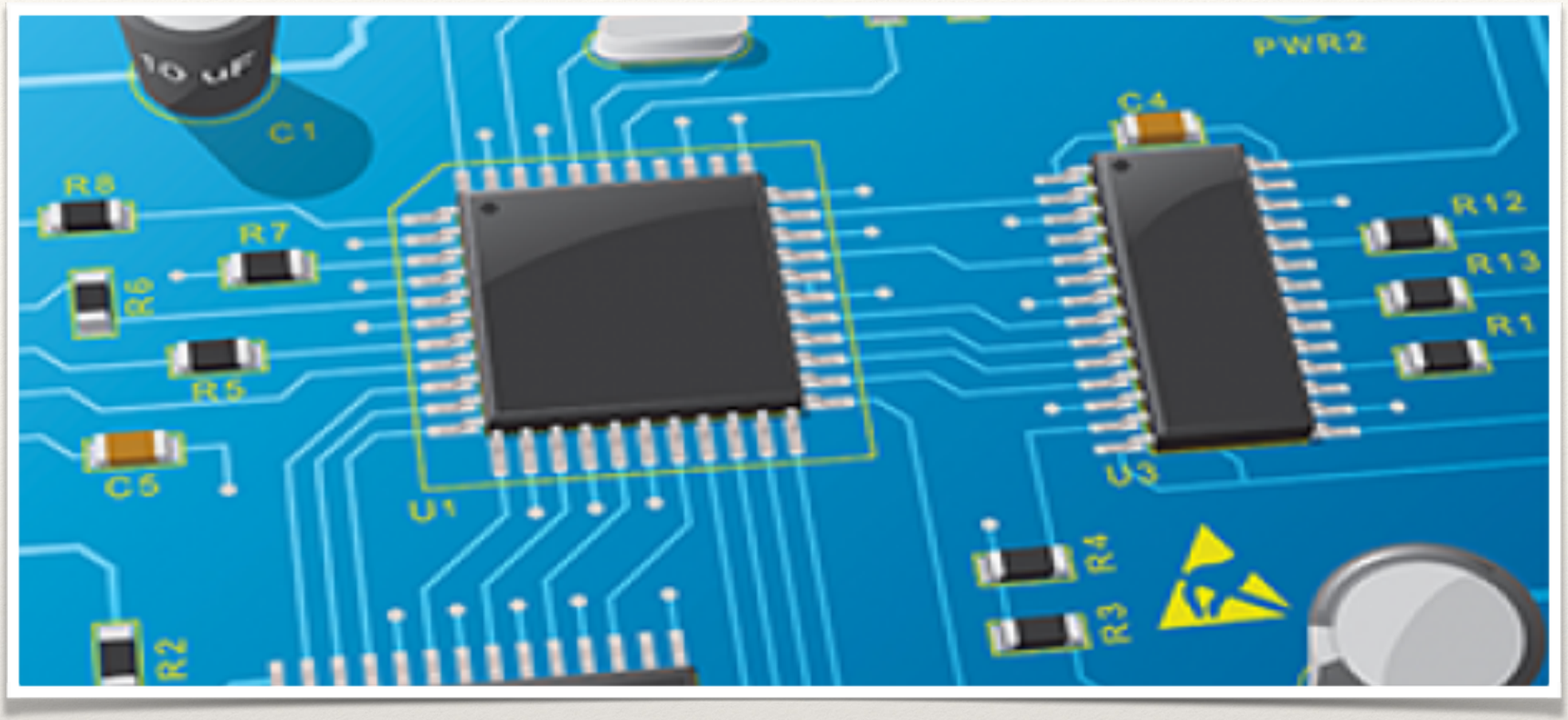
Let's start at the beginning…

Interacting with the Computer

Computer System
Operating System
Compilation

https://d3njjcbhbojbot.cloudfront.net/api/utilities/v1/imageproxy/https://coursera.s3.amazonaws.com/topics/comparch/large-icon.png

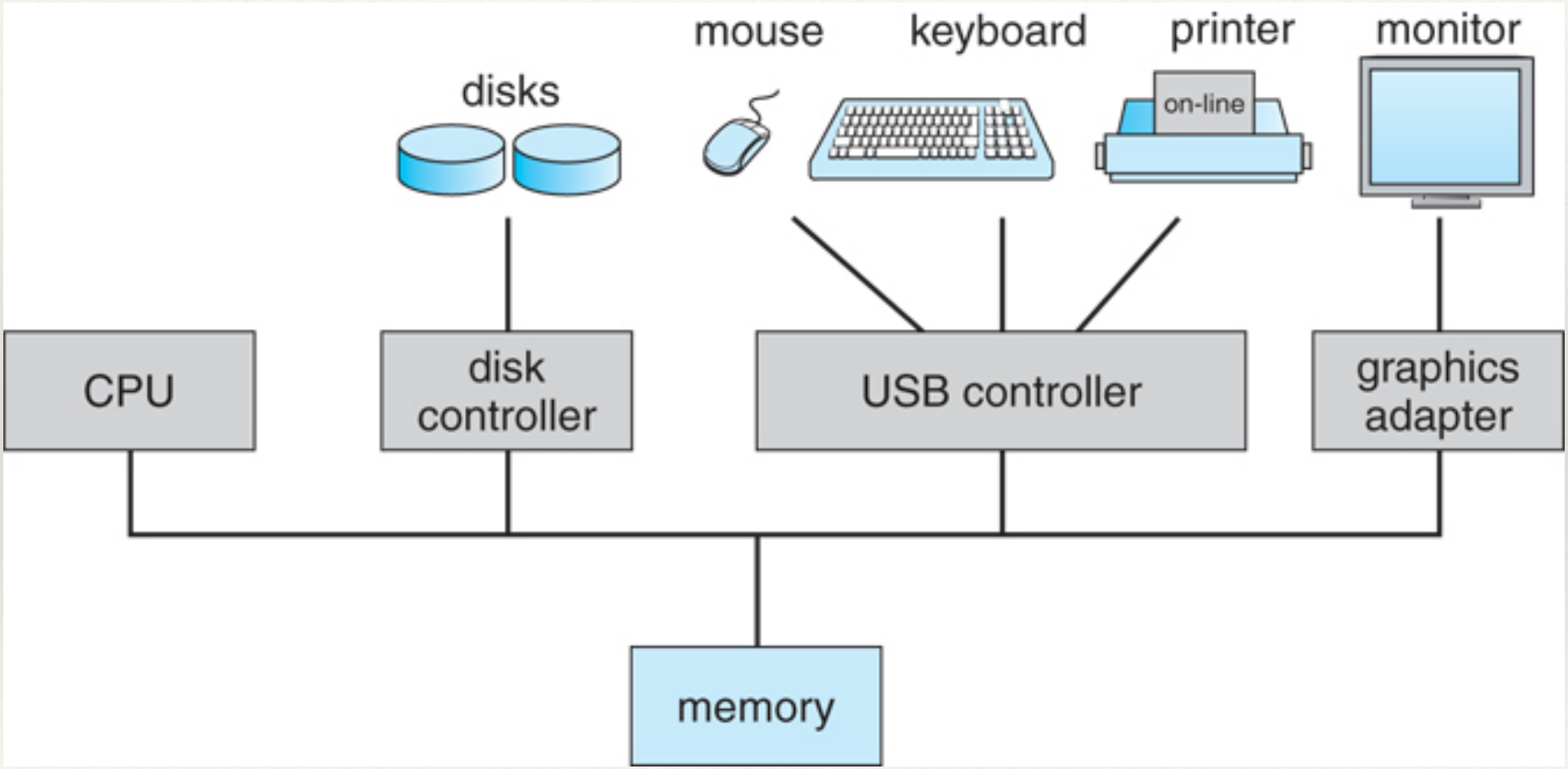


Hardware Structure

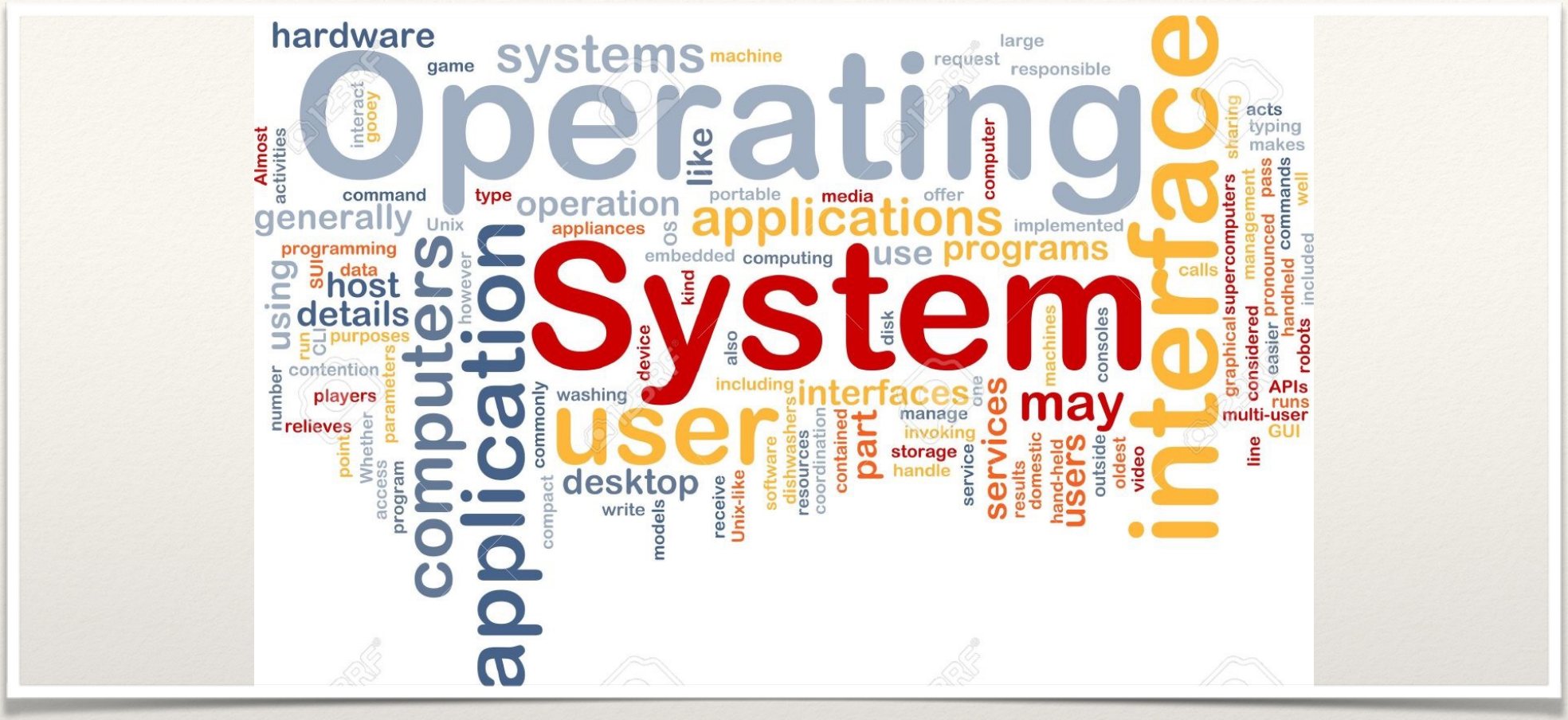
Computer System

Computer Architecture

What is a Computer?



http://previews.123rf.com/images/kgtoh/kgtoh0910/kgtoh091001789/5753313-Word-cloud-concept-illustration-of-operating-system-Stock-Illustration.jpg

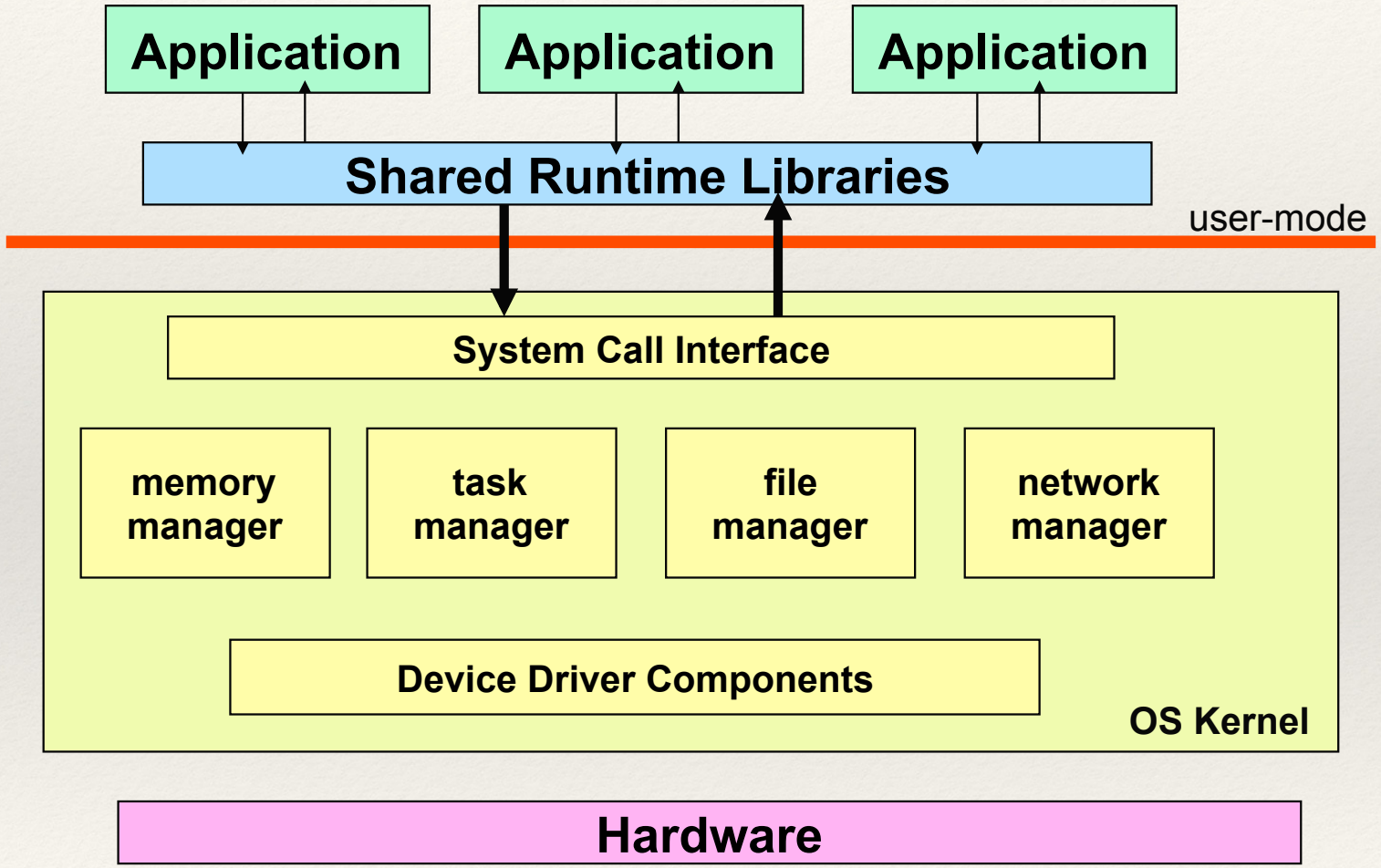


The bridge between hardware and software

Operating System

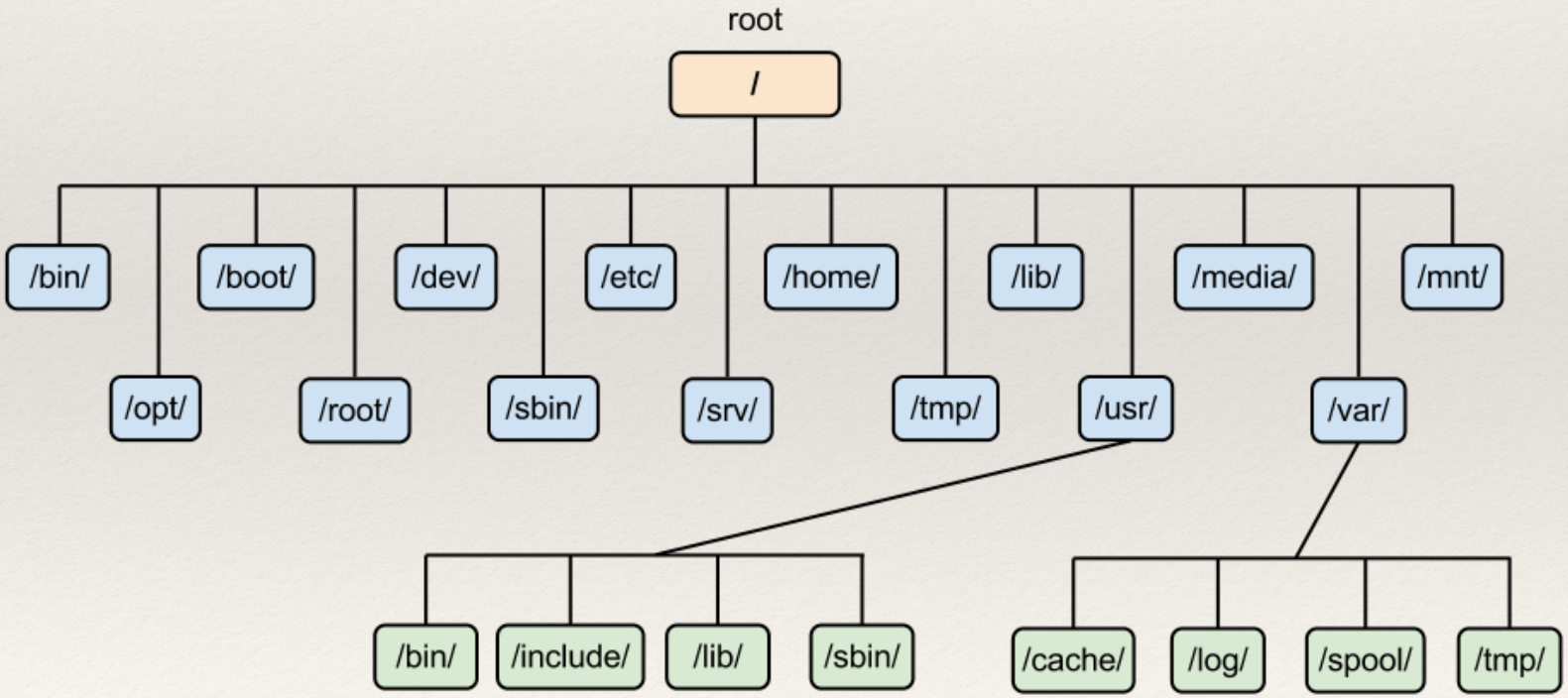
Unix and Unix-like systems

What is an Operating System?



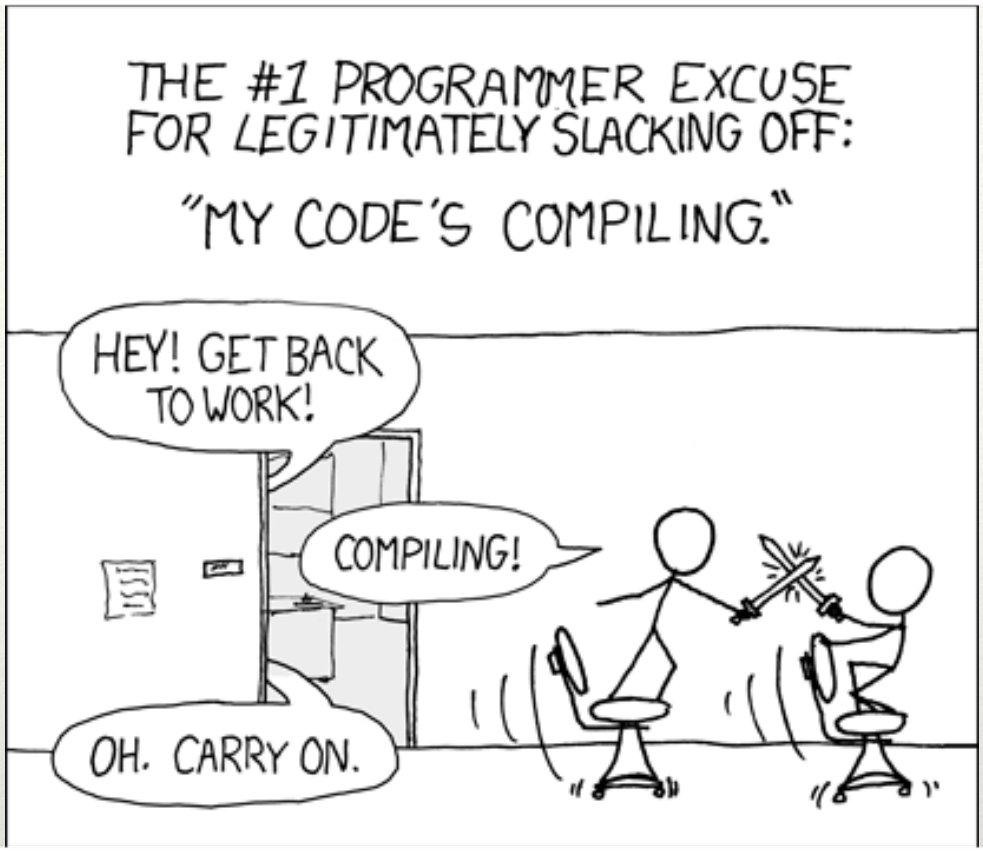
The Linux File System

- It is organized in a hierarchical (tree) structure.
- The top-most point is the **root** of the tree.



https://freedompenguin.com/wp-content/uploads/2015/10/linux-filesystem.png

http://imgs.xkcd.com/comics/compiling.png

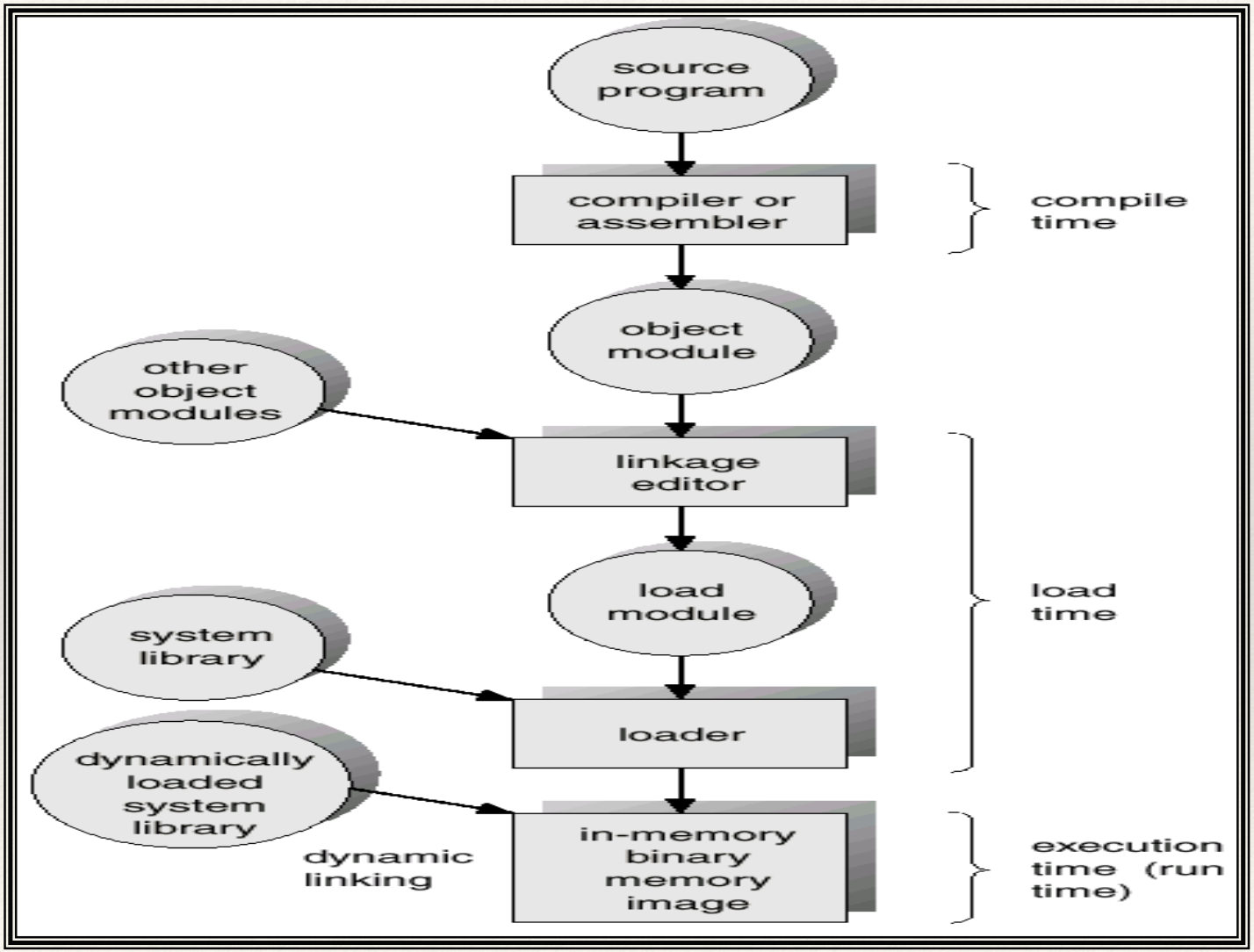


Source code to executable code…

Compilation

Compilers
Linking
Libraries

Code: From Source to Execution



What is a Compiler?

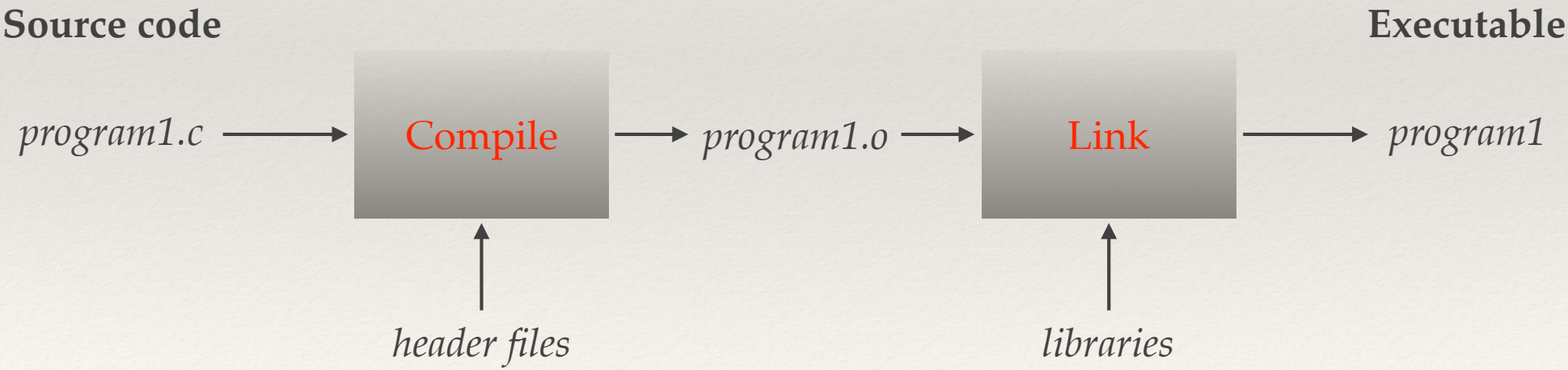
- A compiler converts source code (programming language) into binary (machine readable) form.

```
$ gcc program1.c -o program1
```

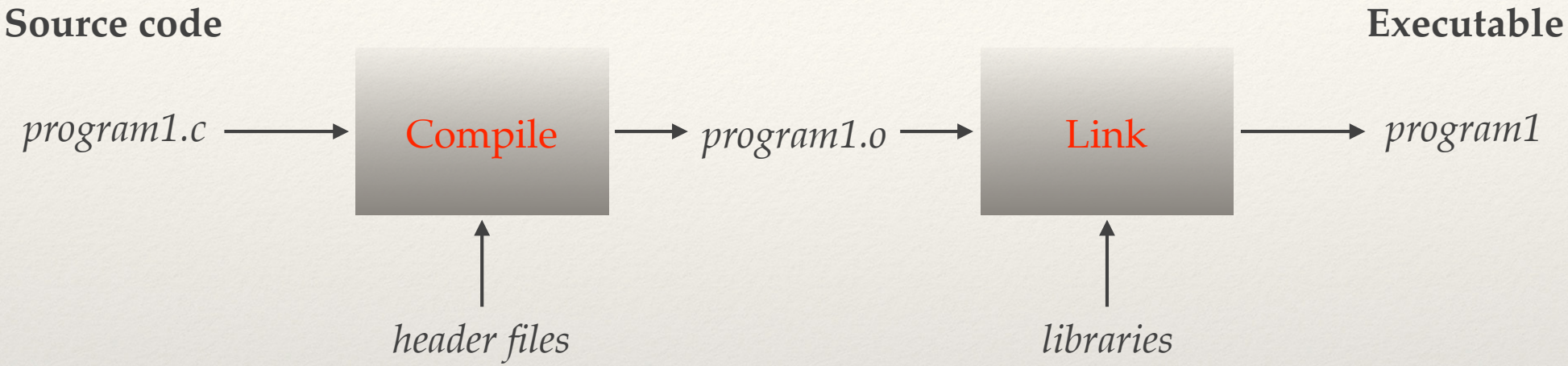
- `-o` means name the executable output of the compiler `program1`
- Without `-o` the executable output is named `a.out`
- The name of the executable is not required to have a particular extension (*i.e.* `program1.exe`) in Unix/Linux systems.

Operations of the Compiler

- The compiler performs two operations:
  - compiling
  - linking

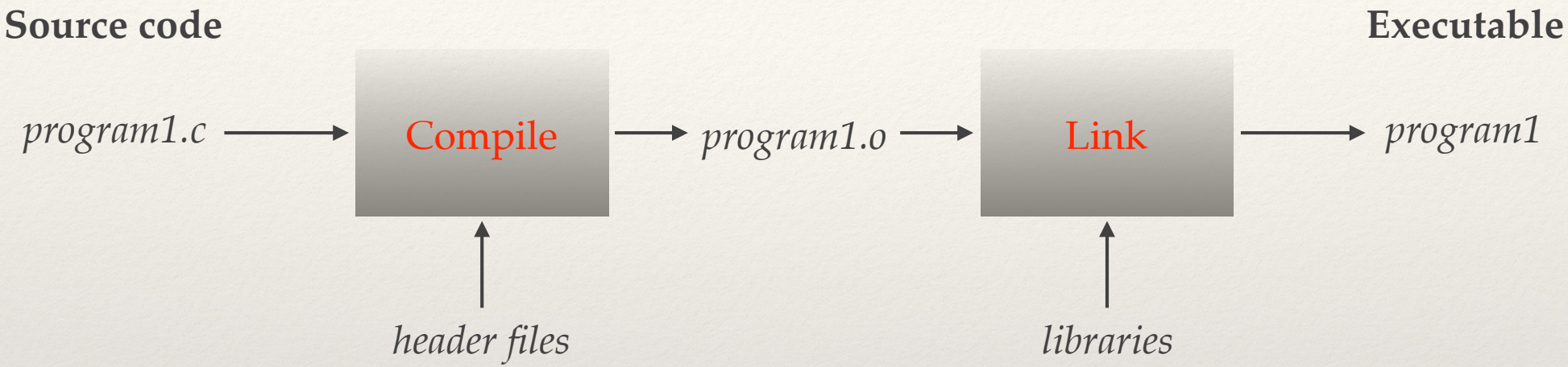


Operations of the Compiler



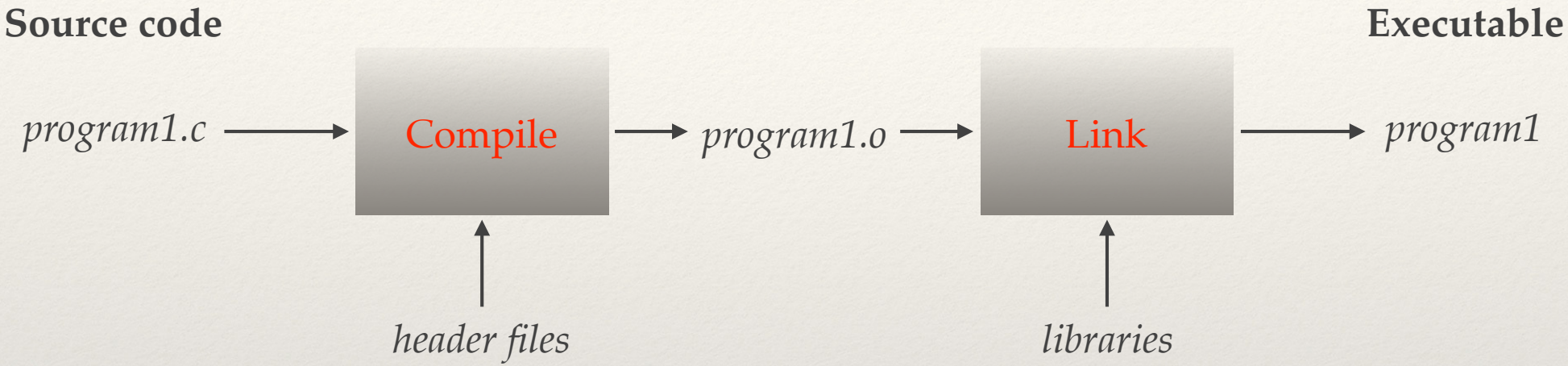
- Header files describe the contents of a library.
- This includes data structures, variables, and functions.
- This description is required during compilation so the compiler knows what the library calls look like.

Operations of the Linker



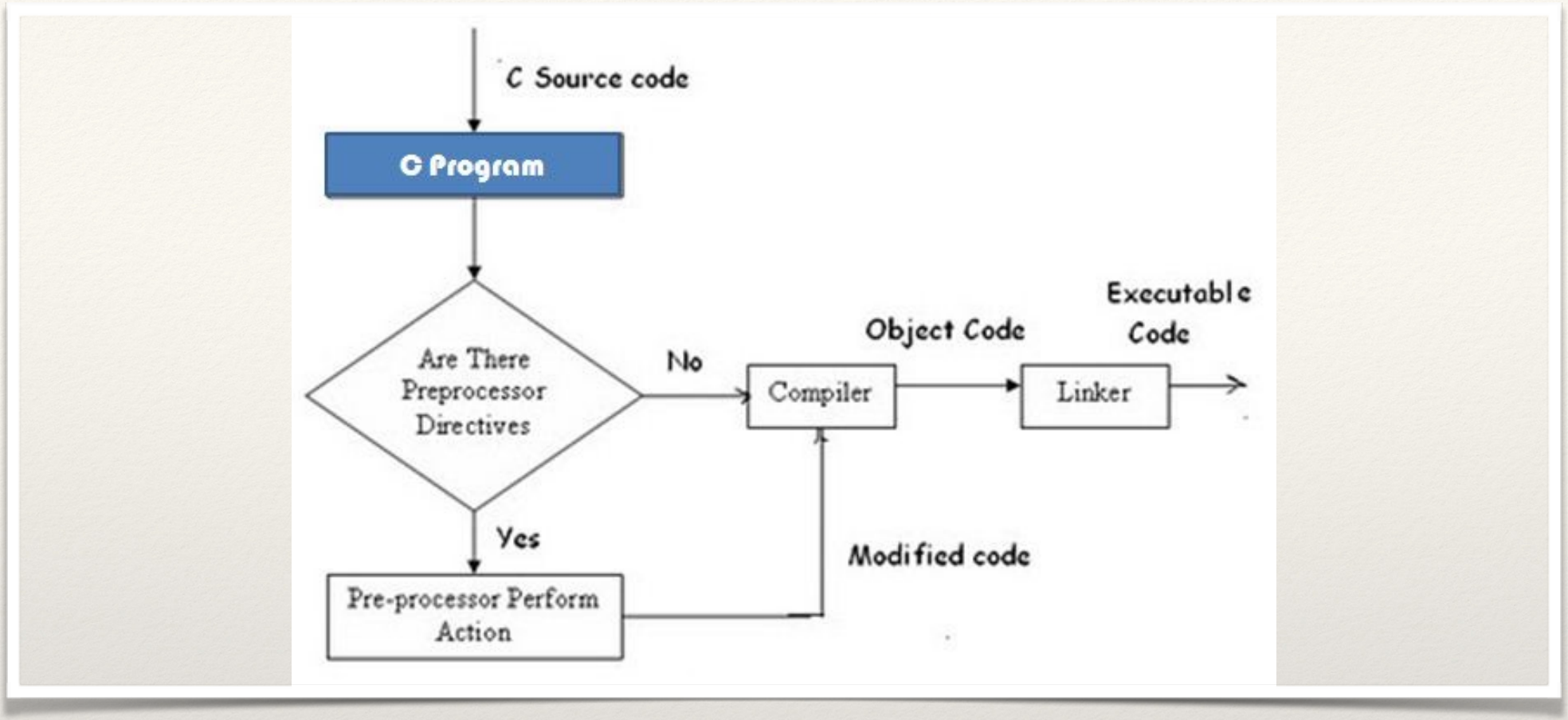
- The linker takes your compiled code and the libraries and combines them into a single executable program.
- Your program is a combination of your compiled source code and the libraries you utilized.

Operations of the Linker



- Libraries provide many convenient functions that you need and do not want to write yourself.
- Examples include I/O (`printf`, `fgets`), math functions, string manipulation.

http://img.c4learn.com/2012/01/C-Preprocessor.jpg



Constructing a C program…

The C Preprocessor

Macros
Includes
Conditional Compilation

Text: Chapter 4.11

The C Preprocessor

- The preprocessor is a program that runs **before** the compiler.
- It performs three (3) operations all of which involve replacing something in the source code before compilation.
  - Macro Processing
  - Includes
  - Conditional Compilation
- *https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html*

Macro Processing

- Macro processing substitutes values and operations in the code.

```
#define LINE_LENGTH 1024
```

  - Anywhere in your code where you use `LINE_LENGTH` will be replaced by `1024`.
  - Macros are usually in UPPER CASE
  - Anything beginning with # (number sign) is a macro.

Macro Processing Examples

- **Example**: Replace an operation - the **square** macro.

```
#define SQUARE(a)   ((a) * (a))
```

- **Question**: What does the following do?

```
#define M(a,b)  ((a) < (b) ? (a):(b))
```

- https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html#SEC24 explains why this is not a *"**safe**"* macro definition if an argument, when called, contains a ***side effect***.

Macro Processing Examples

```
#define SQUARE(a)   (a * a)

/* used code */
SQUARE(x + 5)

/* turns into */
x + 5 * x + 5
```

which is $6x + 5$, *not* $(x + 5)^2$

contains a ***side effect***.

Macro Processing

- **Benefit** to replacing an operation with a macro:
  - All code is placed into your program - there is not any overhead as there would be with a **function call**.
- **Warning**
  - Make sure that you define and use the macro correctly or you may experience unpredictable results when it is expanded. You must consider side effects.

Includes

- You can include C header files in your program.

```
#include <stdio.h>
```

- **Header files** store function definitions and variables that must be shared over multiple files.
- There are 2 ways to write includes:
  - `#include <file>`
  - `#include “file”`

https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html#SEC4

Includes: Header Files

```
#include <file>
```

- This is used for system header files.
- It searches for a file named `file` in a standard list of system directories.

```
#include “file"
```

- This is used for header files of your own program.
- It searches for a file named file first in the current directory, then in the same directories used for system header files.

Conditional Compilation

- Conditional compilation allows/disallows the compilation of sections of a program.
- This is especially useful for enclosing testing code so that it is only included when testing and not in the production code.
- It can also be used for **portability** - different code can be activated on different systems.

Conditional Compilation

```
#if expression
controlled text
#endif
```

- *expression* is a C expression of type integer.

```
#if 0                                   0 is NOT TRUE
 printf ( “Hello\n” );   Code will not be compiled.
#endif
```

Conditional Compilation

- **Example**: *Debugging*
  - Adding code to your program that is only executed during testing.

```
#define DEBUG 0
#if DEBUG
  …testing code…
#endif
```

Set to 0 or 1 to turn debugging OFF or ON

Conditional Compilation

- If the value of `DEBUG` is set to **TRUE** (*i.e.* not equal to 0) then the code between the `#if` and `#endif` will be compiled.
- If `DEBUG == 0` (**FALSE**) then the code will not be included during compilation.
- In the example, `DEBUG` is a macro.
- There are two ways to set the value of a macro:
  - In the program
  - During compilation (command line)

Conditional Compilation

- Setting the value in the program using #define

```
#define DEBUG 0
#define DEBUG 1
```

- Setting the value during compilation using the -D flag

```
gcc myProgram.c -o myProgram -DDEBUG=0
gcc myProgram.c -o myProgram -DDEBUG=1
```

Conditional Compilation

- Conditionals that test whether just one name is defined are very common so there is a short form:

  `#ifdef name`

  - is equivalent to `#if defined (name)`

  `#ifndef name`

  - is equivalent to `#if ! defined (name)`

Conditional Compilation

```
#ifdef DEBUG
printf(“This is a debugging statement");
    ...
#endif
```

- The `#ifdef` directive specifies that if `DEBUG` exists as a defined macro then the statements between the `#ifdef` directive and the `#endif` directive are retained.
- To turn on debugging statements, include a definition:

```
#define DEBUG  1
```

Conditional Compilation

- Let's look at the debugging example again

```
#ifdef DEBUG
… testing code …
#endif
```

- To turn **on** debugging:

```
gcc myProgram.c -o myProgram -DDEBUG
```

- To turn **off** debugging:

```
gcc myProgram.c -o myProgram
```

Conditional Compilation

```
#include <stdio.h>
#define DEBUG 0
int main ()
{
   printf ( "This is before the debugging code\n" );
#if DEBUG
   printf ( "Debug statement\n" );
#endif
   printf ( "This is after the debugging code\n" );
}
```

testingCode0.c and testingCode1.c

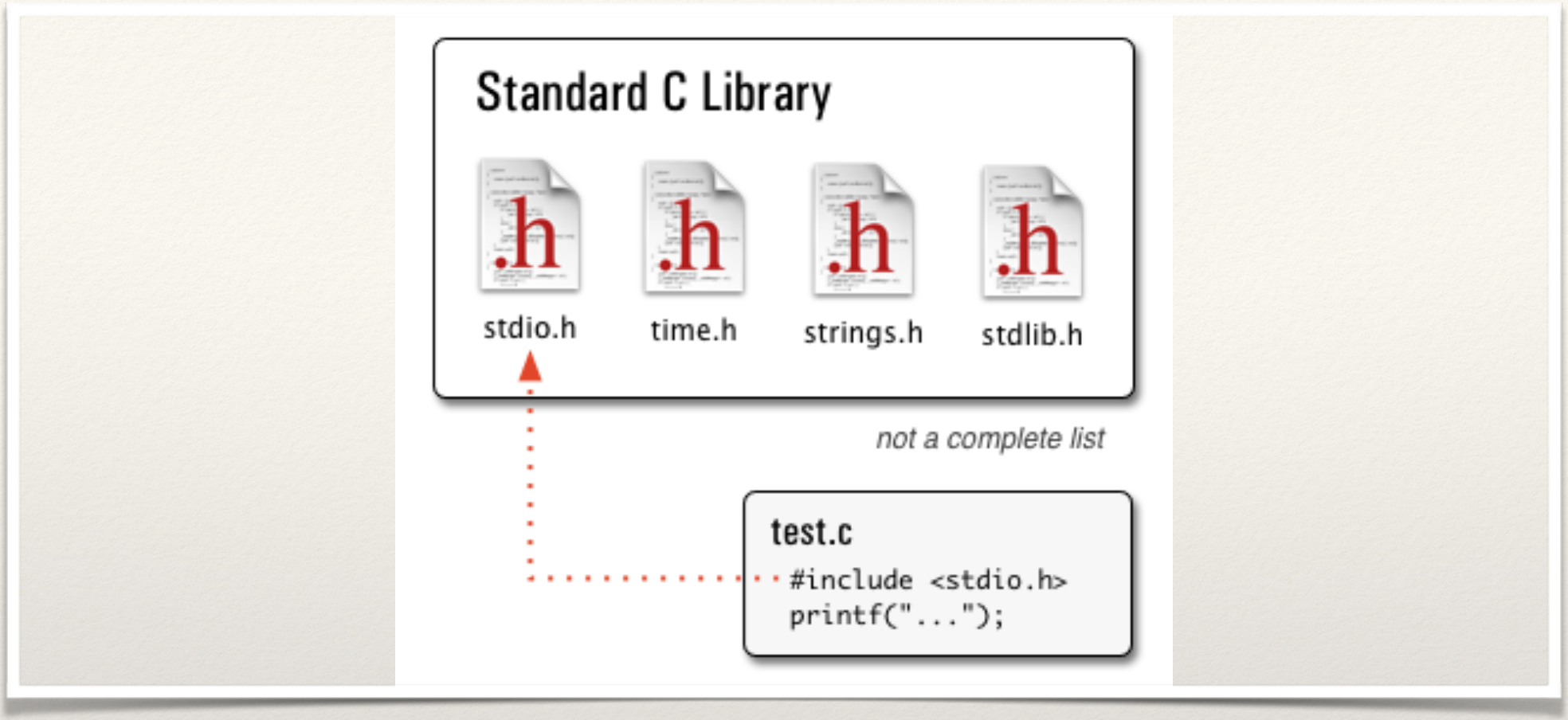
```
$ gcc testingCode0.c -o testingCode
$ gcc testingCode1.c -o testingCode
```

Conditional Compilation

```
#include <stdio.h>
int main ()
{
   printf ( "This is before the debugging code\n" );
#if DEBUG
   printf ( "Debug statement\n" );
#endif
   printf ( "This is after the debugging code\n" );
}
```

```
$ gcc testingCode.c -o testingCode -DDEBUG=1
$ gcc testingCode.c -o testingCode -DDEBUG=0
```

http://cocoadevcentral.com/images/articles/000081-libraries.png



More than just your code…

Libraries

Static
Dynamic

Text: Appendix B: Standard Library

Libraries

- A **library** is a collection of *precompiled* functions.
- Libraries are:
  - Reusable
  - Usually stored in `/lib` and `/usr/lib` on a Unix system

Libraries

- Always have an associated **header** file (`.h`) that contains constants and function definitions
  - Usually in `/usr/include`
  - Headers let the compiler know the structure of the functions.
  - This includes name, parameter number and types and return types.

Types of Libraries

- There are two types of libraries:
  - **Static Library**
    - When a program is compiled the library contents are copied into the executable and stored within it.

Types of Libraries

- **Dynamic Library**
  - When the program is executed the library is connected (linked) to the executable in memory.
  - The library copy is not stored in the executable file.
  - Also called **Shared** Library.
  - Updating a dynamic library means that existing programs that use it will gain the benefits and detriments of the changes to the library.

Naming Libraries

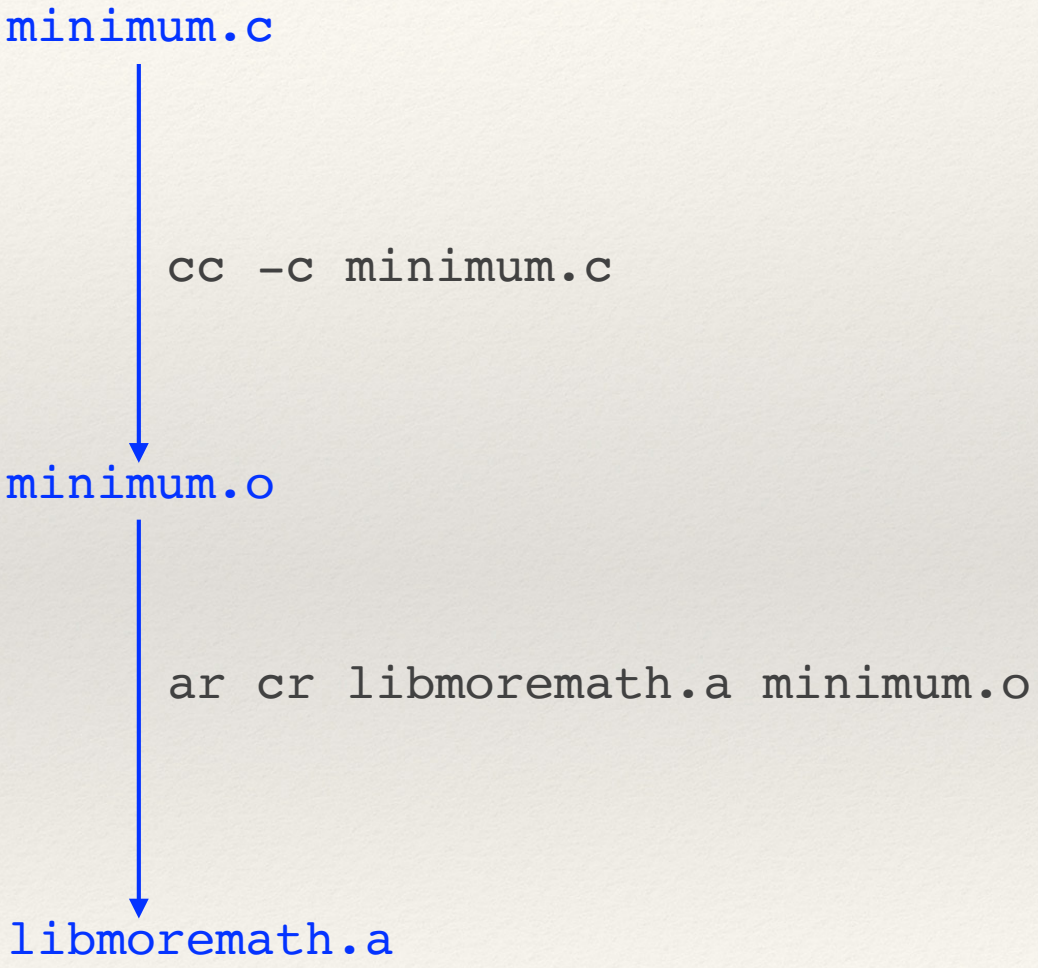
- All library names begin with `lib`.
- Static libraries end with `.a`
- Dynamic libraries end with `.so` (Unix: **s**hared **o**bject) or `.dll` (Windows: **d**ynamic **l**inked **l**ibrary).
- Shared objects can have version numbers after the `.so`; for example, `libm.so.3`, `libm.so.3.2` - versions 3 and 3.2 of the math library.

Naming Libraries

- Libraries often have multiple names created through the use of symbolic links.
- *E.g.* `libm.so.3.2` may be linked to the names `libm.so.3` and `libm.so` so that all three names point to the same library file.
- Some common library names:
  - `libm.so` - Math library
  - `libc.so` - Standard C library

Creating a Static Library

- **Source code** for library procedures
  - Does NOT contain a `main`
- Compile into an **object** file
  - `-c` means do not create an executable
- `.o` file contains compiled procedures
  - will be used to create library
- `cr` flags = create library and replace old .o's with new .o's



Creating and Using a Static Library

minimum.c

```
int minimum ( int numA, int numB )
{
   if ( numA < numB ) {
      return ( numA );
   } else {
      return ( numB );
   }
}
```

maximum.c

```
int maximum ( int numA, int numB )
{
   if ( numA > numB ) {
      return ( numA );
   } else {
      return ( numB );
   }
}
```

moremath.h

```
int minimum(int a,int b);
int maximum(int a,int b);
```

```
$ gcc -c minimum.c
$ gcc -c maximum.c
$ ar cr libmoremath.a minimum.o maximum.o
```

Creating and Using a Static Library

```
#include <stdio.h>
#include "moremath.h"

int main ()
{
   int numberA, numberB;

   numberA = 5;
   numberB = 8;

   printf ( "Number A is %d and Number B is %d\n”,
            numberA, numberB );

   printf ( "The minimum of these two numbers is %d\n”,
            minimum(numberA,numberB) );
   printf ( "The maximum of these two numbers is %d\n”,
            maximum(numberA,numberB) );
}
```

```
$ gcc moremathTest.c libmoremath.a -o moremathTest
```

Creating and Using a Static Library

```
$ ./moremathTest
Number A is 5 and Number B is 8
The minimum of these two numbers is 5
The maximum of these two numbers is 8
```

libmoremath.a	static library
moremath.h	header file associated with the library
minimum.c maximum.c	source code

```
$ gcc moremathTest.c libmoremath.a -o moremathTest
```

```
$ gcc moremathTest.c -L/home/socs/CIS2500 -lmoremath
   -o moremathTest
```

directory that contains the library

do not use `lib` *or* `.a`