Outline

1. More on statements
2. Arrays
   - General arrays
   - Exercises
   - Strings
3. Functions
   - Function definition and declaration
   - Exercises
4. Visibility, scope and lifetime
5. Structures
6. Casting
7. More on input/output
   - Files
   - Exercises
do while loop

- An alternative way to write a loop is to use the do – while loop

```
do {
    statement1;
    statement2;
    ...
} while(condition);
```

- The main difference between the while and the do – while is that
  - in the while loop the condition is evaluated before every iteration,
  - in the do – while case the condition is evaluated after every iteration

- Hence, with do – while the loop is always performed at least once

Nested loops

- It is possible to define a loop inside another loop. This is very useful in many cases in which we have to iterate on two variables
- What does the following program do?

```
dloop.c

int main()
{  
    int i, j;
    
    printf("%d\n", 2);
    
    for (i = 3; i <= 100; i = i + 1) {
        for (j = 2; j < i; j = j + 1) {
            if (i % j == 0) break;
            if (j > sqrt(i)) {
                printf("%d\n", i);
                break;
            }
        }
    }
    return 0;
}
```
Exercises

1. Write the equivalence between `while` and `do - while`.
2. Write the equivalence between `for` and `do - while`.
3. Write a program that, given two numbers, finds all common factors between them.
   - Example 1: 12 and 15, will output 3
   - Example 2: 24 and 12, will output 2, 3, 4, 6

Reading C programs

It is very important to be able to learn how to read C programs written by someone else.

- Please, take your time to read programs!
- You must look at a program as you were the processor: try to “execute a program” on paper, writing down the values of the variables at every step.
- Also, please try to write “clean” programs!
  - so that other programs will find easy to read your own programs.
Switch - case

Sometimes, we have to check several alternatives on the same value; instead of a sequence of if-then-else, we can use a switch case statement:

```c
int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    switch(number) {
        case 0 :
            printf("None
");
            break;
        case 1 :
            printf("One
");
            break;
        case 2 :
            printf("Two
");
            break;
        case 3 :
        case 4 :
        case 5 :
            printf("Several
");
            break;
        default :
            printf("Many
");
            break;
    }
    return 0;
}
```

Arrays

Instead of single variables, we can declare arrays of variables of the same type

- They have all the same type and the same name
- They can be addressed by using an index

```c
int i;
int a[10];
```

- Very important: If the array has N elements, index starts at 0, and last element is at N-1
- In the above example, last valid element is `a[9]`
You have no more than 5 minutes to complete this exercise!

Modify the previous program, so that the user can specify the number of times the two dices will be rolled.

Check that the user do not inserts a negative number in which case you print out an error and exit.
Index range

- What happens if you specify an index outside the array boundaries?

- The compiler does not complain, but you can get a random run-time error!

- Consider the following program: what will happen?

```c
#include <stdio.h>

int main()
{
    int i;
    int a[10];

    for (i=0; i<15; i++) {
        a[i] = 0;
        printf("a[%d] = %d\n", i, a[i]);
    }

    printf("Initialization completed!\n");

    return 0;
}
```

Questions

- Index out of bounds is a programming error
  - Why the compiler does not complain?
  - Why the program does not complain at run-time?
- What is the memory allocation of the program? Where is the array allocated?
Initialization

Arrays can be initialized with the following syntax

```c
int a[4] = {0, 1, 2, 3};
```

This syntax is only for static initialization, and cannot be used for assignment

```c
int a[4];
      a = {0, 1, 2, 3};  // syntax error!
```

Matrix

Two- and three-dimensional arrays (matrices):

```c
double mat[3][3];
int cube[4][4][4];

mat[0][2] = 3.5;
```

Static and dynamic initialisation

```c
#include <stdio.h>

int main()
{
    int i;
    double mat[3][3] = {
        {0, 0, 0},
        {0, 0, 0},
        {0, 0, 0}
    };

    mat[0][2] = 3.5;
    for (i=0; i<9; i++) {
        mat[i/3][i%3] = 2.0;
    }
    printf("Done\n");
    return 0;
}
```
Exercises

- Given 2 arrays of doubles of length 3 that represents vector in a 3-dimensional space, compute the scalar product and the vectorial product
- Given an array of 30 integers, compute max, min and average

Strings

- There is not a specific type for strings in C
- A string is a sequence of char terminated by value 0
- To store strings, it is possible to use arrays of chars
  \[
  \text{char name[20];}
  \]
- Initialization:
  \[
  \text{char name[20] = "Lipari";}
  \]
  But again, this syntax is not valid for assignments!
- In memory:
  \[
  \begin{array}{cccccccc}
  \text{L} & \text{i} & \text{p} & \text{a} & \text{r} & \text{i} & 0 & \text{Lipari} \\
  \end{array}
  \]
**String length**

- **Important**: if you need a string with 10 characters, you must declare an array of 11 characters! (one extra to store the final 0)

Here is an example of how to compute the length of a string

```c
char s[20];
...
// how many valid characters in s?
int i;
for (i=0; i<20; i++) if (s[i] == 0) break;

if (i<20) printf("String is %d characters long\n", i);
else printf("String is not valid!\n");
```

**String content**

- What is in a string?

```c
#include <stdio.h>

int main()
{
   int i;
   char str[20] = "donald duck"
;
   for (i=0; i<20; i++)
      printf("%d ", str[i]);
   printf("\n");
}
```
String manipulation functions

- `int strcpy(char s1[], char s2[])`;
  - copies string `s2[]` into string `s1`
- `int strcmp(char s1[], char s2[])`;
  - compare strings alphabetically
- `int strcat(char s1[], char s2[])`;
  - append `s2[]` to `s1`
- `int strlen(char s[])`;
  - computes string length
- `printf("%s", str)`;
  - prints string on screen

Safe versions

- Previous functions are not safe: if the string is not well terminated, anything can happen
- There are safe versions of each:
  - `int strncpy(char s1[], char s2[], int n)`;
    - copies at most `n` characters
  - `int strncat(char s1[], char s2[], int n)`;
    - appends at most `n` characters
  - `int strncmp(char s1[], char s2[], int n)`;
    - compares at most `n` characters
Function definition and declaration

A function is defined by:
- a unique name
- a return value
- a list of arguments (also called parameters)
- a body enclosed in curly braces

An example: this function elevates a double number to an integer power

```c
/* returns the power of x to y */
double power(double x, int y)
{
    int i;
    double result = 1;
    for (i=0; i < y; i++)
        result = result * x;
    return result;
}
```
Function call

- This is how the function is called.
- The formal parameters $x$ and $y$ are substituted by the actual parameters (the values of $x$ and $y$).

```c
int main()
{
    double myx;
    int myy;
    double res;

    printf("Enter x and y\n");
    printf("x? ");
    scanf("%lg", &myx);
    printf("y? ");
    scanf("%d", &myy);

    res = power(myx, myy);
    printf("x^y = %lg\n", res);
}
```

Parameters

- Modifications on local parameters have no effect on the caller.

```c
int multbytwo(int x)
{
    x = x * 2;
    return x;
}

int main()
{
    ...
    i = 5;
    res = multbytwo(i);
    /* how much is i here? */
    ...
}
```

- $x$ is just a copy of $i$.
- modifying $x$ modifies the copy, **not** the original value.
- We say that in C parameters are passed **by value**.
- There is only one exception to this rule: arrays
  - An array parameter is never copied, so modification to the local parameter are immediately reflected to the original array.
Array parameters

```c
#include <stdio.h>

void swap (int a[])
{
    int tmp;
    tmp = a[0];
    a[0] = a[1];
    a[1] = tmp;
    return;
}

int main()
{
    int my[2] = {1,5}
    printf("before swap: %d %d", my[0], my[1]);

    swap(my);

    printf("after swap: %d %d", my[0], my[1]);
}
```

- The array is not copied
- modification on array `a` are reflected in modification on array `my`
  - (this can be understood better when we study pointers)
- Notice also:
  - the `swap` function does not need to return anything: so the return type is `void`
  - the array `my` is initialised when declared

Exercises

1. Write a function that, given a string, returns its length
2. Write a function that, given two strings `s1` and `s2`, returns 1 if `s2` is contained in `s1`
3. Write a function that given a string, substitutes all lower case characters to upper case
## Definitions

- **Global variables** are variables defined outside of any function.
- **Local variables** are defined inside a function.
- **The visibility** (or scope) of a variable is the set of statements that can “see” the variable.
  - Remember that a variable (or any other object) must be declared before it can be used.
- **The lifetime** of a variable is the time during which the variable exists in memory.

## Examples

```c
#include <stdio.h>

int pn[100];
int is_prime(int x)
{
    int i, j;
    ...
}

int temp;

int main()
{
    int res;
    char s[10];
    ...
}
```

- `pn` is a global variable
  - Scope: all program
  - Lifetime: duration of the program

- `x` is a parameter
  - Scope: body of function `is_prime`
  - Lifetime: during function execution

- `i, j` are local variables
  - Scope: body of function `is_prime`
  - Lifetime: during function execution

- `temp` is a global variable
  - Scope: all objects defined after temp
  - Lifetime: during function execution

- `res` and `s[]` are local variables
  - Scope: body of function `main`
  - Lifetime: duration of the program
A **global variable** is declared outside all functions
- This variable is created before the program starts executing, and it exists until the program terminates
- Hence, it's **lifetime** is the program duration

The **scope** depends on the point in which it is declared
- All variables and functions defined after the declaration can use it
- Hence, it's scope depends on the position

**Local variables**

Local variables are defined inside functions

```c
int g;

int myfun()
{
    int k; double a;
    ...
}

int yourfun()
{
    ...
}
```

- **k** and **a** are local to **myfun()**
- In function **yourfun()**, it is possible to use variable **g** but you cannot use variable **k** and **a** (out of scope)

**k** and **a** cannot be used in **yourfun()** because their scope is limited to function **myfun()**.
Local variable lifetime

- Local variables are created only when the function is invoked;
- They are destroyed when the function terminates;
  - Their lifetime corresponds to the function execution;
  - Since they are created at every function call, they hold only temporary values useful for calculations.

```c
int fun(int x)
{
    int i = 0;
    i += x;
    return i;
}
int main()
{
    int a, b;
    a = fun(5);
    b = fun(6);
    ...
}
```

- `i` is initialized to 0 at every `fun()` call;
- At this point, `a` is 5 and `b` is 6;

Modifying lifetime

- To modify the lifetime of a local variable, use the `static` keyword.

```c
int myfun()
{
    static int i = 0;
    i++;
    return i;
}
int main()
{
    printf("%d ", myfun());
    printf("%d ", myfun());
    ...
}
```

- This is a static variable: it is initialised only once (during the first call), then the value is maintained across successive calls;
- This prints 1;
- This prints 2;
Hiding

- It is possible to define two variables with the same name in two different scopes.
- The compiler knows which variable to use depending on the scope.
- It is also possible to hide a variable.

```c
int fun1()
{
    int i;
    ...
}

int fun2()
{
    int i;
    ...
    i++;
}
```

---

Structure definition

- In many cases we need to aggregate variables of different types that are related to the same concept.
- Each variable in the structure is called a field.
- The structure is sometimes called record.
- Example.

```c
struct student {
    char name[20];
    char surname[30];
    int age;
    int marks[20];
    char address[100];
    char country[100];
};

struct student s1;
```

```c
struct position {
    double x;
    double y;
    double z;
};

struct position p1, p2, p3;
```
Accessing data

To access a field of a structure, use the *dot notation*

```c
struct student s1;
...
printf("Name: %s\n", s1.name);
printf("Age: %d\n", s1.age);
```

```c
#include <math.h>

struct position p1;
...
p1.x = 10 * cos(0.74);
p1.y = 10 * sin(0.74);
```

Array of structures

It is possible to declare array of structures as follows:

```c
struct student my_students[20];
int i;

my_student[0].name = "...";
my_student[0].age = "...";
...

for (i=0; i<20; i++) {
    printf("Student %d\n", i);
    printf("Name: %s\n", my_student[i].name);
    printf("Age: %d\n", my_student[i].age);
    ...
}
```
Other operations with structures

- When calling functions, structures are passed by value
  - that is, if you modify the parameter, you modify only the copy, and the original value is not modified
- Initialization: you can use curly braces to initialize a structure

```c
struct point {
    double x;
    double y;
};

struct point x = {0.5, -7.1};
```

Copying structures

- You can use normal assignment between structures of the same type
  - the result is a field-by-field copy

```c
struct point {
    double x;
    double y;
};

struct point x = {4.1, 5.0};

struct point y;

y = x;
```
Converting variables between types

- Sometimes we need to convert a variable between different types
- Example:

  ```c
  int a = 5;
  double x;
  x = a;  // Here we have an implicit conversion from int to double; the compiler does not complain
  x = a / 2;  // Here we have an implicit conversion from int to double. However, the conversion is performed on the result of the division; therefore the result is 2 and not 2.5 as one might expect!
  a = x * 2;  // Here we have a conversion from double to int. With this conversion, we might lose in precision, hence the compiler issues a warning
  ```

Explicit casting

- It is possible to make casting explicit as follows

  ```c
  int a;
  double x;
  x = ((double) a) / 2;  // Here the conversion is not explicit. First, a is converted to double; then, the division is performed (a fractional one); then the result (a double) is assigned to x.
  a = (int)(x * 2);  // Here the compiler does not issue any warning, because the programmer has made it explicit that he/she wants to do this operation.
  ```
In the next slides we will present a quick overview of some functions to manipulate file.
These are useful to solve some exercises.
We will come back to these functions at some point.

Files

A file is a sequence of bytes, usually stored on mass-storage devices.
- We can read and/or write bytes from/to files sequentially (as in magnetic tapes).
- File can contain sequences of bytes (binary) or sequence of characters (text files).
  - There is really no difference: a character is nothing more than a byte.
  - It’s the interpretation that counts.
Before operating on a file, we must *open* it
then we can operate on it
finally we have to *close the file* when we have done

In a C program, a file is identified by a variable of type

```c
FILE *
```

* The * denotes a pointer: we will see next lecture what a pointer is

---

**Opening a file**

To open a file, call *fopen*

```c
FILE *fopen(char *filename, char *mode);
```

* filename and mode are strings
  - filename is the name of the file (may include the path, relative or absolute)
  - mode is the opening mode
    - “r” for reading or “w” for writing or “a” for writing in append mode

Example: open a file in reading mode

```c
FILE *myfile;

myfile = fopen("textfile.txt", "r");
...

fclose(myfile);
```
At this stage, we will consider only text files

You can use fprintf() and fscanf(), similar to the functions we have already seen

```c
#include <stdio.h>

FILE *myfile;

int main()
{
    int a, b, c;
    char str[100];

    myfile = fopen("textfile.txt", "r");

    fscanf(myfile, "%d %d", &a, &b);
    fscanf(myfile, "%s", str);
    fscanf(myfile, "%d", &c);

    printf("what I have read:
");
    printf("a = %d b = %d c = %d\n", a, b, c);
    printf("str = %s\n", str);
}
```
```c
#include <stdio.h>

FILE *myfile1;
FILE *myfile2;

int main()
{
    int i, nlines = 0;
    char str[255];

    myfile1 = fopen("textfile.txt", "r");
    myfile2 = fopen("copyfile.txt", "w");
    fgets(str, 255, myfile1);

    while (!feof(myfile1)) {
        fprintf(myfile2, "%s", str);
        nlines++;
        fgets (str, 255, myfile1);
    }

    printf("file has been copied!\n");
    printf("%d lines read\n", nlines);
}
```

Exercises with files

- Write a program that reads a file line by line and prints every line reversed
  - **Hint:** Write a function that reverts a string

- Write a function that reads a file and counts the number of words
  - **Hint:** two words are separated by spaces, commas ",", full stop ".", semicolon ";", colon ":" ,question mark "?", exclamation mark "!", dash "-", brackets. see http://en.wikipedia.org/wiki/Punctuation
  - this is called tokenize