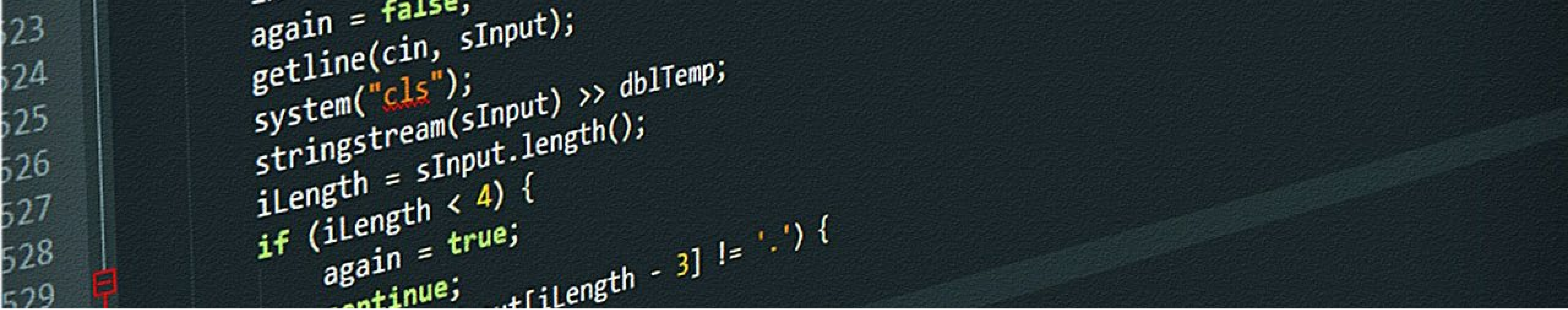


Thomas

C23-03 – Object orientation

Advanced algorithms and programming

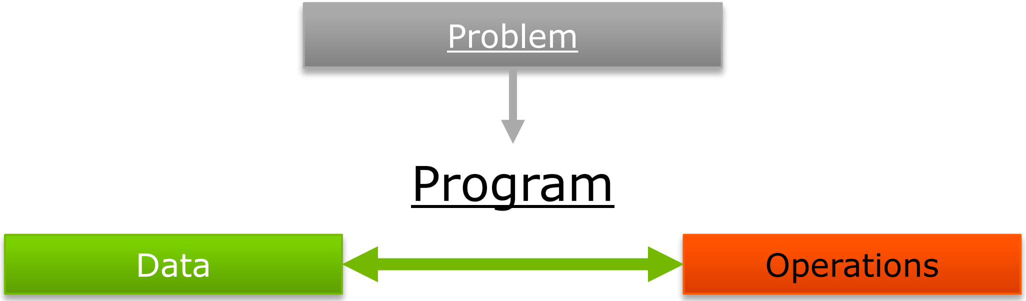
v5

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Films created using material from R. Peters and P. Puschmann / HTW Berlin

Object orientation
Introduction

- In software development, concrete problems (mostly from the real world) are modeled in the form of a computer program.
- A program essentially consists of (1) data and (2) operations that operate on the data.

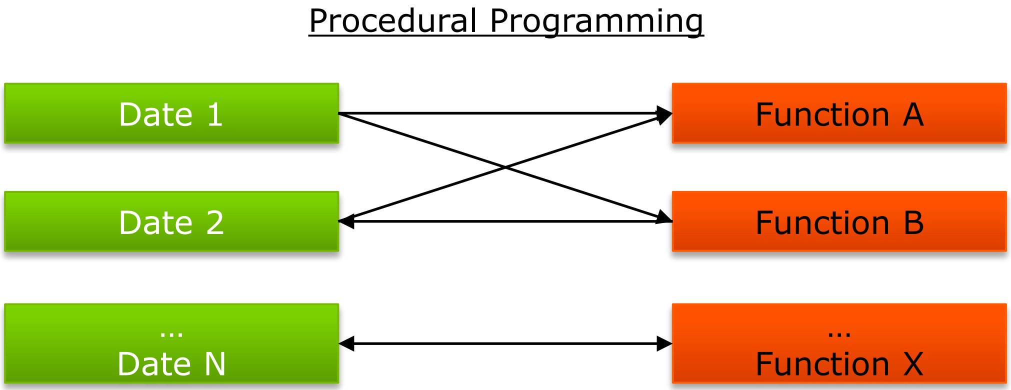


The basic approach can vary greatly in different programming languages.
(see programming paradigms)

Object orientation

Introduction – Procedural programming

- In procedural programming, data and functions are handled separately. In principle, all data types can be changed by all functions in the program.



Object orientation

Introduction – Object orientation

- In real life, there are many things / objects. These are distinguished mainly by:
  - Features
  - Functionalities (what you can do with them)
- Properties and functionalities belong together and define an object
- Often, an object is responsible for its own data and has functionalities to modify this data

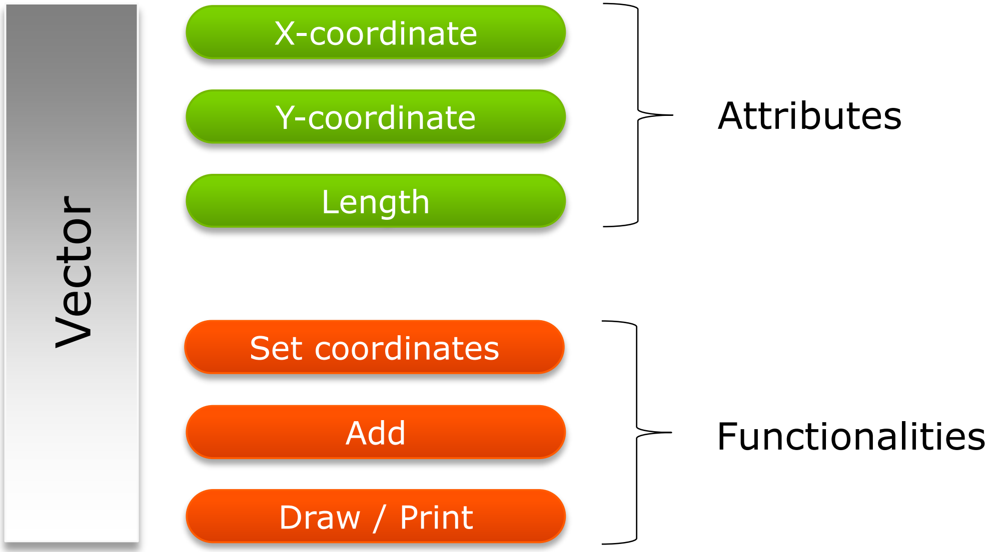
Object	Features	Functionalities
Car	Performance, number of doors, seats, speed, color, consumption, ...	drive, refuel, open / close door, park, flash, ...
Crayon	Length, color, condition of the tip, thickness, ...	paint, sharpen, ...
Matrix	Number of rows / columns, values	add, multiply, invert, ...
user account	Name, Password, E-Mail ...	create, log in, log out, delete, change data, ...

Object orientation

Introduction – Object orientation

Example - Two-dimensional vector

- The vector is an object with properties (attributes) and functionalities



Object orientation
Classes vs. objects

Classes

- Classes bundle data (attributes) and functions
  - Access to data "from outside" can be restricted (encapsulation)
  - Well defined classes ensure that their data is always valid and consistent (they "take responsibility" for the data)
- Classes help with a clear, modular structure of code
- Classes are extensible (via inheritance)

Objects

- Objects are concretizations (instances) of classes
- Objects have functions and data of the class, but the values of the data / attributes can differ from object to object
- Example: "Human" is the class. "Paul" and "Mary" are objects (instances of the class).

Object orientation
Classes

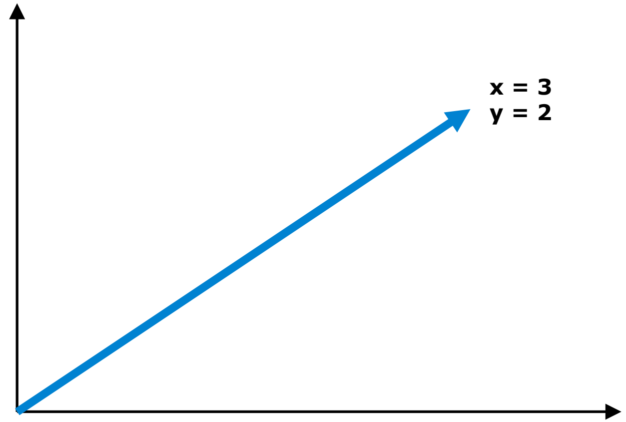
Structure
- Keyword: `class`
- Declaration and usage similar to a `struct`
- Functions within classes are called "method" or "member function
- Data within classes is called "member variables

Visibility of member functions and variables
- `public` visible outside
- `private` or no keyword visible only within the class

Object orientation
Classes – Example

Task

- Define a class `Vector`
  - Member variables for the positions x and y
  - Member function to set these values.



Object orientation
Classes – Example

Solution

```
class Vector
{
public: // publicly accessible area
 void setXY( double x, double y)
    {
 m_x = x; m_y = y;
    }

private: // private area
    double m_x, m_y;
};
```

Object orientation
Classes – Instances

- To use a class, an instance of the class must be declared, in C++ often called "creating an object".
- Any number of objects (instances) of a class can be created. Member variables are created for each object individually in memory [1].

- Instances of classes are essentially created like instances of basic types:

class_id var_id1, var_id2, ... var_idn;

Example:

```
int i1, i2;

Vector v1, v2;
```

[1] Exception: Static member variables are created only once per class. Details later in the course.

Object orientation

Classes – Member functions

- Member functions can be defined inside and outside the (Nota: Declaration always within the class).
- For definitions outside the class, the scope operator "::" must be used

```
class Vector
{
public: // publicly accessible area
 void setXY( double x, double y)
    {
 m_x = x; m_y = y;
    }

private: // private area
    double m_x, m_y;
};
```

```
class Vector
{
public: // publicly accessible area
 void setXY(double x, double y);

private: // private area
    double m_x, m_y;
};

void Vector::setXY( double x, double y)
{
 m_x = x; m_y = y;
}
```

Object orientation
Design Criteria

Note: These criteria are relevant for evaluation of exercises, project and exam.

Clarify competencies

- Determine "who" is responsible for what
- Select and apply relevant program structures and patterns (Doc - View, Model - View - Controller ...)
- Example: Encapsulate output and menu control in a user interface class

Protect data

- Guarantee consistency and meaningfulness of data at all times
- Prevent unwanted manipulations
- Use visibility mechanisms (`private`, `protected`)

Optimize design for changeability

- Implement design in such a way that as little source code as possible has to be changed for future changes

Classes

Access control

- Within a class, all member functions can access all data of the class.
- Outside of this, only data in the public area can be accessed.

- Within all member functions a `this` pointer is available. It always points to the address of the object that called the function. This pointer is passed (not visible to the programmer) as the first function argument.
- With the `this` pointer you can access all functions and data of an object within a class.
- The `this` - pointer does not have to be specified explicitly (with a few exceptions).
  It is used implicitly, if it is not specified!

Classes
Access control


```
class Vector2
{
public:
    // Compiler Intern: void setXY(Vector2* this, int x, int y);
    void setXY(int x, int y)
    {
        // 'this' pointer to distinguish local variable from member variable
        this-> x = x;
        this-> y = y;
        updateVectorLength(); // Access to private member functions
    }

private:
    void updateVectorLength()
    {
        // if 'this' is not specified, it is used implicitly!
        length = sqrt(x*x + y*y);
    }

    int x, y;
    double length;
};
```

Classes
Access control


```
int main()
{
    Vector2 vec;
    const int x = 2; // OK. Local variable has no influence on the class

    vec.setXY(x, 3); // COMPILER INTERNAL: setXY(& vec, x, 3)

    vec.length = 2; // COMPILER ERROR! (access to private member variable)
    vec.updateVectorLength() // COMPILER ERROR! (access to private function)
}
```

Classes
Access control


```
#pragma once

class Vector3
{

public:

    void setXY(int x, int y);           // Koordinaten setzen

    Vector3 add(const Vector3* pOther);     // addieren

    void print();                           // ausgeben

private:

    // private Helper-Funktion
    void updateVectorLength();

    int     m_x, m_y; // Kodierrichtlinie:
    double  m_length; // Prefix 'm_', um Membervariablen zu kennzeichnen

};
```

Classes

Access control


```
#include <iostream>
#include <cmath>
#include "Vector3.h"

void Vector3::setXY(int x, int y) {
    m_x = x; m_y = y;
    updateVectorLength();
}

Vector3 Vector3::add(const Vector3* pOther) {
    Vector3 result;
    result.m_x = m_x + pOther->m_x; // Eine Klasse hat private Zugriffsrechte
    result.m_y = m_y + pOther->m_y; // auf alle Objekte ihres Typs!
    result.updateVectorLength();
    return result;
}

void Vector3::print() {
    std::cout << "X-Koord: " << m_x << ", Y-Koord: " << m_y << ", Laenge: " << m_length << std::endl;
}

void Vector3::updateVectorLength() {
    m_length = sqrt(m_x * m_x + m_y * m_y);
}
```

```
#include "Vector3.h"

int main()
{
    Vector3 v1, v2;

    v1.setXY(1.0, 1.0);
    v2.setXY(.20, -2.0);

    Vector3 v3 = v1.add(&v2);
}
```

Klassen
Scope-Operator – Zugriff auf globale Funktionen

- Aufruf globaler Funktionen aus einer Klasse heraus über den Scope-Operator
- Die Klasse ‚Vector3' hat eine Member-Funktion mit dem Namen ‚add'. Falls eine gleichnamige Funktion außerhalb der Klasse existiert, kann diese nur über den Scope-Operator verwendet werden: ::add()

```
int add(int x, int y) { return x + y; } // globale Funktion

Vector3 Vector3::add(const Vector3* pOther)
{
    Vector3 result;
    // Die lokale Funktion Vector3::add überdeckt die globale Funktion!
    // Der Zugriff auf die globale Funktion muss über den Scope-Operator erfolgen
    result.m_x = ::add(m_x, pOther->m_x);
    result.m_y = m_y + pOther->m_y;
    result.updateVectorLength();
    return result;
}
```

Classes

Scope-Operator – Implementation of member functions

- The implementation of member functions differs slightly in syntax from normal functions. The class name is linked with the name of the member function using the scope operator '::'.

```
return_type Class_name::member_function( ... )
{
    // implementation
}
```

The scope operator is necessary because the elements of a class occupy a different namespace than global elements.

Classes

struct and class

- struct in C++ are very similar to class
- struct also can contain member functions
- struct can have access control with private, protected und public
- In struct, all elements are public by default

```
#include <iostream>

struct StructExample
{
    void print()
    {
        m_a = 3; m_b = 5;
        std::cout<<  "A = " << m_a <<", B = " << m_b <<std::endl;
    }
private:
    int m_a, m_b;
};

int main()
{
    StructExample se;
    se.print();
}
```

Classes

struct and class

- `class` could be used like `struct` if all members are made `public`
  (bad style – data should always be encapsulated)

```cpp
#include <iostream>

class PublicClassExample
{
public:
    void print() { std::cout<<  "A = " << m_a <<", B = " << m_b <<std::endl; }

    int m_a, m_b; // schlechter Stil: keine Datenkapselung!
};

int main()
{
    PublicClassExample pce = { 1, 4 };
    pce.print();
}
```

C++ basics
Default parameters

- Function parameter may have default values – so-called default parameters
  - Default parameters may be omitted when calling a function
  - No regular parameters are allowed behind the first default parameter

```
#include <iostream>

void printValues(int a, int b = 10, int c = 20, int d = 30)
{
    std::cout<<"a=" <<a <<", b=" <<b <<", c=" <<c <<", d=" <<d <<std::endl;
}

// Nicht erlaubt! Default-Parameter müssen immer rechts stehen:
// void printValues2(int a = 0, int b) {}

int main()
{
    printValues(1, 5);          // "a=1, b=5, c=20, d=30"
    printValues(1, 3, 8);       // "a=1, b=3, c=8, d=30"
    printValues(1, 4, 9, 20);   // "a=1, b=4, c=9, d=20"
}
```

C++ basics

References

- A reference referring to a variable is similar to a second name for this variable
- References are declared with ‚&' behind the variable type
- References must be initialized immediately after declaration

- References do not require memory (different to pointers)
- References must be of the same type as the referenced variable (`const` modifier is allowed)
- References do not extend the lifetime of variables (pay attention to function return values!)

```
int a = 10;
int& rA = a; // 'rA' refers to same content as'a'!
```

C++ basics
References


```
int value = 2;
int& rValue = value;
int* pValue = &value;

std::cout << "Original-Wert: " << value   <<", Adresse: " << &value  << std::endl
          << "Referenz-Wert: " << rValue  <<", Adresse: " << &rValue << std::endl
          << "Zeiger-Wert:   " << pValue  <<", Adresse: " << &pValue
          << ", Inhalt: " << *pValue << std::endl;
```

```
Original-Wert: 2, Adresse: 0x28fec8
Referenz-Wert: 2, Adresse: 0x28fec8
Zeiger-Wert: 0x28fec8, Adresse: 0x28fec4, Inhalt: 2
```

Adresse	**Inhalt**	**Variablen-Name**
28fec4h	28fec8h	pValue
28fec8h	2	Value, rValue

C++ basics

References

```cpp
int x = 1;
int& y = x; // Referenz auf x

y = 2;  // ändert auch den Wert von x

std::cout
    <<&x <<": x = " <<x <<std::endl
    <<&y <<": y = " <<y <<std::endl;
```

Output:

```
0x28fec8: x = 2
0x28fec8: y = 2
```

C++ basics
References

Negative example:

```
int& add(int x, int y)
{
    int z = x + y;
    return z;
}
```

The lifetime of ‚z‘ ends with the end of the function!

→ Return value is invalid
→ Program likely crashes

C++ basics

References – Best practice

- Useful for passing parameters to functions:
  Use "call by reference" for all non-basic data types (no int, double, float ...)
  - Parameter transfer as const reference, if data are not changed
  - Don’t use references as function return value!
    Exception: The data was previously passed into the function via "call by reference".

C++ basics
References

Useful example, considering the „best practices“:

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


std::string& addFileExtension(std::string& rFilename,
    const std::string& rFileExtension)
{
    rFilename += "." + rFileExtension;
    return rFilename; // OK: String-Objekt kommt von außerhalb.
}


int main()
{
    std::string filename = "Readme";
    std::cout << "Dateiname: " << addFileExtension(filename, "txt");
}
```

C++ basics

Function overloading

- In C++, several functions may have the same name

- Differentiation via list of function parameters
- Return value type not used for differentiation

C++ basics
Function overloading

```
#include <iostream>

void f(int a, double b)            { std::cout<<"F1" <<std::endl; }
void f(int a, int b)               { std::cout<<"F2" <<std::endl; }
void f(int a, int b, int c)        { std::cout<<"F3" <<std::endl; }
void f(int a)                      { std::cout<<"F4" <<std::endl; }
void f(unsigned int a)             { std::cout<<"F5" <<std::endl; }

// void f(int a, int b, int c = 0) { std::cout<<"F6" <<std::endl; }  Fehler (identisch mit F2 & F3)

void f(int a, float b)             { std::cout<<"F7" <<std::endl; }

// void f(int& a)                  { std::cout<<"F8" <<std::endl; }  Fehler (identisch mit F4):

void f(int* a)                     { std::cout<<"F9" <<std::endl; }

int main() {
    int var{ 1 };
    // F7        F2        F5       F4      F1            F3               F9
    f(1, 2.0f);  f(1, 2);  f(1u);   f(1);   f(1, 2.0);    f(1, 2, 3.0);    f(&var);
}
```

Constructors
Basics

Constructors

- ... are specific member functions of a class
- ... have the same name as the class
- ... have no return type (not even `void`)
- ... are called automatically when an object is created
- ... can be overloaded (i.e., several different constructors are possible)

- To be able to instantiate a class, at least one constructor must be declared `public`

Constructors
Basics


```cpp
#include < iostream>

class Box
{
public:
    Box(int width, int height, int length)
    {
        m_width = width;
        m_height = height;
        m_length = length;
        std::cout << "Box created: " << width << " x "<< height << " x "<< length << std::endl;
    }
private:
    int m_width, m_height, m_length;
};

int main()
{
    Box box{ 10, 20, 30 };
}
```

Constructors
Calling constructors

Initialize object via constructor:

```
Box box{ 10, 15, 12 };
Box box(10, 15, 12);
```

implicit notation

```
Box box = Box(10, 15, 12);
```

explicit notation

With dynamic memory allocation:

```
Box* box_ptr = new Box(1,2,3); // do not forget to delete!
```

For arrays:

```
Box boxes[] = { {1,2,3}, {10,2,10}, {20,4,10} };
Box boxes[] = { Box(1,2,3), Box(10,2,10), Box(20,4,10) };
```

For constructors without parameters (standard constructor):

```
Box box1;
Box kiste2 = Box();
```

Constructors
Standard constructor

- A standard constructor is a constructor that expects no parameters
- Implicitly created by compiler if no other constructor defined by the programmer

```
Class_name()
{

}
```

Constructors
Standard constructor


```
class Box
{
public:

     Box(int width, int height, int length) { }
};

int main()
{
     Box package; // ERROR: no default constructor defined!

 return 0;
}
```

Error: No standard constructor created automatically, because another constructor defined by programmer

Constructors

Standard constructor

```cpp
#include < iostream>
class Box
{
public:

    Box()
    {
        m_width = 1; m_height = 1; m_length = 1;
        std::cout << "Standard constructor called!\n";
    }

    Box(int width, int height, int length)
    {
        m_width = width;
        m_height = height;
        m_length = length;
        std::cout << "Constructor called!\n";
    }

private:
    int m_width, m_height, m_length;
};
```

```cpp
int main()
{
    Box box1;
    Box box2(1, 2, 3);
}
```

```
Standard constructor called!
Constructor called!
```

Constructors
Standard constructor


```cpp
#include <iostream>

class Box
{
public:

    Box(int width = 1, int height = 1, int length = 1)
    {
        m_width = width; m_height = height; m_length = length;
        std::cout << "Box created: " << width << "x" << height << "x" << length << std::endl;
    }

private:
    int m_width, m_height, m_length;
};

int main()
{
    Box box1 = Box(10, 3, 5);
    Box box2; // OK. (Own standard constructor available)
    Box box3(10, 3);
}
```

```
Box created: 10x3x5
Box created: 1x1x1
Box created: 10x3x1
```

Constructors
Copy constructor

- Copy constructors are used to initialize an object by another object

```
Class_name(const Class_name& other) { /* ... */ }
```

- Copy constructors are implicitly created by the compiler,
  if the programmer does not define a specific copy constructor

- Compiler-created standard copy constructors copy the value of all member variables to the new destination
  - Custom-defined copy constructor are required when using pointers or references as member variables

Constructors
Copy constructor


```
#include < iostream>
class Box
{
public:
    // own standard constructor
    Box()
    {
        m_width = 1; m_length = 1; m_height = 1;
        std::cout << "standard constructur called: " << m_width << " x " << m_length;
        std::cout << " x " << m_height << "\n";
    }

private:
    int m_width, m_length, m_height;
};

int main()
{
    Box box1; // standard constructor
    Box box2{ box1 }; // call to compiler-generated copy constructor
}
```

Constructors

Copy constructor


```
#include < iostream>
class Box
{
public:
 // own standard constructor
    Box()
    {
        m_width = 1; m_length = 1; m_height = 1;
        std::cout << "standard constructur called: " << m_width << " x " << m_length;
        std::cout << " x " << m_height << "\n";
    }

    // own copy constructor, comment-out to demonstrate built-in copy constructor
    Box(const Box& otherBox) {
        m_width =  otherBox.m_width;
        m_length = otherBox.m_length;
        m_height = otherBox.m_height;
        std::cout << "copy constructor called: " << m_width << " x " << m_length;
        std::cout << " x " << m_height << "\n";
    }
};
```

```
int main()
{
    Box box1 = Box(); // standard constructor
    Box box2{ box1 }; // copy constructor
    Box box3( box1 ); // copy constructor
    Box box4 = box1; // copy constructor
}
```

```
standard constructur called: 1 x 1 x 1
copy constructor called: 1 x 1 x 1
copy constructor called: 1 x 1 x 1
copy constructor called: 1 x 1 x 1
```

Constructors

Copy constructor – pointer problems

```cpp
#include < iostream>

class PointerExampleClass
{
public:
    PointerExampleClass(int val) { m_pPointer = new int(val);
}

    void print() { std::cout << "Value: " << *m_pPointer
            << std::endl; }

    void setValue(int val) { *m_pPointer = val; }

    void destroy()
    {
        std::cout << "memory in address " << std::hex
            << m_pPointer
            << " is released..." << std::dec << std::endl;
        delete m_pPointer;
    }

private:
    int* m_pPointer;
};
```

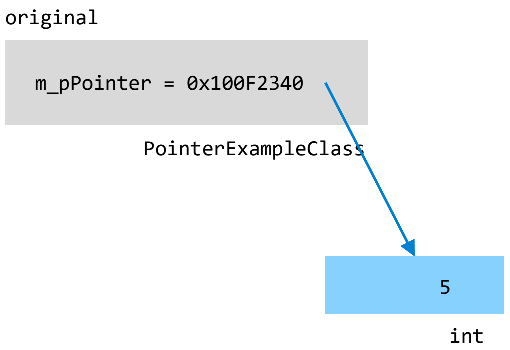
```cpp
int main()
{
    PointerExampleClass original(5);
    PointerExampleClass copy(original);
    original.print();
    copy.setValue(2); // also changes the original!
    original.print();
    copy.destroy();
    original.setValue(10); // -> probable crash!
    original.print();
    original.destroy(); // -> sure crash!
}
```

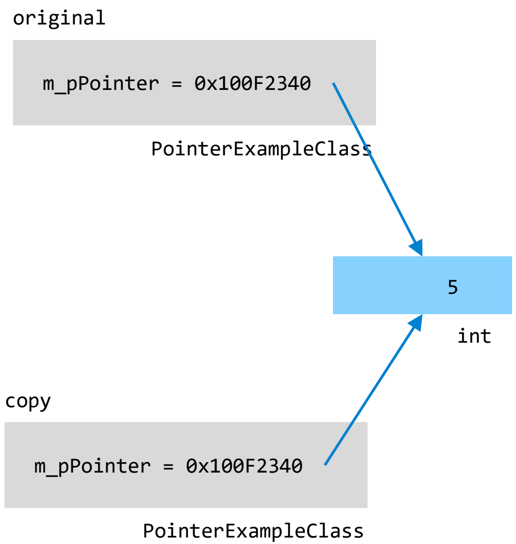
```
Value:  5
Value:  2
memory in address 00CF80B0 is released...
Value:  10
memory in address 00CF80B0 is released...
```

Both objects operate on the same memory area!
Must be avoided if not explicitly required!!!

Constructors

Copy constructor – pointer problems





Instance of `PointerExampleClass` with `int` variable allocated in heap

Copy of `PointerExampleClass` instance created with built-in copy constructor, using same `int` variable as original → problem!

Constructors

Copy constructor

By implementing the copy constructor, separate memory areas are used!

```
#include < iostream>

class PointerExampleClass
{
public:

    PointerExampleClass(int val) { m_pPointer = new int(val); }

// This copy constructor creates a separate memory for pointers:
PointerExampleClass(const PointerExampleClass& rOther)
{
    m_pPointer = new int(*rOther.m_pPointer);
}

void print() { std::cout << "Value: " << *m_pPointer << std::endl; }
void setValue(int val) { *m_pPointer = val; }
void destroy()
{
    std::cout << "memory in address " << std::hex << m_pPointer
        << " is released..." << std::dec << std::endl;
    delete m_pPointer;
}

private:
    int* m_pPointer;
};
```

Constructors
Initializer lists

- Initializer lists are used in constructors and initialize member variables!
- Initializer lists are required to initialize the following member variable types:
  - Constants
  - References
  - Member objects
  - Initialization of base classes

```
Class_name() :
 m_element1{init_expression}, m_element2{init_expression} [, ... ]
{
     // ...
}
```

Attention: The order in the initialization list is irrelevant.
The execution order depends on the order of declaration of the member variables!

Constructors

Initializer lists


```
#include < iostream>

class Box
{
public:

    Box() { std::cout << "Box created!" << std::endl; }

private:
    const int m_width, m_height;
    int m_length;
    int& m_rLength;
};

int main()
{
    Box box; // ERROR: Constants and refs must be initialized!
}
```

Attention: Constants and references must be initialized in the constructor!

```
Ausgabe anzeigen von: Build
1>------ Erstellen gestartet: Projekt: InitializerLists1, Konfiguration: Debug Win32 ------
1>InitializerLists1.cpp
1>C:\Users\admin\source\repos\C23-03\InitializerLists1\InitializerLists1.cpp(7,8): error C2789: Box::m_width: Ein Objekt eines durch eine Konstante qualifizierten Typs muss initialisiert werden.
1>C:\Users\admin\source\repos\C23-03\InitializerLists1\InitializerLists1.cpp(10): message : Siehe Deklaration von "Box::m_width"
1>C:\Users\admin\source\repos\C23-03\InitializerLists1\InitializerLists1.cpp(7,8): error C2789: Box::m_height: Ein Objekt eines durch eine Konstante qualifizierten Typs muss initialisiert werden.
1>C:\Users\admin\source\repos\C23-03\InitializerLists1\InitializerLists1.cpp(10): message : Siehe Deklaration von "Box::m_height"
1>C:\Users\admin\source\repos\C23-03\InitializerLists1\InitializerLists1.cpp(7,8): error C2530: "Box::m_rLength": Verweise müssen initialisiert werden
1>Die Erstellung des Projekts "InitializerLists1.vcxproj" ist abgeschlossen -- FEHLER.
========== Erstellen: 0 erfolgreich, 1 fehlerhaft, 0 aktuell, 0 übersprungen ==========
```

Constructors

Initializer lists


```cpp
#include < iostream>

class Box
{
public:
    // ERROR: Constants and refs must be initialized!
    Box()
    {
        m_width = 1; m_height = 1; m_length = 1;
        m_rLength = m_length;
        std::cout << "Box created!" << std::endl;
    }

private:
    const int m_width, m_height;
    int m_length;
    int& m_rLength;
};

int main()
{
    Box box;
}
```

Constructors

Initializer lists


```
#include < iostream>

class Box
{
public:

    // This one works
    Box() : m_width{ 1 }, m_height{ 1 }, m_length{ 1 }, m_rLength{ m_length }
    {
        std::cout << "Box created!" << std::endl;
    }

private:
    const int m_width, m_height;
    int m_length;
    int& m_rLength;
};

int main()
{
    Box box;
}
```

Constructors

Initializer lists – member objects

```cpp
#include < iostream>

class  ClassA
{
public:
    ClassA(int a, int b) : m_1{ a }, m_2{ b }
    {
        std::cout << "constructor classA: " << a << ","
            << b << std::endl;
    }

private:
    int m_1, m_2;
};

class ClassB
{
public:
    ClassB(int a, int b) : m_a2{ a, b }, m_a1{ 1, 1 }
    {
        std::cout << "constructor classB" << std::endl;
    }

private:
    ClassA m_a1; // Member object
    ClassA m_a2; // Member object
};
```

```cpp
int main()
{
    ClassB b(2, 2);
}
```

```
constructor classA: 1,1
constructor classA: 2,2
constructor classB
```

Note: See the order in which the constructors are processed!

Constructors
Conversion constructors

Conversion constructors

- ... convert another type into the type of the class
- ... have exactly one argument
- ... allow an implicit constructor call using the '=' operator
  (the keyword `explicit` can switch off this behavior)
- ... allow the compiler to perform conversions to the class type
  (the keyword `explicit` can switch off this behavior)

Beware of unwanted conversions!

Constructors
Conversion constructors

Example (part 1)

```
#include < iostream>
class C
{
public:
    C(int a, int b) : m_a{ a }, m_b{ b }
    { }
private:
    int m_a, m_b;
};

class ConvertExample
{
public:
    // rudimentary conversion constructors, just for illustration
    ConvertExample(int a) { std::cout << "Int-Constructor" << std::endl; };
    ConvertExample(double a) { std::cout << "Double-Constructor" << std::endl; };
    ConvertExample(const C& rC) { std::cout << "ClassC-Constructor" << std::endl; };
    explicit ConvertExample(const char* text)
    {
        std::cout << "String-Constructor" << std::endl;
    };
    void doSomething(const ConvertExample& e)
    {
        std::cout << "doSomething with 'ConvertExample'" << std::endl;
    }
};
```

Constructors

Conversion constructors

Example (part 2)

```
Int-Constructor
Int-Constructor
Double-Constructor
Double-Constructor
ClassC-Constructor
ClassC-Constructor
String-Constructor

Conversions:
Int-Constructor
doSomething with 'ConvertExample'
Double-Constructor
doSomething with 'ConvertExample'
ClassC-Constructor
doSomething with 'ConvertExample'
```

```
int main()
{
    ConvertExample a1 = 1;
    ConvertExample a2{ 1 };

    ConvertExample b1 = 2.0;
    ConvertExample b2{ 2.0 };

    ConvertExample c2 = C(1, 2);
    ConvertExample c1{ C(1, 2) };

    // ConvertExample d2 = "Hello World!"; // does not work, because of 'explicit':
    ConvertExample d1("Hello World!");

    std::cout << "\nConversions: " << std::endl;

    a1.doSomething(1); // implicit conversion (int -> ConvertExample)
    a1.doSomething(1.0); // implicit conversion (double -> ConvertExample)
    a1.doSomething(C(2, 2)); // implicit conversion (double -> ConvertExample)
    //a1.doSomething("Hello"); // does not work (because of 'explicit')
}
```

Destructors

- Destructors
  - ... are called as soon as the lifetime of an object ends
    (the object is destroyed or released)
  - ... to clean up within the class (e.g. release dynamically allocated memory)

- Only one destructor per class is allowed

```
~Class_name()
{
    /* clean up */
}
```

Destructors

Example
(part 1)

```
#pragma once

class DestructorExample
{
public:
    // Standard constructor with initializer list
    DestructorExample() : m_pPointer{ new int(0) } { }
    // Constructor with initializer list
    DestructorExample(int val) : m_pPointer{ new int(val) } { }
    // copy constructor
    DestructorExample(const DestructorExample& rOther);

    // destructor
    ~DestructorExample();

    int getValue();
    void setValue(int val);

private:
    int* m_pPointer;
};
```

Destructors

Example (part 2)

```
#include <iostream>
#include "Destructor1.h"

DestructorExample::DestructorExample(const DestructorExample& rOther)
    : m_pPointer{ new int(*rOther.m_pPointer) }
{ }

DestructorExample::~DestructorExample()
{
    std::cout << "memory in address " << std::hex << m_pPointer
        << " is released..." << std::dec << std::endl;
    delete m_pPointer;
}

int DestructorExample::getValue()
{
    return *m_pPointer;
}

void DestructorExample::setValue(int val)
{
    *m_pPointer = val;
}
```

Destructors

Example (part 3)

```cpp
#include <iostream>
#include "Destructor1.h"

int main()
{
    DestructorExample original(5);

    std::cout << "Value: " << original.getValue() << std::endl;

    {
        DestructorExample copy(original);
        copy.setValue(2);
        std::cout << "Lifetime of 'copy' ends here!" << std::endl;
    }

    std::cout << "Value: " << original.getValue() << std::endl;
    original.setValue(10); // OK.
    std::cout << "Value: " << original.getValue() << std::endl;
}
```

```
Value: 5
Lifetime of 'copy' ends here!
memory in address 01335BB0 is released...
Value: 5
Value: 10
memory in address 01335B80 is released...
```

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

www.htw-berlin.de