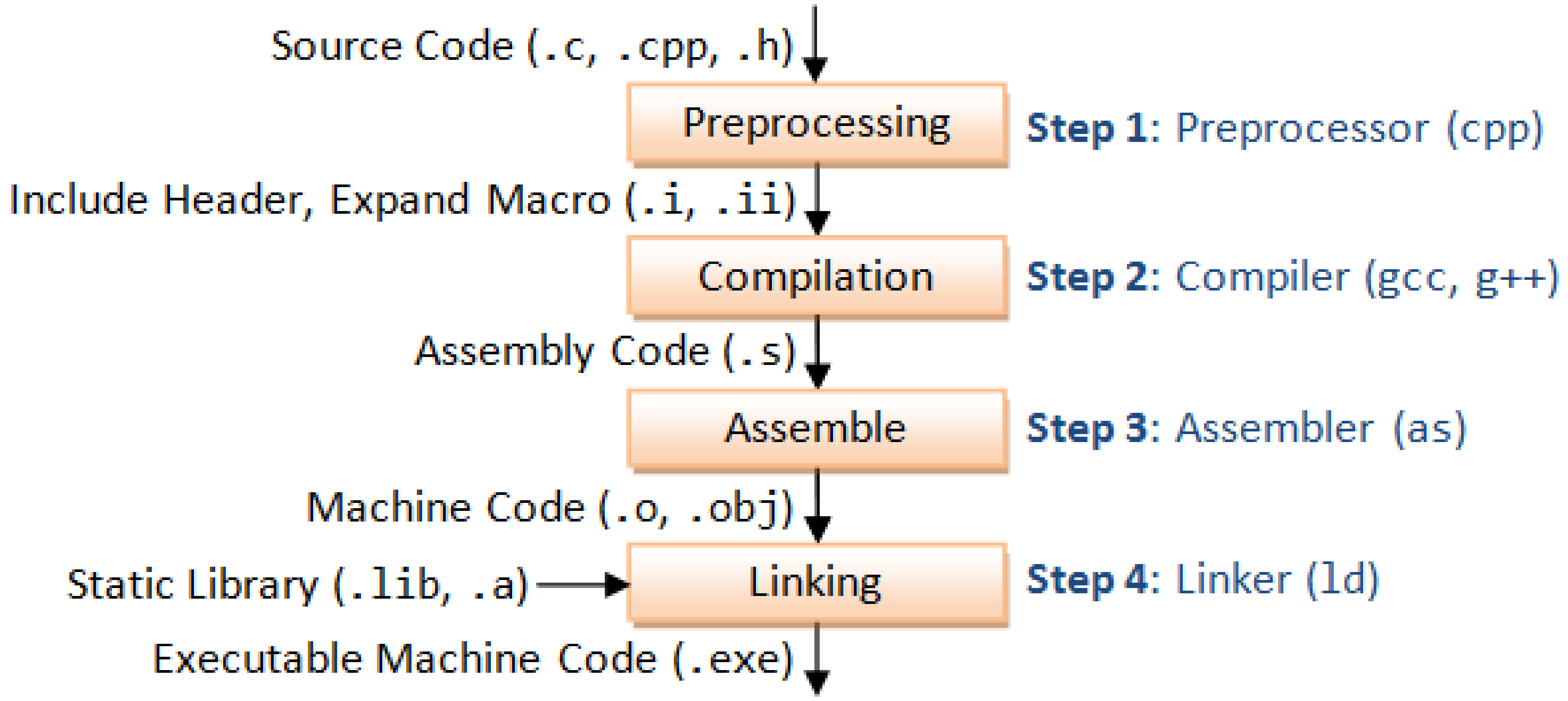
בניית תכנית

הידור וקישור

- Large programs can be divided into smaller files and compiled in isolation from the rest of the program
- The compiler then generates, for each source file, an **object** file
- The object file contains machine code, it typically has an extension .o
- object files (and libraries) are combined into an executable file by the **linker**



הידור וקישור



Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

```
gcc -save-temps hello.cpp
```

הצהרה והגדרה

- A **declaration** informs the compiler that a variable or function is defined elsewhere and describes their proper **usage**
- A variable or a function can be **declared many times**
- The **definition** of a variable or a function **allocates storage** for the variable or the function code
- There can be only **one definition** for a variable or a function

declare1.cpp, declare2.cpp

קבצי כותרת

- Every name must be **declared** or **defined** before it is used
- To compile a separate file, the compiler must know about the data and functions that are defined in other files
- Programs made up of multiple files, need a centralized location for related declarations
- Headers files normally contain **extern variable** declarations, **function** declarations and **class** definitions
- Files that **use** or **define** these entities have to include the header
- Files that **include** the header are guaranteed to use the **same declaration** for a given entity
- Should a declaration require change, only the header needs to be updated

קבצי כותרת

- A header should **not** contain **data definitions** or **function definitions**:

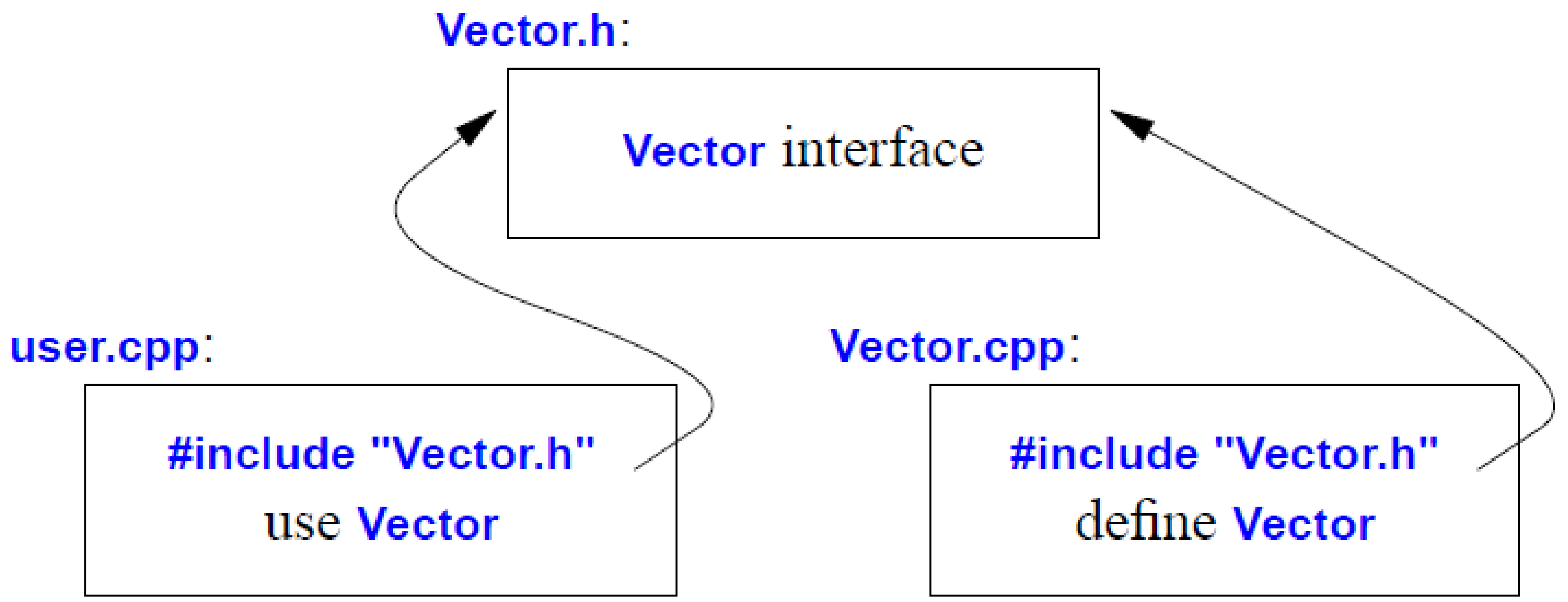
```
int a;
char get(char* p) { return *p; }
```

- A **class definition** may be included in multiple source files:

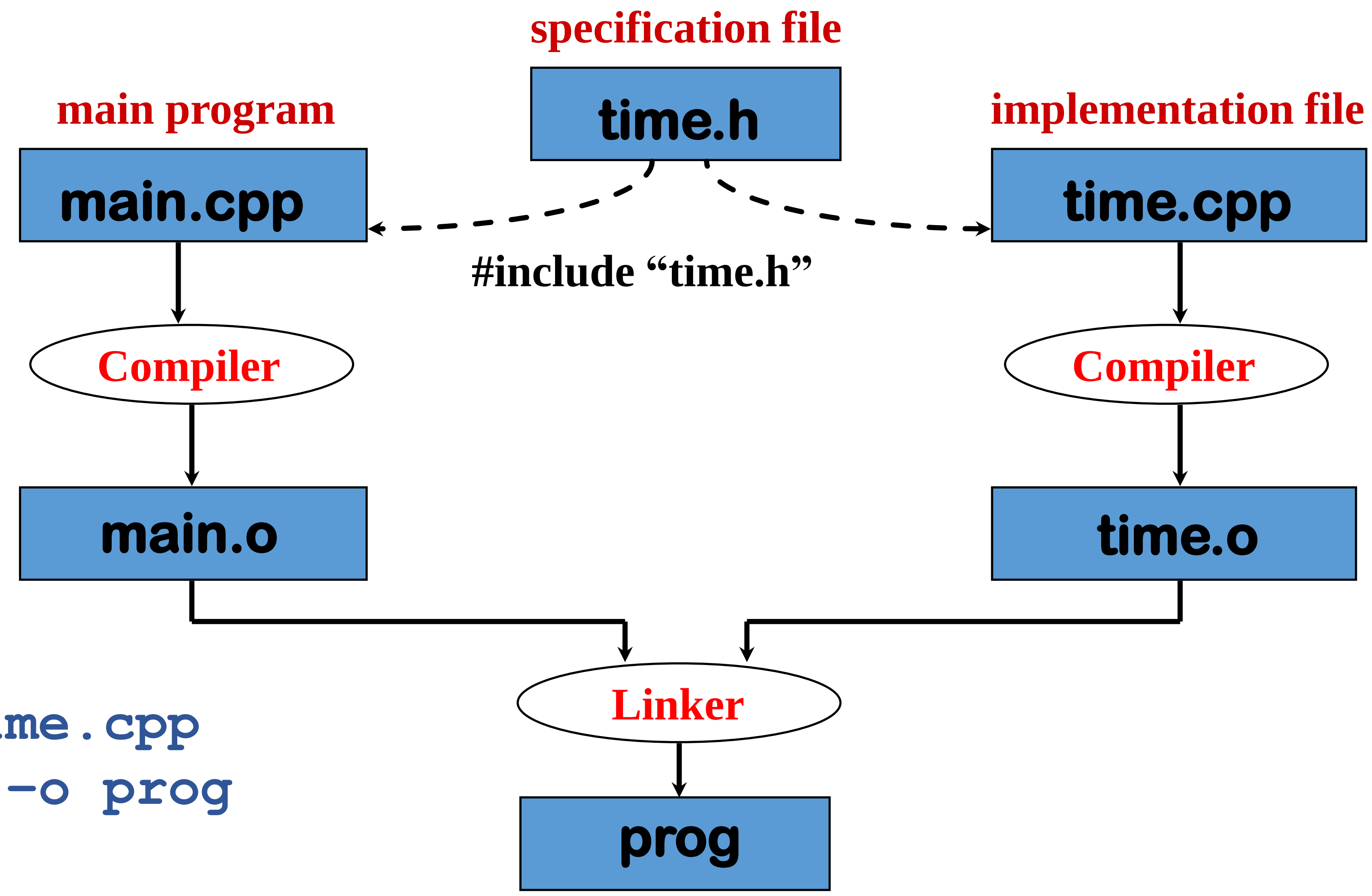
```
// s.h:
class S { int a; char b; };
// file1.cpp:
#include "s.h"
// file2.cpp:
#include "s.h"
```

קבצי כותרת

- The declarations that specify the interface to a class are placed in a header file
- Users will include that file, to access that interface
- To ensure consistency, the .cpp file providing the implementation will also include the .h file providing its interface



הידור וקישור עם קובץ כותרת



```
g++ -c main.cpp time.cpp
g++ main.o time.o -o prog
```
Or
```
g++ main.cpp time.cpp -o prog
```

הכלת קבצי כותרת ומניעת הכלות כפולות

- The #include directive takes one of two forms:

```
#include <iostream> // standard-library headers
                      // .h suffix not needed
#include "my_file.h"      // header supplied by program
```

- Headers often #include other headers which may cause multiple definitions
- To guard against including a header more than once, preprocessor variables are used
- A preprocessor variable has two states, defined or not yet defined

```
#ifndef SALESITEM_H
#define SALESITEM_H
// Definition of Sales_item class and related functions
#endif
```

- Or

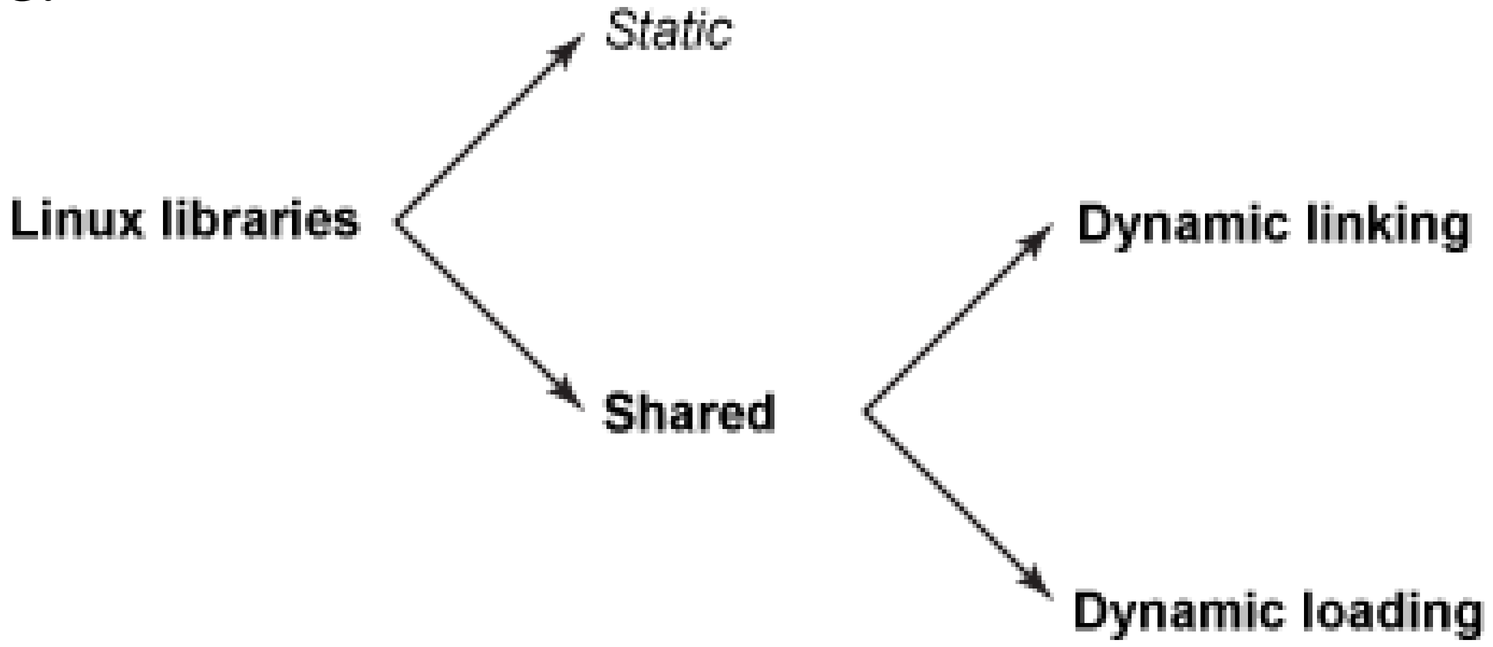
```
#pragma once  // directive to compiler
// Definition of Sales_item class and related functions
```

למה ספריות ?

- How to package functions commonly used by programmers?
    - Math, I/O, memory management, string manipulation, etc.
- **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
        - Space and time inefficient
- **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
        - More efficient, but burdensome on the programmer

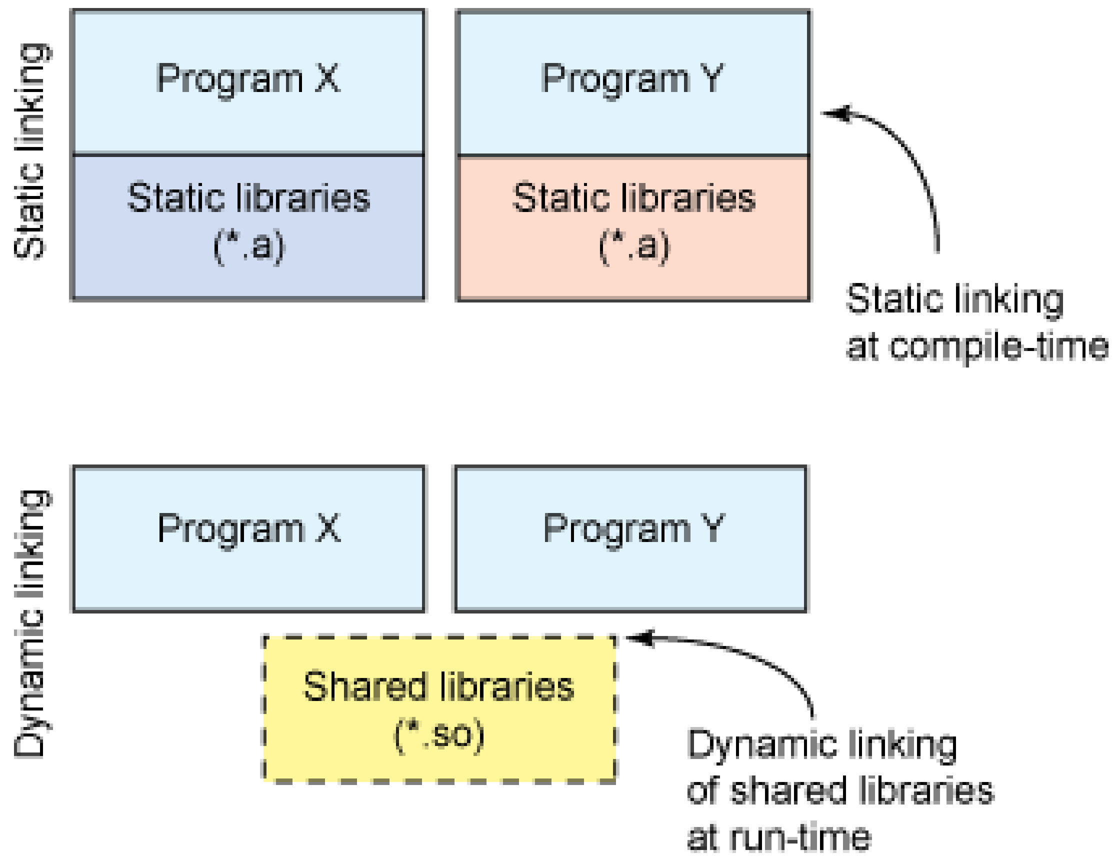
ספריות סטטיות

- Concatenate related relocatable object files into a single file with an index (archive file, `.a` suffix)
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link it into the executable.



ספריות סטטיות - חסרונות

- Duplication in the stored executables (every function needs libc)
- Duplication in the running executables
- Bug fixes of system libraries require each application to relink



ספריות דינמיות

- Modern solution: Shared Libraries
  - Object files that contain code and data that are loaded and linked into an application *dynamically*
  - Also called: dynamic link libraries, DLLs, `.so` files
- Dynamic linking can occur when executable is first loaded and run (load-time linking).
- Dynamic linking can also occur after program has begun (run-time linking).
- Shared library routines can be shared by multiple processes at run-time

בניית תכנית באמצעות make

- **make** is a utility for building projects having multiple files
- **make** compiles only those modules that have changed and the modules that depend upon them
- To use **make**, a file named **makefile** (or **Makefile)** has to be prepared, the file describes the relationships among files, and the commands for updating each file
- The **make** command reads the file named **makefile** and perform all necessary recompilations
- The **make** program uses the **makefile** and the last modification times of the files to decide which of the files need to be updated
- For each of those files, it issues the commands recorded in the **makefile**

כללי הקובץ makefile

- makefile consists of **rules** with the following format:

```
target: prerequisites
[TAB]   command
```

- Example

```
myprog: myprog.cpp myprog.h
[TAB] g++ -o myprog myprog.cpp
clean:
[TAB] rm myprog
```

- make updates a **target** that depends on **prerequisite** files if any of them have been **modified** since the target was last modified, or if the target **does not exist**

כללי הקובץ makefile

- By default, **make** starts with the first rule, thus:

```
make
```

- with no arguments , reads the **makefile** in the current directory and processes the **first rule**
- But before make can process this rule, it must check if the prerequisite files depend on other files, and process the rules for them
- Thus, other rules are processed because their targets appear as prerequisites of the goal
- If some other rule is not depended on by the goal, that rule is not processed, unless you tell make to do so, e.g.:

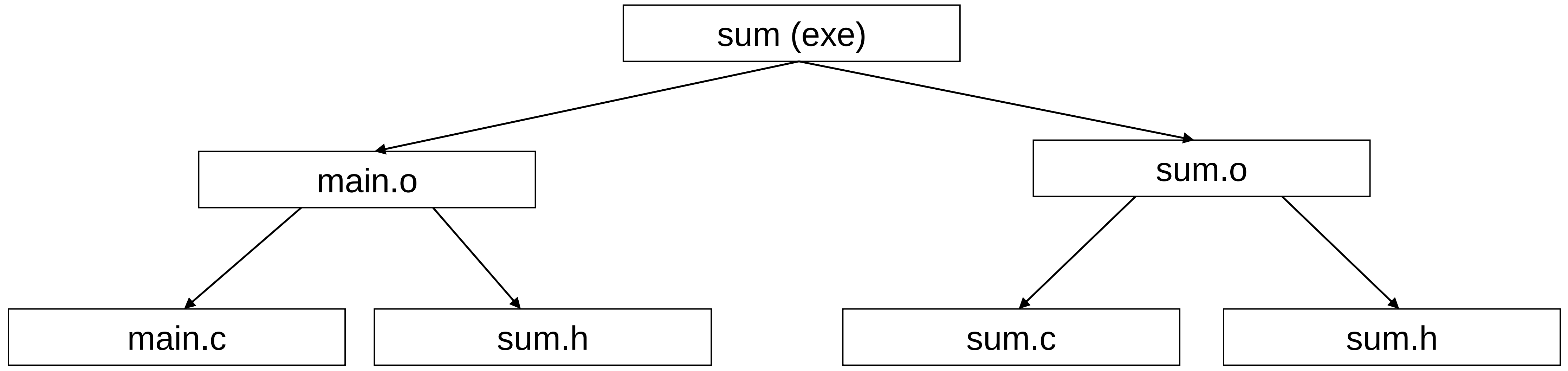
```
make clean
```

דוגמה לקובץ makefile

```
sum: main.o sum.o
	g++ -o sum main.o sum.o
main.o: main.cpp sum.h
	g++ -c main.cpp
sum.o: sum.cpp sum.h
	g++ -c sum.cpp
clean:
	rm sum main.o sum.o
```

גרף הקדימות של make

- In order to ensure that rebuilding commands are executed in the correct order, make constructs a Precedence Graph
- The Graph is processed using a depth first search starting at the root



משתנים של הקובץ makefile

- In our example, we had to list all the object files twice in the rule for **sum** and also in the rule for **clean**
- Such duplication is error-prone, we can simplify the makefile by using a variable
- We would define a **variable** OBJECTS:

```
OBJECTS = main.o sum.o
```

- Then, to put a list of the object file names, we can write $(OBJECTS)

דוגמה לקובץ makefile עם משתנים

```
OBJFILES = main.o sum.o
PROGRAM = sum
CC = g++
CFLAGS = -g -std=c++11 -Wall

$(PROGRAM): $(OBJFILES)
	$(CC) $(CFLAGS) -o $(PROGRAM) $(OBJFILES)
. . .
clean:
	rm $(PROGRAM) $(OBJFILES)
```