EBERHARD KARLS UNIVERSITÄT TÜBINGEN

FACULTY OF SCIENCE

Applied Bioinformatics

Programming in C/C++

- Inheritance and Polymorphism -

Inheritance

Class relations

OOM - Object Oriented Modeling

Gives us a “language” to model and communicate e.g., class relations using UML

Important class relations:

Association

- The interaction and communication between classes

Aggregation (or composition)

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

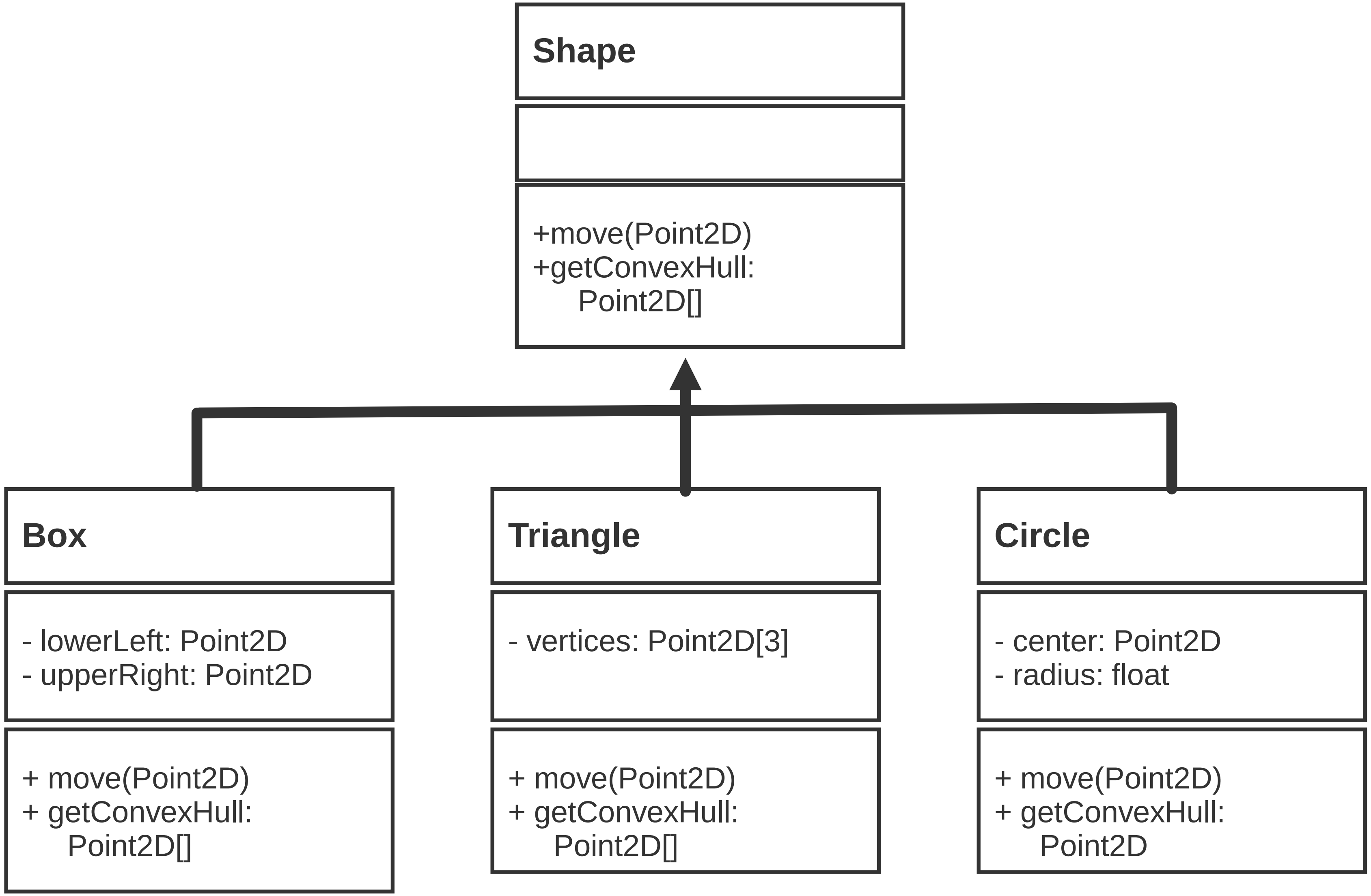
Generalization and inheritance

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

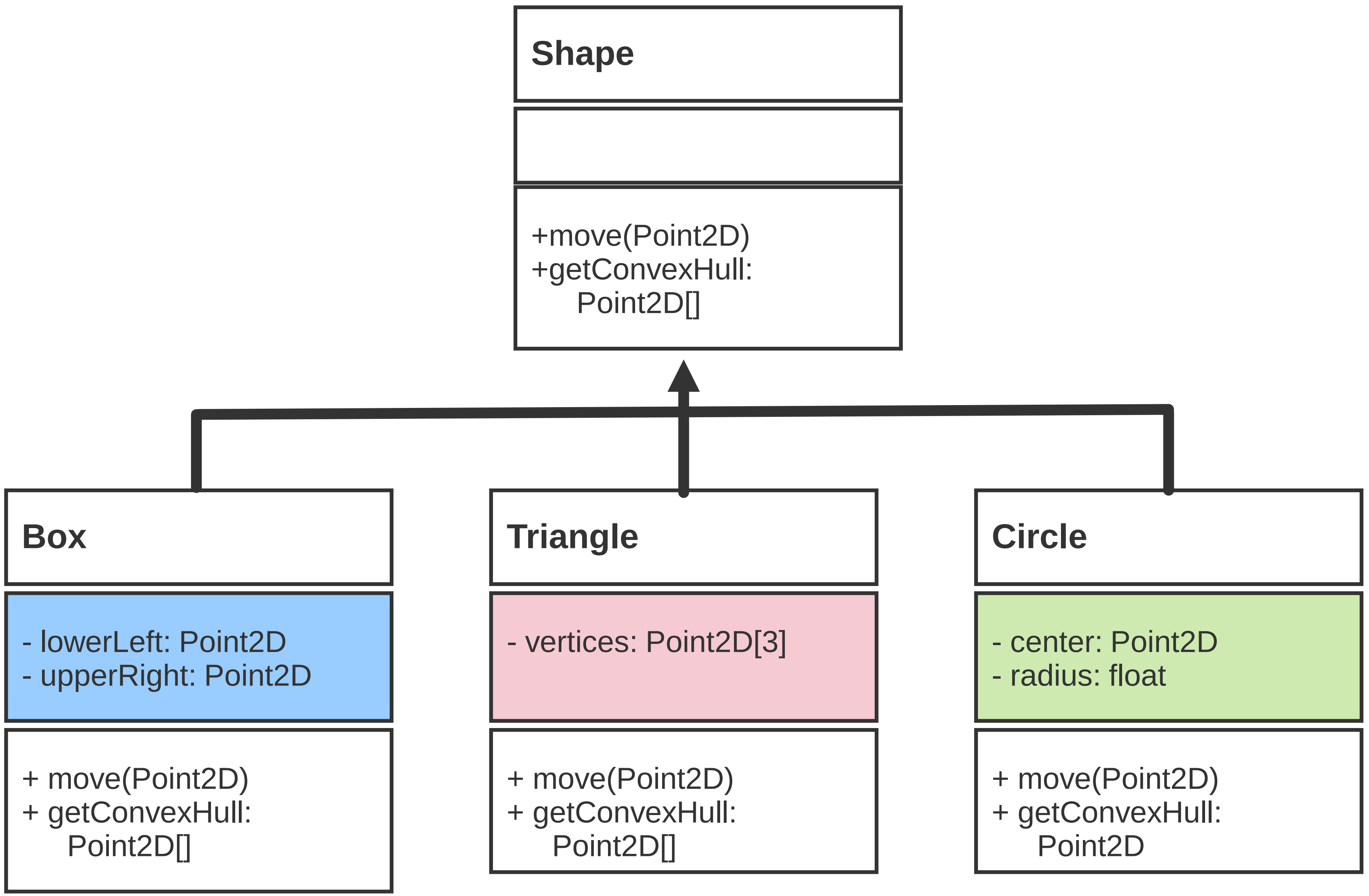
Example

Shape
+move(Point2D) +getConvexHull: Point2D[]

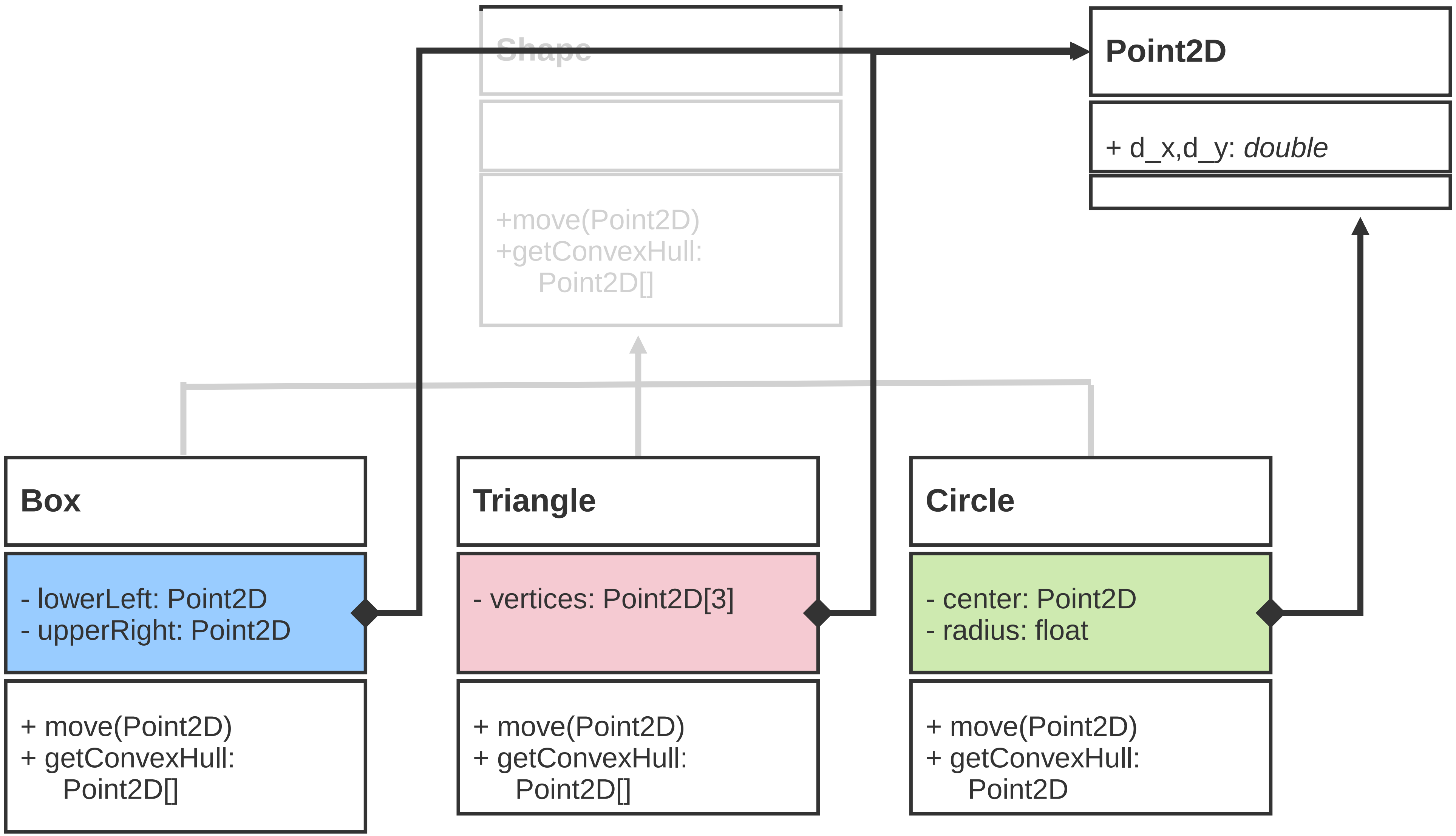
Example


Shape
+move(Point2D)
+getConvexHull:
Point2D[]
Box
- lowerLeft: Point2D
- upperRight: Point2D
+ move(Point2D)
+ getConvexHull:
Point2D[]
Triangle
- vertices: Point2D[3]
+ move(Point2D)
+ getConvexHull:
Point2D[]
Circle
- center: Point2D
- radius: float
+ move(Point2D)
+ getConvexHull:
Point2D

Example

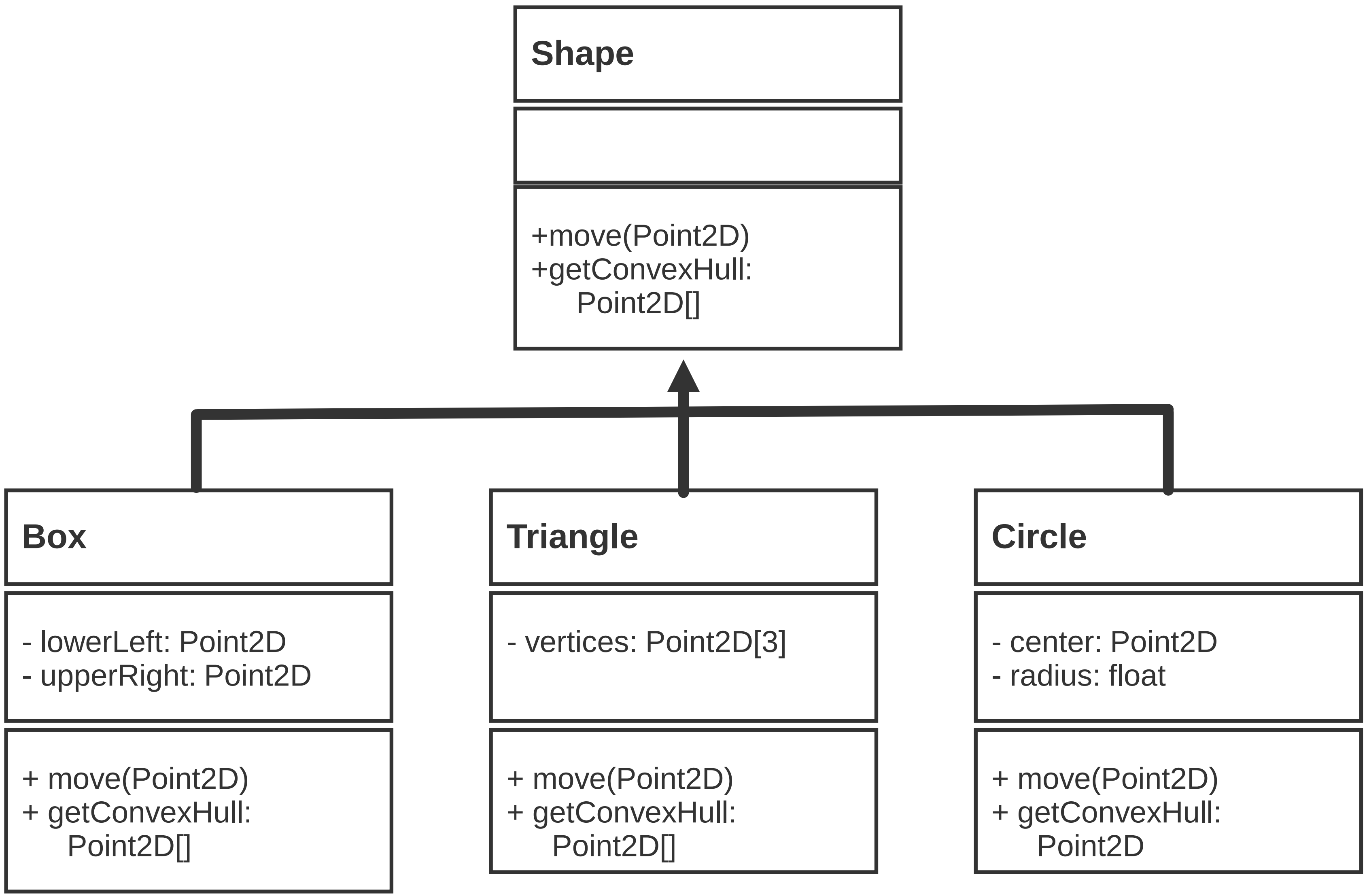

Shape
+move(Point2D)
+getConvexHull:
Point2D[]
Box
- lowerLeft: Point2D
- upperRight: Point2D
+ move(Point2D)
+ getConvexHull:
Point2D[]
Triangle
- vertices: Point2D[3]
+ move(Point2D)
+ getConvexHull:
Point2D[]
Circle
- center: Point2D
- radius: float
+ move(Point2D)
+ getConvexHull:
Point2D

Example

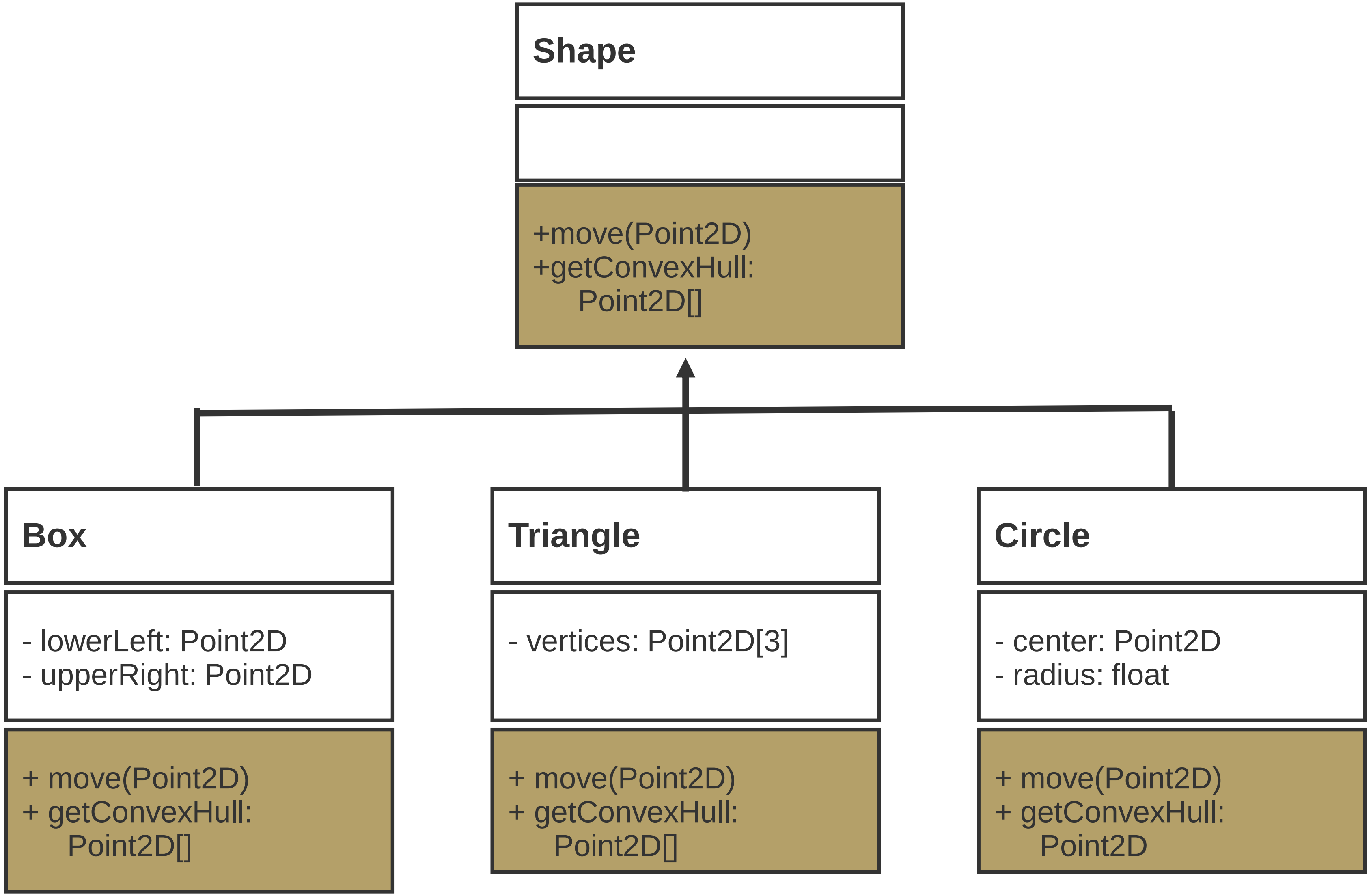


Shape
+move(Point2D)
+getConvexHull:
Point2D[]
Point2D
+ d_x,d_y: double
Box
- lowerLeft: Point2D
- upperRight: Point2D
+ move(Point2D)
+ getConvexHull:
Point2D[]
Triangle
- vertices: Point2D[3]
+ move(Point2D)
+ getConvexHull:
Point2D[]
Circle
- center: Point2D
- radius: float
+ move(Point2D)
+ getConvexHull:
Point2D

Example


Shape
+move(Point2D)
+getConvexHull:
Point2D[]
Box
- lowerLeft: Point2D
- upperRight: Point2D
+ move(Point2D)
+ getConvexHull:
Point2D[]
Triangle
- vertices: Point2D[3]
+ move(Point2D)
+ getConvexHull:
Point2D[]
Circle
- center: Point2D
- radius: float
+ move(Point2D)
+ getConvexHull:
Point2D

Example

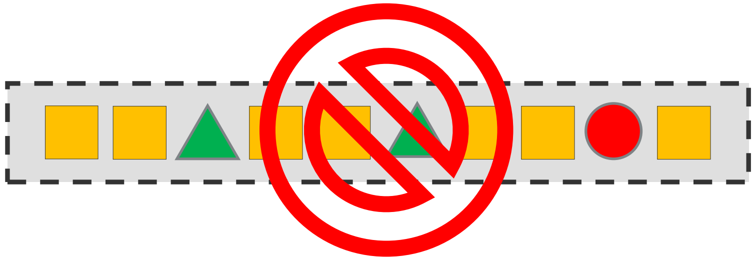

Shape
+move(Point2D)
+getConvexHull:
Point2D[]
Box
- lowerLeft: Point2D
- upperRight: Point2D
+ move(Point2D)
+ getConvexHull:
Point2D[]
Triangle
- vertices: Point2D[3]
+ move(Point2D)
+ getConvexHull:
Point2D[]
Circle
- center: Point2D
- radius: float
+ move(Point2D)
+ getConvexHull:
Point2D

Polymorphism through Inheritance

- Inheritance allows us to use the base class instead of treating each derived classes explicitly.

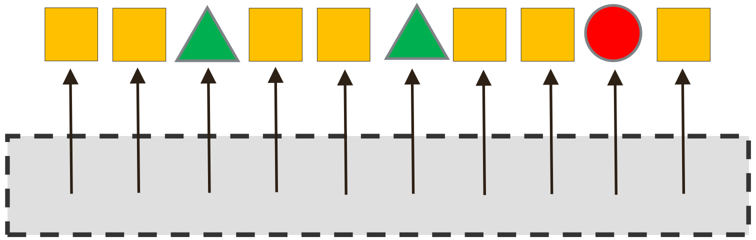
- Motivation:
  - Store many shapes in single data structure (array, `std::vector`, …)
  - `move()` all shapes vs. move triangles, boxes, circles separately
  - `getConvexHull()` of all shapes vs. querying triangles, boxes, circles individually

- Storing different types in `std::vector` not allowed.



- First part of solution:

Storing `Shape*` in `vector` is allowed (`Shape` pointer have same type and size).



```cpp
class Shape {
public:
    int x;
};

class DerivedShape : public Shape  {
public:
    int y;
};

int main() {
    Shape  objA;
    DerivedShape  objDA;
    objDA.x = 1;
    objDA.y = 2;
    objA = objDA;  // ok. as objDA is of type A - copies x only
    objDA = objA;  // wrong
    Shape  *ptrA = &objDA;
    *ptrA = objA; // ok. as *ptrA points to an object of type Shape-
>changes x
}
```

Access to attribute of the base class

Polymorphism: Use like object of the base class

`virtual` – Function Call of the Derived Classes

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Methods may be implemented differently in derived classes.

Problem: Which method is called, that of the base or that of the derived class?

- The `virtual` keyword in the base class indicates that the method will be overridden if a method with same signature exists in a derived class.

- The `overwrite` keyword after the method signature in the derived class is optional (`C++11`) and indicates that we intend to overwrite a method from the base class.

- Second part of solution: When **pointers or references** to an object of a base class are used **dynamic binding** occurs. If a method is marked virtual, the overridden version in the derived class will be called.

Note: the exact **type of the object is not** known **until runtime**. However, the base object is guaranteed to be **part of** the present object .

Example

```cpp
class Shape {
public:
  int x;
  void print() { cout << "base class: " << x << endl; }
};

class DerivedShape : public Shape {
public:
  int y;
  void print()
    { cout << "class DerivedShape: " << x << " " << y << endl; }
};

int main() {
  Shape objA;
  DerivedShape objDA;  objDA.x = 1;  objDA.y = 2;
  Shape *ptrA = &objDA;
  ptrA->print();
}
```

Output:

```
base class: 1
```

Example

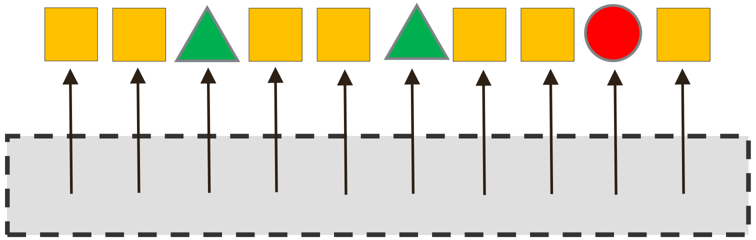
```cpp
class Shape {
public:
  int x;
  virtual void print() { cout << "base class: " << x << endl; }
};

class DerivedShape : public Shape {
public:
  int y;
  void print() override
    { cout << "class DerivedShape: " << x << " " << y << endl; }
};

int main() {
  Shape objA;
  DerivedShape objDA;  objDA.x = 1;  objDA.y = 2;
  Shape *ptrA = &objDA;
  ptrA->print();
}
```

Output:

```
class DerivedShape : 1 2
```

- Close to our goal:
  - Store derived types
  - Execute function that is specific to the implementation in the subclass

```
vector<Shape*> v;
v.push_back(new Box());
v.push_back(new Box());
v.push_back(new Triangle());
// …
for (auto e : v){ e->move({3.0,5.0}); }
// …
```



- **Missing**:
  - Construction
  - Destruction

Constructors in Derived Classes

Recall: A constructor of the derived class always **calls the constructor of the base class first**

implicit

```
class A {...};

class DerivedA : public A {
public:
 DerivedA()
 {
   cout << "constructor DerivedA"
        << endl;
 }
};
```

explicit

```
class A {...};

class DerivedA : public A {
public:
 DerivedA() : A()
 {
   cout << "constructor DerivedA"
        << endl;
 }
};
```

```cpp
class A {
public:
  int x;
  A() { cout << "constructor A" << endl; }
  virtual void print() { cout << "class A: " << x << endl; }
};

class DerivedA : public A {
public:
  int y;
  DerivedA() { cout << "constructor DerivedA" << endl; }
  void print() override { cout << "class DerivedA: " << x << " " << y << endl; }
};

int main() {
  A objA;
  DerivedA objDA;
}
```

Output:

```
constructor A
constructor A
constructor DerivedA
```

Recall: Destructor

- Cleaning up at the end of the life of an object
- No parameters
- Name: `~T()`
- Most frequent task: Release of resources possibly requested in the constructor.

```
class MyClass {
    ...
public:
    ~MyClass();   // destructor (no parameters)
    ...
};
```

Destructors of Derived Classes

- The **destructor of the base class is always called** after the destructor of the derived class is finished.
  - Overwriting the destructor does not change the execution of the base class destructor
  - Unlike copy constructors and assignment operators.
- Important:
  - **The destructor of the base class should always be declared `virtual`.**
  - Correct handling for `delete` on pointers of the base class.
  - Danger of memory leaks!

```
class A {
public:
    int x;
    ~A() { cout << "destruct A" << endl; };
};

class DerivedA  : public A {
public:
    int y;
    ~DerivedA() { cout << "destruct Derived A" << endl; };
};

int main() {
    A* ptrA = new A;
    A* ptrDA = new DerivedA;

    delete ptrA;
    delete ptrDA;
}
```

Output:

```
destruct A
destruct A
```

```cpp
class A {
public:
    int x;
    virtual ~A() { cout << "destruct A" << endl; };
};

class DerivedA  : public A {
public:
    int y;
    ~DerivedA() { cout << "destruct Derived A" << endl; };
};

int main() {
    A* ptrA = new A;
    A* ptrDA = new DerivedA;
    delete ptrA;
    delete ptrDA;
}
```

Output:

```
destruct A
destruct Derived A
destruct A
```

Arrays of Objects

- When creating arrays, the **default constructor** is called for each object
- When destroying the array (also at the end of a block) the **destructor** is called
- Applies to static and dynamic allocation (**`new`** and **`delete`**)

```
class A {
public:
  A() { cout << "Constructor A" << endl; }
  ~A() { cout << "Destructor A" << endl; }
};

class B {
public:
  B() { cout << "Constructor B" << endl; }
  ~B() { cout << "Destructor B" << endl; }
};

int main(int argc, char **argv) {
  A arrayA[3];
  B *arrayB;
  arrayB = new B[4];
  delete[] arrayB; // explicit destruction of arrayB elements
} // implicit destruction of arrayA elements
```

```
Constructor A
Constructor A
Constructor A
Constructor B
Constructor B
Constructor B
Constructor B
Destructor B
Destructor B
Destructor B
Destructor B
Destructor A
Destructor A
Destructor A
```

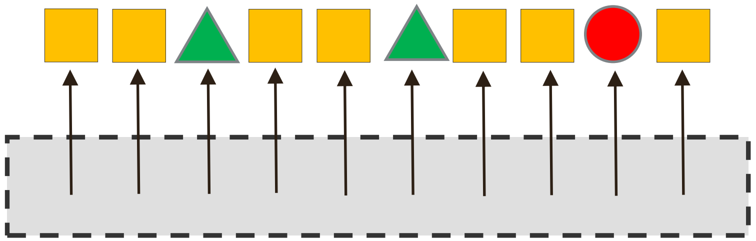
Summary

- We can store pointers to a base class `Shape*` in a container (e.g., `arrays` or `std::vector<Shape*>, ...`)

- Using **dynamic binding** to **virtual member functions** we achieve polymorphism at run-time.

- Calling a function on each `Shape*` in the container executes the corresponding implementation of Triangle, Box, Circle.

```
vector<Shape*> v;
v.push_back(new Box());
v.push_back(new Triangle());
// ...
for (auto e : v) { e->move();}
// ...
for (auto e : v) { delete e; }
```



- We are storing pointers -> deleting elements calls destructor and frees memory.

Detail: Abstract Base Class

- **Common use:**
  - Base class declares **common interface and data** for the derived classes
  - Declaration as **pure virtual function**: There is no definition!

```
class A {
public:
 int x;
 virtual void print() = 0; // pure virtual
};
```

- Definition of class A never complete!
- No objects **`class A`** possible, but pointers **`class A*`**
- `print()` **must** be defined in derived (non-abstract) classes.

Note: No attributes at all? We get an interface class:

```
class InterfacePrintable {
public:
  virtual void print() = 0;
};
```

Detail: Internals of Dynamic Dispatch

```
class Shape {
public:
  void print() { cout << ... }
};

class DerivedShape : public Shape  {
public:
  void draw() { ... }
};
```

Without virtual functions:

- Method call is like a function call
- Code located somewhere in memory gets called
- Access to object data through hidden parameter *this*

```
Shape *base = new Shape();
base->print();
```

```
Shape *derived = new DerivedShape();
derived->print()
```

code: memory

functions
```
print() : 0x1710
draw()  : 0x2814
...
```

Detail: Internals of Dynamic Dispatch

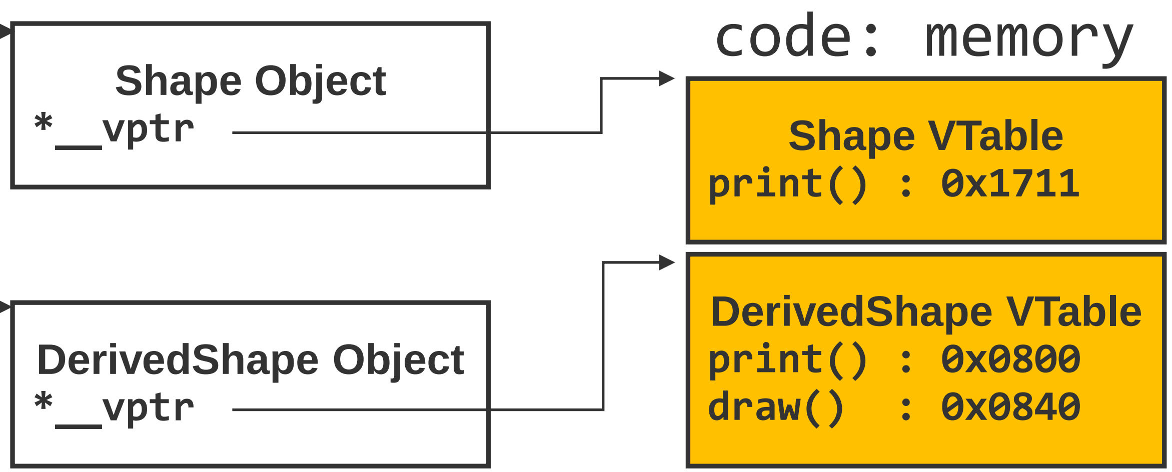
```cpp
class Shape {
public:
  virtual void print() { cout << ... }
};

class DerivedShape : public Shape  {
public:
  void print() override { cout << ... }
  virtual void draw() { ... }
};
```

With virtual functions:

- Compiler creates **Vtable** (**per class**) and **VPTR** (**per object**) to track which virtual function to call.
- Memory and performance overhead!
- Exact details: implementation specific

```cpp
Shape *base = new Shape();
base->print();
```

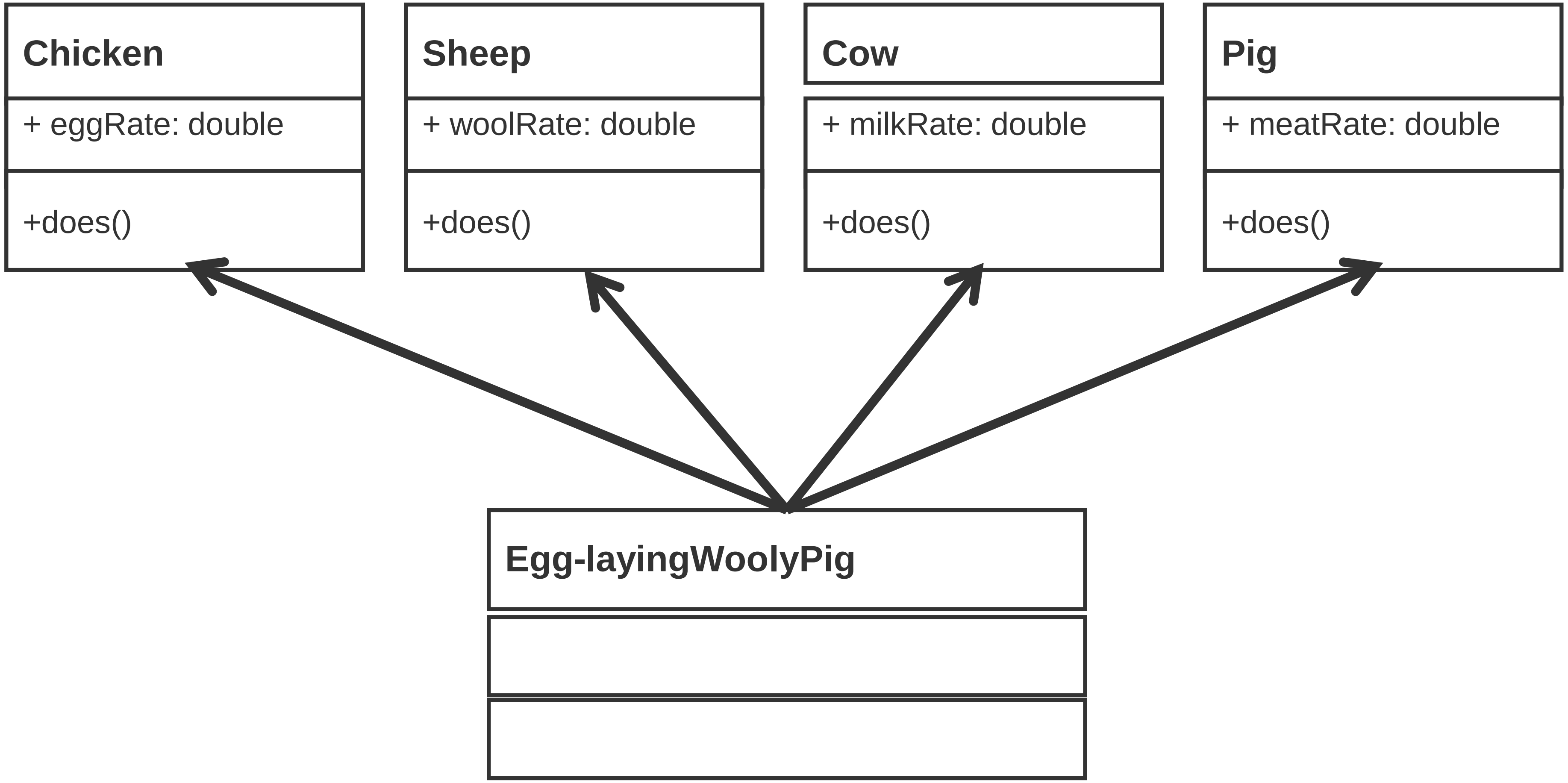
```cpp
Shape *derived = new DerivedShape();
derived->print()
```



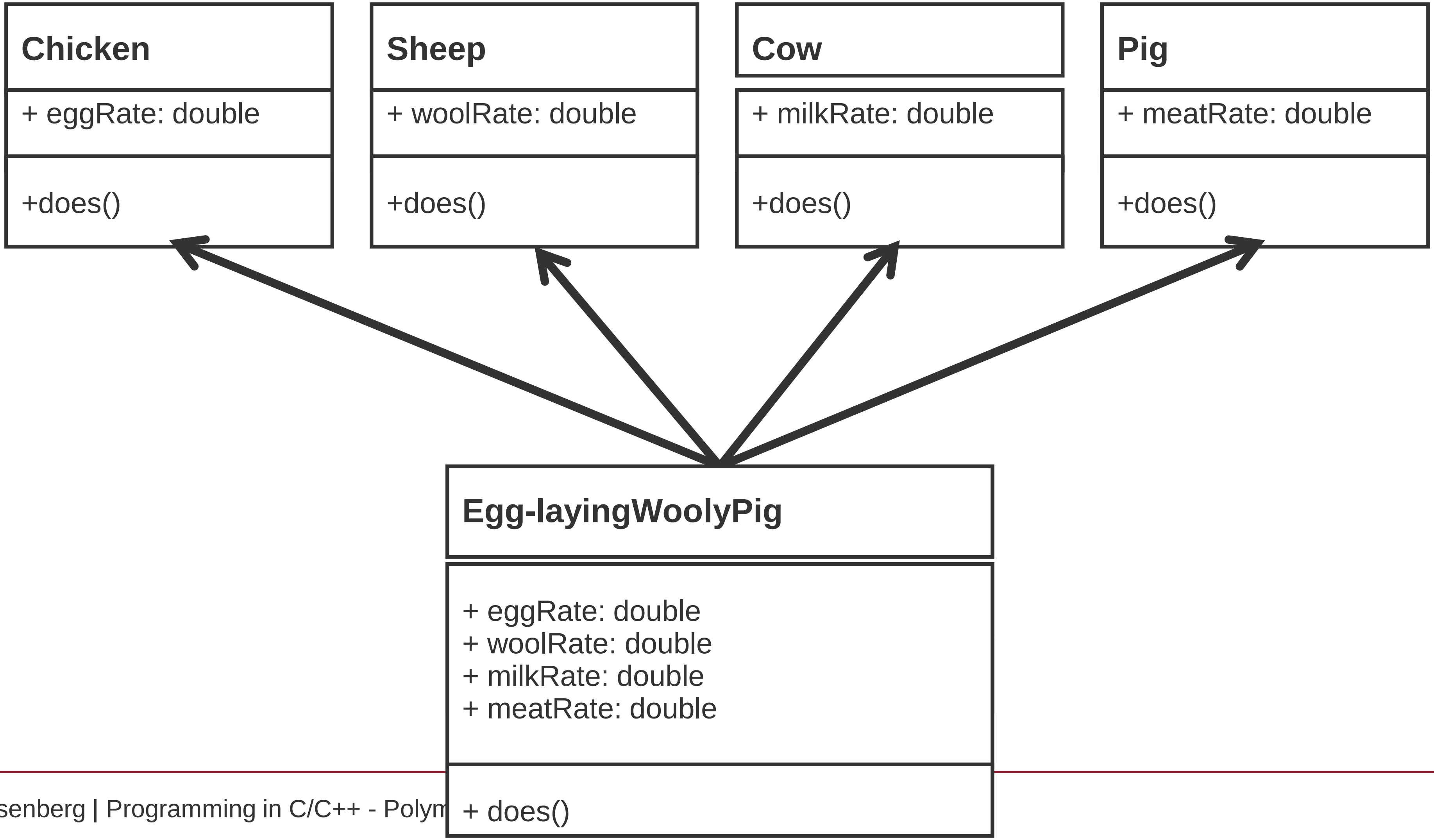
Detail: Multiple Inheritance

- C++ supports multiple inheritance
- Dangerous: resort to it if absolute must

Multiple Inheritance



Multiple Inheritance



Multiple Inheritance

Most common use:

- Derived from several **interface classes:** IPrintable, IDrawable, …

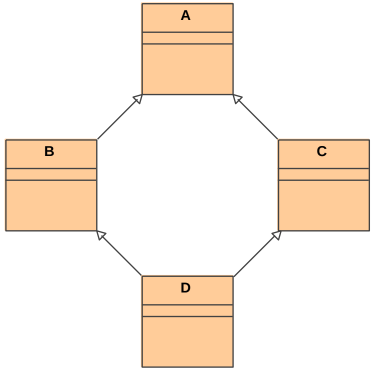
Everything else: **likely evil ;)**

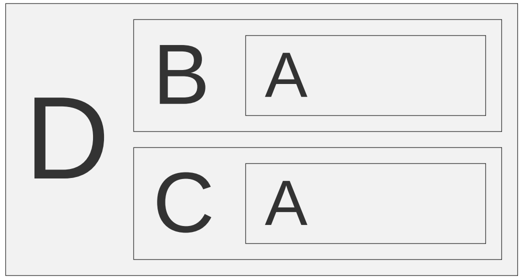
- Derived class inherits from several parent classes
  - Unification of all attributes
  - Unification of all methods

Multiple Inheritance

- What could possibly go wrong?
  - Attributes with the same name
  - Methods with the same signature

- "Deadly diamond of derivation"





- **Recommendation**: Do not use multiple inheritance if possible

```cpp
class A {
public:
  int x = 0;
  virtual void print() const {
    cout << "class A: " << x << endl; }
};

class B {
public:
  int y = 1;
  virtual void print() const {
    cout << "class B: " << y << endl; }
};

class Derived : public A, public B {
public:
  virtual void print() const {
    cout << "class Derived: " << x << endl; }
};
...
  Derived obj;
  obj.print();    // Derived::print()
  obj.A::print(); // print() of Class A
  obj.B::print(); // print() of Class B
```

final

- Multiple inheritance often leads to complex code and errors.
- The same is true for deep inheritance hierarchies and unnecessary application of inheritance.

Recommendation: Prefer composition over inheritance. If you need inheritance keep shallow inheritance hierarchies.

Similar to Java, C++ has the `final` keyword to **protect a class against derivation**:

```
struct B final : public Base    // struct B is final
{
 // …
};

struct C : B                    // error: B is final
{
};
```

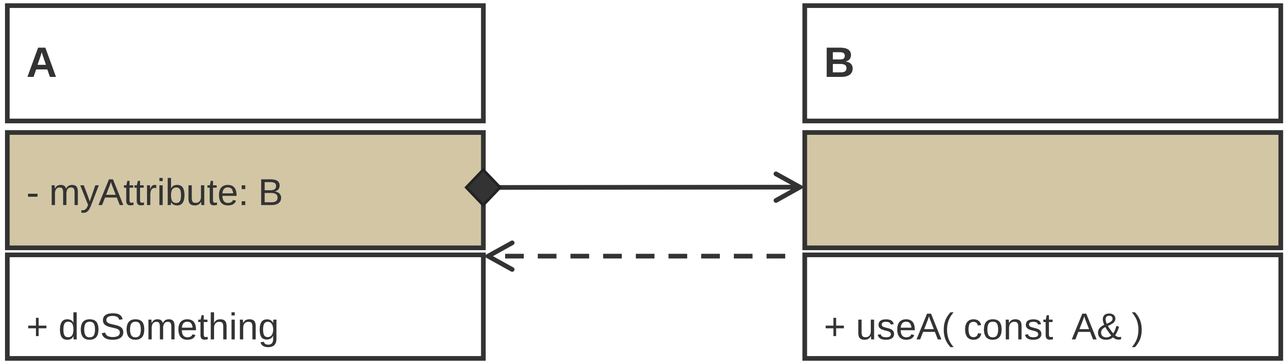
- Detail: `final` can also be put at method declaration to protect it against override.

`forward` declarations

- Help to resolve circular includes/dependencies
- Can speed up compilation by reducing includes

Forward Declarations

- Each function and type must be declared before it can be used
- Sometimes the full declaration is not possible beforehand
- Example:



```cpp
class A; // forward declaration

class B {
public:
  int x;
  B(int _n) : x(_n) {}
  void useA(const A &_a) const; // ok to use A as reference in declaration
};

class A {
  B myAttribute; // B is already fully declared, save to use its structure
public:
  A(int x) : myAttribute(x) {}
  void doSomething() const { cout << myAttribute.x << endl; }
}; // full declaration and definition of A

// now we can access A's structure
void B::useA(const A &_a) const {
  cout << "using A: ";
  _a.doSomething();
}
int main() {
  A objA(3);
  B objB(4);
  objB.useA(objA);
}
```

Forward Declaration and Header Files

A.hpp

```
#ifndef A_HPP
#define A_HPP

#include "B.hpp"

class A {
protected:
     B myAttribute;

public:
     A(int x);
     void doSomething() const;
};

#endif
```

B.hpp

```
#ifndef B_HPP
#define B_HPP

class A;

class B {
public:
  int x;

  B(int _n);

  void useA(const A&_a) const;
};

#endif
```

Forward Declaration and corresponding declaration

A.cpp

```
#include <iostream>
#include "A.hpp"

A::A(int x) : myAttribute(x) {
}

void A::doSomething() const {
  std::cout << myAttribute.x << std::endl;
}
```

B.cpp

```
#include "B.hpp"
#include "A.hpp"

B::B(int _n) : x(_n) {}

void
B::useA(const A& _a) const {
  std::cout << "using A: ";
  _a.doSomething();
}
```

const-ness

- const pointers and pointers to const
- const methods

Const and Pointer

- Pointer to constants: changing the **values** is not possible
- Constant pointer: changing the **address** is not possible

```
const int *ptrToConst; // (const int)*, not const (int*)
int *const constPtr;

const int *const constPtrToConst;
```

[Wikipedia]

Example `const` Reference: Copy Constructor

- **Copy constructor** (argument: other object of the class)
  - to create a copy

```
Point2D::Point2D( const Point2D& _other ) {
    d_x = _other.d_x; d_y = _other.d_y;
}

...

Point2D a;
a.d_x = 1; a.d_y = 2;

Point2D b(a); // explicit call to copy constructor
```

Temporary Objects and `const` References

- A function with a non-const reference can theoretically change the contents of the reference
- Temporary objects exist only on the stack during parameter passing, i.e. they must not be modified

```
// Improperly declared function: parameter should be const reference
void print_me_bad(std::string &s) { std::cout << s << std::endl; }

// Properly declared function: function has no intent to modify s:
void print_me_good(const std::string &s) { std::cout << s << std::endl; }
...
std::string hello("Hello");
print_me_bad(hello);                 // Compiles ok; hello is not temporary
print_me_bad(std::string("World")); // Compile error; temporary object
print_me_bad("!"); // Compile error; compiler wants to construct a temporary
                   // std::string from const char*
print_me_good(hello);                 // Compiles ok
print_me_good(std::string("World")); // Compiles ok
print_me_good("!");                   // Compiles ok
```

- What about `*this`? Is there a way to express that the object is not changed?

`Const` Methods

- Methods can also be declared `const`
- Const methods do not change attributes

```
class A {
public:
  int x;
  void print() const {
    // x can not be changed in this function
    cout << "class A: " << x << endl;
  }
};
```

- Within `const` methods only `const` methods may be called
- Only `const` methods may be called at a `const` **reference**
- Verification during compilation

Recommendation: declare methods as `const` if they don't change `*this`

`Const`-Correctness

- Can promote efficiency
- For your own programming style
- Makes code much easier to read and reason about

Casts

Type Convertierung – Cast

- A pointer can be converted to a pointer of any other type
- The access to the content is not always useful
- Explicit Cast
- Dynamic Cast
- Static Cast
- Reinterpret Cast
- Const Cast

Explicit Cast – C-style


```
(new_type) variableOfOldType
```

- Conversion to another type (if conversion method exists)

```
int i =(int)doubleValue;
```

- Conversion of pointers to any other pointer **is always possible.**
- Dereferencing and accessing the content may lead to unexpected results or run-time errors.

```
class A; // with method doSomething()
class B;

A* ptrA = new A;
B* ptrB = (B*)ptrA;  // dangerous

ptrB->doSomething(); // dangerous

delete ptrA;
```

dynamic_cast

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Only possible on pointers and references to classes

```
dynamic_cast<new_class*>( pointerOfOldClass )
```

- Ensures that after the cast a complete object can be accessed
- Works e.g., as a **cast to a base class**
- Evaluated at runtime
- Returns `nullptr` if cast is not possible (or exception for invalid pointers, references)

```
class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;

    pd = dynamic_cast<CDerived*>(pba);
    if (pd == nullptr) cout << "Null pointer on first type-cast" << endl;

    pd = dynamic_cast<CDerived*>(pbb);
    if (pd == nullptr) cout << "Null pointer on second type-cast" << endl;
}
```

Ouput:

```
Null pointer on second type-cast
```

static_cast

- Possible on all types

```
static_cast<new_type>( variableOfOldType )
```

- Cast between any classes and types, only if type conversion is allowed
- **Test at compile time**, no tests at runtime

```
char c = 4;
int  i = 15;

i = static_cast<int>(c));
cout << "static: " << i << endl;

...

i = *( static_cast<int*>(&c));
cout << "static: " << i << endl;
```

Output:

```
static: 4
```

Compile Error:

```
error: invalid static_cast from type
'char*' to type 'int*'
```

reinterpret_cast

- Allows casting between arbitrary pointer types

```
reinterpret_cast<new_type*>( pointerOfOldType )
```

- On dereferencing, the bit pattern is simply interpreted as a new type

- Result is machine dependent, potentially dangerous

```
char c = 4;
int  i = 15;

i = *( reinterpret_cast<int*>(&c));
cout << "reinterpret: " << i << endl;
```

Output:

```
reinterpret: 3844
```

const_cast

- Converts const to non-const and vice versa, potentially dangerous

```
const_cast<old_type*>( pointerOfOldType )
```

- e.g. if a non-const is absolutely needed for a function call.

```
void print (char * str)
{
  cout << str << endl;
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

[cplusplus.com – Tutorial]

Querying the Current Type

```
typeid( expression )
```

- Returns a structure that provides information about the type, including the type name

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

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

Output:

```
a and b are of
different types:
a is: int *
b is: int
```

[cplusplus.com – Tutorial]

Lambda Functions

Reoccurring Similar Tasks

```
template <typename TElem>
void square_all_elements(std::vector<TElem> &vec) {
  for (TElem &elem : vec)
    elem = elem * elem;
}
```

```
template <typename TElem>
void squareroot_all_elements(std::vector<TElem> &vec) {
  for (TElem &elem : vec)
    elem = std::sqrt(elem); // here: sqrt
}
```

- Two implementations seem tedious
- What if we want to create a function "F_on_all_elements" and define F separately?

Lambda Functions

```
template <typename TElem, typename TLambda>
void on_all_elements(std::vector<TElem> &vec, TLambda const & mylambda) {
  for (TElem &elem : vec)
    elem = mylambda(elem); // apply the lambda function
}
```

```
int main() {
  auto square = [](auto const &elem) { return elem * elem; };
  auto square_root = [](auto const &elem) { return std::sqrt(elem); };

  std::vector<double> ds{0.2, 1.5, 2};

  on_all_elements(ds, square); // ds == { 0.04, 2.25, 4 }
  on_all_elements(ds, square_root); // ds == { 0.20, 1.50, 2 }
}
```

Lambda Functions

```
auto square = [](auto const &elem) { return elem * elem; };
```

- Lambdas are *objects* and each lambda has a distinct type
- Can be saved in variable with deduced type (`auto`)
- Functions can take them as template parameters.

- A minimal Lambda that does nothing is `[](){};`

- `[]` introduces a Lambda definition
- `()` contains the parameters, just like with ordinary functions except that auto is valid
- `{}` contains the body of the lambda function

- The return type of the Lambda is deduced by default

Lambda Functions – Capture

```
...
std::vector<double> ds{0.2, 1.5, 2};
double off = 10.0;     // variable in local scope
auto add = [&off](auto const &elem) { return elem + off; };
           // ↑ grant access to local variable as reference
on_all_elements(ds, add);  // ds == { 10.20, 11.50, 12 }
// off == 10
```

- `[<capture>]` introduces a Lambda definition with access to the local context

- [var1, var2] – access to comma separated list of variables (*by-copy*)
- [&var1, &var2] – access to comma separated list of variables (*by-reference*)
- `[=]` – access to **all variables** in the current scope (*by-copy*)
- `[&]` – access to **all variables** in the current scope (*by-reference*)
- `[this]` – access to **all members**

Note: The STL has many functions that accept lambdas (sorting, accumulation, …)

Polymorphism - Summary

	Dynamic	static
how	**Inheritance + virtual functions**	**overloaded functions + templates**
model	object-oriented	generic programming
dispatch at	run-time	compile-time
run-time	slower	faster
good for	complex and deep type hierarchies; frameworks; large “in-house” solutions	Flat hierarchies; “external types”; generic libraries; high-performance
difficulty	easier to program	more difficult to program

Summary

- Classes and Inheritance
- Multiple Inheritance
- `final`
- `const`-Correctness
- Casts
- Lambda Functions