Object-Oriented Programming for Scientific Computing

Formalia, Introduction and Quick Recap

Ole Klein

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

14. April 2015

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

Content

- Short recapitulation of basics of object-oriented programming in C++ (classes, inheritance, methods and operators)
- Constant values and objects
- Error handling (exceptions)
- Dynamic polymorphism (virtual inheritance)
- Static polymorphism (templates)
- The C++ Standard Template Library (STL containers, iterators, and algorithms)
- Traits, Policies
- Design Patterns
- Template Metaprogramming
- C++-Threads

Throughout the lectures the changes by the C++11 standard will be taken into consideration.

Lecture Website

`http://conan.iwr.uni-heidelberg.de/teaching/ooprogram_ss2015/`

- Lecture notes (by Olaf Ippisch)
    - exist in German on the lecture website
    - will be updated after the lecture series
- Lecture slides
    - are being translated into English and possibly extended
    - can be found on the website after the lecture
- Document with a short overview of procedural C++ commands
- Exercise sheets

Exercises and Exam

Exercises:

- Thursdays, 14:15 - 15:45, INF350 (OMZ), room U014
- 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 bachelor and master students
- 50% of the points in the exercises required for admission

Registration

Registration links (also listed on the lecture website):

- Master students:
  `www.mathi.uni-heidelberg.de/muesli/user/register`
- PhD students:
  `www.mathi.uni-heidelberg.de/muesli/user/register_other`

Credit points:

- Master students: 6 points for passing the exam
- PhD students: 4 points for lecture + participation in exercises
- Participation in exam possible for PhD students if capacities allow
- All credit points subject to negotiation with graduate school

What are Attributes of a Good Program?

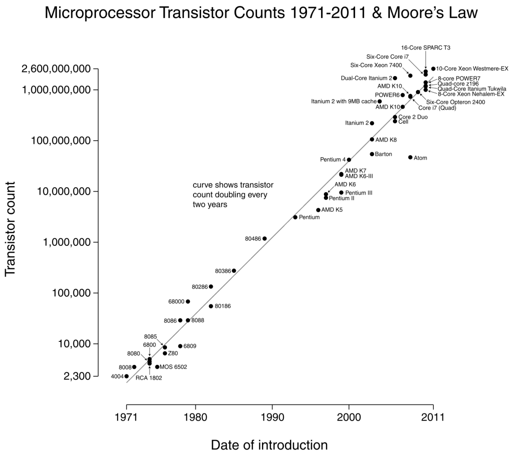
- Correct / bug free
- Efficient
- Easy to handle
- Easy to understand
- Expandable
- Portable

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 the 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



Complexity of Programs

Time	Processor	System Clock [MHz]	Cores	RAM [MB]	Disk [MB]	Linux Kernel [MB]
1982	Z80	6	1	0.064	0.8	0.006 (CPM)
1988	80286	10	1	1	20	0.020 (DOS)
1992	80486	25	1	20	160	0.140 (0.95)
1995	PII	100	1	128	2'000	2.4 (1.3.0)
1999	PII	400	1	512	10'000	13.2 (2.3.0)
2001	PIII	850	1	512	32'000	23.2 (2.4.0)
2007	Core2 Duo	2660	2	1'024	320'000	302 (2.6.20)
2010	Core i7-980X AMD 6174	3333 (3600) 2200	6 12	4'096	2'000'000	437 (2.6.33.2)
2013	Core i7-3970X AMD 6386 SE	3500 (4000) 2800 (3500)	6 16	8'192	4'000'000	482 (3.8.7)

For Instance: DUNE

http://dune-project.org/

- Framework for the solution of Partial Differential Equations
- Developed by working groups at the Universities of Freiburg, Heidelberg, Munster, the Free University of Berlin and the RWTH Aachen
- 13 Core Developers, many more developers
- Currently (April 2015) 184,820 lines of code
- Additionally e.g. 60,755 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

Programming Paradigms

- Functional programming (e.g. Haskell, Scheme)
  - Program consists only of functions
  - There are no loops, repetitions are realized via recursion
- Imperative programming
  - Program consists of a sequence of instructions
  - Variables can store intermediate values
  - There are special instructions which change the sequence of execution, e.g. for repetitions

Imperative Programming Models

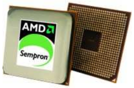
- Procedural programming (e.g. C, Fortran, Pascal, Cobol, Algol)
  - Program is divided into small parts (procedures or functions)
  - Data is only stored locally and deleted when the procedure exits
  - Persistent data is exchanged via arguments and return values or saved as a global variables
- Modular programming (e.g. Modula-2, Ada)
  - Functions and data are combined into modules that are responsible for the execution of particular tasks
  - These can in large parts be programmed and tested seperately

The Object-Oriented Programming Approach

In analogy to mechanical engineering:

- Splitting the program into independent components
- Determination of the necessary functionality required to provide this component
- All required data for this is managed within the component
- Components are connected via interfaces
- Using the same interface for specialized components which execute the same tasks

Example: Computer



Benefits

- The components can be developed independently
- If better versions of a component 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 mechanisms supporting this manner to structure a program:

Classes	define components. They 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 the data from other parts of the program
Inheritance	facilitates a uniform and common implementation of specialized components
Abstract base classes	define 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 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 methods and attributes
};
```

The class declaration defines the interface and the essential characteristics of the component
A class has *attributes* (variables to store the data) and *methods* (the functions provided by a class). The definition of attributes and the declaration of methods are enclosed in braces. After the closing brace comes 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.

Brooks (1975)

Encapsulation (II)

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

The keyword `public:` is followed by 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. It should **not** be possible to access stored data from outside the class to ensure data integrity.

Encapsulation (III)

```
struct MatrixClass
{
    // 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 `class` are `private`. If a class is defined with the keyword `struct`, as in `struct MatrixClass`, then all methods are `public` by default. Apart from that `class` and `struct` are identical.

Definition of Attributes

```
class MatrixClass
{
  private:
    std::vector<std::vector<double> > a_;
    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 nam The line is terminated with a semicolon.
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

C++11: New Datatypes

- 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)=1 <= sizeof(short) <= sizeof(int) <=
     sizeof(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]
```

- 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 value.
- `intptr_t` and `uintptr_t` supply data types with the right length to store a pointer.

C++11: Compiling Programs

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

- To translate a program with C++11 constructs the parameter `-std=c++11` should be given to `g++` as above. This holds for version 4.7 and above of `g++` and recent versions of `clang`, for older versions the parameter is called `-std=c++0x`.
- Information about the support of C++11 throughout the different versions of `g++` can be found at `http://gcc.gnu.org/projects/cxx0x.html`

Declaration of Methods

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

A method declaration always consists of four parts:

- the type of the return value
- the name of the function
- a list of arguments (at least the argument types) separated by commas and enclosed in parentheses
- a semicolon

If a method does not return a value, the type of the return value is `void`. If a method has no arguments, the parentheses remain empty.

Definition of Methods

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

The method definition (i.e. the 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 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 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.

Overloading of Methods

```
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) 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.

Constructors

```
class MatrixClass
{
  public:
    MatrixClass();
    MatrixClass(int numRows, int numCols);
    MatrixClass(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 the 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 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 an ordinary method, constructors can be defined inside or outside the class definition.
- Constructors can also be used to initialize attributes with values. The initializer list consists of the variable name followed by the value to be used for initialization (constant or variable) in parentheses separated by commas. It appears after the closing parenthesis of the argument, separated by a colon.

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

```
class MatrixClass
{
  private:
    std::vector<std::vector<double> > a_;
    int numRows_ = 0;
    int numCols_ = 0;
  public:
    MatrixClass();
    MatrixClass(int numRows, int numCols, double value);
    MatrixClass(int numRows, int numCols) :
        MatrixClass(numRows,numCols,0.0)
    {}
};
```

- C++11 additionally allows non-static members of classes to be initialized with a default valuein their definition. If the member appears in the initializer list of a constructor that 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 MatrixClass
{
  public:
    ~MatrixClass();
};
```

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

Default Methods

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

- Constructor without argument: `T();` (recursively calls the constructors of the attributes). The default constructor is only generated if no other constructors are defined.
- Copy constructor: `T(const T&);` (memberwise copy)
- Destructor: `~T();` (recursively calls the destructor of the attributes)
- assignment operator: `T& operator= (const T&);` (memberwise copy)
- address operator: `int operator& ();` (returns the storage address of the object)

Copy Constructor and Assignment Operator

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

- Sometimes one wants to prevent the generation of certain default methods, e.g. so that no objects of a class can be created when using only static attributes and methods.
- Until now 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 can be done easier and more clearly now with the keyword `default`.

Operator Overloading

- In C++ it is possible to redefine operators like + or − for a user-defined class.
- Operators are defined as 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.

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

- Operators can be defined both as a method of an object as well as ordinary (non-member) functions.
- The number of arguments depends on the operator.

Unary Operators

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

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

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

Binary Operators

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

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

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

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 `MatrixClass operator*(MatrixClass& A, double b)`, but not `A = 2.1 * A`
- There is a simple trick to implemtent both efficiently: one defines the combined assignment operator, e.g. `operator*=` for multiplication, within the class and two non-member functions outside 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 the signature `operator++()` that doesn't contain any argument. The argument `int` of the postfix version is not used and serves only to distinguish both alternatives.
- Note that the postfix operator cannot return a reference 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 MatrixClass
{
  public:
    double& operator()(int i, int j);
    std::vector<double>& operator[](int i);
    MatrixClass (int);
};
```

- The operators for square brackets and parenthesis 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 parenthesis can use an arbitrary number of arguments.
- Both can be overloaded several times.

Conversion Operators

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

- Conversion 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 type into which the operator converts (separated by a blank)
- 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 function of a class knows the object from which it has been called.
- Each function of a class 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:

```
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;
}
```

Example: Matrix Class

This example implements a class for matrices.

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

Header of the Matrix Class

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

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

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_ = 0;
    int numCols_ = 0;
};

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);
```

Implementation of the Matrix Class

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

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

Application using the Matrix Class

```
#include "matrix.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 (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
- Overloading of functions allows flexibility