EBERHARD KARLS UNIVERSITÄT TÜBINGEN
FACULTY OF SCIENCE
Applied Bioinformatics

Programming in C/C++

- Object-oriented Programming -

Object-oriented Programming

- Introduction to OOP

Differences to imperative programming

- Traditional (**imperative**) programming languages (e.g. C)
- Programs are consequences of
  - Instructions and
  - Calling subroutines
- There is a **separation of data** and **subroutines / functions.**
- The **program sequence** is **centrally** controlled or coordinated (main program).

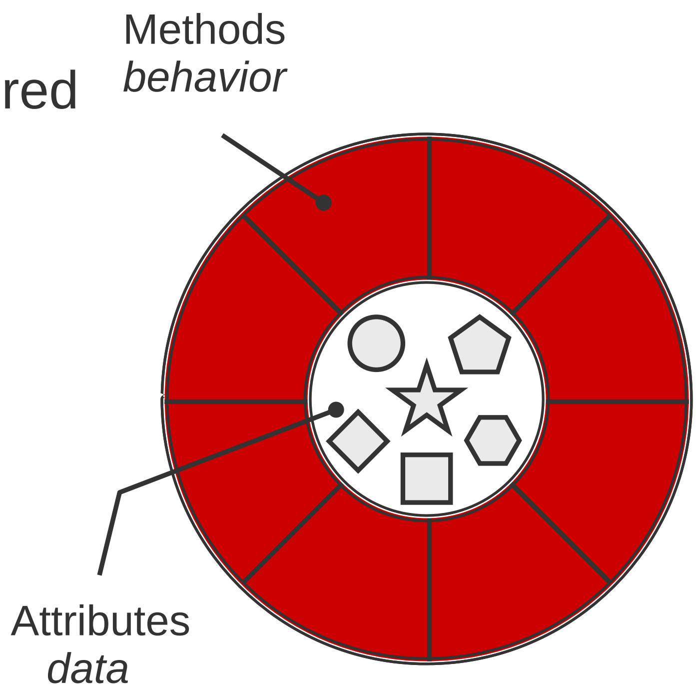
Introduction to Object-oriented Programming (2)

Differences to imperative programming – paradigm shift

- **Bundling and structuring of data:**
  - **Data** (= attributes) and **functions** (= methods) are **considered as a unit.**
  - They are combined into **objects.**

- **Information hiding:** Data can only be accessed / altered through object-specific methods (**encapsulation**).

- Communication between objects via **method calls**



Object-oriented Programming (OOP)

Motivation: Why OOP?

Simplifies building large and complex software projects that are subject to changes.

- **Easier adaptation to new**
  - Situations
  - Applications
  - Platforms
- **Reusability**
  - Don’t need to reinvent the wheel
  - Use of ready-made building blocks
- **Robustness**
  - Easier testing
  - Error handling

Object-oriented Programming (OOP)

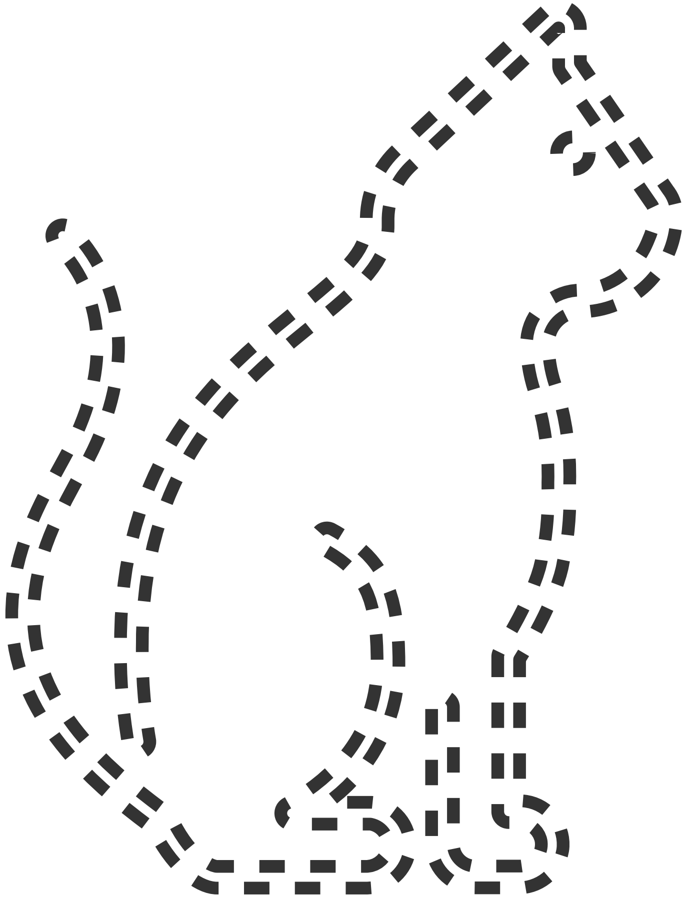
- In OOP or in the design of object-oriented software, classes with certain **attributes** as well as necessary **methods** (state change, object calculations and manipulations) are identified.
- Key structuring mechanisms are:
  - **Abstraction** - that is, the mapping of a section of the real world into a model in which only the important aspects of the real object are considered.
  - **Encapsulation** - i.e. the encapsulation of data from unwanted access / modification / manipulation.
- Other concepts of OOP that will be covered later are:
  - **Inheritance**
  - **Dynamic binding and polymorphism**
- Object-oriented languages, such as C++, offer various mechanisms for implementing the principles of OOP.

Class and (Object) Instance

- A **class** is a blueprint for several similar **objects**:
  - It defines the basic **properties** and **behavior** of its objects.
  - These **properties are only created abstractly**
    - The **class defines**, which **characteristics** its **objects will have**.
    - Only when objects are created they are assigned to **concrete values**.
- An **object** that is created from a class according to these specifications is called an **(object) instance of** this class.
- Each **instance** of a **class** carries is **own data**.

Class and (Object) Instance

class
(abstract)



class name

Cat

Attributes **(State)**

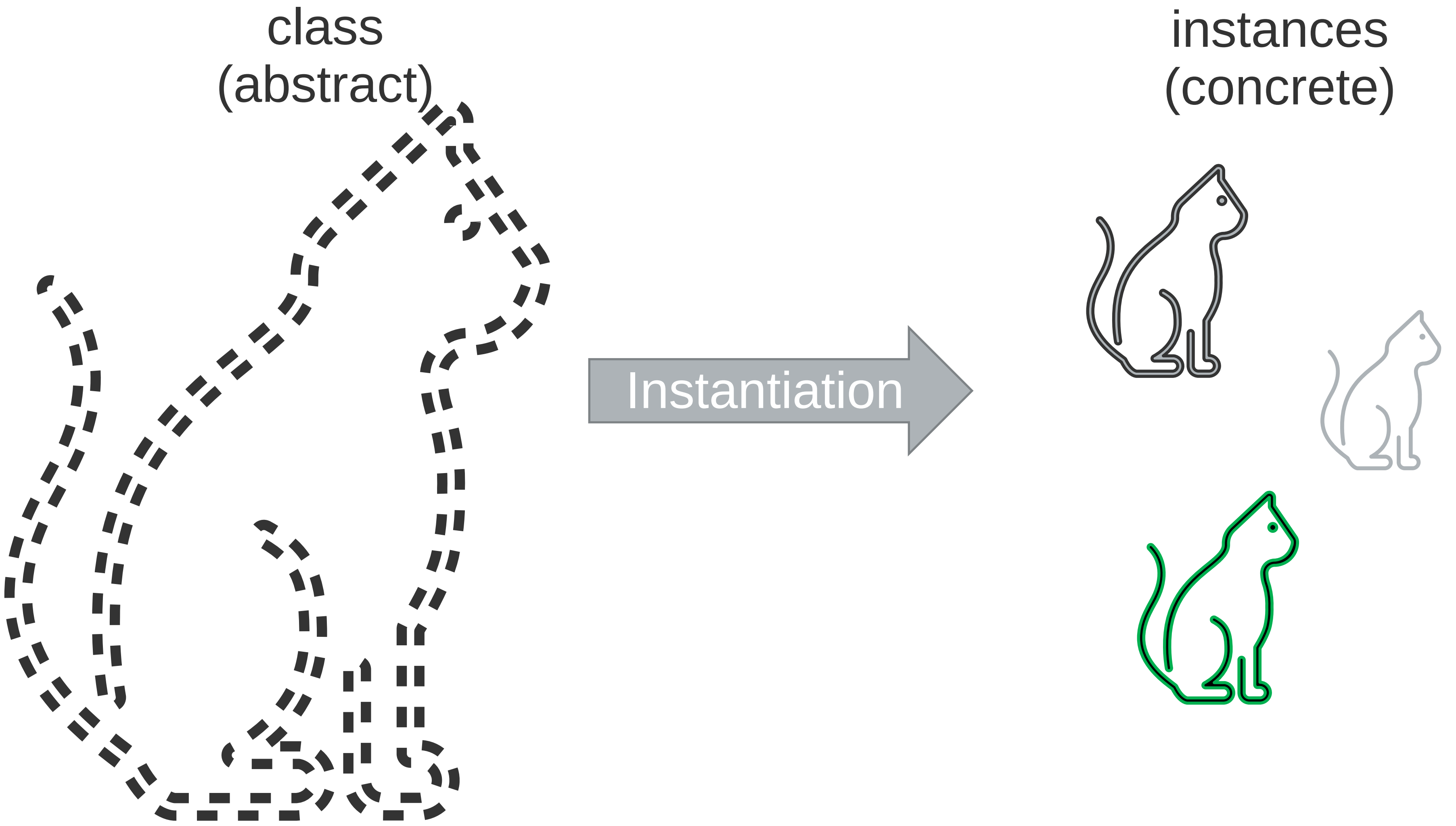
- color
#age : int = 0
+weight : float

Methods **(Behavior)**

-purr()
+pet()

access
+ public
- private
protected

Class and (Object) Instance



Class and (Object) Instance

Class
(abstract)

Class: Car

- License plate
- Color
- Chassis number
- Current steering direction
- Current gear
- Current speed

- Brake
- Accelerate
- Turn left/right
- Change gear
- Switch on headlights
- …

Instantiation

Instance
(concrete)

Object: mycar

- **License plate**: TÜ-WW 1234
- **Color**: red
- **Chassis nummer**: 149837498324798
- **Current steering direction**: 3° left
- **Current gear**: 2
- **Current speed**: 20 m/s

- Brake
- Accelerate
- Turn left/right
- Change gear
- Switch on headlights
- …

Example: `Adventurer.cpp`

Introductory / Code reading example

- Introduce language constructs/syntax (**constructor**, `public`, `private, ...`)
- Focus on **few** new concepts which will be discussed in detail later
- For more experienced programmers: think about what YOU would improve?

Adventurer:

- **name**, **health** and **skill** can be queried by the user
- Can **train** to improve his fighting skills
- If he get‘s **hit**, he/she looses health

Data representation

- Previously, we would have used primitive data types

- Example: <u>**one**</u> **Adventurer**

```
// properties of one adventurer as a set of variables
string name = "Bilbo";
double health = 5.0;
double skills = 2.0;
...
```

Data representation

- Example: **multiple adventurers**

```
// properties of each adventurer in ordered array
string* name = new string[3]{"Bilbo", "Frodo", "Sam", "Peregrin"};
double* health = new double[3]{5.0, 12.0, 20.0, 18.0};
double* skill = new double[3]{2.0, 9.0, 9.0, 6.0};
// later delete[] …
```

- **Problems:**
  - Construction and destruction of adventurers needs to be taken care of.
  - Difficult to keep track with indices because data of each adventurer is not bundled but spread out over multiple arrays.
  - Makes functions to modify specific adventurers more complex.

- **Solution:** OOP modelling with classes

```cpp
class Adventurer {
```

Adventurer.cpp

```cpp
    /** Attributes */
private:
    string name_; // name of the adventurer
    double health_; // current health of adventurer
    double skill_; // current skill of a.

public:
    /** Constructor */
    Adventurer(string _name, double _health, double _skill) {
        name_ = _name;
        health_ = _health;
        skill_ = _skill;
    }

    void train(double hours) {
            skill_ *= 1. + hours / 100.;
    }

}; // end class adventurer
```

Adventurer.cpp

```cpp
class Adventurer {

    /** Attributes */
private:
    string name_; // name of the adventurer
    double health_; // current health of adventurer
    double skill_; // current skill of a.

public:
    /** Constructor */
    Adventurer(string _name, double _health, double _skill) {
        name_ = _name;
        health_ = _health;
        skill_ = _skill;
    }

    void train(double hours) {
            skill_ *= 1. + hours / 100.;
    }

}; // end class adventurer
```

Adventurer.cpp

```
class Adventurer {

    /** Attributes */
private:
    string name_; // name of the adventurer
    double health_; // current health of adventurer
    double skill_; // current skill of a.

public:
   /** Constructor */
    Adventurer(string _name, double _health, double _skill) {
        name_ = _name;
        health_ = _health;
        skill_ = _skill;
    }

    void train(double hours) {
            skill_ *= 1. + hours / 100.;
    }

}; // end class adventurer
```

Adventurer.cpp

```
class Adventurer {

   /** Attributes */
private:
   string name_; // name of the adventurer
   double health_; // current health of adventurer
   double skill_; // current skill of a.

public:
   /** Constructor */
   Adventurer(string _name, double _health, double _skill) {
      name_ = _name;
      health_ = _health;
      skill_ = _skill;
   }

   void train(double hours) {
         skill_ *= 1. + hours / 100.;
   }

...
```

```
    void hit() {
        health_--;
    }

    double getFightFactor() {
        return skill_ * health_ / 10.;
    }

    string getName() {
        return name_;
    }

    double getHealth() {
        return health_;
    }

    double getSkill() {
        return skill_;
    }

    void print() {
        cout << name_ << " "
             << " has a health of " << health_
             << " and fighting skills " << skill_
             << "." << endl;
    }

}; // end class adventurer
```


Adventurer.cpp

```
bool simulateFight(Adventurer& adventurer, Adventurer& opponent) {
    while (adventurer.getHealth() > 0) {
      float r = (float)rand()/(1.+RAND_MAX); // rand. number: [0.0, 1.0]
      bool opponentHit = adventurer.getFightFactor() > r * opponent.getFightFactor();
      if (opponentHit) {
            healthOpponent--;
            if (healthOpponent < 0)return true; // opponent lost
        } else { // adventurer got hit
            adventurer.hit();
        }
    }
    return false; // adventurer lost
}

int main(int argc, char** argv) {
    srand(time(NULL)); /* initialize random seed: */

    Adventurer frodo("Frodo", 12.0, 9.0);
    frodo.print();

    frodo.train(60);
    cout << "Info: " << frodo.getName() << " "
          << " has improved his fighting power to "
          << ((int) frodo.getFightFactor()) << "." << endl;

    Adventurer shelob("Shelob", 12.0, 9.0);
    bool won = simulateFight(frodo, shelob);
    cout << "Info: " << frodo.getName() << " has "
          << (won ? "won" : "lost") << " the fight against his first monster." << endl;
   return 0;
}
```

Adventurer.cpp

Classes

- Declaration and Definition / Header and Source Files
- Constructor / Destructor
- Encapsulation: get/set methods
- Inheritance

Classes in C++

- Class types are declared with `class`
- Classes have:
  - **Attributes (data)**
  - **Methods (functions, operators, constructors/destructors)**
  - **Type definitions**

```
class Adventurer {

private:
    string name_;
...

public:
...
    Adventurer(string _name, int _health,
        int _skill) { ... }

    void train(double hours) {
            skills_ *= 1. + hours / 100.;
    }
}; // ; important
```

- Access rights defines what is visible:
  - `public` - visible to anyone
  - `protected` - visible in this class or derived classes
  - `private` - only visible in this class

Classes - Header Files

- In the Adventurer example, we declared and defined the class in a single .cpp file.
- C++ encourages **separate declaration** and **definition** in **header** and **source** files.

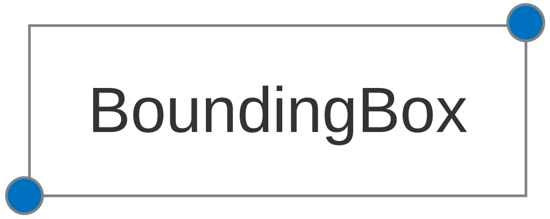
- **Declaration of** a class (announcement of the interface):
  - Declarations are typically stored in **header files** (`.h or .hpp`)
  - **Types** and **names** of attributes
  - **Signatures** and **names** of methods

  -> Enough information for the compiler:
    - to know size of type to allocate sufficient memory
    - perform function calls

- **Definition of** a class
  - Definitions are typically stored in **source files** (`.cpp`)
  - Actual **implementation** of the methods

Example: Bounding Box

- Box defined by two corner points:



- Pure declarations of class **`Point2D`** and **`BoundingBox`**, <u>no implementation:</u>

Point2D.hpp

```
class Point2D {
// private: attributes
    double d_x;
    double d_y;
public:
    Point2D(double _x, double _y); // constructor
};
```

BoundingBox.hpp

```
class BoundingBox {
    Point2D d_lowerLeft;
    Point2D d_upperRight;
public:
    BoundingBox(Point2D _lowerLeft, Point2D _upperRight);
};
```

Header Files

- Declarations of Point2D in `Point2D.hpp` and BoundingBox in `BoundingBox.hpp`

- BoundingBox needs to know about class Point2D:

BoundingBox.hpp

```
#include "Point2D.hpp" // "->look for file in local directory
```

- Common error: multiple declaration -> always prevent with header guards:

```
#ifndef POINT2D_HPP
#define POINT2D_HPP

...    // actual declarations here

#endif /* POINTS2D_HPP */
```

or shorter:

```
#pragma once
```

Working example: Separate compilation

main.cpp

```cpp
#include "example.hpp"

int main()
{
    example_func();
}
```

example.hpp
(declaration)

```cpp
#pragma once

void example_func();
```

example.cpp
(definition)

```cpp
#include <iostream>
#include "example.hpp“

void example_func()
{
    std::cout << "!!!";
}
```

- every pair of **.cpp** + **.hpp** is compiled separately (= a ***translation unit***)
- faster builds (only parts are rebuilt that changed)
- libraries used by many programs are shared (less memory used)
- but doesn't work for templates (later more…)

Note: Never make large namespaces in your **header** file(s) visible with e.g.

! `using namespace std;` because this will "pollute" the namespace in every file that (directly or through other headers) includes your header file.

Example: Bounding box

- So far pure **declaration** (**.hpp**)
- **Definitions** are now added to source file (**.cpp**):

Point2D.**hpp**
(declaration)

```
class Point2D {
// private:
    double d_x;
    double d_y;
public:
    Point2D(double _x, double _y);
};
```

Point2D.**cpp**
(definition)

```
#include "Point2D.hpp"
// definition of custom constructor
Point2D::Point2D(double _x, double _y)
{
 d_x = _x;
 d_y = _y;
};
```

- When we define class methods separately, we need to prefix methods with the class *name*`::`

Example: `Vector3d`

3-dimensional vectors

Vector3d
- x, y, z : float = 0.0
+ print() + add(other : Vector3d) : Vector3d + subtract(other : Vector3d) : Vector3d + multiply(s : float) : Vector3d + dot(other : Vector3d) : float + normalize()

- **Features**
  - Dimensionality: 3 (`float x, y, z`)
- **Standard operations**
  - Component addition, subtraction
    `v1 + v2 = ( v1.x + v2.x, v1.y + v2.y, v1.z + v2.z )`
  - Multiplication with scalar value
    `s * v1 = (s * v1.x, s * v1.y, s * v1.z)`
  - Scalar product (dot product)
    `< v1, v2 > = v1.x * v2.x + v1.y * v2.y + v1.z * v2.z`
  - Normalize to unit length

```
class Vector3d
{
  private:
    double x{}, y{}, z{};
  public:
    Vector3d(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}

    void print() { cout << x << " " << y << " " << z << endl; }

    Vector3d add(const Vector3d &_other) const {
        return Vector3d(x + _other.x, y + _other.y, z + _other.z);
    }

    Vector3d subtract(const Vector3d &_other) const {
        return Vector3d(x - _other.x, y - _other.y, z - _other.z);
    }

    Vector3d multiply(float s) const {
        return Vector3d(x * s, y * s, z s);
    }

    float dot(const Vector3d &_other) const { return x * _other.x + y * _other.y + z * _other.z; }

     void normalize() {
       const float len = std::sqrt(x_ * x_ + y_ * y_ + z_ * z_);
       x /= len;
       y /= len;
       z /= len;
     }
};
```

Constructor / Destructor / Assignment Operator

- Member variables **should** be correctly initialized and properly cleaned up.
- **Important**: if an object is in a valid and well-defined state after construction, we can easier ensure that method calls **keep** it in a valid state until the destruction.
  -> **class invariants**

- **Constructor**
  - Method is called when an object of the class is created.

- **Assignment Operator / `operator=(…)`**
  - Method is called to copy an existing object to **`*this`** (=the current instance).

- **Destructor**
  - Method is called when the life cycle of an object instance is ended (explicitly or implicitly).
    For example, at the end of the block it was created in.

Constructor

- **Constructors** are always named like the class
- No return value, no parameters

- Example (only declaration):

```
class Vector3d {
public:
    Vector3d();   // the default constructor
    Vector3d( double _x, double _y, double _z ); // a custom constructor
};
```

- Like functions, constructors can also have **default arguments**:

```
Vector3d( double _x = 0, double _y = 0, double _z = 0 );
```

Constructor

- Class can also be declared completely without a constructor. In that case, the compiler will <u>implicitly</u> **synthesize** constructors.

```
class Vector3d {
public:
    // <- compiler implicitly synthesizes default constructors
};
```

- We can also <u>explicitly</u> add the synthesized default constructor -> less code if other constructors prevent that the compiler adds an implicit default constructor.

```
class Vector3d {
public:
    Vector3d() = default;   // explicitly default-ed default constructor
};
```

Special Constructors

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Default constructor implementation**: like a normal method in cpp file

```
Vector3d::Vector3d() {
    x = 0;
    y = 0;
    z = 0;
}
```

- **Better:** List attributes in order of declaration, initialization value in brackets `()`:

```
Vector3d::Vector3d() : x(0), y(0), z(0)
{
}
```

```
Vector3d one;      // calls default constructor
Vector3d arr[200]; // calls default constructors
                   // because Vector3d is not a primitive type
```

- Detail: What would a synthesized (or `default`-ed) default constructor do?

```
class Vector3d
{
    double x{}, y{}, z{};
```

vs.

```
class Vector3d
{
    double x, y, z;
```

Destructor

- Automatically called to clean up at the end of the life of an object
- Exactly one destructor for each class ***`~classname()`***
- No return value, no parameters
- **synthesized by the compiler** (can also be `default`-ed)
- Frequent tasks:
  - Release of dynamic allocated resources (`delete, delete[]`)
  - Closing database connections, complete writing of buffered data, etc.

```
class A
{
  public:
    A() : data_(new int[10]) {} // user defined default constructor

    ~A() // user defined destructor
    {
      delete [] data_;
    }
  protected:
    int* data_;
};
```

Special Constructors - copy constructor

- **Copy constructor** (argument: other object of the class)
  - to create a copy
  - **synthesized by the compiler**: copy values of attributes (can be `default`-ed)

```
Vector3d::Vector3d(const Vector3d &_other) :
    x(_other.x),
    y(_other.y),
    z(_other.z)
{
}

...
Vector3d a;
Vector3d b(a); // constructs b using the copy constructor
```

Assignment Operator

- To be able to assign objects we need an assignment operator
- Similar to copy constructor (argument: other object of the class)
  - might need to clean up some internal state, e.g., free up previous members
  - create a copy, i.e. copy all data
  - returns reference to the updated object instance: `*this`
  - **synthesized by the compiler**: copy of the values of the attributes (can be `default`-ed)

```
Vector3d & Vector3d::operator=(const Vector3d &_other) {
    x = _other.d_x;
    y = _other.d_y;
    z = _other.d_z;
    return *this; // return reference to assigned instance
}
...
Vector3d a,b,c;
b = a; // calls assignment operator
c = b; // calls assignment operator
a = c = b; // why does this work?
```

Assignment Operator - Detail

- Assignment of primitive types return references to the assigned instance
- This allows us to write:

```
a = b = c = 0;
```

- Because the assignment operator is right-associative, this is equivalent to:

```
a.operator=(b.operator=(c.operator=(0)));
```

- Here you can easier see that the returning a reference allows to chain multiple assignments.
- To support this behavior for classes the signature of the assignment operator looks like this:

```
X& X::operator = (const X&)
```

Rule of Three

- **Rule of Three:**

  If a class needs a *user-defined* **copy constructor**, a *user-defined* **copy assignment operator** or a *user-defined* **destructor** -> the class most often needs all three!

- This is easy to understand:
  - If you need a *user-defined* destructor then you probably need to manually free some internal data structures.
  - These data structures need to be explicitly initialized in each constructor.
  - Because these data structures are not trivially copyable (a synthesized copy constructor or assignment operator don't work. They would just copy pointer instead of data.) you need to provide user-defined ones.

Classes as Types

- Same usage as primitive data types
- Declaration — definition (heap) — definition (stack)

```
class A;
```

```
A* ptrObjA = new A();
```

```
A objA;
```

- Initialization

```
A objA( myArg );
A objA = A( myArg );
```

...

- Assignment

```
A objB; objB = objA;
```

- Destruction (Destructor)

```
delete ptrObjA;
```

```
{ A objA; }
```

Parameter passing:

- As argument (by Value) — (by Reference) — (by Pointer)

```
void myFunc( A objA );
```

```
void myFunc( A & refObjA );
```

```
void myFunc( A *ptrObjA );
```

- As return value of a function

```
A myFunc()
```

Methods (non-static)

- **Recall:** Methods are functions that exist at the class level and define the operations available for its object instances.

- Methods **must** be declared in the class but **can** be defined in the class.

- The methods can access all attributes (class variables).

- Every object also contains an implicit attribute with name this - a pointer to the object itself.

```
void Vector3d::setXToFive()
{
  x = 5;        // also works: this->x = 5;
}
```

- Important: Methods always use to the values of the current instance.
- Methods and constructors can be overloaded like normal functions.

Operators

- Operators:
  - Like methods: invoked by name
  - **But:** they can also be invoked directly via their operator.

- User defined types act like built-in types.

- Be aware of the different return values:
  - `operator+=   -> reference to self`
  - `operator+    -> new object`

```
class Complex
{
  public:
    double re{};
    double im{};
    Complex & operator+=(Complex const & c)
    {
        re += c.re;
        im += c.im;
        return *this;
    }
    Complex operator+(Complex const & c)
    {
        Complex tmp{re, im};
        tmp += c;
        return tmp;
    }
};
c2.operator+=(c1);  // invoke via name
c2 += c1 + c2;      // invoke via +
```

Recommendation: only define **arithmetic operators** if meaning is obvious – e.g., without reading the class documentation.

Encapsulation: get / set methods

- Usually, data of an object should be hidden from direct access from the outside.

- Advantage:
  - Representation of data internally can be changed without the interface having to change
  - Easier to ensured that object is always kept in a valid state.

- Common approach:
  - all attributes are **`private`** or **`protected`**
  - Provide **`public`** methods:
    Accessors to **`get`**.. and **`set`**.. the values.

```
class A
{
  public:
  A : a(0) {}

  void setA(int b) { a = b; }
  int getA() { return a; }

  private:
  int a;
};

A a;
int x = a.a; // error, private!
int y = a.getA(); // OK, public
a.setA(4); // OK, public
```

Encapsulation: const Methods

- Setter change class attributes.
- Other accessors could return references to class attributes and potentially allow changes from outside.
- If we defined our object as const we can't use these methods.
- Example on the right:
  error: passing `const String' as `this' argument of `std::string& String::getData()' discards qualifiers
- Like variables we can also declare methods as `const`.
- `const` methods can't change class attributes -> e.g., use for get methods.

Recommendation: make all methods that don't change class attributes const.

```
class String
{
  public:
  int size()
  { return s.size(); }
  string& getData()
  { return s_; }

  protected:
  string s_;
};
…
const String s;
s.getData() = "Hi!";
```

Access to Attributes and Methods - example

```
class A;
```

- A class introduces a new namespace, here A.
- Access to the namespace for definitions **::**

```
A::myFunc() { ... };
```

- Access to public attributes / methods of an object **.**

```
A objA;

objA.myAttrib = 4;
objA.myFunc();
```

- Access via pointer **->** or **(*).**

```
A *objPtr = & objA;

objPtr->myAttrib = 4;
objPtr->myFunc();
```

or

```
(*objPtr).myAttrib = 4;
(*objPtr).myFunc();
```

Access to Attributes and Methods - example

```
class A {
public:
    int x;
    bool isZero() { return x == 0; }
};


...
A objA;

objA.x = 3; // write
if (!objA.isZero())
  cout << "objA is not zero!" << endl;

// access by pointer
A *objPtr = &objA;
objPtr->x = 0; // write via ptr
if (!objPtr->isZero())
  cout << "objPtr is not zero!" << endl;
```

Static Methods and Attributes

- A class can also provide functions that are **not linked to a special instance**, so-called **static methods**
  - Think of it as a **function** that **just exists in the namespace of the class**.
  - A static method can’t access (non-static) class attributes of an instance
- **static attributes:** they exist only once for the whole class, are only constructed once, and are the same (global) for **all** class instances.
- Detail: Construction of static attributes happen in a thread-safe way.
- Keyword: `static`

Example:

```
class A {
    static int s_numberInstances;
public:
   static int getNumInstances() { return s_numberInstances; }
   A() { s_numberInstances++; }
};

int A::s_numberInstances = 0; // init. is mandatory

int main() {
   A a, b, c;
   A arr[10];
   cout << "created " << A::getNumInstances() << " instances" << endl;
}
```

independent of a specific instance

Conversion: Constructors and Operators

- **Recall**: primitive types are implicitly converted.

- **Conversion constructor** (**one** argument: object of other class we want to convert **from**)

```
class B {…}
class C
{
  public:
      size_t j{};
      explicit C(B const & b) { j = b.i; }      // enforce explicit conversion from B to C
};

B b;
C c{b};
// C c = b; // error: no explicit call of the conversion constructor

void foo(const C & c) { /*...*/ }
foo(C{b});
// foo(b);   // error: no explicit call of the conversion constructor
```

- Without `explicit` conversion happens implicit which often leads to hard to debug errors.
- **Danger zone:** `Vector3d( double _x = 0, double _y = 0, double _z = 0 );`

Conversion: Constructors and Operators

- **Conversion operator** (argument: object of other class we want to convert **to**)

```
class A {
  public:
     size_t i{};
     explicit operator C() const { C c; c.j = a.i; return c; }
}

A a;
C c{a};     // conversion operator C() in class A is used to convert a to C
// C c = a; // error: no explicit call to the conversion operator C()

void foo(const C & c) { /*...*/ }
foo(C{a});
// foo(a);   // error: no explicit call of the conversion operator C()
```

- Without `explicit` conversion happens implicit which often leads to hard to debug errors.

Recommendation: If possible, try to avoid implicit conversion by using the `explicit` keyword.

Type definitions

- We can define synonyms for existing types: `typedef int IndexType;`
- Better readability
- Types can also be defined inside the scope of a class:
  - *nested classes*
  - `typedef`
- Type can be used from outside of the class with the qualified name *class::type*
- Can be `public, private, protected`

- Used a lot in STL and template code

```
…
typedef int MyIntegerType;


class MyIntegerContainer
{
  public:
  typedef int value_type;

  typedef vector<int>::iterator iterator;

  void insert(value_type v);
  void erase(value_type v);

  iterator begin();
  iterator end();
...

};
…
MyIntegerContainer::iterator it;
```

Class relations

So far, we looked at:

Association

- The interaction and communication between classes

Aggregation (or composition)

- "has a" relationship
- A class has an object of another class as an attribute

Now:

Generalization and inheritance

- "is a" relationship
- Inheritance from the more general to the more specific class

Association

- Interaction and communication between classes.
- Example: Using another class as a parameter, calling `public methods`

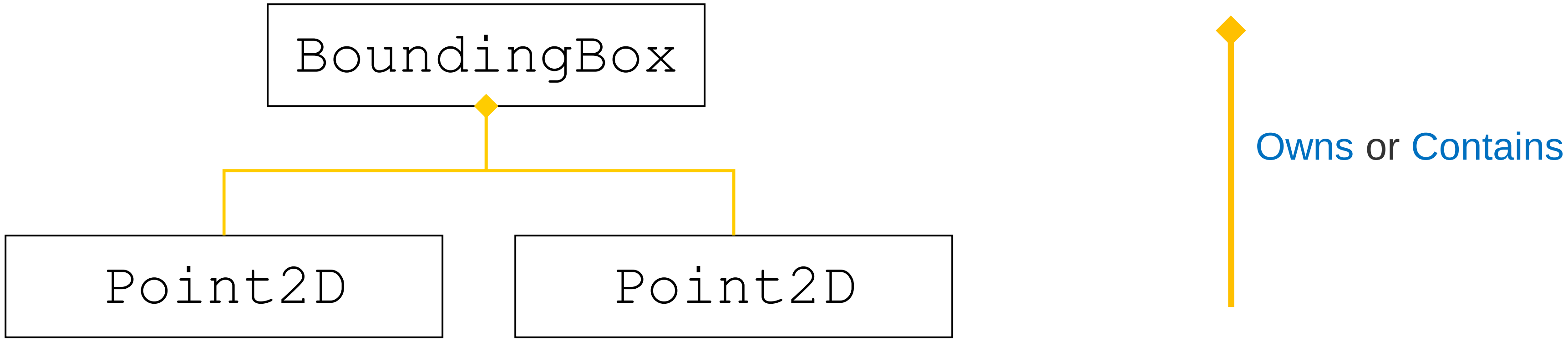
```
bool BoundingBox::enclose( Point2D extrema[] ) { ...
 lowerLeft = extrema[i].min( lowerLeft );
 upperRight = extrema[i].max( upperRight ); ...
  if ( lowerLeft.isSmaller( upperRight )) { ...
```

```
class Point2D { ...
public:
  Point2D min( const Point2D& _oPoint ) const;
  Point2D max( const Point2D& _oPoint ) const;
};
```

Aggregation (Composition)

- Class A is part (attribute) of B
- "*has a*" relationship

- Example: BoundingBox has/owns two Point2D




```
class BoundingBox {
 Point2D d_lowerLeft;
 Point2D d_upperRight;
public:
 BoundingBox( Point2D _lowerLeft, Point2D _upperRight ); ...
};
```

Attribute constructors

- **Recall**: The constructor ensures that class attributes are initialized

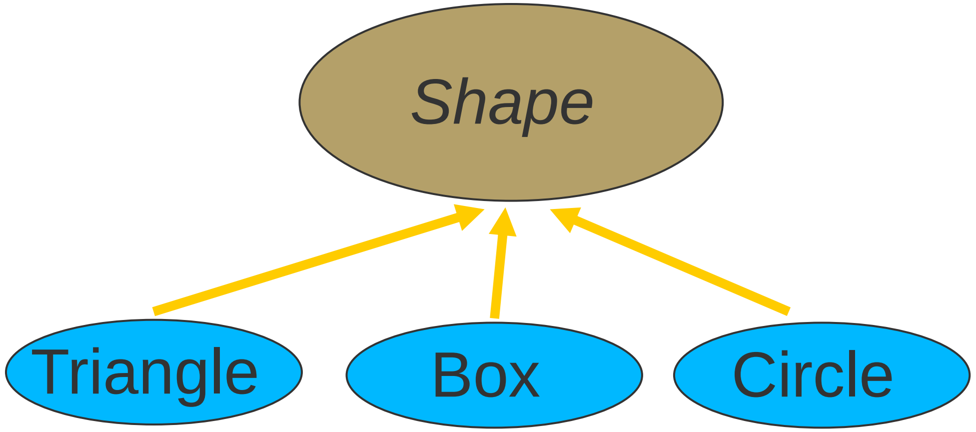
```
class BoundingBox {
    BoundingBox(Point2D _lowerLeft, Point2D _upperRight) :
        d_lowerLeft(_lowerLeft),
        d_upperRight(_upperRight)
    {
        // body of the constructor
    }
};
```

- Initialization of the attributes happens before executing the body of the constructor

Inheritance

- We can build hierarchies of classes using inheritance.

- Example:



  - Triangle, Box, Circle are also Shapes at the same time
  - Shape introduces the basic functionality
  - Triangle, Box, Circle implement these differently

Class relationship - generalization and inheritance

- Inheritance
  - The child (derived) class inherits methods and attributes from the parent class (base class).
  - The derived class is also **an instance** of the base class and can be used like it
  - In addition, the derived class can have additional elements, or change the behaviour of existing methods

- **Base classes describe the general case, derived classes special cases.**

- Example:
  - Class Vector2D extends the class Point2D

```
class Vector2D : public Point2D;
```

Access rights - overview

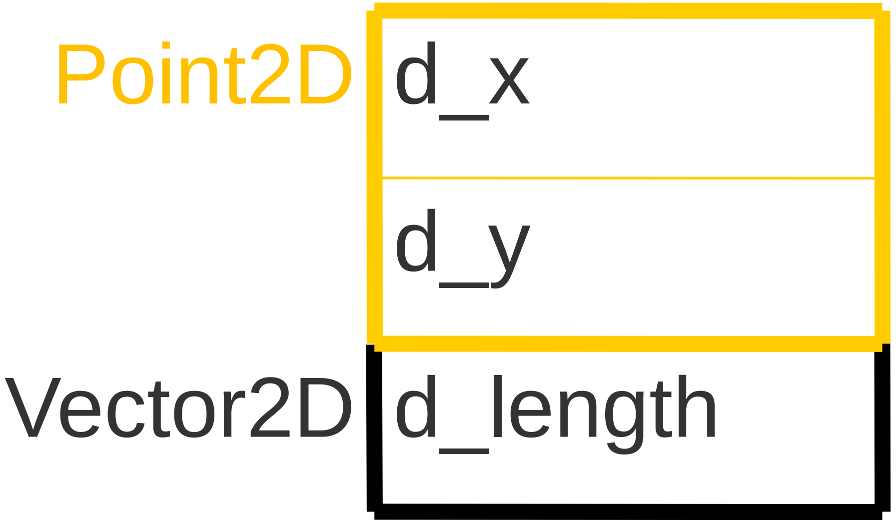
```
class Vector2D : public Point2D; // public inheritance
```

Access in a base class	Access in a derived class		
	Public Inheritance	Protected Inheritance	Private Inheritance
`private`	*Not accessible*	*Not accessible*	*Not accessible*
`protected`	`protected`	`protected`	`private`
`public`	`public`	`protected`	`private`

Layout of the derived class

- The object of the derived class contains an object of the parent class
- Methods of both classes can be called (as long as they are not protected by access operators)

- Example: `class Vector2D : public Point2D;`



Constructor and Destructor of the Derived Class

- Constructor of the derived class calls the base class constructor first.
- Then initializes the attributes of the derived class

```
class Vector2D: public Point2D {
        int d_length;
public:
        Vector2D(double _x = 0.0, double _y = 0.0) :
            Point2D(_x, _y)
        {
            d_length = std::sqrt(dot(*this));
        }
};
```

Structs

- struct vs. class

struct

- In C++ (not C!) **`struct`** are just classes with default public visibility.

```
struct Complex
{

    double re;
    double im;
};
```

```
class Complex
{
 public:
    double re;
    double im;
};
```

- Can also contain everything normal classes have methods, operators, typedefs, access specifiers etc.

Summary

- OOP Principles
  - Abstraction
  - Encapsulation
  - Modularity

- Classes
  - `.hpp, .cpp`
  - attributes and methods
  - Const methods
  - constructor, destructor
  - assignment operators
  - conversion

- Outlook
  - Run-time polymorphism