# Introduction to Scientific Programming

## Part II: Programming in C

Prof. Dr. Dirk Praetorius

Julian Streitberger, MSc

---


Institute of Analysis
and Scientific Computing

# Software

- You need a C compiler!
- For Windows10, we recommend the Windows Subystem for Linux (WSL)
  - An installation guide (in German) is available in the TUWEL course
- For Mac OS, we recommend to install Xcode (including the command line tools)
  - An installation guide (in German) is available in the TUWEL course
- If you have Linux, you are already ready to start!
- If you need help with the installation, contact me as soon as possible

# Course material

- Slides (formally nothing else necessary)

# Optional literature

- Brian W. Kernighan, Dennis M. Ritchie
  *The C programming language*
- Bradley L. Jones, Peter Aitken, Dean Miller
  *C programming in one hour a day*
- Klaus Schmaranz
  *Softwareentwicklung in C* (in German!)

- Bjarne Stroustrup
  *Programming – Principles and practice using C++*
  *The C++ programming language*
- Siddhartha Rao
  *C++ in one hour a day*
- Klaus Schmaranz
  *Softwareentwicklung in C++* (in German!)

# The first program in C

- Program & Algorithm
- Source code & Executable
- Compiler & Interpreter
- Syntax error & Runtime error
- How to write a program in C?

- `main`
- `printf` (print text to the screen)
- `#include <stdio.h>`

# Nota bene!

- Recall that MATLAB is an interpreted language
  - The interpreter executes source code line-by-line during the "translation"
    - i.e., translate and execute at the same time
- C is a compiled programming language
  - The compiler "translates" the source code and produces a stand-alone program written in assembly language (executable)
    - i.e., first translate, then execute
- Compiled code is *system-dependent*,
  - In principle, the code can run only on the system on which it has been compiled
- Source code is *system-independent*,
  - The code can be compiled also on other systems
- C compilers are not all equal
  - Before any exercise session, compile and test any program with the C compiler `gcc` on `lva.student.tuwien.ac.at`
  - Non-compiling code = bad impression and possibly also worse grade. . .

# How to write a program in C?

- Start your favorite text editor
  - e.g., `nano`, `emacs`, `vim`, `gedit`, `atom`, ...
    - at home, you can use any IDE,
      - e.g., Xcode under Mac OS)
  - At some point you should be familiar with `nano` for your later class on parallel programming on the Vienna Scientific Cluster
- Open a (new) file `name.c`
  - The filename extension `.c` is typical for programs in C
- Write the *source code* (= program)
- Don't forget to save the file
- Compile the code, e.g., open a shell and type `gcc name.c`
- If there are no errors, one gets the *executable* `a.out` (`a.exe` under Windows)
- This can be executed with `a.out` or `./a.out`
- Compile with `gcc name.c -o output` creates the executable `output` instead of `a.out`

# The first program in C

```
1 #include <stdio.h>
2
3 main() {
4   printf("Hello World!\n");
5 }
```

- Line numbers do *not* belong to the code (included to ease the following description)
- Every program in C has line 3 and line 5
- The execution of a program in C starts *always* from `main()`, independently of where `main()` is located in the code
- In C, curly brackets {...} contain so-called *blocks*
- The main program `main()` always constitutes a block
- Statements end with a semicolon; see line 4
- `printf` prints text to the screen (in quotes),
  - `\n` determines a new line
- Quotes *must* be in the same line
- Line 1: Inclusion of the C standard library for input-output (more info later)

# main() vs. int main()

```
1 #include <stdio.h>
2
3 main() {
4   printf("Hello World!\n");
5 }
```

- The C programming language has evolved over the years
- `main() {` in line 3 is C89 standard
- C99 and C++ require `int main() {`

```
1 #include <stdio.h>
2
3 int main() {
4   printf("Hello World!\n");
5   return 0;
6 }
```

- Meaning:
  - main() communicates with the operative system
  - main() returns an error code via return
  - Return value zero = no error occurred
- In this case return 0; meaningful
  - More details later; see functions
- Consequence:
  - If the compiler does not accept the previous code (or gives annoying warnings), use this one!

# Syntax error

- Syntax = Dictionary & Grammar of a language
- Syntax error = Wrong expression or wrong use
  - Detected by the compiler, which returns an error code

```
1 main() {
2   printf("Hello World!\n");
3 }
```

- Warning, inclusion of `stdio.h` missing

```
wrongworld1.c:2: warning: incompatible implicit
declaration of built-in function printf
```

- C++ compiler gives an error due to missing `int`

```
wrongworld1.c:1: error: C++ requires a type
specifier for all declarations
```

```
1 #include <stdio.h>
2
3 main() {
4   printf("Hello World!\n")
5 }
```

- Error code, semicolon at the end of line 4 missing

```
wrongworld2.c:5: error: syntax error before } token
```

# Runtime error

- Error occurring when the program is executed
  - Usually more difficult to detect
  - Should be avoided with careful programming

# Variables

- What is a variable?
- Declaration & Initialization
- Data types `int` and `double`
- Assignment operator `=`
- Arithmetic operations `+ - * / %`
- Type casting

- `int`, `double`
- `printf` (print value of a variable to the screen)
- `scanf` (read value of a variable from the keyboard)

# Variable

- Variable = symbolic name (identifier) of a storage location (memory address) containing some quantity of information (value)

# Variable names (identifiers)

- Made of letters, digits and underscore _
  - Maximum length = 31
  - The first character cannot be a digit
- Variable names are case-sensitive
  - i.e. `Var`, `var`, `VAR` are three different variables
- **Convention:** `lowercase_with_underscores`

# Data types

- The data type of a variable must be declared before using it
- Elementary data types:
  - Floating-point numbers for values in $\mathbb{Q}$, $\mathbb{R}$, e.g., `double`
  - Integer for values in $\mathbb{N}$, $\mathbb{Z}$, e.g., `int`
  - Characters (letters), e.g., `char`

## Declaration

- Declaration = Creation of a variable
  - Assignment of a symbolic name to a storage location and specification of the data type
  - `int x;` declares a variable `x` of type `int`
  - `double var;` declares `var` of type `double`

## Initialization

- Declaring a variable only assigns a storage location to it
- If no value is explicitly assigned, the value will be random
- Right after the declaration, a new value should be assigned, i.e., initialization
  - `int x;` (declaration)
  - `x = 0;` (initialization)
- Declaration & initialization simultaneously
  - `int x = 0;`

# A first example with `int`

```
1 #include <stdio.h>
2
3 main() {
4   int x = 0;
5
6   printf("Input: x=");
7   scanf("%d",&x);
8   printf("Output: x=%d\n",x);
9 }
```

- Inclusion of input-output functions (line 1)
  - `printf` prints text (e.g., the value of a variable) to the screen
  - `scanf` reads the value of a variable from the keyboard
- Percent sign `%` in lines 7–8 introduces a placeholder

| data type | placeholder `printf` | placeholder `scanf` |
|---|---|---|
| `int` | `%d` | `%d` |
| `double` | `%f` | `%lf` |

- Note the symbol `&` for `scanf` in line 7
  - `scanf("%d",&x)`
  - But: `printf("%d",x)`
- Forgetting `&` introduces a runtime error
  - The compiler does not report the error (no syntax error!)

# The same example with `double`

```
1 #include <stdio.h>
2
3 main() {
4   double x = 0;
5
6   printf("Input: x=");
7   scanf("%lf",&x);
8   printf("Output: x=%f\n",x);
9 }
```

- Note the placeholders in lines 7–8
  - `scanf("%lf",&x)`
  - but: `printf("%f",x)`
- Use of `%f` in line 7 $\Rightarrow$ Wrong reading!
  - Probably a runtime error!

# Assignment operator

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x = 1;
 5   int y = 2;
 6
 7   int tmp = 0;
 8
 9   printf("a) x=%d, y=%d, tmp=%d\n",x,y,tmp);
10
11   tmp = x;
12   x = y;
13   y = tmp;
14
15   printf("b) x=%d, y=%d, tmp=%d\n",x,y,tmp);
16 }
```

- The symbol `=` is the assignment operator
  - Assignment always from the right (value) to the left (variable)
- `x = 1;` assigns the value `1` on the right-hand side to the variable `x` on the left-hand side
- `x = y;` assigns the value of the variable `y` to the variable `x`
  - In particular, x and y have the same value afterwards
  - Swapping the value of two variables usually requires an auxiliary variable
- Output:

  ```
  a) x=1, y=2, tmp=0
  b) x=2, y=1, tmp=1
  ```

# Arithmetic operators

- The action of an operator can depend on the data type!
- Operators for integers:
  - `a=b`, `-a` (sign)
  - `a+b`, `a-b`, `a*b`, `a/b` (division without remainder)
    `a%b` (modulus operator)
- Operators for floating point numbers:
  - `a=b`, `-a` (sign)
  - `a+b`, `a-b`, `a*b`, `a/b` ("standard" division)
- Attention: `2/3` is zero (division without remainder)
- Building blocks of floating point numbers:
  - Minus sign - (if negative)
  - Predecimal positions
  - Decimal separator (point)
  - Decimal positions
  - `e` or `E` with *integer* exponent ($n$-th power of 10)
    e.g., 2e2 = 2E2 = $2 \cdot 10^2 = 200$
- Hence: `2./3.` is floating point division $\approx 0.\overline{6}$

# Type casting

- Operators can also work with variables/values of different type
- Before execution, the values are converted to the same data type (type casting)

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x = 1;
 5   double y = 2.5;
 6
 7   int sum_int = x+y;
 8   double sum_dbl = x+y;
 9
10   printf("sum_int = %d\n",sum_int);
11   printf("sum_dbl = %f\n",sum_dbl);
12 }
```

- Which data type does `x+y` have in lines 7–8?
  - The "stronger" data type, i.e., `double`
  - Type casting of the value of `x` to `double`
- Line 7: Type casting from `double` to `int`
  - Truncation, no rounding!
- Output:

```
sum_int = 3
sum_dbl = 3.500000
```

# Implicit type casting

```
 1 #include <stdio.h>
 2
 3 main() {
 4   double dbl1 = 2 / 3;
 5   double dbl2 = 2 / 3.;
 6   double dbl3 = 1E2;
 7   int int1 = 2;
 8   int int2 = 3;
 9
10   printf("a) %f\n",dbl1);
11   printf("b) %f\n",dbl2);
12
13   printf("c) %f\n",dbl3 * int1 / int2);
14   printf("d) %f\n",dbl3 * (int1 / int2) );
15 }
```

- Output:

  a) 0.000000

  b) 0.666667

  c) 66.666667

  d) 0.000000

- Why the result 0 in a) and d) ?
  - 2, 3 are `int` ⇒ `2/3` is division without remainder
- If an arithmetic operator is applied to variables of different type, they are cast to the "stronger" type
  - See lines 5, 13, and 14
  - `2` is `int`, `3.` is `double` ⇒ `2/3.` is `double`

# Explicit type casting

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int a = 2;
 5   int b = 3;
 6   double dbl1 = a / b;
 7   double dbl2 = (double) (a / b);
 8   double dbl3 = (double) a / b;
 9   double dbl4 = a / (double) b;
10
11   printf("a) %f\n",dbl1);
12   printf("b) %f\n",dbl2);
13   printf("c) %f\n",dbl3);
14   printf("d) %f\n",dbl4);
15 }
```

- It is possible to tell the compiler how to interpret a variable
  - Precede the operation with the desired data type (in brackets)
- Output:

  a) 0.000000

  b) 0.000000

  c) 0.666667

  d) 0.666667
- In lines 7–9: explicit type casting (all from `int` to `double`)
- In lines 8–9: implicit type casting

# Error sources in type casting

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int a = 2;
 5   int b = 3;
 6   double dbl = (double) a / b;
 7
 8   int i = dbl;
 9
10   printf("a) %f\n",dbl);
11   printf("b) %f\n",dbl*b);
12   printf("c) %d\n",i);
13   printf("d) %d\n",i*b);
14 }
```

- Output:

  a) 0.666667

  b) 2.000000

  c) 0

  d) 0

- Implicit type casting should be avoided!
  - i.e., use explicit type casting
- Save intermediate results of computations in the right data type!

# Conditionals

- Comparison / relational operators == != > >= < <=
- Logical operators ! && ||
- True/false for statements
- Conditional statements

- `if`
- `if - else`

# Logical operators

- Let `a, b` be two variables (possibly of different type)
  - Comparison (e.g., `a < b`) returns `1` if true, or returns `0` if false
- Overview of comparison operators:

| | |
|---|---|
| `==` | equality (different from assignment `=`) |
| `!=` | inequality |
| `>` | strictly larger |
| `>=` | larger or equal |
| `<` | strictly smaller |
| `<=` | smaller or equal |

- Advice: Put comparisons in brackets!
  - Not always necessary, but sometimes helpful!
- Further logical operators:

| | |
|---|---|
| `!` | not |
| `&&` | and |
| `\|\|` | or |

# Logical concatenation

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int result = 0;
 5
 6   int a = 3;
 7   int b = 2;
 8   int c = 1;
 9
10   result = (a > b > c);
11   printf("a) result=%d\n",result);
12
13   result = (a > b) && (b > c);
14   printf("b) result=%d\n",result);
15 }
```

- Output:

  a) result=0

  b) result=1

- Why does line 10 return false and line 13 true?
  - Evaluation from the left to the right:
    - `a > b` is true, returns value `1`
    - `1 > c` is false, returns value `0`
    - Altogether, `a > b > c` returns `0` (false)!
  - Statement in line 10 is not properly formulated!

# if-else

- Simple conditional statement: *if - then - else*
- `if (condition) statementA else statementB`

- After `if` there is the condition, *always* in brackets
- After the condition: *no* semicolon
- The condition is *false*, if it is 0 or if its evaluation is 0, otherwise it is *true*
  - Condition true ⇒ `statementA` is executed
  - Condition false ⇒ `statementB` is executed
- The statement consists of
  - either one line
  - or more lines in curly brackets `{ ... }` (block)
- The `else`-part is optional
  - i.e., `else statementB` can be omitted

# Example for `if`

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x = 0;
 5
 6   printf("Input x=");
 7   scanf("%d",&x);
 8
 9   if (x < 0)
10     printf("x=%d is negative\n",x);
11
12   if (x > 0) {
13     printf("x=%d is positive\n",x);
14   }
15 }
```

- Use proper indentation (it facilitates readability!)
- Attention: non-use of blocks `{...}` is sometimes source of mistakes
- One could continue with `else` in line 11

# Example for `if-else`

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int var1 = -5;
 5   double var2 = 1e-32;
 6   int var3 = 5;
 7
 8   if (var1 >= 0) {
 9     printf("var1 >= 0\n");
10   }
11   else {
12     printf("var1 < 0\n");
13   }
14
15   if (var2) {
16     printf("var2 != 0, i.e., condition is true\n");
17   }
18   else {
19     printf("var2 == 0, i.e., condition is false\n");
20   }
21
22   if ( (var1 < var2) && (var2 < var3) ) {
23     printf("var2 lies between the others\n");
24   }
25 }
```

- A condition is true if the value $\neq 0$
  - e.g., line 15, more explicit: `if (var2 != 0)`
- Output:

```
var1 < 0
var2 != 0, i.e., cond.  is true
var2 lies between the others
```

# Even or odd?

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x = 0;
 5
 6   printf("Input x=");
 7   scanf("%d",&x);
 8
 9   if (x > 0) {
10     if (x%2 != 0) {
11       printf("x=%d is odd\n",x);
12     }
13     else {
14       printf("x=%d is even\n",x);
15     }
16   }
17   else {
18     printf("Error: Input has to be positive!\n");
19   }
20 }
```

- The program checks if a given number x is odd or even
- Conditional statements can be nested:
  - Indentation makes the code more clear
    - Formally not needed, but fundamental!
  - Dependencies are emphasized

# Sorting two numbers in ascending order

```
 1 #include <stdio.h>
 2
 3 main() {
 4   double x1 = 0;
 5   double x2 = 0;
 6   double tmp = 0;
 7
 8   printf("Unsorted input:\n");
 9   printf(" x1=");
10   scanf("%lf",&x1);
11   printf(" x2=");
12   scanf("%lf",&x2);
13
14   if (x1 > x2) {
15     tmp = x1;
16     x1 = x2;
17     x2 = tmp;
18   }
19
20   printf("Output sorted in ascending order:\n");
21   printf(" x1=%f\n",x1);
22   printf(" x2=%f\n",x2);
23 }
```

- Input of two numbers $x_1, x_2 \in \mathbb{R}$ (possibly unsorted)
- Numbers are sorted in ascending order
  - i.e., they are swapped if needed
- Sorted numbers are printed to the screen

# Inside or outside?

```
 1 #include <stdio.h>
 2
 3 main() {
 4   double r = 0;
 5   double x1 = 0;
 6   double x2 = 0;
 7   double z1 = 0;
 8   double z2 = 0;
 9   double dist2 = 0;
10
11   printf("Radius of the circle r=");
12   scanf("%lf",&r);
13   printf("Center of the circle x = (x1,x2)\n");
14   printf(" x1=");
15   scanf("%lf",&x1);
16   printf(" x2=");
17   scanf("%lf",&x2);
18   printf("Point in the plane z = (z1,z2)\n");
19   printf(" z1=");
20   scanf("%lf",&z1);
21   printf(" z2=");
22   scanf("%lf",&z2);
23
24   dist2 = (x1-z1)*(x1-z1) + (x2-z2)*(x2-z2);
25   if ( dist2 < r*r ) {
26       printf("z lies inside the circle\n");
27   }
28   else {
29     if ( dist2 > r*r ) {
30       printf("z lies outside of the circle\n");
31     }
32     else {
33       printf("z lies on the boundary of the circle\n");
34     }
35   }
36 }
```

# Equality vs. Assignment

- Recall: `if (a==b)` vs. `if (a=b)`
  - Both are syntactically correct!
  - `if (a==b)` checks the validity of the equality
    - This is usually what one desires
  - But: `if (a=b)`
    - The value of `b` is assigned to `a`
    - Condition is true if the value of `b` is $\neq 0$
    - It is bad programming style!
    - Some compilers give a warning

# Blocks

- Blocks `{...}`
- Declaration of variables
- Lifetime & scope
- Local & global variables

# Lifetime & scope

- Lifetime of a variable
  = period in which a storage location is assigned to the variable
  = period in which the variable exists
- Scope of a variable
  = period in which a variable is accessible
  = period in which the value of a variable can be read/changed
- Relation: scope $\subseteq$ lifetime

# Global & local variables

- Global variables = variables with global lifetime
  - Exist until the end of the program
  - Might only have local scope
  - Are declared outside of `main`
- Local variables = variables with local lifetime
- **Convention:** Identify variables from names
  - Local variables: `lowercase_with_underscores`
  - Global variables: `underscore_also_at_the_end_`

# Blocks

- Blocks are delimited by curly brackets { ... }
- Each block starts with the declaration of the additional variables needed
  - Variables *can* be declared only at the beginning of a block
  - Later in C++, variables can be declared everywhere in a block, but this not recommend!
- The variables declared inside a block are forgotten after the end of the block (= deleted)
  - i.e., end of their lifetime
  - They are local variables
- Nesting { ... { ... } ... }
  - Nesting is possible
  - Variables from an external block can be read or changed inside an internal block, but *not* the other way around
  - Changes remain valid, i.e., lifetime & scope are inherited only from the outside to the inside
  - If a variable `var` is declared both in an internal and in an external block, the "external" `var` is hidden in the internal block and becomes accessible again (with the same value as before) at the end of the internal block
    - i.e., the "external" `var` is not in internal scope
    - This is bad programming style!

# Easy example

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x = 7;
 5   printf("a) %d\n", x);
 6   x = 9;
 7   printf("b) %d\n", x);
 8   {
 9     int x = 17;
10     printf("c) %d\n", x);
11   }
12   printf("d) %d\n", x);
13 }
```

- Two different *local* variables x
  - Declaration + Initialization (lines 4 and 9)
  - Assignment (Line 6)

- Output:

  a) 7

  b) 9

  c) 17

  d) 9

# More complicated example

```
 1 #include <stdio.h>
 2
 3 int var0 = 5;
 4
 5 main() {
 6   int var1 = 7;
 7   int var2 = 9;
 8
 9   printf("a) %d, %d, %d\n", var0, var1, var2);
10   {
11     int var1 = 17;
12
13     printf("b) %d, %d, %d\n", var0, var1, var2);
14     var0 = 15;
15     var2 = 19;
16     printf("c) %d, %d, %d\n", var0, var1, var2);
17     {
18       int var0 = 25;
19       printf("d) %d, %d, %d\n", var0, var1, var2);
20     }
21   }
22   printf("e) %d, %d, %d\n", var0, var1, var2);
23 }
```

- Output:

  ```
  a) 5, 7, 9
  b) 5, 17, 9
  c) 15, 17, 19
  d) 25, 17, 19
  e) 15, 7, 19
  ```

- Two variables with name `var0` (line 3 and 18)
  - Name convention ignored on purpose
- Two variables with name `var1` (line 6 and 11)

# 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
  - zero or more input parameters
  - zero or one return values
- 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 calling code 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 for functions with return value: One forgets to write `return value;`
    - Then, return value is random
    - Chaos (= runtime error)

## Variables

- All variables declared throughout a function block are local
- All input parameters in the signature declare local variables
- 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

```
 1 #include <stdio.h>
 2
 3 double square(double x) {
 4   return x*x;
 5 }
 6
 7 main() {
 8   double x = 0;
 9   printf("Input x = ");
10   scanf("%lf",&x);
11   printf("%f^2 = %f\n",x,square(x));
12 }
```

- 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
5.000000^2 = 25.000000
```

# Ex: Minimum of two numbers

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

- 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

```
 1 #include <stdio.h>
 2
 3 void test(int x) {
 4   printf("a) x=%d\n", x);
 5   x = 43;
 6   printf("b) x=%d\n", x);
 7 }
 8
 9
10 main() {
11   int x = 12;
12   printf("c) x=%d\n", x);
13   test(x);
14   printf("d) x=%d\n", x);
15 }
```

- 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

```
 1 void test(int y) {
 2   printf("a) y=%d\n", y);
 3   y = 43;
 4   printf("b) y=%d\n", y);
 5 }
 6
 7
 8 main() {
 9   int x = 12;
10   printf("c) x=%d\n", x);
11   test(x);
12   printf("d) x=%d\n", x);
13 }
```

- *Call by reference* would yield the following output:

  c) x=12

  a) y=12

  b) y=43

  d) x=43

- This is not what C does; see last slide!
  - Later, C++ will also provide *call by reference* (though with a different notation)
- Call by reference in C is realized with pointers (more details will follow later!)

# Type casting & call by value

```
 1 #include <stdio.h>
 2
 3 double divide(double, double);
 4
 5 main() {
 6   int int1 = 2;
 7   int int2 = 3;
 8
 9   printf("a) %f\n", int1 / int2 );
10   printf("b) %f\n", divide(int1,int2));
11 }
12
13 double divide(double dbl1, double dbl2) {
14   return(dbl1 / dbl2);
15 }
```

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

  a) 0.000000

  b) 0.666667

# Type casting (negative example!)

```
 1 #include <stdio.h>
 2
 3 int isEqual(int, int);
 4
 5 main() {
 6   double x = 4.1;
 7   double y = 4.9;
 8
 9   if (isEqual(x,y)) {
10       printf("x == y\n");
11   }
12   else {
13       printf("x != y\n");
14   }
15 }
16
17 int isEqual(int x, int y) {
18   if (x == y) {
19     return 1;
20   }
21   else {
22     return 0;
23   }
24 }
```

- 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

```
 1 #include <stdio.h>
 2
 3 int factorial(int n) {
 4   if (n <= -1) {
 5     return -1;
 6   }
 7   else {
 8     if (n > 1) {
 9       return n*factorial(n-1);
10     }
11     else {
12       return 1;
13     }
14   }
15 }
16
17 main() {
18   int n = 0;
19   int nfac = 0;
20   printf("n=");
21   scanf("%d",&n);
22   nfac = factorial(n);
23   if (nfac <= 0) {
24     printf("Wrong input!\n");
25   }
26   else {
27     printf("%d!=%d\n",n,nfac);
28   }
29 }
```

# 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

```
 1 #include <stdio.h>
 2
 3 double f(double x) {
 4   return x*x + 1/(2 + x) - 2;
 5 }
 6
 7 double bisection(double a, double b, double tol){
 8   double m = 0.5*(a+b);
 9   if ( b - a <= 2*tol ) {
10     return m;
11   }
12   else {
13     if ( f(a)*f(m) <= 0 ) {
14       return bisection(a,m,tol);
15     }
16     else {
17       return bisection(m,b,tol);
18     }
19   }
20 }
21
22 main() {
23   double a = 0;
24   double b = 10;
25   double tol = 1e-12;
26   double x = bisection(a,b,tol);
27
28   printf("Approximate zero x=%g\n",x);
29   printf("Function value f(x)=%g\n",f(x));
30 }
```

- 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 that is machine dependent
- 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 consist 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 -ℓm` links the math library
  - Standard libraries automatically linked e.g., no additional option needed for `stdio`

# Mathematical functions

- Declaration of mathematical 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 $x^y$ is given by `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 $x = (x_0, \ldots, x_{N-1}) \in \mathbb{R}^N$:
  - `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 $0, \ldots, N-1$
  - The indices are not $1, \ldots, N$
  - Indexing with $1, \ldots, N$ is used, e.g., in
    - Mathematics
    - Other programming languages (e.g., Matlab)
    - NOT in C!
- Simultaneuous 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

```
 1 #include <stdio.h>
 2
 3 main() {
 4   double x[3] = {0,0,0};
 5
 6   printf("Input of a vector in R^3:\n");
 7   printf("x_0 = ");
 8   scanf("%lf",&x[0]);
 9   printf("x_1 = ");
10   scanf("%lf",&x[1]);
11   printf("x_2 = ");
12   scanf("%lf",&x[2]);
13
14   printf("x = (%f, %f, %f)\n",x[0],x[1],x[2]);
15 }
```

- 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 $0, \ldots, N-1$
  - 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

```
 1 #include <stdio.h>
 2
 3 void callByReference(double y[3]) {
 4   printf("a) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
 5   y[0] = 1;
 6   y[1] = 2;
 7   y[2] = 3;
 8   printf("b) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
 9 }
10
11 main() {
12   double x[3] = {0,0,0};
13
14   printf("c) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
15   callByReference(x);
16   printf("d) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
17 }
```

- 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 an $M \times N$ 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 an $N \times P$ matrix of type `type`, `x[j][k]` is a vector of type `type` (with length $P$), while each `x[j][k][p]` is a variable of type `type`
  - ...

# The `for` loop

- Mathematical symbols $\sum_{j=1}^{n}$ and $\prod_{j=1}^{n}$
- Count-controlled loops

- `for`

# Loops

- Loops allow for repeating the execution of one command (or more)
- You might need a loop, while dealing with
  - Vectors & Matrices
  - Counters $j = 1, \ldots, n$
  - Sums $\sum_{j=1}^{n} a_j := a_1 + a_2 + \cdots + a_n$
  - Products $\prod_{j=1}^{n} a_j := a_1 \cdot a_2 \cdots a_n$
  - Expressions like, e.g., *as long as* or *until*
- Classification
  - Count-controlled loops (`for`): Repeat an action for a specific number of times
  - Condition-controlled loops: Repeat an action
    - while some condition is satisfied
    - or until some condition is violated

# The `for` loop

- `for (init. ; cond. ; step-expr.) statement`
- Working principles of a `for` loop
  - (1) Execution of `init.` (initialization)
  - (2) Break, if the condition `cond.` is not satisfied
  - (3) Execution of `statement`
  - (4) Execution of `step-expr.`
  - (5) Go back to (2)
- `statement` is
  - a single line of code
  - more lines of code in curly brackets `{...}`, i.e., a block

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5
 6   for (j=5; j>0 ; j=j-1)
 7     printf("%d ",j);
 8
 9   printf("\n");
10 }
```

- `j=j-1` in line 6 is an assignment, no equality!
- Output:

  5 4 3 2 1

# Read and print a vector

```
 1 #include <stdio.h>
 2
 3 void scanVector(double input[], int dim) {
 4   int j = 0;
 5   for (j=0; j<dim; j=j+1) {
 6     input[j] = 0;
 7     printf("%d: ",j);
 8     scanf("%lf",&input[j]);
 9   }
10 }
11
12 void printVector(double output[], int dim) {
13   int j = 0;
14   for (j=0; j<dim; j=j+1) {
15     printf("%f ",output[j]);
16   }
17   printf("\n");
18 }
19
20 main() {
21   double x[5];
22   scanVector(x,5);
23   printVector(x,5);
24 }
```

- Functions must know array length
  - Passed as input parameter
- Arrays are passed via call by reference

# Conventions (recall)

- Local variables are `lowercase_with_underscores`
- Global variables have `underscore_afterwards_`
- Constants are `UPPERCASE_WITH_UNDERSCORES`
- Functions are `firstWordLowercaseNoUnderscores`

# Minimum of a vector

```
 1 #include <stdio.h>
 2 #define DIM 5
 3
 4 void scanVector(double input[], int dim) {
 5   int j = 0;
 6   for (j=0; j<dim; j=j+1) {
 7     input[j] = 0;
 8     printf("%d: ",j);
 9     scanf("%lf",&input[j]);
10   }
11 }
12
13 double min(double input[], int dim) {
14   int j = 0;
15   double minval = input[0];
16   for (j=1; j<dim; j=j+1) {
17     if (input[j]<minval) {
18       minval = input[j];
19     }
20   }
21   return minval;
22 }
23
24 main() {
25   double x[DIM];
26   scanVector(x,DIM);
27   printf("Minimum of the vector: %f\n", min(x,DIM));
28 }
```

- Note the structure (follow it for your exercises)
  - The length of the vector is a constant in `main`
    - i.e., the length cannot be changed
  - It is an input parameter in `scanVector`
    - i.e., the function works for any length

# Example: Sum symbol $\sum$

- Computation of the sum $S = \sum_{j=1}^{N} a_j$
  - Shorthand notation $\sum_{j=1}^{N} a_j := a_1 + a_2 + \cdots + a_N$
- Auxiliary partial sum $S_k = \sum_{j=1}^{k} a_k$
- Then, it holds that
  - $S_1 = a_1$
  - $S_2 = S_1 + a_2$
  - $S_3 = S_2 + a_3$ etc.
- Can be realized with $N$ sums
  - Be careful: Assignment, no equality
    - $\mathtt{S} = \mathtt{a}_1$
    - $\mathtt{S} = \mathtt{S} + \mathtt{a}_2$
    - $\mathtt{S} = \mathtt{S} + \mathtt{a}_3$
    - etc.

# Example: Sum symbol $\sum$

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5   int n = 100;
 6
 7   int sum = 0;
 8
 9   for (j=1; j<=n; j=j+1) {
10     sum = sum+j;
11   }
12
13   printf("sum_{j=1}^{%d} j = %d\n",n,sum);
14 }
```

- The program computes the sum $\sum_{j=1}^{n} j$ for $n = 100$
- Output:

  ```
  sum_{j=1}^{100} j = 5050
  ```

- Be careful: Do not forget to initialize (with zero) the variable for the result; see line 7
  - Otherwise: Wrong/random result!
- `sum += j;` is a shorthand notation for `sum = sum + j;`

# Example: Product symbol $\prod$

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5   int n = 5;
 6
 7   int factorial = 1;
 8
 9   for (j=1; j<=n; j=j+1) {
10     factorial = factorial*j;
11   }
12
13   printf("%d! = %d\n",n,factorial);
14 }
```

- The program computes the factorial $n! = \prod_{j=1}^{n} j$ for $n = 5$
- Output:

  ```
  5!  = 120
  ```

- Be careful: Do not forget to initialize (with one) the variable for the result; see line 7
  - Otherwise: Wrong/random result!
- `factorial *= j;` is a shorthand notation for `factorial = factorial*j;`

# Matrix-vector multiplication

- `for` loops can also be nested
  - e.g., matrix-vector multiplication
- Let $A \in \mathbb{R}^{M \times N}$ matrix, $x \in \mathbb{R}^N$ vector
- Define $b := Ax \in \mathbb{R}^M$ as $b_j = \sum_{k=0}^{N-1} A_{jk} x_k$
  - Recall: Indexing in C starts with 0
- $Ax = b$ is associated with the linear system

$$\begin{array}{ccccccc} A_{00}x_0 & + & A_{01}x_1 & + \ldots + & A_{0,N-1}x_{N-1} & = & b_0 \\ A_{10}x_0 & + & A_{11}x_1 & + \ldots + & A_{1,N-1}x_{N-1} & = & b_1 \\ A_{20}x_0 & + & A_{21}x_1 & + \ldots + & A_{2,N-1}x_{N-1} & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots \\ A_{M-1,0}x_0 & + & A_{M-1,1}x_1 & + \ldots + & A_{M-1,N-1}x_{N-1} & = & b_{M-1} \end{array}$$

- Implementation
  - external loop runs over $j$
  - internal loop to compute the sum

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- Be careful: Initialization `b[j] = 0`!

# Matrices in column-major order

- Many mathematical libraries store matrices columnwise as vectors (column-major order)
  - $A \in \mathbb{R}^{M \times N}$ is stored as $a \in \mathbb{R}^{MN}$
  - $a = (A_{00}, A_{10}, ..., A_{M-1,0},\ A_{01}, A_{11}, \ldots,\ A_{M-1,N-1})$
  - $A_{jk}$ corresponds to $a_\ell$ with $\ell = j + k \cdot M$
- Column-major order must be used if one wants to use those libraries
  - Usually the case with libraries written in Fortran
- Matrix-vector product
  - $b := Ax \in \mathbb{R}^M$, $b_j = \sum_{k=0}^{N-1} A_{jk} x_k$
  - with `double A[M][N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- Matrix-vector product (column-major order)
  - with `double A[M*N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j+k*M]*x[k];
  }
}
```

# Selection sort

- Input: Vector $x \in \mathbb{R}^n$
- Aim: Sorting $x$ so that $x_1 \leq x_2 \leq \cdots \leq x_n$

- Algorithm (Step 1)
  - Seek the minimum $x_k$ of $x_1, \ldots, x_n$
  - Swap $x_1$ and $x_k$, i.e., $x_1$ is smallest entry
- Algorithm (Step 2)
  - Seek the minimum $x_k$ of $x_2, \ldots, x_n$
  - Swap $x_2$ and $x_k$, i.e., $x_2$ is second smallest entry
- After $n-1$ steps, $x$ is sorted

- Note the structure (follow it for your exercises)
  - The length of the vector is a constant in `main`
    - i.e., the length cannot be changed
  - It is an input parameter in `selectionSort`
    - i.e., the function works for any length

```
 1 #include <stdio.h>
 2 #define DIM 5
 3
 4 void scanVector(double input[], int dim) {
 5   int j = 0;
 6   for (j=0; j<dim; j=j+1) {
 7     input[j] = 0;
 8     printf("%d: ",j);
 9     scanf("%lf",&input[j]);
10   }
11 }
12
13 void printVector(double output[], int dim) {
14   int j = 0;
15   for (j=0; j<dim; j=j+1) {
16     printf("%f ",output[j]);
17   }
18   printf("\n");
19 }
20
21 void selectionSort(double vector[], int dim) {
22   int j, k, argmin;
23   double tmp;
24   for (j=0; j<dim-1; j=j+1) {
25     argmin = j;
26     for (k=j+1; k<dim; k=k+1) {
27       if (vector[argmin] > vector[k]) {
28         argmin = k;
29       }
30     }
31     if (argmin > j) {
32       tmp = vector[argmin];
33       vector[argmin] = vector[j];
34       vector[j] = tmp;
35     }
36   }
37 }
38
39 main() {
40   double x[DIM];
41   scanVector(x,DIM);
42   selectionSort(x,DIM);
43   printVector(x,DIM);
44 }
```

# `while`-based loops

- Condition-controlled loops
- Pre-test vs. post-test loops
- Operators `++` and `--`

- `while`
- `do - while`

# The `while` loop

- Structure: `while(condition) statement`
  - `condition` is evaluated
  - If `condition` is true, `statement` is executed
  - This repeats until `condition` becomes false
- `condition` is checked *before* the block is executed
  - So-called pre-test loop
  - It is possible that `statement` is not executed at all, i.e., if `condition` is false
- `statement` can be a block of code

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int counter = 5;
 5
 6   while (counter > 0) {
 7     printf("%d ",counter);
 8     counter = counter-1;
 9   }
10   printf("\n");
11 }
```

- Output:

```
5 4 3 2 1
```

# Operators ++

- ++a and a++ are arithmetically equivalent to a=a+1
- Additional evaluation of the variable a
- Pre-increment ++a
  - First increase, then evaluate
- Post-increment a++
  - First evaluate, then increase

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int a = 0;
 5   int b = 43;
 6
 7   printf("1) a=%d, b=%d\n",a,b);
 8
 9   b = a++;
10   printf("2) a=%d, b=%d\n",a,b);
11
12   b = ++a;
13   printf("3) a=%d, b=%d\n",a,b);
14 }
```

- Output:

```
1) a=0, b=43
2) a=1, b=0
3) a=2, b=2
```

# Operators ++ and --

- Operators similar to `a++` and `++a`
  - Pre-decrement `--a`
    - First decrease, then evaluate
  - Post-decrement `a--`
    - First evaluate, then decrease
- Note the difference in condition-controlled loops!

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int counter = 5;
 5
 6   while (--counter>0) {
 7     printf("%d ",counter);
 8   }
 9   printf("\n");
10 }
```

- Output: `4 3 2 1` (for `--counter` in line 6)
- Output: `4 3 2 1 0` (for `counter--` in line 6)

# Bisection method (revisited)

- 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 $\tau \to 0$

# Bisection method (revisited)

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 double f(double x) {
 5   return x*x + exp(x) -2;
 6 }
 7
 8 double bisection(double a, double b, double tol){
 9   double fa = f(a);
10   double m = 0.5*(a+b);
11   double fm = 0;
12
13   while ( b - a > 2*tol ) {
14     m = 0.5*(a+b);
15     fm = f(m);
16     if ( fa*fm <= 0 ) {
17       b = m;
18     }
19     else {
20       a = m;
21       fa = fm;
22     }
23   }
24   return m;
25 }
26
27 main() {
28   double a = 0;
29   double b = 10;
30   double tol = 1e-12;
31   double x = bisection(a,b,tol);
32
33   printf("Approximate zero x=%g\n",x);
34   printf("Function value f(x)=%g\n",f(x));
35 }
```

- Using the variables `fa` and `fm` avoids a double evaluation of `f`

# Euclidean algorithm

- Input: Two integers $a, b \in \mathbb{N}$
- Task: Compute greatest common divisor $\gcd(a, b) \in \mathbb{N}$

- Euclidean algorithm:
  - If $a = b$, then $\gcd(a, b) = a$
  - If $a < b$, swap $a$ and $b$
  - It holds that $\gcd(a, b) = \gcd(a - b, b)$, because:
    - Let $g$ be a divisor of $a, b$
    - i.e., $ga_0 = a$ and $gb_0 = b$ with $a_0, b_0 \in \mathbb{N}$, $g \in \mathbb{N}$
    - Hence, $g(a_0 - b_0) = a - b$ and $a_0 - b_0 \in \mathbb{N}$
    - i.e., $g$ divides both $b$ and $a - b$
    - Hence, $\gcd(a, b) \leq \gcd(a - b, b)$
    - Similarly, $\gcd(a - b, b) \leq \gcd(a, b)$
  - Replace $a$ with $a - b$ and repeat the above steps

- Algorithm yields $\gcd(a, b)$ in finitely many steps:
  - If $a \neq b$, $n := \max\{a, b\} \in \mathbb{N}$ becomes smaller at each step
  - After finitely many steps, $a \neq b$ does not hold

# Euclidean algorithm

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int a = 200;
 5   int b = 110;
 6   int tmp = 0;
 7
 8   printf("gcd(%d,%d)=",a,b);
 9
10   while (a != b) {
11     if ( a < b) {
12       tmp = a;
13       a = b;
14       b = tmp;
15     }
16     a = a-b;
17   }
18
19   printf("%d\n",a);
20 }
```

- Computation of gcd of $a, b \in \mathbb{N}$
- Basic idea: $\gcd(a, b) = \gcd(a - b, b)$ for $a > b$
- For $a = b$, it holds that $\gcd(a, b) = a = b$

- Output:

```
gcd(200,110)=10
```

# Euclidean algorithm (improved)

- Key part of the previous implementation

```
10   while (a != b) {
11     if ( a < b) {
12       tmp = a;
13       a = b;
14       b = tmp;
15     }
16     a = a-b;
17   }
```

- Recall that $a\%b$ is the remainder of the integer division $a/b$
- Euclidean algorithm iterates $a := a - b$ until $a \leq b$
  - i.e., until $a = a\%b$
  - Finally, it holds that $a = 0$ and $b = \gcd(a, b)$

```
10   while (a != 0) {
11     if ( a < b) {
12       tmp = a;
13       a = b;
14       b = tmp;
15     }
16     a = a%b;
17   }
```

- The remainder always satisfies $a\%b < b$
  - i.e., always variable swap after the computation
  - Finally, it holds that $b = 0$ and $a = \gcd(a, b)$

```
10   while (b != 0) {
11     tmp = a%b;
12     a = b;
13     b = tmp;
14   }
```

# The do-while loop

- Structure: `do statement while(condition)`
  - `statement` is executed and `condition` is evaluated
  - If `condition` is false, break
  - This repeats until `condition` becomes false
- `condition` is checked *after* the block is executed
  - So-called post-test loop
  - At least *one* execution of `statement`
- `statement` can be a block of code

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int counter = 5;
 5
 6   do {
 7     printf("%d ",counter);
 8   }
 9   while (--counter>0);
10   printf("\n");
11 }
```

- Output:

  `5 4 3 2 1`

- `counter--` in line 9 yields the output: `5 4 3 2 1 0`

# Another example

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x[2] = {0,1};
 5   int tmp = 0;
 6   int c = 0;
 7
 8   printf("c=");
 9   scanf("%d",&c);
10
11   printf("%d %d ",x[0],x[1]);
12
13   do {
14     tmp = x[0]+x[1];
15     x[0] = x[1];
16     x[1] = tmp;
17     printf("%d ",tmp);
18   }
19   while(tmp<c);
20
21   printf("\n");
22 }
```

- The Fibonacci sequence tends to infinity
  - $x_0 := 0$, $x_1 := 1$ and $x_{n+1} := x_{n-1} + x_n$ for $n \in \mathbb{N}$
- Task: Given a threshold $c \in \mathbb{N}$, compute the first sequence member satisfying $x_n > c$
- Input $c = 1000$ yields output:

```
c=1000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

# break and continue

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5   int k = 0;
 6
 7   for (j=0; j<4; ++j) {
 8     if (j%2 == 0) {
 9       continue;
10     }
11     for (k=0; k < 10; ++k) {
12       printf("j=%d, k=%d\n",j,k);
13       if (k > 1) {
14         break;
15       }
16     }
17   }
18   printf("End: j=%d, k=%d\n",j,k);
19
20 }
```

- continue and break in statement of the loop
  - continue terminates the current iteration of the loop and continues with the next iteration
  - break terminates the loop and continues executing the code after the loop (if any)
- Output:

```
j=1, k=0
j=1, k=1
j=1, k=2
j=3, k=0
j=3, k=1
j=3, k=2
End:  j=4, k=2
```

# ‘as long as’ vs. ‘until’

```
 1  #include <stdio.h>
 2
 3 main() {
 4   int a = 200;
 5   int b = 110;
 6   int tmp = 0;
 7
 8   printf("gcd(%d,%d)=",a,b);
 9
10   while (1) {
11     if (a == b) {
12       break;
13     }
14     else if ( a < b) {
15       tmp = a;
16       a = b;
17       b = tmp;
18     }
19     a = a-b;
20   }
21
22   printf("%d\n",b);
23 }
```

- `for` and `while` loops runs depending on `condition`
  - i.e., the loop runs as long as `condition` is true
- Algorithm implementation can be only based on a termination condition `done`
  - i.e., it is interrupted if `done` is true
  - i.e., `condition` = logical complement of `done`
- Easy realization via infinite loop and `break`
  - Condition in line 10 is always true
  - Termination only via `break` in line 12

# Comments

- Why comments?

- `//`
- `/* ... */`

# Comments

- Comments are ignored by compiler
- Comments help read/understand the code
- Comments are necessary
  - to be able to understand even own code after some time
  - to help other people understand the code
- Comments are very useful during debugging
  - Commenting/uncommenting localized part of the source code 'to see what happens'
  - e.g., when dealing with syntax errors
- Important rules
  - Avoid special characters
  - Do not comment on each source code line
  - Usually at the beginning of the code, there is a comment with author and date of last change
    - Avoid conflicts with old versions...

# Comments in C

```
 1 #include <stdio.h>
 2
 3 main() {
 4   // printf("1 ");
 5   printf("2 ");
 6   /*
 7     printf("3");
 8     printf("4");
 9   */
10   printf("5");
11   printf("\n");
12 }
```

- In C, there are two types of comments:
  - Single-line comments
    - Start with `//` and until the end of the line
    - e.g., line 4
    - Originated from C++ syntax
  - Multi-line comments
    - Everything between `/*` (begin) and `*/` (end)
    - e.g., lines 6–9
    - `/*` ... `*/` cannot be nested (syntax error)
- Suggestion for a possible personal convention
  - Use `//` for *real* comments
  - Use `/*` ... `*/` for debugging
- Output:

  2 5

# Example: Euclidean algorithm

```
 1 // author: Dirk Praetorius
 2 // last modified: 19.03.2013
 3
 4 // Euclidean algorithm to compute the gcd
 5 // based on gcd(a,b) = gcd(a-b,b) for a>b
 6 // and gcd(a,b) = gcd(b,a)
 7
 8 int euclid(int a, int b) {
 9   int tmp = 0;
10
11   // reduction gcd(a,b) = gcd(a-b,b)
12   // realized via integer division, until
13   // b = 0. Then a==b, thus gcd = a
14
15   while (b != 0) {
16     tmp = b;
17     b = a%b;
18     a = tmp;
19   }
20
21   return a;
22 }
```

# Basic error control

- Avoid runtime error
- Intentional error-caused termination

- `gcc -c`
- `gcc -c -Wall`
- `assert`
- `#include <assert.h>`

# Motivation

- Fact: All programmers make mistakes
  - A program is usually not correct the first time
- Most of the programming time is usually spent in finding and correcting (own) mistakes
- Efficiency in error search is a big distinction between *professionals* and *beginners*
- Syntax errors are easy to identify
  - The compiler reports even the line number
  - Regularly check the syntax while programming
    - `gcc -c name.c` generates only object code
    - `gcc -Wall name.c` enables compiler warnings
- Runtime errors are more difficult to identify
  - The program works, but does do not what is intended
  - Sometimes the error is noticed after a long time
    $\Rightarrow$ This can have undesired effects (see later)

# Good habits that help programmers to avoid mistakes

- Follow programming convention
  - Use meaningful and consistent names
    - e.g., for variables, functions, etc.
  - Use a consistent layout of code
    - Indentation
- Break code into small functions
  - Each function expresses a logical action
    - Make testing and debugging easier
  - Check input for admissibility
    - Abortion if non-admissible
  - Ensure that the output is admissible
- Add explanations with comments in all relevant part of the code
  - e.g., non-obvious conditional statements
  - e.g., functions (specify aim, input, output)
    - meaningful names of variables and functions reduce the number of necessary comments
- Do not write all code at once
  - This is a usual mistake of beginners

# Library assert.h

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3
 4 void test(int x, int y) {
 5   assert(x<y);
 6   printf("It holds x < y\n");
 7 }
 8
 9 main() {
10   int x = 0;
11   int y = 0;
12
13   printf("x = ");
14   scanf("%d",&x);
15
16   printf("y = ");
17   scanf("%d",&y);
18
19   test(x,y);
20 }
```

- **Aim:** Termination with error message, whenever the function notices that input/output is not admissible
- `#include <assert.h>`
  - `assert(condition);` yields erroneous termination if `condition` does not hold
- **Input:**

  ```
  x = 2
  y = 1
  ```

- **Output:**

  ```
  Assertion failed:  (x<y), function test,
  file failedAssert.c, line 5.
  ```

# Example: Euclidean algorithm

```
 1 // author: Dirk Praetorius
 2 // last modified: 30.03.2017
 3
 4 // Euclidean algorithm to compute the gcd
 5 // based on gcd(a,b) = gcd(a-b,b) for a>b
 6 // and gcd(a,b) = gcd(b,a)
 7
 8 int euclid(int a, int b) {
 9   assert(a>0);
10   assert(b>0);
11   int tmp = 0;
12
13   // reduction gcd(a,b) = gcd(a-b,b)
14   // realized via integer division, until
15   // b = 0. Then a==b, thus gcd = a
16
17   while (b != 0) {
18     tmp = a%b;
19     a = b;
20     b = tmp;
21   }
22
23   return a;
24 }
```

- `assert` ensures admissibility of input
  - i.e., it must hold that $a, b \in \mathbb{N}$

# Testing

- Motivation
- Quality assurance
- Types of test

# Mistakes can be costly

- Patriot Missile failure (February 1991)
  - Wrong treatment of rounding error
  - 28 dead, 100 injured
- Sinking of *Sleipner A platform* (August 1991)
  - Wrong finite element analysis
  - Damage 700 million USD
- Explosion of Ariane 5 rocket (June 1996)
  - Conversion `double` $\rightarrow$ `int`
  - Damage 500 million USD

# Quality assurance

- Simple, but true:
  - Fact 1: Software is made by humans
  - Fact 2: Making mistakes is human
  - Consequence: Software can include mistakes
- Desirable: Find errors before it is too late
- The later an error is identified, the more difficult becomes its correction
- Ideal work organization:
  - 1/3 of time for programming
  - 1/3 of time for testing
  - 1/3 of time for documenting
- In practice:
  - Most of the time for programming
  - Much less time for testing
  - Even less time for documenting ;-)

# Testing

- Testing is the process of executing a program with the intent of finding errors
  - G. Myers: *The art of sofware testing* (1979)
- A test is the comparison of the behavior of a program (what it does) with its desired behavior (what it should do)
- In practice, it is impossible to test all functions in a program with all possible combinations of input data
  - i.e., testing is by definition incomplete
- Problems with incomplete testing
  - Tests allow only to find mistakes
  - Tests do not provide a rigorous proof that the code has no errors
  - Test cases themselves may be incorrect
- Important aspects in testing
  - Tests should consider 'realistic input'
  - Tests must be reproducible

# Types of test

- Structural tests (for each function)
  - Are all instructions executed or are there 'dead' parts of the code?
  - Analyze conditional statements!
    - e.g., check both occurrences (true/false) in `if ... else`
- Functional tests (for each function & program)
  - Does the function behave correctly for admissible input parameters? (i.e., is the result correct?)
  - Does the program (or a part of it) behave correctly? (i.e., is the result correct?)
  - Are non-admissible input parameters recognized?
  - Are limit cases and exceptions recognized and treated correctly?
  - What happens in the case of wrong input? (e.g., if the user makes a mistake)

# How to test?

- **Aim:** Is the function / program correct?
- Functional tests need test cases
  - (where the result is known)
- Tests should address conditional statements
  - Use parameters which lead to different outcomes in conditional statements
  - Try to test all outcome combinations
- Which cases are critical?
  - Inspect delicate parts of code
  - Be careful with type casting!
- Start early with testing
  - Test a function right after its implementation
  - Do not wait that the entire code is finished...
- Repeat all (!) tests after a change
  - i.e., test documentation is also important
- From now on, in the exercises you might find the following question:
  - How did you test your program?
  - Provide concrete examples to convince your tutor that you accurately tested your code

# Pointers

- Variables vs. pointers
- Dereferencing (= indirection of pointers)
- Address-of operator &
- Dereference operator *
- Call by reference

# Variables vs. Pointers

- Variable = Symbolic name (identifier) of a storage location (memory address) containing some quantity of information (value) of a specific type (data type)
- Pointer = Variable containing the address of a storage location
- Dereferencing = Accessing the content of a storage location using the corresponding pointer

# Pointers in C

- Pointers and variables are closely related in C:
  - Variable `var` ⇒ `&var` corresponding pointer
  - Pointer `ptr` ⇒ `*ptr` corresponding variable
  - In particular, `*&var` = `var` and `&*ptr` = `ptr`
- Like any other variable, a pointer must be declared before use
- The declaration must include the type of the pointer, since `*ptr` must be a variable
  - `int* ptr;` declares `ptr` as pointer to an `int`
- As usual, simultaneous declaration and initialization are possible
  - `int var;` declares the variable `var` of type `int`
  - `int* ptr = &var;` declares `ptr` and assigns the address location of the variable `var` to it
    - In such assignments the type of the pointer and the variable must coincide
      - Usually the compiler gives a warning, e.g., `incompatible pointer type`
- The same holds for all other data types
- Some tasks are done more easily with pointers
- Other tasks cannot be done without pointers
  - e.g., dynamic memory allocation (see later)

# An elementary example

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int var = 1;
 5   int* ptr = &var;
 6
 7   printf("a) var = %d, *ptr = %d\n", var, *ptr);
 8
 9   var = 2;
10   printf("b) var = %d, *ptr = %d\n", var, *ptr);
11
12   *ptr = 3;
13   printf("c) var = %d, *ptr = %d\n", var, *ptr);
14
15   var = 47;
16   printf("d) *(&var) = %d,", *(&var));
17   printf("*&var = %d\n", *&var);
18
19   printf("e) &var = %p\n", &var);
20 }
```

- `%p` placeholder in `printf` for addresses
- Output:

```
a) var = 1, *ptr = 1
b) var = 2, *ptr = 2
c) var = 3, *ptr = 3
d) *(&var) = 47,*&var = 47
e) &var = 0x7fff518baba8
```

# Call by reference in C

- In C, basic data types are passed to functions via *call by value*
  - e.g., `int`, `double`, pointers
- *Call by reference* can be realized with pointers

```
 1 #include <stdio.h>
 2
 3 void test(int* y) {
 4   printf("a) *y=%d\n", *y);
 5   *y = 43;
 6   printf("b) *y=%d\n", *y);
 7 }
 8
 9 main() {
10   int x = 12;
11   printf("c) x=%d\n", x);
12   test(&x);
13   printf("d) x=%d\n", x);
14 }
```

- Output:

  c) x=12

  a) *y=12

  b) *y=43

  d) x=43

# Summary

- Call by value
  - Functions receive values of variables as input parameters and copy them in local variables
  - Changes to the input parameters are not effective outside of the function
- Call by reference
  - Functions receive addresses of variables as input and assign new symbolic names for these within the respective function
  - Changes to the input parameters are thus effective also outside of the function
- In C, for basic data types, the standard approach is call by value
- Call by reference can be realized with pointers
- Arrays are always passed to functions via call by reference

# Why call by reference?

- Functions in C can have at most 1 return value
- If a functions should have more return values...

# Example

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3 #define DIM 5
 4
 5 void scanVector(double input[], int dim) {
 6   assert(dim > 0);
 7   int j = 0;
 8   for (j=0; j<dim; ++j) {
 9     input[j] = 0;
10     printf("%d: ",j);
11     scanf("%lf",&input[j]);
12   }
13 }
14
15 void determineMinMax(double vector[], int dim,
16                      double* min, double* max) {
17   int j = 0;
18   assert(dim > 0);
19
20   *max = vector[0];
21   *min = vector[0];
22   for (j=1; j<dim; ++j) {
23     if (vector[j] < *min) {
24       *min = vector[j];
25     }
26     else if (vector[j] > *max) {
27       *max = vector[j];
28     }
29   }
30 }
31
32 main() {
33   double x[DIM];
34   double max = 0;
35   double min = 0;
36   scanVector(x, DIM);
37   determineMinMax(x, DIM, &min, &max);
38   printf("min(x) = %f\n", min);
39   printf("max(x) = %f\n", max);
40 }
```

- `determineMinMax` *returns* maximum and minimum of a vector via call by reference

## Remarks about pointer declarations

- Spaces are ignored by the compiler
- `*` is applied only to the successive name
- In particular,
  - `int* pointer;`, `int *pointer;`, and `int*pointer;` are fully equivalent
  - `int* pointer, var;` declares a pointer to `int` and a variable of type `int`
  - `int *pointer1, *pointer2;` declares two pointers to `int`
- To improve readability, try to avoid declarations of lists including variables and pointers

## Pointer & arrays

- Declaration `int array[N];` automatically creates a pointer `array` of type `int*`
- `int array[];` and `int* array;` are equivalent declarations

# Function pointers

- Declaration
- Everything is a pointer!

# Function pointers

- Function calls can be realized with pointers
- Declaration of a function pointer:
  - `<return type> (*pointer)(<input>)`; declares a pointer `pointer` for functions with parameters `<input>` and return value of type `<return type>`
- If the address of a function is assigned to a function pointer, they must have the same structure
  - Same return value
  - Same list of input parameters
- Call of a function via a pointer is a normal function call

# Elementary example

```
 1 #include <stdio.h>
 2
 3 void output1(char* string) {
 4   printf("*%s*\n", string);
 5 }
 6
 7 void output2(char* string) {
 8   printf("#%s#\n", string);
 9 }
10
11 main() {
12   char string[] = "Hello World";
13   void (*output)(char* string);
14
15   output = output1;
16   output(string);
17
18   output = output2;
19   output(string);
20 }
```

- Declaration of a function pointer in line 13
- Assignment to function pointer in lines 15 and 18
- Function calls via pointer in lines 16 and 19
- **Output:**

```
*Hello World*
#Hello World#
```

# Bisection method

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3 #include <math.h>
 4
 5 double bisection(double (*fct)(double x),
 6                  double a, double b, double tol) {
 7   double m = 0;
 8   double fa = 0;
 9   double fm = 0;
10
11   assert(a < b);
12   fa = fct(a);
13   assert(fa*fct(b) <= 0);
14
15   while ( b-a > tol) {
16     m = (a+b)/2;
17     fm = fct(m);
18     if ( fa*fm <= 0 ) {
19       b = m;
20     }
21     else {
22       a = m;
23       fa = fm;
24     }
25   }
26   return m;
27 }
28
29 double f(double x) {
30   return x*x+exp(x)-2;
31 }
32
33 main() {
34   double a = 0;
35   double b = 10;
36   double tol = 1e-12;
37
38   double x = bisection(f, a, b, tol);
39   printf("Approximate zero x=%1.15e\n", x);
40 }
```

- Approximation of zeros of $f(x) = x^2 + e^x - 2$

# Basic data types

- Arrays & pointers
- `sizeof`

# Basic data types

- Basic data types in C
  - Data type for characters (e.g., letters)
    - `char`
  - Data type for integers
    - `int`
  - Data types for floating point numbers
    - `float`
    - `double`

- Pointers are treated as basic data types
- Modifiers can be applied to basic types, e.g.,
  - `short`/`long` modifies the amount of storage allocated for the variable
    - e.g., `long double` is an extended precision floating-point number
    - Amount of storage depends on compiler
- Declaration and use as before
- Arrays and pointers of basic types possible
- More details later!

# The command sizeof

```
 1 #include <stdio.h>
 2
 3 void printSizeOf(double vector[]) {
 4   printf("sizeof(vector) = %d\n", sizeof(vector));
 5 }
 6
 7 main() {
 8   int var = 43;
 9   double array[12];
10   double* ptr = array;
11
12   printf("sizeof(var) = %d\n", sizeof(var));
13   printf("sizeof(double) = %d\n", sizeof(double));
14   printf("sizeof(array) = %d\n", sizeof(array));
15   printf("sizeof(ptr) = %d\n", sizeof(ptr));
16   printSizeOf(array);
17 }
```

- If `var` is a variable with basic data type, `sizeof(var)` returns the size of the variable in bytes
- If `type` is a data type, `sizeof(type)` returns the size of a variable of this type in bytes
- If `array` is a *local static array*, `sizeof(array)` returns the size of the array in bytes
- Internally `ptr` and `array` are two pointers to `double` and contain (= point to) the same storage location
- Output:

```
sizeof(var) = 4
sizeof(double) = 8
sizeof(array) = 96
sizeof(ptr) = 8
sizeof(vector) = 8
```

# Functions

- Basic data types are passed to functions via call by value
- The return value of a function can only be empty (i.e., `void`) or a basic data type

# Arrays do not exist in C!

- Strictly speaking, arrays do not exist in C!
- Declaration `int array[N];`
  - creates a pointer `array` of type `int*`
  - allocates storage of size `N` times the size of an `int` starting at the storage location with address stored in `array`
  - i.e., `array` contains the address of `array[0]`
- This explains why arrays are always passed with call by reference to functions
- What is really happening is that the function receives a pointer to the storage location in which the array is stored

# Runtime error

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3
 4 double* scanVector(int length) {
 5   assert(length > 0);
 6   double vector[length];
 7   int j = 0;
 8   for (j=0; j<length; ++j) {
 9     vector[j] = 0;
10     printf("vector[%d] = ", j);
11     scanf("%lf", &vector[j]);
12   }
13   return vector;
14 }
15
16 main() {
17   double* x;
18   int j = 0;
19   int dim = 0;
20
21   printf("dim = ");
22   scanf("%d", &dim);
23
24   x = scanVector(dim);
25
26   for (j=0; j<dim; ++j) {
27     printf("x[%d] = %f\n", j, x[j]);
28   }
29 }
```

- Syntax of this program is fine
- Problem: The storage for x is declared inside the function and is therefore lost after the function call in line 24, i.e., the pointer in lines 6 and 13 points to some location, which is now inaccessible
- Workaround: call by reference (as before) or manual memory allocation (discussed now)

# Dynamic vectors

- Static & dynamic vectors
- Vectors & pointers
- Dynamic memory allocation

- `stdlib.h`
- `NULL`
- `malloc`, `realloc`, `free`

- `#ifndef ... #endif`

# Static vectors

- `double array[N];` declares a static vector `array` of length `N` with `double` coefficients
  - Indexing `array[j]` with $0 \leq j \leq N-1$
  - `array` is internally of type `double*`
    - It contains the address of `array[0]`
    - It is a so-called base pointer
  - The length `N` cannot be changed during the program execution

- Functions cannot determine the length `N`
  - It must be passed as an input parameter

# Memory allocation

- Manual memory allocation allows for vector with dynamic length

- Inclusion of the standard library `stdlib.h`
  - `#include <stdlib.h>`
  - Commands `malloc`, `free`, `realloc`

- `pointer = malloc(N*sizeof(type));`
  - Memory allocation for a vector of length `N` with coefficients of type `type`
    - `malloc` wants input in bytes $\rightarrow$ `sizeof`
  - `pointer` must be of type `type*`
    - `pointer` receives the address of the first coefficient `pointer[0]` (base pointer)

- **Common (runtime) error:** Forgetting `sizeof`!

- **Watch out!** Allocated memory is not initialized!

- **Convention:** Pointers without memory are initialized with the value `NULL`
  - This immediately leads to an error if one tries to access its (non-existing) storage location

- `malloc` returns `NULL`, if the allocation is unsuccessful
  - i.e., the memory could not be allocated

# Memory deallocation

- `free(pointer)` deallocates previously allocated memory
  - `pointer` must be the output of `malloc`
- **Watch out!**
  - The memory is deallocated, but the `pointer` still exists
  - Subsequent access leads to runtime error
- **Watch out!**
  - Do not forget to deallocate allocated memory and set the corresponding pointer to `NULL`

# Example

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <assert.h>
 4
 5 double* scanVector(int length) {
 6   int j = 0;
 7   double* vector = NULL;
 8   assert(length > 0);
 9
10   vector = malloc(length*sizeof(double));
11   assert(vector != NULL);
12   for (j=0; j<length; ++j) {
13     vector[j] = 0;
14     printf("vector[%d] = ", j);
15     scanf("%lf", &vector[j]);
16   }
17   return vector;
18 }
19
20 void printVector(double* vector, int length) {
21   int j = 0;
22   assert(vector != NULL);
23   assert(length > 0);
24
25   for (j=0; j<length; ++j) {
26     printf("vector[%d] = %f\n", j, vector[j]);
27   }
28 }
29
30 main() {
31   double* x = NULL;
32   int dim = 0;
33
34   printf("dim = ");
35   scanf("%d", &dim);
36
37   x = scanVector(dim);
38   printVector(x, dim);
39
40   free(x);
41   x = NULL;
42 }
```

# Dynamic vectors

- `pointer = realloc(pointer, Nnew*sizeof(type))`
  - Change of memory allocation
    - Additional allocation if `Nnew` $>$ `N`
    - Reduction of allocated memory if `Nnew` $<$ `N`
  - Former content remains (if possible)
  - Return value `NULL` if reallocation unsuccessful

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <assert.h>
 4
 5 main() {
 6   int N = 5;
 7   int Nnew = 10;
 8   int j = 0;
 9
10   int* array = malloc(N*sizeof(int));
11   assert(array != NULL);
12   for (j=0; j<N; ++j){
13     array[j] = j;
14   }
15
16   array = realloc(array, Nnew*sizeof(int));
17   assert(array != NULL);
18   for (j=N; j<Nnew; ++j){
19     array[j] = 10*j;
20   }
21
22   for (j=0; j<Nnew; ++j){
23     printf("%d ", array[j]);
24   }
25   printf("\n");
26   free(array);
27   array = NULL;
28 }
```

- Output:

```
0 1 2 3 4 50 60 70 80 90
```

# Remarks

- Store and do not change base pointer (= output of `malloc` or `realloc`)
  - Necessary for error-free `free` and `realloc`
- Do not forget `sizeof` with `malloc` and `realloc`
  - Type of base pointer must agree with `sizeof`
- Check that the output of `malloc` and `realloc` is not `NULL`, e.g., with `assert`
  - To ensure that the allocation was successful
- Allocated memory is not initialized
  - Initialization right after allocation
- Store the length of a dynamic array
  - It cannot be determined by the program
- Deallocate unneeded memory
  - Deallocate before the end of the block `}`, otherwise the base pointer is gone
- Set pointers without memory to `NULL`
  - Error message if the program tries to access
- Never use `realloc` and `free` on static arrays
  - Runtime error, as deallocation is automatically done by the compiler

# Vector library

- Subdivision of the source code into more files
- Precompiler, Compiler, Linker
- Object code

- `gcc -c`
- `make`

# Source code subdivision

- Subdivide long source codes into more files
- Advantage:
  - More clear structure
  - Creation of libraries
    - Re-use of old code
    - Help to avoid mistakes
- `gcc name1.c name2.c ...`
  - Creation of *one* executable
  - Ordering of the files is not important
  - Similar to `gcc all.c`
    - If `all.c` contains the entire source code
  - Function names must be unambiguous
  - `main()` can appear only once

# Precompiler, Compiler & Linker

- The compilation process consists of four steps
  - 1. Preprocessing
  - 2. Compiling
  - 3. Assembling → Object code
  - 4. Linking → Executable

- Library = Precompiled object code with corresponding header file

- Standard linker in Unix is `ld`

- `gcc -c name.c` performs the compilation process until Step 3
  - Creation of object code `name.o`
  - Also good for debugging of syntax errors

- To compile a file with additional object code:
  - `gcc name.c bib1.o bib2.o ...`
  - `gcc name.o bib1.o bib2.o ...`
  - Ordering and number of files do not matter

- Aim: Create libraries with certain functionality
  - Useful to reduce compilation time and mistakes

# A first library

```
 1 #ifndef _DYNAMICVECTORS_
 2 #define _DYNAMICVECTORS_
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <assert.h>
 7
 8 // allocate + initialize dynamic double vector of length n
 9 double* mallocVector(int n);
10
11 // free a dynamic double vector and set the pointer to NULL
12 double* freeVector(double* vector);
13
14 // extend dynamic double vector and initialize new entries
15 double* reallocVector(double* vector, int n, int nnew);
16
17 // allocate dynamic double vector of length n and read
18 // entries from keyboard
19 double* scanVector(int n);
20
21 // print dynamic double vector of length n to shell
22 void printVector(double* vector, int n);
23
24 #endif
```

- Header file `dynamicvectors.h` of the library
  - All function signatures
  - Comments about function responsibilities
- The header file starts with

  ```
  #ifndef NAME
  #define NAME
  ```
- The header file ends with

  ```
  #endif
  ```
- It allows for multiple inclusions and avoids multiple declarations

# Source code (1/2)

```
 1 #include "dynamicvectors.h"
 2
 3 double* mallocVector(int n) {
 4   int j = 0;
 5   double* vector = NULL;
 6   assert(n > 0);
 7
 8   vector = malloc(n*sizeof(double));
 9   assert(vector != NULL);
10
11   for (j=0; j<n; ++j) {
12     vector[j] = 0;
13   }
14   return vector;
15 }
16
17 double* freeVector(double* vector) {
18   free(vector);
19   return NULL;
20 }
21
22 double* reallocVector(double* vector, int n, int nnew) {
23   int j = 0;
24   assert(vector != NULL);
25   assert(n > 0);
26   assert(nnew > 0);
27
28   vector = realloc(vector, nnew*sizeof(double));
29   assert(vector != NULL);
30   for (j=n; j<nnew; ++j) {
31     vector[j] = 0;
32   }
33   return vector;
34 }
```

- Inclusion of header files (line 1)
  - `#include "..."` including the full path
  - `#include <...>` for the standard directory

# Source code (2/2)

```
36 double* scanVector(int n) {
37   int j = 0;
38   double* vector = NULL;
39   assert(n > 0);
40
41   vector = mallocVector(n);
42   assert(vector != NULL);
43
44   for (j=0; j<n; ++j) {
45     printf("vector[%d] = ", j);
46     scanf("%lf", &vector[j]);
47   }
48   return vector;
49 }
50
51 void printVector(double* vector, int n) {
52   int j = 0;
53   assert(vector != NULL);
54   assert(n > 0);
55
56   for (j=0; j<n; ++j) {
57     printf("%d: %f\n", j, vector[j]);
58   }
59 }
```

# Main program

```
 1 #include "dynamicvectors.h"
 2
 3 main() {
 4   double* x = NULL;
 5   int n = 10;
 6   int j = 0;
 7   x = mallocVector(n);
 8   for (j=0; j<n; ++j) {
 9     x[j] = j;
10   }
11   x = reallocVector(x,n,2*n);
12   for (j=n; j<2*n; ++j) {
13     x[j] = 10*j;
14   }
15   printVector(x,2*n);
16   x = freeVector(x);
17 }
```

- Main program includes the header of the library
- Compilation via
  - `gcc -c dynamicvectors.c`
    - Creation of object code `dynamicvectors.o`
  - `gcc dynamicvectors_main.c dynamicvectors.o`
    - Creation of executable `a.out`

# Static libraries and make

```
1 exe : main.o dynamicvectors.o
2         gcc -o exe main.o dynamicvectors.o
3
4 main.o : dynamicvectors_main.c dynamicvectors.h
5         gcc -c dynamicvectors_main.c -o main.o
6
7 dynamicvectors.o : dynamicvectors.c dynamicvectors.h
8         gcc -c dynamicvectors.c
```

- UNIX command `make` allows to treat code dependencies automatically
  - Automatization saves time in compiling
  - New object code is created only for changed source code
- Calling `make` builds targets specified in `Makefile`
- Calling `make -f filename` uses `filename` instead
- The file includes dependencies and commands, e.g.,
  - Line 1 = Dependencies (without indentation)
    - The file `exe` depends on...
  - Line 2 = Commands (one tab-indentation)
    - If `exe` is older than its dependencies, the command is executed (and only in that case!)

# Strings

- Static & dynamic strings
- `"..."` vs. `'...'`

- `string.h`

# Strings

- Strings = Arrays of `char`
  - Static: `char array[N];`
    - `N` = Static length
    - Simultaneous declaration and initialization

      `char array[] = "text";`
  - Dynamic: Type `char*`
    - Same approach as before for vectors

- Define static strings with double quotation marks `"..."`
- Access to single characters with single quotation marks `'...'`
- Access to sub-strings not possible

- Strings end with the null character `\0`
  - Take this into account when determining the length of dynamic strings
    - Allocate 1 byte more
    - Do not forget `\0` at the end
  - For static strings this is automatic

    i.e., effective length `N+1` and `array[N]='\0'`

- Static strings can also be passed to functions (in double quotation marks)
  - e.g., `printf("Hello World!\n");`

# Functions for string manipulation

- Important functions in `stdio.h`
  - `sprintf`: Conversion variable → string
  - `sscanf`: Conversion string → variable
- Several functions in `stdlib.h`, e.g.,
  - `atof`: Conversion string → `double`
  - `atoi`: Conversion string → `int`
- Many functions in `string.h`, e.g.,
  - `strchr`, `memchr`: Search for `char` in a string
  - `strcmp`, `memcmp`: Compare two strings
  - `strcpy`, `memcpy`: Copy strings
  - `strlen`: Determine string length (without `\0`)
- Include header files with `#include <name>`!
- Be careful when working with strings: Functions cannot know whether the storage allocated for the output string is sufficient (→ Runtime error!)

# Example

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <assert.h>
 5
 6 char* cloneString(char* source) {
 7   int length = 0;
 8   char* result = NULL;
 9   assert(source != NULL);
10
11   length = strlen(source);
12   result = malloc((length+1)*sizeof(char));
13   strcpy(result, source);
14   return result;
15 }
16
17 main() {
18   char* string1 = "Hello World?";
19   char* string2 = cloneString(string1);
20   string2[11] = '!';
21   printf("%s %s\n", string1, string2);
22 }
```

- Output:

  ```
  Hello World? Hello World!
  ```

- Declaration and initialization of strings with double quotation marks `"..."` creates a static string with null character at the end (line 18)
- Access to single characters of a string with single quotation marks `'...'` (line 20)
- Placeholder for strings in `printf` is `%s` (line 21)

# Structures

- Why structures?
- Members
- Point operator `.`
- Arrow operator `->`
- Shallow copy vs. deep copy

- `struct`
- `typedef`

# Declaration of structures

- **Functions**
  - Callable group of statements that together perform a task
  - Abstraction (structured programming)
- **Structures**
  - Combination of variables of different types in a new data type
  - Abstraction with data
- **Example:** Management of SciProgMath students
  - The same data for each student

```
 1 // Declaration of structure
 2 struct _Student_ {
 3   char* firstname;
 4   char* lastname;
 5   int studentID;
 6   int studiesID;
 7   double pointsExam;
 8   double pointsExercise;
 9 };
10
11 // Declaration of corresponding data type
12 typedef struct _Student_ Student;
```

- Semicolon after structure declaration block
- Creation of new variable type `Student`

# Structures & Members

- Variables of a structure are called members
- Access to members with point operator
  - `var` variable of type `Student`
  - e.g., member `var.firstname`

```
 1 // Declaration of structure
 2 struct _Student_ {
 3   char* firstname;
 4   char* lastname;
 5   int studentID;
 6   int studiesID;
 7   double pointsExam;
 8   double pointsExercise;
 9 };
10
11 // Declaration of corresponding data type
12 typedef struct _Student_ Student;
13
14 main() {
15   Student var;
16   var.firstname = "Max";
17   var.lastname = "Mustermann";
18   var.studentID = 0;
19   var.studiesID = 680;
20   var.pointsExam = 25.;
21   var.pointsExercise = 35.;
22 }
```

# Remarks on structures

- Originally, the C standard did not allow to use
  - Structures as input parameter of a function
  - Structures as output parameter of a function
  - Assignment operator (=) for an entire structure

- In the meantime, it is allowed, however:
  - Pass structures to functions dynamically (via pointers)
  - Write assignment (= copy) by yourself
  - Assignment (=) creates a so-called *shallow copy*

- Shallow copy:
  - Only the first level is copied
  - i.e., values for basic variables
  - i.e., addresses for pointers
  - Moreover: Copy has the same dynamic data

- Deep copy:
  - All levels of a structure are copied
  - Plus: Copy of the dynamic data

# Structures: Memory allocation

- Create also the functions
  - `newStudent`: Memory allocation and initialization
  - `freeStudent`: Memory deallocation
  - `cloneStudent`: Complete copy of the full structure including the dynamic data
    e.g., member `firstname` (so-called *deep copy*)
  - `copyStudent`: Copy of the first level excluding the dynamic data (so-called *shallow copy*)

```
 1 Student* newStudent() {
 2   Student* pointer = malloc(sizeof(Student));
 3   assert( pointer != NULL);
 4
 5   (*pointer).firstname = NULL;
 6   (*pointer).lastname = NULL;
 7   (*pointer).studentID = 0;
 8   (*pointer).studiesID = 0;
 9   (*pointer).pointsExam = 0.;
10   (*pointer).pointsExercise = 0.;
11
12   return pointer;
13 }
```

# Structures & arrow operator

- In the program, `pointer` is of type `Student*`
- Access to members, e.g., `(*pointer).firstname`
  - Better syntax `pointer->firstname`
- Structures never static, always dynamic
  - Directly use `student` for type `Student*`
- Better implementation of the function `newStudent` below

```
 5 // Declaration of structure
 6 struct _Student_ {
 7   char* firstname;
 8   char* lastname;
 9   int studentID;
10   int studiesID;
11   double pointsExam;
12   double pointsExercise;
13 };
14
15 // Declaration of corresponding data type
16 typedef struct _Student_ Student;
17
18 // allocate and initialize new student
19 Student* newStudent() {
20   Student* student = malloc(sizeof(Student));
21   assert(student != NULL);
22
23   student->firstname = NULL;
24   student->lastname = NULL;
25   student->studentID = 0;
26   student->studiesID = 0;
27   student->pointsExam = 0.;
28   student->pointsExercise = 0.;
29
30   return student;
31 }
```

# Structures: Memory deallocation

- Deallocation of a dynamic variable of type `Student`
- Be careful: Deallocate the dynamically allocated memory before deallocating the pointer to the structure

```
33 // free memory allocation
34 Student* freeStudent(Student* student) {
35   assert(student != NULL);
36
37   if (student->firstname != NULL) {
38     free(student->firstname);
39   }
40
41   if (student->lastname != NULL) {
42     free(student->lastname);
43   }
44
45   free(student);
46   return NULL;
47 }
```

# Shallow Copy

- Copy of a dynamic variable of type Student
  - Copy of only the first level of the structure excluding the dynamically allocated memory

```
49 // shallow copy of student
50 Student* copyStudent(Student* student) {
51   Student* copy = newStudent();
52   assert(student != NULL);
53
54   // Watch out! Pointer!
55   copy->firstname = student->firstname;
56   copy->lastname = student->lastname;
57
58   // Copy of the simple data
59   copy->studentID = student->studentID;
60   copy->studiesID = student->studiesID;
61   copy->pointsExam = student->pointsExam;
62   copy->pointsExercise = student->pointsExercise;
63
64   return copy;
65 }
```

# Deep Copy

- Copy of a dynamic variable of type Student
  - Copy of all levels of the structure
    including the dynamically allocated memory
- Be careful: Copy also the member with dynamically allocated memory

```
67 // deep copy of student
68 Student* cloneStudent(Student* student) {
69   Student* copy = newStudent();
70   int length = 0;
71   assert( student != NULL);
72
73   if (student->firstname != NULL) {
74     length = strlen(student->firstname)+1;
75     copy->firstname = malloc(length*sizeof(char));
76     assert(copy->firstname != NULL);
77     strcpy(copy->firstname, student->firstname);
78   }
79
80
81   if (student->lastname != NULL) {
82     length = strlen(student->lastname)+1;
83     copy->lastname = malloc(length*sizeof(char));
84     assert(copy->lastname != NULL);
85     strcpy(copy->lastname, student->lastname);
86   }
87
88   copy->studentID = student->studentID;
89   copy->studiesID = student->studiesID;
90   copy->pointsExam = student->pointsExam;
91   copy->pointsExercise = student->pointsExercise;
92
93   return copy;
94 }
```

# Arrays of structures

- Aim: Generate array with SciProgMath students
- No static arrays, dynamic arrays
  - Student data are of type `Student`
  - Hence, they are managed with type `Student*`
  - Hence, an array of type `Student**` is needed

```
1 // Declare array
2 Student** participant = malloc(N*sizeof(Student*));
3
4 // Allocate memory for participants
5 for (j=0; j<N; ++j){
6   participant[j] = newStudent();
7 }
```

- Access to members as before
  - `participant[j]` has type `Student*`
  - Hence, e.g., `participant[j]->firstname`

# Nesting of structures

```
 1 struct _Address_ {
 2   char* street;
 3   char* number;
 4   char* city;
 5   char* zip;
 6 };
 7 typedef struct _Address_ Address;
 8
 9 struct _Employee_ {
10   char* firstname;
11   char* lastname;
12   char* title;
13   Address* home;
14   Address* office;
15 };
16 typedef struct _Employee_ Employee;
```

- Organize data of employees
  - Name, private address, work address
- For `employee` of type `Employee*`
  - `employee->home` is a pointer to `Address`
  - Hence, e.g., `employee->home->city`
- Be careful with allocating, deallocating, copying

# Structures & mathematics

- Structures for mathematical objects:
  - Vectors in $\mathbb{R}^n$

# Structures and vectors

```
 1 #ifndef _STRUCT_VECTOR_
 2 #define _STRUCT_VECTOR_
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <assert.h>
 7 #include <math.h>
 8
 9 // declaration of new data type Vector
10 typedef struct _Vector_ {
11   int dimension;
12   double* coefficient;
13 } Vector;
14
15 // Allocate and initialize new vector of length n
16 Vector* newVector(int dimension);
17
18 // free storage of allocated vector and return NULL
19 Vector* freeVector(Vector* X);
20
21 // return length of a vector
22 int getVectorDimension(Vector* X);
23
24 // return coefficient Xi of vector X
25 double getVectorCoefficient(Vector* X, int i);
26
27 // assign new value to coefficient Xi of vector X
28 void setVectorCoefficient(Vector* X, int i, double Xi);
29
30 // some example functions...
31 Vector* inputVector();
32 double normVector(Vector* X);
33 double scalarProductVector(Vector* X, Vector* Y);
34
35 #endif
```

- Data type to store $x \in \mathbb{R}^n$
  - Dimension $n$ of type `int`
  - Array coefficients $x_j$ to store `double`

# Allocate a vector

- The vector length $n \in \mathbb{N}$ is passed to the function
- Structure allocation, assignment of dimension $n$
- Allocation and initialization of the vector array

```
 3 Vector* newVector(int dimension) {
 4   int i = 0;
 5   Vector* X = NULL;
 6
 7   assert(dimension > 0);
 8
 9   X = malloc(sizeof(Vector));
10   assert(X != NULL);
11
12   X->dimension = dimension;
13   X->coefficient = malloc(dimension*sizeof(double));
14   assert(X->coefficient != NULL);
15
16   for (i=0; i<dimension; ++i) {
17     X->coefficient[i] = 0;
18   }
19   return X;
20 }
```

# Deallocate a vector

- Deallocate array
- Deallocate structure
- Return value NULL

```
22 Vector* freeVector(Vector* X) {
23   assert(X != NULL);
24   assert(X->coefficient != NULL);
25   free(X->coefficient);
26   free(X);
27
28   return NULL;
29 }
```

# Access to structures

- Good programming style (unfortunately not very common): No direct access to structure members
- Better approach:
  - Write `set` and `get` functions for each member
  - So-called *mutator functions*

```
31 int getVectorDimension(Vector* X) {
32   assert(X != NULL);
33   return X->dimension;
34 }
35
36 double getVectorCoefficient(Vector* X, int i) {
37   assert(X != NULL);
38   assert((i >= 0) && (i < X->dimension));
39   return X->coefficient[i];
40 }
41
42 void setVectorCoefficient(Vector* X, int i, double Xi){
43   assert(X != NULL);
44   assert((i >= 0) && (i < X->dimension));
45   X->coefficient[i] = Xi;
46 }
```

- Writing data is not allowed without `set`!
- Reading data is not allowed without `get`!
- This approach is more compatible with later changes of the data structure
- This approach helps to avoid data inconsistencies (and very often also runtime errors)

# Example: Read vector

```
48 Vector* inputVector() {
49
50   Vector* X = NULL;
51   int i = 0;
52   int dimension = 0;
53   double input = 0;
54
55   printf("Dimension of vector n=");
56   scanf("%d", &dimension);
57   assert(dimension > 0);
58
59   X = newVector(dimension);
60   assert(X != NULL);
61
62   for (i=0; i<dimension; ++i) {
63     input = 0;
64     printf("x[%d]=", i);
65     scanf("%lf", &input);
66     setVectorCoefficient(X, i, input);
67   }
68
69   return X;
70 }
```

- Read $n \in \mathbb{N}$ and a vector $x \in \mathbb{R}^n$ from the keyboard

# Example: Euclidean norm

```
72 double normVector(Vector* X) {
73
74   double Xi = 0;
75   double norm = 0;
76   int dimension = 0;
77   int i = 0;
78
79   assert(X != NULL);
80
81   dimension = getVectorDimension(X);
82
83   for (i=0; i<dimension; ++i) {
84     Xi = getVectorCoefficient(X, i);
85     norm = norm + Xi*Xi;
86   }
87   norm = sqrt(norm);
88
89   return norm;
90 }
```

- Compute $\|x\| := \left( \sum_{j=1}^{n} x_j^2 \right)^{1/2}$ for $x \in \mathbb{R}^n$

# Example: Scalar product

```
 92 double scalarProductVector(Vector* X, Vector* Y) {
 93
 94   double Xi = 0;
 95   double Yi = 0;
 96   double product = 0;
 97   int dimension = 0;
 98   int i = 0;
 99
100   assert(X != NULL);
101   assert(Y != NULL);
102
103   dimension = getVectorDimension(X);
104   assert(dimension == getVectorDimension(Y));
105
106   for (i=0; i<dimension; ++i) {
107     Xi = getVectorCoefficient(X, i);
108     Yi = getVectorCoefficient(Y, i);
109     product = product + Xi*Yi;
110   }
111
112   return product;
113 }
```

- Compute $x \cdot y := \sum_{j=1}^{n} x_j y_j$ for $x, y \in \mathbb{R}^n$