Part 1: From Java to C++

Prof. Dr. Ulrik Schroeder

C++ - Einführung ins Programmieren

WS 03/04

http://learntech.rwth-aachen.de/lehre/ws04/c++/folien/index.html

"C++ is like teenage sex." **

- It is on everyone's mind all the time.
- Everyone talks about it all the time.
- Everyone thinks everyone else is doing it.
- Almost no one is really doing it.
- The few who are doing it are:
  - doing it poorly.
  - sure it will be better the next time.
  - not practicing it safely.

**) Graffiti found in a toilet stall at the faculty of Computer Science, Technion – IIT, Haifa, Israel on 8 November 1993.

„C++ wasn't designed to be a nice, pure language, good for teaching students how to program, ...

... it was designed to be a powerful tool for professional programmers solving real problems in diverse domains. The real world has some rough edges, so it's no surprise there's the occasional scratch marring the finish of the tools on which the pros rely."

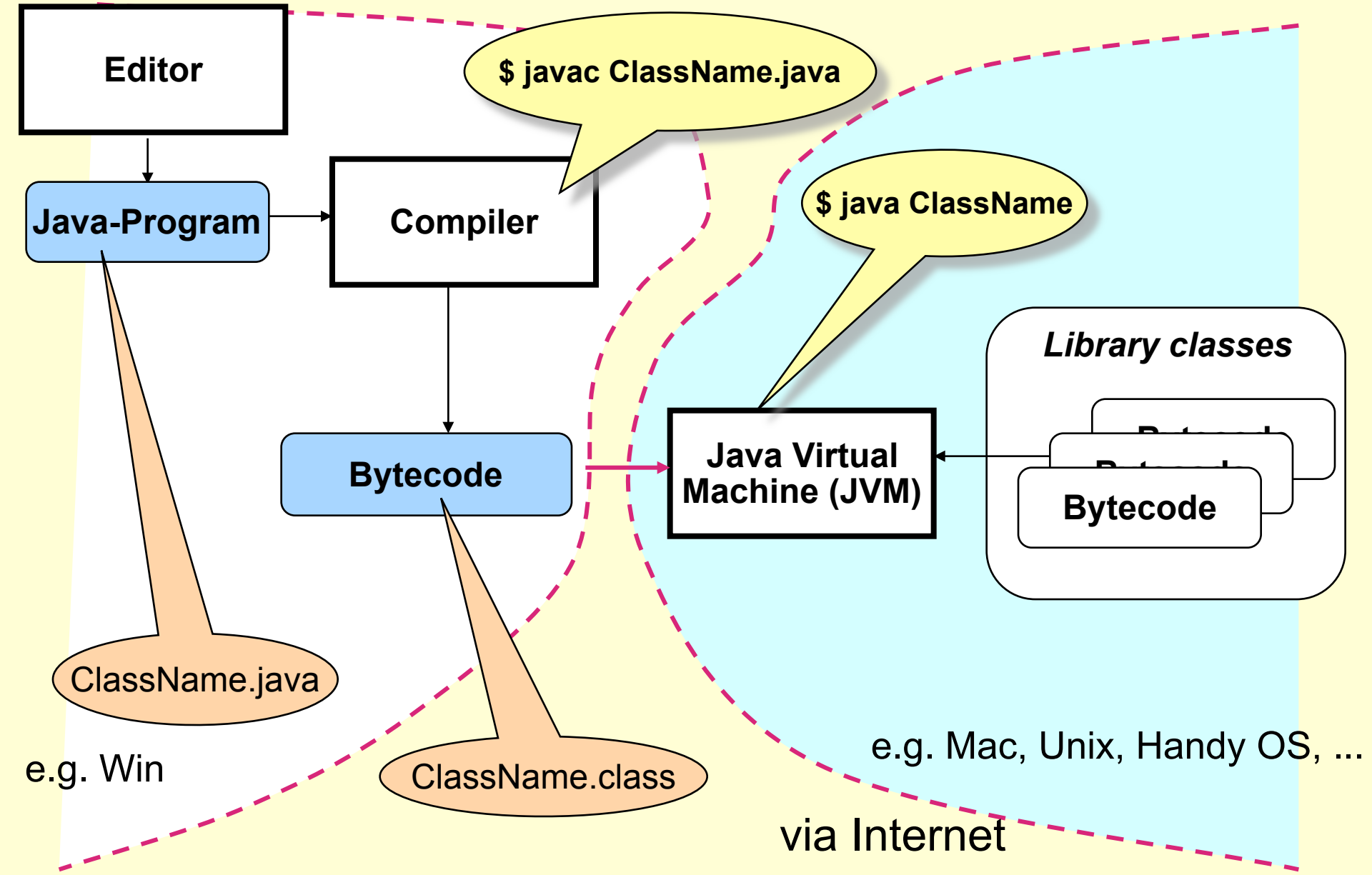
Important aspect: make your choice deliberately!

Think in Java, utilize lower levels of programming only when necessary.

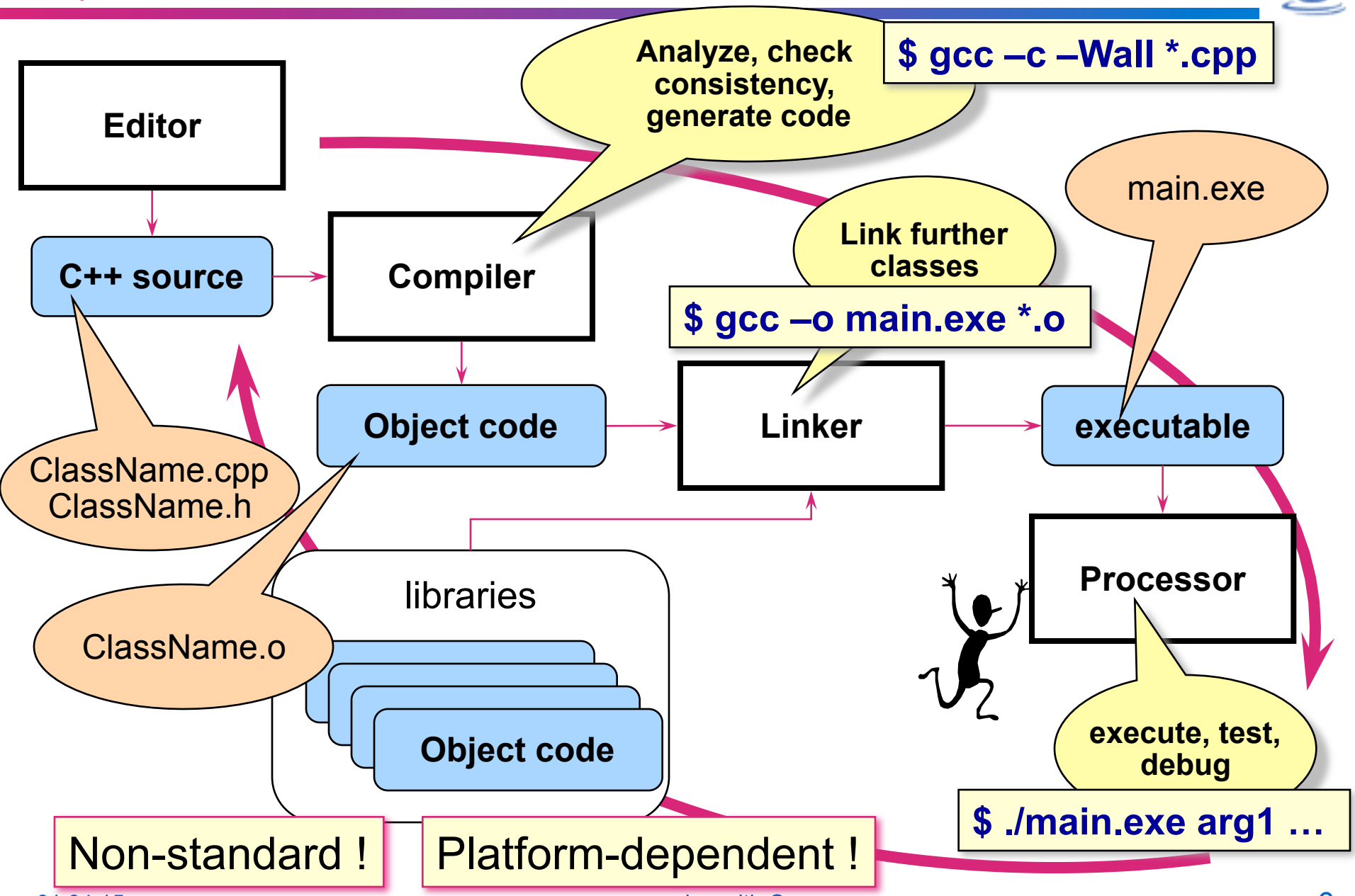
Content

1. Basic concepts, differences to Java, OO structure
   - program structure, classes, constructors, basics of inheritance
   - [formatted] IO, file IO, program arguments & conversion
   - basic types, strings, …
2. Pointers, references and memory management
3. Dynamic programming, recursive structures
4. C++ classes, method and operator definitions, inline and friend functions, operator overloading
5. Object-orientation, multiple inheritance, dynamic binding of virtual methods, template classes
6. STL -- standard template library

Java: compilation and invocation

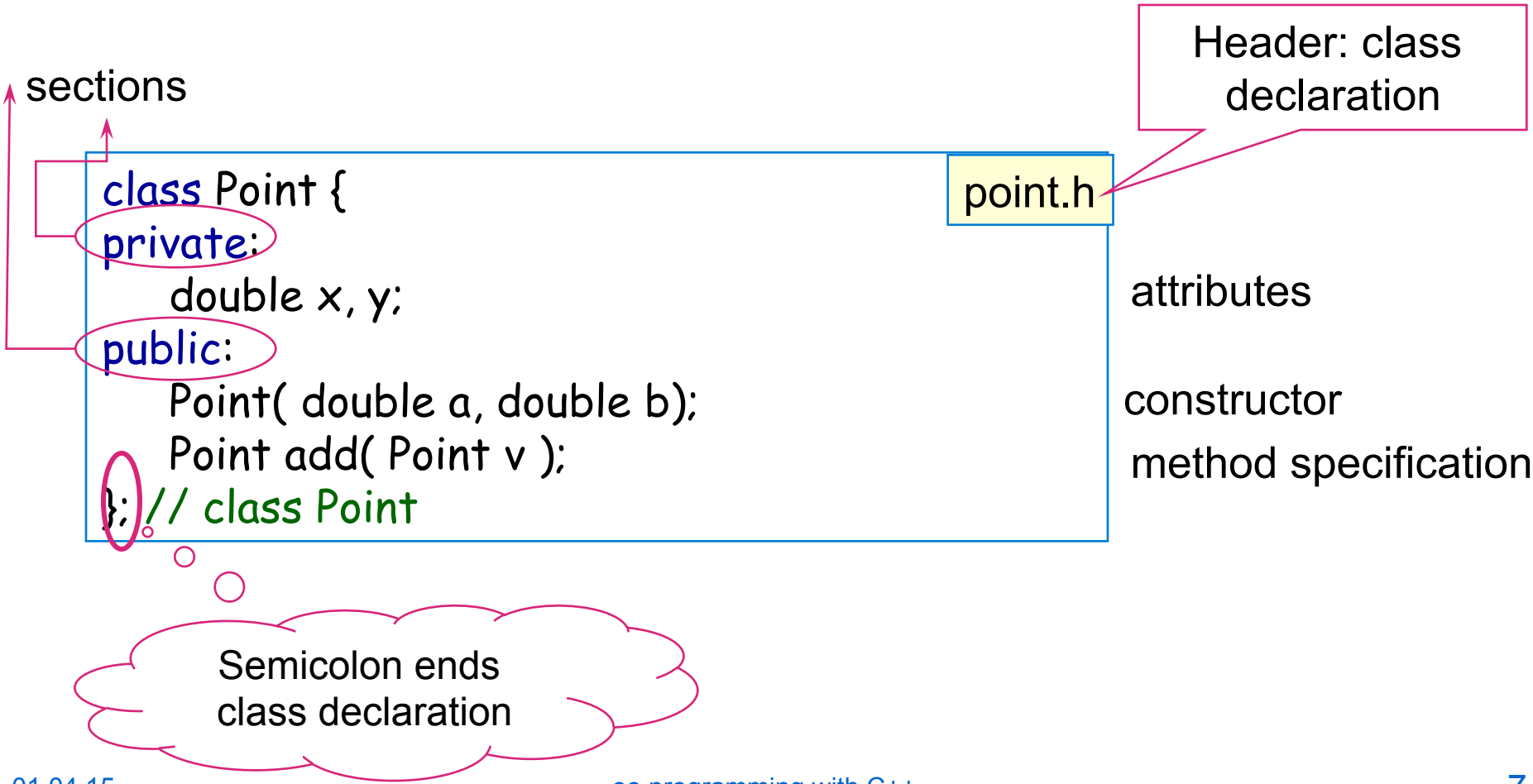


C++ programming cycle



Separation of declaration & definition

- Sufficient for compilation (compiler checks consistency within clients without reading definitions)
- Manually be kept up to date after changes (source of errors)



Definition of methods (of a class)

- Define methods in separate file
- Include header file (declaration)

Association to class by scope operator

point.cpp

```
#include "point.h"
Point:: Point( double a, double b): x( a ), y( b ) { };
Point Point::add( Point other ) {
    return Point( x + other.x, y + other.y );
} // Point::add( )
```

initialize attributes

returns new Point Object

point.h

```
#ifndef POINT_H
#define POINT_H
class Point {
private:
    double x, y;
public:
    Point( double a=0, double b=0);
    Point add( Point v );
}; // class Point
#endif POINT_H
```

Prevent multiple declarations

default values for arguments

StdConstructor Point( )
== Point(0) == Point(0, 0)

Preprocessor changes source before compilation (e.g. include, conditional compilation, …)

Main program

- global main method (not within any class)

point.cpp

```
#include „point.h"
...
Point Point::add( Point v ) { return Point( x + v.x, y + v.y ); }
void Point::print( ) { cout << "(" << x << "|" << y << ")"; }
```

instead of Java toString( ) method

could have also been defined in point.cpp

main.cpp

```
int main ( ) {
    Point u( 1.0 ), v( -0.5, 4.2 ), w;
    w = u.add( v );
    cout << "u = "; u.print( );  cout << endl;
    cout << "v = "; v.print( );  cout << endl;
    cout << "w = "; w.print( ); cout << endl;
    return 0;
} // main( )
```

object creation (without new), using three different constructors

System.out.println(…)

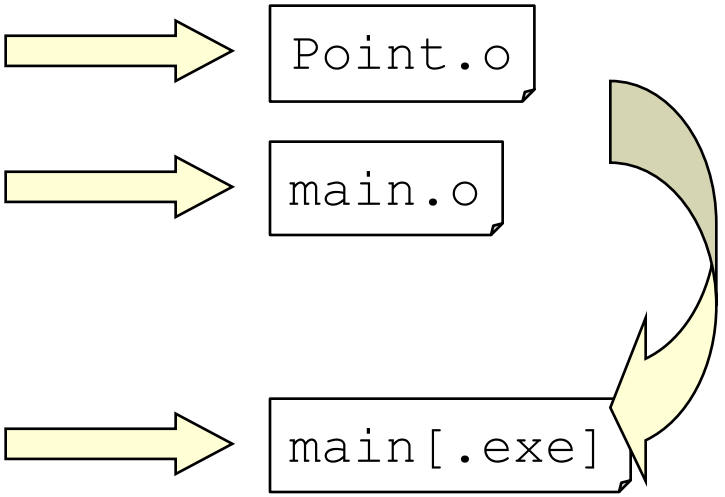
```
u = (1|0)
v = (-0.5|4.2)
w = (0.5|4.2)
```

Compilation & Linkage

- Compile each unit separately

```
$ gcc -c -Wall Point.cpp
```

```
$ gcc -c -Wall main.cpp
```



- Link program

```
$ gcc -o main main.o Point.o
```

- Execute & test program

```
$ ./main
```

- Shortcut: Compile & Link

```
$ gcc -o main main.cpp Point.o
```

Separation of declaration and definition

- Advantages
  - Emphasizes „information hiding principle“
    - Client reads declaration of class interface only, no implementation details, ...
      - matter of tools => javadoc, Eiffel IDE, ...
    - Header and object code can be delivered (without sources)
- Disadvantages
  - Information redundant, kept in separate places
  - Dangerous after changes
  - More responsibility for the programmer instead of tools

Summary class structure

- Class definition in header file
  - only prototypes of methods (except inline – later)
  - definition of private/protected/public section (attributes/methods)
  - constructors with initializer list
  - preprocessor directives to avoid multiple includes
- Class implementation
  - implementation of methods (associated to class by scope operator)
  - global functions (class correlation by scope operator)
  - default values for parameters
  - include definitions (preprocessor directives)
- Object instantiation without new operator
- Program code in *.o files, linked file in *.exe
  - 20 times larger than Java code

Inheritance (Java example)


```
public class Student {


 int key;
 int matrikelnr;
 boolean male;
 String vorname, nachname;


public String toString () {
 String anrede;
 if   (male) anrede = "Herr ";
 else        anrede = "Frau ";

 return anrede + vorname +
        " " + nachname; }

...}
```

```
public class Angestellter {


 String stellung;
 int key;
 boolean male;
 String vorname, nachname;


public String toString () {
 String anrede;
 if   (male) anrede = "Herr ";
 else        anrede = "Frau ";

 return anrede + vorname +
        " " + nachname; }

...}
```

Inheritance (Java example)

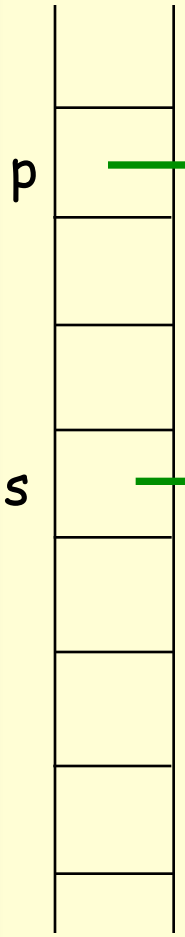

```
public class Student
    extends Person {

    protected int matrikelnr;
    ...
}
```

```
public class Angestellter
    extends Person {

    protected String stellung;
    ...
}
```

```
public class Person {
    protected int key;
    protected boolean male;
    protected String vorname, nachname;

    public String toString ( ) {
        String anrede;
        if   (male) anrede = "Herr ";
        else        anrede = "Frau ";

        return anrede +  " " + vorname +  " " + nachname; }

...} // class Person
```

Objects in class hierarchies

```
Person p = new Person ();

Student s = new Student ();


p = s;


System.out.println (s.key +
      ", " + s.matrikelnr);

System.out.println (p.key +
      ", " + p.matrikelnr);


s = (Student) p;
```



Objekt **Person**

Methode	Attribute
toString	int key String vorname String nachname boolean male

Objekt **Student**

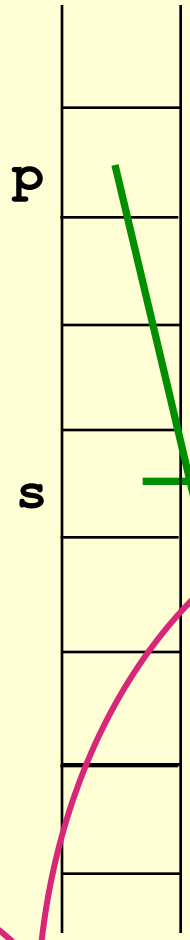
Attribut
int matrikelnr

Objekt **Person**

Methode	Attribute
toString	int key String vorname String nachname boolean male

Objects in class hierarchies

```
Person p = new Person ();

Student s = new Student ();
...

p = s;


System.out.println (s.key +
      ", " + s.matrikelnr);

System.out.println (p.key +
      ", " + p.matrikelnr);

System.out.println (p.key +
      ", "
+ ( (Student)p ).matrikelnr);
```



Objekt **Person**

Methode	Attribute
toString	int key String vorname String nachname boolean male

Objekt **Student**

Attribut
int matrikelnr

Objekt **Person**

Methode	Attribute
toString	int key String vorname String nachname boolean male

object itself still holds attribute matrikelnr !!!

Polymorph variables

- Greek expression for "various shapes"
- data structure of "similar objects"
  - traditionally: each object contains a tag field
    - switch statement to determine, which is the current object
- OO: each object brings its own, special method (dynamic binding)

```
for ( int i = 0; i < allObjects; i++ ) {
    System.out.println( personList[ i ] );
} // for each object
```

each person object can be printed as String

all objects bring their implementation how to be converted to String

Class hierarchy in C++

```
class Person { ... }; // class Person
```

inherit

```
class Student : public Person { ... };
```

use

exactly the same in C++ except for syntax

```
int main( ) {
    Student s(42, "Prefect", "Ford", false, 123456);
    Person p = s;
    s.display( );
    p.display( );
    return 0;
} // main( )
```

Person.cpp

```
void Person::display( ) {
    if ( female ) cout<<"Ms. "; else cout<<"Mr. ";
    cout << pre << " " << name << " (key: " << p.key << ").";
} // display( )
```

```
Mr. Ford Prefect (key: 42).
```

Redefining inherited methods

```
class Student : public Person {
public:
    Student( int k=0, string n=NULL, string p=NULL, bool f=true, int m=111111 )
        : Person(k, n, p, f), matnr(m) { };

    // redefinition of inherited methods (same signature!)
    void display( ) {
            cout << "Student " << pre << " " << " << name << " id= " << matnr << endl;
    }
    // additional methods
    int getMatnr( ) { return matnr; }
    void setMatnr( int m ) { matnr = m; }

protected:
    int matnr;
}; // class Student : Person
```

constructor of base class executed first

inline definitions of (small) methods

Polymorph variables

```
int main( ) {
    Student s(42, "Prefect", "Ford", false, 123456);
    Person p = s;
    s.display( );
    p.display( );
    return 0;
} // main( )
```

```
Student Ford Prefect - MatNr: 123456
```

No late binding without Pointer to object

(which we will introduce later)

```
Mr. Ford Prefect (key: 42).
```

Also: Method must be declared to be virtual in superclass

Virtual method

```
class Person {
public:
      Person( int k=0, string n=NULL, string p=NULL, bool f=true )
            : key(k), name(n), pre(p), female(f) { };

      virtual void display( ) {
        if ( female ) cout<<"Ms. "; else cout<<"Mr. ";
        cout<< pre <<" "<< name <<" (key: "<< key <<")."<<endl;
      } // display( )

protected:
      int   key;
      string name, pre;
      bool female;
}; // class Person
```

Can be redefined and bound dynamically

Abstract Methods

- A superclass can specify methods, which must be implemented with the specified signature by all their heirs
  - Abstract method (C++ pure virtual)
- The class is called abstract class or interface (if all methods are abstract)

```
class Comparable {
public:
    // to be implemented by heirs:
    virtual bool equal( const Comparable other ) = 0;
    virtual bool lessThan( const Comparable other ) = 0;
    // implementation of all other comparators
    virtual bool lessOrEq( Comparable other ) {
        return equal( other ) || lessThan( other );
    } // lessOrEqual( )
    ...
}; // class Comparable
```

Java: `abstract boolean lessThan( Comparable other );`

Will use the implementation of the heir!

Restrictions

```
#include "Comparable.h"
class Person : Comparable {
public:
    bool equals( Person other ) { return id == other.id; }
    ...
}; // class Person
```

- You can not create objects from abstract classes
  Comparable c; /* ERROR */
- An heir must implement all abstract methods or else it is also abstract
  - Person implements lessThan( ) and equals( )
- You need the concept of dynamic binding and polymorph variables
  - Will be discussed later in C++ after we know about Pointer to objects

Java vs. C++

- no convention for names (capital letter for classes …)
- method implementations separated from class definition and prototype declarations
- default values for arguments
- private/public sections
- preprocessor statements
- no automatic late binding of methods (needs virtual declaration and explicit reference semantics)
- Pure virtuals methods are abstract (=> abstract classes, no interfaces)

- program structure
  - hybrid language: there are global data structures and methods (like main) → violating information hiding principle
    - main is not a class method, …

IO in C++

- generic output stream with operator <<
- cout is declared in <iostream> within namespace std

```
#include <iostream>

using namespace std;

...

int main( ) {

      cout << "Hello!";

} // main( )
```

```
std::cout << ...
```

all declarations in <...> without .h are declared in namespace std!

Input

```
#include <iostream>
using namespace std;

int main( ) {
        int a;
        cout << "Enter value a= ";
        cin >> a;
        cout << "a^2 = " << a*a << endl;
} // main( )
```

These declaration are always necessary for IO (sometimes omitted on slides!)

what happens if you enter abc?

?

abc5

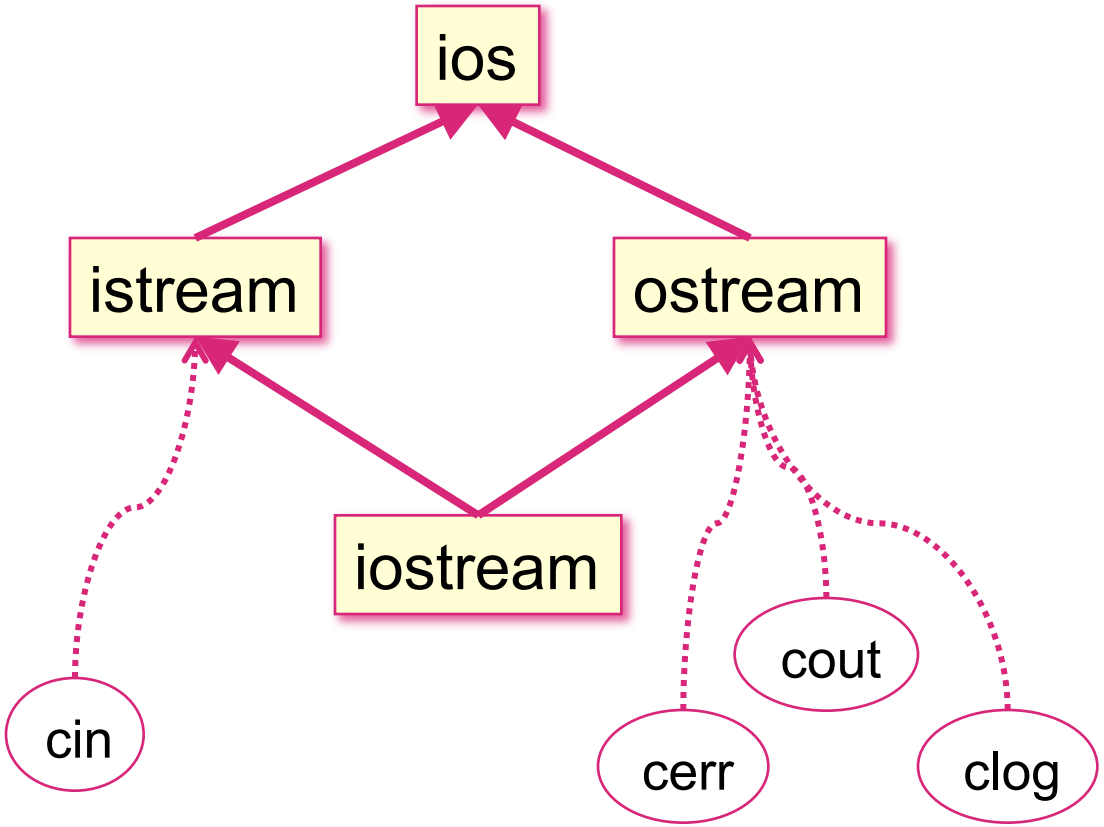
5abc

cout << "\n"

BTW: There is also cerr to print out error messages.

io class hierarchy

- base class ios for basic FLAGS
- ostream and istream
- iostream combines both via multiple inheritance



predefined objects

IO formatting and manipulators

- flags can be set via manipulators or procedures
- defining radix (base) of integers
- precision of floating Point values, fix Point, scientific ...

```
#include <iostream>
using namespace std;
int main( ) {
      int c;
      cout << "Please enter an integer value for variable c:\n   c = ";
      cin >> c;
      cout << "\toctal\tdecimal\thexadecimal\n"
           << "\t0_"     << oct << c
           << "\t"       << dec << c
           << "\t0x"     << hex << c <<  endl;
      return 0; // "normal" end of execution
} // main( )
```

Formatting Floating Point values

- precision (standard only 6 !!)
- showPoint, fixed, scientific

```
cout << 42.12345678 << endl;

cout.precision( 4 );

cout << "precision: " << cout.precision( ) << endl;

cout << 42.12345678 << endl;

cout.setf(ios::fixed);

cout << "fixed: " << 42.12345678 << endl;
```

⇨ 42.1235

⇨ precision: 4

⇨ 42.12

⇨ 42.1234

IO fields

#include <iomanip>

- width of output field for next argument
- set alignment
- define fill in character

```
cout << '|' << setw( 10 ) <<"xxx" <<'|' << endl;
```

```
|       xxx|
```

```
cout << '|' << setw( 10 ) << setfill('*')<<"xxx" <<'|' << endl;
```

```
|*******xxx|
```

```
cout << '|' << setw( 10 ) << left <<"xxx" <<'|' << endl;
cout << '|' << setw( 10 ) << internal <<"xxx" <<'|' << endl;
cout << '|' << setw( 10 ) << right <<"xxx" <<'|' << endl;
```

```
|xxx*******|
|****xxx***|
|*******xxx|
```

cout.setf(ios::left); ??? not supported ???

Formatted input

- similar manipulators

```
/* formatted input */
int i; double x;
cout << "Enter a hex value: ";
cin >> hex >> i;
cout << "i = " << i << " = 0x" << hex << i << endl;

cout << "Enter a floating Point value: ";
cin >> x;
cout << "x = " << x << endl;
```

ab → i = 171 = 0xab

123
123.456
123456e-3 → x=123.456000

File IO

- same concepts as interactive IO (inherit from iostream)
- open/close file, <<, >> operators

```
#include <iostream> // import std::cout with <<, endl
#include <fstream>  // import ifstream, open( ), get, EOF
using namespace std;
int main( ) {
    ifstream fin;
    cout << "============================fileio.cpp\n";
    fin.open( "fileio.cpp" );
    char ch; int i = 0;
    while( ( ch = fin.get( ) ) != EOF ) {
        i++;          cout << ch;
    } // while
    cout << "\nRead " << i << " bytes. Ciao.\n";
} // main( )
```

ofstream fout;

fout.open("copy.txt");

fout << ch

Example

What is the effect of this algorithm?

```
void display( ofstream out ) {

    out << "\n     ";       // header
    for( int f2 = 1 ; f2 <= 10 ; ++f2 )
      out << setw( 5 ) << f2;
    out << "\n    ----------------------------------------------\n";

    for( int f1 = 1 ; f1 <= 10 ; ++f1 ) { // content
      out << setw( 4 ) << f1 << " |";
      for(int f2 = 1 ; f2 <= 10 ; ++f2 )
        out << setw( 5 ) << f1 * f2;
      out << endl;
    } // for all lines of the table

} // display( )
```

```
$ ./1mal1.exe
          1    2    3    4    5    6    7    8    9   10
     ------------------------------------------------------
   1 |    1    2    3    4    5    6    7    8    9   10
   2 |    2    4    6    8   10   12   14   16   18   20
   3 |    3    6    9   12   15   18   21   24   27   30
   4 |    4    8   12   16   20   24   28   32   36   40
   5 |    5   10   15   20   25   30   35   40   45   50
   6 |    6   12   18   24   30   36   42   48   54   60
   7 |    7   14   21   28   35   42   49   56   63   70
   8 |    8   16   24   32   40   48   56   64   72   80
   9 |    9   18   27   36   45   54   63   72   81   90
  10 |   10   20   30   40   50   60   70   80   90  100
```

```
int main( ) {
    ofstream fout;
    fout.open( "datei.txt“ );
    display( fout ); …
```

Passing arguments to the program

- First argument is name of program (argv[0])
- argument count

```
#include <iostream> // import std::cout with <<, endl
using namespace std;
int main( int argc, char* argv[ ], char* env[ ] ) {
    cout << "Program: " << argv[ 0 ] << endl; // immer definiert!
    cout << argc << " arguments: ";
    for ( int i = 1; i < argc; i++ )
        cout << " " << argv[ i ];
    cout << endl;
} // main( )
```

```
$ ./arg eins zwei drei
Program: ./arg
3 arguments: eins zwei drei
```

Conversion of program arguments

```
#include <iostream> // import cout
#include <sstream>  // stringstream
using namespace std;
int main( int argc, char* argv[ ] ) {
      if ( argc > 1 ) {
            int i;
            istringstream conv( argv[ 1 ] );
            conv >> i;
            if ( conv.fail( ) ) return 1;
            cout << i << "^2 = " << i*i << endl;
      } // if enough arguments
      return 0;
} // main( )
```

$ square 5

"5"

5

$5^2 = 25$