ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Signals, Instruments, and Systems – W2

C Programming (continued)

Resources

Remember `man`?

- You can use `man` to show information about functions in the standard libraries of C:
  - e.g. `man printf`
  - or `man atan2`
- To type in the terminal!

`man` or not `man`?

- `man` pages can be confusing at first, but they are worth it.
- Alternatively, you can use:
  - **Google**: “[function name] cplusplus”
  - http://www.cplusplus.com/reference/clibrary/
  - ex: http://www.cplusplus.com/reference/clibrary/cs**tdio/printf/**

    stdio.h `printf` function

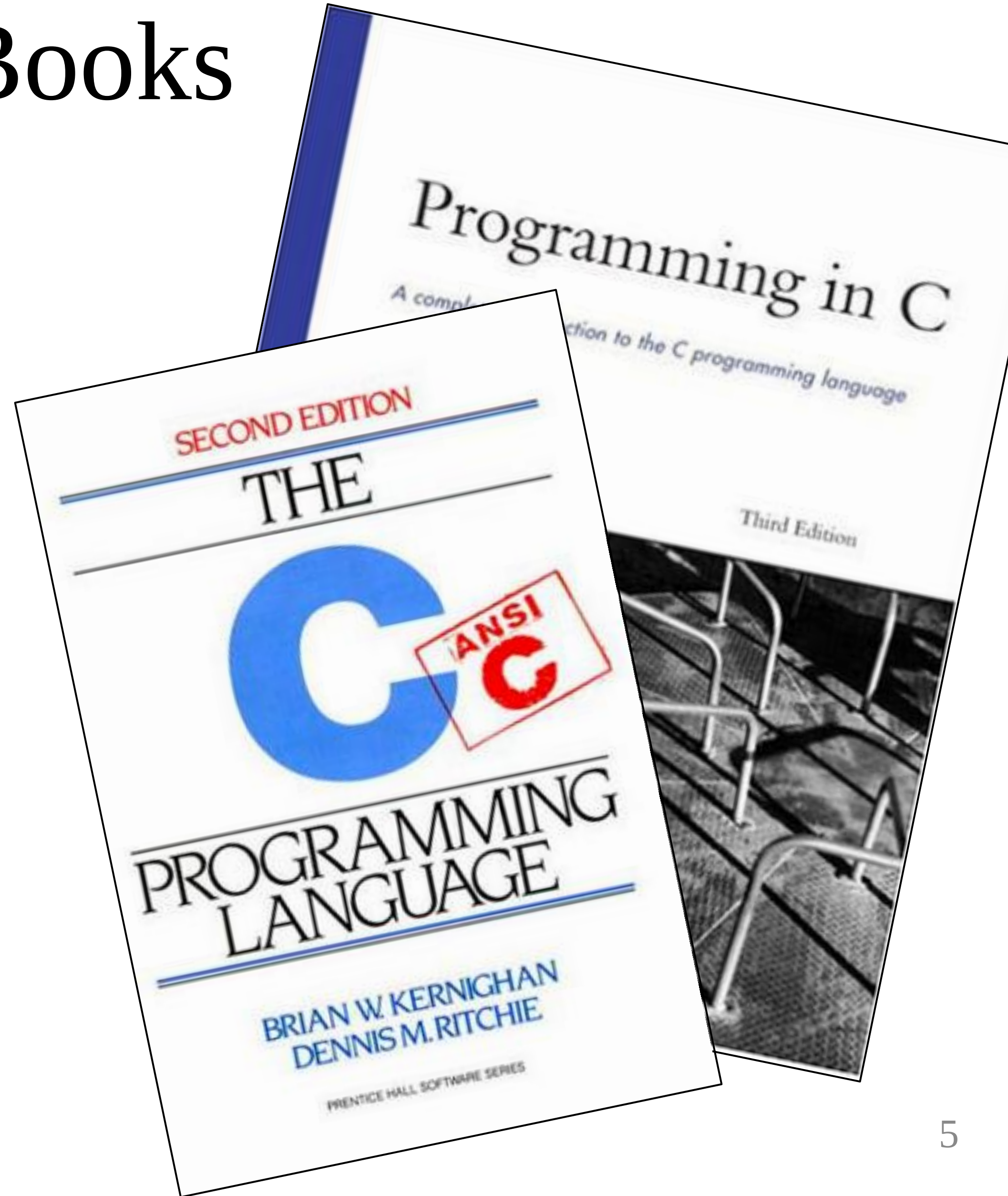
Books

Programming in C
Stephen G. Koch

ISBN-13: 978-0672326660

C Programming Language
Brian W. Kernighan,
Dennis M. Ritchie

ISBN-13: 978-0131103627



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

c

Operators - Logical

Operator	Meaning
a < b	less than
a <= b	less than or equal
a > b	greater than
a >= b	greater than or equal
a == b	equal to
a != b	not equal to
a && b	logical AND
a || b	logical OR

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Operators - Arithmetic

Operator	Meaning
a + b	addition
a - b	subtraction
a * b	multiplication
a / b	division
a % b	modulo (integer remainder)

Operators - Shortcuts

Operator	Meaning
a += b	addition
a -= b	subtraction
a *= b	multiplication
a /= b	division
a %= b	modulo (integer remainder)

Operators - Unary

Operator	Meaning
a++	postfix increment
++a	prefix increment
a--	postfix decrement
--a	prefix decrement

Operators - Unary

```
#include <stdio.h>

int main() {
  int i = 0;
  while (i++ < 3) {
    printf("iteration %d\n", i);
  }

  return 0;
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Operators - Unary

```
#include <stdio.h>

int main() {
  int i = 0;
  while (++i < 3) {
    printf("iteration %d\n", i);
  }

  return 0;
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR
~a	bitwise NOT

Question:
20 << 2 = ?

Note: Bitwise calculus is used in advanced code (compression, encryption, or optimizations) and also in embedded systems

Binary numbers

0	0	1	0	1	0	1	1

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1	1

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Disal

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	1

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	1
0	0	32	0	8	0	2	1

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	1

$$0 + 0 + 32 + 0 + 8 + 0 + 2 + 1 = 43$$

Operators - Bitwise

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>a << b</td><td>left shift</td></tr>
<tr><td>a >> b</td><td>right shift</td></tr>
<tr><td>a | b</td><td>bitwise OR</td></tr>
<tr><td>a & b</td><td>bitwise AND</td></tr>
<tr><td>a ^ b</td><td>exclusive OR</td></tr>
</table>

Question:

20 << 2 = ?

20 >> 2 = ?

20 | 6 = ?

20 & 6 = ?

20 ^ 6 = ?

Hint:

10100 << 2 = 1010000

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	
20 | 6 = ?		
20 & 6 = ?		
20 ^ 6 = ?		

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	
20 & 6 = ?		
20 ^ 6 = ?		

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	
20 & 6 = ?		
20 ^ 6 = ?		

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	22
20 & 6 = ?	10100 & 00110 = 00100	
20 ^ 6 = ?		

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	22
20 & 6 = ?	10100 & 00110 = 00100	4
20 ^ 6 = ?	10100 ^ 00110 = 10010	

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	22
20 & 6 = ?	10100 & 00110 = 00100	4
20 ^ 6 = ?	10100 ^ 00110 = 10010	18

Functions

- Repetition:
  - If part of the code needs to be repeated several times (more than one), create a function!
- Structure:
  - If part of the code seems like a subtask of your complete code, create a function!

Functions: why?

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

=

```
int main() {
  int i, j;
  double **A =
    malloc(3*sizeof(double *));
  for (i = 0; i < 3; i++) {
    A[i] =
      malloc(3*sizeof(double *));
  }
  A[0][2] = 2.0;
  for (i = 0; i < 3; i++) {
    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: how?

- Functions can return a value
- Functions can also take inputs.

```
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:

```
#include <stdio.h>

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

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

VS

```
#include <stdio.h>

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

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

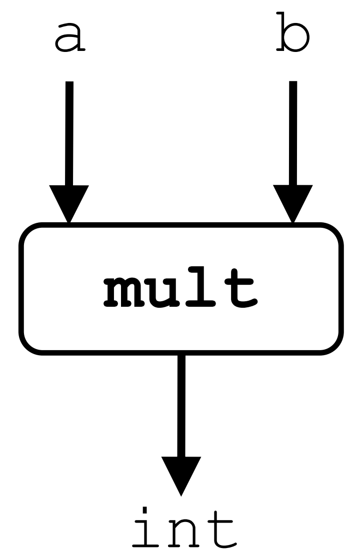
Functions

- 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:

Library:	**Example:**
`stdio.h`	`printf(const char* format,…)`
`math.h`	`sqrt(double x)`
`time.h`	`gettimeofday()`
`stdlib.h`	`rand()`

Usage:

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

`#include "my_library.h"` : your own collection of function declarations

Argument passing in C

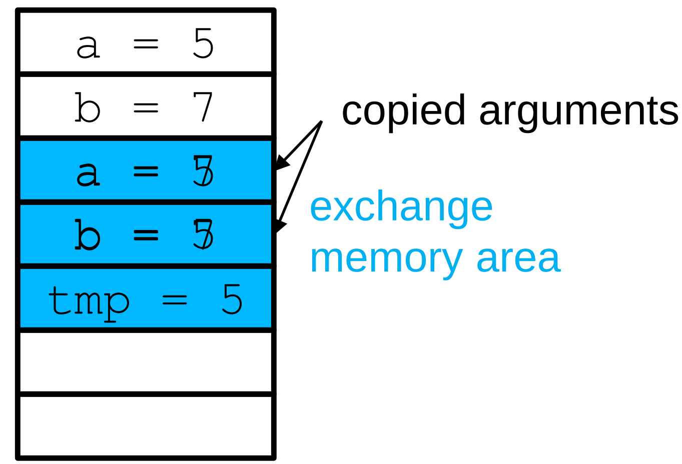
- Arguments are always passed *by value* in C function calls! This means that **local copies** of the values of the arguments are passed to the routines!

```
#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
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```

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
}
```



Computer memory

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!

```
#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:

```
#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!

scope of b

- 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
#include <stdio.h>

#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: π

```
#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 functionalities
2. declare the functions
3. main
4. declare needed variables at the beginning of the block
5. call your function
6. return

Example: π

```
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:
  - Def.: `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** 1 like Matlab).

Arrays

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

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

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

```
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

```
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:

```
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

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

```
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:
  - *`struct_name`* ***`name;`***
  - e.g. `pose_t vehicle_position;`
- To access part of a structure:
  - *`name.member`*
  - e.g. `vehicle_position.x = 3.0;`
  - or `double a = vehicle_position.angle;`
- Assignment ("=") copies entire struct

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
```

matrix.c

```
#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

Example: Matrices (2)

matrix.h

```
#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)

matrix.c (1)

```
#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;
}
```

Example: Matrices (3)

matrix.c (2)

```
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)

matrix.c (3)

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

  for (i = 0; i < A.nrows; i++) {
    printf("   |");
    for (j = 0; j < A.ncols; j++) {
      printf("%8.2f ", A.values[i][j]);
    }
    printf("|\n");
  }

  return;
}
```

Example: Matrices (5)

main.c

```
#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)

stdout

```
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***