Introduction to Scientific Programming

Olaf Ippisch
email: Olaf.Ippisch@ipvs.uni-stuttgart.de

30. Juli 2009

Inhaltsverzeichnis

1 Introduction 4
1.1 Subject of the Lecture 4

2 Repetition 13
2.1 Constants 19

3 Number Representation in Computers 20

4 Repetition 22

5 Real Numbers 22

6 Round-off Errors 26

7 Conditioned Execution 26

8 Blocks 29

9 Functions 29
9.1 Mathematical Functions of the Standard Library 29
9.2 Function Arguments 31

10 Exercises 32

11 Functions 35
11.1 Local Variables 35
11.2 Call by Value and Call by Reference 36
11.3 Function Overloading 38

12 Loops 40

13 Loops 46

14 Formatted IO 47

15 Comments 48

16 Runtime Measurement 49

17 Exercises 50

18 Arrays in C++ 58

19 Solution of Linear Equation Systems 62

20 Self-defined Variable Types 65

21 Advantages of object-oriented programming 69

22 Object-oriented programming in C++ 71

23 Classes 78

24 Direct Solution of Linear Equation Systems 80

25 Tridiagonal Matrices 81

26 Default Methods 85

27 Constant Objects 86

28 Operators 87

29 Example Improved Matrix Class 90

30 Preprocessor 96

31 Inheritance 97

32 Inheritance 98

33 Virtual Functions 100

34 Interface Base Classes 102

35 Interpolation 102
35.1 Interpolation with Polynomials 102
35.2 Example 105

36 Namespaces 108

37 Makefiles 109

38 Streams 110

39 Homework 115

40 Homework 116

41 Generic Programming 117
41.1 Templates 117
41.2 Non-Numerical `MatrixClass` with Templates 119
41.3 Derived `NumMatrixClass` with Templates 120
41.4 Application of `MatrixClass` with Templates 122
41.5 The Standard Template Library (STL) 123

42 Résumé 128

1 Introduction

1.1 Subject of the Lecture

Intention of the Lecture

- Other lectures cover theoretical or applied aspects of modeling and simulation (equations, material properties, mathematical aspects of partial differential equations, numerical methods)
- In this lecture we will learn how to solve numerical problems with our own programs
- To realize this we will need knowledge on
  - computer science
  - programming (C++)
  - numerical algorithms
- In a second lecture (Advanced Scientific Programming) this knowledge is used to implement a (parallel) solver for a real problem (ground water flow, linear elasticity)

Aims

- Learn an industry standard programming language
- Learn modern programming techniques
- Get a better understanding for the solution of numerical problems with a computer and the limitations
- Get an insight in the operation of simulation programs

Prerequisites

- Basic knowledge of numerical mathematics
- Readiness to do the programming exercises

Topics

- Computer Science Basics
- Programming
  - Structure of C++ Programs
  - Variables
  - Input/Output
  - Control Structures (Conditional Execution/Loops)
  - Functions
  - Arrays
  - Containers

  - Classes
  - Inheritance
  - Interfaces and Abstract Classes
  - Generic Programming

Topics (ctd.)

- Numerics
  - Floating Point Numbers
  - Interpolation
  - Numerical Differentiation
  - Numerical Integration
  - Solution of Linear Equation Systems

Practical Exercises and Exam

- Programming exercises each week (solution has to be presented by a student in the next lecture)
- Three homeworks (have to be handed in)
- 50 percent of the obtainable points necessary as a precondition for the final test
- Written examination at the end of the Course

Course Homepage

The address of the course homepage is

`http://www.ipvs.uni-stuttgart.de/abteilungen/sgs/lehre/lehrveranstaltungen/vorlesungen/WS0809/commasc6/en`

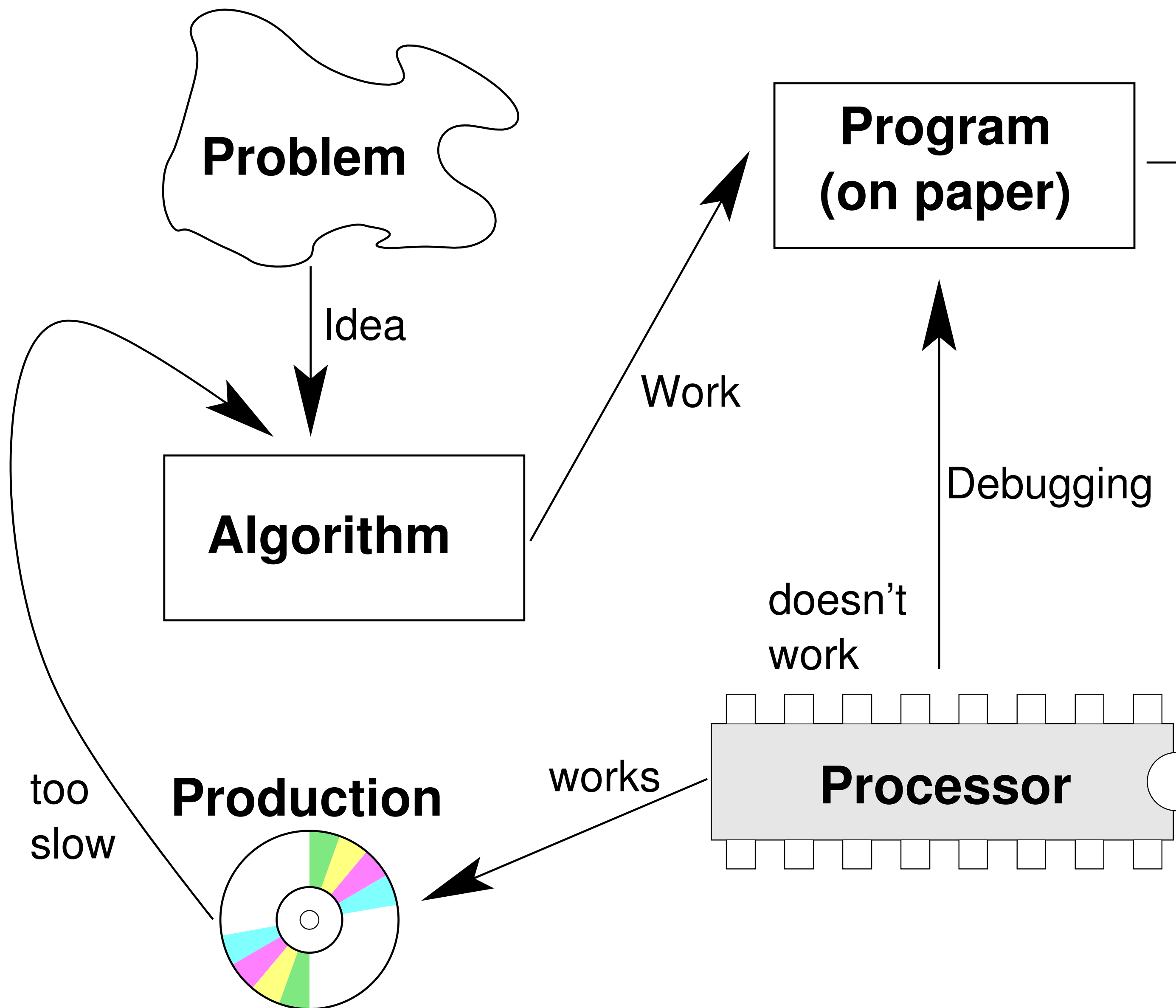
You will find there:

- the transparencies shown in the lectures (after the lecture)
- the exercises and homeworks
- links to additional material and a list of suggested books
- instructions how to get a C++ compiler

COMMAS computer lab

- The COMMAS computer lab is located in room 2.166 (just the next door)
- The available computers are running Linux (OpenSUSE)
- There is also a printer available
- Linux is the default operating system for all COMMAS lectures
- All the software necessary to do the exercises is already installed

How to solve a Problem with a Computer?



Algorithm

- is a recipe for the solution of a problem
- gives a sequence of steps to reach the target
- has to be
  - precise
  - non-ambiguous
  - complete

  - finite and composed of finite operations
  - correct

- Example: Sort the names of the Commas-students

Sort Names of COMMAS-Students

Soyer, Ahmet Mert Raina, Arun Olivares Garcia, Cesar Fernando Gorne Zoppi, Christofer Elias Tunuguntala, Deepak Halit, Demir Naik, Dhavalnitinbabu Ramasamy, Ellankavi Rajabali, Fatemeh Gültekin, Furkan Al-Tameemi, Hamza A. Hussain Shah, Kaushal Aram, Maedeh Salman, Marwan A. Khalifa, Mohamed Abdel Salam, Mohamed Rani Rahman, Mohammad Tanvir Ahamed, Mohi Uddin Mahdi, Mottahedi Tkachuk, Mykola Pillai, Rachana Roshan, Rakesh Narizhnyy, Roman Kulkarni, Romit Ashok Sridhar, Sabari Nathan Patil, Sandeep Parasharam Ravipati, Satya Krishna Alizadeh Sabet, Sepideh Janwe, Snehal Pradeep Spreng, Stefan Vallabhuni, Tarun Dhanpal, Yogesh

What is a good Algorithm

Generality Can the algorithm be applied to a wide variety of cases (e.g. an integration algorithm can integrate smooth and non-smooth functions or scalar and vector functions)?

Complexity How does the numerical effort scale with the size N of the problem (e.g. the number of names to be sorted)? Sorting can be done e.g. by

- selection sort (minimal amount of copy operations, no additional memory needed, but does not scale well, is $O(N^2)$).
- merge sort (needs twice the amount of memory, but is $O(N \cdot \mathsf{ld}(N))$ which is optimal)

Program

- A program is used to describe an algorithm in a form, which can be understood by a computer

What is the Computers Language?

- The Central Processing Units of computers are complex arrays of switches
- The machine language therefore consists of sequences of 0 and 1 (for 0 for off and 1 for on)
- One digit of a machine code instruction is called `bit`
- Several bits form an instruction word, which can be followed by one or more arguments
- The CPU is capable of doing integer and floating point calculations and conditional execution. It can fetch instructions and values from the memory and write them to the memory
- To facilitate program development assembler languages are used, which replace the machine code instruction sequences of 0 and 1 with instruction words (e.g. `mov, add`)
- Each processor or processor family has its own machine language

Program

- A program is used to describe an algorithm in a form, which can be understood by a computer
- There is a huge variety of different programming languages due to historical and functional reasons.

  They differ in

  - time necessary for the development of a program
  - time necessary for the translation of the program into machine language
  - execution speed of the final programs
  - readability of the code
  - amount of error checking done by the programming environment
  - portability
  - extensibility

High-level and low-level Programming Languages

There are high-level and low-level programming languages.

- Low-level languages are
  - very close to the machine language of a processor
  - fast to translate and execute
  - hard to read
  - hard to port to other processor architectures
- High-level languages are
  - closer to human language
  - easier to read and maintain
  - better portable
  - often slower than low-level languages

Faculty

in x86-Assembler

```
.globl factorial
factorial:
        movl $1, %eax
        jmp .L2
.L3:
        imull 4(%esp), %eax
        decl 4(%esp)
.L2:
        cmpl $1, 4(%esp)
        jg .L3
        ret
```

in C++

```
int factorial (int n)
{
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

Compiler Languages and Interpreter Languages

Programming languages either use a compiler or an interpreter to translate the program text into machine language.

- An interpreter translates the program text command for command into machine language during each execution.

  This makes program execution slower, but the same program can easily be executed on different architectures, it can be executed immediately after it is written and can be changed during execution

- A compiler translates the whole program text once into a binary executable file.

  This only has to be done once. The binary executable can only be used on one architecture. It executes faster, but during program development one always has to wait until the compiler has finished before the program can be tested.

Faculty

in Python content of program file faculty.py

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)

print factorial(10)
```

in C++ content of program file faculty.cc

```
#include <iostream>

int factorial (int n)
{
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}

int main ()
{
    std::cout << factorial(10);
    std::cout << std::endl;
}
```

Execution:

```
python faculty.py

g++ -o faculty faculty.cc
./faculty
```

Complexity of Programs

With the speed of computers and the amount of available storage also the size of programs increased:

Time	Proc.	Freq. [MHz]	RAM	Disc	Linux Kernel
1982	Z80	6	64KB	800KB	6KB (CPM)
1988	80286	10	1MB	20MB	20KB (DOS)
1992	80486	25	20MB	160MB	140KB (0.95)
1995	PII	100	128MB	2GB	2.4MB (1.3.0)
1999	PII	400	512MB	10GB	13.2MB (2.3.0)
2001	PIII	850	512MB	32GB	23.2MB (2.4.0)
2007	Core2 Duo	2660	1024MB	320GB	302MB (2.6.20)

C++

In this lecture we use the programming language C++, which is

- was developed by Bjarne Stroustrup in 1999, based on C (developed in the 1960s by Kernighan & Ritchie)
- an industry standard (ISO/IEC 14882:1998, new version ISO/IEC 14882:2003)
- a high-level programming language, but also allows to write code which is close to machine language
- a compiled language (wide variety of very good compilers available producing fast executables)
- allows object-oriented programming
- is supported on all architectures
- many libraries exist, which enhance the language
- influenced the development of Java, C#, D, ...

C and C++ are used in many software project and also many operating systems are written using C or C++.

The first C++ Program

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

The C++ Standard Library

The `#include <iostream>` command enables the compiler to use the `iostream`-part of the standard library which provides the input and output on the screen.

The Standard Library is part of the C++ ISO-standard. It provides

- input/output on the screen and on files
- mathematical functions

- complex numbers
- strings
- containers to efficiently store information
- error handling
- management of dynamic memory
- ...

The main-function

- If a C++ program is started, the `main` function is called first.
- Every C++ program has to provide a `main` function else the compilation will fail.
- The keyword `int` specifies that the function `main` returns an integer value. We also say that the function `main` is of type integer. The function `main` always has to have the return type `int`
- The commands which are executed when `main` is called are enclosed between curly brackets `{ }`
- The last command in `main` should be a `return` statement returning 0 if everything worked fine or something else if the program terminates due to errors
- Each command in C++ is terminated by a semicolon

Output on Streams

```
std::cout << "Hello␣World!"<< std::endl;
```

- In C++ output is written on an output device with the operator `<<`
- `std::cout` is the standard output device, which is directed to the screen.
- The values written to the output device can be characters, strings and numerical values
- If a numerical expression is written after `<<` only the result is written to the output device e.g.

  ```
  std::cout << (2*3)+4;
  ```

  writes `10` to the screen)
- If `std::endl` is written to a stream, a line feed is written

The gcc/g++ Compiler

The `g++` compiler is

- an open-source project developed by programmers all over the world
- available for free
- available for numerous operating systems and architectures
- has a lot of good optimization
- implements the whole C++ standard

There is a lot of useful tools available, the debugger `gdb`, the profiler `gprof`, ...

Writting a C++ program

1. Write the program file with your favorite text editor e.g. kate, emacs, vim, ...
2. Store the file on disk (e.g. in `hello_world.cc`). C++ programs usually have the extension `.cc`, `.cxx` or `.cpp`. Files which contain program code are called source file.
3. If not already available, open a terminal.
4. Compile the file with

   ```
   g++ -o <executable name> <source file name>
   ```

   The result of the compilation which can be run is called executable. The name of the executable is given after `-o` If now name is given the name `a.out` is used. E.g.

   ```
   g++ -o hello_world hello_world.cc
   ```

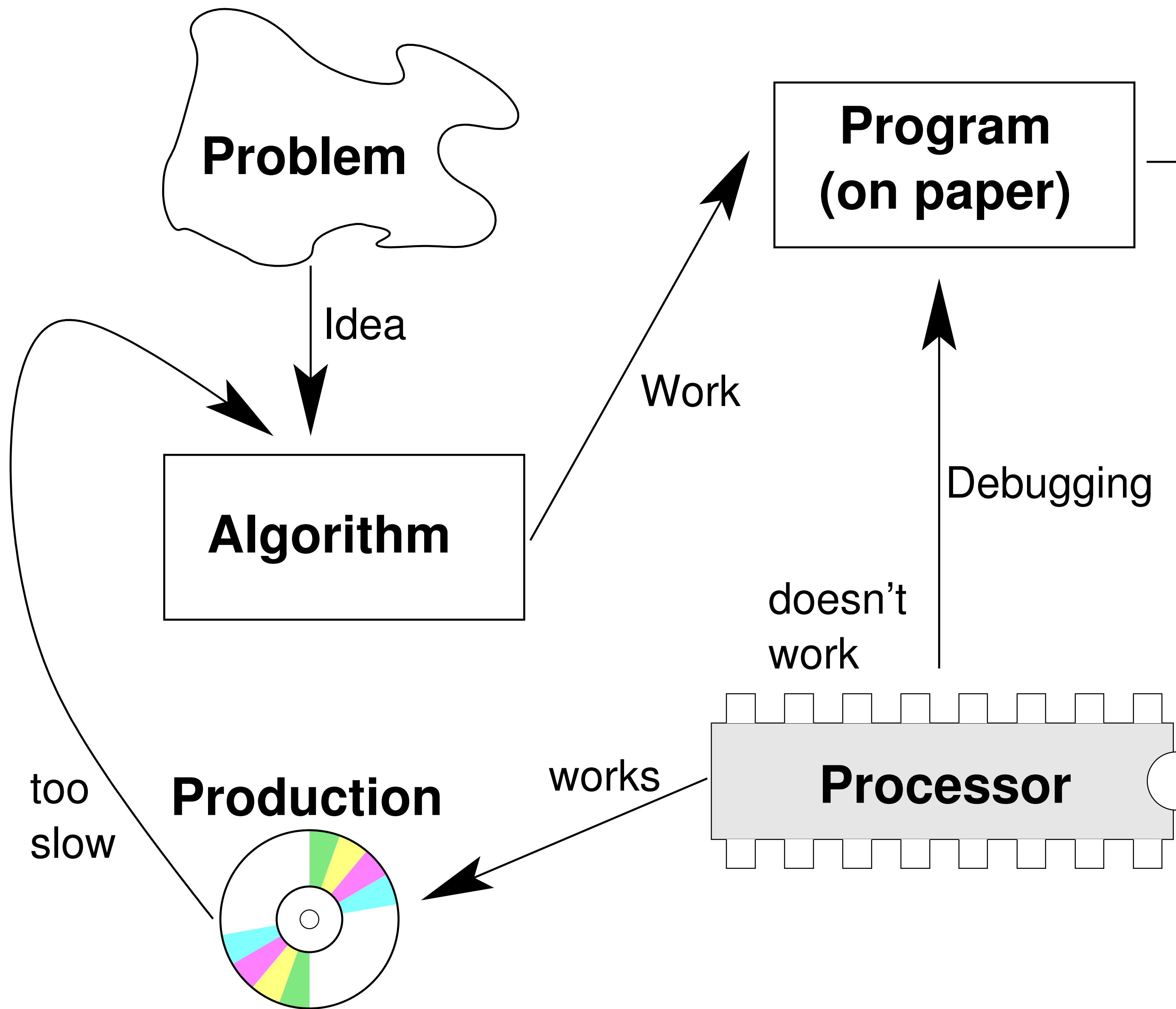
5. Run the executable by typing the name of the executable, e.g.

   ```
   hello_world
   ```

Exercises

1. Install `g++` on your computer
2. Write a program which writes `Hello world, NAME!` on the screen, where NAME is your first name
3. Write a program which writes the result of $(18/3) \cdot 7$ to the screen
4. Compile and run the programs

How to solve a Problem with a Computer?



Algorithm

- is a recipe for the solution of a problem
- gives a sequence of steps to reach the target
- has to be
  - precise
  - non-ambiguous

  - complete
  - finite and composed of finite operations
  - correct

- Example: Sort the names of the Commas-students

Program

- A program is used to describe an algorithm in a form, which can be understood by a computer

C++

In this lecture we use the programming language C++, which is

- was developed by Bjarne Stroustrup in 1999, based on C (developed in the 1960s by Kernighan & Ritchie)
- an industry standard (ISO/IEC 14882:1998, new version ISO/IEC 14882:2003)
- a high-level programming language, but also allows to write code which is close to machine language
- a compiled language (wide variety of very good compilers available producing fast executables)
- allows object-oriented programming
- is supported on all architectures
- many libraries exist, which enhance the language
- influenced the development of Java, C#, D, ...

C and C++ are used in many software project and also many operating systems are written using C or C++.

Output on Streams

```
std::cout << "Hello␣World!"<< std::endl;
```

- In C++ output is written on an output device with the operator `<<`
- `std::cout` is the standard output device, which is directed to the screen.
- The values written to the output device can be characters, strings and numerical values
- If a numerical expression is written after `<<` only the result is written to the output device e.g.

  ```
  std::cout << (2*3)+4;
  ```

  writes `10` to the screen)
- If `std::endl` is written to a stream, a line feed is written

Operators

- Like in ordinary mathematics operators connect two values
- Operators are `+ - * /`. For integer values there is also the operator `\%` (called "modulo")
- For integer values, the division operator `/` calculates the integer part of the result, the modulo operator `\%` calculates the rest
- There is a given order of execution called precedence.
  - The operators `* / \%` are evaluated first
  - Then the operators `+ -` are evaluated
  - If there is more than one operator with the same precedence, the evaluation is done from left to right
  - The precedence can be altered with brackets

Operator Precedence Example

- Without Brackets

```
10 % 6 / 2 + 7 * 2 - 3 =
   4   / 2 +   14  - 3 =
      2    +   14  - 3 =
          16       - 3 = 13
```

- With Brackets

```
10 % (6 / 2) + 7 * (2 - 3) =
10 %    3    + 7 *  (-1)   =
      1      -      7      = -6
```

Variables

- Variables are named locations to store values
- In C++ every variable has to be defined before it is used. This is done by stating its type and name
- There are several types of variables:

  Integer Numbers numerical values without fractional portion, e.g. `1, 98, -112`
  Real Numbers numerical values with (possible) fractional portion, e.g. `-8.1, 3e6, 4.0`
  Chars single letters, e.g. `'a', 'Z', '0', '?', '+'`
  Strings a multitude of letters (a word, a sentence, a paragraph), e.g. `"example", "this␣is␣also␣a␣strin`
  Boolean Values variables that either have the value `true` or `false`

Variable Names

Any variable name is allowed that

- starts with a letter (lower or upper case)
- may contain small and capital letters, numbers and underscores
- is not one of the keywords used in C++

```
asm auto bool break case catch char class
const const_cast continue default delete do
double dynamic_cast else enum explicit export
extern false float for friend goto if inline
int long mutable namespace new operator private
protected public register reinterpret_cast return
short signed sizeof static static_cast struct
switch template this throw true try typedef
typeid typename union unsigned using virtual
void volatile wchar_t while
```

C++ is case sensitive `name` and `Name` are two different variables!

Definition of Variables

A variable is defined with:

```
type variableName;
```

More than one variables of the same type can be defined at once with:

```
type variableName1, variableName2;
```

Example definition

```
int var1;
float var2;
char var3;
std::string var4;
bool var5, var6;
```

Initialization of and Assignment to Variables

A variable is given a value with the operator = If a variable is given a value

- directly at its definition this is called initialization
- Sometimes afterwards this is called assignment.

Example:

```
int var1=5;
int var2;
float var3, var4;
char var5;
std::string var6;
bool var7 = false;
var2 = 35;
var3=8.3;
var4=2.1E-3;
var5='R';
var6="example string";
```

Better use speaking Names!!!

Improved Example

```
int numElements=5;
int age;
float pi, speed;
char initial;
std::string name;
bool error = false;
age = 35;
pi=8.3;
speed=2.1E-3;
initial='R';
name="Olaf Ippisch";
```

Don't forget to Initialize your Variables!

- Variables do not have a default value in C++ to optimize the performance.
- As long as no value has been given to a variable it contains some arbitrary value.
- Be careful always to initialize a variable before you use it.

```
#include <iostream>

int main()
{
    int a;
    std::cout << a << std::endl;
}
```

delivered e.g. `-1209042944`, `-1208760320`, `-1208809472` and `-1208608768` when it was called four times

Conversion between Integer and Real Variables

- If integer values are assigned to real variables the conversion is done automatically without warning

- If real values are assigned to integer variables the compiler issues a warning. The fractional portion is dropped during the assignment.
- The conversion can be done explicitly by writing

  ```
  float a = 3.1;
  int b = int(a);
  int c = (int)a;
  a = (float)b;
  a = float(c);
  ```

  The compiler warning then knows that this conversion is intended and suppresses the warning.

String Variables

- The data type for strings is `std::string`
- Strings can be added
- `char`-variables can be added to strings. e.g.

  ```
  std::string name = "Olaf";
  name = name + '␣' + "Ippisch";
  ```

- However, the value at the left hand side of the `"+"` operator has allways to be a variable of type `std::string`. The following code is *not* working:

  ```
  std::string name = "Olaf" + '␣' + "Ippisch";
  ```

  Neither is this:

  ```
  std::string name = "Olaf";
  name = "Ippisch" + ',' + name;
  ```

- You can use explicit type conversion:

  ```
  std::string name = "Olaf";
  name = std::string("Ippisch") + ',' + name;
  ```

Input from Streams

```
int main()
{
    std::cout << "Please␣enter␣your␣age:␣";
    int age;
    std::cin >> age;
    std::cout << "You␣are␣" << age << "␣years␣old" << std::endl;
}
```

- In C++ input is read from an input device with the operator `>>`
- `std::cin` is the standard input device, which is directed to the keyboard.

- The values read from the input device can be characters, strings and numerical values
- Input is terminated when the `<return>`-key is pressed
- If more values are entered than read, only the necessary values are used, the rest is stored
- It is not possible to read from `std::cout` or write to `std::cin`

Input from Streams

```
#include <iostream>

int main()
{
    std::cout << "Please␣enter␣your␣age:␣";
    int age;
    std::cin >> age;
    std::cout << "You␣are␣" << age << "␣years␣old" << std::endl;
    std::cout << "Please␣enter␣your␣name:␣";
    std::string name;
    std::cin >> name;
    std::cout << "Your␣name␣is␣" << name << std::endl;
}
```

Common Pitfalls with input

- If you give more arguments than necessary, the rest is stored and used for later input

```
Please enter your age: 37 42
You are 37 years old
Please enter your name: Your name is 42
```

- Even for strings everything after the first space or tab is ignored

```
Please enter your age: 37
You are 37 years old
Please enter your name: Olaf Ippisch
Your name is Olaf
```

- If we change the order this gets even worse:

```
Please enter your name: Olaf Ippisch
Your name is Olaf
Please enter your age: You are -1078521160 years old
```

2.1 Constants

Constants

A constant value is defined with:

```
const type variableName = value;
```

The value of a constant can not be altered after the definition.

Examples:

```
const int NUM_POINTS = 10;
const double PI = 3.14159265358979323846;
```

Constants have several advantages:

- It is guaranteed that their value does not change
- If a constant value is needed more than once, it can be changed more easily if it is a constant variable and not a constant number
- If the compiler knows that a variable is constant it can perform special optimizations
- The compiler can check the type of constants

3 Number Representation in Computers

Integer Variables

- Integer numbers are stored as binary numbers in a computer

$$\begin{aligned} 11001101 &= 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \\ &= 128 + 64 + 0 + 0 + 8 + 4 + 0 + 1 = 205 \end{aligned}$$

- for signed integers the bit at the front is used for the sign.
- negative numbers are stored as "Two's Complement" (for details see the wikipedia article)
- The range of numbers, which can be stored in a computer variable is limited by the amount of memory used for the variable
- As the zero is coded as a positive number, the range for negative numbers is by one larger: e.g. with 1 byte the range is $[-2^7 : 2^7 - 1] = [-128 : 127]$ and with two bytes $[-2^{15} : 2^{15} - 1] = [-32768 : 32767]$
- There are also unsigned variables, where the sign bit can be used to store larger numbers e.g. with 1 byte the range is then $[0 : 2^8 - 1] = [0 : 255]$ and with two bytes $[0 : 2^{16} - 1] = [0 : 65536]$

Range of Integer Variables

Data Type	Size	Range
`short`	2 bytes	$[-32768 : 32767]$
`int`	4 bytes	$[-2147483648 : 2147483647]$
`long`	4 bytes	[−2147483648L:2147483647L]
`long` (64-bit OS), `long long`	8 bytes	[−9223372036854775807L: 9223372036854775807LL]
`unsigned short`	2 bytes	[0U:65536U]
`unsigned int`	4 bytes	[0U:4294967295U]
`unsigned long` (32-bit)	4 bytes	[0 : 4294967295UL]
`unsigned long` (64-bit OS) `unsigned long long`	8 bytes	[0 : 18446744073709551615ULL]

Solution of the Exercises

1. Write a program which writes `Hello world, NAME!` on the screen, where NAME is your first name

```
#include <iostream>

int main()
{
    std::cout << "Hello␣world,␣Olaf!" << std::endl;
    return(0);
}
```

2. Write a program which writes the result of $(18/3) \cdot 7$ to the screen

```
#include <iostream>

int main()
{
    std::cout << (18/3)*7 << std::endl;
    return(0);
}
```

New Exercises

1. Write a program which prompts you for your first name, your family name and your age and writes to the screen something like

```
Your name is Olaf Ippisch.
You are 37 years old.
```

2. Modify the program to additionally calculate to how many month, days, hours, minutes and seconds your age corresponds (you can neglect leap years). The output should be something like

```
Your name is Olaf Ippisch.
You are 37 years old.
This corresponds
to 444 month
or 13505 days
or 324120 hours
or 19447200 minutes
or 1166832000 seconds
```

New Exercises

3. Which integer variable type would you need to store the amount of seconds if the maximal age you expect is 100 years?

4. Write a Programm which assigns 10^{10} to an integer variable of type `int` and writes the content of the variable to the screen. Add 1 to the variable and write the result to the screen as well What do you get? Why?

4 Repetition

5 Real Numbers

Real Numbers

The size of numbers occurring in science is very different:

- Plank constant: $6.6260693 \cdot 10^{-34}$ J s
- Rest mass of an electron: $9.11 \cdot 10^{-31}$ kg
- Avogadro constant: $6.021415 \cdot 10^{23}$ mol$^-1$

As shown here, they are therefore often written as value times order of magnitude.

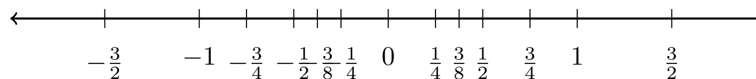
Real Numbers

- On a computer real numbers are stored in the form $x = m \cdot 2^e$ where m is the so called mantissa and e is the exponent.
- 2 is the natural basis for a digital computer
- For the mantissa we use $m = \pm \sum_{i=1}^{r} m_i 2^{-i}$, for the exponent $e = \pm \sum_{i=1}^{s} e_i 2^i$
- For the mantissa the most forward bit always has to be one (normalization)

Real Numbers

- Not any value can be represented with such a "floating point value"
- The precision of the floating point number is controlled by the number of bits r used for the mantissa
- The range of the floating point number is controlled by the number of bits s of the exponent

Example: $r = 2$, $s = 1$



Possible values of m: $\pm\frac{1}{2}, \pm\frac{3}{4}$
Possible values of e: $1, 0, -1$
Possible values of x:
$\pm\frac{1}{2} \cdot 2^{-1}, \pm\frac{1}{2} \cdot 2^0, \pm\frac{1}{2} \cdot 2^1, \pm\frac{3}{4} \cdot 2^{-1}, \pm\frac{3}{4} \cdot 2^0, \pm\frac{3}{4} \cdot 2^1, 0$

IEEE-754 Standard

defines `float` (single precision), `double` and `long double` (double-extend)

Details for `double`

- Total size: 64 bits
- Exponent: 11 bits
- Mantissa: 53 bits (1 bit sign, 52 bits for values)
- As the most forward bit is always one (standardization) and $x = 0$ is coded in a special way, the first bit is not stored
- Exponent is a bit complicated
  - 11 bits give a range of $c \in [0 : 2047]$
  - $c = 0$ is used to signal that $x = 0$ (the whole floating point value is zero, not only the exponent)
  - $c = 2047$ is used to signal that the result of a computation is not a number (NaN), which can occur e.g. if a value is divided by zero
  - The actual exponent is calculated from $e = c - 1022$ for $c \in [1 : 2046]$, which gives a range of $e \in [-1021 : 1024]$

Machine Precision

- As not every value can be stored as floating point number, numbers are rounded to the nearest floating point number
- The relative error due to rounding can be estimated to be smaller than

  $$\left| \frac{x - \mathsf{rd}(x)}{x} \right| \leq 2^{-r}$$

  This is also called machine precision *eps*. For a `double` value according to the IEEE 754-standard with $r = 53$ this would mean a machine precision of $eps = 2^{-53} = 1.11 \cdot 10^{-16}$
- Even if x and y are floating point values, the result of a floating point calculation is not necessarily also a floating point value. The IEEE 754-standard defines that for the operators $+ - */\sqrt{x}$ the result is first calculated exactly and then rounded, so the error of the result is also smaller the *eps*.

Range of Real Variables

Data Type	Size	Range	Precision
`float`	4 bytes	$[2^{-125} : 2^{128}] \approx \pm[10^{-38} : 10^{38}]$	7 digits
`double`	8 bytes	$[2^{-1021} : 2^{1024}] \approx \pm[10^{-308} : 10^{308}]$	15 digits
`long double`	10 bytes	$[2^{-16381} : 2^{16384}] \approx \pm[10^{-4932} : 10^{4932}]$	18 digits

Consequences

The result of floating point operations is often not exact

- If a very small number is added to a large one, the result is equal to the larger number if the difference in the order of magnitude of the two numbers is larger than the machine precision.

  with `float`-precision

  ```
  1.129873e3 + 1e-5 = 1.129873e3
  ```

- If a difference between two nearly identical numbers is calculated, the result has only very few digits different from zero (we also call this *significant* digits). This phenomenon is called *extinction*.

  with `float`-precision

  ```
  1.129873e3 - 1.129871e3 = 2.000000e-3
  ```

Numerical algorithms have to be formulated carefully to take this into account!

Power Series for e^x

e^x can be calculated with a power series

$$e^x = 1 + \sum_{n=1}^{\infty} \frac{x^n}{n!} = 1 + \sum_{n=1}^{\infty} y_n.$$

We start to calculate approximate solutions $S_i(x)$ with

$$\begin{aligned} y_1 &= x \\ S_1 &= 1 + y_1 \end{aligned}$$

and calculate $n = 2, 3, \ldots$ with

$$\begin{aligned} y_n &= \frac{x}{n} y_{n-1} \\ S_n &= S_{n-1} + y_n. \end{aligned}$$

Power Series for e^x, positive x

for $x = 1$ and `float`-precision the first 7 digits are correct

```
  #                  S_n                    y_n
  1     2.000000000000000e+00    1.000000000000000e+00
  2     2.500000000000000e+00    5.000000000000000e-01
  3     2.666666746139526e+00    1.666666716337204e-01
  4     2.708333492279053e+00    4.166666790843010e-02
  5     2.716666936874390e+00    8.333333767950535e-03
  6     2.718055725097656e+00    1.388888922519982e-03
  7     2.718254089355469e+00    1.984127011382952e-04
  8     2.718278884887695e+00    2.480158764228690e-05
  9     2.718281745910645e+00    2.755731884462875e-06
 10     2.718281984329224e+00    2.755731998149713e-07
...
100     2.718281984329224e+00    0.000000000000000e+00
 ex     2.718281828459045E0
```

for $x = 5$ and `float`-precision we also get 7 significant digits

```
...
 21      1.484131774902344e+02      9.333108209830243e-06
  ex     1.484131591025766E2
```

Power Series for e^x, negative x

for $x = -1$ and `float`-precision we get only 6 significant digits

```
   #                      S_n                        y_n
...
  10      3.678794205188751e-01      2.755731998149713e-07
  11      3.678793907165527e-01     -2.505210972003624e-08
  12      3.678793907165527e-01      2.087675810003020e-09
  ex      3.678794411714423E-1
```

for $x = -5$ we get 4 significant digits

```
   1     -4.000000000000000e+00     -5.000000000000000e+00
   2      8.500000000000000e+00      1.250000000000000e+01
   3     -1.233333396911621e+01     -2.083333396911621e+01
   4      1.370833396911621e+01      2.604166793823242e+01
...
  15      1.118892803788185e-03     -2.333729527890682e-02
  16      8.411797694861889e-03      7.292904891073704e-03
...
  28      6.737461313605309e-03      1.221854423194557e-10
...
 100      6.737461313605309e-03      0.000000000000000e+00
  ex      6.737946999085467E-3
```

Power Series for e^x, $x = -20$

For $x = -20$ and `float`-precision there is no convergence at all and the result is wrong by eight orders of magnitude

```
   #                      S_n                        y_n
   1     -1.900000000000000e+01     -2.000000000000000e+01
   2      1.810000000000000e+02      2.000000000000000e+02
   3     -1.152333374023438e+03     -1.333333374023438e+03
   4      5.514333496093750e+03      6.666666992187500e+03
   5     -2.115233398437500e+04     -2.666666796875000e+04
...
  31     -1.011914250000000e+06     -2.611609750000000e+06
  32      6.203418750000000e+05      1.632256125000000e+06
  33     -3.689042500000000e+05     -9.892461250000000e+05
  34      2.130052500000000e+05      5.819095000000000e+05
  35     -1.195144687500000e+05     -3.325197187500000e+05
  36      6.521870312500000e+04      1.847331718750000e+05
...
  65      7.566840052604675e-01     -4.473213550681976e-07
  66      7.566841244697571e-01      1.355519287926654e-07
  67      7.566840648651123e-01     -4.046326296247571e-08
  68      7.566840648651123e-01      1.190095932912527e-08
  ex      2.061153622438557E-9
```

Power Series for e^x, $x = -20$, higher Precision

For $x = -20$ and `double`-precision the result is still wrong by a factor of three

```
   #                      S_n                        y_n
...
  27     -5.180694836889297e+06     -1.232613988175268e+07
  28      3.623690792934047e+06      8.804385629823344e+06
...
```

```
  94      6.147561828914626e-09      1.821561256740375e-24
  95      6.147561828914626e-09     -3.834865803663947e-25
  ex      2.061153622438557E-9
```

Only with quad-precision (there are special libraries for this) we get 15 significant digits (with approx. 30 digits machine precision)

```
   #                         S_n                              y_n
...
 117 2.0611536224385583392700458752947E-9 -4.1852929339382073650363741579941E-41
 118 2.0611536224385583392700458752947E-9 7.0937168371834023136209731491427E-42
  ex 2.0611536224385578279659403801558E-9
```

6 Round-off Errors

Round-off Errors

- A value with fraction $\hat{x}$ can not always be converted exactly to a floating-point number x. There is an average round-off error $|\hat{x} - x| = \epsilon$ (machine precision)
- The result of a calculation is not exact, but exactly rounded. This means that e.g. $x_1 \odot x_2 = \mathrm{rd}(x_1 \cdot x_2) = (x_1 \cdot x_2)(1 + \epsilon_*)$ with $|\epsilon_*| \leq \epsilon$
- While the result of $(x_1^2 - x_2^2)$ and $(x_1 - x_2) \cdot (x_1 + x_2)$ are equal in exact calculations, they are not necessarily equal if calculated with a computer.

7 Conditioned Execution

Conditioned execution

It is often necessary to have code, which is only executed if a certain condition is true or false

- The keyword `if` is followed by a condition and a statement. The statement is only executed if the condition is true. The condition has to be in brackets.

```
if (condition)
    do_something;
```

Use indentation to make your programs more readable!

Conditioned execution (2)

- The statement can be either nothing, one command or a block. A block is a sequence of commands included between curly brackets.

```
if (condition);
if (condition)
    do_something;
if (condition)
{
    do_something;
    and_do_even_more;
}
```

- An alternative statement can be specified after the keyword `else`, which is executed if the condition is not true.

```
if (condition)
    do_something;
else
    do_something_else;
```

Chained and Nested Conditioned execution

- Conditions can be chained

```
if (condition1)
    do_something;
else if (condition2)
    do_something_if_second_condition_true;
else
    do_something_else;
```

- or nested

```
if (condition1)
    do_something;
else
{
    if (condition2)
        do_something_if_second_condition_true;
    else
        do_something_else;
}
```

Conditioned execution example

```
#include <iostream>

int main()
{
    int number;
    std::cout << "Please␣enter␣a␣number:␣";
    std::cin >> number;
    if (number>0)
        std::cout << "The␣number␣is␣positive" << std::endl;
    else if (number == 0)
    {
        std::cout << "The␣number␣is␣zero";
        std::cout << std::endl;
    }
    else
        std::cout << "The␣number␣is␣negative" << std::endl;
}
```

Conditional and Boolean Operators

Possible conditional operators are

==	equals	a==b
!=	is not equal to	a!=b
>	is greater than	a>b
<	is less than	a<b
>=	is greater or equal	a>=b
<=	is less or equal	a<=b

Conditions can be combined with the operators

&&, and	and (both conditions have to be true)	(a==b)&&(a<c)
||, or	or (at least one condition has to be true)	(a==b)||(a<c)
!, not	not (the condition has to be false)	!(a==b) is identical to a!=b

It is necessary to have brackets around the combined condition statement.

Conditioned Execution example rewritten

```
#include <iostream>

int main()
{
    int number;
    std::cout << "Please␣enter␣a␣number:␣";
    std::cin >> number;
    if (not((number<0) and (number==0)))
        std::cout << "The␣number␣is␣positive" << std::endl;
    else if (number == 0)
    {
        std::cout << "The␣number␣is␣zero";
        std::cout << std::endl;
    }
    else
        std::cout << "The␣number␣is␣negative" << std::endl;
}
```

Conditioned Execution example rewritten

```
#include <iostream>

int main()
{
    int number;
    std::cout << "Please␣enter␣a␣number:␣";
    std::cin >> number;
    if (!((number<0) && (number==0)))
        std::cout << "The␣number␣is␣positive" << std::endl;
    else if (number == 0)
    {
        std::cout << "The␣number␣is␣zero";
        std::cout << std::endl;
    }
    else
        std::cout << "The␣number␣is␣negative" << std::endl;
}
```

Operator Precedence revisited

- The operator precedence including logical operators is:
    1. `!`, `not` (logical not)
    2. `* / \%`
    3. `+ -`
    4. `< <= > >=`
    5. `== !=`
    6. `&& and`
    7. `|| or`
- If there is more than one operator with the same precedence, the evaluation is done from left to right
- The precedence can be altered with brackets

8 Blocks

Blocks

- A set of statements and variable definitions enclosed in curly brackets is called a "block".
- Variables defined in a block are known only inside the block
- If a Variable exists outside the block and a new variable with the same name is defined inside a block, the variable outside is hidden by the local variable.

```
float a = 2.1;
{
    int a= 3;
    std::cout << "Inside␣the␣block␣a␣=␣" << a << std::endl;
}
std::cout << "Outside␣the␣block␣a␣=␣" << a << std::endl;
```

The output is then:

```
Inside the block a = 3
Outside the block a = 2.1
```

9 Functions

9.1 Mathematical Functions of the Standard Library

Mathematical Functions of the Standard Library

The standard library defines a lot of usefull functions. To be able to use all of the following functions you need to `#include <cmath>` The functions expect `double` arguments and the return type is also `double`

C++ name	function
`pow(x,y)`	x^y
`sin(x)`	
`cos(x)`	
`tan(x)`	
`asin(x)`	$\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$
`acos(x)`	$\cos^{-1}(x)$ in range $[0, \pi]$
`atan(x)`	$\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$
`sinh(x)`	
`cosh(x)`	
`tanh(x)`	
`exp(x)`	e^x
`log(x)`	$\ln(x)$
`sqrt(x)`	$\sqrt{x}$
`fabs(x)`	$\|x\|$
`floor(x)`	largest integer not greater than x; example: `floor(5.768)` $= 5$
`ceil(x)`	smallest integer not less than x; example: `ceil(5.768)` $= 6$
`fmod(x,y)`	floating-point remainder of x/y with the same sign as x

Using mathematical functions

```
#include<cmath>

int main()
{
    double log = log(17.0);
    double angle = 1.5;
    double height = sin(angle);
}
```

Using mathematical functions

- Function calls can be nested.
- The function arguments can be mathematical expressions.
- These expressions can again contain function calls.
- The expressions are evaluated first, the result is passed to the function.

```
#include<cmath>

int main()
{
    double pi = acos(-1.0);
    double angle = 1.5;
    double x = cos (angle + pi/2);
    x = exp(log(10.0));
}
```

Defining new Functions

```
return_type function_name(list_of_arguments)
{
    statements
}
```

- A function in C++ is a piece of code, which fullfills a certain task. It usually returns a result.
- A function is defined by stating the return type, the function name and a list of arguments in brackets and afterwards the code of the function in curly brackets (in a block).
- The rules for function names are the same as for variable names.

Function Example

```
#include <iostream>

int Faculty(int x)
{
    if (x<=1)
        return 1;
    else
        return Faculty(x-1)*x;
}

int main()
{
    std::cout << "The␣faculty␣of␣5␣is:␣";
    std::cout << Faculty(5) << std::endl;
    std::cout << "The␣faculty␣of␣10␣is:␣";
    std::cout << Faculty(10) << std::endl;
    return 0;
}
```

9.2 Function Arguments

Function Arguments

- Inside a function the variables defined outside the function are not known
- To pass values to a function they have to be given as function arguments
- If a function calls itself, the variables inside the two realisations of the function are different as well (as each function is a new block).
- A function can get an arbitrary number of arguments.
- Each argument consists of a type and a variable name.
- The arguments are separated by commas.

Example

```
int Power(int x, int exp)
{
    if (exp==1)
        return x;
    else
        return Power(x,exp-1)*x;
}
```

```
int main()
{
   int a = Power(2,2);    // a is 4.0
   int b = Power(2,3);    // b is 8.0
}
```

`void`

- Functions do not have to have arguments or return values.
- The argument or return type is then `void`.
- If there is no argument the function can also just have empty brackets.
- Functions with `void` return type correspond to the procedures of other programming languages

Default Arguments

- The last arguments of a function can have default values.
- Default arguments are usefull, if a argument is rarely used and has a reasonable default value.
- The default values are specified only once either in the declaration or the definition, whichever comes first.

```
int Power(int x, int n=2)
{
    if (n==1)
        return x;
    else
        return Power(x,n-1)*x;
}

int main()
{
   int a = Power(2);   // a is 4.0
   int b = Power(2,3); // b is 8.0
}
```

10 Exercises

Solution of the Exercises

1. Write a program which prompts you for your first name, your family name and your age and writes to the screen something like

```
Your name is Olaf Ippisch.
You are 37 years old.
```

2. Write modify the program to additionally calculate to how many month, days, hours, minutes and seconds your age corresponds (you can neglect leap years). The output should be something like

```
Your name is Olaf Ippisch.
You are 37 years old.
This corresponds
to 444 month
or 13505 days
or 324120 hours
or 19447200 minutes
or 1166832000 seconds
```

3. Which integer variable type would you need to store the amount of seconds if the maximal age you expect is 100 years?

4. Write a Programm which assigns 10^{10} to an integer variable of type `int` and writes the content of the variable to the screen. Add 1 to the variable and write the result to the screen as well What do you get? Why?

New Exercises

1. Fast Powerfunction

   While computing the power function some multiplication can be saved. For example

$$x^8 = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$$

   can be computed in the following three steps

$$\begin{aligned} x^2 &= x \cdot x \\ x^4 &= x^2 \cdot x^2 \\ x^8 &= x^4 \cdot x^4 \end{aligned}$$

   such as only 3 instead of 7 multiplication are needed.

   A more general recursive formulation of this fast computation is given in the following:

$$x^n = \begin{cases} (x^{n/2})^2 & \text{if } n \text{ even} \\ x \cdot x^{n-1} & \text{if } n \text{ odd} \\ x & \text{if } n = 1 \end{cases}$$

New Exercises

1. (ctd.)

   a) Write a function `int PowerFast(int x, int exp)` that computes the power function for exponents `exp` ≥ 1 in the fast way as presented above.

   b) Write a `main()` program to test your function. In the main routine also test if your implementation yields the same result as the `int Power(int x, int exp)` function presented in the lecture. Otherwise you have made an error :-).

New Exercises

2. Number representation
   a) The determinant of a 2×2 matrix A is calculated as

$$\det(A) = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

Write a C++ function `float determinant(float a, float b, float c, float d)` which calculates the determinant. Compute the determinant $det(A)$ of the matrix

$$A = \begin{pmatrix} 100 & 0.01 \\ -0.01 & 100 \end{pmatrix}.$$

The exact solution is 10000.0001.

Be careful! The C++ compiler automatically rounds the output of `float` and `double` values to make the result more readable. As we are interested in the non-rounded value we have to switch of the rounding.

New Exercises

2. (ctd.)
   a) (ctd.) With the following main program all digits are written to the screen:

```
int main ()
{
    cout.precision(10);
    cout << determinant(100, 0.01, -0.01, 100);
};
```

Why is the result not correct? What happens if you use the data type `double` instead of `float`?

   b) Calculate $(a+b)+c$ and $a+(b+c)$ with a C++ program for $a = 10^n$, $b = -10^n$ and $c = 10^{-n}$ and $n = 6, 7, 8, \ldots, 14$. For which n is the addition on your Computer no longer associative if you use `float`? Why? Please also use the main program from above to get all digits.

Solution of Exercise 1

```
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";
    std::string firstName;
    std::cin >> firstName;
    std::cout << "Please enter your family name: ";
    std::string familyName;
    std::cin >> familyName;
    std::cout << "Please enter your age: ";
```

```
    int age;
    std::cin >> age;
    std::cout << "Your␣name␣is␣" << firstName << "␣";
    std::cout << familyName << std::endl;
    std::cout << "You␣are␣" << age << "␣years␣old" << std::endl;
}
```

Solution of Exercise 2

```
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please␣enter␣your␣first␣name:␣";
    std::string firstName;
    std::cin >> firstName;
    std::cout << "Please␣enter␣your␣family␣name:␣";
    std::string familyName;
    std::cin >> familyName;
    std::cout << "Please␣enter␣your␣age:␣";
    long long age;
    std::cin >> age;
    std::cout << "Your␣name␣is␣" << firstName;
    std::cout << "␣" << familyName << std::endl;
    std::cout << "You␣are␣" << age << "␣years␣old" << std::endl;
    std::cout << "This␣corresponds" << std::endl
    std::cout << "to␣" << age*12 << "␣month" << std::endl;
    std::cout << "or␣" << age*365 << "␣days" << std::endl;
    std::cout << "or␣" << age*365*24 << "␣hours" << std::endl;
    std::cout << "or␣" << age*365*24*60 << "␣minutes" << std::endl;
    std::cout << "or␣" << age*365*24*3600 << "␣seconds" << std::endl;
}
```

Solution of Exercise 3

3. Which integer variable type would you need to store the amount of seconds if the maximal age you expect is 100 years?

   `unsigned int`. You can also use `long` on a 64 bit system or `long long` on a 32 bit system

4. Write a program which assigns 10^{10} to an integer variable of type `int` and writes the content of the variable to the screen. Add 1 to the variable and write the result to the screen as well What do you get? Why?

   The compiler gives a `warning: overflow in implicit constant conversion`

   The output of the program is:

   ```
   2147483647
   -2147483648
   ```

11 Functions

11.1 Local Variables

Local Variables

- Variables defined inside a function or a block are called local variables.
- Local variables are stored on the *Stack*.
- The *Stack* is a data structure, which stores information about the active subroutines of a computer program.
- For each call of a function a new set of variables is created.
- After a function terminates this variables are deleted.

11.2 Call by Value and Call by Reference

References

In C++ we can define references to variables.

- References are no new variables, they are only defining a different name for the same variable.
- A reference must have the same type as the original variable.
- The variable has to exist already
- It has to be initialized at the definition.
- A reference is fixed to one variable, it is not possible to assign a different one afterwards.
- More than one reference to a variable are allowed
- A reference can also be initialized from a reference

References Example

```
#include<iostream>

int main()
{
    int a = 12;
    int &b = a;    // defines a reference
    int &c = b;    // allowed
    float &d = a; // illegal not the same type
    int e = b;
    b = 2;
    c = a * b;
    std::cout << a << std::endl;
    std::cout << e << std::endl;
}
```

Call by Value

Variables can be passed to functions by two different ways

- If a normal variable is passed to a function, a new copy of the variable is created on the stack each time the function is called.
- This is called *Call by Value*
- If the variable is changed in the function, the original variable is not changed.

```
double SquareCopy(double x)
{
    x = x * x;
    return x;
}
```

Call by Reference

- If a reference to a variable is passed to a function only a reference to the original variable is created on the stack.
- This is called *Call by Reference*
- All changes to the variable in the function also change the original variable
- This makes it possible to write functions, which return more than one result and functions without return value (procedures).

```
void SquareRef(double &x)
{
    x = x * x;
}
```

Constant Function Arguments

Function arguments can be defined to be constant. The values can then not be modified inside the function. This is especially usefull for references. It avoids the copying without risking a modification of the original variable

```
int Square(const int x)
{
    x = x * x;    // illegal, compiler error
    return x;
}

double Square(const double &x)
{
    x = x * x;    // illegal, compiler error
    return x;
}

float Square(const float &x)
{
    return x*x;
}
```

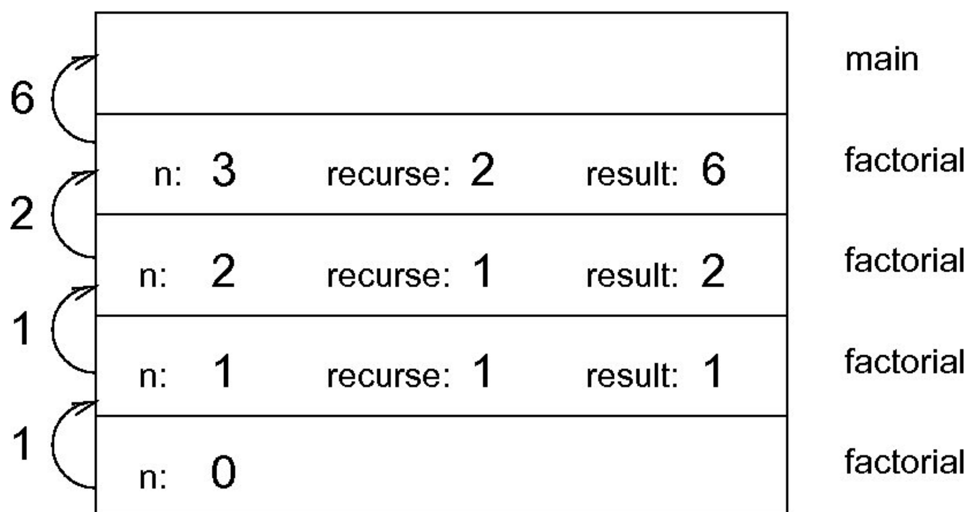
Recursive Programming

- Recursive Programming is a special technique, where a function calls itself to fullfill a certain task.
- A stop condition is necessary to avoid infinite recursion

```
#include <iostream>

int Factorial(unsigned int n)
{
    if (n==1)
        return 1;
    else
    {
        int recurse = Factorial(n-1);
        int result = n * recurse;
        return result;
    }
}

int main()
{
    std::cout << "The␣faculty␣of␣10␣is:␣" << Factorial(10) << std::endl;
    return 0;
}
```



11.3 Function Overloading

Function Overloading

Two functions may have the same name, if the arguments are different (a different return type is not enough)

```
#include <iostream>

void Print(const int i)
{
    std::cout << "␣Here␣is␣int␣" << i << std::endl;
}
```

```
void Print(const double f)
{
    std::cout << "␣Here␣is␣float␣" << f << std::endl;
}

void Print(const std::string text)
{
    std::cout << "␣Here␣is␣string␣" << text << std::endl;
}

int main()
{
    Print(10);
    Print(10.10);
    Print("ten");
}
```

Function Declaration

In C++ a function has to be at least declared before it is first used. A declaration states the type, the name and the argument list followed by a semicolon. It is enough to specify the type of each argument, a name is optional and can be different than in the definition.

```
#include <iostream>

int Faculty(int x); // this is the declaration

int main()
{
    std::cout << "The␣faculty␣of␣10␣is:␣" << Faculty(10) << std::endl;
    return 0;
}

int Faculty(int x) // this is the definition
{
    // here is the implementation
    int result=1;
    for (int i=2;i<=x;++i)
        result*=i;
    return result;
}
```

Libraries and Header Files

The declaration of functions is especially important for the development of libraries. In libraries usefull functions are distributed as a binary file (the result of the compilation of the source code) and a header file, listing the return type, name and argument list of the available functions. Thus the compiler can check the function call and insert a place holder. The binary code is inserted later by the *linker*.

Advantages:

- The source code of a library does not need to be compiled each time it is used
- The source code of commercial libraries can be kept private

The command `#include <filename>` tells the *preprocessor* to read the header file `filename` from disk.

Header files usually have the extension `.h`. Only the header files of the standard library have no extensions.

12 Loops

Loops

There is nearly no program which does not need to do something repeatedly.

In C++ there are three different types of loops.

There are loops where something has to be done

- a fixed number of times
- only if a condition is true
- at least once and be repeated while a condition is true

The `while`-loop

- The `while`-loop is only executed if the condition after while is true. It is repeated as long as the condition remains true.

```
int i=0;
while (i<10)
{
    std::cout << i*i << std::endl;
    i = i + 1;
}
```

- Of course this can easily be changed to a loop which is only executed if the condition is false:

```
int i=0;
while (!(i>=10))
{
    std::cout << i*i << std::endl;
    i = i + 1;
}
```

The `do while`-loop

- The `do while` is executed at least once. It is repeated while the condition remains true.

```
int i=0;
do
{
    std::cout << i*i << std::endl;
    i = i + 1;
} while (i<10);
```

- or false

```
int i=0;
do
{
    std::cout << i*i << std::endl;
    i = i + 1;
} while (!(i>10));
```

The `for`-loop

- The `for`-loop is usually executed a fixed number of times. The counter-variable can be defined and initialised in the loop.

  ```
  for (int i=0;i<10;++i)
  {
      std::cout << i*i << std::endl;
  }
  ```

- or descending

  ```
  for (int i=9;i>=0;--i)
      std::cout << i*i << std::endl;
  ```

Complex `for`-loops

The `for`-loop is actually a very powerfull construct.

- The counter can not only be integer
- The condition can be any arbitrary complex condition
- The increment is also very flexible

```
for (double i=0.0;i<1.7;i=i+0.1)
{
    std::cout << i*i << std::endl;
}

std::cout << std::endl << std::endl;

// better:
for (int i=0;i<17;++i)
{
    std::cout << 0.1*i*0.1*i << std::endl;
}
```

Abbreviated forms

There are several abbreviated forms in C++ which save some writing:

`++i`	`i = i + 1`	increments a variable by one
`i++`	`i = i + 1`	increments a variable by one
`--i`	`i = i - 1`	decrements a variable by one
`i--`	`i = i - 1`	decrements a variable by one
`i+=a`	`i = i + a`	adds a value to a variable
`i-=a`	`i = i - a`	subtracts a value from a variable
`i*=a`	`i = i * a`	multiplies a variable with a value
`i/=a`	`i = i / a`	divides a variable by a value

Loops

```
// while-loop
int i=0;
while (i<10)
{
    std::cout << i*i << std::endl;
    i = i + 1;
}

// do-while-loop
int i=0;
do
{
    std::cout << i*i << std::endl;
    i = i + 1;
} while (!(i>10));

// for-loop
for (int i=9;i>=0;--i)
    std::cout << i*i << std::endl;
```

New Exercises

1. Calculate e^x with the power series

$$e^x = 1 + \sum_{n=1}^{\infty} \frac{x^n}{n!} = 1 + \sum_{n=1}^{\infty} y_n.$$

to calculate approximate solutions $S_i(x)$ use

$$\begin{aligned} y_1 &= x \\ S_1 &= 1 + y_1 \end{aligned}$$

and calculate $n = 2, 3, \ldots$ with

$$\begin{aligned} y_n &= \frac{x}{n} y_{n-1} \\ S_n &= S_{n-1} + y_n. \end{aligned}$$

Check the convergence for the examples above. How many significant digits do you get for $x \in \{1, 5, 20, -1, -5, -20\}$ with `double` and `long double` precision floating point variables. Use the floating-point manipulators to get the output with the desired precision and in scientific number format.

New Exercises

2. Change the above program to calculate e^x for negative exponents using

$$e^{-x} = \frac{1}{e^x}$$

Is the result better? Why?

New Exercises

3. *Function with selectable precision*

   a) Write a function `double Exp(double x, double eps, int n)`, which automatically calculates the exponential with the optimal algorithm (using $e^{-x} = \frac{1}{e^x}$). Use as terminating condition, that the increment is smaller than a certain values `eps` or the number of iterations is larger than a certain value n. Set the default values 10^{-16} for `eps` and 500 for `n` to be able to call `Exp(5.0)`, `Exp(5.0,1e-10)` or `Exp(5.0,1e-10,100)`.
   b) Test the program and compare the value with the result of the function `exp` provided by the system library for $x \in \{1.1, -4.9, 13, -17.5\}$.

Solution of the Exercises

1. Fast Powerfunction

   While computing the power function some multiplication can be saved. For example

$$x^8 = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$$

   can be computed in the following three steps

$$\begin{aligned} x^2 &= x \cdot x \\ x^4 &= x^2 \cdot x^2 \\ x^8 &= x^4 \cdot x^4 \end{aligned}$$

   such as only 3 instead of 7 multiplication are needed.

   A more general recursive formulation of this fast computation is given in the following:

$$x^n = \begin{cases} (x^{n/2})^2 & \text{if } n \text{ even} \\ x \cdot x^{n-1} & \text{if } n \text{ odd} \\ x & \text{if } n = 1 \end{cases}$$

Solution of the Exercises

1. (ctd.)

   a) Write a function `int PowerFast(int x, int exp)` that computes the power function for exponents `exp` ≥ 1 in the fast way as presented above.
   b) Write a `main()` program to test your function. In the main routine also test if your implementation yields the same result as the `int Power(int x, int exp)` function presented in the lecture. Otherwise you have made an error :-).

Solution of Exercise 1

```
#include<iostream>

int PowerFast(int x, int n)
{
    if (n==1)
        return x;
    if ((n%2) == 0)
    {
        int result;
        result = PowerFast(x,n/2);
        return result * result;
    }
    else
        return x * PowerFast(x,n-1);
}


int Power(int x, int n)
{
    if (n==1)
        return x;
    else
        return x*Power(x,n-1);
}
```

Solution of Exercise 1 (ctd.)

```
int main()
{
    std::cout.precision(10);
    std::cout << "Result␣of␣slow␣power␣caculation:␣"
              << Power(2,25) << std::endl;
    std::cout << "Result␣of␣fast␣power␣caculation:␣"
              << PowerFast(2,25) << std::endl;
}
```

Solution of the Exercises

2. Number representation
   a) The determinant of a 2×2 matrix A is calculated as

$$\det(A) = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

Write a C++ function `float determinant(float a, float b, float c, float d)` which calculates the determinant. Compute the determinant $det(A)$ of the matrix

$$A = \begin{pmatrix} 100 & 0.01 \\ -0.01 & 100 \end{pmatrix}.$$

The exact solution is 10000.0001.

Be careful! The C++ compiler automatically rounds the output of `float` and `double` values to make the result more readable. As we are interested in the non-rounded value we have to switch of the rounding.

Solution of the Exercises

2. (ctd.)

   a) (ctd.) With the following main program all digits are written to the screen:

```
int main ()
{
     cout.precision(10);
     cout << determinant(100, 0.01, -0.01, 100);
};
```

   Why is the result not correct? What happens if you use the data type `double` instead of `float`?

   b) Calculate $(a+b)+c$ and $a+(b+c)$ with a C++ program for $a = 10^n$, $b = -10^n$ and $c = 10^{-n}$ and $n = 6, 7, 8, \ldots, 14$. For which n is the addition on your Computer no longer associative if you use `float`? Why? Please also use the main program from above to get all digits.

Solution of Exercise 2.1

```
#include<iostream>

float DeterminantF(float a, float b, float c, float d)
{
        return a*d-b*c;
}

double DeterminantD(double a, double b, double c, double d)
{
        return a*d-b*c;
}

int main (void)
{
        std::cout.precision(12);
        std::cout << "Determinant of ( 100 1.0e-2, -1.0e-2 100 ) is ";
        std::cout << DeterminantF(100.0, 1.0e-2, -1.0e-2, 100.0)
         << " (with float) " << std::endl << "and ";
        std::cout << DeterminantD(100.0, 1.0e-2, -1.0e-2, 100.0)
         << " (with double)." << std::endl << std::endl;

        std::cout << "Determinant of ( 100 1.0e-10, -1.0e-10 100 ) is ";
        std::cout << DeterminantF( 1.00001, 1.0, -1.0, 1.0)
         << " (with float) " << std::endl << "and ";
        std::cout << DeterminantD( 1.00001, 1.0, -1.0, 1.0)
         << " (with double)." << std::endl << std::endl;
```

Solution of Exercise 2.2

```
    float a = 1.0e6;
    float b = -1.0e6;
    float c = 1.0e-6;

    std::cout << a << " " << b << " " << c << std::endl;
    std::cout << (a+b)+c << std::endl;
    std::cout << a+(b+c) << std::endl;
    std::cout << std::endl;

    a = 1.0e7;
```

```
    b = -1.0e7;
    c = 1.0e-7;

    std::cout << a << "␣" << b << "␣" << c << std::endl;
    std::cout << (a+b)+c << std::endl;
    std::cout << a+(b+c) << std::endl;
    std::cout << std::endl;

    a = 1.0e8;
    b = -1.0e8;
    c = 1.0e-8;

    std::cout << a << "␣" << b << "␣" << c << std::endl;
    std::cout << (a+b)+c << std::endl;
    std::cout << a+(b+c) << std::endl;
    std::cout << std::endl;
```

Solution of Exercise 2: Output

```
Determinant of ( 100 1.0e-2, -1.0e-2 100 ) is 10000 (with float)
and 10000.0001 (with double).

Determinant of ( 100 1.0e-10, -1.0e-10 100 ) is 2.00001001358 (with float)
and 2.00001 (with double).

1000000 -1000000 9.99999997475e-07
9.99999997475e-07
9.99999997475e-07

10000000 -10000000 1.00000001169e-07
1.00000001169e-07
9.99998519546e-08

100000000 -100000000 9.99999993923e-09
9.99999993923e-09
9.99716576189e-09

1000000000 -1000000000 9.99999971718e-10
9.99999971718e-10
9.89530235529e-10

10000000000 -10000000000 1.00000001335e-10
1.00000001335e-10
0
```

13 Loops

Scope of Variables

Variables defined inside a `for`-loop are only valid inside the loop

```
#include <iostream>

int main()
{
    int a = 2;
    for (int i=0; i<10; ++i)
    {
        int x = a*2;  // x is always 4
        std::cout << i << ":␣" << x << std::endl;
        x = x + 1;
    }
    std::cout << i << ":␣" << x << std::endl;
    // error i and x are undefined
```

```
}
```

14 Formatted IO

IO-Manipulators

It is possible to change the properties of output (and sometimes input) by writting so-called IO-Manipulators to the output stream (or reading them from the input stream). To use the modifiers with arguments you need to

```
#include <iomanip>
```

The manipulators without arguments are already included in `#include <iostream>`

Integer Manipulators

dec	Turns on the dec flag
oct	Turns on the oct flag
hex	Turns on the hex flag

```
#include <iostream>

int main()
{
    int a;
    std::cout << "Please␣enter␣a␣number␣";
    std::cin >> std::oct >> a;

    std::cout << "The␣number␣was␣octal␣" << std::oct << a;
    std::cout << "␣which␣is␣in␣decimal␣" << std::dec << a << std::endl;

}
```

Floating-Point Manipulators

fixed	Turns on the fixed flag
scientific	Turns on the scientific flag
setprecision(int p)	Sets the number of digits of precision to p
setw(int w)	Sets the width of the next field to w

```
#include <iostream>
#include <iomanip>

int main()
{
    std::cout << "1/3␣is␣with␣three␣digits␣␣" << std::setw(15);
    std::cout << std::setprecision(3) << 1./3. << std::endl;
    std::cout << "1/3␣is␣with␣twelve␣digits␣" << std::setw(15);
    std::cout << std::setprecision(12) << 1./3. << std::endl;
}
```

```
1/3 is with three digits           0.333
1/3 is with twelve digits  0.333333333333
```

Formating Manipulators

left	Turns on the left flag
right	Turns on the right flag

```
#include <iostream>
#include <iomanip>

int main()
{
    std::cout << std::left;
    std::cout << "1/3 is with three digits  " << std::setw(15);
    std::cout << std::setprecision(3) << 1./3. << std::endl;
    std::cout << "1/3 is with twelve digits " << std::setw(15);
    std::cout << std::setprecision(12) << 1./3. << std::endl;
}
```

```
1/3 is with three digits  0.333
1/3 is with twelve digits 0.333333333333
```

Boolean Manipulators

boolalpha	Turns on the boolalpha flag
noboolalpha	Turns off the boolalpha flag

```
#include <iostream>

int main()
{
    bool a = true;
    std::cout << "without boolalpha flag the value of a is ";
    std::cout << a << std::endl;
    std::cout << "with boolalpha flag the value of a is ";
    std::cout << std::boolalpha << a << std::endl;
}
```

```
without boolalpha flag the value of a is 1
with boolalpha flag the value of a is true
```

15 Comments

Comments

- Comments can be used to include documentation directly into the program.
- Comments begin with double-slashes // and continue for the rest of the line, e.g.

```
// This is a comment
// Each new line of comments has to have a new double-slash
```

- Comments are ignored by the compiler.
- Comment blocks start with /* and end with */. All text between this two lines is ignored. Comment blocks can not be nested.

```
/* This is a comment block.
   It can span several lines */
```

- Comment blocks are very usefull to force the compiler to ignore parts of the program (e.g. for debugging) ⇒ use double-slashes for comments

16 Runtime Measurement

Measuring Computation Time

- For the optimization of programs it is often helpful to know how much runtime is consumed by a function.
- A very simple possibility to measure the execution time of a function (or some arbitrary part of the program) is by using the `clock()` function. It returns the number of clock ticks since the programm was started.
- To use the function you have to `#include <ctime>`.

Measuring Computation Time

- To get the runtime of a function in seconds, call clock before the beginning and after the end of the function, subtract the value and divide the result by the constant `CLOCKS_PER_SEC`. The measured time is the real processor time. In multi-tasking systems it is automatically corrected for the share of the processor time the application gets.
- If the runtime of the function is too short to be measured precisely, it helps to include the function in a loop and call it repeatedly. The runtime of one function call can then be obtained by division of the total runtime by the number of function calls.

```
#include <ctime>   //enables the use of function clock()
#include <cmath>
#include <iostream>

int main()
{
    const int REPETITIONS=100000000;
    clock_t begin_rec = clock(); // begin of runtime measurement

    double x = 1000.;
    double n = 0.3333;
    // call of Power function
    double result;
    for (int i=0;i<REPETITIONS;++i)
        result = pow(x,n);

    clock_t end_rec = clock(); // end of runtime measurement

    std::cout << "Execution time of the power function to calculate "
          << x << "^" << n << " = " << result << " was "
          << double(end_rec - begin_rec)/CLOCKS_PER_SEC//REPETITIONS
          << " seconds" << std::endl;
    return 0;
}
```

Solution of the Exercises

1. Calculate e^x with the power series

$$e^x = 1 + \sum_{n=1}^{\infty} \frac{x^n}{n!} = 1 + \sum_{n=1}^{\infty} y_n.$$

to calculate approximate solutions $S_i(x)$ use

$$\begin{aligned} y_1 &= x \\ S_1 &= 1 + y_1 \end{aligned}$$

and calculate $n = 2, 3, \ldots$ with

$$\begin{aligned} y_n &= \frac{x}{n} y_{n-1} \\ S_n &= S_{n-1} + y_n. \end{aligned}$$

Check the convergence for the examples above. How many significant digits do you get for $x \in \{1, 5, 20, -1, -5, -20\}$ with `double` and `long double` precision floating point variables. Use the floating-point manipulators to get the output with the desired precision and in scientific number format.

Solution of the Exercises

2. Change the above program to calculate e^x for negative exponents using

$$e^{-x} = \frac{1}{e^x}$$

Is the result better? Why?

Solution of the Exercises

3. *Function with selectable precision*

   a) Write a function `double Exp(double x, double eps, int n)`, which automatically calculates the exponential with the optimal algorithm (using $e^{-x} = \frac{1}{e^x}$). Use as terminating condition, that the increment is smaller than a certain values `eps` or the number of iterations is larger than a certain value n. Set the default values 10^{-16} for `eps` and 500 for `n` to be able to call `Exp(5.0)`, `Exp(5.0,1e-10)` or `Exp(5.0,1e-10,100)`.

   b) Test the program and compare the value with the result of the function `exp` provided by the system library for $x \in \{1.1, -4.9, 13, -17.5\}$.

New Exercises: Approximation of π

Given the set A_n:

$$A_n = \left\{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid x^2 + y^2 \le n^2\right\}$$

write a C++ function `int WithinCircle(int n)` that calculates the number of elements of the set A_n. *Hint:* This corresponds to the number of points of a $(2n + 1) \times (2n + 1)$-grid that lie within a circle of radius n or directly on it.

How can π be approximated using the computed number of elements?

Print out the appoximated value of π for $n = 10$, $n = 50$, $n = 100$, $n = 500$ and $n = 1000$.

New Exercises: The Fibonacci Sequence

The Fibonacci sequence: 1 1 2 3 5 8 13 21 34 $\cdots$ is created by successively adding the last two numbers of a sequence to create the next number in the sequence:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n) &= Fib(n-1) + Fib(n-2) \end{aligned}$$

$Fib(n)$ can be computed using recursion as well as iteratively:

- As presented in the lecture the basic idea behind recursion is breaking down a problem into smaller problems, until a solvable portion is found, and then using that answer to solve the rest of the problem. A recursive function calls itself with different parameters, and defines an exit clause that is guaranteed to be reached.
- An iterative function includes a loop, which iterates a pre-determined number of times, or checks for an exit clause after or before every repetition.

New Exercises: The Fibonacci Sequence

In this exercise we will compute $Fib(n)$ both recursively and iteratively.

- Write a C++ function `int Fibonacci_Recursive(int n)` that accepts a number n ($n \ge 0$) and computes $Fib(n)$ recursively.
- Write a second C++ function `int Fibonacci_Iterative(int n)` that computes $Fib(n)$ iteratively.
- Write a `main()` program to test both functions. The program should ask for a positive number n, compute both `Fibonacci_Recursive(n)` and `Fibonacci_Iterative(n)` and compare the computation time.
- When computing $Fib(n)$ recursively, how many times are $Fib(0)$ or $Fib(1)$ called? How does the running time of `Fibonacci_Recursive(n)` grow with n?

Expected Solution

```
#include<iostream>

int main()
{
    long double x;
```

```
    std::cout << "Bitte␣geben␣Sie␣eine␣Zahl␣ein:␣";
    std::cin >> x;
    long double increment = x;
    long double sum = 1.;
    int n = 1;
    do
    {
        sum += increment;
        std::cout << n << ".␣␣sum:␣" << sum;
        std::cout << "␣␣␣increment:␣" << increment << std::endl;
        ++n;
        increment *= x/n;
    } while (n<100);
}
```

Improved Solution

```
#include<iostream>
#include<iomanip>
#include<cmath>

int main()
{
    long double x;
    std::cout << "Bitte␣geben␣Sie␣eine␣Zahl␣ein:␣";
    std::cin >> x;
    long double increment = x;
    long double sum = 1.;
    int n = 1;
    std::cout << std::setprecision(19) << std::scientific;
    do
    {
        sum += increment;
        std::cout << n << ".␣␣sum:␣" << sum;
        std::cout << "␣␣␣increment:␣" << increment << std::endl;
        ++n;
        increment *= x/n;
    } while ((n<1000) and (fabs(increment/sum)>1e-20));
```

Improved Solution (ctd.)

```
    std::cout << "Exact␣solution:␣" << exp(x) << std::endl;
    std::cout << "Error:␣" << sum-exp(x) << std::endl;
    std::cout << "Relative␣Error:␣" << (sum-exp(x))/exp(x) << std::endl;
    int sigDigits = int(log10(fabs(exp(x)/(sum-exp(x)))));
    if (sigDigits<0)
        sigDigits=0;
    std::cout << "Significant␣digits:␣" << sigDigits << std::endl;
}
```

Solution of the Exercises

2. Change the above program to calculate e^x for negative exponents using

$$e^{-x} = \frac{1}{e^x}$$

Is the result better? Why?

Expected Solution

```
#include<iostream>

int main()
{
    long double x;
    std::cout << "Bitte␣geben␣Sie␣eine␣Zahl␣ein:␣";
    std::cin >> x;
    long double increment=x;
    if (x<0)
        increment = -x;
    long double sum = 1.;
    int n = 1;
    do
    {
        sum += increment;
        if (x<0)
            std::cout << n << ".␣␣sum:␣" << sum;
        else
            std::cout << n << ".␣␣sum:␣" << 1./sum;
        std::cout << "␣␣␣increment:␣" << increment << std::endl;
        ++n;
        if (x<0)
            increment *= -x/n;
        else
            increment *= x/n;
    } while (n<100);
}
```

Improved Solution

```
int main()
{
    double x;
    std::cout << "Bitte␣geben␣Sie␣eine␣Zahl␣ein:␣";
    std::cin >> x;
    int sign=1;
    if (x<0)
    {
        sign=-1;
        x=-x;
    }
    double increment = x;
    double sum = 1.;
    int n = 1;
    std::cout << std::setprecision(16) << std::scientific;
    do
    {
        sum += increment;
        if (sign<0)
            std::cout << n << ".␣␣S:␣" << 1./sum;
        else
            std::cout << n << ".␣␣S:␣" << sum;
        std::cout << "␣␣␣increment:␣" << increment << std::endl;
        ++n;
        increment *= x/n;
    } while ((n<1000) and (fabs(increment/sum)>1e-16));
```

Improved Solution (ctd.)

```
    if (sign<0)
    {
        x=-x;
        sum = 1./sum;
```

```
    }
    std::cout << "Exact␣solution:␣" << exp(x) << std::endl;
    std::cout << "Error:␣" << sum-exp(x) << std::endl;
    std::cout << "Relative␣Error:␣" << (sum-exp(x))/exp(x) << std::endl;
    int sigDigits = int(log10(fabs(exp(x)/(sum-exp(x)))));
    if (sigDigits<0)
        sigDigits=0;
    std::cout << "Significant␣digits:␣" << sigDigits << std::endl;
}
```

The result is better in the second case, as there are no longer negative terms in the series and cancellation is avoided.

```
#include<iostream>
#include<iomanip>
#include<cmath>

double Exp(double x, double eps=1e-16, int n=500)
{
    if (x<0)
        return 1./Exp(-x,eps,n);
    double increment = x;
    double result = 1.;
    int i = 1;
    do
    {
        result += increment;
        ++i;
        increment *= x/i;
    } while ((i<n) and (fabs(increment/result)>eps));
    return result;
}



int main()
{
    double x;
    std::cout << "Please␣enter␣a␣number:␣";
    std::cin >> x;
    std::cout << std::scientific << std::setprecision(16);
    std::cout << "Approximate␣Solution:␣" << Exp(x,1e-10,100) << std::endl;
    std::cout << "Exact␣solution:␣" << exp(x) << std::endl;
    std::cout << "Error:␣" << Exp(x,1e-10,100)-exp(x) << std::endl;
    std::cout << "Relative␣Error:␣" << (Exp(x,1e-10,100)-exp(x))/exp(x) << std::endl;
    int sigDigits = int(log10(fabs(exp(x)/(Exp(x,1e-10,100)-exp(x)))));
    if (sigDigits<0)
        sigDigits=0;
    std::cout << "Significant␣digits:␣" << sigDigits << std::endl;
}
```

Steps in Program Development

- Take small steps
- Test your code after each step
- Gradually make your program more complex

Practical Programming Examples

- Write a program that can convert decimal to binary numbers. The binary numbers can be represented as string.

Practical Programming Examples

- Pascals' Triangle:

  With Pascal's Triangle it is possible to determine the coefficients $p_{n,i}$, $n \geq 1$, $0 \leq i \leq n$ of the binomial formula for $(a+b)^n$. The coefficient $p_{n,i}$ is the sum of the two values above it. The outermost coefficients $p_{n,0}$ and $p_{n,n}$ are always one.

				1					$n = 0$
			1		1				$n = 1$
		1		2		1			$n = 2$
	1		3		3		1		$n = 3$
1		4		6		4		1	$n = 4$

		$p_{0,0}$		
	$p_{1,0}$		$p_{1,1}$	
$p_{2,0}$		$p_{2,1}$		$p_{2,2}$

with

$$p_{n,i} = \begin{cases} p_{n-1,i-1} + p_{n-1,i} & \text{if } 0 < i < n \\ 1 & \text{if } i = 0 \text{ or } i = n \\ 0 & \text{else} \end{cases}$$

  Write a recursive function `int pascal(int n, int i)` which calculates the coefficient $p_{n,i}$

Solution of the Exercises: Approximation of π

Given the set A_n:

$$A_n = \left\{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid x^2 + y^2 \leq n^2\right\}$$

write a C++ function `int WithinCircle(int n)` that calculates the number of elements of the set A_n. *Hint:* This corresponds to the number of points of a $(2n+1) \times (2n+1)$-grid that lie within a circle of radius n or directly on it.

How can π be approximated using the computed number of elements?

Print out the appoximated value of π for $n = 10$, $n = 50$, $n = 100$, $n = 500$ and $n = 1000$.

Solution of the Exercises: The Fibonacci Sequence

The Fibonacci sequence: 1 1 2 3 5 8 13 21 34 $\cdots$ is created by successively adding the last two numbers of a sequence to create the next number in the sequence:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n) &= Fib(n-1) + Fib(n-2) \end{aligned}$$

$Fib(n)$ can be computed using recursion as well as iteratively:

- As presented in the lecture the basic idea behind recursion is breaking down a problem into smaller problems, until a solvable portion is found, and then using that answer to solve the rest of the problem. A recursive function calls itself with different parameters, and defines an exit clause that is guaranteed to be reached.
- An iterative function includes a loop, which iterates a pre-determined number of times, or checks for an exit clause after or before every repetition.

Solution: Approximation of π

```
#include<iostream>
#include<cmath>

int WithinCircle(int n)
{
    int result=0;
    for (int i=0;i<(2*n+1);++i)
        for (int j=0;j<(2*n+1);++j)
        {
            int x = i-n;
            int y = j-n;
            if ((x*x+y*y)<=n*n)
                ++result;
        }
    return result;
}

void OutputPi(int n)
{
    std::cout.precision(10);
    int within = WithinCircle(n);
    std::cout << "radius␣" << n << "␣yields␣" << within
              << "␣elements.␣Pi␣is␣approx␣"
              << (4.*within)/((2*n+1)*(2*n+1)) << std::endl;
}
```

Solution: Approximation of π

```
int main()
{
    OutputPi(10);
    OutputPi(50);
    OutputPi(100);
    OutputPi(500);
    OutputPi(1000);
    OutputPi(10000);
    std::cout << "Correct␣result␣is␣" << M_PI << std::endl;
}
```

```
radius 10 yields 317 elements. Pi is approx 2.875283447
radius 50 yields 7845 elements. Pi is approx 3.076169003
radius 100 yields 31417 elements. Pi is approx 3.110517066
radius 500 yields 785349 elements. Pi is approx 3.13512262
radius 1000 yields 3141549 elements. Pi is approx 3.138409806
radius 10000 yields 314159053 elements. Pi is approx 3.141276395
Correct result is 3.141592654
```

Solution: Better Approximation of π

```
#include<iostream>
#include<cmath>

int WithinCircleFast(int n)
{
    int result=0;
    for (int i=0;i<n+1;++i)
        for (int j=1;j<n+1;++j)
        {
            if ((i*i+j*j)<=n*n)
                ++result;
        }
    result *= 4;
```

```
    ++result;
    return result;
}

void OutputPi(int n)
{
    std::cout.precision(10);
    int within = WithinCircleFast(n);
    std::cout << "radius␣" << n << "␣yields␣" << within
              << "␣elements.␣Pi␣is␣approx␣"
              << double(within)/(n*n) << std::endl;
}
```

Solution: Better Approximation of π

```
int main()
{
    OutputPi(10);
    OutputPi(50);
    OutputPi(100);
    OutputPi(500);
    OutputPi(1000);
    OutputPi(10000);
    std::cout << "Correct␣result␣is␣" << M_PI << std::endl;
}
```

```
radius 10 yields 317 elements. Pi is approx 3.17
radius 50 yields 7845 elements. Pi is approx 3.138
radius 100 yields 31417 elements. Pi is approx 3.1417
radius 500 yields 785349 elements. Pi is approx 3.141396
radius 1000 yields 3141549 elements. Pi is approx 3.141549
radius 10000 yields 314159053 elements. Pi is approx 3.14159053
Correct result is 3.141592654
```

Solution: The Fibonacci Sequence

```
#include <ctime>
#include <iostream>

int fibIter(unsigned int n)
{
    unsigned int x1=1,x2=1;

    for (int i=1;i<n;++i)
    {
        int temp = x1 + x2;
        x1=x2;
        x2=temp;
    }

    return x2;
}

int fibRec(unsigned int n)
{
    if ((n==0)||(n==1))
        return 1;
    else
        return fibRec(n-1) + fibRec(n-2);
}
```

```
int main()
{
    const int REPETITIONS=100000;
    const int n=25;
    clock_t begin_rec = clock(); // begin of runtime measurement

    unsigned int result;
    for (int i=0;i<REPETITIONS;++i)
        result = fibRec(n);
    clock_t end_rec = clock(); // end of runtime measurement

    std::cout << "Execution␣time␣of␣recursive␣function␣for␣n␣=␣"
          << n << "␣=␣" << result << "␣was␣"
          << double(end_rec - begin_rec)/CLOCKS_PER_SEC/REPETITIONS
          << "␣seconds" << std::endl;

    begin_rec = clock(); // begin of runtime measurement

    for (int i=0;i<REPETITIONS*1000;++i)
        result = fibIter(n);
    end_rec = clock(); // end of runtime measurement

    std::cout << "Execution␣time␣of␣iterative␣function␣for␣n␣=␣"
          << n << "␣=␣" << result << "␣was␣"
          << double(end_rec - begin_rec)/CLOCKS_PER_SEC/(1000*REPETITIONS)
          << "␣seconds" << std::endl;

    return 0;
}
```

```
Execution time of recursive function for n = 25 = 121393 was 0.000728 seconds
Execution time of iterative function for n = 25 = 121393 was 3.22e-08 seconds
```

18 Arrays in C++

Arrays in C++

- The most comfortable way to get an array in C++ is the Vector-Class of the Standard Template Library (STL), which is part of the Standard Library.
- Vectors are sets of values, where each element of the vector can be accessed by an index. The first index is zero.
- To use vectors you have to `#include <vector>`. Vectors are part of the namespace std.
- Vectors are created like ordinary variables. The variable type is `std::vector<type>`, where `type` is the variable type of the elements of the vector.

  ```
  std::vector<int> intVector;
  ```

- The length (amount of elements) of the vector can be spezified in brackets after the variable name of the vector.

  ```
  std::vector<int> intVector(7);
  ```

Using Vectors

- A default value for each element can be spezified after the size. If this is not given the elements are not initialized.

```
std::vector<int> intVector(7,0);
```

- Vectors can also be initialized as a copy of an existing vector with the same element type.

```
std::vector<int> intVector(7,0);
std::vector<int> secondVector(intVector);
std::vector<int> thirdVector = intVector;
```

- The elements are accessed by specifying the index in square brackets after the variable name. The index of the first element is 0 the index of the last element is $\mathsf{size} - 1$

```
intVector[1] = 3; // sets second element to three
```

Using Methods of Vectors

Vectors have spezial functions (methods) which can be called by stating the variable name, a point and the name of the function.

```
int size = intVector.size();   // size() returns the length of the vector
```

name of method	purpose
`size()`	returns length of vector
`resize(int newSize)`	change the length of vector. Elements at the beginning remain the same. If the new size is smaller the vector is truncated.
`front()`	returns a reference to the first element
`back()`	returns a reference to the last element
`push_back(value)`	add an element at the end (and increase size by one)
`clear()`	erases all elements (size is zero afterwards)

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<double> a(7);    // define a vector with 7 elements
    std::cout << a.size() << std::endl;  // output size
    for (int i=0;i<7;++i)
        a[i] = i*0.1;            // assign values to the elements
    double d = 4 * a[2];         // use third element
    std::vector<double> c(a);    // create a copy of vector a
    // output last elements of a and c
    std::cout << a.back() << " " << c.back() << std::endl;
    std::vector<std::string> b; // define empty vector b
    b.resize(3);                 // set size of b
    for (int i=2;i>=0;--i)
        std::cin >> b[i];        // read elements from keyboard
    b.resize(4);                 // redefine the size of b
    b[3] = "blub";               // set third element
    b.push_back("blob");         // add an element at the end
    // write the elements of b to the screen
    for (int i=0;i<b.size();++i)
        std::cout << b[i] << std::endl;
    b.clear();
    std::cout << b.size() << std::endl;
}
```

C-Arrays

- C arrays of a fixed size are defined by stating the type the variable name and the number of elements after the variable name in square brackets
- The elements are accessed by specifying the index in square brackets after the variable name
- There is no way to query the size of a C array
- There is no way to resize a C array
- The index of the first element is 0 the index of the last element is size − 1

```
int main()
{
    int numbers[27];
    for (int i=0;i<27;++i)
        number[i]=i*i;
    for (int i=0;i<27;++i)
        std::cout << i << "␣" << number[i] << std::endl;
}
```

Pointers

- Pointers are a concept which is very close to the physical hardware of a computer
- Pointers store the address of a variable in memory
- A pointer to a variable of a certain type is defined by writing an asterisk `*` in front of the variablename, e.g. `int *intPointer;`
- The content of a variable can be addressed over its pointer by writing an asterisk `*` in front of the variable name
- If a pointer is not pointing to a variable it is good coding style to give him the value `0`.

Pointers

- The address of a variable can be obtained by writing an ampersand `&` in front of it, e.g.

  ```
  int a = 2;
  int *intPointer = &a;
  *intPointer = 4;
  ```

- The operator `++/--` do not increase/decrease a pointer by one byte, but by once the size of the variable type to which it is pointing.
- If an integer i is added/subtracted to/from a pointer it is changed by i times the size of the variable type to which it is pointing

C-Arrays

- If a C-array is created, the array variable is a pointer to the first element
- The use of the square bracket operator `a[i]` is equivalent to an pointer access `*(a+i)`

```
int main()
{
    int numbers[27];
    for (int i=0;i<27;++i)
        number[i]=i*i;
    for (int i=0;i<27;++i)
        std::cout << i << "␣" << number[i] << std::endl;
}
```

C-Strings

- C-Strings are C-arrays of chars.
- C-Strings a null terminated, i.e. after the last character of a C-String there is an entry with the value zero. Thus C-Strings can be shorter than the reserved array size.
- The length of a C-String can be accessed by the function `strlen(stringVariable)`
- To use strlen you have to `#include<cstring>`

```
#include<iostream>
#include<cstring>

int main()
{
    char name[100];
    std::cout << "Please␣enter␣a␣string" << std::endl;
    std::cin >> name;
    std::cout << "The␣length␣of␣" << name << "␣is␣" << strlen(name)
              << std::endl;
}
```

C-Strings versus Strings

- The size of a C-String can not be adjusted. If more characters are read than the size of the string the rest is written over the following memory space and my overwrite other variables
- Therefore you should use C++ strings
- The length of a C++ string can be accessed by adding `.length()` to the variable name

```
#include<string>
#include<iostream>

int main()
{
    std::string name;
    std::cout << "Please␣enter␣a␣string" << std::endl;
    std::cin >> name;
    std::cout << "The␣length␣of␣" << name << "␣is␣" << name.length()
              << std::endl;
}
```

Do not use C-Arrays and C-Strings!!!

C-arrays and C-strings are unflexible and error-prone. Do not use C-Arrays if possible. Use STL-Vectors and Strings!!!!

Exercises

1. Write a program which reads five double values, stores them in a vector, writes them to the screen in reversed order and calculates mean and variance of the values
2. Change the program to sort the numbers and write them to the screen in ascending order
3. Shrink/enlarge a vector and check the contents of the remaining/newly created elements
4. Write the address of a variable to the screen. Store it in a pointer, increment the pointer and write it to the screen again. Do this for different variable types to check their size.

19 Solution of Linear Equation Systems

Linear Equation Systems

A unique solution for a quadratic linear equation system

$$Ax = b,$$

where A is a $n \times n$ matrix and b is a vector of length n, exists if (all conditions are equivalent)

- $\mathsf{rank}(A) = n$
- $\det(A) \neq 0$
- all eigenvalues of A are different from zero
- all equations of the linear equation system are linearly independent

Classification of Matrices

Depending on the amount of non-zero elements in A we distinguish

fully occupied matrices (also called dense systems) most of the elements have values different from zero.

- Zero elements are not treated sperately
- The full matrix is stored as a two-dimensional array

sparse matrices only $O(n)$ or at most $O(n \log(n))$ elements of the matrix have values different from zero.

- Zero elements are not stored to save memory
- Special data structures are used to store the matrix depending on the structure of the matrix (→ Commas Advanced Scientific Programming).

  In many practical cases sparse matrices have a fixed amount of non-zero elements per line.

Classification of Solvers

We distinguish two different solution strategies

Direct solution methods yield in a known number of operations the solution x for each solvable system $Ax = b$ if exact arithmetics is used.

Iterative solution methods construct starting from an initial value x^0 a sequence

$$x^0, x^1, \ldots, x^k, \ldots \qquad \text{with} ||x - x^k|| \to 0 \text{ for } k \to \infty$$

They are often especially well suited for sparse matrices (→ Commas Advanced Scientific Programming).

Gauss-Elimination

Gauss-Elimination is a direct solution method for dense linear equation systems exploiting the fact that

- upper or lower triangular matrices are easy to solve
- A linear equation system remains basically unchanged if
  - two equations are exchanged
  - a multiple of one equation is added to an other equation

Elementary Gauß-Elimination

```
input n, (a_ij)
for m=0:n-2 do
    for i = m+1:n-1 do
        q_im = a_im/a_mm
        a_im=0
        for j=m+1:n-1 do
            a_ij=a_ij-q_im*a_mj
        end
        b_i=b_i-q_im*b_m
    end
end
```

Elementary Gauß-Elimination: Backward Insertion

```
x_{n-1} = b_{n-1}/a_{(n-1)(n-1)}
for i=n-2:0 do
    x_i=b_i
    for j=i+1:n-1
```

$x_i = x_i - a_{ij} * x_j$
end
$x_i = x_i / a_{ii}$
end output(x_i)

How to Store Matrices

- Matrices can be stored in different ways.
- They are often optimized for matrices with special properties.

Storing Matrices in a Vector

- Matrices can be stored in a vector.
- The matrix elements are stored either line by line or column by column
- The size has to be stored seperately or can be calculated from the vectors size for quadratic matrices
- The index of the element a_{ij} of a numCol $\times$ numRow matrix A in the vector is calculated by
  - `index = i + j * numRow` if the matrix is stored column by column
  - `index = i * numCol + j` if the matrix is stored line by line.
- This concept can be easily adapted to store three- or higher dimensional matrices without changing the datatype of the matrix.

Storing Square Matrices in a Vector

```
#include<iostream>
#include<vector>

int main()
{
    int sizeA = 3;
    // create a sizeA x sizeA quadratic matrix
    std::vector<int> A(sizeA*sizeA,0);
    // set the diagonal elements to one
    for (int i=0;i<sizeA;++i)
        A[i*sizeA+i] = 1;
    // print matrix
    for (int i=0;i<sizeA;++i)
    {
        for (int j=0;j<sizeA;++j)
            std::cout << A[i*sizeA+j] << " ";
        std::cout << std::endl;
    }
}
```

Storing Rectangular Matrices in a Vector (line by line)

```
#include<iostream>
#include<vector>

int main()
{
    int numCol = 5;
    int numRow = 3;
    // create a numCol x numRow matrix
    std::vector<int> A(numRow*numCol,0);
    // set the diagonal elements to one
    for (int i=0;i<numRow;++i)
        A[i*numCol+i] = 1;
    // print matrix
    for (int i=0;i<numRow;++i)
    {
        for (int j=0;j<numCol;++j)
            std::cout << A[i*numCol+j] << "␣";
        std::cout << std::endl;
    }
}
```

20 Self-defined Variable Types

Self-defined Variable Types (Structures)

- In C++ it is possible to create user-defined variable types, which are a composition of different variable types. They are called structures.
- Structures are defined by stating the keyword `struct`, the name of the variable type and a list of sub-variable types and sub-variable names in curly brackets:

  ```
  struct Point
  {
      double x,y;
      std::string name
  }
  ```

  The structure definition has to be outside any function definition.
- Variables with the newly defined variable type are defined like ordinary variables:

  ```
      Point center;
  ```

Using Structures

- Directly at the definition structures can be initialized with a list of all the variable values in curly brackets:

  ```
      Point center = {4.0,-2.3,"Center␣of␣the␣Circle"};
  ```

- The elements of the structure are acessed via the variable name, followed by a point and the name of the sub-variable:

  ```
      center.x = 4.0;
      center.y = -2.3;
      center.name = "Center␣of␣the␣Circle";
  ```

- Structures can be copied and initialized from variables of the same type:

  ```
      Point middle = center;
      middle.x = 3.0;
      center = middle;
  ```

Matrices using Structures

```
#include<iostream>
#include<vector>

struct Matrix
{
    int numCol, numRow;
    std::vector<int> elem;
};

int main()
{
    Matrix A = {4,3,std::vector<int>(4*3,0)};
    A.elem[0] = A.elem[5] = A.elem[10] = 1;
    for (int i=0;i<A.numRow;++i)
    {
        for (int j=0;j<A.numCol;++j)
            std::cout << A.elem[i*A.numCol+j] << "␣";
        std::cout << std::endl;
    }
}
```

Improved Matrices using Structures

```
#include<iostream>
#include<vector>

struct Matrix
{
    int numCol, numRow;
    std::vector<int> elem;
};

void InitMatrix(Matrix &A, int numCol, int numRow, int initialValue=0)
{
    A.numCol = numCol;
    A.numRow = numRow;
    A.elem.resize(numCol*numRow);
    for (int i=0;i<A.elem.size();++i)
        A.elem[i] = initialValue;
}

int &AccessElement(Matrix &A, int i, int j)
{
    return A.elem[i*A.numCol+j];
}
```

Improved Matrices using Structures

```
void PrintMatrix(Matrix &A)
{
    for (int i=0;i<A.numRow;++i)
    {
        for (int j=0;j<A.numCol;++j)
            std::cout << AccessElement(A,i,j) << "␣";
        std::cout << std::endl;
    }
}

int main()
{
    Matrix A;
    InitMatrix(A,4,3);
```

```
    AccessElement(A,0,0) = 1;
    AccessElement(A,1,1) = 1;
    AccessElement(A,2,2) = 1;
    PrintMatrix(A);
}
```

Matrix as Vector of Vectors

```
#include<iostream>
#include<vector>

int main()
{   // define matrix
    std::vector<std::vector<int> > A(2,std::vector<int> (3));
    for (int i=0;i<A.size();++i)
    {
        for (int j=0;j<A[i].size();++j)
        {
            if (i==j)
                A[i][j]=1;
            else
                A[i][j]=0;
        }
    }
    for (int i=0;i<A.size();++i)   // print matrix
    {
        for (int j=0;j<A[i].size();++j)
            std::cout << A[i][j] << "␣";
        std::cout << std::endl;
    }
}
```

Matrix as Vector of Vectors

```
int main()
{   // define matrix
    std::vector<std::vector<int> > A(3,std::vector<int> (3,2));
    // use matrix
    A[0][0] = A[1][1] = A[2][2] = 1;
    A[0][2] = A[2][0] = 0;
    for (int i=0;i<A.size();++i)   // print matrix
    {
        for (int j=0;j<A.size();++j)
            std::cout << A[i][j] << "␣";
        std::cout << std::endl;
    }
    // resize matrix
    A.resize(4);
    for (int i=0;i<A.size();++i)
        A[i].resize(4);
    // print matrix
    std::cout << std::endl;
    for (int i=0;i<A.size();++i)
    {
        for (int j=0;j<A.size();++j)
            std::cout << A[i][j] << "␣";
        std::cout << std::endl;
    }
}
```

Matrix as Vector of Vectors using Structures

```
#include<iostream>
#include<vector>

struct Matrix
{
    int numCol, numRow;
    std::vector<std::vector<int> > elem;
};

void InitMatrix(Matrix &A, int numCol, int numRow, int initialValue=0)
{
    A.numCol = numCol;
    A.numRow = numRow;
    A.elem.resize(numRow);
    for (int i=0;i<numRow;++i)
    {
        A.elem[i].resize(numCol);
        for (int j=0;j<numCol;++j)
            A.elem[i][j] = initialValue;
    }
}
```

Matrix as Vector of Vectors using Structures

```
int &AccessElement(Matrix &A, int i, int j)
{
    return A.elem[i][j];
}

void ResizeMatrix(Matrix &A, int numCol, int numRow)
{
    A.numCol = numCol;
    A.numRow = numRow;
    A.elem.resize(numRow);
    for (int i=0;i<numRow;++i)
        A.elem[i].resize(numCol);
}

void PrintMatrix(Matrix &A)
{
    for (int i=0;i<A.numRow;++i)
    {
        for (int j=0;j<A.numCol;++j)
            std::cout << AccessElement(A,i,j) << "␣";
        std::cout << std::endl;
    }
}
```

Matrix as Vector of Vectors using Structures

```
int main()
{
    Matrix A;
    InitMatrix(A,4,3);
    AccessElement(A,0,0) = 1;
    AccessElement(A,1,1) = 1;
    AccessElement(A,2,2) = 1;
    PrintMatrix(A);
    std::cout << std::endl;
    ResizeMatrix(A,4,4);
    PrintMatrix(A);
}
```

Solution of Exercises

1. Write a program which reads five double values, stores them in a vector, writes them to the screen in reversed order and calculates mean and variance of the values
2. Change the program to sort the numbers and write them to the screen in ascending order
3. Shrink/enlarge a vector and check the contents of the remaining/newly created elements
4. Write the address of a variable to the screen. Store it in a pointer, increment the pointer and write it to the screen again. Do this for different variable types to check their size.

Exercises

1. Write a main program in which the following linear equation system is solved using the Gauß-Algorithm.

$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 5 \\ -4 \\ 4 \\ -4 \\ 5 \end{pmatrix}$$

   The exact solution of the linear equation system is $x = \{3, 1, 3, 1, 3\}^T$.

21 Advantages of object-oriented programming

How should a good program be?

- Correct/bugfree
- Efficient
- Easy to use
- Comprehensible
- Extendable
- Portable

Developments of the last years

- Computers are faster and cheaper
- The size of programs increases from several hundred lines to hundreds of thousands
- This results also in an increase of the complexity of programs
- Programs are developed in larger groups not by single programmers
- Parallel computing is more and more important

Complexity of Programs

Zeit	Proz	Takt [MHz]	RAM	Disk	Linux Kernel
1982	Z80	6	64KB	800KB	6KB (CPM)
1988	80286	10	1MB	20MB	20KB (DOS)
1992	80486	25	20MB	160MB	140KB (0.95)
1995	PII	100	128MB	2GB	2.4MB (1.3.0)
1999	PII	400	512MB	10GB	13.2MB (2.3.0)
2001	PIII	850	512MB	32GB	23.2MB (2.4.0)
2007	Core2 Duo	2660	1024MB	320GB	302MB (2.6.20)

What to do?

In analogy to mechanical-engineering:

- Split the program in self-contained components
- Determine the necessary functions each component has to provide
- Store all the data necessary for the work inside the corresponding component
- Connect different components via interfaces
- Use the same interface for specialized components which do the same work

Example: Computer

What do we gain?

- Components can be developed separately
- If better versions of a component are available they can be exchanged without modifying the rest of the system
- It is easy to use several instances of a component

How does C++ help?

C++ provides a number of mechanisms supporting this way of structuring computer programs

Classes define components. They are like a description, what a component does and which properties it has (like what a functions typical graphic card provides)

Objects are realizations of a Class (like a graphic card with a certain serial number)

Encapsulation hinders side effects by data hiding

Inheritance facilitates central implementation of code for specialized components

Abstract classes define general interfaces

Virtual functions allow the choice at run-time which specialization of a component should be used

22 Object-oriented programming in C++

Object-oriented programming: example

contents of matrix.h:

```
#include<vector>

class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    double &Elem(int i, int j);
    void Print();
    int Rows();
    int Cols();

  private:
    std::vector<std::vector<double> > a_;
    int numRows_;
    int numCols_;
};
```

Class declaration

```
class MatrixClass
{
// a list of the methods and attributes
};
```

The `class` declaration specifies the interface and the essential properties of the component. A class has *attributes* (all the variables to store data) and *methods* (all the functions a class provides). The definition of the attributes and the declaration of methods is always enclosed in braces. The right brace is always followed by a semicolon.

Class declarations are usually stored in a file with the ending '.hh' or '.h', so called *include*-files.

Encapsulation

1. One must provide the intended user with all the information needed to use the module correctly, and with nothing more.
2. One must provide the implementor with all the information needed to complete the module, and with nothing more.

David L. Parnas (1972)

... but much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies.

Brooks (1975)

Encapsulation

```
class MatrixClass
{
  public:
    // a list of public methods
  private:
    // a list of private methods and attributes
};
```

After `public:` follows the description of the interface, i.e. of the methods which can be called from objects of other classes.

After `private:` follows a description of attributes and methods which are only available to objects of the same class. This is usually the data needed by the component and some special methods only needed internally. The data stored in a class should **not** be directly accessible by objects of other classes to facilitate later modifications of the implementation of the class, i.e. of the way data is stored and the functionality is provided.

Attribute definitions

```
class MatrixClass
{
  private:
    std::vector<std::vector<double> > a_;
    int numRows_;
    int numCols_;
    // further private methods and attributes
};
```

A attribute definition like any variable definition in C++ consists of the type and the variable name. The line is terminated by a semicolon. Possible types are e.g.

- `float` and `double` for single and double precision floating point variables
- `int` and `long int` for integer variables
- `bool` for boolean values
- `std::string` for character strings

Method declarations

```
class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    double &Elem(int i, int j);
};
```

A method declaration always consists of four parts:

- the return type
- the name of the method
- a list of the arguments enclosed in parentheses
- a semicolon

If a method does not return any value, the return type is `void`. If a method does not receive any arguments the parentheses are empty.

Method definitions

```
class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    double &Elem(int i, int j)
    {
        return(a_[i][j]);
    }
};
```

Methods can be defined (i.e. the actual program code of the method is given) directly in the class definition (so called inline functions).

Method definitions (II)

```
void MatrixClass::Init(int numRows, int numCols)
{
    a_.resize(numRows);
    for (int i=0;i<a_.size();++i)
        a_[i].resize(numCols);
    numRows_=numRows;
    numCols_=numCols;
}
```

If methods are defined outside the class definition (this is usually done in a seperate file with the ending .cpp, .cc or .cxx), the name of the function must be prefixed with the name of the class and two colons:

Method overloading

```
class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    void Init(int numRows, int numCols, double value);
    double &Elem(int i, int j);
};
```

Two methods (or functions) may have the same name if their arguments differ in number or type. This is called function overloading. A different return type is not sufficient.

Constructor

```
class MatrixClass
{
  public:
    MatrixClass();
    MatrixClass(int numRows, int numCols);
    MatrixClass(int numRows, int numCols, double value);
};
```

- Every class has some special methods without return type, the constructors and the destructor.
- Constructors are executed, when an object of the class is defined, before any other methods can be called or attributes can be used. There can be more than one constructor if the argument lists are different.
- A default constructor (without arguments) is generated automatically. If a constructor with arguments is specified, the default constructor is *not* generated.
- The constructors must be public.

```
class MatrixClass
{
  public:
    MatrixClass()
    {
        // some code to execute at initialization
    };
};

MatrixClass::MatrixClass(int numRows, int numCols) :
    a_(numRows,std::vector<double> (numCols)),
    numRows_(numRows),
    numCols_(numCols)
{
    // some other code to execute at initialization
}
```

Like a normal method constructors can be specified either in the class declaration or seperately. Constructors can also be used to initialize attributes with values.

Destructor

```
class MatrixClass
{
  public:
    ~MatrixClass();
};
```

- There is always only one destructor, which is called when an object of the class is destroyed.
- The destructor never has arguments.
- In this lecture we will not need to write a destructor of our own. This is only necessary if dynamic memory is allocated.
- The destructor must be public.

Complete Example: Header File `matrix.h`

```
#include<vector>

class MatrixClass
{
  public:
    void Resize(int numRows, int numCols);
    void Resize(int numRows, int numCols, double value);
    double &Elem(int i, int j);
    void Print();
    int Rows()
    {
        return numRows_;
    }
    int Cols()
```

```
    {
        return numCols_;
    }

    MatrixClass() : a_(0),
            numRows_(0),
            numCols_(0)
    {};
```

Complete Example: Header File `matrix.h` (II)

```
    MatrixClass(int numRows, int numCols) :
            a_(numRows),
            numRows_(numRows),
            numCols_(numCols)
    {
        for (int i=0;i<numRows_;++i)
            a_[i].resize(numCols_);
    };

    MatrixClass(int numRows, int numCols, double value)
    {
        Resize(numRows,numCols,value);
    };

  private:
    std::vector<std::vector<double> > a_;
    int numRows_;
    int numCols_;
};
```

Complete Example: Source File `matrix.cc`

```
#include "matrix.h"
#include<iomanip>
#include<iostream>
#include<cstdlib>

void MatrixClass::Resize(int numRows, int numCols)
{
    a_.resize(numRows);
    for (int i=0;i<a_.size();++i)
        a_[i].resize(numCols);
    numRows_=numRows;
    numCols_=numCols;
}

void MatrixClass::Resize(int numRows, int numCols, double value)
{
    a_.resize(numRows);
    for (int i=0;i<a_.size();++i)
    {
        a_[i].resize(numCols);
        for (int j=0;j<a_[i].size();++j)
            a_[i][j]=value;
    }
    numRows_=numRows;
    numCols_=numCols;
```

Complete Example: Source File `matrix.cc` (II)

```
double &MatrixClass::Elem(int i, int j)
```

```
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    else if ((j<0)||(j>=numCols_))
    {
        std::cerr << "Illegal␣column␣index␣" << j;
        std::cerr << "␣valid␣range␣is␣(0:" << numCols_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    else
        return a_[i][j];
```

Complete Example: Source File matrix.cc (III)

```
void MatrixClass::Print()
{
    std::cout << "(" << numRows_ << "x";
    std::cout << numCols_ << ")␣matrix:" << std::endl;
    for (int i=0;i<numRows_;++i)
    {
        std::cout << std::setprecision(1);
        for (int j=0;j<numCols_;++j)
            std::cout << std::setw(5) << a_[i][j] << "␣";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

Complete Example: Main File gaussmatrix.cc

```
#include "matrix.h"
#include<iostream>

std::vector<double> Solve(MatrixClass A, std::vector<double> b)
{
    const int n=A.Rows();
    for (int m=0;m<n-1;++m)
        for (int i=m+1;i<n;++i)
        {
            double q = A.Elem(i,m)/A.Elem(m,m);
            A.Elem(i,m) = 0.0;
            for (int j=m+1;j<n;++j)
                A.Elem(i,j) = A.Elem(i,j)-q*A.Elem(m,j);
            b[i] -= q*b[m];
        }
    std::vector<double> x(b);
    x.back()/=A.Elem(n-1,n-1);
    for (int i=n-2;i>=0;--i)
    {
        for (int j=i+1;j<n;++j)
            x[i] -= A.Elem(i,j)*x[j];
        x[i]/=A.Elem(i,i);
    }
    return(x);
}
```

Complete Example: Main File gaussmatrix.cc (II)

```
int main()
{   // define matrix
    MatrixClass A(5,5,0.0);
    for (int i=0;i<A.Rows();++i)
        A.Elem(i,i) = 2.0;
    for (int i=0;i<A.Rows()-1;++i)
        A.Elem(i+1,i) = A.Elem(i,i+1) = -1.0;
    // print matrix
    A.Print();
    // define vector b
    std::vector<double> b(5);
    b[0] = b[4] = 5.0;
    b[1] = b[3] = -4.0;
    b[2] = 4.0;
    // solve
    std::vector<double> x = Solve(A,b);
    std::cout << "The solution with the ordinary Gauss Elimination is: ";
    for (int i=0;i<x.size();++i)
        std::cout << x[i] << "  ";
    std::cout << std::endl;
}
```

Complete Example: Output

```
(5x5) matrix:
   2    -1     0     0     0
  -1     2    -1     0     0
   0    -1     2    -1     0
   0     0    -1     2    -1
   0     0     0    -1     2

The solution with the ordinary Gauss Elimination is: 3  1  3  1  3
```

Homework

1. Study the class MatrixClass.
2. Compile the example
3. Change the sample program to create a scalable test:
   - Create a $n \times n$ matrix A like the one you used last week with $a_{ij} = \begin{cases} 2 & \text{if } i = j \\ -1 & \text{if } |i-j| = 1 \\ 0 & \text{else} \end{cases}$
   - Multiply the matrix with a vector $\vec{x}$ of length n with $x_i = \begin{cases} 3 & \text{if } i \text{ odd} \\ 1 & \text{if } i \text{ even} \end{cases}$ to obtain $\vec{b} = A\vec{x}$
   - Solve the linear equation system $A\vec{y} = \vec{b}$ and check if $\vec{x} = \vec{y}$
   - Of course for $n = 5$ you should get the same vector $\vec{b}$ as last week

23 Classes

Example: Bank Account

```
#include<iostream>

class Account
{
  public:
    Account(int seedCapital);
    ~Account();
    int balance();
    int withDraw(int amount);
    int deposit(int amount);
  private:
    int balance_;
};
```

```
Account::Account(int seedCapital)
{
    balance_=seedCapital;
    std::cout << "Account␣with␣" << balance_ << "␣created" << std::endl;
}

Account::~Account()
{
    std::cout << "Account␣with␣" << balance_ << "␣dissolved" << std::endl;
}

int Account::balance()
{
    return balance_;
}

int Account::withDraw(int amount)
{
    balance_ -= amount;
    return balance_;
}

int Account::deposit(int amount )
{
    balance_ += amount;
    return balance_;
}
```

```
int main()
{
    Account account1(100), account2(200);
    account1.withDraw(50);
    account2.deposit(300);
    std::cout << "Current␣balance␣of␣account2␣is␣"
              << account2.balance() << std::endl;
    account2.withDraw(600);
}
```

Pivoting

- Simple Gauss Elimination fails if the diagonal element is zero. This does not mean in any case that the system is not solvable.
- *Solution*: Search for a element in the remaining rows and columns which is not zero. Exchange rows and or columns of the equation system (solution is unchanged under these transformations)
- Pivoting can also reduce extinction effects. It is helpfull to first scale the equation system by multiplication with the diagonal matrix $Ax = b \to DAx = Db$, with :

$$d_{ii} = \left(\sum_{j=0}^{n-1} |a_{ij}| \right)^{-1} \tag{1}$$

  to get comparable coefficients.
- The exchanges involve a reordering of the unknowns. This has to be reversed to get the correct solution.

Pivoting Strategies

Strategies for Pivoting:

Column pivoting In elimination step k search $\max(|a_{ik}| \ |i \geq k)$. Exchange rows i and k to make a_{ik} the diagonal element.

Total pivoting In elimination step k search $\max(|a_{ij}| \ |i,j \geq k)$. Exchange rows i and k and columns j and k to make a_{ij} the diagonal element.

LU-Decomposition

The Gaussian Elimination can be rewritten as a decomposition of the matrix A in a lower diagonal matrix L and an upper diagonal matrix U

$$Ax = L \cdot U \cdot x = b$$

The elements of L above the diagonal are zero, the elements on the diagonal are $l_{ii} = 1$. The elements below the diagonal are the multiplication factors occuring in the gaussian elimination:

```
const int n=A.size();
for (int m=0;m<n-1;++m)
    for (int i=m+1;i<n;++i)
    {
        double q = A[i][m]/A[m][m];
        for (int j=m+1;j<n;++j)
            A[i][j] = A[i][j]-q*A[m][j];
        L[i][m] = q;
    }
```

As the diagonal elements of L are always 1 and the lower diagonal elements of A after the gaussian elimination are always 0, the coefficients of L can also be stored in A below the diagonal to save memory.

Solving after LU-Decomposition

While the LU-Decomposition needs $O(N^3)$ operations, the solution of $L \cdot U \cdot x = b$ for a different right hand side b needs only $O(N^2)$ operations. First the equation system $L \cdot d = b$ is solved, then $U \cdot x = d$.

```
const int n=A.size();
x.front()/=1.0;
std::vector<double> x(b);
for (int m=0;m<n-1;++m)
{
    for (int i=m+1;i<n;++i)
        x[i] -= L[i][m]*x[m];
    x[i]/=1.0;
}
x.back()/=A[n-1][n-1];
for (int i=n-2;i>=0;--i)
{
    for (int j=i+1;j<n;++j)
        x[i] -= A[i][j]*x[j];
    x[i]/=A[i][i];
}
```

25 Tridiagonal Matrices

Gauï¿½-Elimination for Tridiagonal Matrices

Problem to solve:

$$\begin{array}{rrrrcr} 5x & -3y & & & = & 7 \\ x & +4y & -2z & & = & 6 \\ & -y & +3z & +w & = & -4 \\ & & 2z & +w & = & -15 \end{array}$$

Matrix represenation:

$$\begin{pmatrix} 5 & -3 & 0 & 0 & : & 7 \\ 1 & 4 & -2 & 0 & : & 6 \\ 0 & -1 & 3 & 1 & : & -4 \\ 0 & 0 & 2 & 1 & : & -15 \end{pmatrix}$$

Gauï¿½-Elimination for Tridiagonal Matrices

Transformations:

$$z_2 - \frac{1}{5} z_1 = \begin{pmatrix} 0 & \frac{23}{5} & -2 & 0 & : & \frac{23}{5} \end{pmatrix}$$

$$z_3 - \frac{(-1)}{\left(\frac{23}{5}\right)} z_2' = \begin{pmatrix} 0 & 0 & \frac{59}{23} & 1 & : & -3 \end{pmatrix}$$

$$z_4 - \frac{2}{\left(\frac{59}{23}\right)} z_3' = \begin{pmatrix} 0 & 0 & 0 & \frac{249}{59} & : & -\frac{747}{59} \end{pmatrix}$$

Upper Diagonal Matrix:

$$\Rightarrow \begin{pmatrix} 5 & -3 & 0 & 0 & : & 7 \\ 0 & \frac{23}{5} & -2 & 0 & : & \frac{23}{5} \\ 0 & 0 & \frac{59}{23} & 1 & : & -3 \\ 0 & 0 & 0 & \frac{249}{59} & : & -\frac{747}{59} \end{pmatrix}$$

Gauï¿½-Elimination for Tridiagonal Matrices

Solutions:

$$\begin{aligned} w &= -\frac{747}{59}\left(\frac{59}{249}\right) = -3 \\ z &= \frac{23}{59}\left(-3 - 1(-3)\right) = 0 \\ y &= \frac{5}{23}\left(\frac{23}{5} - 1(-2)\cdot 0\right) = 1 \\ x &= \frac{1}{5}\left(7 - (-3)\cdot 1\right) = 2 \end{aligned}$$

For tridiagonal matrices only one multiplication and one addition is needed per reduction step. The same is true for the backwards insertion.

Gauï¿½-Elimination for Tridiagonal Matrices

input n, (l_i), (d_i), (u_i), (b_i)
for i=1:n-1 do
  if $d_{i-1} = 0$ then stop
  $d_i = d_i - (l_{i-1}/d_{i-1}) * u_{i-1}$
  $b_i = b_i - (l_{i-1}/d_{i-1}) * b_{i-1}$
end

$x_{n-1} = b_{n-1}/d_{n-1}$
for i=n-2:0 do
  $x_i = (b_i - u_i * x_{i+1})/d_i$
end
output(x_i)

tridiagmatrix.h

```
#include<vector>

class TridiagMatrixClass
{
  public:
    void Init(int dim, double initialValue=0.0);
    double Get(int i, int j);
    void Set(int i, int j, double value);
    void Print();
    std::vector<double> Solve(std::vector<double> b);
    TridiagMatrixClass()
    {}
    TridiagMatrixClass(int dim) :
        dim_(dim),
        l_(dim),
```

```
        d_(dim),
        u_(dim)
    {}
    TridiagMatrixClass(int dim, double value) :
        dim_(dim),
        l_(dim,value),
        d_(dim,value),
        u_(dim,value)
    {}
```

tridiagmatrix.h

```
  private:
    int dim_;
    std::vector<double> l_;
    std::vector<double> d_;
    std::vector<double> u_;
};
```

tridiagmatrix.cc

```
void TridiagMatrixClass::Init(int dim, double initialValue)
{
    dim_ = dim;
    l_.resize(dim);
    d_.resize(dim);
    u_.resize(dim);
    for (int i=0;i<dim_;++i)
        l_[i] = d_[i] = u_[i] = initialValue;
}

double TridiagMatrixClass::Get(int i, int j)
{
    if (i==j)
        return d_[i];
    else if (i==j+1)
        return l_[i];
    else if (i==j-1)
        return u_[i];
    else
        return (0);
}
```

tridiagmatrix.cc

```
void TridiagMatrixClass::Set(int i, int j, double value)
{
    if (i==j)
        d_[i] = value;
    else if (i==j+1)
        l_[i] = value;
    else if (i==j-1)
        u_[i] = value;
    else
    {
        std::cerr << "index␣(" << i << "," << j << ")␣not␣allowed␣";
        std::cerr << "for␣tridiagonal␣matrix!" << std::endl;
    }
}
```

tridiagmatrix.cc

```
void TridiagMatrixClass::Print()
{
    std::cout << "(" << dim_ << "x" << dim_;
    std::cout << ")␣matrix:" << std::endl;
    for (int i=0;i<dim_;++i)
    {
        std::cout << std::setprecision(3);
        for (int j=0;j<dim_;++j)
            std::cout << std::setw(5) << Get(i,j) << "␣";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

tridiagmatrix.cc

```
std::vector<double> TridiagMatrixClass::Solve(std::vector<double> x)
{
    std::vector<double> d(d_);
    for (int i=1;i<dim_;++i)
    {
        if (d[i-1]==0.0)
        {
            std::cerr << "can␣not␣solve␣the␣linear␣equation␣system";
            std::cerr << std::endl;
            exit(EXIT_FAILURE);
        }
        d[i]=d[i]-l_[i]/d[i-1]*u_[i-1];
        x[i]=x[i]-l_[i]/d[i-1]*x[i-1];
    }

    x.back()/=d.back();
    for (int i=dim_-2;i>=0;--i)
        x[i] = (x[i]-u_[i]*x[i+1])/d[i];
    return(x);
}
```

gausstridiag.cc

```
#include<iostream>
#include"tridiagmatrix.h"

int main()
{
    TridiagMatrixClass A(5,0.0);
    for (int i=0;i<5;++i)
        A.Set(i,i,2.0);
    for (int i=0;i<4;++i)
    {
        A.Set(i,i+1,-1.0);
        A.Set(i+1,i,-1.0);
    }
    A.Print();
    std::vector<double> b(5);
    b[0] = b[4] = 5.0;
    b[1] = b[3] = -4.0;
    b[2] = 4.0;
    std::vector<double> x = A.Solve(b);
    std::cout << "The␣solution␣is:␣";
    for (int i=0;i<x.size();++i)
        std::cout << x[i] << "␣␣";
    std::cout << std::endl;
```

```
    A.Print();
}
```

26 Default Methods

Dynamic Storage Allocation

- Besides local variables, which exist only for the life time of a block or function, there are so called dynamic variables.
- Dynamic variables are created and destroyed explicitly by the programmer
- Dynamic variables don't have names. They can only be accessed with pointers

Dynamic Storage Allocation

- Pointers can be used to dynamically allocate memory from the computer using the keyword `new`, e.g.

  ```
  double *a = new double;
  int *b = new int[32];
  ```

  The second line reserves an array with 32 elements.
- If the memory is no longer needed it has to be freed with the keyword `delete` followed by the variable name. If an array is freed, rectangular brackets are placed between `delete` and the variable name, e.g.

  ```
  delete a;
  delete [] b;
  ```

Dangers with Dynamic Storage Allocation

- If a pointer variable is deleted (e.g. because the end of the function is reached in which it was defined) before the memory is freed again, it is still blocked by the program, but no longer accessible (and the available memory gets smaller by this amount)
- If you write a class which has pointers as attributes using dynamic memory allocation you have to write a destructor which frees the allocated memory if the object is deleted.
- The biggest danger with pointers is that you can change then (accidentally or on purpose) to point on regions of the memory, which you did not allocate before and therefore are not allowed to access. In the best case this results in your program terminating with a *segmentation fault* in the worst case it can be used by viruses to corrupt your system. Dealing with pointers is the most difficult task when programming with C.

Dynamic Storage Allocation Today

Today dynamic memory allocation is much less important, as most things can be done better and safer by using STL-containers. If you write a new container it is still very important.

Default Methods

For each `class T` the compiler automatically creates five methods if they are not defined by the user:

- Constructor without arguments: `T();` (recursively calls constructors of attributes)
- Copy Constructor: `T(const T&);` (memberwise copy)
- Destructor: `\~T();` (recursively calls destructors of attributes)
- Assignment operator: `T &operator= (const T&);` (memberwise copy)
- Address operator: `int operator& ();` (returns address of object)

Copy Constructor and Assignment Operator

```
class MatrixClass
{
  public:
    // assigment operator
    MatrixClass &operator=(MatrixClass &A);
    // copy constructor
    MatrixClass(const MatrixClass &A);
    MatrixClass(int i, int j, double value);
};

int main()
{
    MatrixClass A(4,5,0.0);
    MatrixClass B = A; // copy constructor
    A = B; // assignment operator
}
```

- The copy constructor is called when a new object is created, which's content is copied from an existing object.
- The assignment operator is called when an object is assigned a new value.
- A default version for both is created by the compiler.

27 Constant Objects

Constant Objects

- If objects are defined to be constant, (e.g. `const TridiagMatrixClass A(B)` or `int f(const MatrixClas` only member functions which do not change the content of the object can be called.
- The compiler assumes that all methods could possibly alter the content of the object
- Methods which do not alter the object have to be marked by writting `const` after the method name and argument list in the class declaration `double Get(int i, int j) const;`
- The specifier `const` is part of the function name. A const and non-const version with the same name and the same arguments is allowed and often necessary: `double Get(int i, int j) const`
- The compiler complains about methods which are marked `const` and return a non-const reference to an attribute or alter the content of the object

Constant Objects

- As it is part of the function name the `const` has also to be given if the method is *defined* outside the class

```
// can be called for constant objects
// can not modify content
double MatrixClass::Get(int i, int j) const
{
    return a_[i][j];
}

// can only be called for non-constant objects
// can modify content
double &MatrixClass::Get(int i, int j)
{
    return a_[i][j];
}
```

28 Operators

Operator Overloading

- In C++ it is possible to define the behaviour of operators like + or − for user defined objects.
- Operators are defined like ordinary functions. The function name is `operator` followed by the operator symbol e.g. `operator+`
- As any ordinary method an operator has an return type and arguments:

  `MatrixClass operator+(MatrixClass A);`
- Operators can either be defined as methods of an object or as non-member functions.

Unary Operators

```
class MatrixClass
{
  public:
    MatrixClass operator-();
    MatrixClass (int);
};

MatrixClass operator+(MatrixClass A);
```

- Unary operators are: `++ -- + - ! ~ & *`
- An unary operator can either be defined as a member function without argument or as an non-member function with one argument.
- One has to choose which of the two possibilities are realized. The two versions with the same arguments can not be distinguished by the compiler e.g. `MatrixClass operator+(MatrixClass A)` and `MatrixClass MatrixClass::operator+()`

Binary Operators

```
class MatrixClass
{
  public:
    MatrixClass operator+(MatrixClass A);
    MatrixClass (int);
};

MatrixClass operator+(MatrixClass A, MatrixClass B);
```

- A binary operator can either be defined as a member function with one argument or as an non-member function with two arguments.
- Possible binary operators are: `* / % + - & ^ | < > <= >= == != && || >> <<`
- The operators which modify an object `+= -= /= *= %= &= ^= |=` can only be realized as member functions.

Binary Operators

- If an arithmetic operator takes arguments of a different type it is only valid for this sequence of arguments, e.g. with `MatrixClass operator*(MatrixClass A, double b)` you can write `A = A * 2.1` but not `A = 2.1 * A`
- There is an easy trick to implement both efficiently: you define an `operator*=` inside the class and two non-member functions with two arguments which use this operator.

Increment and Decrement

- Prefix and postfix versions of increment and decrement are available
- The postfix version (`a++`) is defined as `operator++(int)`, while the prefix version is defined as `operator++()`. The `int` argument of the postfix version is not used it is only necessary to distiguish the to variants.

```
class Ptr_to_T
{
    T *p;

  public:
    Ptr_to_T &operator++();     // prefix version
    Ptr_to_T operator++(int);   // postfix version

    Ptr_to_T &operator--();     // prefix version
    Ptr_to_T operator--(int);   // postfix version
}
```

The bracket operators

```
class MatrixClass
{
  public:
    double &operator()(int i, int j);
    std::vector<double> &operator[](int i);
    MatrixClass (int);
};
```

- The rounded and rectangular brackets operator can also be overloaded. This enables us to write things like `A[i][j]=12` or `A(i,j)=12`
- The rectangular bracket operator always takes only one element.
- The rounded bracket operator can take an arbitrary number of arguments

Assignment Operator

```
class MatrixClass
{
  public:
    MatrixClass &operator=(MatrixClass &A);
};
```

- The assignment operator is called when an object is assigned a new value.
- A default version is created by the compiler.

Conversion Operators

- Conversion operators are used to convert user defined variables to one of the built in types.
- The name of a conversion operator is `operator` plus the name of the variable type to which the operator converts.
- Conversion operators have an empty argument list
- Conversion operators are constant methods.

Conversion Operators (II)

```
#include<iostream>
#include<cmath>

class Complex
{
  public:
    operator double() const
    {
        return sqrt(re_*re_+im_*im_);
    }
    Complex(double real, double imag) : re_(real), im_(imag)
    {};
  private:
    double re_;
```

```
    double im_;
};

int main()
{
    Complex a(2.0,-1.0);
    double b = 2.0 * a;
    std::cout << b << std::endl;
}
```

Self-Reference

- Each member function knows from what object it was called.
- Each member function can pass a reference to its object
- The name of the reference is `*this`
- The self-reference is necessary e.g. for operators which modify an object:

```
MatrixClass &MatrixClass::operator*=(double x)
{
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]*=x;
    return *this;
}
```

29 Example Improved Matrix Class

Example: Improved Matrix Class

This example realizes an improved matrix class and includes a simple test application.

- `matrix.h`: contains the definition of the matrix class
- `matrix.cc`: contains the implementation of the methods of `MatrixClass`
- `main.cc`: is a sample application demonstrating the use of `MatrixClass`

Example: Improved MatrixClass Header

```
#include<vector>

class MatrixClass
{
  public:
    void Resize(int numRows, int numCols);
    void Resize(int numRows, int numCols, double value);
    // access elements
    double &operator()(int i, int j);
    double operator()(int i, int j) const;
    std::vector<double> &operator[](int i);
    const std::vector<double> &operator[](int i) const;
    // arithmetic functions
    MatrixClass &operator*=(double x);
    MatrixClass &operator+=(const MatrixClass &b);
    std::vector<double> Solve(std::vector<double> b) const;
    // output
    void Print() const;
```

```
    int Rows() const
    {
        return numRows_;
    }
    int Cols() const
    {
        return numCols_;
    }

    MatrixClass() : a_(0), numRows_(0), numCols_(0)
    {};

    MatrixClass(int dim) : a_(dim), numRows_(dim), numCols_(dim)
    {
        for (int i=0;i<numRows_;++i)
            a_[i].resize(numCols_);
    };

    MatrixClass(int numRows, int numCols) :
            a_(numRows), numRows_(numRows), numCols_(numCols)
    {
        for (int i=0;i<numRows_;++i)
            a_[i].resize(numCols_);
    };

    MatrixClass(int numRows, int numCols, double value)
    {
        Resize(numRows,numCols,value);
    };

    MatrixClass(std::vector<std::vector<double> > a)
    {
        a_=a;
        numRows_=a.size();
        if (numRows_>0)
            numCols_=a[0].size();
        else
            numCols_=0;
    }

    MatrixClass(const MatrixClass &b)
    {
        a_=b.a_;
        numRows_=b.numRows_;
        numCols_=b.numCols_;
    }

  private:
    std::vector<std::vector<double> > a_;
    int numRows_;
    int numCols_;
};

std::vector<double> operator*(const MatrixClass &a,
                              const std::vector<double> &x);
MatrixClass operator*(const MatrixClass &a,double x);
MatrixClass operator*(double x,const MatrixClass &a);
MatrixClass operator+(const MatrixClass &a,const MatrixClass &b);
```

Example: Improved MatrixClass Source

```
#include "matrixnew.h"
#include<iomanip>
#include<iostream>
```

```
void MatrixClass::Resize(int numRows, int numCols)
{
    a_.resize(numRows);
    for (int i=0;i<a_.size();++i)
        a_[i].resize(numCols);
    numRows_=numRows;
    numCols_=numCols;
}

void MatrixClass::Resize(int numRows, int numCols, double value)
{
    a_.resize(numRows);
    for (int i=0;i<a_.size();++i)
    {
        a_[i].resize(numCols);
        for (int j=0;j<a_[i].size();++j)
            a_[i][j]=value;
    }
    numRows_=numRows;
    numCols_=numCols;
}

double &MatrixClass::operator()(int i,int j)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if ((j<0)||(j>=numCols_))
    {
        std::cerr << "Illegal␣column␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numCols_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i][j];
}

double MatrixClass::operator()(int i,int j) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if ((j<0)||(j>=numCols_))
    {
        std::cerr << "Illegal␣column␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numCols_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i][j];
}

std::vector<double> &MatrixClass::operator[](int i)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
```

```
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}

const std::vector<double> &MatrixClass::operator[](int i) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}

MatrixClass &MatrixClass::operator*=(double x)
{
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]*=x;
    return *this;
}

MatrixClass &MatrixClass::operator+=(const MatrixClass &x)
{
    if ((x.numRows_!=numRows_)||(x.numCols_!=numCols_))
    {
        std::cerr << "Dimensions␣of␣matrix␣a␣(" << numRows_
                  << "x" << numCols_ << ")␣and␣matrix␣x␣("
                  << numRows_ << "x" << numCols_ << ")␣do␣not␣match!";
        exit(EXIT_FAILURE);
    }
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<x.numCols_;++j)
            a_[i][j]+=x[i][j];
    return *this;
}

std::vector<double> MatrixClass::Solve(std::vector<double> b) const
{
    std::vector<std::vector<double> > a(a_);
    for (int m=0;m<numRows_-1;++m)
        for (int i=m+1;i<numRows_;++i)
        {
            double q = a[i][m]/a[m][m];
            a[i][m] = 0.0;
            for (int j=m+1;j<numRows_;++j)
                a[i][j] = a[i][j]-q*a[m][j];
            b[i] -= q*b[m];
        }
    std::vector<double> x(b);
    x.back()/=a[numRows_-1][numRows_-1];
    for (int i=numRows_-2;i>=0;--i)
    {
        for (int j=i+1;j<numRows_;++j)
            x[i] -= a[i][j]*x[j];
        x[i]/=a[i][i];
    }
    return(x);
}

void MatrixClass::Print() const
{
```

```
    std::cout << "(" << numRows_ << "x";
    std::cout << numCols_ << ")␣matrix:" << std::endl;
    for (int i=0;i<numRows_;++i)
    {
        std::cout << std::setprecision(3);
        for (int j=0;j<numCols_;++j)
            std::cout << std::setw(5) << a_[i][j] << "␣";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

MatrixClass operator*(const MatrixClass &a,double x)
{
    MatrixClass temp(a);
    temp *= x;
    return temp;
}

MatrixClass operator*(double x,const MatrixClass &a)
{
    MatrixClass temp(a);
    temp *= x;
    return temp;
}

std::vector<double> operator*(const MatrixClass &a,
                              const std::vector<double> &x)
{
    if (x.size()!=a.Cols())
    {
        std::cerr << "Dimensions␣of␣vector␣" << x.size();
        std::cerr << "␣and␣matrix␣" << a.Cols() << "␣do␣not␣match!";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    std::vector<double> y(a.Rows());
    for (int i=0;i<a.Rows();++i)
    {
        y[i]=0.0;
        for (int j=0;j<a.Cols();++j)
            y[i]+=a[i][j]*x[j];
    }
    return y;
}

MatrixClass operator+(const MatrixClass &a,const MatrixClass &b)
{
    MatrixClass temp(a);
    temp += b;
    return temp;
}
```

Example: Improved MatrixClass Application

```
#include "matrixnew.h"
#include<iostream>

int main()
{   // define matrix
    MatrixClass A(4,6,0.0);
    for (int i=0;i<A.Rows();++i)
        A[i][i] = 2.0;
    for (int i=0;i<A.Rows()-1;++i)
        A[i+1][i] = A[i][i+1] = -1.0;
```

```
    MatrixClass B(6,4,0.0);
    for (int i=0;i<B.Cols();++i)
        B[i][i] = 2.0;
    for (int i=0;i<B.Cols()-1;++i)
        B[i+1][i] = B[i][i+1] = -1.0;
    // print matrix
    A.Print();
    B.Print();
    MatrixClass C(A);
    A = 2*C;
    A.Print();
    A = C*2.;
    A.Print();
    A = C+A;
    A.Print();

    const MatrixClass D(A);
    std::cout << "Element 1,1 of D is " << D(1,1) << std::endl;
    std::cout << std::endl;
    A.Resize(5,5,0.0);
    for (int i=0;i<A.Rows();++i)
        A(i,i) = 2.0;
    for (int i=0;i<A.Rows()-1;++i)
        A(i+1,i) = A(i,i+1) = -1.0;
    // define vector b
    std::vector<double> b(5);
    b[0] = b[4] = 5.0;
    b[1] = b[3] = -4.0;
    b[2] = 4.0;
    std::vector<double>x = A*b;
    std::cout << "A*b = ( ";
    for (int i=0;i<x.size();++i)
        std::cout << x[i] << "  ";
    std::cout << ")" << std::endl;
    std::cout << std::endl;
    // solve
    x = A.Solve(b);
    A.Print();
    std::cout << "The solution with the ordinary Gauss Elimination is: ( ";
    for (int i=0;i<x.size();++i)
        std::cout << x[i] << "  ";
    std::cout << ")" << std::endl;
}
```

Example: Improved MatrixClass Output

```
(4x6) matrix:
    2    -1     0     0     0     0
   -1     2    -1     0     0     0
    0    -1     2    -1     0     0
    0     0    -1     2     0     0

(6x4) matrix:
    2    -1     0     0
   -1     2    -1     0
    0    -1     2    -1
    0     0    -1     2
    0     0     0     0
    0     0     0     0

(4x6) matrix:
    4    -2     0     0     0     0
   -2     4    -2     0     0     0
    0    -2     4    -2     0     0
    0     0    -2     4     0     0
```

```
(4x6) matrix:
    4     -2      0      0      0      0
   -2      4     -2      0      0      0
    0     -2      4     -2      0      0
    0      0     -2      4      0      0
```

Example: Improved MatrixClass Output

```
(4x6) matrix:
    6     -3      0      0      0      0
   -3      6     -3      0      0      0
    0     -3      6     -3      0      0
    0      0     -3      6      0      0

Element 1,1 of D is 6

A*b = ( 14   -17   16   -17   14   )

(5x5) matrix:
    2     -1      0      0      0
   -1      2     -1      0      0
    0     -1      2     -1      0
    0      0     -1      2     -1
    0      0      0     -1      2

The solution with the ordinary Gauss Elimination is: ( 3  1  3  1  3  )
```

Homework

In Assignment 2 you have defined a class `Polynom`.

- Add a unary `operator-` to your class `Polynom`
- Add binary `operator+`, `operator+=`, `operator-`, `operator-=`, `operator*` and `operator*=` to your class `Polynom`. Carefull: you have to check the degree of the resulting polynomial...
- Add a binary `operator*` for the multiplication of a polynomial with a `double` value from both sides.

30 Preprocessor

Preprocessor Macros

- Before the actual program is translated to machine language by the compiler the so-called preprocessor is executed
- The preprocessor includes the header file
- It can manipulate text files by the definitions of macros
- The syntax for defining a macro is `#define <macroname> <commands>` e.g. `#define PI 3.1415`
- In C++ macros should not be used any more to define constants or code
- Certain parts of the program can be made to be only translated if a certain macro is defined

```
#ifdef _MYMACRO_
// some code
#endif
```

- The conditions `#ifdef` and its negation `#ifndef` exist

Multiple Header Inclusion Prevention

- Macros can be used to prevent the multiple inclusion of the same header file
- The content of the header file is placed into a condition block:

```
#ifndef _MYMACRO_
#define _MYMACRO_
// content of header file
#endif
```

- Thus the content of the header file is read once and the macro is defined. Afterwards the content is skipped, as the macro already exitst

31 Inheritance

Inheritance

- Inheritance is used if two classes do nearly the same but with small differences or to extend the functionality of an existing class.
- Inheritance means that a class gets all the methods and attributes another class has already defined.
- Inheritance is a way to
  - improve code reuse
  - reduce the possibility of errors (especially copy and paste errors)
  - unify interfaces
  - extend classes, where the implementation is only available as binary

Inheritance

```
class B : public A
{
}
```

- Class B can be derived from another class A (called base class), which means that it gets all the attributes and methods of class A.
- If the inheritance is public, all public methods of class A are also public methods of class B.
- If the inheritance is private, all public methods of class A are private methods of class B.
- Private members of class A can *never* be accessed from class B.
- Functions of the base class can be redefined in the derived class.

Protected Members

```
class A
{
    public:
        ...
    private:
        ...
    protected:
        int c;
}

class B : public A
{ }
```

- Apart from private and public members, there is a third type: protected members.
- Protected members can only be accessed from the class itself and from derived classes.
- Protected members of class A like c can be accessed from class B, but not from outside the class.

Initialisation of derived classes

```
class B : public A
{
    B(double c) : A(c)
    {};
}
```

If a base class must be initialised, this is done in the constructor in the same way as the initialisation of attributes. The base class must be the first element in the initialisation list. In this case the constructor of B is called with the argument c.

Passing derived Objects

Objects of a derived class can be passed to functions which expect an object of the base class as argument, but

- only the base class part of the object is copied if the function is called by value
- only the base class part is accessible over the reference if the object is called by reference

This means in particular that for functions which have been redefined in the derived class the version of the base class is called.

32 Inheritance

Inheritance

- Inheritance is used if two classes do nearly the same but with small differences or to extend the functionality of an existing class.
- Inheritance means that a class gets all the methods and attributes another class has already defined.

- Inheritance is a way to
  - improve code reuse
  - reduce the possibility of errors (especially copy and paste errors)
  - unify interfaces
  - extend classes, where the implementation is only available as binary

Inheritance

```
class B : public A
{
}
```

- Class B can be derived from another class A (called base class), which means that it gets all the attributes and methods of class A.
- If the inheritance is public, all public methods of class A are also public methods of class B.
- If the inheritance is private, all public methods of class A are private methods of class B.
- Private members of class A can *never* be accessed from class B.
- Functions of the base class can be redefined in the derived class.

Protected Members

```
class A
{
    public:
        ...
    private:
        ...
    protected:
        int c;
}

class B : public A
{ }
```

- Apart from private and public members, there is a third type: protected members.
- Protected members can only be accessed from the class itself and from derived classes.
- Protected members of class A like c can be accessed from class B, but not from outside the class.

Initialisation of derived classes

```
class B : public A
{
    B(double c) : A(c)
    {};
}
```

If a base class must be initialised, this is done in the constructor in the same way as the initialisation of attributes. The base class must be the first element in the initialisation list. In this case the constructor of B is called with the argument c.

33 Virtual Functions

Passing derived Objects

Objects of a derived class can be passed to functions which expect an object of the base class as argument, but

- only the base class part of the object is copied if the function is called by value
- only the base class part is accessible over the reference if the object is called by reference

This means in particular that for functions which have been redefined in the derived class the version of the base class is called.

Problems with Inheritance

```
#include<iostream>

class A
{
    public:
        int doWork(int a)
        {
            return(a);
        }
};

class B : public A
{
    public:
        int doWork(int a)
        {
            return(a*a);
        }
};

int doSomeOtherWork(A &object)
{
    return(object.doWork(2));
}
```

Problems with Inheritance (II)

```
int main()
{
    A objectA;
    B objectB;
    std::cout << objectA.doWork(2) << ",␣";
    std::cout << objectB.doWork(2) << ",␣";
    std::cout << doSomeOtherWork(objectA) << ",␣";
    std::cout << doSomeOtherWork(objectB) << std::endl;
}
```

Output:

```
2, 4, 2, 2
```

Virtual Functions

```
#include<iostream>

class A
{
    public:
        virtual int doWork(int a)
        {
            return(a);
        }
};

class B : public A
{
    public:
        int doWork(int a)
        {
            return(a*a);
        }
};

int doSomeOtherWork(A &object)
{
    return(object.doWork(2));
}
```

Virtual Functions (II)

```
int main()
{
    A objectA;
    B objectB;
    std::cout << objectA.doWork(2) << ",␣";
    std::cout << objectB.doWork(2) << ",␣";
    std::cout << doSomeOtherWork(objectA) << ",␣";
    std::cout << doSomeOtherWork(objectB) << std::endl;
}
```

Output:

```
2, 4, 2, 4
```

Virtual Functions

- If functions are declared `virtual` in the base class, the function of the derived class is called, even if the object of the derived class is used with a reference of the type of the base class.
- The definition of the virtual function in the derived class must be *exactly* the same as in the base class, else the function is overloaded.
- It is not obligatory to repeat the keyword virtual in the derived class.

34 Interface Base Classes

Interface Classes

```
class BaseClass
{
    public:
        virtual int functionA(double x) = 0;
        virtual void functionB(int y) = 0;
        virtual ~BaseClass()
        {};
}
```

- The purpose of interface base classes (also termed abstract base classes) is to define a common interface for derived classes.
- Interface base classes usually have no attributes (i.e. they contain no data).
- The functions in the interface base class are usually pure virtual, which means that their functionality is only implemented in the derived classes. This is marked by placing `= 0` after the function declaration.
- No objects of an interface base class can be defined if they contain pure virtual functions, only references and pointers.

35 Interpolation

35.1 Interpolation with Polynomials

Interpolation with Polynomials

- Interpolation is a method of constructing new data points from a discrete set of known data points.
- The data points are either from an experiment or from the evaluation of a function.
- Interpolation with Polynomials is especially important as most algorithms in numerical mathematics are based on the analysis of polynomials. Other functions are interpolated or approximated by polynomials.

Polynomial Interpolation

We have a sequence of points $(x_i, y_i), i = 0, \dots n$ and want to determine the coefficients a_j of a polynomial of degree n so that

$$p(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \dots + a_n x_i^n = y_i, \qquad i = 0, \dots, n,$$

To get the coefficients we have to solve the linear equation system

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} \quad .$$

The matrix is called *Vandermonde's*[1] *matrix*. This LES is hard and expensive to solve. For other representations of the polynomial the coefficients are easier to determine.

Lagrange Interpolation

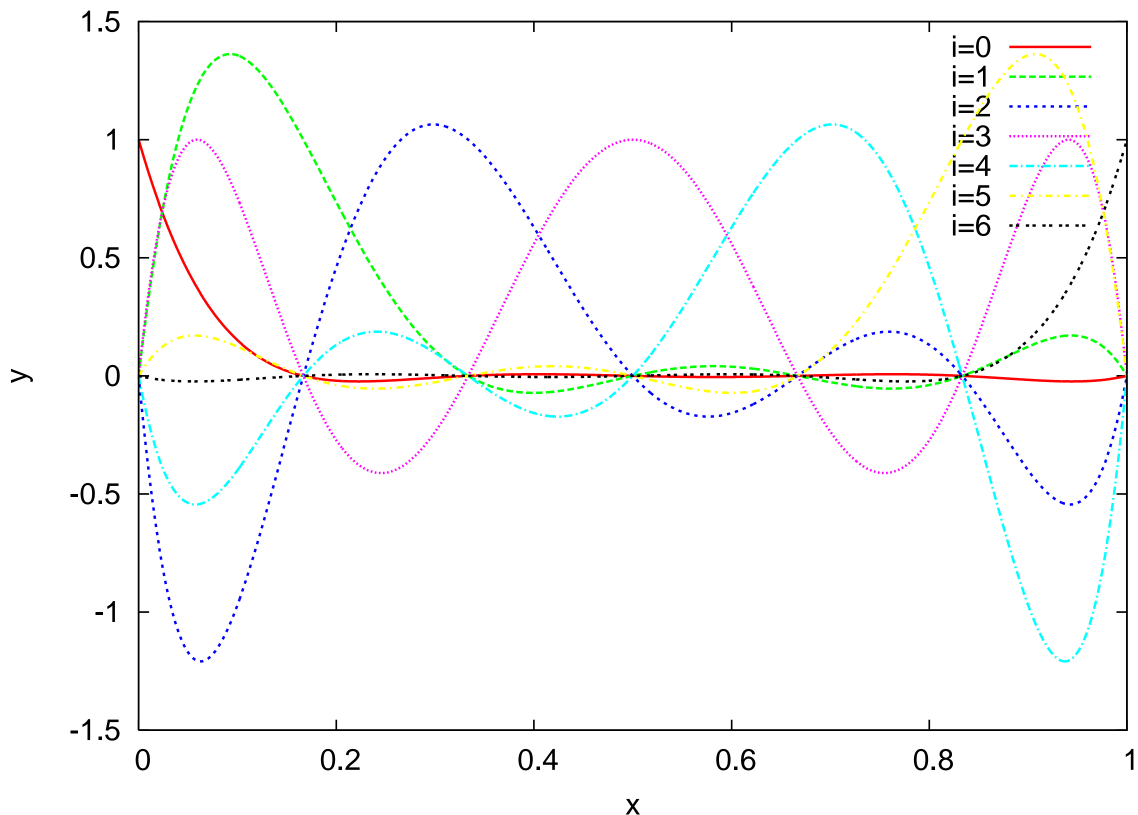
- The Lagrange basis polynomials are defined by

$$L_i^{(n)}(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} \qquad i = 0, \ldots, n$$

- Each of this basis polynomials is a polynom of degree n.
- $L_i^{(n)}(x_k) = \begin{cases} 1 & \text{for } i = k \\ 0 & \text{else} \end{cases}$
- Each polynomial of degree n has a unique representation as a sum of multiples of the Lagrange basis polynomials.
- The interpolation polynomial is then easily obtained by

$$p(x) = \sum_{i=0}^{n} y_i L_i^{(n)}(x)$$

Lagrange Basis Polynomials of Degree 6



[1]Alexandre-Théophile Vandermonde, 1735-1796, french Mathematician.

Approximation Error

If we approximate a function by a polynomial we make an error (of course exactly at the interpolation points the error is zero).

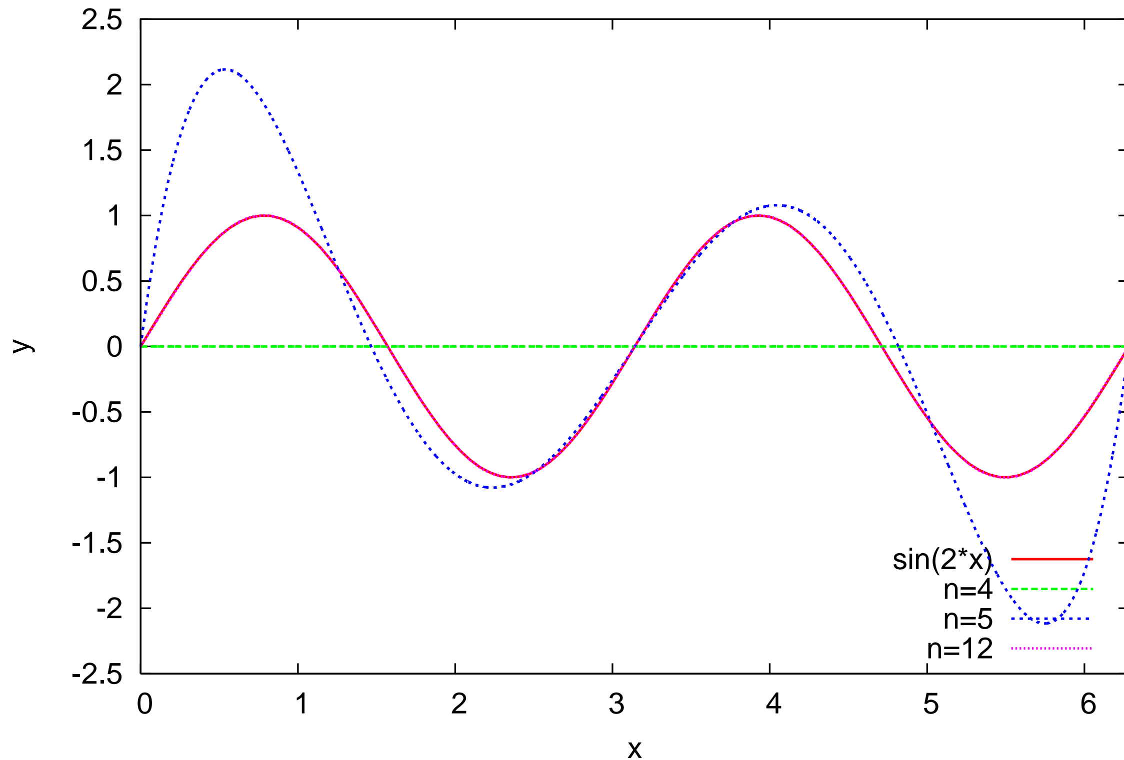
The approximation error can be shown to be

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!} \prod_{j=0}^{n} (x - x_j)$$

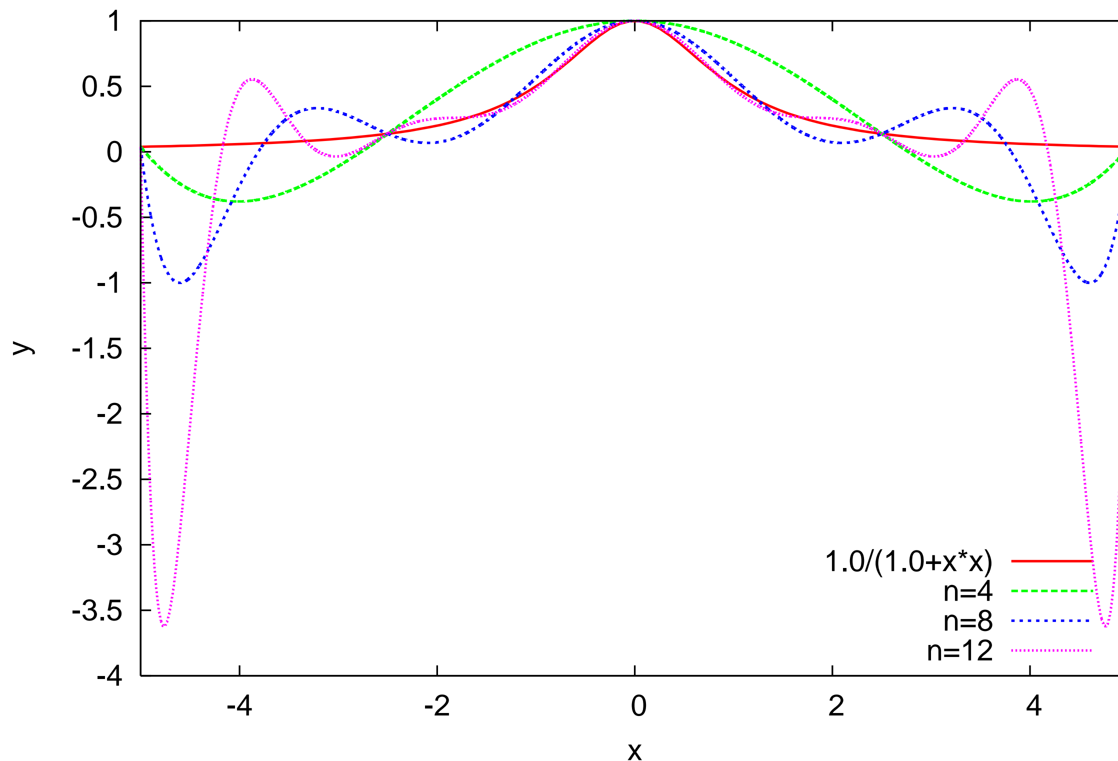
If the sampling points are equidistant and the $n+1$st derivative of f is bounded by M, the approximation error is

$$\begin{aligned} |f(x) - p(x)| &= \frac{\left|f^{(n+1)}(\xi_x)\right|}{(n+1)!} \underbrace{\prod_{j=0}^{n} |x - x_j|}_{\leq h \cdot h \cdot 2h \cdots nh} \\ &\leq \sup_{\xi \in (a,b)} \left|f^{(n+1)}(\xi)\right| \frac{1}{(n+1)!} h^{n+1} n! \\ &= \frac{M}{n+1} h^{n+1} \end{aligned}$$

Interpolation of $\sin(2x)$



Interpolation of $\frac{1}{1+x^2}$



35.2 Example

Example: Lagrange-Interpolation

The example makes an Lagrange interpolation for $\cos(2x+1)$. The files are

- `interpolation.h`: contains the definition of an interface class for interpolation
- `functor.h`: contains the definition of an interface for a functor, i.e. a class which is used to define an pass functions
- `cosine.h`: contains the definition and implementation of a special functor, the $\cos(ax+b)$
- `lagrange.h`: contains the definition of a class, which is a special realization of a interpolation class performing Lagrange interpolation
- `lagrange.cc`: contains the implementation of the methods of `LagrangeClass`
- `main.cc`: is an example program demonstrating the use of `CosinusClass` and `LagrangeClass`

interpolation.h

```
#ifndef INTERPOLATIONCLASS_H
#define INTERPOLATIONCLASS_H

class InterpolationClass
```

```
{
  public:
    virtual ~InterpolationClass()
    {}
    virtual double operator()(double x) = 0;
};

#endif
```

functor.h

```
#ifndef FUNCTORCLASS_H
#define FUNCTORCLASS_H

#include<string>

// Base class for arbitrary functions

class FunctorClass
{
  public:
    virtual ~FunctorClass()
    {}
    virtual double operator()(double x) = 0;
    virtual std::string Name() = 0;
};

#endif
```

cosinus.h

```
#ifndef COSINUSCLASS_H
#define COSINUSCLASS_H

#include <cmath>
#include "functor.h"

class CosinusClass : public FunctorClass
{
  public:
    CosinusClass(double a=1.0, double b=0.0) : a_(a), b_(b)
    {};
    virtual ~CosinusClass()
    {}
    virtual double operator()(double x)
    {
        return cos(a_*x+b_);
    }
    virtual std::string Name()
    {
        return "Cosinus";
    }
  private:
    double a_,b_;
};

#endif
```

lagrange.h

```
#ifndef LAGRANGECLASS_H
#define LAGRANGECLASS_H
```

```
#include <vector>
#include "interpolation.h"
#include "functor.h"

class LagrangeClass : public InterpolationClass
{
  public:
    LagrangeClass(const std::vector<double> &x,const std::vector<double> &y);
    LagrangeClass(FunctorClass &function, double min, double max,
                  int numPoints=4);
    ~LagrangeClass()
    {}
    virtual double operator()(double x);
  private:
    std::vector<double> x_;
    std::vector<double> y_;
};

#endif
```

lagrange.cc

```
#include <iostream>
#include "lagrange.h"

LagrangeClass::LagrangeClass(const std::vector<double> &x,
                             const std::vector<double> &y) : x_(x), y_(y)
{
    if (x.size() != y.size())
    {
        std::cerr << "LagrangeClass:␣Size␣of␣x␣and␣y␣vectors␣does␣not␣match";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    for (int i=1;i<x_.size();++i)
    {
        if (x_[i] <= x_[i-1])
        {
            std::cerr << "LagrangeClass:␣x␣vector␣not␣in␣ascending␣order␣";
            std::cerr << "or␣two␣equal␣x-values" << std::endl;
            exit(EXIT_FAILURE);
        }
    }
}

LagrangeClass::LagrangeClass(FunctorClass &function, double min, double max,
                             int numPoints) : x_(numPoints), y_(numPoints)
{
    if (min>=max)
    {
        std::cerr << "LagrangeClass:␣Interpolation␣range␣less␣or␣equal␣zero";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    double interval=(max-min)/(numPoints-1);
    x_.front()=min;
    for (int i=1;i<(numPoints-1);++i)
        x_[i]=min+i*interval;
    x_.back()=max;
    for (int i=0;i<numPoints;++i)
        y_[i] = function(x_[i]);
}

double LagrangeClass::operator()(double x)
{
```

```
    double result=0.0;
    for (int i=0;i<x_.size();++i)
    {
        double factor=y_[i];
        for (int j=0;j<x_.size();++j)
            if (i!=j)
                factor*=(x-x_[j])/(x_[i]-x_[j]);
        result+=factor;
    }
    return result;
}
```

main.cc

```
#include <fstream>
#include <iostream>
#include "lagrange.h"
#include "cosinus.h"

int main()
{
    CosinusClass cosine(2.0,1.0);
    double min=0.0;
    double max=2.*M_PI;
    int numPoints;
    std::cout << "Please␣enter␣the␣number␣of␣interpolation␣points:␣";
    std::cin >> numPoints;
    LagrangeClass interp(cosine,min,max,numPoints);
    const int numOutputPoints=20*numPoints;
    double interval=(max-min)/(numOutputPoints-1);
    std::ofstream output("cosine.dat");
    for(int i=0;i<numOutputPoints;++i)
        output << min+i*interval << "␣" << interp(min+i*interval) << std::endl;
    output << std::endl;
}
```

36 Namespaces

Namespaces

- Namespaces allow to group classes, objects and functions under a name. This way the global name space can be divided in “sub-spaces”, each one with its own name.
- The format of namespaces is:

  ```
  namespace identifier
  {
  // classes, functions etc. belonging to the namespace
  }
  ```

  Where identifier is any valid identifier.
- The keyword `using` is used to introduce a name from a namespace into the current declarative region. For example `using namespace std;`

Namespace Example

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

```
// namespaces
#include <iostream>

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main ()
{
  std::cout << first::var << endl;
  std::cout << second::var << endl;
  return 0;
}
```

37 Makefiles

Makefiles

- Complex C++ programs often consist of many different header and source files.
- Compilation times for a whole project can be quite large
- It is not necessary to recompile all source files if only one of them was changed
- `make` is a program which makes it possible to compile only the files which are affected by the change
- A `Makefile` describes which files belong to a project and how they are compiled and linked

Makefile Example

```
test_fraction: fraction_test.o fraction.o
        g++ fraction_test.o fraction.o -o test_fraction

fraction_test.o: fraction_test.cc fraction.h
        g++ -c fraction_test.cc

fraction.o: fraction.cc fraction.h
        g++ -c fraction.cc
```

Tutorials on Makefiles

http://www.sethi.org/classes/cet375/lab_notes/ lab_04_makefile_and_compilation.html http://myweb.stedwar
http://mrbook.org/blog/tutorials/make/ http://www.metalshell.com/view/tutorial/120/ http://www.eng.hawaii.edu/Tu

Alternative: IDE's

- Integrated Development Environments (IDE's) combine the features of an editor, a build environment (like make) and a debugger
- Example: Eclipse C/C++ Development Environment (`http://www.eclipse.org/cdt`)
- Eclipse is powerful and free but complex

38 Streams

Streams

Streams are an abstraction for input and output devises.
Input and Output can be read from and written to

- the screen
- a string
- a file

in exactly the same way

The only difference is the variable type of the output device. All output devices are objects of classes derived from the same base class.

- `std::cin` is the predefined device for input from the keyboard
- `std::cout` and `std::cerr` are the predefined devices for output on the screen

File Output

- File output devices can be defined to write data to files.
- These devices are defined like ordinary variables.
- The type of devices for file output is `std::ofstream`.
- The file `fstream` must be included to use file i/o.
- The name of the file to open is either given to the constructor or in the open function.
- A string can not be used directly as a filename. Its method `c_str()` has to be used

```
std::ofstream outfile("output_file");
string filename("testfilename.dat");
std::ofstream outfile2;
outfile2.open(filename.c_str());
```

File Output

- If an output file does not exist, it is created.
- If an output file already exists, it is overwritten.
- To add the output at the end of the file you can add `std::ios::app` as second argument in the `ofstream` constructor: `ofstream outfile("output_file",std::ios::app);`
- You can check if the file opening failed by checking `if (!outfile)`
- Output is written in the same way as on `cout` and `cerr`:
  - Values are written to the stream with `<<`
  - All output flags can also be used for output files.
- Files can be closed with the method `close()` and other files opened with the same `ofstream` variable with `open("newfilename")`
- If a output stream variable is deleted by the compiler (e.g. when leaving a function) the file is closed automatically

File Input

- File input devices can be defined to read data from files.
- The type of devices for file input is `std::ifstream`.
- The file `fstream` must be included to use file i/o.
- The name of the file to open is either given to the constructor or in the open function.

```
std::ifstream infile("input_file");
infile.close();
infile.open("testfilename.dat");
```

File Input

- If an input file does not exist, arbitrary values are read.
- You can check if the file opening or any input operation failed by checking `if (!infile)`
- Input is read in the same way as from `std::cin`:
  - Values are read from the stream with `>>`
  - All input flags can also be used for input files.

File Input/Output

```
#include<cstdlib>
#include<iostream>
#include<fstream>

int main()
{
    std::ifstream infile("input_file");
    if (!infile)
    {
        std::cerr << "Opening␣of␣input␣file␣failed!" << std::endl;
        exit(EXIT_FAILURE);
    }
    int numPoints;
    infile >> numPoints;
    if (!infile)
    {
        std::cerr << "Reading␣from␣input␣file␣failed!" << std::endl;
        exit(EXIT_FAILURE);
    }
    std::ofstream outfile("output_file");
    if (outfile)
        outfile << 5*numPoints << std::endl;
}
```

Input of Lines

- Sometimes it is easier to read whole lines from the stream. This can be done with the function `getline(istream,string)`.
- The first argument of getline is an input stream (either a file stream or `cin`)
- The second argument has to be a string.
- The function returns `true` if the input was successful

Input of Lines containing Spaces

```
#include<iostream>

int main()
{
    std::cout << "Please␣enter␣your␣full␣name:␣";
    std::string name;
    getline(std::cin,name);
    std::cout << "Your␣name␣is␣" << name;
}
```

Input of Lines: Example

```
#include<cstdlib>
#include<iostream>
#include<fstream>

int main()   // copies file linewise and counts the number of characters
{
    std::ifstream infile("geometricalshape.h");
    std::ofstream outfile("output_file");
    if ((!infile)||(!outfile))
```

```
    {
        std::cerr << "Opening␣of␣input␣or␣output␣file␣failed!" << std::endl;
        exit(EXIT_FAILURE);
    }
    std::string line;
    int numChars=0;
    while (getline(infile,line))
    {
        outfile << line << std::endl;
        numChars+=line.size();
    }
    std::cout << numChars << "␣characters␣copied" << std::endl;
}
```

Reading from strings

You can create an `istringstream` to read values from a string. The string from which the values are read from can either be passed to the constructor

```
string string1 = "25";
istringstream stream1(string1);
```

or it can be set with the method `str(string s)`

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
```

If a string stream is to be reused the method `clear()` has to be called before.

You have to `#include<sstream>` to use string streams.

```
#include<iostream>
#include<string>
#include<sstream>

int main()
{
    int x;
    double y;
    std::istringstream s("10␣15␣25");
    for (int i=0;i<3;++i)
    {
        s >> x;
        std::cout << x << "␣squared␣is␣" << x*x << std::endl;
    }
    std::istringstream s2;
    s2.str("10␣15␣25");
    for (int i=0;i<3;++i)
    {
        s2 >> x;
        std::cout << x << "␣squared␣is␣" << x*x << std::endl;
    }
    s.clear();
    s.str("2.2␣4.4␣8.8␣16.6");
    for (int i=0;i<4;++i)
    {
        s >> y;
        std::cout << y << "␣squared␣is␣" << y*y << std::endl;
    }
}
```

Using `istringstream` with line-by-line input

Reading from strings is often used in combination with the getline function to read input line-by-line and process if afterwards

```
#include<iostream>
#include<string>
#include<sstream>

int main()
{
    std::string line;
    std::cout << "Please␣enter␣your␣full␣name:␣";
    getline(std::cin,line);
    std::string name;
    std::istringstream instream(line);
    instream >> name;
    std::cout << "Your␣first␣name␣is␣" << name << std::endl;
    instream >> name;
    std::cout << "Your␣second␣name␣is␣" << name << std::endl;
}
```

Writing to strings

- You can write values to strings like to a file, by defining a variable of type `ostringstream`.
- You can then write to this stream like to `std::cout` using the operator `<<`
- All output flags can be used
- The result is a string, which can be obtained with the method `str()`.

Writing to Strings Example

Writing to strings can be used e.g. to compose filenames:

```
#include<iostream>
#include<string>
#include<sstream>
#include<fstream>

int main()
{
    for (uint i=2;i<6;++i)
    {
        std::ostringstream s1;
        s1 << "dividable_by_" << i;
        std::ofstream outfile(s1.str().c_str());
        for (int j=1;j<=100;++j)
        {
            if ((j%i)==0)
                outfile << j << std:: endl;
        }
    }
}
```

Command Line Arguments

- `int main` can also be defined to have two arguments.
- The first argument is an `int`-variable usually called `argc`, which gives the number of arguments given to the program on the command line.

- The second argument is `char * argv[]`, which is an array of arrays of char (in other words an array of strings). Each of the elements of this array can be accessed with the operator `[]`. For example `std::cout << argv[1] << std::endl` would write the first command line argument to the screen.

- `argv[0]` always contains the name of the program.

- The function `atoi()` can convert command line arguments to integer variables

- The function `atof()` can convert command line arguments to `double` variables

```
// Example for the main routine
#include<cstdlib>
#include<iostream>

int main(int argc, char* argv[])
{
    std::cout << "Name␣of␣the␣program:␣" << argv[0] << std::endl;
    // argv[1] should contain the command line argument n
    if (argc>2)
    {
        int n = atoi(argv[1]);      // atoi() converts argv[1] to int
        double x = atof(argv[2]);   // atof() converts argv[2] to double
        std::cout << n << "␣times␣" << x << "␣is␣" << n*x << std::endl;
    }
    return(EXIT_SUCCESS);
}
```

Program call

```
./comm_test 2 3.1
```

Output

```
Name of the program: ./comm_test
2 times 3.1 is 6.2
```

39 Homework

Homework

- Write a program, which asks for an input file name, opens the file for reading. If the file does not exist ask for a new name.

- Ask the user for an output file name. Open the output file.

- Read the input file line-by-line.

- Replace all spaces in each line by underscores (remember that strings are like vectors of chars...). Write the result to the output file.

- Count the number of chars in the file (you can add the length of the lines...)

- At the end of the program output the number of chars in the input file to the screen.

40 Homework

Homework

- Write a program, which asks for an input file name, opens the file for reading. If the file does not exist ask for a new name.
- Ask the user for an output file name. Open the output file.
- Read the input file line-by-line.
- Replace all spaces in each line by underscores (remember that strings are like vectors of chars...). Write the result to the output file.
- Count the number of chars in the file (you can add the length of the lines...)
- At the end of the program output the number of chars in the input file to the screen.

Solution

```
#include<iostream>
#include<fstream>
#include<string>

int main()
{
    std::cout << "Please␣enter␣the␣name␣of␣the␣input␣file:";
    std::string infilename;
    getline(std::cin,infilename);
    std::ifstream infile(infilename.c_str());
    while (infile.fail())
    {
        std::cerr << "Opening␣of␣input␣file␣" << infilename
                  << "␣failed!" << std::endl;
        std::cout << "Please␣enter␣the␣name␣of␣the␣input␣file:";
        getline(std::cin,infilename);
        infile.clear();
        infile.open(infilename.c_str());

    }
    std::cout << "Please␣enter␣the␣name␣of␣the␣output␣file:";
    std::string outfilename;
    getline(std::cin,outfilename);
    std::ofstream outfile(outfilename.c_str());
    while (outfile.fail())
    {
        std::cerr << "Opening␣of␣output␣file␣" << outfilename
                  << "␣failed!" << std::endl;
        std::cout << "Please␣enter␣the␣name␣of␣the␣output␣file:";
        getline(std::cin,outfilename);
        outfile.clear();
        outfile.open(outfilename.c_str());
    }
    std::string line;
    int numChars=0;
    while (getline(infile,line))
    {
        for (unsigned int i=0;i<line.size();++i)
            if ('␣'==line[i])
                line[i]='_';
        outfile << line << std::endl;
        numChars+=line.size();
```

```
    }
    std::cout << "The␣file␣" << infilename << "␣contains␣"
              << numChars << "␣characters" << std::endl;
}
```

41 Generic Programming

Generic Programming

- Often the same algorithms are needed for different data types.
- Without generic programming one has to write the function for all data types, which is tedious and error-prone, e.g.

```
int Square(int x)
{
    return(x*x);
}

long Square(long);
{
    return(x*x);
}
```

```
float Square(float);
{
    return(x*x);
}

double Square(double);
{
    return(x*x);
}
```

- Generic programming allows to write the algorithm once and parametrise it with the data type.
- Templates can be used for classes or functions.

41.1 Templates

Template functions

- A template function starts with the keyword `template` and a list of one or more template arguments in angle brackets separated by commas:

```
template<class T> T Square(T a)
{
    return(a*a);
}
```

- If a template is used, the compiler can automatically generate the function from the template function according to the function arguments (as with overloading the return type is not relevant)
- The template arguments can also be specified explicitly in angle brackets:

```
    std::cout << Square<int>(4) << std::endl;
```

Template functions

```
#include<cmath>
#include<iostream>

template<class T> T Square(T a)
{
    return(a*a);
}

int main()
{
    std::cout << Square<int>(4) << std::endl;
    std::cout << Square<double>(M_PI) << std::endl;
    std::cout << Square(3.14) << std::endl;
}
```

Template functions

The argument types must fit the declaration

```
#include<cmath>
#include<iostream>

template<class U> const U &max(const U &a, const U &b)
{
    if (a>b)
        return(a);
    else
        return(b);
}

int main()
{
    std::cout << max(1,4) << std::endl;
    std::cout << max(3.14,7.) << std::endl;
    std::cout << max(6.1,4) << std::endl; // compiler error
    std::cout << max<double>(6.1,4) << std::endl; // correct
    std::cout << max<int>(6.1,4) << std::endl; // warning
}
```

Usefull predefined template functions

The C++ standard library already provides some useful template functions:

- `const T &std::min(const T &, const T &)` minimum of a and b `int c = std::min(a,b);`
- `const T &std::max(const T &, const T &)` maximum of a and b `int c = std::max(a,b);`
- `void std::swap(T &, T &)` swap a and b `std::swap(a,b);`

Template classes, Non-type Template arguments, default arguments

```
template<class T, int dimension = 3> class NumericalSolver
{
    ...
```

```
    private:
        T variable;
}
```

- Template arguments can be used in class declarations.
- Not only types, but also values can be used as template arguments
- If templates are used in a class definition, the last template arguments can have default values.
- The name of a class is the class name plus the template parameters
- In a derived class it is often necessary to prefix members of the base class with `this->` (this is also true for non-template classes).

Usefull predefined classes: Pairs

The C++ standard library also provides a useful type:

```
std::pair<int,double> a;
a.first=2;
a.second=5.;
std::cout << a.first << "␣" << a.second << std::endl;
```

Pair allows e.g. functions to return two values.

41.2 Non-Numerical `MatrixClass` with Templates

Non-Numerical `MatrixClass` with Templates

```
#include<vector>
#include<iostream>
#include<iomanip>

template<class T>
class MatrixClass
{
  public:
    void Resize(int numRows, int numCols);
    void Resize(int numRows, int numCols, T value);
    std::vector<T> &operator[](int i);
    T &operator()(int i, int j);
    T operator()(int i, int j) const;
    const std::vector<T> &operator[](int i) const;
    void Print() const;
    int Rows() const
    {
        return numRows_;
    }
    int Cols() const
    {
        return numCols_;
    }

    MatrixClass() : a_(0),
            numRows_(0),
            numCols_(0)
    {};
```

```
    MatrixClass(int numRows, int numCols) :
            a_(numRows),
            numRows_(numRows),
            numCols_(numCols)
    {
        for (int i=0;i<numRows_;++i)
            a_[i].resize(numCols_);
    };

    MatrixClass(int numRows, int numCols, T value)
    {
        Resize(numRows,numCols,value);
    };

    MatrixClass(std::vector<std::vector<T> > a)
    {
        a_=a;
        numRows_=a.size();
        if (numRows_>0)
            numCols_=a[0].size();
        else
            numCols_=0;
    }

    MatrixClass(const MatrixClass &b)
    {
        a_=b.a_;
        numRows_=b.numRows_;
        numCols_=b.numCols_;
    }

  protected:
    std::vector<std::vector<T> > a_;
    int numRows_;
    int numCols_;
};

template<class T>
std::vector<T> &MatrixClass<T>::operator[](int i)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is (0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}
```

41.3 Derived NumMatrixClass with Templates

Derived NumMatrixClass with Templates

```
#include"matrixcore.h"

template<class T>
class NumMatrixClass : public MatrixClass<T>
{
  public:
    std::vector<T> Solve(std::vector<T> b);
    NumMatrixClass &operator*=(const NumMatrixClass &b);
    NumMatrixClass &operator*=(T x);
    NumMatrixClass &operator/=(T x);
    NumMatrixClass &operator+=(const NumMatrixClass &b);
```

```
    NumMatrixClass &operator-=(const NumMatrixClass &b);
    NumMatrixClass() : MatrixClass<T>()
    {};

    NumMatrixClass(int numRows, int numCols) :
                   MatrixClass<T>(numRows,numCols)
    {};

    NumMatrixClass(int numRows, int numCols, T value) :
                   MatrixClass<T>(numRows,numCols,value)
    {};

    NumMatrixClass(std::vector<std::vector<T> > a) : MatrixClass<T>(a)
    {};
};

template<class T>
NumMatrixClass<T> &NumMatrixClass<T>::operator*=(const NumMatrixClass<T> &x)
{
    if (x.numRows_!=this->numCols_)
    {
        std::cerr << "Dimensions␣of␣matrix␣a␣(" << this->numRows_
                  << "x" << this->numCols_ << ")␣and␣matrix␣x␣("
                  << x.numRows_ << "x" << x.numCols_
                  << ")␣do␣not␣match!" << std::endl;
        exit(EXIT_FAILURE);
    }
    NumMatrixClass temp(*this);
    Resize(this->numRows_,x.numCols_,0);
    for (int i=0;i<temp.numRows_;++i)
        for (int j=0;j<x.numCols_;++j)
            for (int k=0;k<temp.numCols_;++k)
                this->a_[i][j]+=temp.a_[i][k]*x.a_[k][j];
    return *this;
}

template<class T>
NumMatrixClass<T> &NumMatrixClass<T>::operator*=(T x)
{
    for (int i=0;i<this->numRows_;++i)
        for (int j=0;j<this->numCols_;++j)
            this->a_[i][j]*=x;
    return *this;
}
template<class T>
NumMatrixClass<T> operator*(const NumMatrixClass<T> &a,
                            const NumMatrixClass<T> &b)
{
    NumMatrixClass<T> temp(a);
    temp *= b;
    return temp;
}

template<class T>
NumMatrixClass<T> operator*(const NumMatrixClass<T> &a,T x)
{
    NumMatrixClass<T> temp(a);
    temp *= x;
    return temp;
}

template<class T>
NumMatrixClass<T> operator*(T x,const NumMatrixClass<T> &a)
{
    NumMatrixClass<T> temp(a);
    temp *= x;
```

```
    return temp;
}
```

41.4 Application of MatrixClass with Templates

```
#include "matrixcore.h"

template<class T> void Sort(MatrixClass<T> &A)
{
  for (size_t i=0;i<A.Rows();++i)
    for (size_t j=i+1;j<A.Rows();++j)
    {
      if (A(j,1)<A(i,1))
      {
        for (size_t k=0;k<A.Cols();++k)
          std::swap(A(i,k),A(j,k));
      }
    }
}

int main()
{   // define matrix
  MatrixClass<std::string> A(3,2);
  A(0,0)="Olaf";
  A(0,1)="Ippisch";
  A(1,0)="Dan";
  A(1,1)="Popovic";
  A(2,0)="Peter";
  A(2,1)="Bastian";
  A.Print();
  Sort(A);
  A.Print();
}

#include "nummatrix.h"

int main()
{   // define matrix
  NumMatrixClass<double> A(4,6,0.0);
  for (int i=0;i<A.Rows();++i)
    A[i][i] = 2.1;
  for (int i=0;i<A.Rows()-1;++i)
    A[i+1][i] = A[i][i+1] = -1.0;
  NumMatrixClass<double> B(6,4,0.0);
  for (int i=0;i<B.Cols();++i)
    B[i][i] = 1.9;
  for (int i=0;i<B.Cols()-1;++i)
    B[i+1][i] = B[i][i+1] = -1.0;
  // print matrix
  A.Print();
  B.Print();
  NumMatrixClass<double> C;
  C=A*B;
  C.Print();
  C=B*A;
  C.Print();
  C = 2.3*A;
  C.Print();
  C = C/2.3;
  C.Print();
}

#include "nummatrix.h"
```

```
int main()
{   // define matrix
  NumMatrixClass<int> A(4,6,0);
  for (int i=0;i<A.Rows();++i)
    A[i][i] = 2;
  for (int i=0;i<A.Rows()-1;++i)
    A[i+1][i] = A[i][i+1] = -1;
  NumMatrixClass<int> B(6,4,0);
  for (int i=0;i<B.Cols();++i)
    B[i][i] = 2;
  for (int i=0;i<B.Cols()-1;++i)
    B[i+1][i] = B[i][i+1] = -1;
  // print matrix
  A.Print();
  B.Print();
  NumMatrixClass<int> C;
  C=A*B;
  C.Print();
  C=B*A;
  C.Print();
  C = 3*A;
  C.Print();
  C = C/3;
  C.Print();
}
```

41.5 The Standard Template Library (STL)

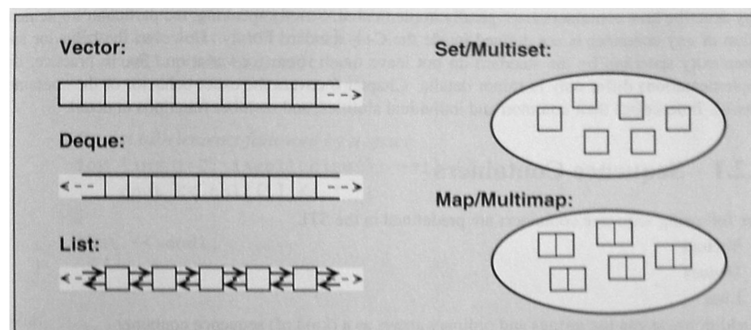
The Standard Template Library (STL)

The STL is

- a collection of useful template functions and classes.
- available for all modern C++ compilers.
- optimised for efficiency.
- needs a bit of explanation, but can make life much easier.

STL-Containers

- The STL defines containers and algorithms to use them.
- Containers are used to manage a collection of elements
- There is a wide variety of containers optimized for different purposes:



Sequence Containers

Sequence Containers are ordered collections, where each element has a certain position.

Vector is a dynamic array of elements.

- elements can be accessed directly with an index (random access).
- appending and removing at the end is fast.

Deque , the "double-ended" queue, is a dynamic array.

- can grow in both directions.
- elements can be accessed directly with an index, however, if elements are inserted at the front, the index of a certain element might change.
- appending and removing elements at the end and the beginning is fast.

Sequence Containers

Sequence Containers are ordered collections, where each element has a certain position.

List is a double-linked list of elements.

- no direct access to elements.
- to access the tenth element, one has to start at the beginning and travers the first nine elements.
- inserting is fast at any position.

Associative Containers

Associative containers keep the elements automatically in a sorted order

Set is a sorted collection of items, where each item can only occur once.

Multiset like set, but multiple occurances of the same item are allowed.

Map contains elements that are pairs of a key and a value. The map is sorted by the key. Each key may only occur once.

Multimap like map, but more than one pairs with the same key are allowed.

STL-Vector

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<double> a(7);
    std::cout << a.size() << std::endl;
    for (int i=0;i<7;++i)
        a[i] = i*0.1;
    double d = 4 * a[2];
    std::vector<double> c(a);
    std::cout << a.back() << "␣" << c.back() << std::endl;
    std::vector<std::string> b;
    b.resize(3);
    for (int i=2;i>=0;--i)
        std::cin >> b[i];
```

```
    b.resize(4);
    b[3] = "blub";
    b.push_back("blob");
    for (int i=0;i<b.size();++i)
        std::cout << b[i] << std::endl;
}
```

STL-Iterators

An iterator is a pointer-like object, which can be used to traverse a container. An iterator points to a certain position of a container. It provides the operations:

Operator `*` returns the element at the current position

Operator `++` advances the iterator to the next element

Operator `->` can be used to access members of an element if it has any

Operator `==` **or** `!=` returns if two iterators point (not) to the same position

Operator `=` assigns an iterator

STL-Iterators

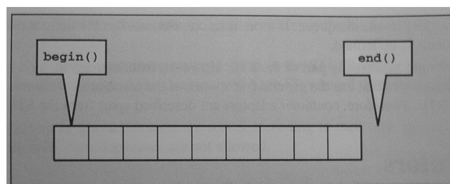
All STL-container classes define an iterator, e.g.

```
std::vector<double>::iterator i;
std::vector<int>::iterator i;
std::list<std::string>::iterator i;
```

and provide the members:

begin() returns an iterator to the first element of the container

end() returns an iterator that represents the end of the container (one past the last element)



They can be used to traverse a container.

STL-Iterators

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<double> a(7);
    for (int i=0;i<7;++i)
```

```
        a[i] = i*0.1;
    for (std::vector<double>::iterator i = a.begin();
         i != a.end(); ++i)
        std::cout << *i << std::endl;
}
```

Advantages of using Iterators

- Iterators produce correct loops over a container even if its content changes. While

  ```
  for (int i=0;i<7;++i)
      a[i] = i*0.1;
  ```

  might only iterate over a part of the vector

  ```
  for (std::vector<double>::iterator i = a.begin();
       i != a.end(); ++i)
      std::cout << *i << std::endl;
  ```

  always iterates over the whole container.

- Iterators provide a generic interfaces to iterate over a container (which works for all types of STL-containers) which is especially helpful for writing template functions.

STL-List

```
#include <list>
#include <iostream>

int main()
{
    std::list<double> a;
    std::cout << a.size() << std::endl;
    for (int i=0;i<7;++i)
        a.push_back(i*0.1);
    std::cout << a.size() << std::endl;
    for (int i=0;i<3;++i)
        a.push_front(i+0.1);
    std::cout << a.size() << std::endl << std::endl;
    for (std::list<double>::iterator i=a.begin();i!=a.end();++i)
        std::cout << (*i) << ", ";
    std::cout << std::endl << std::endl;;
    for (std::list<double>::iterator i=a.begin();i!=a.end();++i)
    {
        if ((*i)==0.4)
            a.insert(i,0.41);
    }
    std::cout << a.size() << std::endl << std::endl;
    for (std::list<double>::iterator i=a.begin();i!=a.end();++i)
        std::cout << (*i) << ", ";
    std::cout << std::endl;
}
```

STL-Deque

```
#include <deque>
#include <iostream>
int main()
{
    std::deque<double> a;
    std::cout << a.size() << std::endl;
```

```
    for (int i=0;i<7;++i)
        a.push_back(i*0.1);
    std::cout << a.size() << std::endl;
    for (int i=0;i<3;++i)
        a.push_front(i+0.1);
    std::cout << a.size() << std::endl << std::endl;
    for (std::deque<double>::iterator i=a.begin();i!=a.end();++i)
        std::cout << (*i) << ", ";
    std::cout << std::endl << std::endl;;
    for (std::deque<double>::iterator i=a.begin();i!=a.end();++i)
        if ((*i)==0.4)
        {
            a.insert(i,0.41);
            ++i;
        }
    std::cout << a.size() << std::endl << std::endl;
    for (std::deque<double>::iterator i=a.begin();i!=a.end();++i)
        std::cout << (*i) << ", ";
    std::cout << std::endl;
}
```

Example with Vector Iterators

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    // create empty vector for strings
    vector<string> sentence;

    // reserve memory for five elements to avoid reallocation
    sentence.reserve(5);

    // append some elements
    sentence.push_back("Hello,");
    sentence.push_back("how");
    sentence.push_back("are");
    sentence.push_back("you");
    sentence.push_back("?");

    // print elements separated with spaces
    copy (sentence.begin(), sentence.end(),
          ostream_iterator<string>(cout," "));
    cout << endl;

    // print ''technical data''
    cout << "  max_size(): " << sentence.max_size() << endl;
    cout << "  size():     " << sentence.size()     << endl;
    cout << "  capacity(): " << sentence.capacity() << endl;

    // swap second and fourth element
    swap (sentence[1], sentence[3]);

    // insert element "always" before element "?"
    sentence.insert (find(sentence.begin(),sentence.end(),"?"),
                     "always");

    // assign "!" to the last element
    sentence.back() = "!";
```

```
    // print elements separated with spaces
    copy (sentence.begin(), sentence.end(),
          ostream_iterator<string>(cout," "));
    cout << endl;

    // print ''technical data'' again
    cout << "  max_size(): " << sentence.max_size() << endl;
    cout << "  size():     " << sentence.size()     << endl;
    cout << "  capacity(): " << sentence.capacity() << endl;
}
```

Output

```
Hello, how are you ?
  max_size(): 1073741823
  size():     5
  capacity(): 5
Hello, you are how always !
  max_size(): 1073741823
  size():     6
  capacity(): 10
```

Literature on Using the STL

N. M. Josuttis: The C++ Standard Library: A Tutorial and Reference, Addison-Wesley Longman, Amsterdam 1999

42 Résumé

Résumé of the Commas C6 Course

Programming

- Introduction of basic programming techniques (Variables, Conditional Execution, Loops)
- Procedural Programming (Functions, Recursion)
- Object-Oriented Programming (Classes, Objects, Operators, Inheritance, Virtual Functions, Interface Classes)
- Templates
- STL containers

Numerical Applications

- Floating Point Numbers/Round-off Errors
- Direct Solution of Linear Equation Systems
- Polynomial Interpolation
- Numerical Integration

How to become a good Programmer

You can read Peter Norvigs article “Teach yourself programming in ten years!” (http://www.norvig.com/21-). He recommends

- Find a way to enjoy programming (If not with C++, then with easier languages like Python)
- Talk with other programmers and read other people’s programs
- Practice programming
- If you want take further courses on computer science
- Take part in development projects, try to be once the worst, once the best programmer in the team
- Learn different programming languages
- Find out more about your computer (how long do certain instructions take, how fast is memory access etc.)