Fall 2021
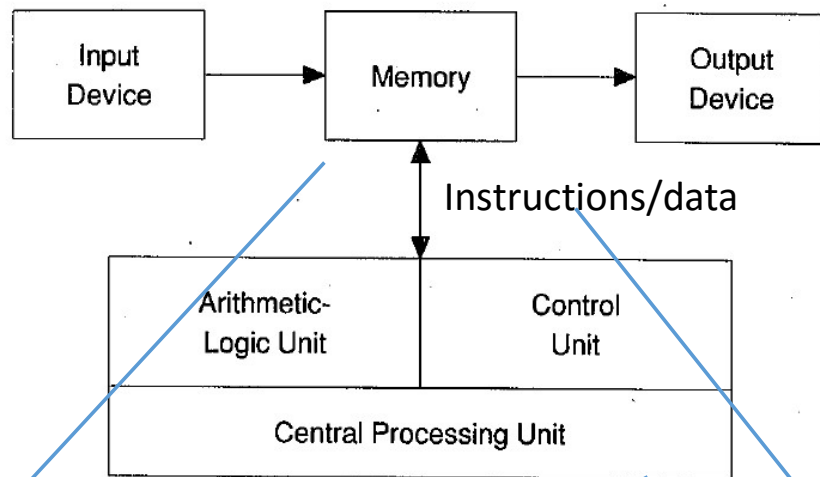# CENG 140
(Covid Online Education Semester)

Dr. İsmail Sengör Altıngövde

Principal Components of a Computer System

Input Device → Memory → Output Device

Instructions/data

Arithmetic-Logic Unit | Control Unit

Central Processing Unit

10110001

Opcode:
say, for ADD   operands

0

100 **10110001**
101 **01100011**
102 **11111011**
103
104
105
106
107
108

- Memory: sequence of storage locations (cells)

- Each location has an address

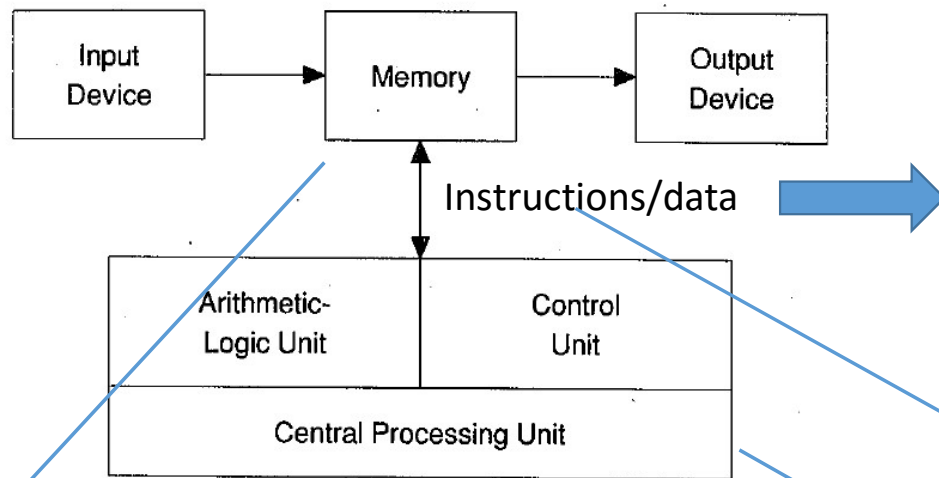- Each location is called word that is made up of bits

Instruction: operation and operand(s)

CPU works in **fetch-decode-execute** cycle

15

Principal Components of a Computer System

- Von Neuman architecture
  - Data and programs are **stored** in the same memory
  - CPU is separate and ins/data must be transmitted
  - Sequential execution of stored program

- Memory: sequence of storage locations (cells)
- Each location has an address
- Each location is called word that is made up of bits

Instruction: operation and operand(s)

CPU works in **fetch-decode-execute** cycle
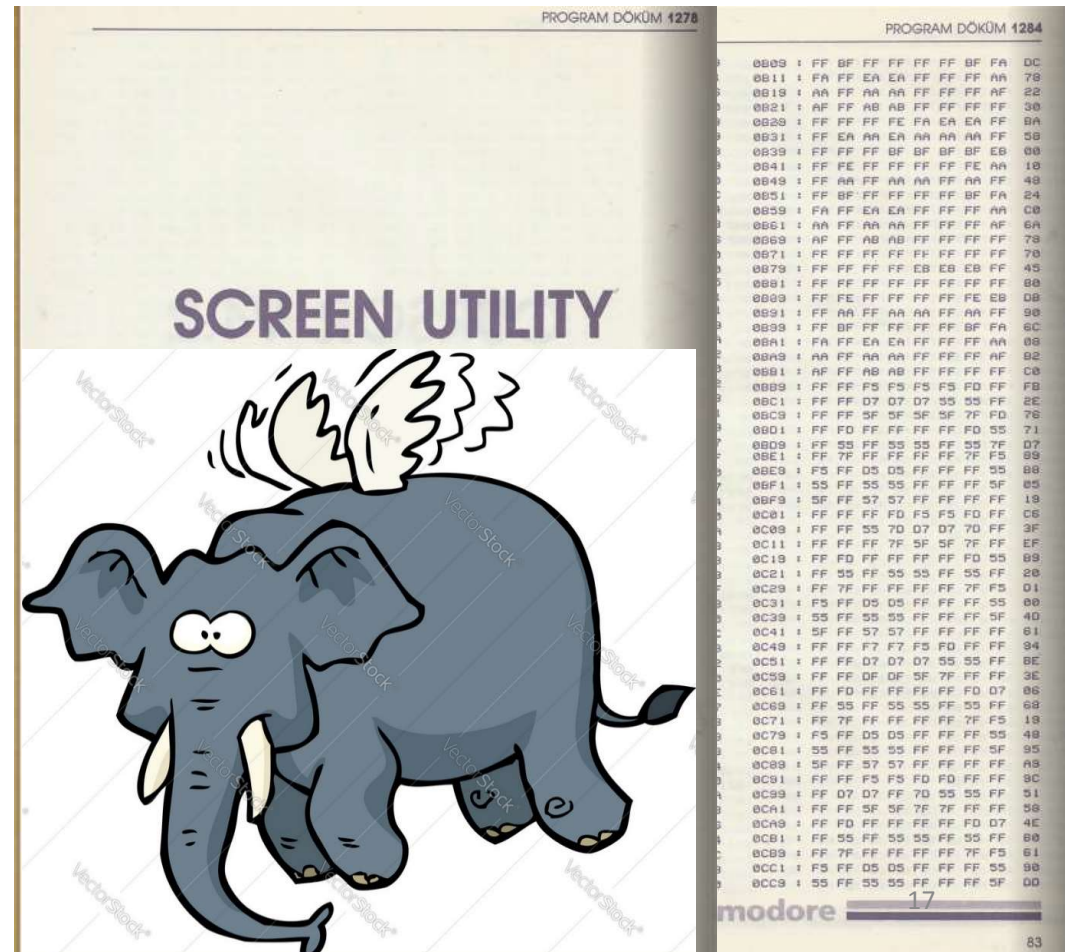
16

Programming Languages: Machine Language

- Every computer has its own language, **machine language**, that depends on the specific hardware of the program.

Programming Languages: Machine Language

```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
. . .
11001000 00000001 00000000 00000000   00000000 00000000
```

- Machine-dependent

- Programmer must know numeric codes of operations and
  - must keep track of the address of the data items in memory (modification is difficult!)

- Cumbersome and error-prone!

18

Programming Languages: Assembly Language

```
GO:  MOV       B  A
     ADD       C  A
     HALT
B    .WORD     100
C    .WORD     150
A    .WORD     0
     .END      GO
```

- Advantages of writing in assembly language rather than machine language
  - Use of symbolic operation codes rather than numeric (binary) ones
  - Use of symbolic memory addresses rather than numeric (binary) ones
- An assembler translates an assembly language program to machine code (or, object code).

Still hard to write and not portable!

since producing machine code is our objective/target
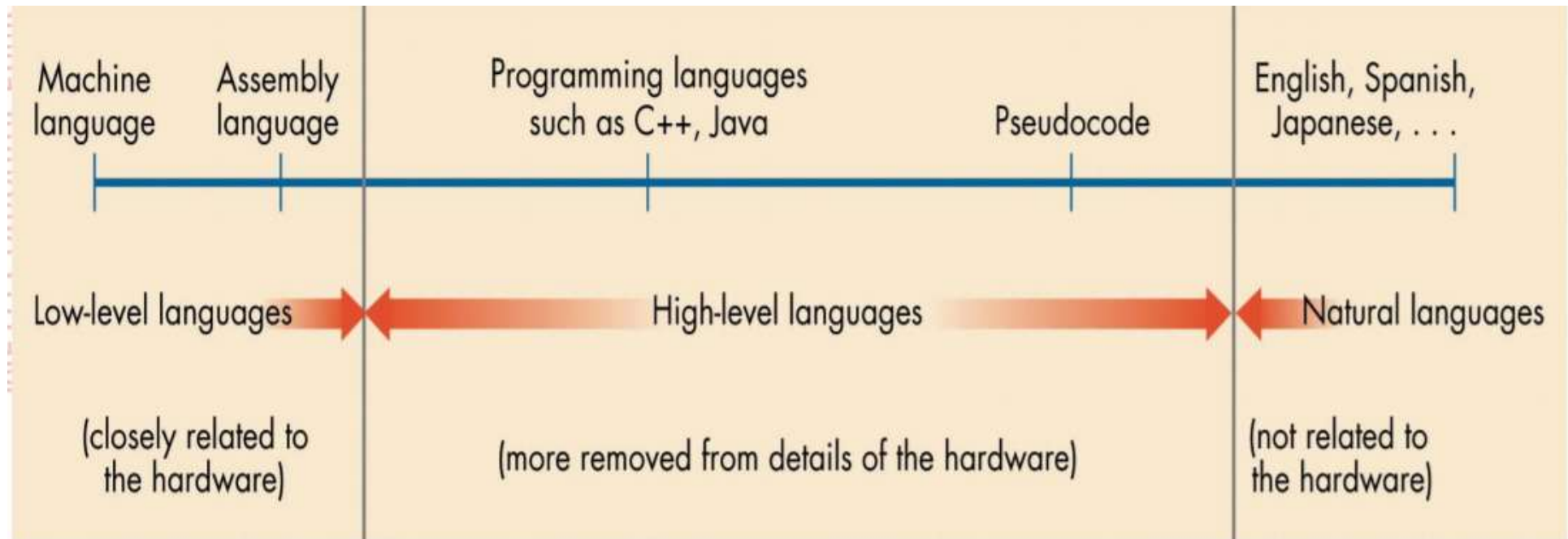
Programming Languages: High-level

- More similar to our spoken languages
- Relieve the programmers from the burden of low-level details and focus on the real problem!
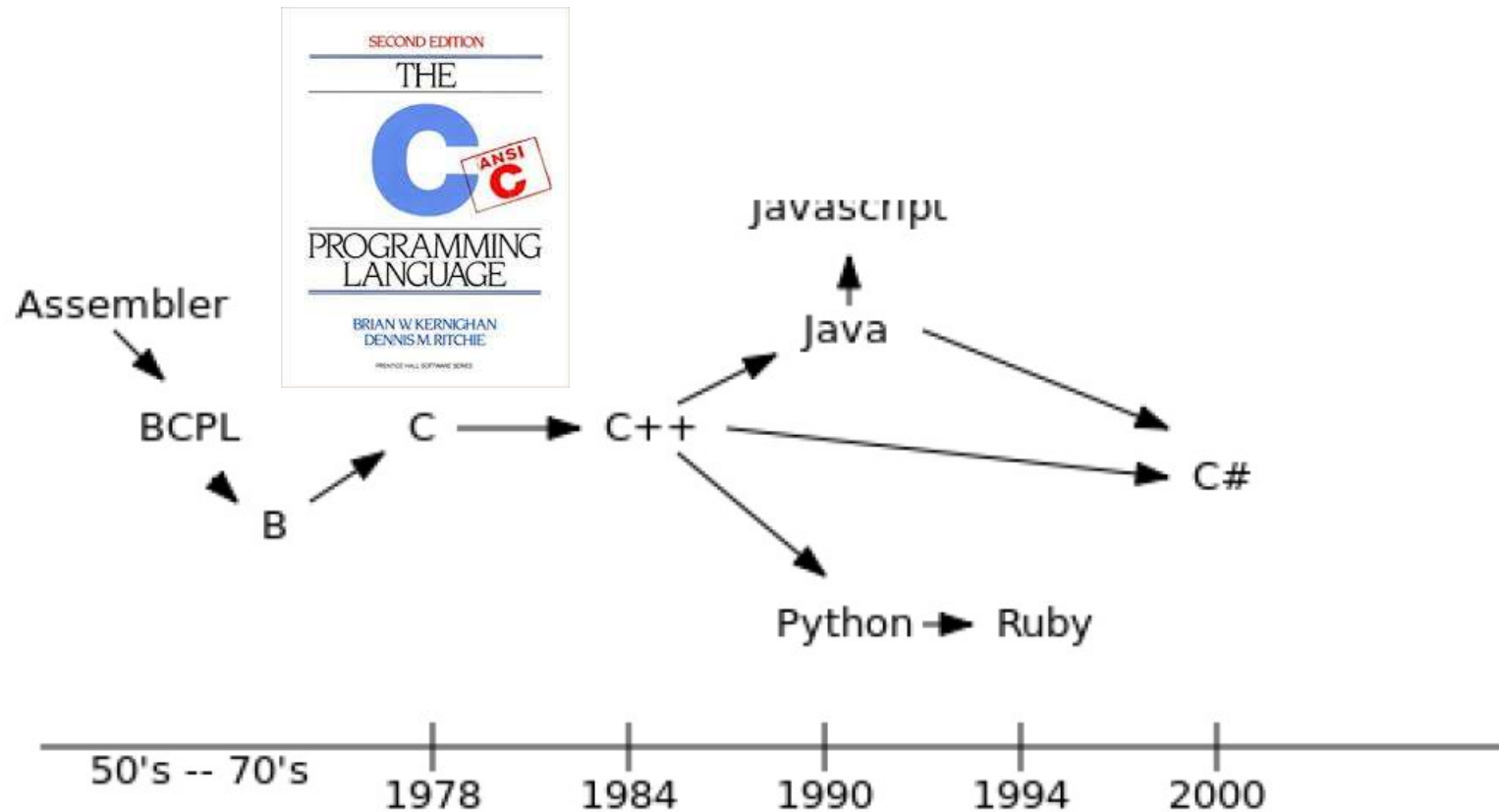- Compiled or interpreted.

Higher-Level
Language Version

$B = 100$
$C = 150$
$A = B + C$

Summary:

Machine language	Assembly language	Programming languages such as C++, Java	Pseudocode	English, Spanish, Japanese, . . .

Low-level languages ← High-level languages → Natural languages

(closely related to the hardware) — (more removed from details of the hardware) — (not related to the hardware)

Programming Languages Genealogy

- Ansi Standardization for C in 1988-89 → We will consider **ANSI-C!!!**

When to use C

- Working close to hardware
  - Operating System
  - Device Drivers

- Need to violate type-safety
  - Pack and unpack bytes

- Cannot tolerate overheads
  - No garbage collector
  - No array bounds check
  - No memory initialization
  - No exceptions

When not to use C

- Use JAVA or C# for . . .
  - Quick prototype
  - Compile-once Run-Everywhere

- Reliability is critical, and performance is secondary
  - C can be very reliable, but requires programmer discipline
  - For many programs, JAVA can match C performance, but not always…

How to be a good programmer

- Practice, practice, practice!

Programming Process

- The process of developing a computer program to solve a problem involves the following steps:
    - **Problem Definition**
    - **Program Design**
    - **Program Coding**
    - Testing & Debugging
    - Documentation

Programming Process: Program Design

- Once you defined the problem, devise an **ALGORITHM**, a finite sequence of steps, to solve the problem!
  - Let's start with «Everyday Algorithms»

  [**Special thanks to Dr. David Davenport**]

Everyday Algorithms: Changing a flat tire

- Explain how to change a flat tyre.
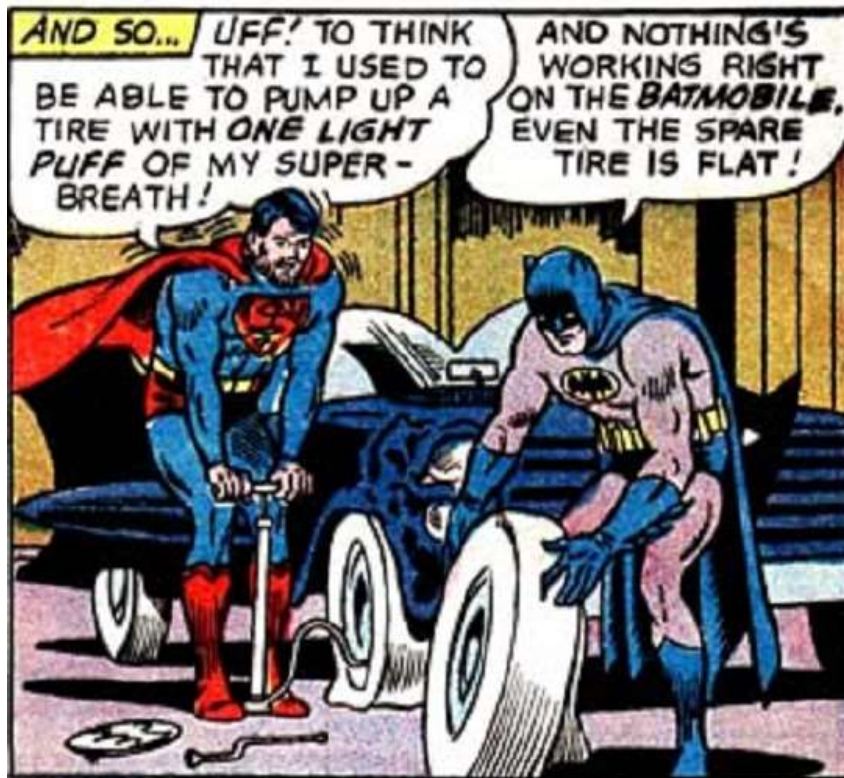
Provide an "**algorithm**" (set of instructions or a procedure) that solves the problem.

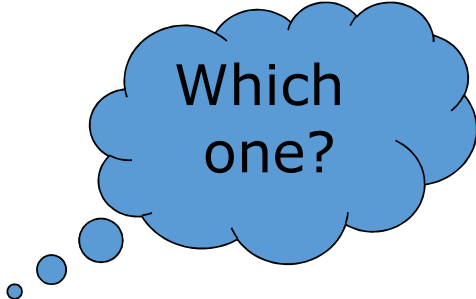# Everyday Algorithms: Changing a flat tire

- Write the solution out in natural language.
- Refine the wording, make it as <u>clear</u> and <u>concise</u> as possible.
  - grammar not so important; "**pseudocode**"
- <u>Number</u> the steps (1, 2, 3…) to indicate the <u>order</u> in which they must be done.
- Draw boxes around each of the <u>sub-parts</u>, especially those that are done repeatedly.
- Use <u>indentation</u> to clearly show sub-steps.

Changing a flat-tyre

Sample solution_1
1. Ensure the car is parked safely and cannot roll.
2. Remove wheel with flat tyre
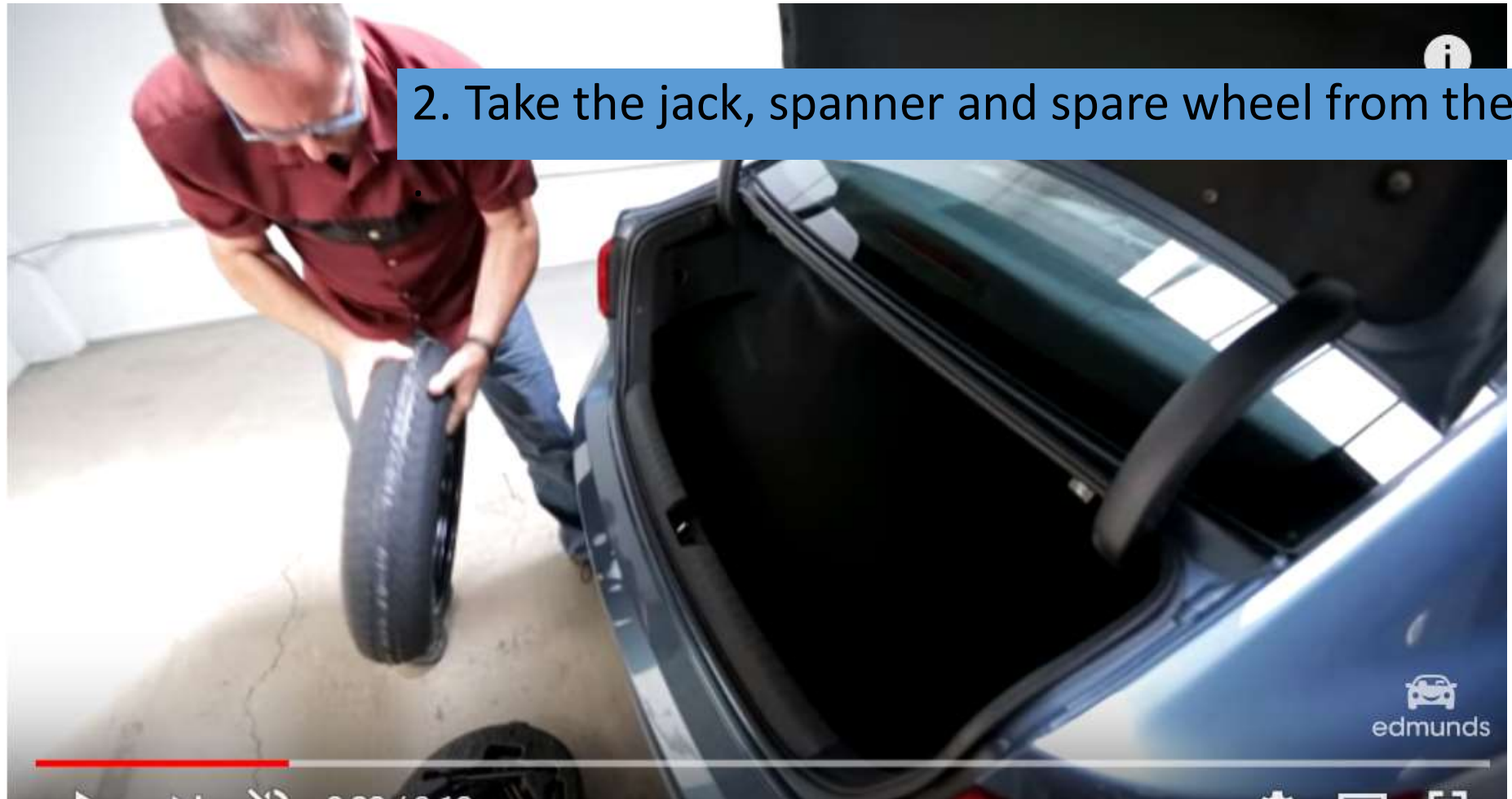3. Replace it with the spare wheel
4. Drive off

Which one?

Sample solution_2
1. Ensure the car is parked safely and cannot roll.
2. Take the jack, spanner and spare wheel from the trunk.
3. Slacken the wheel nuts on the wheel with the flat tyre
4. Jack the car up so the wheel with the flat tyre can be removed.
5. Remove all the nuts from wheel with the flat tyre
6. Take the wheel off and place it in the trunk
7. Put the spare wheel on in place of the one with the flat tyre
8. Put the nuts on and tighten as much as possible
9. Lower the car off the jack
10. Fully tighten the nuts on the new wheel
11. Put the jack and spanner back in the trunk
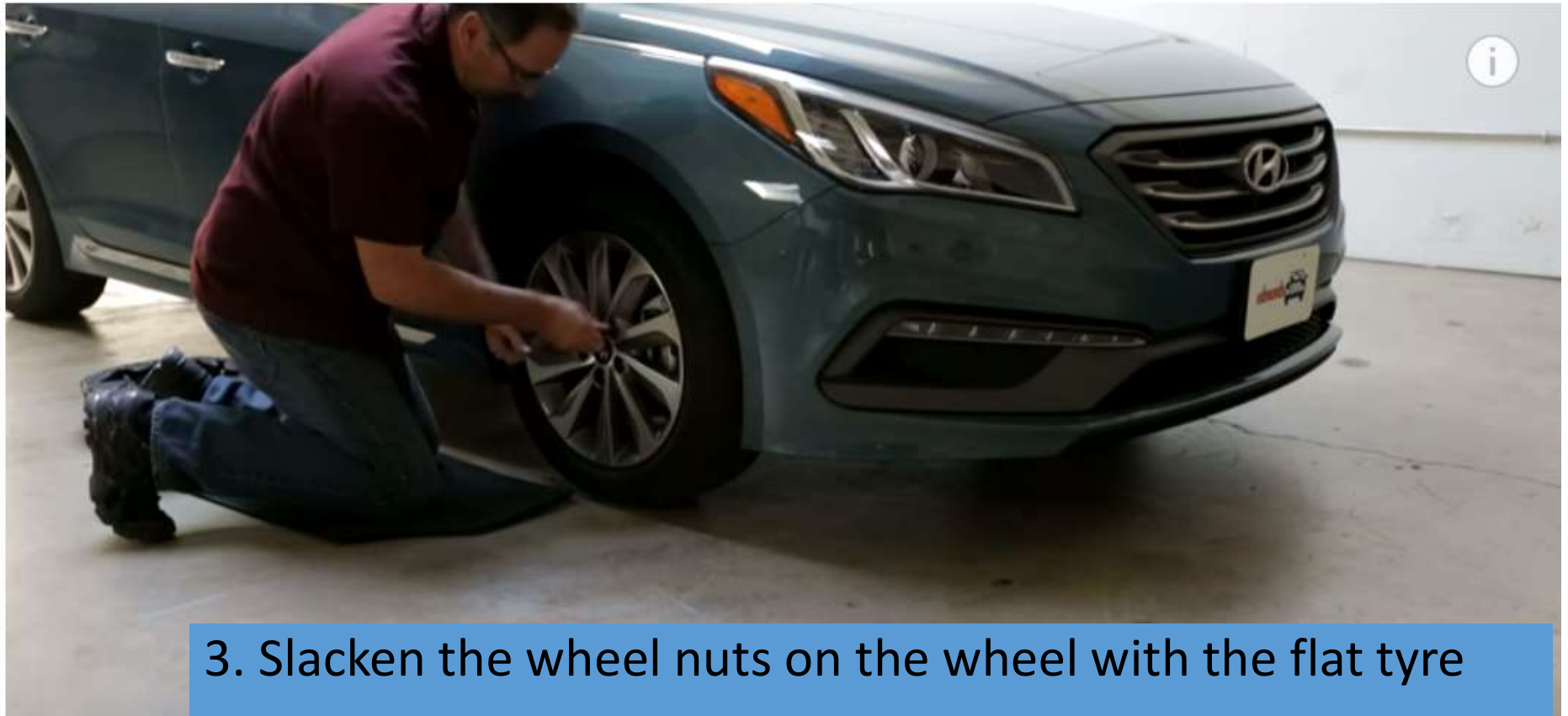12. Drive off

[Dr. David Davenport]

1. Ensure the car is parked safely and cannot roll.

2. Take the jack, spanner and spare wheel from the trunk

3. Slacken the wheel nuts on the wheel with the flat tyre
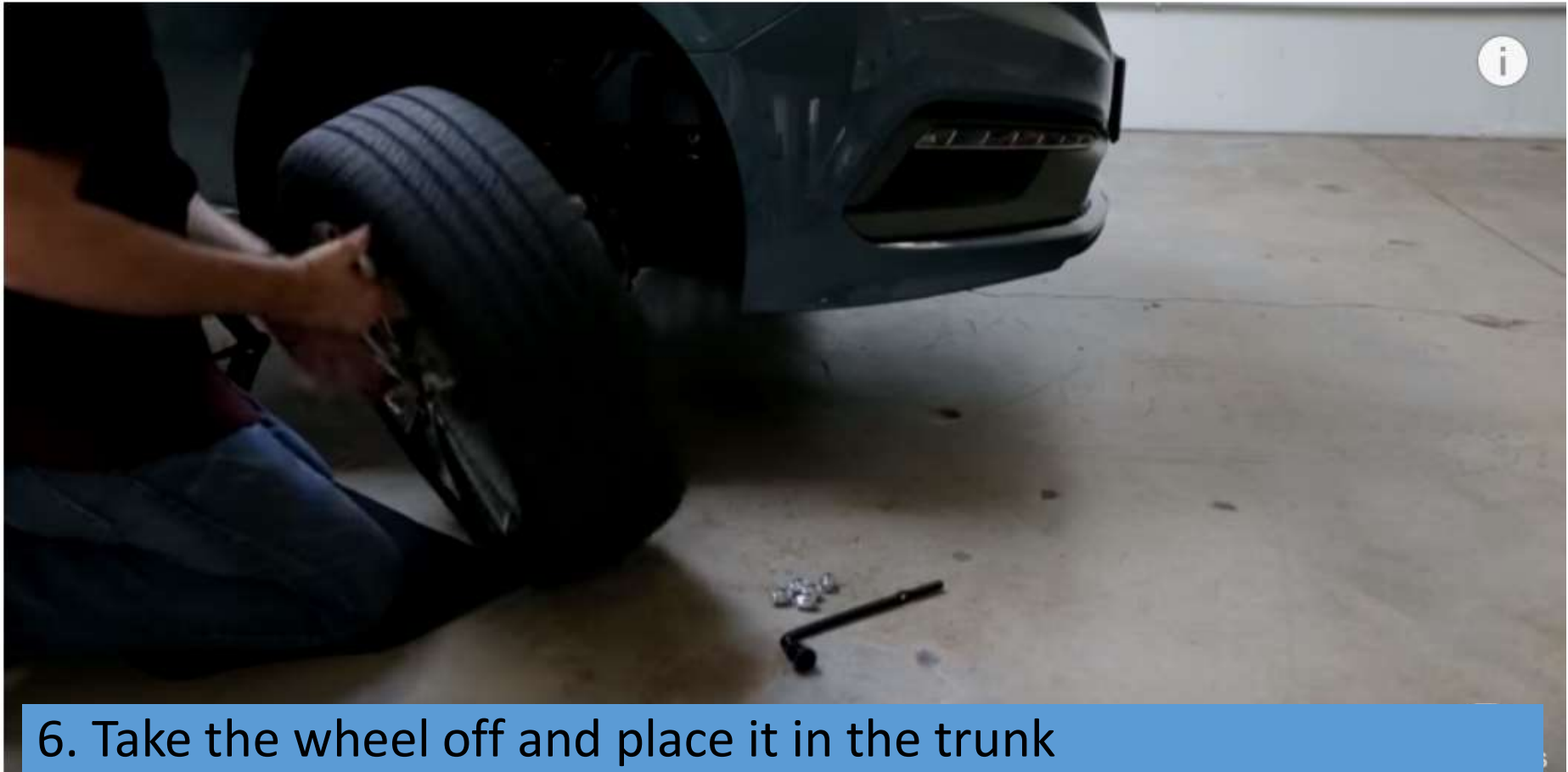
.

4. Jack the car up so the wheel with the flat tyre can be removed.

5. Remove all the nuts from wheel with the flat tyre

.

6. Take the wheel off and place it in the trunk

7. Put the spare wheel on in place of the one with the flat tyre

8. Put the nuts on and tighten as much as possible

9. Lower the car off the jack

10. Fully tighten the nuts on the new wheel

11. Put the jack and spanner back in the trunk

12 Drive off

Changing a flat-tyre

Sample solution_1
1. Ensure the car is parked safely and cannot roll.
2. Remove wheel with flat tyre
3. Replace it with the spare wheel
4. Drive off

Are both correct?

Which is better?

Sample solution_2
1. Ensure the car is parked safely and cannot roll.
2. Take the jack, spanner and spare wheel from the trunk.
3. Slacken the wheel nuts on the wheel with the flat tyre
4. Jack the car up so the wheel with the flat tyre can be removed.
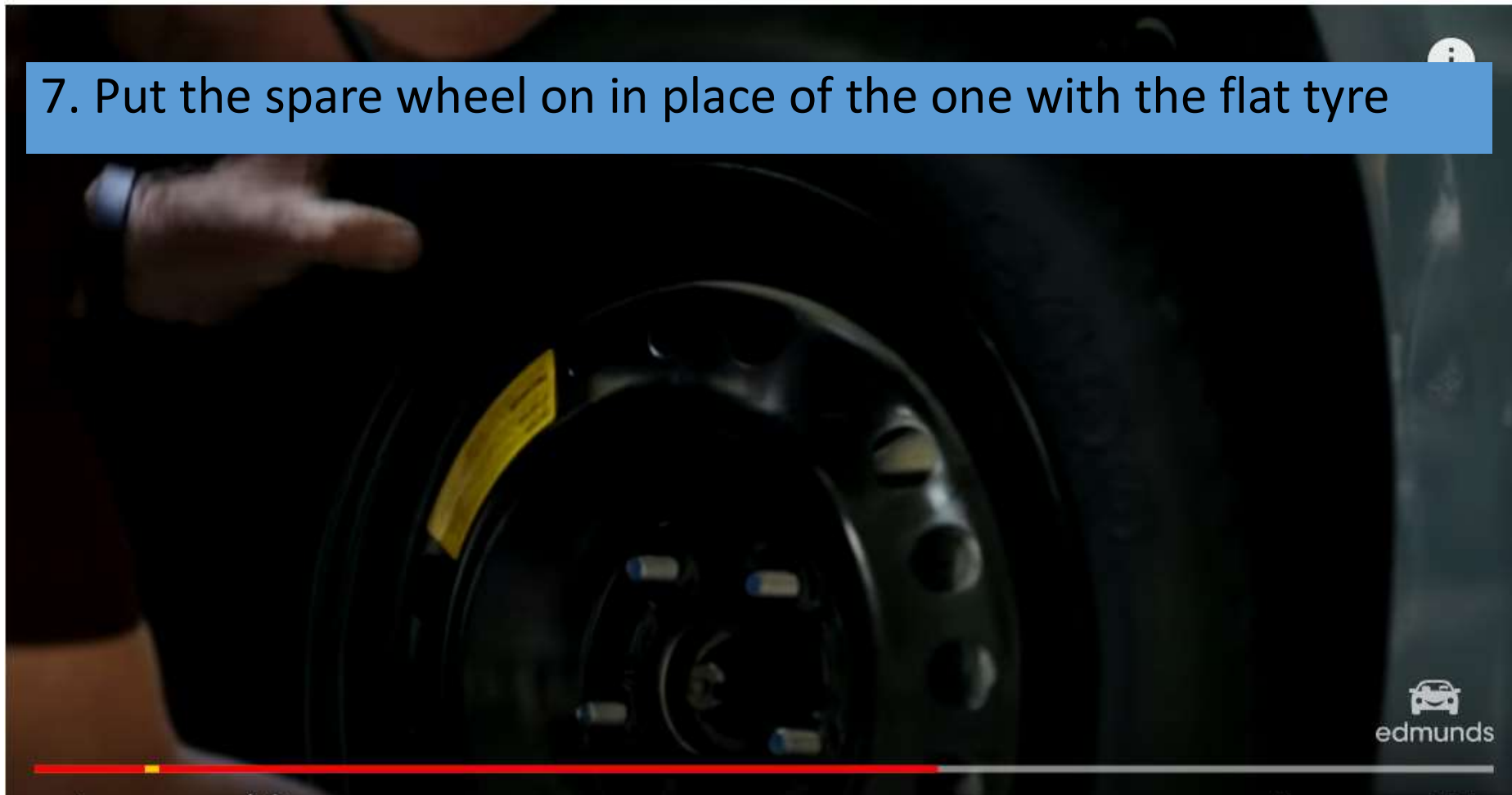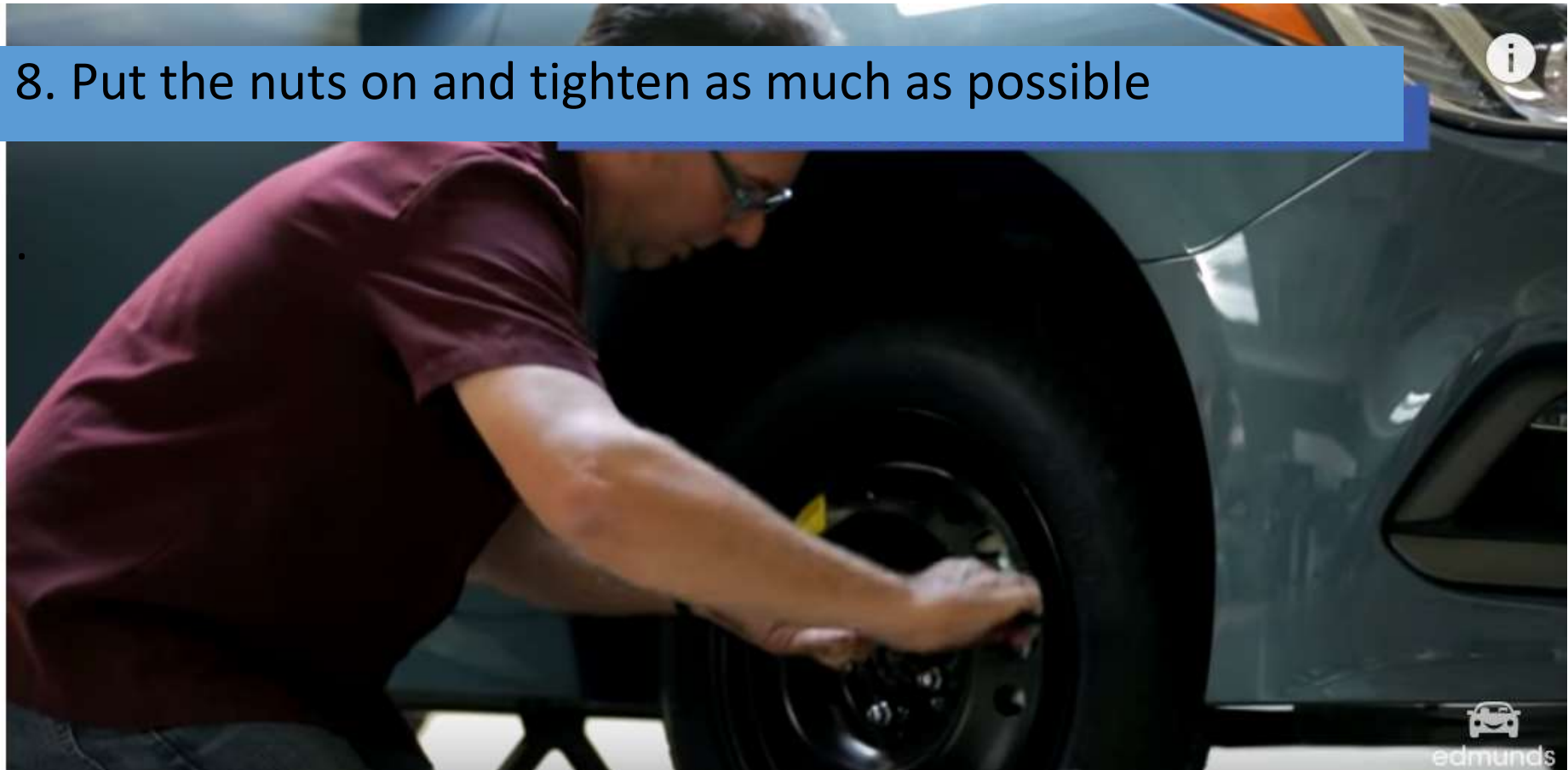5. Remove all the nuts from wheel with the flat tyre
6. Take the wheel off and place it in the trunk
7. Put the spare wheel on in place of the one with the flat tyre
8. Put the nuts on and tighten as much as possible
9. Lower the car off the jack
10. Fully tighten the nuts on the new wheel
11. Put the jack and spanner back in the trunk
12. Drive off

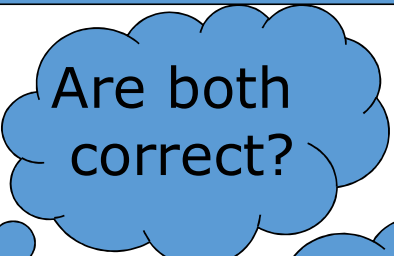[Dr. David Davenport]

Changing a flat-tyre

Sample solution_3

1. Ensure the car is parked safely and cannot roll.
2. Remove wheel with flat tyre
    1. Take the jack, spanner and spare wheel from the trunk.
    2. Slacken the wheel nuts on the wheel with the flat tyre
    3. Jack the car up so the wheel with the flat tyre can be removed.
    4. Remove all the nuts from wheel with the flat tyre
    5. Take the wheel off and place it in the trunk
3. Replace it with the spare wheel
    1. Put the spare wheel on in place of the one with the flat tyre
    2. Put the nuts on and tighten as much as possible
    3. Lower the car off the jack
    4. Fully tighten the nuts on the new wheel
    5. Put the jack and spanner back in the trunk
4. Drive off

The best of both solutions?

44

[Dr. David Davenport]

Changing a flat-tyre

- Do high-level minimal solution first
- Check & if ok
- then add detail to each piece as necessary
  - only need to think about piece, not whole

Ask whether each piece provides enough information to sub-contract it so that you could be sure that the result would be correct (without any additional information or clarification being needed).

Changing a flat-tyre

Sample solution_4
1. Ensure the car is parked safely and cannot roll.
2. Remove wheel with flat tyre
3. Replace it with the spare wheel
4. Drive off

The best?

3. Replace it with the spare wheel
   1. Put the spare wheel on in place of the one with the flat tyre
   2. Put the nuts on and tighten as much as possible
   3. Lower the car off the jack
   4. Fully tighten the nuts on the new wheel
   5. Put the jack and spanner back in the trunk

2. Remove wheel with flat tyre
   1. Take the jack, spanner and spare wheel from the trunk.
   2. Slacken the wheel nuts on the wheel with the flat tyre
   3. Jack the car up so the wheel with the flat tyre can be removed.
   4. Remove all the nuts from wheel with the flat tyre
   5. Take the wheel off and place it in the trunk

[Dr. David Davenport]

46

Changing a flat-tyre

PreCondition:
- Have car with flat tyre
- Have spare wheel with good tyre
- *Have a jack*
- *Have a spanner for wheel nuts*

Precondition: the state the world/system required for this solution to work correctly

What state must the world/system be in for this solution to work?

Sample solution_1
1. Ensure the car is parked safely and cannot roll.
2. Remove wheel with flat tyre
3. Replace it with the spare wheel
4. Drive off

What state is it in afterwards?

PostCondition:
- Have car with four good wheels
- *Have flat tyre in boot for repair!*

Assuming PreConditions held!

PostConditions: the state of the system/world after the algorithm is executed assuming its preconditions were met!

47
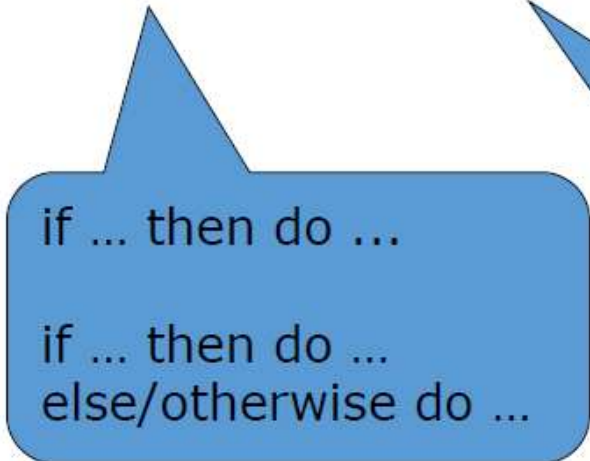
[Dr. David Davenport]

Program Design: Algorithms

- An **algorithm** is a procedure/method comprising the steps to solve a given problem.

- It must
  - be understandable, effective, correct, & **must stop**

- Algorithms comprise:
  - Sequence, decision, repetition

Do … and
next do …
next do …

if … then do …

if … then do …
else/otherwise do …
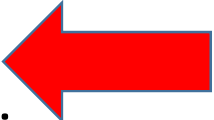
for all/each/every
do …

while … do …

repeat … until …

[Dr. David Davenport]

In memory of Dr. David Davenport

- 35+ years in teaching CS
- His programming motto that I never forget:

"Right at the first time"

- Requires an excellent understanding of the problem
- Requires an excellent desing
- Requires an excellent implementation

49

Programming Process

- The process of developing a computer program to solve a problem involves the following steps:
  - Problem Definition
  - Program Design
  - **Program Coding** ⬅
  - Testing & Debugging
  - Documentation

Today:

Our first C program saying "Hello World"...!

jenn schiffer
@jennschiffer

Follow

world never says "hello" back

Abstractions conf returns 2019 @abstractionscon
What five words best describe programming?
Show this thread

Program Coding in C

- Gross Layout of a C program
- C programs are made up of one or more **functions**
  - each performing a particular task (hence, top-down design nicely fits to C)
- We encourage **structured programming** to code C functions
  - i.e., algorithm is written using a combination of 3 control structures (sequential, selective, and repetitive)

Program

declarations (Global)

functions

Declarations (Local)

statements   expressions

keywords   names, literals   operators
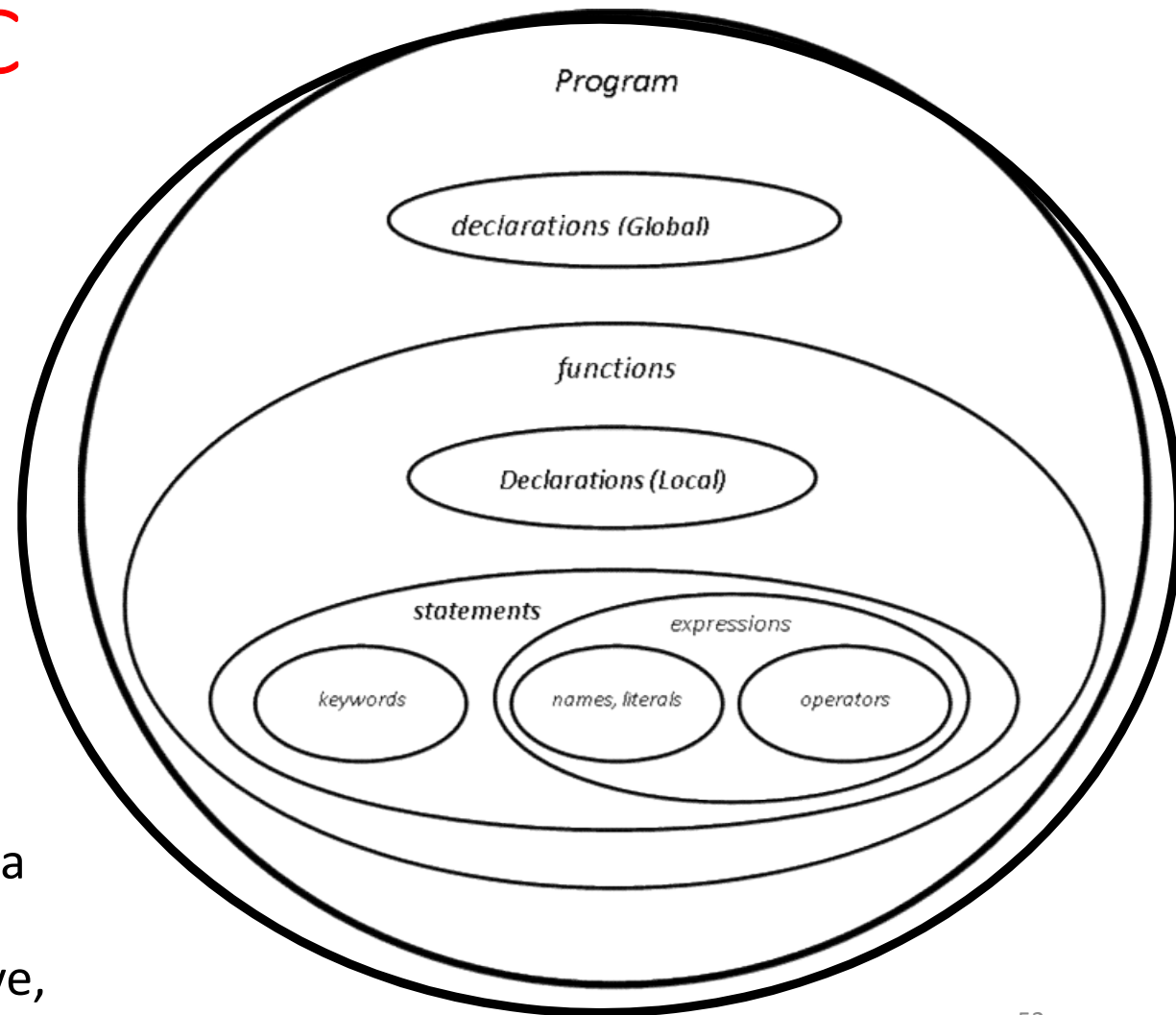
My first program: Hello world!

1) C programs are made up of one or more **functions** each performing a particular task

2) The execution of a program starts from a special function, called **main**
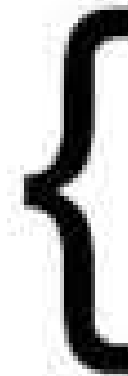
Beginning of function body

End of function body

My first program: Hello world!

#include <stdio.h>

void indicates main expects no arguments

int main(**void**)
{

   printf("Hello world\n");
   return 0;

}

return type of main

2) The execution of a program starts from a special function, called **main**

 • when the program is executed, **main** is called by the *environment*

3) When a function is **called**,

 • We usually provide a list of values, i.e.,**arguments**, as input to function

 • A C function may return a value (as output) to its caller, using a **return** statement

 • **main** returns its return status to environment; value 0 denotes normal termination
   [**main** function return value is not a big deal for us in Ceng140!]

f(2, 5)   7

f(x,y)

54

My first program: Hello world!

```c
#include <stdio.h>

int main(void)
{
    printf("Hello world\n") ;
    return 0 ;
}
```

4) **Statements** specify the computing operations to achieve the desired result

- **Semi-colon** denotes the termination of the statement

5) Case does matter! And C is free-format!

- **Return 0** not equivalent to **return 0**
- **inT mAin** not equivalent to **int main**

`int main(void) { printf("Hello world\n") ; return 0 ; }`

55

My first program: Hello world!

```
#include <stdio.h>

int main(void)
{
    printf("Hello world\n") ;
    return 0 ;
}
```

6) Program execution stops when
- End of main function is reached
- An exit call is reached
- Program is interrupted or it crushes

My first program: Hello world!

```c
#include <stdio.h>

int main(void)
{
    printf("Hello world\n") ;
    return 0 ;
}
```

For commonly used functions!

7) printf is a **call** to a *standard library* function

- printf has a **string argument** (a sequence of characters between quotation marks), it prints the characters in the string on display and \n is taking the cursor to the new line

57

My first program: Hello world!

```c
#include <stdio.h>

/* My first program */
int main(void)
{
    /* My code will print
       smt and then terminates */
    printf("Hello world\n") ;
    return 0 ;
}
```

8) Comments: for humans (us!)

- multi-line: /* Here I write a
                      comment */
- Till the end of current line: // Short!
  (**Not** ANSI style)

Let's define our first function

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

Parameter declarations:
The arguments expected by a function are specified in a parenthesized **parameter list** after func name

Return type

```c
int square(int a)
{
    return a*a;
}
```

Function <u>definition</u>

```c
int main(void)
{   square(5);
    printf("Hello world\n") ;
    return 0 ;  }
```

Function <u>call</u> with argument 5

Remark: We talk a lot about ... types!

- C is a statically typed language
  - Variables have **types**, so are the functions.
  - Type declarations give invariants that hold for all executions of a program! (e.g., a variable always holds an integer)
  - Types are checked by the compiler and may yield compile-time errors!

- Python is a dynamically typed language!

```
int square(int a)
{
  return a*a;
}
```

```python
def square(a):
    return a*a
```

More on types next week!

```c
#include <stdio.h>
#include <stdlib.h>
/* Func definition for square */
int square(int a)
{
    return a*a;
}
int main(void)
{   square(5);
    printf("Hello world\n") ;
    return 0 ;  }
```

In ANSI-C, if a function is used before it is **defined**, it has to be **declared** first:

```c
int square(int a); /* Func declaration */

int main(void)
{   square(5);
    printf("Hello world\n") ;
    return 0 ;  }
```

/* And then define square function below, or maybe in another source file */

Time to compile... but, wait!

- We just said:
  - In ANSI-C, if a function is used before it is **defined**, it has to be **declared** first
  - We are using printf in our main function, where is it defined or declared?

```
#include <stdio.h>

int main(void)
{

    printf("Hello world\n") ;
    return 0 ;

}
```

- A header file (*.h) contains declarations and macro definitions (no executable code)
  - For system header files: <AAA.h>
  - For user-defined header files: "AAA.h"

- Including a header file is same as **copying** the file content to the source file

What does stdio.h look like?

Including a header file is same as **copying** the file content to the source file

```
/*          Copyright (c) 1988 AT&T            */
/*            All Rights Reserved    */
/*          THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T        */
/*          The copyright notice above does not evidence any        */
/*          actual or intended publication of such source code.      */
/*
 * User-visible pieces of the ANSI C standard I/O package.
 */


#ifndef _STDIO_H
#define          _STDIO_H
.....
.. int printf(...);
```

#include <stdio.h>

int main(void)

{

    printf("Hello world\n") ;

    return 0 ;

}

Compiling our first C code

```
#include <stdio.h>

/* Func definition for square */
int square(int a)
{
    return a*a;
}
int main(void)
{   square(5);
    printf("Hello world\n") ;
    return 0 ;  }
```
test.c

Ready to compile!

linux$ **gcc –o test test.c** ➡ Takes source file **test.c** &
generates **test** as the executable output file

stdio.h

libc (includes .o files for functions in stdio.h)

System include files

System libraries

Source File	→	Preprocessor	→	Compiler	→	Assembler	→	Object File	→	Linker	→	Executable File	→	Loader

test.c

User include files

test.o

User libraries

test

Memory

Remove comments,
interpret directives (lines with #)
[mostly textual operations, nothing really smart!]

Mostly machine code, but not executable yet
Why?

65

System include files

libc (includes .o files for functions in stdio.h)

System libraries

Source File

Preprocessor → Compiler → Assembler → Object File → Linker → Executable File

test.c

test

User include files

User libraries

0101010101 **?**	printf.o
1101010101	
0101000110	square
0010101010	
1101010110	
0101000110	main
0010101010	
1101010110	→ Call square
?	→ Call printf

CODE – compiled and linked

66

So…

- An h file is simply contains the declarations of some functions (and other things).
  - Functions are usually implemented in a separate c file and compiled to the object file (.o)
  - We include h files to tell the compiler that such functions exist (and their input/output)
  - During linking their object files (.o files) are linked to our own program's object file and we get the final executable file

System and user libraries

- In test.c printf is advertised in stdio.h so we include this file, and since it is a system function, compiler/linker knows where to find its o file
  - Its o file is in system library!
- Why is this great?
  - No need to create your printf function → **code sharing**
  - Everybody can be responsible for coding certain functions → **modularity**
  - Details of a function can be hidden → **information hiding**
  - Can we also create our own h files and their object files?

TRY THIS AT HOME

Creating and compiling our library

```
#include <stdio.h>

#include <stdlib.h>

/* Func definition for square */

int square(int a)
{

    return a*a;

}

int main(void)

{   square(5);

    printf("Hello world\n") ;

    return 0 ;  }                    test.c
```

- Suppose I want to create a simple library for math functions
  - So I will move square function to there
  - But I still want others (like test.c) to be able to use this square function

Create myMath.h file

/* This is my super naive math library */

/* 2019 Ceng 140 */

/* First function I provide is square (yes, really!) */

/* So here goes the **declaration** of square function */

int square (int a);

myMath.h

Create myMath.c file

```
/* This is my super naive math library */
/* 2019 Ceng 140 */
/* First function I provide is square (yes, really!) */
/* So here goes the definition of square function */
int square(int a)
{
    return a*a;
}
```

myMath.c

My new test.c ➡

```c
#include <stdio.h>
#include <stdlib.h>
#include "myMath.h"
 /* Func definition for square */
int square(int a)
{
    return a*a;
}
int main(void)
{   square(5);
    printf("Hello world\n") ;
    return 0 ;  }
```

```c
/* So here goes the declaration of square */
int square (int a);
```
myMath.h

```c
/* So here goes the definition of square */
int square(int a)
{
    return a*a;
}
```
myMath.c

72
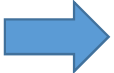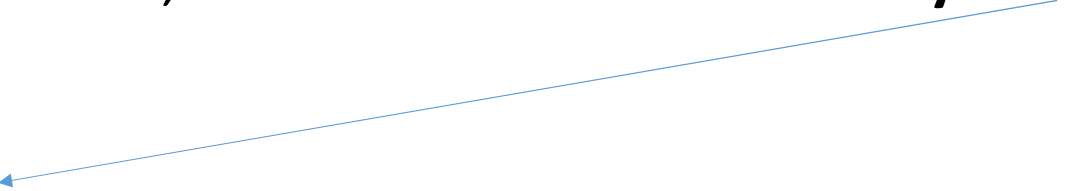
How to compile?

- **gcc –c myMath.c** ➡ -c flag only compiles until linking stage thus, this command will create **myMath.o**

- **gcc –o test test.c myMath.o** ➡ This command will:
  1) Compile uncompiled sources
  2) Link them (test.o and myMath.o) to create the executable called **test**

Summary

System include files → Preprocessor
User include files → Preprocessor

Source File → Preprocessor → Compiler → Assembler → Object File → Linker → Executable File → Loader → Memory

System libraries → Linker
User libraries → Linker

---

gcc –c myMath.c

Source File
mymath.c

1) Compile to create object file

Object File
mymath.o

---

gcc –o test test.c myMath.o

Source File
test.c

Object File
test.o

Linker → Executable File
test

1) Compile uncompiled sources
2) Link them (test.o and myMath.o) to create the executable called **test**

74

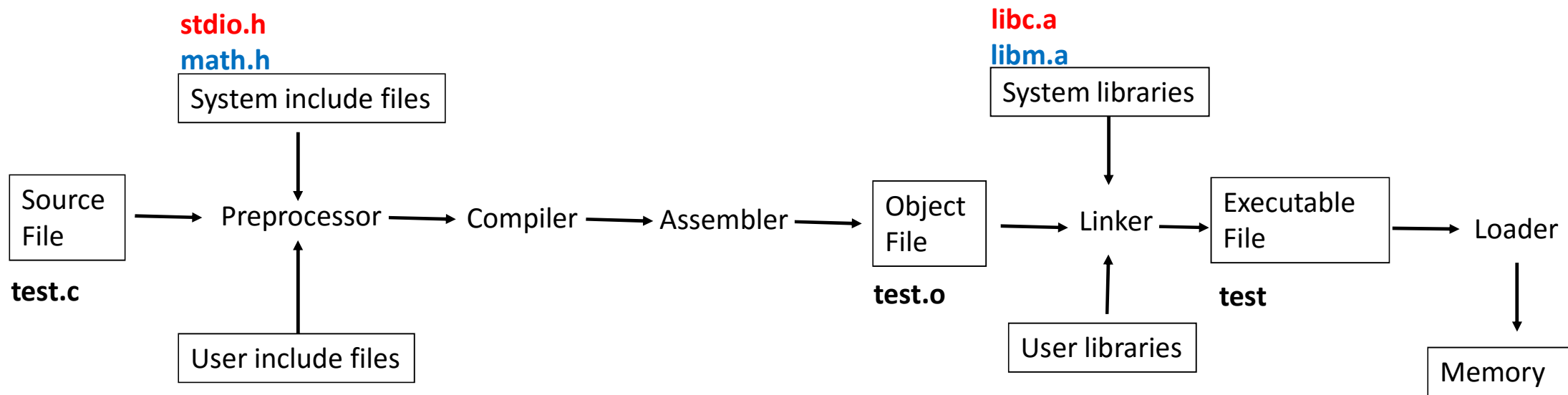# Using C's own math library

- Library: A package of object (.o files)
- C math libray
  - Header file (**math.h**),
  - o.files are in library **libm.a (**401 object files for math (sin, cos, log, exp, sqrt, …)

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Using C's own math library

stdio.h
math.h

System include files

libc.a
libm.a

System libraries

Source File → Preprocessor → Compiler → Assembler → Object File → Linker → Executable File → Loader

test.c

User include files

test.o

User libraries

test

Memory

libc is standard and linked automatically. How about **libm**?

the compiler is told to **link against a library** with the command line option **-l***name* where the library is called **lib***name* **.so**

gcc –o test test.c **-lm**

Take-away messages

- C programs are made up of one or more **functions**

- The execution of a program starts from a special function, called **main**

- A C program can **define** several functions for various tasks
  - Furthermore, can use functions defined in system or user libraries
  - To do so, the program should include the header files, which consist of function **declarations** (because a function must be either defined or declared before it is used)

- gcc with –c flag creates .o files for our source codes w/o main
  - Include headers also in corresponding C file for type checking

- gcc creates executable files by compiling and linking necessary object files

Now, you know more than this guy! ;)

Beyond

- Static vs dynamic libraries/linking:

- Static library → lib*something*.**a** Dynamic library → lib*something*.**so**

- If a copy of the libraries is **physically part of the executable**, then we say the executable has been *statically linked*

- If the executable merely contains filenames that enable the loader to find the program's library references at **runtime**, then we say it has been *dynamically linked*. Pros:
  - Smaller executable size (on disk and in memory)
  - Several executables share a single copy of library at runtime
  - Easy versioning/maintaining of libraries

Beyond

- The command line argument to the C compiler doesn't mention the entire pathname to the library file. It doesn't even mention the full name of the file in the library directory! Instead, the compiler is told to **link against a library** with the command line option **-l*name*** where the library is called **lib*name* .**so
  - Important: C_INCLUDE_PATH, LIBRARY_PATH env variables
  - Or, use gcc options –I and –L to specify include and library paths
  - It is also possible to link directly with individual static or dynamic library files by specifying the <u>full path </u>to the library on the command line.

Beyond

Table 5-1. Library Conventions Under Solaris 2.x

`#include` Filename	Library Pathname	Compiler Option to Use
<math.h>	/usr/lib/libm.so	`-lm`
<math.h>	/usr/lib/libm.a	`-dn -lm`
<stdio.h>	/usr/lib/libc.so	linked in automatically
"/usr/openwin/include/X11.h"	/usr/openwin/lib/libX11.so	`-L/usr/openwin/lib -lX11`
<thread.h>	/usr/lib/libthread.so	`-lthread`
<curses.h>	/usr/ccs/lib/libcurses.a	`-lcurses`
<sys/socket.h>	/usr/lib/libsocket.so	`-lsocket`

Beyond

- Dynamic linking is default for most gcc versions
  - Whenever a static library 'libNAME.a' would be used for linking with the option -lNAME the compiler first checks for an alternative shared library with the same name and a '.so' extension.
  - However, when the executable file is started its loader function must find the shared library in order to load it into memory. (i.e., so file must be in the /usr/lib/ or its path must be added to load path LD_LIBRARY_PATH environment variable)
  - Static linking can be forced via –static option of gcc (It is still useful for certain applications, for efficiency or security purposes).

- https://www.linuxtopia.org/online_books/an_introduction_to_gcc/gccintro_23.html [also try _24, 25.html and _17 to _20]

Beyond

- With dynamic libraries, *all* the library symbols go into the virtual address space of the output file, and *all* the symbols are available to all the other files in the link. In contrast, static linking only looks through the archive for the *undefined* symbols presently known to the loader at the time the archive is processed, so linking order is IMPORTANT!
    - So if you mention/link static library before your own code, will not compile
- Interpositioning
    - Interpositioning (some people call it "interposing") is the practice of supplanting a library function by a user-written function of the same name
    - May be very dangerous!
    - Beware of C reserved keywords and AVOID using them as identifier names in your code (like: abort, abs, sin, acos, CHAR_MAX, INT_MAX…)

Beyond

- Computer Systems: A Programmer's Perspective, Bryant & O'hallaron
  - See Chapter 7: Linking