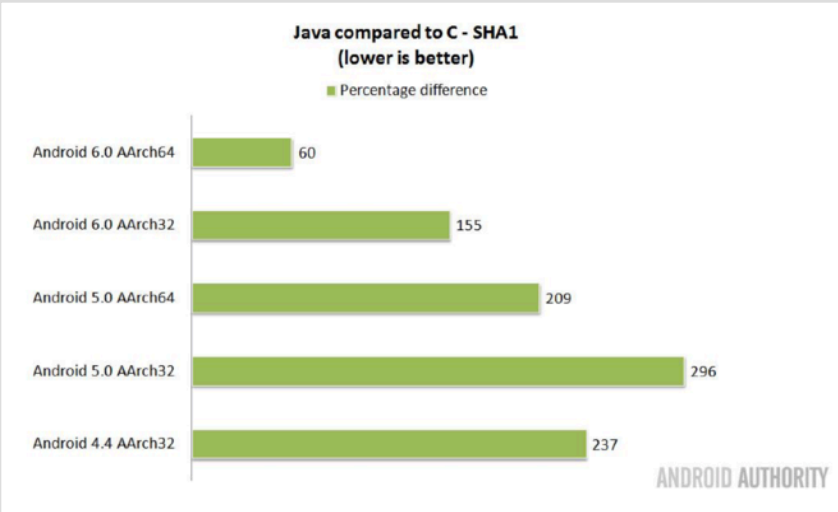
# Design and development of embedded systems for the Internet of Things (IoT)

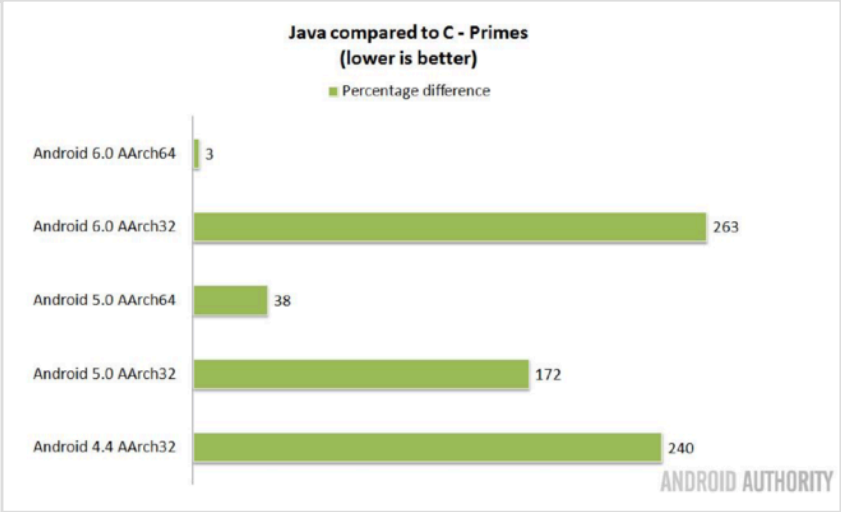
Fabio Angeletti
Fabrizio Gattuso


SAPIENZA
UNIVERSITÀ DI ROMA

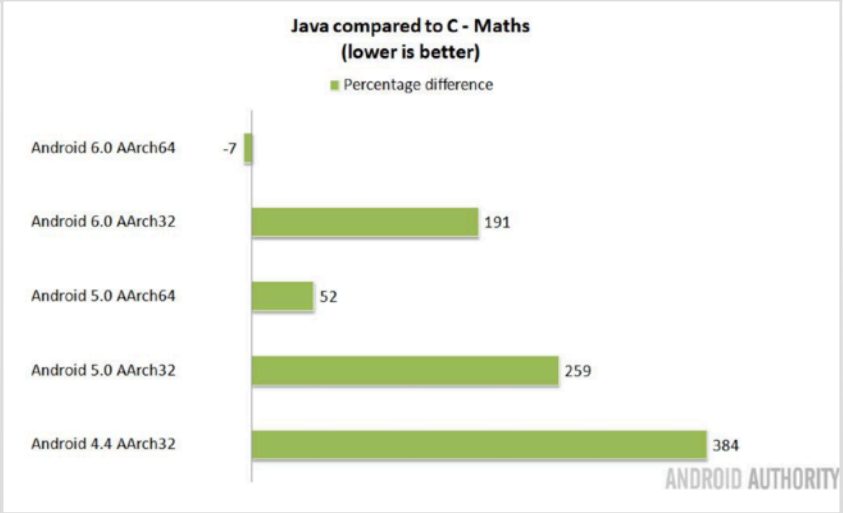
W • SENSE
INTEGRATED CABLELESS SOLUTIONS

# Why C?




Test on 21 Android Devices with 32-bits and 64-bits processors and different versions of Android.



# C in a nutshell

C is not Python or Java.

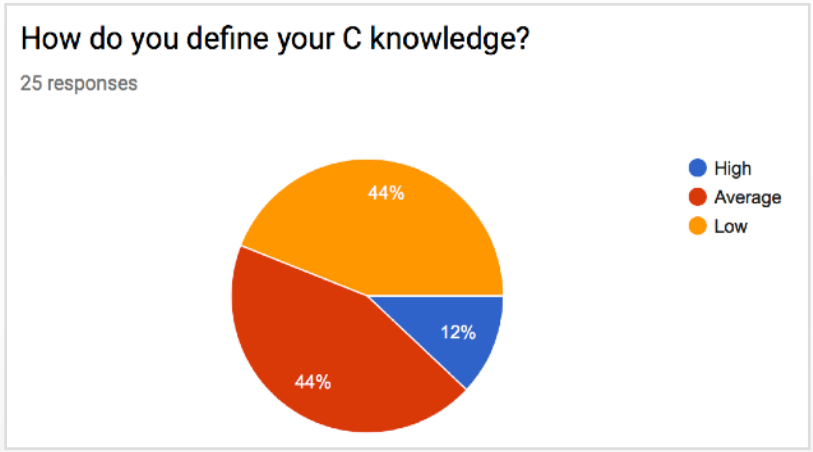
C is hard to learn and sometimes can be hard to handle.

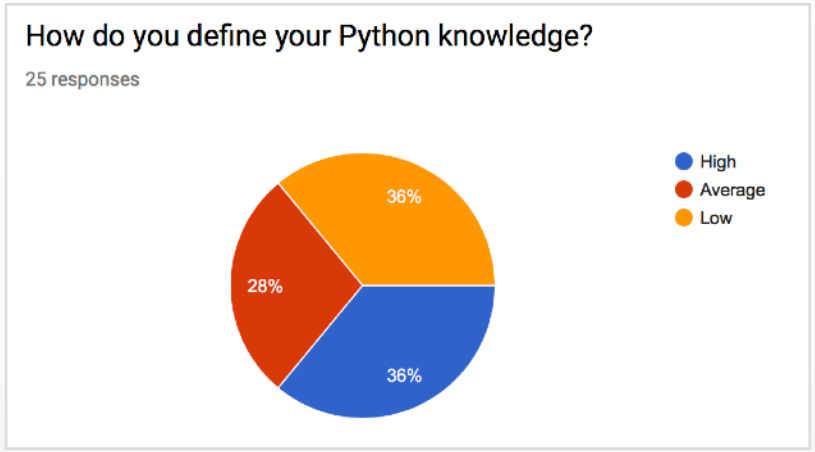
# C is everywhere

C is fast.
C is a small language.
C is designed to be compiled by simple compilers.
C map efficiently machine instructions.
C is portable and works on every platforms.

C is used by the 99% of the Operating Systems.
Every developer must know at least the basic notions.

# The reality




Something is wrong!

# C is powerful

C offers some special features ideally for system programming:

- Explicit memory management
- Explicit error detection
- Low-level features like bit operations
- No complex pre-built data-structures

This means more control but more possible mistakes.

# The first example

```
/* Hello World program */

#include <stdio.h>

int main() {
  printf("Hello World.\n”);
  return 0;
}
```

**Compile phase:** gcc -o hello hello.c

**Execute phase:** ./hello

Anyone doesn’t know this stuff?

# Primitive types

The basic and the most powerful one: **void**

The none type: **NULL**

The integers: **char, short, int, long, long long**

The floating points: **float, double, long double**

**Only by the standard C99 is define the boolean type but usually it is not used to back compatibility. Use instead 1 or 0.**

**Every type can be signed or not signed.**

**The size of every type depends on the platforms.**

**chart ->** 1 bytes **short ->** 2 bytes **int ->** 4 bytes **long ->** 8 bytes

**float ->** 4 bytes **double ->** 8 bytes **long double ->** 16 bytes

# Integer with different bases

Usually we represent integer with base 10 but sometimes is not the best choice for the embedded systems.

C allows to handle some special representation adding the following prefixes:

**Hexadecimal** (base 16):

**0x** *(%x to use with printf)*

**Octal** (base 8):

**0** *(%o to use with printf)*

There isn't a standard support for the binary representation but you can write your own functions to translate a base 10 int to a base 2 int.

**I hope you will do for the next lesson!**

# Bitwise operations in C

| Symbol | Operator | Example |
|---|---|---|
| & | Bitwise AND | **101 & 001 = 001** |
| \| | Bitwise Inclusive OR | **110 \| 101 = 111** |
| ^ | XOR | **110 ^ 100 = 011** |
| << | Left shift | **00110 << 2 = 1100** |
| >> | Right shift | **00110 >> 2 = 0011** |
| ~ | Bitwise Not (1 complement) | **~110 = 001** |

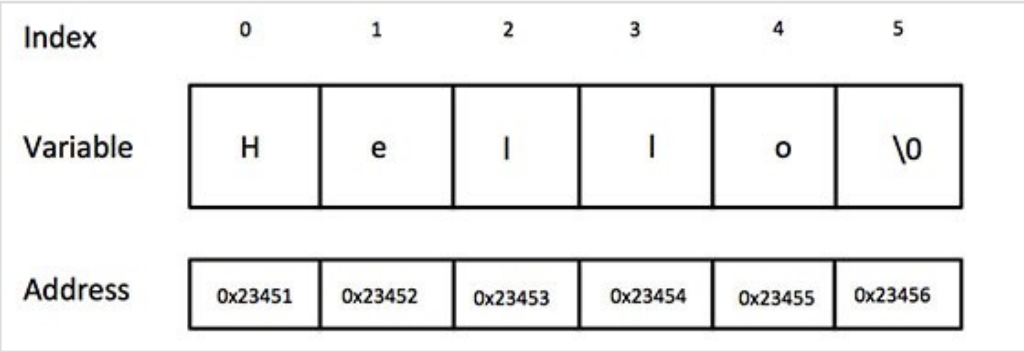
# Not so complex types

There isn't a predefined type for a string or for other structures like map/ dictionary without using external libraries (not always suggested).

**What is a string in C?**

**A sequence of chars terminated by a \0.**

```
char[] course = "IoT17\0";
```




**There are some utilities function to manipulate strings (string.h):**

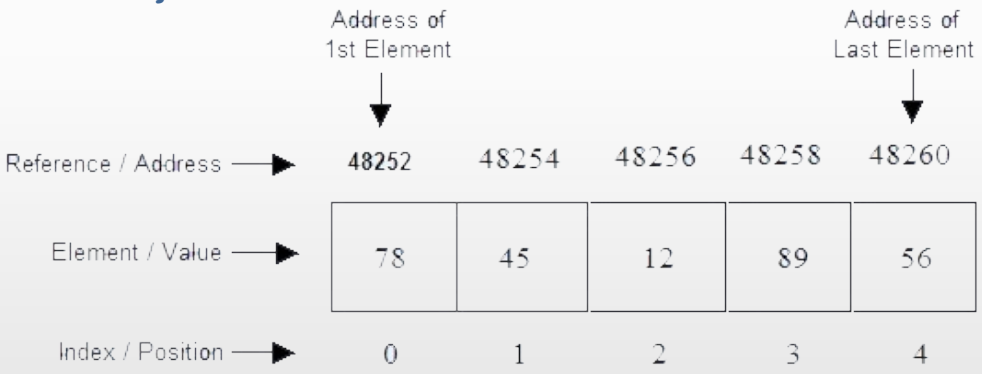
`strncpy, strncmp, strncat`

# Not so complex types (2)

**What is an array in C?**

**Same story. A sequence of a predefined type but without \0 char.**

```
int a[10];
for (int i = 0; i < 10; i++) {
  a[i] = 0;
}
```




**There are some utilities function to manipulate data:**

`memcpy, memcmp`

# Enumerations

**An enum is a user-defined data type that consists of constants.**

```
enum months {
  JANUARY,
  FEBRUARY,
  MARCH
};

enum months {
  JANUARY = 1,
  FEBRUARY = 3,
  MARCH
};
```

```
enum week {sunday, monday, tuesday,
wednesday, thursday, friday, saturday};

int main() {
  enum week today;
  today = wednesday;
  printf("Day %d", today + 1);
  return 0;
}
```

**Each element of enum gets an integer value and used like an integer.**

# Structures

**A struct is used to aggregate different data with the same meaning.**

```
struct birthday {
  char* name;
  enum months month;
  int day;
  int year;
};

struct my_birthday;
my_birthday.name = “Fabrizio\0”
my_birthday.month = JUNE;
```

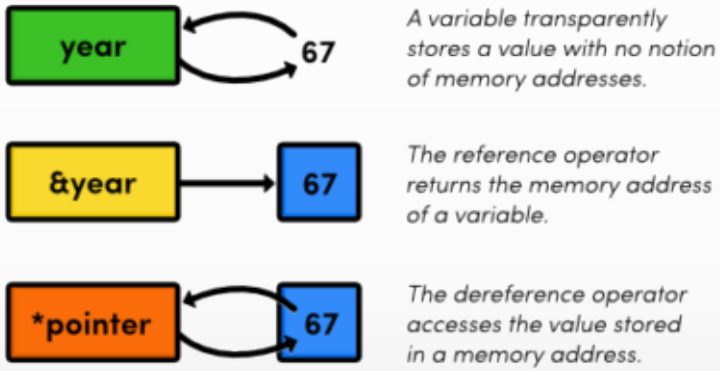
**Struct can refer also to other structures.**

# SEEMS EASY, DOESN'T IT?

# Pointers

**A pointer is a special variable that contains variable address.**

```
int* pointer = &var;
```




&var is the address of the variable

*var is the pointer to the variable

**If you assign a value to the pointer you modify also the original variable:**

```
*pointer = 2;
```

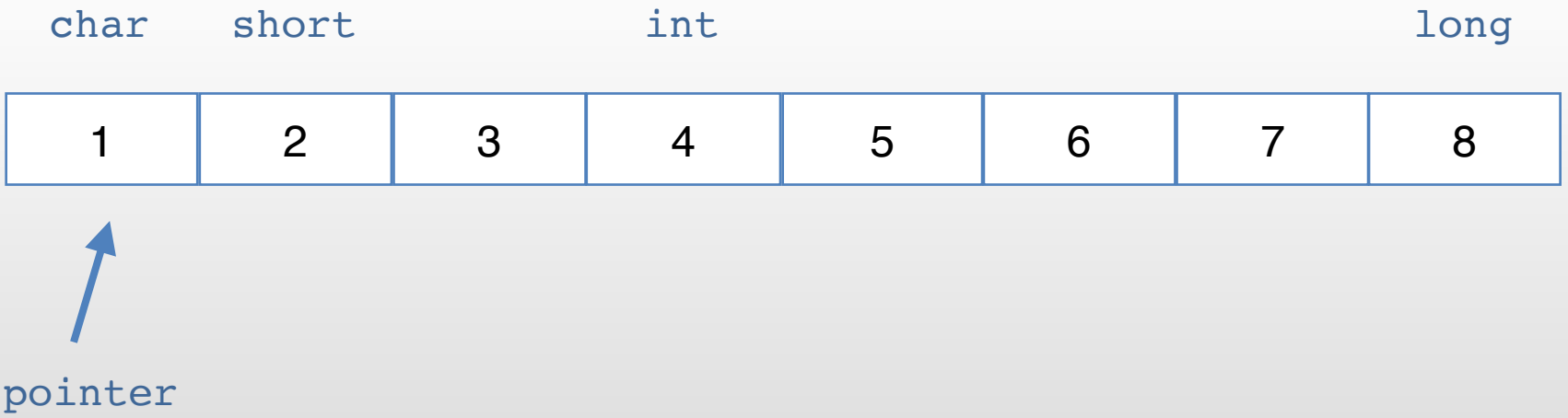
# Pointers (2)

**A pointer variable must be always initialized**

**with a valid address**

**or with the NULL value.**

**A pointer point to a address but you have to specify a type to understand what is written inside the address.**



# Enter the `void`

Void means nothing, emptiness. A function *"returning"* void actually does not return anything.

The size of void is undefined for this reason.

```
void* pointer;
is used to store addresses of unspecified types
```

You can *cast* to a specific type to do specific tasks.

```
void* pointer;
(int) pointer;
```

**NEVER READ OR WRITE A NOT ALLOWED MEMORY AREA**

# Pointer arithmetic

```
char* pointer;
```



```
char my_array[5];
my_array[3] = 0;
```

```
&my_array += 3 or
&my_array += 3 * sizeof(type)
```

# Memory management

## Global Variables

- Declared outside the body function
- Space allocated statically before program execution
- Cannot deallocate space until program finishes
- Name has to be unique for the whole program

## Local Variables

- Declared in the body of the function
- Space allocated when entering the function
- Space automatically deallocated when functions returns

**NEVER REFER TO A LOCAL VARIABLE AFTER THE FUNCTION RETURN**

# Memory management (2)

**Dynamic Variables**

- Memory has to be explicitly allocated
- Memory has to be explicitly deallocated **(only one time)**
- Dynamic variables are allocated in the the heap
  (a special area of memory)

**Allocate memory:**

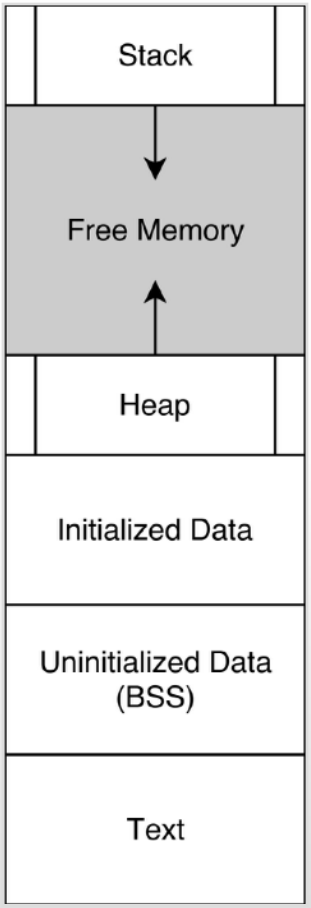
```
void* malloc(int)
```

**Deallocate memory:**

```
void* free(void*)
```

**Reallocate memory:**

```
void* realloc(void *ptr, size_t size)
```



# Functions

In C you can pass the values to the functions by:

- Value
- Reference

```
/* arguments passed by value */
int sum(int a, int b) {
  return (a + b); /* return by value */
}

/* arguments passed by reference */
int psum(int* pa, int* pb) {
  return ((*pa) + (*pb));
}

int psum(int *p) {
  *p = a + b; /* return by reference */
}
```

# Functions (2)

In C is also defined a pointer to a function. Usually used to force another programmer to implement their own version of common functionalities.

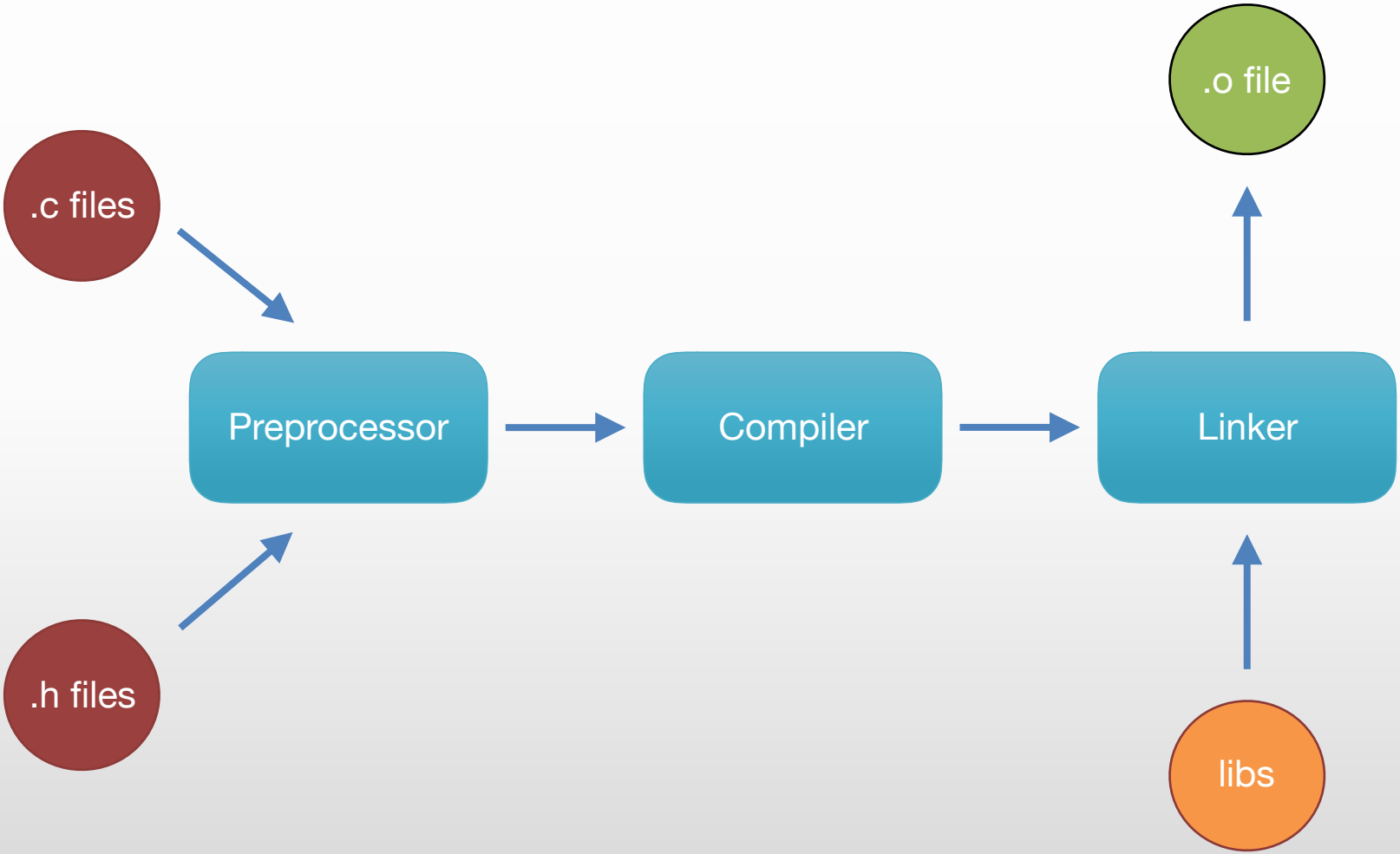
- The OS needs specific tasks such as sendPacket() and receivePacket(). We can implement our version and pass to a function by reference.
- Another example is to implement specific comparator or iterator such as is common in Java Programming.

```
void myproc(int d) {}

void mycaller(void (*f)(int), int param) {
  f(param); /* call function f with param */
}

void main(void) {
  mycaller(myproc, 10); /* call myproc using mycaller */
}
```

# Compiling phase



# Preprocessor commands

We can use special functions of preprocessors to help us

- **Import modules**

```
/* include standard library declaration */
#include <stdio.h>
/* include custom declarations */
#include “myheader.h”
```

- **Symbol definition**

```
#define DEBUG 0
#define MAX_LIST_LENGTH 100
if (DEBUG)
  printf("Max length of list is %d.\n", MAX_LIST_LENGTH);
```

# Preprocessor commands (2)

- **Conditional compilation**

```
#ifdef DEBUG
  printf("DEBUG: line " _LINE_ " has been reached.\n");
#endif

#ifndef HEADER_H
  #include “header.h”
#endif
```

- **Macro definition**

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))
int res = min(1, 2);
```

# You have always to remember

- To initialize variables before using, especially pointers.
- The life of pointer should be short and equal to the life of the pointed object.
  - Do not return local variables by reference
  - Do not dereference *(*p)* pointer before initialization or deallocation
- C has no error handler except for assert system. You always should do error handling by your hand.
- C require practices, focus and you should always follow standards, convention and what suggested by the operating system or by the platform you are working on. Check Google and StackOverflow.
- On desktop you can use valgrind to check your program or gdb to debug.