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”

  stdio.h   printf function
Books

Programming in C
Stephen G. Koch

C Programming Language
Brian W. Kernighan, Dennis M. Ritchie
Main differences with Matlab

• Matlab is an interpreted language
  – Interpreted: The code is transformed into machine executable code at run-time

• Matlab is optimized for matrix operations

• Syntax differences (loops, functions, etc…)

• No variable declarations
## Operators - Logical

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>less than</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>less than or equal</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>greater than</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>a == b</td>
<td>equal to</td>
</tr>
<tr>
<td>a != b</td>
<td>not equal to</td>
</tr>
<tr>
<td>a &amp;&amp; b</td>
<td>logical AND</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
### Operators - Arithmetic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>addition</td>
</tr>
<tr>
<td>a - b</td>
<td>subtraction</td>
</tr>
<tr>
<td>a * b</td>
<td>multiplication</td>
</tr>
<tr>
<td>a / b</td>
<td>division</td>
</tr>
<tr>
<td>a % b</td>
<td>modulo (integer remainder)</td>
</tr>
</tbody>
</table>
## Operators - Shortcuts

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a += b</td>
<td>addition</td>
</tr>
<tr>
<td>a -= b</td>
<td>subtraction</td>
</tr>
<tr>
<td>a *= b</td>
<td>multiplication</td>
</tr>
<tr>
<td>a /= b</td>
<td>division</td>
</tr>
<tr>
<td>a %= b</td>
<td>modulo (integer remainder)</td>
</tr>
</tbody>
</table>
# Operators - Unary

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a++</td>
<td>postfix increment</td>
</tr>
<tr>
<td>++a</td>
<td>prefix increment</td>
</tr>
<tr>
<td>a--</td>
<td>postfix decrement</td>
</tr>
<tr>
<td>--a</td>
<td>prefix decrement</td>
</tr>
</tbody>
</table>
Operators - Unary

```c
#include <stdio.h>

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

    return 0;
}
```
Operators - Unary

```c
#include <stdio.h>

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

    return 0;
}
```
# Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR</td>
</tr>
<tr>
<td>~a</td>
<td>bitwise NOT</td>
</tr>
</tbody>
</table>

**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

\[ \begin{array}{cccccccc}
2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
\end{array} \]
Binary numbers

<table>
<thead>
<tr>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
# Binary numbers

<table>
<thead>
<tr>
<th></th>
<th>2^7</th>
<th>2^6</th>
<th>2^5</th>
<th>2^4</th>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

The binary representation of the number 180 is 001110011.
Binary numbers

$$2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

$$128 \quad 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1$$

$$0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1$$

$$0 + 0 + 32 + 0 + 8 + 0 + 2 + 1 = 43$$
# Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR</td>
</tr>
</tbody>
</table>

**Question:**

- 20 << 2 = ?
- 20 >> 2 = ?
- 20 | 6 = ?
- 20 & 6 = ?
- 20 ^ 6 = ?

**Hint:**

- 10100 << 2 = 1010000
Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR OR</td>
</tr>
</tbody>
</table>

Question:  
20 << 2 = ?  
20 >> 2 = ?  
20 | 6 = ?  
20 & 6 = ?  
20 ^ 6 = ?

Hint:  
10100 << 2 = 1010000  
10100 >> 2 = (00)101

Answer:  
80
# Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR</td>
</tr>
</tbody>
</table>

## Question:

- 20 << 2 = ?
- 20 >> 2 = ?
- 20 | 6 = ?
- 20 & 6 = ?
- 20 ^ 6 = ?

## Hint:

- 10100 << 2 = 1010000
- 10100 >> 2 = (00)101
- 10100 | 00110 = 10110

## Answer:

- 80
- 5
Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR</td>
</tr>
</tbody>
</table>

**Question:**
- $20 << 2 = ?$
- $20 >> 2 = ?$
- $20 | 6 = ?$
- $20 & 6 = ?$
- $20 ^ 6 = ?$

**Hint:**
- $10100 << 2 = 1010000$
- $10100 >> 2 = (00)101$
- $10100 | 00110 = 10110$

**Answer:**
- $80$
- $5$
Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR</td>
</tr>
</tbody>
</table>

Question:  
20 << 2 = ?  
20 >> 2 = ?  
20 | 6 = ?  
20 & 6 = ?  
20 ^ 6 = ?

Hint:  
10100 << 2 = 1010000  
10100 >> 2 = (00)101  
10100 | 00110 = 10110  
10100 & 00110 = 00100

Answer:  
80  
5  
22
## Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR</td>
</tr>
</tbody>
</table>

### Question:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Hint</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 &lt;&lt; 2 = ?</td>
<td>10100 &lt;&lt; 2 = 1010000</td>
<td>80</td>
</tr>
<tr>
<td>20 &gt;&gt; 2 = ?</td>
<td>10100 &gt;&gt; 2 = (00)101</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>6 = ?</td>
<td>10100</td>
</tr>
<tr>
<td>20 &amp; 6 = ?</td>
<td>10100 &amp; 00110 = 00100</td>
<td>4</td>
</tr>
<tr>
<td>20 ^ 6 = ?</td>
<td>10100 ^ 00110 = 10010</td>
<td></td>
</tr>
</tbody>
</table>
# Operators - Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>right shift</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>a ^ b</td>
<td>exclusive OR</td>
</tr>
</tbody>
</table>

## Question:

<table>
<thead>
<tr>
<th>Question</th>
<th>Hint</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 &lt;&lt; 2 = ?</td>
<td>10100 &lt;&lt; 2 = 1010000</td>
<td>80</td>
</tr>
<tr>
<td>20 &gt;&gt; 2 = ?</td>
<td>10100 &gt;&gt; 2 = (00)101</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>6 = ?</td>
<td>10100</td>
</tr>
<tr>
<td>20 &amp; 6 = ?</td>
<td>10100 &amp; 00110 = 00100</td>
<td>4</td>
</tr>
<tr>
<td>20 ^ 6 = ?</td>
<td>10100 ^ 00110 = 10010</td>
<td>18</td>
</tr>
</tbody>
</table>
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?

```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[0][2] = 2.0;
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 3; j++) {
                printf("%.2f ", A[i][j]);
            }
            printf("\n");
        }
        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.
• Unlike Java, C can only back reference:

```c
#include <stdio.h>

int main(int argc, char *args[]){
    printf("Hello World!");
    return 0;
}

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

```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:

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

- Here are some typical examples:

  \[
  \text{int \ mult(int \ a, \ int \ b);} \\
  \text{double \ cos(double \ theta);} \\
  \text{double \ norm(double* \ v);} \\
  \]

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

  \[
  \text{void \ display_matrix(double** \ m);} \\
  \]
Libraries

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

**Library:**

- `stdio.h`
- `math.h`
- `time.h`
- `stdlib.h`

**Example:**

- `printf(const char* format,...)`
- `sqrt(double x)`
- `gettimeofday()`
- `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!

```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
computer:~> Exchange: a = 7, b = 5
computer:~> Main: 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
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;  // scope of b
    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
#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: \( \pi \)

```
#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
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;
}
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

```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:**

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

• Structures can potentially replace objects (from Java).

• 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:
  – `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

```c
matrix.h

#ifndef _MATRIX_H
#define _MATRIX_H

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

```c
#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
```

`matrix.h`
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;
}
```
Example: Matrices (3)

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

<table>
<thead>
<tr>
<th>Makefile</th>
<th>Build and run</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC = gcc</td>
<td></td>
</tr>
<tr>
<td>main: matrix.o main.o</td>
<td></td>
</tr>
<tr>
<td>clean:</td>
<td></td>
</tr>
<tr>
<td>rm -f *.o main</td>
<td></td>
</tr>
<tr>
<td>&gt;make</td>
<td></td>
</tr>
<tr>
<td>&gt;./main</td>
<td></td>
</tr>
</tbody>
</table>
Example: Matrices (7)

\[
\begin{align*}
A &= \begin{pmatrix} 1.00 & 2.00 \\ 3.00 & 4.00 \end{pmatrix} \\
B &= \begin{pmatrix} 1.00 & 2.00 \\ 3.00 & 4.00 \end{pmatrix} \\
A + B &= \begin{pmatrix} 2.00 & 4.00 \\ 6.00 & 8.00 \end{pmatrix} \\
A' + B &= \begin{pmatrix} 2.00 & 5.00 \\ 5.00 & 8.00 \end{pmatrix}
\end{align*}
\]
Conclusion
Summary

• You have seen almost all basics of C
• Next week, you will see pointers and dynamic memory allocation
• C resembles Java in syntax, but not in execution.
• Matlab is a very useful mathematical tool, but is weak for real-time processing, and not suitable for embedded systems