Object-Oriented Programming for Scientific Computing

Administrativa, Introduction and Quick Recap

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

18. April 2017

Prerequisites and Objectives

Prerequisites

- Advanced knowledge of a programming language
- At least procedural programming in C / C++
- Willingness to program in practice

Objectives

- Improved programming skills
- Introduction of modern programming models
- Strong focus on topics of relevance to Scientific Computing

Course Outline

- Short recapitulation of basics of object-oriented programming in C++ (classes, inheritance, methods and operators)
- Scientific Computing and best practices
- Smart Pointers and Constness
- Error handling (exceptions)
- Dynamic polymorphism (virtual inheritance)
- Static polymorphism (templates)
- The C++ Standard Template Library (STL)
- Traits and Policies
- Design Patterns
- Template Metaprogramming
- C++ Threads

Throughout the lectures the changes by the C++{11,14,17} standards will be taken into account.

Lecture Website

Link: `conan.iwr.uni-heidelberg.de/teaching/ooprogram_ss2017/`

Content:

- Link to draft of official standard
- Literature recommendations and quick reference
- GitLab instructions (for exercises)
- Lecture slides (updated after each lecture)
- Exercise sheets (same as above)

MUESLI links (also listed on the lecture website):

- Registration: `muesli.mathi.uni-heidelberg.de/user/register`
- Course page: `muesli.mathi.uni-heidelberg.de/lecture/view/728`

Exercises and Exam

Exercises:

- Thursdays, 16:00 - 18:00, INF205, Computer Pool
- New exercises every week: on the website after the lecture
- To be handed in right before the lecture on Tuesday
- Geared towards g++ and Linux
- Correction, grading etc. depends on course size

Exam:

- Will take place in the last week of the semester
- Mandatory for all participants
- 50% of the points in the exercises required for admission

What are Attributes of a Good Program?

Correct / bug free:

Bugs may lead to wrong results and therefore conclusions

Efficient:

Bad coding choices can increase the runtime by orders of magnitude

Easy to handle:

A tool that is difficult to use will not become adopted

Easy to understand:

You or someone else will have to be able to maintain the code

Expandable:

Unforeseen changes in requirements make extensions necessary

Portable:

The program should work in all relevant environments (e.g. clusters)

Developments in Recent Years

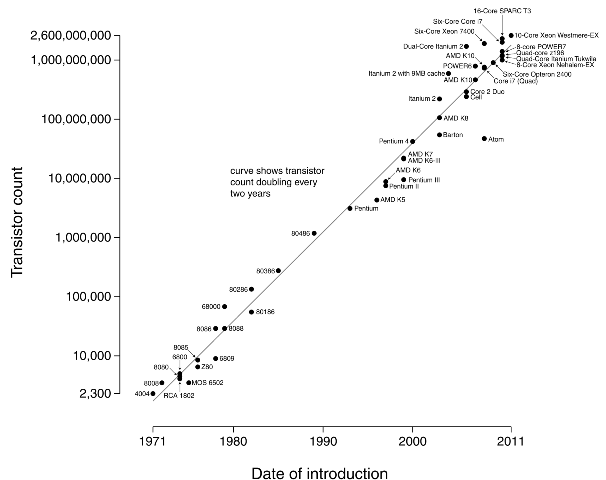
- Computers have become faster and cheaper
- Program size has risen from several hundred to hundreds of thousands of lines
- This led to an increase in complexity of programs
- Programs are now developed in larger groups, not by individual programmers
- Parallel computing is becoming increasingly important, as by now almost all computers are sold with multiple cores

Moore's Law

The number of transistors on processors doubles approximately every two years

But: the number of transistors *per processor core* stagnates

Figure source: Wikipedia



Moore's Law (II)

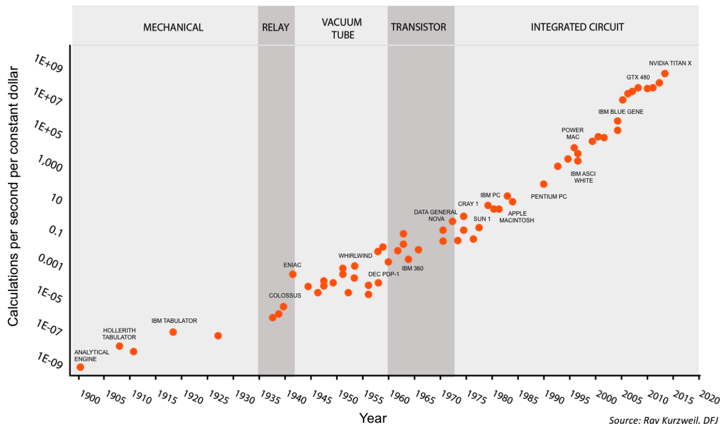


Figure source: Wikipedia

Wirth's Law

Moore's Law states that computers should become roughly twice as efficient every two years, but the responsiveness of (consumer) software packages tends to stay the same.

This tendency of sofware bloat is known as Wirth's Law or the Great Moore's Law Compensator.

Main reasons:

- Under-optimization of code due to available resources (Jevons paradox, rebound effect)
- Layering of code, various levels of indirection
- Inclusion of non-essential but convenient features

> Our tests show that Windows Vista and Office 2007 not only smash Redmond's previous records for weight gain, but given the same hardware diet, run at less than half the speed of generation XP
>
> *Randall C. Kennedy*

Complexity of Programs

Year	Processor	System Clock [MHz]	Cores	RAM [MB]	Disk [GB]	Linux Kernel [MB]	
1982	Z80	6	1	0.064	0.0008	0.006	(CPM)
1988	80286	10	1	1	0.02	0.020	(DOS)
1992	80486	25	1	20	0.16	0.140	(0.95)
1995	PII	100	1	128	2	2.4	(1.3.0)
1999	PII	400	1	512	10	13.2	(2.3.0)
2001	PIII	850	1	512	32	23.2	(2.4.0)
2007	Core2 Duo	2660	2	1'024	320	302	(2.6.20)
2010	Core i7-980X AMD 6174	3333 (3600) 2200	6 12	4'096	2'000	437	(2.6.33.2)
2013	Core i7-3970X AMD 6386 SE	3500 (4000) 2800 (3500)	6 16	8'192	4'000	482	(3.8.7)
2016 / 2017	Core i7-6950X AMD Ryzen	3000 (4000) 3000 (4100) ?	10 16	16'384	10'000	680	(4.10.10)

For Instance: DUNE

Distributed and Unified Numerics Environment

Main features: *generic, efficient, parallel*

- Framework for the solution of partial differential equations (PDEs)
- Developed by working groups at the universities of Freiburg, Heidelberg, Munster, the Free University of Berlin and the RWTH Aachen
- 14 Core Developers, many more developers
- Currently (April 2017) 320,822 lines of code (core + extensions)
- Additionally e.g. 58,147 lines of code in downstream module PDELab
- Users at many other universities and in industry
- Intensive use of modern C++ constructs presented in this course

History of Dune

Date	Ver.	Core	PDELab
12/2007	1.0	91,851	
04/2008	1.1	100,273	
04/2009	1.2	113,858	8,756
04/2010	2.0	145,520	19,933
05/2012	2.1	157,870	33,324
08/2013	2.2	164,936	51,529
07/2014	2.3	179,062	54,311
06/2016	2.4	180,303	59,040
03/2017	2.5	265,704	58,100

Evolution of project size follows typical patterns:

- Initial phase, rapid growth due to addition of features / constructs
- Consolidation phase, slowdown due to restructuring

(Large jump in 2017 is inclusion of former large external module UGGrid, predecessor of Dune.)

Programming Paradigms

Functional programming (e.g. Haskell, Scheme):

- Program consists only of functions
- No loops, repetitions are realized via recursion
- No concept of state

Imperative programming (e.g. Fortran, C++):

- Program consists of a sequence of instructions
- Variables can store intermediate values
- Special instructions which change execution order, e.g. for repetitions

Imperative Programming Models

Procedural programming (e.g. C, Fortran, Pascal, Cobol, Algol):

- Program is divided into small parts (procedures or functions)
- Data is only stored locally and deleted when procedure exits
- Persistent data is
  - exchanged via arguments and return values
  - saved as global variables

Modular programming (e.g. Modula-2, Ada):

- Functions and data are combined into modules
- Modules are responsible for execution of particular tasks
- Can in large parts be programmed and tested seperately

The Object-Oriented Programming Approach

Examples: Java, C++

In analogy to mechanical engineering:

- Split the program into independent components
- Determine necessary functionality to provide this component
- All required data for this is managed within the component
- Components are connected via interfaces
- Use the same interface for specialized components executing the same tasks

Benefits of OOP

Another close analogy: a computer is built from components that

- fulfill a specific role (CPU, RAM, hard disk, graphics card, . . .)
- have specifications and responsabilities
- are connected through interfaces
- can be exchanged and upgraded

Benefits of this approach:

- Components can be developed independently
- If better versions become available, they can be used without major changes to the rest of the system
- It's easy to use multiple implementations of the same component

How Does C++ Help?

C++ provides several support mechanisms:

- **Classes** define components, are like a description of what a component does and what properties it has (like the functionality a specific graphics card provides)
- **Objects** are realizations of the class (like a graphics card with a specific serial number)
- **Encapsulation** prevents side effects by hiding data from other parts of the program
- **Inheritance** facilitates a uniform and consistent implementation of specialized components, with abstract base classes as standard interfaces
- **Virtual functions** allow to select between different specializations of a component at runtime
- **Templates** increase efficiency when the choice of specialization is known at compilation time

Example

```
#include<vector>

class Matrix
{
  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> > entries;
    int numRows;
    int numCols;
};
```

Class Declaration

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

- The class declaration defines interface and essential characteristics of the component.
- A class has *attributes* (variables to store data) and *methods* (the functions provided by a class).
- The definition of attributes and the declaration of methods are enclosed in braces, followed by a a mandatory semicolon.
- Class declarations are usually saved in a file with the extension '.hh' or '.h' , so-called *header 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. [...] *Representation is the essence of programming.*

Brooks (1975)

Encapsulation (II)

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

The keyword `public:` marks the the description of the interface, i.e. the methods of the class which can be accessed from the outside.

The keyword `private:` accompanies the definition of attributes and methods that are *only available to objects of the same class*. This includes the data and implementation-specific methods reqired by the class. To ensure data integrity it should *not* be possible to access stored data from outside the class.

Encapsulation (III)

```
struct Matrix
{
    // a list of public methods
  private:
    // a list of private methods and attributes
};
```

If no keywords are given all data and methods of a class defined with the keyword `class` are `private`. If a class is defined with the keyword `struct`, as in `struct Matrix`, then all methods are `public` by default. Apart from this distinction `class` and `struct` are identical.

Definition of Attributes

```
class Matrix
{
  private:
    std::vector<std::vector<double> > entries;
    int numRows;
    int numCols;
    // further private methods and attributes
};
```

The definition of a class attribute in C++ is identical to any other variable definition and consists of a data type and a variable name.
Possible types are, e.g.:

- `float` and `double` for floating point numbers with single and double precision
- `int` and `long` for integer numbers
- `bool` for logical states
- `std::string` for strings

Fundamental Data Types

`void`:

- type with empty set of values
- represents "nothing" (return type) or "anything" (pointers)

`bool`:

boolean truth values, `true (1)` and `false (0)`

Char: representation of characters

Name	Purpose	Size	
`char`	ASCII / UTF-8	$\geq$ 8 bit	(C++14)
`wchar_t`	wide chars	$\geq$ 32 bit	(except for Windows)
`char16_t`	UTF-16	$\geq$ 32 bit	(C++11)
`char32_t`	UTF-32	$\geq$ 64 bit	(C++11)

Fundamental Data Types (II)

Integer:

Name		Size	
`short`,	`unsigned short`	$\geq$ 16 bit	
`int`,	`unsigned int`	$\geq$ 16 bit	
`long`,	`unsigned long`	$\geq$ 32 bit	
`long long`,	`unsigned long long`	$\geq$ 64 bit	(C++11)

Floating point:

Name	Usual Size
`float`	32 bit
`double`	64 bit
`long double`	80 bit

- 80 bit: native floating point representation in Floating Point Unit (FPU)
- If in doubt, use `double` for computations, `float` for storage if space is an issue

C++11: New Data Types

The variable length and thus the range of values of `short`, `int` and `long` (and their unsigned variants) isn't well defined in C und C++. It is only guaranteed that

```
sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
```

Apart from `long long`, C++11 introduces new data types with guaranteed lengths and range of values:

```
int8_t   [-128:127]        uint8_t   [0:255]
int16_t  [-32768:32767]    uint16_t  [0:65535]
int32_t  [-2^31:2^31-1]    uint32_t  [0:2^32-1]
int64_t  [-2^63:2^63-1]    uint64_t  [0:2^64-1]
```

C++11: New Data Types (II)

Additionally there are variants that start with `int_fast` or `uint_fast` (e.g. `int_fast8_t`). These provide the fastest data type on the respective architecture that has at least the appropriate length.

Data types beginning with `int_least` or `uint_least` produce the shortest data types that have the suitable range of values.

`intptr_t` and `uintptr_t` are data types with the right length to store a pointer, and `intmax_t` and `uintmax_t` are the largest available integer types.

C++{11,14,17}: Compiling Programs

```
g++ -std=c++11 -o executable_file source.cc
g++ -std=c++14 -o executable_file source.cc
g++ -std=c++17 -o executable_file source.cc
```

- To translate a program with constructs from C++11 or above it may be necessary to specify the standard on the command line.
- The details depend on the compiler (option / flag) and its version (default standard).

Before GCC 6.1: default `c++98`

GCC 6.1 and above: default `c++14`

Older versions of compilers may reqire using `c++0x` instead of `c++11`. The same holds for `c++14` (`c++1y`) and `c++17` (`c++1z`).

Information about standard support throughout different versions of `g++` can be found at http://gcc.gnu.org/projects/cxx-status.html

Declaration of Methods

```
class Matrix
{
  public:
    void init(int numRows_, int numCols_);
    double& elem(int i, int j);
};
```

A method declaration always consists of four parts:

- type of return value
- name of function
- list of arguments in parentheses
- a semicolon

The return value is `void` if a function does not return a value.

If a method has no arguments, the parentheses remain empty.

Definition of Methods

```
class Matrix
{
  public:
    void init(int numRows_, int numCols_);
    inline double& elem(int i, int j)
    {
      return entries[i][j];
    }
};
```

The method definition (i.e. listing of the actual function code) can be placed directly in the class (so-called inline functions).

In the case of inline functions the compiler can omit the function call and use the code directly. With the keyword `inline` in front of the function's name one can explicitly tell it to do that, but this is typically not necessary with modern compilers.

Definition of Methods (II)

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

If methods are defined outside the definition of a class (this is often done in files with the ending `.cpp`, `.cc` or `.cxx`), then the name of the method must be prefixed with the name of the class followed by two colons.

This is part of a concept called “namespaces”, which will be discussed in more detail later.

Overloading of Methods

```
class Matrix
{
  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) in C++ can have the same name if they differ in the number or type of arguments. This is called function overloading. A different type of the return value is not sufficient.

A method may have default arguments, e.g.

```
void init(int numRows_, int numCols_, double value = 0.)
```

The default value is used if the corresponding variable is omitted when the method is called. (But would lead to a name resolution conflict in the example above.)

Constructors

```
class Matrix
{
  public:
    Matrix();
    Matrix(int numRows_, int numCols_);
    Matrix(int numRows_, int numCols_, double value);
};
```

- Every class has methods without return value with the same name as the class itself: one or more constructors and the destructor.
- Constructors are executed when an object of a class is defined, before any other method is called or attributes may be used. They are used for initialization.
- There may be more than one constructor. The same rules as for overloaded methods apply.
- If there is no constructor which is `public`, objects of the class cannot be created.

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

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

- Like an ordinary method, constructors can be defined either inside or outside of the class definition.
- Constructors can also be used to initialize attributes with values using an initializer list:
  - consists of variable names followed by values to be used for initialization (constant or variable) in parentheses
  - appears after the closing parenthesis of the arguments, separated by a colon

C++11: In-Class Initialization, Delegating Constructors

```
class Matrix
{
  private:
    std::vector<std::vector<double> > entries;
    int numRows = 0;
    int numCols = 0;
  public:
    Matrix();
    Matrix(int numRows_, int numCols_, double value);
    Matrix(int numRows_, int numCols_) :
        Matrix(numRows_,numCols_,0.)
    {}
};
```

- C++11 additionally allows non-static members of classes to be initialized with a default value in their definition. If the member appears in the initializer list of a constructor that local value takes precedence.
- Constructors can call other constructors. The constructor receiving the remaining arguments is determined as in the case of overloaded functions.

Destructor

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

- There is only one destructor per class. It is called when an object of the class is destroyed.
- The destructor has no arguments (the brackets are therefore always empty).
- Writing your own destructor is necessary, e.g., when the class uses dynamically allocated memory.
- The destructor should always be `public`.

Default Methods

If they aren't explicitly defined differently, the compiler automatically generates the following five methods for each class `class T`:

- Default constructor without argument: `T();` (recursively calls constructors of attributes)
- Copy constructor: `T(const T&);` (memberwise copy)
- Destructor: `~T();` (recursively calls destructor of attributes)
- assignment operator: `T& operator= (const T&);` (memberwise copy)
- address operator: `int operator& ();` (returns storage address of object)

Default constructors are only generated if no other constructors are defined.

Assignment operators are only generated if none of the attributes is declared `const`.

C++11 and above: move constructor and move assignment operator (later)

Copy Constructor and Assignment Operator

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

int main()
{
  Matrix A(4,5,0.);
  Matrix B = A; // copy constructor
  A = B;        // assignment operator
}
```

- The copy constructor is called when a new object is created as a copy of an existing object. This often happens implicitly (e.g. when creating temporary objects).
- The assignment operator is called when an existing object is assigned a new value.

C++11: Management of Default Methods

```
class Matrix
{
  public:
    // Prohibit copying of matrices
    Matrix& operator=(const Matrix& A) = delete;
    Matrix(const Matrix& A) = delete;
    // prevent automatic conversion of short
    Matrix(int i, int j, double value);
    Matrix(short i, short j, double value) = delete;
    virtual ~Matrix() = default;
};
```

- Sometimes one wants to prevent the generation of certain default methods, e.g. so that only one object of a class can be created (later: singleton pattern).
- Previously, one had to create the default methods and declare them `private`.
- With C++11 this can be achieved by using the keyword `delete`.
- Classes with virtual functions should have a virtual destructor even if the actual class doesn't require any. This is easy and clear in C++11 with the keyword `default`.

Operator Overloading

- In C++ it is possible to redefine operators like $+$ or $-$ for a user-defined class.
- Operators are defined like usual functions. The function's name is `operator` followed by the symbol of the operator, for example `operator+`.
- Operators require the definition of a return type value and argument list just as ordinary methods.

```
Matrix operator+(Matrix& A);
```

- Operators can be defined
  - as a method of an object
  - as ordinary (non-member) functions.
- The number of arguments depends on the operator and cannot be changed in redefinitions.

Unary Operators

```
class Matrix
{
  public:
    Matrix operator-();
};

Matrix operator+(const Matrix& A);
```

- Unary operators are: `++ -- + - ! ~ & *`
- A unary operator can be defined as
  - a class function without an argument
  - a non-member function with one argument.
- The programmer must choose one of these two options:
  `Matrix& operator++(Matrix A)` and
  `Matrix& Matrix::operator++()`
  would both be called using `++A`, and it is impossible for the compiler to distinguish between the two variants.

Binary Operators

```cpp
class Matrix
{
  public:
    Matrix operator+(Matrix& A);
};

Matrix operator+(Matrix& A, Matrix& B);
```

- A binary operator can be defined as
  - a class function with one argument
  - a non-member function with two arguments.
- Possible operators are: `* / % + - & ^ | < > <= >= == != && || >> <<`
- Operators which change an argument, such as `+= -= /= *= %= &= ^= |=`, can only be implemented as a class function (method).

Binary Operators (II)

When an operator has arguments of different types, it is only responsible for exactly this sequence of arguments, e.g. the expression `A = A * 2.1` may use the operator defined by `Matrix operator*(Matrix& A, double b)`, but not `A = 2.1 * A`.

There is a simple trick to implement both versions efficiently:

- define the combined assignment operator, e.g. `operator*=` for multiplication, within the class
- implement two non-member functions outside the class that use this operator.

Increment and Decrement

- There are both prefix and postfix versions of the increment and decrement operators
- The postfix version (`a++`) is defined through `operator++(int)`, while the prefix version uses `operator++()`, which doesn't have an argument. The argument `int` of the postfix version is not used and serves only to distinguish both alternatives (i.e., this is literally a hack).
- Note that the postfix operator cannot return a reference since it should return the unaltered original state of its argument.

```
class Ptr_to_T
{
  T* p;

  public:
    Ptr_to_T& operator++();      // Prefix version
    Ptr_to_T  operator++(int);   // Postfix version
}

Ptr_to_T& operator++(T&);        // Prefix version
Ptr_to_T  operator++(T&, int);   // Postfix version
```

The Parenthesis Operators

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

- The operators for square brackets and parentheses can also be overloaded. This can be used to write expressions such as `A[i][j]=12` or `A(i,j)=12`.
- The operator for brackets takes always exactly one argument.
- The operator for parentheses can use an arbitrary number of arguments.
- Both can be overloaded several times.

Conversion Operators

```
class Complex
{
  public:
    operator double() const;
};
```

- Conversion operators (cast operators) can be used to convert user-defined variables into one of the built-in types.
- The name of a conversion operator is `operator` followed by the resulting type (separated by a blank).
- Conversion operators are constant methods.

Conversion Operators(II)

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

class Complex
{
  public:
    operator double() const
    {
        return std::sqrt(re * re + im * im);
    }
    Complex(double real, double imag) : re(real), im(imag)
    {}
  private:
    double re;
    double im;
};

int main()
{
    Complex a(2.,-1.);
    double b = 2. * a;
    std::cout << b << std::endl;
}
```

Self-Reference

- Each method of a class knows the object from which it has been called.
- Each such function receives a pointer / a reference to this object.
- The name of the pointer is `this`, the name of the reference is `*this`.
- The self-reference is e.g. necessary for operators modifying an object:

```
Matrix& Matrix::operator*=(double x)
{
    for (int i = 0; i < numRows; ++i)
        for (int j = 0; j < numCols; ++j)
            entries[i][j] *= x;
    return *this;
}
```

- Another reason for using `this` can be the combination of templates and inheritance: grants access to attributes in the templatized base class.

Example: Matrix Class

The following example implements a class for matrices.

- `matrix.h`: contains the definition of `Matrix`
- `matrix.cc`: contains the implementation of the methods of `Matrix`
- `testmatrix.cc`: is a sample application to illustrate the use of `Matrix`

Included in class:

- Examples of operators above
- Proper constness
- Delegating constructors (C++11)
- Attribute default values (C++11)

Missing:

- Separation interface / implementation (private methods)
- Move constructor / assignment operator (later)
- Exceptions instead of hard errors (later)
- Use of templates (later)

Header of Matrix Class

```
#include<vector>

class Matrix
{
  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
    Matrix& operator*=(double x);
    Matrix& operator+=(const Matrix& b);
    std::vector<double> solve(std::vector<double> b) const;
    // output
    void print() const;
    int rows() const
    {
        return numRows;
    }
    int cols() const
    {
        return numCols;
    }
```

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

Matrix(int dim) : Matrix(dim,dim)
{};

Matrix(int numRows_, int numCols_, double value)
{
    resize(numRows_,numCols_,value);
};
```

```
    Matrix(std::vector<std::vector<double> > a)
    {
        entries = a;
        numRows = a.size();
        if (numRows > 0)
            numCols = a[0].size();
        else
            numCols = 0;
    }

    Matrix(const Matrix& b)
    {
        entries = b.entries;
        numRows = b.numRows;
        numCols = b.numCols;
    }

  private:
    std::vector<std::vector<double> > entries;
    int numRows = 0;
    int numCols = 0;
};

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

Implementation of Matrix Class

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

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

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

```
double& Matrix::operator()(int i, int j)
{
    if (i < 0 || i >= numRows)
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣[0:" << numRows-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if (j < 0 || j >= numCols)
    {
        std::cerr << "Illegal␣column␣index␣" << j;
        std::cerr << "␣valid␣range␣is␣[0:" << numCols-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return entries[i][j];
}
```

```
double Matrix::operator()(int i, int j) const
{
    if ( i < 0 || i >= numRows)
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣[0:" << numRows-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if (j < 0 || j >= numCols)
    {
        std::cerr << "Illegal␣column␣index␣" << j;
        std::cerr << "␣valid␣range␣is␣[0:" << numCols-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return entries[i][j];
}
```

```
std::vector<double>& Matrix::operator[](int i)
{
    if (i < 0 || i >= numRows)
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣[0:" << numRows-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return entries[i];
}

const std::vector<double>& Matrix::operator[](int i) const
{
    if (i < 0 || i >= numRows)
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣[0:" << numRows-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return entries[i];
}
```

```
Matrix& Matrix::operator*=(double x)
{
    for (int i = 0; i < numRows; ++i)
        for (int j = 0; j < numCols; ++j)
            entries[i][j] *= x;
    return *this;
}

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

```
std::vector<double> Matrix::solve(std::vector<double> b) const
{
    std::vector<std::vector<double> > a(entries);
    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.;
            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 Matrix::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) << entries[i][j] << "␣";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

Matrix operator*(const Matrix& a, double x)
{
    Matrix output(a);
    output *= x;
    return output;
}
```

```
Matrix operator*(double x, const Matrix& a)
{
    Matrix output(a);
    output *= x;
    return output;
}

std::vector<double> operator*(const Matrix& 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.;
        for (int j = 0; j < a.Cols(); ++j)
            y[i] += a[i][j] * x[j];
    }
    return y;
}
```

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
    Matrix output(a);
    output += b;
    return output;
}
```

Possible improvements:

- Template parameter for type of entries
- Template template parameter for internal container (later)
- Iterators and range-based for loops (later)
- Policy for `solve` method (later)

Application using Matrix Class

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

int main()
{   // define matrix
    Matrix 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.;
    Matrix 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();
    Matrix C(A);
    A = 2 * C;
    A.print();
    A = C * 2.;
    A.print();
    A = C + A;
    A.print();
```

```
    const Matrix D(A);
    std::cout << "Element␣1,1␣of␣D␣is␣" << D(1,1) << std::endl;
    std::cout << std::endl;
    A.resize(5,5,0.);
    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.;
    // define vector b
    std::vector<double> b(5);
    b[0] = b[4] = 5.;
    b[1] = b[3] = -4.;
    b[2] = 4.;
    std::vector<double>x = A * b;
    std::cout << "A*b␣=␣(␣";
    for (size_t 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 (size_t i = 0; i < x.size(); ++i)
        std::cout << x[i] << "␣␣";
    std::cout << ")" << std::endl;
}
```

Output of the Application

```
(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: Output of the Application (II)

```
(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  )
```

Summary

- Scientific software is nowadays written collaboratively
- Parallel computing is necessary to utilize all resources of modern computers
- Object-oriented programming divides given problems into small subtasks and components that are responsible for these tasks
- Encapsulation highlights interfaces and hides implementation details
- C++ classes have methods that deal with the creation and destruction of objects (constructors / destructors), methods and operators
- Overloading of functions allows flexibility