Signals, Instruments, and Systems – W2
Part I: Course Organization, Team, and Content
Team beyond this course

Distributed Intelligent Systems and Algorithms Laboratory:

https://disal.epfl.ch

• Instructor: Alcherio Martinoli
• Guest lecturers: Chiara Ercolani, Kagan Erünsal, Faëzeh Rahbar
• Teaching assistants:
  – Chiara Ercolani (Head TA, PhD student)
  – Kagan Erünsal (TA, PhD student)
  – Anwar Quraishi (TA, PhD student)
  – Faëzeh Rahbar (TA, PhD student)
  – Aurélien Brun (Help TA, master student)
• Support staff:
  – Cyrill Baumann (PhD student)
Course Public Pages

https://disal.epfl.ch/teaching/signals_instruments_systems/

• Previous editions (5 years)
• Examples of course projects and lab verification tests
Course Rationale, Content, and Prerequisites
Typical Field Instrumentation for Environmental Monitoring

Ultrasound anemometer

Laser-based disdrometer

Integrated compact weather station
(temperature, humidity, anemometer disdrometer)

Data logger
Local Climate Monitoring

Features:
- Very low sampling frequency < 1Hz
- Very low power consumption: 25mW
- Solar panel
- Radio communication

Sensors:
- Air Temperature and Humidity
- Infrared Surface Temperature
- Anemometer
- Solar Radiation
- Pluviometer
- Soil moisture
- Soil pressure

Visit http://www.sensorscope.ch for further details!
Ecosystem Monitoring

Science
• Understand response of wild populations (plants and animals) to habitats over time.
• Develop in situ observation of species and ecosystem dynamics.

Techniques
• Data acquisition of physical and chemical properties, at various spatial and temporal scales, appropriate to the ecosystem, species and habitat.
• Automatic identification of organisms (current techniques involve close-range human observation).
• Measurements over long period of time, taken in-situ.
• Harsh environments with extremes in temperature, moisture, obstructions, ...

Source: D. Estrin, UCLA
Embedded Systems at the Heart of Modern Environmental Engineering

• Moving the lab to the field is “in”!

• Most of the applications require large spatial distributions (scale of the domain >> scale of a node) → sensor networks

• The underlying hardware/software technology (at the single device level) share the same principles
What These Systems Share at their Core?

- Sensing
- Processing
- Mobility

Communication

- Processing
- Visualization
- Storing

In-situ instrument

Transportation channel

Base station

The goal of this course is to shed light on this process and blocks!
What This Course Is About

• Fundamentals of computer science:
  – Basics of computer architecture
  – C programming consolidation (vs. Matlab)
  – C for embedded and real-time systems

• Fundamentals of signal processing:
  – Analog/digital signals, sampling and reconstruction
  – Time/frequency domains and transforms
  – Filters, converters

• Fundamentals of embedded systems:
  – Microprocessors, microcontrollers, memory
  – Sensors and actuators
  – Basic control and communication techniques and concepts
Prerequisites and Related Courses

• Fundamentals in C and Matlab programming
• Fundamentals in probability calculus
• Fundamentals in analysis (Analysis I to IV, ODE, Fourier series and analysis, transforms)
• The Introduction to Control of Dynamical Systems course can help
• ICC can also help with fundamentals in signal processing (though only time domain)
• Some overlap with Quantitative Method II for signal processing fundamentals
Rationale (1)

- This course must allow you to become a power user of the key instruments in environmental engineering used nowadays (sensor networks, meteorological stations, data loggers, etc.) and in the near future (drones, remotely operated vehicles, robotic sensor networks, etc.)

- Being a power user means not only being an advanced user but also understand enough to collaborate with EE/CS/ME engineers to design the instruments of the future in environmental engineering; idea: complexity shifted more and more from hardware to software → a lot of domain knowledge can be transferred to the instruments at software level
Rationale (2)

• Well-balanced course: theory, algorithms, tools and practical exercises

• It should prepare you to better follow a number of master courses (especially in the new Environmental Monitoring and Modeling specialization)

• This course can be considered as an elevator to the master course “Distributed Intelligent Systems” for SIE students where various other sections and programs attend

• It should get you prepared for carrying out a design/semester/master project at DISAL
Organization of the Course
Credits and Workload

• 5 ECTS
• 1 ECTS = 30 h workload → 150 h workload total
• Rough breakdown
  – 60 h lecture (including reading, exam preparation)
  – 45 h lab (including verification test and its preparation)
  – 45 h course project (including implementation, reporting, and defense)
Grade

• Final written exam: 180 minutes
• 50% performance during semester, 50% performance during the exam (compromise US/Europe style)
• During semester: lab verification test (20%, tested material of 6 labs, about 30 h effort) and course project (30%)
• During final exam: all covered material is subject to examination, but the exam is open book (no electronic equipment allowed other than a basic, non programmable pocket calculator)
Lecture Notes & Reading

• Policy: no manuscript, but slides and your own notes
• Preliminary lecture slides in pdf format available for download on the course web site before each lecture (wed evening), definitive ones after lecture (couple of days max); e-mail notification only when definitive slides are posted
• Reading will be added on Moodle as the course progresses
• Access to Moodle: in principle you should be all set because compulsory course and therefore inclusion by default; if issues contact sis-ta@groupes.epfl.ch
Tools used in Exercises and Course Projects

• C programming environment
• Matlab
• High-fidelity robotic simulator (Webots)
• Real devices (desktop robots: e-puck; desktop sensor nodes: DISAL Arduino Xbee node)
• OS: UNIX (Linux)
e-puck Robot (Labs & Projects)

Computation and memory

Communication

Actuators

Sensors

I/O
Webots (Labs & Projects)

- High-fidelity simulation
- Discrete sensor and actuators
- Noise and nonlinear characteristics faithfully reproduced
- Different trade-offs faithfulness/computational cost
DISAL Arduino Xbee Node

- Arduino Mega 2560 board (ATMega 2560 microcontroller)
- Zigbee-complaint transceiver (Xbee)
- On-board mini-display
- Sensors:
  - Light sensor (TSL2561-T, ams)
  - Humidity and Temperature sensor (SHT20, Sensirion)
  - Digital Accelerometer (MMA8652, NXP)
- 14 hours autonomy fully on (70 mA on 1000 mAh Li-Po battery)
- Can be programmed in C leveraging Arduino libraries
Labs (1)

• Lab session 15:15-18:00 on Tue, GR B0 01
• Mini-tutorial (5-10 min) by the main lab designer at the beginning of the lab
• 10 lab sets total
• Official solution available on Moodle after the lab session
• sis-ta@groupes.epfl.ch for ANY issue!
Labs (2)

- Lab assignment posted by Monday of the same week at latest

- Lab presence not compulsory but
  - We do not repeat labs
  - If you do not come to labs you will have a hard time in the lab verification test and being efficient in the course project
  - Certain labs involve HW which we do not distribute outside the lab sessions

- I think it is a lot of fun and really helps you understand lecture stuff

- Assistants are well prepared (2 TAs, designer and tester, and help TA)

- **Dedicated feedback forms** for each lab on Moodle
Lab Verification Test

- Worked well in the last 6 editions of our courses
- In Week 8 (concerns content of Lab 1-6)
- 3 h duration, during exercise session (in the computer room)
- Graded and individually reviewed (no official solution)
- **Note:** content of Lab 7-10 will be exploited in the course projects and verified in the final exam
Suggestions for a Successful Exercise Series

From our teaching experience:

• Read the assignment in advance (i.e. before the lab), in this way you will be more efficient when the TAs are around for helping you on the toughest questions…

• Take advantage of office hours (upon appointment using the sis-ta list) if additional explanations are needed

• Do the labs seriously, do not wait for the solution distributed a couple of days later.

• The strategy above will avoid panic reactions at the lab verification test
Course Project (1)

- Course project list distributed in Week 4, kick-off in Week 7
- 45 h effort, defenses last week of the semester during lecture and lab time
- Team of 3 students (ad hoc solution for numbers of students not divisible by 3)
- Will distribute HW/SW at home
- Short progress report (compulsory but not graded) in Week 10
- Final report to be submitted (pre-established max # of pages and format) by end of Week 14 (Fri May 29)
Course Project (2)

- Final presentation in front of the class (10 minutes + 10 min questions)
- Each of the team members has to present
- Pre-established regular office hours:
  - 1 h kick-off in Week 7
  - 1 h each week for each TA in pre-established slots (Week 8 to 12)
  - Week 13: 3 h of the lab session
Suggestions for a Successful Course Project

From our teaching experience:

• Exploit office hours for the course project in an efficient way (e.g., ask the toughest technical questions but do not ask the project supervisor to debug your code!)

• Plan your effort (milestones, time, constraints, etc.), coordinate your team

• Arrive at the progress report milestone with project objective understood, reading and tool familiarization over, and preliminary implementation results
Final Notes
12th Iteration at ENAC

- Same organization and content as last year; only minor improvement when possible
- Course project: kick-off anticipated of 1 week, defense before report submission, no review of other team’s reports, all finished within the semester
- Last edition in this format
- Next edition in the WS 2020-2021:
  - no C refreshing
  - more embedded C
  - more signal processing
  - more leverage of the DISAL Arduino Xbee nodes
  - no course project but two lab verification tests
Signals, Instruments, and Systems – W2
Part II: C Programming
(Continued)
Functions
Functions

When do we use functions?

• Repetition:
  – If part of the code needs to be repeated several times (more than once)

• Structure:
  – If part of the code seems like a subtask of your complete code
Functions

```c
int main() {
    matrix A =
        createMatrix(3,3);
    A[0][2] = 2.0;
    printMatrix(A);
    destroyMatrix(A);
    return 0;
}
```

```c
int main() {
    int i, j;
    double **A =
        malloc(3*sizeof(double *));
    for (i = 0; i < 3; i++) {
        A[i] =
            malloc(3*sizeof(double *));
        A[i][2] = 2.0;
        for (j = 0; j < 3; j++) {
            printf("%.2f ", A[i][j]);
        }
        printf("\n");
    }
    for (i = 0; i < 3; i++) {
        free(A[i]);
    }
    free(A);
    return 0;
}
```
Functions

- Functions take inputs and return a value

  \[
  \text{type name(type1 arg1, type2 arg2, ...);} \\
  \]

- Examples:

  ```
  double cos(double angle);
  void my_function();
  ```
Functions

• Before using a function, it has to be declared.
• C can only back reference:

```c
#include <stdio.h>

int main(int argc, char *args[]){
    print_hello_world();
    return 0;
}

void print_hello_world(){
    printf("Hello World!");
}
```
Functions

- Functions must be declared using the following syntax:

  ```
  type name(type1 arg1, type2 arg2, ...);
  ```

- Here are some typical examples:

  ```
  int mult(int a, int b);
  double cos(double theta);
  double norm(double* v);
  ```

- Sometimes, you do not want your functions to return a value. You can use the keyword `void`

  ```
  void display_matrix(double** m);
  ```
# Libraries

- Libraries provide special functionality in the form of collections of ready-made functions:

<table>
<thead>
<tr>
<th>Library</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdio.h</td>
<td>printf(const char* format, ...)</td>
</tr>
<tr>
<td>math.h</td>
<td>sqrt(double x)</td>
</tr>
<tr>
<td>time.h</td>
<td>gettimeofday()</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>rand()</td>
</tr>
</tbody>
</table>

**Usage:**

```c
#include <stdlib.h>
#include "my_library.h"  // your own collection of function declarations
```
Argument passing in C

- Arguments are always passed *by value* in C function calls. This means that **local copies** of the values of the arguments are passed to the routines.

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(a, b);

    printf("Main: a = %d, b = %d\n", a, b);

    return 0;
}
```

```
computer:~> ./exchange
Exchange: a = 7, b = 5
Main: a = 5, b = 7
```
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(a, b);

    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}

Output:
computer:~> ./exchange
What happens?

#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;
  exchange(a, b);
  printf("Main: a = %d, b = %d\n", a, b);
  return 0
}

Output:

computer:~> ./exchange
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}
```

Output:
```bash
computer:~> ./exchange
```

Computer memory:
```
| a = 5 |
| b = 7 |
```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    
exchange(a,b);

    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}
```

Output:
```
computer:~> ./exchange
```

#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}

Output:
computer:~> ./exchange
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}
```

Output:
```bash
computer:~> ./exchange
```

```plaintext
Exchanged Values:

<table>
<thead>
<tr>
<th>Memory Area</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>7</td>
</tr>
<tr>
<td>tmp</td>
<td>5</td>
</tr>
</tbody>
</table>
```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;  // Error: copied arguments
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}
```

Output:
```
computer:$ ./exchange
```

#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}

Output:
computer:~> ./exchange
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(a,b);

    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}
```

Output:
```bash
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
```
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}

Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(a, b);

    printf("Main: a = %d, b = %d\n", a, b);
    return 0
}
```

Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```
How to solve the problem?

- By using **pointers**, i.e. variables that contain the address of another variable

```c
#include <stdio.h>

void exchange(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
    printf("Exchange: a = %d, b = %d\n", *a, *b);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(&a,&b);

    printf("Main: a = %d, b = %d\n", a, b);

    return 0;
}
```

**Output:**

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 7, b = 5
```

**int *a and int *b are pointers!**
Variable scope: local and global

- Any variable has a **scope**, i.e. a region where this variable can be used (read and/or write).
- In C, since variables must be declared at the beginning of the function, the scope of a variable is the function block:

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

What about this `b`? It is a different variable, with a different scope!

- The scope of a variable does not extend beyond function calls
- Use global variables if you want to use a **unique** variable in multiple functions.
Global variables

- A variable is **global** when it is declared outside of any block.
- Generally, try to **avoid using them**! If you want to use a constant value (known at compile time), rather use a **symbolic constant**.
- Using symbolic constants is way more efficient and allows the compiler to perform a better optimization of your code, but you cannot change the value of this constant in the code.

```c
#include <stdio.h>

int unit_cost = 10; // global variable

int total_cost(int units) {
    return unit_cost * units;
}

int main() {
    int units = 12;
    int total = 0;

    total = total_cost(units);

    printf("%d units at %d CHF each cost %d CHF\n", units, unit_cost, total);

    return 0;
}
```

```c
#define UNIT_COST 10 // symbolic constant

int total_cost(int units) {
    return UNIT_COST * units;
}

int main() {
    int units = 12;
    int total = 0;

    total = total_cost(units);

    printf("%d units at %d CHF each cost %d CHF\n", units, UNIT_COST, total);

    return 0;
}
```
Example: \pi

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

double compute_pi(int p);

int main(int argc, char *args[]){
    int precision;
    double pi;
    if (argc < 2) {
        fprintf(stderr, "Usage: %s
            [precision]\n", args[0]);
        return -1;
    }
    precision = atoi(args[1]);
    pi = compute_pi(precision);
    printf("The final value is %.6f\n", pi);
    printf("The real value is %.6f\n", M_PI);
    return 0;
}
```

1. include the needed header files
2. declare the functions
3. main
4. declare needed variables at the beginning of the block
5. call your function
6. return
Example: $\pi$

```c
double compute_pi(int p){
    int i;
    int inside = 0;
    double ratio;

    srand(time(NULL));

    for (i = 0; i < p; i++) {
        double x = 2.0*(double)(rand() - RAND_MAX/2)/(double)RAND_MAX;
        double y = 2.0*(double)(rand() - RAND_MAX/2)/(double)RAND_MAX;
        if (x*x + y*y < 1) inside++;
    }

    ratio = (double)inside/(double)p;
    return ratio*4.0;
}
```

- **6** write the declared function
- **7** declare variables
- **8** return value
Arrays

• To declare an array:
  – type `name[size];`
  – e.g. `double vector[3];`
  – or `double matrix[4][6];`

• To access:
  – `name[index]`
  – e.g. `vector[1] = 4.5;`
  – or `double a = matrix[0][2];`

Remark: indices start at 0 (not at 1 like Matlab).
Arrays

- For an image, you can use a 2D array

```java
Double epuck[640][480];

And you can use nested loops to parse and process this image:

```java
double epuck2[640][480];

for (i = 0; i < 640; i++) {
    for (j = 0; j < 480; j++) {
        epuck2[640-i-1][j] = epuck[i][j];
    }
}
```

What is the transformation performed by this program?
Example: Arrays

```c
int main() {
    int i, j;
    double A[3][3];

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            A[i][j] = 0;
        }
    }

    A[0][2] = 2.0;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%.2f ", A[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```
Strings

- There is no string type in C.
- Strings (sequences of characters) are represented by “arrays” of chars terminated by the character zero '\0'.
- C offers some specialized functions on this type of strings (see string.h, e.g.: strlen, strcmp).
- More details in the next lecture.
Types

• Variables have types that are fixed and checked by the compiler (however, the compiler will perform implicit conversions where possible – watch out!)
  – Example:   int a = 3.1415 / 2.0; // result: 1 (integer)

• Define your own types with `typedef`:

```c
double calc_vel(int dt, double x); // built-in types

better:

typedef int milliseconds_t;
typedef double distance_t;
typedef double velocity_t;
[...]
velocity_t calc_vel(milliseconds_t dt, distance_t x);  
```
Structures

• Bundle of variables

• Just like any other variable a structure needs to be declared before being used:

```c
typedef struct {
    double x;
    double y;
    double angle;
} pose_t;
pose_t p;
```

• The new structure represents a new variable type.
Structures

• To declare a variable from a structure:
  \[- \text{struct\_name } \text{name}; \]
  \[- \text{e.g. } \text{pose\_t } \text{vehicle\_position}; \]

• To access part of a structure:
  \[- \text{name.} \text{member} \]
  \[- \text{e.g. } \text{vehicle\_position.} \text{x } = \text{3.0}; \]
  \[- \text{or } \text{double } \text{a } = \text{vehicle\_position.} \text{angle}; \]

• Assignment ("=") copies entire structure
Example: Structures

typedef struct {
   double x;
   double y;
} vec2;

double scalar_prod(vec2 v1, vec2 v2) {
   return v1.x*v2.x + v1.y*v2.y;
}

int main() {
   vec2 v1, v2;
   v1.x = 1.0;
   v1.y = 1.0;
   v2.x = 0.0;
   v2.y = 2.0;
   printf("Scalar product equal to %.2f\n", scalar_prod(v1, v2));
   return 0;
}
File Organization

• Group functions into source files by theme
• Declare related functions in the corresponding header file

```
matrix.h

#ifndef _MATRIX_H
#define _MATRIX_H

matrix_t transpose(matrix_t A);
void print(matrix_t A);

#endif

#include "matrix.h"

matrix.t transpose(matrix_t A) {
  ...
}

void print(matrix_t A) {
  ...
}
```
Example: Matrices (1)

• Example of all elements you have seen until now

• we will create:
  – a minimalist matrix-library
  – a main function that uses it
  – a Makefile that compiles it
#ifndef _MATRIX_H
#define _MATRIX_H

#define MAX_ROWS 100
#define MAX_COLS 100

typedef struct {
    double values[MAX_ROWS][MAX_COLS];
    unsigned int nrows;
    unsigned int ncols;
} matrix_t;

matrix_t transpose(matrix_t A);
matrix_t add(matrix_t A, matrix_t B);

void print(matrix_t A);

#endif
Example: Matrices (2)

```c
#include "matrix.h"
#include <stdio.h>

matrix_t transpose(matrix_t A) {
    matrix_t B;
    unsigned int i, j;
    B.nrows = A.ncols;
    B.ncols = A.nrows;
    for (i = 0; i < A.nrows; i++) {
        for (j = 0; j < A.ncols; j++) {
            // Simply assign columns to rows
            B.values[j][i] = A.values[i][j];
        }
    }

    return B;
}
```
matrix_t add(matrix_t A, matrix_t B){
  matrix_t C;
  unsigned int i, j;

  C.nrows = 0;
  C.ncols = 0;
  // Sanity check
  if (A.nrows != B.nrows || A.ncols != B.ncols) return C;

  C.nrows = A.nrows;
  C.ncols = A.ncols;

  for (i = 0; i < A.nrows; i++) {
    for (j = 0; j < A.ncols; j++) {
      C.values[i][j] = A.values[i][j] + B.values[i][j];
    }
  }
  return C;
}
Example: Matrices (4)

```c
void print(matrix_t A) {
        unsigned int i, j;

        for (i = 0; i < A.nrows; i++) {
            printf("   |\n"),
            for (j = 0; j < A.ncols; j++) {
                printf("%8.2f ", A.values[i][j]);
            }
            printf("|\n")
        }
        printf("|\n");
        return;
    }
```
#include <stdio.h>
#include "matrix.h"

int main(int argc, char *args[]) {
    matrix_t A, B, C;
    A.nrows = 2;   A.ncols = 2;
    B.nrows = 2;   B.ncols = 2;
    A.values[0][0] = 1.0;   A.values[0][1] = 2.0;
    A.values[1][0] = 3.0;   A.values[1][1] = 4.0;
    B.values[0][0] = 1.0;   B.values[0][1] = 2.0;
    B.values[1][0] = 3.0;   B.values[1][1] = 4.0;
    printf("A = \n");   print(A);
    printf("B = \n");   print(B);
    C = add(A,B);
    printf("A + B = \n");   print(C);
    C = add(transpose(A), B);
    printf("A' + B = \n");   print(C);
    return 0;
}
Example: Matrices (6)

Makefile

CC = gcc

main: matrix.o main.o

clean:
  rm -f *.o main

Build and run

>make
>./main
### Example: Matrices (7)

```plaintext
A =
|   1.00   2.00 |
|   3.00   4.00 |

B =
|   1.00   2.00 |
|   3.00   4.00 |

A + B =
|   2.00   4.00 |
|   6.00   8.00 |

A' + B =
|   2.00   5.00 |
|   5.00   8.00 |
```
Conclusion
Summary

• You have seen almost all basics of C
• Next week, you will see *pointers* and *dynamic memory allocation*