# Scientific programming in mathematics

Prof. Dirk Praetorius

Priv.-Doz. Giovanni Di Fratta

Institute of Analysis
and Scientific Computing

# General information

- Rights & Duties
- Grading
- Course material
- Schedule

# Homepages

- TISS homepage (login required)

  https://tiss.tuwien.ac.at + search for lecture
  (courseNr=101776, semester=2021S)
  - Registration
  - Lecture forum

- TUWEL homepage (login required)

  https://tuwel.tuwien.ac.at/course/view.php?id=35497
  - Rights & Duties
  - Grading
  - Schedule & Course material
  - Weekly assignments (Download & Handing-in)

# Rights & Duties

- Structure of the lecture
  - Reading course
  - Exercise sessions
    - Friday, 10:15–11:45
    - Mandatory attendance
    - Start → March 12, 2021

- Grading
  - 50% → Performance in final test
    - at the end of June
  - 50% → Performance in exercise sessions
    - Number of solved exercises
    - Active participation

- Help for students
  - via Email
  - Lecture forum on TISS
    - 24/7

# Exercise sessions

- Fridays, 10:15-11:45, via ZOOM
- 12 sessions
  - 1: March 12, 2021
  - 2: March 19, 2021
  - 3: March 26, 2021

  **Easter Break: Monday, 29 March 2021 to Saturday, 10 April 2021**
  - 4: April 16, 2021
  - 5: April 23, 2021
  - 6: April 30, 2021
  - 7: May 7, 2021

  **Rectors Day/dies academicus (no classes): Friday, 14 May 2021**
  - 8: May 21, 2021
  - 9: May 28, 2021
  - 10: June 4, 2021
  - 11: June 11, 2021
  - 12: June 18, 2021
- Mandatory attendance
- `lva.student.tuwien.ac.at`

## Course material

- Slides (formally nothing else is necessary)

## Books (optional!)

- 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>`

# Program

- A computer program (or, briefly, a program) is a collection of statements, written in a programming language, that performs a specific task when executed by a computer
  - Statement = declaration or instruction
    - Declaration = e.g., definition of variables
    - Instruction = ‘do something’
  - Example: Search for a phonebook entry
  - Example: Compute the value of a integral

# Algorithm

- An algorithm is a finite sequence of unambiguous operations which specify how to solve a problem (or a class of problems)
  - Example: Compute the solution of a linear system of equations via Gaussian elimination
  - Example: Compute the zero of a quadratic polynomial using the quadratic formula
- There exist many algorithms to solve a problem
  - Not all algorithms are ‘good’
    - What does ‘good’ mean? (see later)

# Source code

- Text of a computer program written in a programming language
- It is processed step-by-step while executing or compiling
- In the easiest situation: sequentially
  - Line-by-line
  - From the top to the bottom

# Programming language

- Programming languages can be classified into interpreted and compiled languages
- The interpreter executes source code line-by-line during the 'translation'
  - i.e., translate and execute at the same time
  - e.g., Matlab, Java, PHP
- The compiler 'translates' the source code and produces a stand-alone program written in a machine-dependent object code (executable)
  - i.e., first translate, then execute
  - e.g., C, C++, Fortran
- Alternative classification:
  - Imperative languages, e.g., Matlab, C, Fortran
  - Object-oriented languages, e.g., C++, Java
  - Functional languages, e.g., Lisp

# Be careful!

- C is a compiled programming language
- 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 compiler 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 negative evaluation. . .

# How to write a program in C?

- Start your favorite text editor
  e.g., `nano`, `emacs`, `vim`, `gedit`, `atom`, ...
- 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 for didactic purposes)
- 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 operatingsystem
  - 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 `int main() {`

```
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`
- `int x;` declares a variable `x` of type `int`

# 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

```
#include <stdio.h>

main() {
  int x = 1;
  int y = 2;

  int tmp = 0;

  printf("a) x=%d, y=%d, tmp=%d\n",x,y,tmp);

  tmp = x;
  x = y;
  y = tmp;

  printf("b) x=%d, y=%d, tmp=%d\n",x,y,tmp);
}
```

- The symbol `=` is the assignment operator
  - Assignment always from the left to the right
- `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)
- Some notation for floating point numbers:
  - Minus sign -, if negative
  - Predecimal positions
  - Decimal separator (point)
  - Decimal positions
  - e or E with *integer* exponent
    (10th power!), e.g., $2\mathrm{e}2 = 2\mathrm{E}2 = 2 \cdot 10^2 = 200$
- Hence: `2./3.` is floating point division $\approx 0.\overline{6}$

# Type casting

- Operators can work also with variables with different type
- Before execution the variables 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 has `x+y` in lines 7–8?
  - The ‘strongest’ 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, type casting to the 'strongest' 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

```
#include <stdio.h>

main() {
  int a = 2;
  int b = 3;
  double dbl = (double) a / b;

  int i = dbl;

  printf("a) %f\n",dbl);
  printf("b) %f\n",dbl*b);
  printf("c) %d\n",i);
  printf("d) %d\n",i*b);
}
```

- 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!

# Simple conditional statements

- Logical operators `==` `!=` `>` `>=` `<` `<=`
- Logical connectives `!` `&&` `||`
- True/false for statements
- Conditional statements

- `if`
- `if - else`

# Logical operators

- Let a, b 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 (≠ assignment operator) |
| != | inequality |
| > | strictly larger |
| >= | larger than or equal to |
| < | strictly smaller |
| <= | smaller than or equal to |

- Advice: Put comparisons in brackets!
  - Not always necessary, but sometimes helpful!
- Logical connectives:

| | |
|---|---|
| ! | 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 do 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 $\Rightarrow$ `statementA` is executed
  - Condition false $\Rightarrow$ `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., cond. is true\n");
17   }
18   else {
19     printf("var2 == 0, i.e., cond. 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?

```
#include <stdio.h>

main() {
  int x = 0;

  printf("Input x=");
  scanf("%d",&x);

  if (x > 0) {
    if (x%2 != 0) {
      printf("x=%d is odd\n",x);
    }
    else {
      printf("x=%d is even\n",x);
    }
  }
  else {
    printf("Error: Input has to be positive!\n");
  }
}
```

- 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

```
#include <stdio.h>

main() {
  double x1 = 0;
  double x2 = 0;
  double tmp = 0;

  printf("Unsorted input:\n");
  printf(" x1=");
  scanf("%lf",&x1);
  printf(" x2=");
  scanf("%lf",&x2);

  if (x1 > x2) {
    tmp = x1;
    x1 = x2;
    x2 = tmp;
  }

  printf("Output sorted in ascending order:\n");
  printf(" x1=%f\n",x1);
  printf(" x2=%f\n",x2);
}
```

- 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?

```
#include <stdio.h>

main() {
  double r = 0;
  double x1 = 0;
  double x2 = 0;
  double z1 = 0;
  double z2 = 0;
  double dist2 = 0;

  printf("Radius of the circle r=");
  scanf("%lf",&r);
  printf("Center of the circle x = (x1,x2)\n");
  printf(" x1=");
  scanf("%lf",&x1);
  printf(" x2=");
  scanf("%lf",&x2);
  printf("Point in the plane z = (z1,z2)\n");
  printf(" z1=");
  scanf("%lf",&z1);
  printf(" z2=");
  scanf("%lf",&z2);

  dist2 = (x1-z1)*(x1-z1) + (x2-z2)*(x2-z2);
  if ( dist2 < r*r ) {
      printf("z lies inside the circle\n");
  }
  else {
    if ( dist2 > r*r ) {
      printf("z lies outside of the circle\n");
    }
    else {
      printf("z lies on the boundary of the circle\n");
    }
  }
}
```

# 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 memory location is allocated 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 $\leq$ lifetime

# Global & local variables

- Global variables = variables with global lifetime
  - Exist until the end of the program
  - Have possibly 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
- 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)