Functions

- Function
- Input/output parameters
- Call by value / call by reference

- `return`
- `void`

Functions

- Function = callable group of statements that together perform a task
  - `output = function(input)`
    - Input parameter `input`
    - Output parameter (return value) `output`
- Why functions?
  - Decomposition of a large problem into manageable small problems
  - Structured programming (levels of abstraction)
  - Reuse of program code
- A function consists of signature and body
  - Signature = name & input/output parameters
    - Number & ordering are important!
  - Body = Implementation of the function

Name convention

- Local variables `lowercase_with_underscores`
- Global variables `underscore_also_at_the_end_`
- Functions `firstWordLowercaseNoUnderscores`

Functions in C

- In C functions are allowed to have
  - more than one or no parameters
  - one or zero return value
- Return value must be an elementary data type
  - e.g., `double`, `int`
- The signature has the following form
  `<type of return value> <function name>(parameters)`
  - Function without return value:
    - `<type of return value>` = `void`
  - Else: `<type of return value>` = data type
  - `parameters` = list of input parameters
    - separated by commas
    - specify data type *before* each parameter
    - *no* parameter ⇒ empty brackets `()`
- The body is a block
  - Go back to the main program either with `return` or, for functions without return value (`void`), at the end of the function block
  - Go back to the main program with `return output`, if the variable `output` should be returned
  - Common mistake: forget `return`
    - Then, return value is random
    - Chaos (= runtime error)

Variables

- All variables declared throughout a function block are local
- All elementary variables declared in the signature are local
- Functions receive input parameters as values, recall type casting!

Call by value

- Call by value = When the function is called, the values of the input parameters are copied into local variables used inside the function
  - New storage locations are allocated and the values of the input parameters are copied
  - Changes made inside the function have no effect on the input parameters

Example: squaring

```
#include <stdio.h>

double square(double x) {
  return x*x;
}

main() {
  double x = 0;
  printf("Input x = ");
  scanf("%lf",&x);
  printf("%f^2 = %f\n",x,square(x));
}
```

- Compiler must know the function before calling it
  - Define a function before its first call
- Execution always starts with `main()`
- The variable `x` in the function `square` and the variable `x` in the function `main` are different!
- Giving 5 as input value yields the output

```
Input x = 5
5^2 = 25.000000
```

Example: Minimum of two numbers

```c
#include <stdio.h>

double min(double x, double y) {
  if (x > y) {
    return y;
  }
  else {
    return x;
  }
}

main() {
  double x = 0;
  double y = 0;

  printf("Input x = ");
  scanf("%lf",&x);
  printf("Input y = ");
  scanf("%lf",&y);
  printf("min(x,y) = %f\n",min(x,y));
}
```

- Input of 10 and 2 yields the output

```
Input x = 10
Input y = 2
min(x,y) = 2.000000
```

- Typical structure of a program in exercises:
  - Function with a specific functionality
  - Main program which
    - reads the input parameters from the keyboard
    - calls the function
    - prints the output to the screen

Declaration of functions

```
1  #include <stdio.h>
2
3  double min(double, double);
4
5  main() {
6    double x = 0;
7    double y = 0;
8
9    printf("Input x = ");
10   scanf("%lf",&x);
11   printf("Input y = ");
12   scanf("%lf",&y);
13   printf("min(x,y) = %f\n",min(x,y));
14 }
15
16 double min(double x, double y) {
17   if (x > y) {
18     return y;
19   }
20   else {
21     return x;
22   }
23 }
```

- Too many functions might make the code heavy
  - Declare all functions at the beginning, see line 3
    - Compiler knows how the function operates
  - Full code of the function follows, see lines 16–23
- Alternative declaration = Function code without body
  - `double min(double x, double y);`
    see lines 3 and 16
- Keywords: *forward declaration* and *prototype*

Call by value

```c
#include <stdio.h>

void test(int x) {
  printf("a) x=%d\n", x);
  x = 43;
  printf("b) x=%d\n", x);
}


main() {
  int x = 12;
  printf("c) x=%d\n", x);
  test(x);
  printf("d) x=%d\n", x);
}
```

- Output:

  c) x=12

  a) x=12

  b) x=43

  d) x=12

Call by reference

- In other programming languages, it is not the value of a variable that is passed to a function as input parameter, but rather its memory address (call by reference)
  - In this way, changes in the variable can be also seen outside of the function

```
void test(int y) {
  printf("a) y=%d\n", y);
  y = 43;
  printf("b) y=%d\n", y);
}


main() {
  int x = 12;
  printf("c) x=%d\n", x);
  test(x);
  printf("d) x=%d\n", x);
}
```

- This source code is not a C code!
  - Just to explain the concept!
- *Call by reference* would yield the following output:

  c) x=12

  a) y=12

  b) y=43

  d) x=43

- Call by reference in C realized with pointers (more details later!)

Type casting & call by value

```
#include <stdio.h>

double divide(double, double);

main() {
  int int1 = 2;
  int int2 = 3;

  printf("a) %f\n", int1 / int2 );
  printf("b) %f\n", divide(int1,int2));
}

double divide(double dbl1, double dbl2) {
  return(dbl1 / dbl2);
}
```

- Type casting from int to double in the function
- Output:

  a) 0.000000

  b) 0.666667

Type casting (negative example!)

```
#include <stdio.h>

int isEqual(int, int);

main() {
  double x = 4.1;
  double y = 4.9;

  if (isEqual(x,y)) {
      printf("x == y\n");
  }
  else {
      printf("x != y\n");
  }
}

int isEqual(int x, int y) {
  if (x == y) {
    return 1;
  }
  else {
    return 0;
  }
}
```

- Output:

  ```
  x == y
  ```

- But actually $x \neq y$!
  - Implicit type casting from `double` to `int` via truncation, because input parameters are `int`
- Pay attention to type casting while dealing with functions

Recursion

- What is a recursive function?
- Example: Factorial
- Example: Bisection method

Recursive function

- A function is recursive, if it calls itself
- Natural concept in mathematics:
  - $n! = n \cdot (n-1)!$
- Philosophy: Reduce a problem to a 'smaller' (= easier) problem of the same type
- Be careful!
  - Recursion must end
  - Termination condition is important
  - e.g., $1! = 1$
- Often recursion can be replaced by loops (more details later!)
  - usually recursion more elegant
  - usually loops more efficient

Example: Factorial

```
#include <stdio.h>

int factorial(int n) {
  if (n <= -1) {
    return -1;
  }
  else {
    if (n > 1) {
      return n*factorial(n-1);
    }
    else {
      return 1;
    }
  }
}

main() {
  int n = 0;
  int nfac = 0;
  printf("n=");
  scanf("%d",&n);
  nfac = factorial(n);
  if (nfac <= 0) {
    printf("Wrong input!\n");
  }
  else {
    printf("%d!=%d\n",n,nfac);
  }
}
```

Bisection method

- Input
  - Continuous function $f : [a, b] \to \mathbb{R}$ satisfying $f(a)f(b) \leq 0$
  - Tolerance $\tau > 0$
- Intermediate value theorem
  - There exists $x \in [a, b]$ with $f(x) = 0$
- Task
  - Find approximation of a zero of f
  - i.e., find $x_0 \in [a, b]$ with the following property: $\exists\, x \in [a, b]$ such that $f(x) = 0$ and $|x - x_0| \leq \tau$
- Bisection method = Interval halving method
  - As long as $|b - a| > 2\,\tau$
    - Compute midpoint m of $[a, b]$ and $f(m)$
    - If $f(a)f(m) \leq 0$, consider $[a, m]$
    - Otherwise consider $[m, b]$
  - $x_0 := m$ is the desired approximation
- The method terminates after a finite number of steps
- Convergence towards $x \in [a, b]$ with $f(x) = 0$ as $\tau \to 0$

Example: Bisection method

```c
#include <stdio.h>

double f(double x) {
  return x*x + 1/(2 + x) - 2;
}

double bisection(double a, double b, double tol){
  double m = 0.5*(a+b);
  if ( b - a <= 2*tol ) {
    return m;
  }
  else {
    if ( f(a)*f(m) <= 0 ) {
      return bisection(a,m,tol);
    }
    else {
      return bisection(m,b,tol);
    }
  }
}

main() {
  double a = 0;
  double b = 10;
  double tol = 1e-12;
  double x = bisection(a,b,tol);

  printf("Approximate zero x=%g\n",x);
  printf("Function value f(x)=%g\n",f(x));
}
```

- Placeholder for `double` in `printf`
  - `%f` fixed-point number representation `1.30278`
  - `%e` exponential representation `-5.64659e-13`
  - `%g` most appropriate between `%f` and `%e`

Mathematical functions

- Compilation process
- Object code
- Libraries
- Mathematical functions

- `#define`
- `#include`

Compilation process

- Compilers convert a C program into an executable
- Compilation process consists of four steps
  - Preprocessing
  - Compiling
  - Assembling
  - Linking

Preprocessing

- Removes comments
- Expands macros and included files
- Preprocessor commands *always* start with `#` and *never* end with semicolon, e.g.,
  - `#define text replacement`
    - In all successive lines of code `text` is replaced by `replacement`
    - Useful to define constants
    - **Convention:** `UPPERCASE_WITH_UNDERSCORES`
  - `#include file`
    - Includes the file `file`

Compiling & Assembling

- Compiler translates preprocessed (source) code into assembly code
- Assembler translates assembly code into object code
- Object code = Machine code, where symbolic names (e.g., function names) are still available

Linking

- Linker includes further object code
  - e.g., libraries (= collection of functions)
- Symbolic names in object code are replaced by addresses
- Creation of executable program

If you are curious...

- Compile code with `gcc -save-temps filename.c`
  - `filename.i` = Output of preprocessor
  - `filename.s` = Output of compiler
  - `filename.o` = Output of assembler (object code)

Libraries & Header files

- Libraries (e.g., mathematical functions) always consists of 2 files
  - Object code
  - Associated header file
- Header file contains declaration of all functions available in the library
- If you want to use a library, you must include the corresponding header file in your source code
  - `#include <header>` includes the header file `header` from the standard folder `/usr/include/`
    - * e.g., `math.h` (header file for math. library)
  - `#include "file"` the header file `file` from the *current* folder (e.g., downloads from internet)
- Moreover, object code of the library must be linked
  - Its location must be told to `gcc` with the option -ℓ (and -L)
  - e.g., `gcc file.c -`ℓ`m` links the math. library
  - Standard libraries automatically linked e.g., no additional option needed for `stdio`

Mathematical functions

- Declaration of math. functions in `math.h`
  - Function input and output are of type `double`
- If you need a function of the math. library
  - In source code: `#include <math.h>`
  - Compile source code with linker option `-ℓm` to create the executable `output`, i.e., `gcc file.c -o output -ℓm`
- Among others, this library provides
  - Trigonometric functions
    - `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`
  - Exponential and logarithm
    - `exp`, `log`, `log10`
  - Power and root functions
    - `pow`, `sqrt` (where $x^y = \mathtt{pow(x,y)}$)
    - **NOT** x^3 via `pow`, **BUT** `x*x*x`
    - **NOT** $(-1)^n$ via `pow`, **BUT** ...
  - Absolute value `fabs`
  - Rounding to integers: `round`, `floor`, `ceil`
- **Be careful:** In the library `stdlib.h` there is `abs`
  - `abs` is absolute value for `int`
  - `fabs` is absolute value for `double`

Elementary example

```
1 #include <stdio.h>
2 #include <math.h>
3
4 main() {
5   double x = 2.;
6   double y = sqrt(x);
7   printf("sqrt(%f)=%f\n",x,y);
8 }
```

- Precompiler commands in lines 1–2
  (without semicolon)
- Compile with `gcc sqrt.c -ℓm`
- If you forget `-ℓm` ⇒ Error message from linker

  ```
  In function ‘main’
  sqrt.c:(.text+0x24):  undefined reference to ‘sqrt’
  collect2:  ld returned 1 exit status
  ```

- Output:

  ```
  sqrt(2.000000)=1.414214
  ```

Arrays

- Vectors & Matrices
- Operator [...]
- Matrix-vector multiplication
- Linear system of equations

Vectors

- Declaration of a vector $x = (x_0, \dots, x_{N-1}) \in \mathbb{R}^N$:
  - `double x[N];` declares a `double`-vector `x`
- Access to the coefficients
  - `x[j]` refers to x_j
  - Each `x[j]` is of type `double`
- Analogous declaration for other data types
  - `int y[N];` declares a `int`-vector `y`
- Watch out for the coefficient indexing!
  - In C the indices are $0, \dots, N-1$
  - The indices are not $1, \dots, N$
  - Indexing with $1, \dots, N$ is used, e.g., in
    * Mathematics
    * Other programming languages (e.g., Matlab)
    * NOT in C!
- Simultaneous initialization & declaration possible
  - `double x[3] = {1,2,3};` $\rightarrow x = (1, 2, 3) \in \mathbb{R}^3$
  - Vector initialization allowed only together with declaration
  - Otherwise must be done componentwise
    * i.e., `x[0] = 1; x[1] = 2; x[2] = 3;` is OK
    * `x = {1,2,3}` is not allowed

Example: Reading a vector

```c
#include <stdio.h>

main() {
  double x[3] = {0,0,0};

  printf("Input of a vector in R^3:\n");
  printf("x_0 = ");
  scanf("%lf",&x[0]);
  printf("x_1 = ");
  scanf("%lf",&x[1]);
  printf("x_2 = ");
  scanf("%lf",&x[2]);

  printf("x = (%f, %f, %f)\n",x[0],x[1],x[2]);
}
```

- Printing double via printf with placeholder %f
- Reading double via scanf with placeholder %ℓf

Static arrays

- Array lengths are static
  - Cannot be changed during program execution
    - e.g., $x \in \mathbb{R}^3$ cannot be changed to $x \in \mathbb{R}^5$
- Programs cannot determine array sizes
  - During the execution, a program does not know that a vector $x \in \mathbb{R}^3$ has length 3
  - Task of the programmer!
- Watch out for the coefficient indexing!
  - In C the indices are $0, \dots, N-1$
  - A program does not know if `x[j]` is defined
    - x must have at least length $j+1$
    - Wrong indexing is not a syntax error (= runtime error)
- Arrays cannot be the output of a function
- Arrays are passed to functions via call by reference
- The same holds for matrices or general arrays

Arrays & Call by reference

```c
#include <stdio.h>

void callByReference(double y[3]) {
  printf("a) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
  y[0] = 1;
  y[1] = 2;
  y[2] = 3;
  printf("b) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
}

main() {
  double x[3] = {0,0,0};

  printf("c) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
  callByReference(x);
  printf("d) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
}
```

- Output:

  ```
  c) x = (0.000000, 0.000000, 0.000000)
  a) y = (0.000000, 0.000000, 0.000000)
  b) y = (1.000000, 2.000000, 3.000000)
  d) x = (1.000000, 2.000000, 3.000000)
  ```

- Call by reference for vectors!
- Explanation follows later ($\rightarrow$ pointers)

Wrong indexing for vectors

```
 1 #include <stdio.h>
 2 #define WRONG 1000
 3
 4 main() {
 5   int x[3] = {0,1,2};
 6
 7   x[WRONG] = 43;
 8
 9   printf("x = (%d, %d, %d), x[%d] = %d\n",
10           x[0],x[1],x[2],WRONG,x[WRONG]);
11 }
```

- Line 2 defines the constant `WRONG`
  - **Convention:** Constants are `UPPERCASE_WITH_UNDERSCORES`
- Lines 7, 9–10: Wrong access to vector x
  - Nevertheless neither error message nor warning from the compiler
  - Correct indexing is a task of the programmer!
- Output:

```
x = (0, 1, 2), x[1000] = 43
```

- Runtime error
  - `WRONG` small ⇒ No error message
  - `WRONG` suff. large ⇒ Maybe segmentation fault
    - Attempt to access a forbidden memory location

Matrices

- Matrix $A \in \mathbb{R}^{M \times N}$ is a rectangular structure

$$A = \begin{pmatrix} A_{00} & A_{01} & A_{02} & \dots & A_{0,N-1} \\ A_{10} & A_{11} & A_{12} & \dots & A_{1,N-1} \\ A_{20} & A_{21} & A_{22} & \dots & A_{2,N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{M-1,0} & A_{M-1,1} & A_{M-1,2} & \dots & A_{M-1,N-1} \end{pmatrix}$$

  with coefficients $A_{jk} \in \mathbb{R}$

- Fundamental objects in linear algebra
- Declaration of a matrix $A \in \mathbb{R}^{M \times N}$:
  - `double A[M][N];` declares a `double`-matrix `A`
- Access to the coefficients
  - `A[j][k]` refers to A_{jk}
  - Each `A[j][k]` is of type `double`
- Row-wise initialization together with declaration
  - `double A[2][3] = {{1,2,3},{4,5,6}};`

    declares and initializes $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - Only possible for simultaneous declaration (the same as for vectors)

General arrays

- Vectors are 1-dimensional arrays
- Matrices are 2-dimensional arrays
- In general, given a data type `type`,
  - `type x[N];` declares a vector of length N, where each `x[j]` is a variable of type `type`
  - `type x[M][N];` declares a $M \times N$ matrix, where `x[j]` is a vector of type `type` (with length N), while each `x[j][k]` is a variable of type `type`
  - `type x[M][N][P];` declares a 3-dimensional array, where `x[j]` is a $N \times P$ matrix of type `type`, `x[j][k]` is a vektor of type `type` (with length P), while each `x[j][k][p]` is a variable of type `type`
  - . . .