uOttawa

L’Université canadienne
Canada’s university

Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

Université d’Ottawa | University of Ottawa

uOttawa.ca

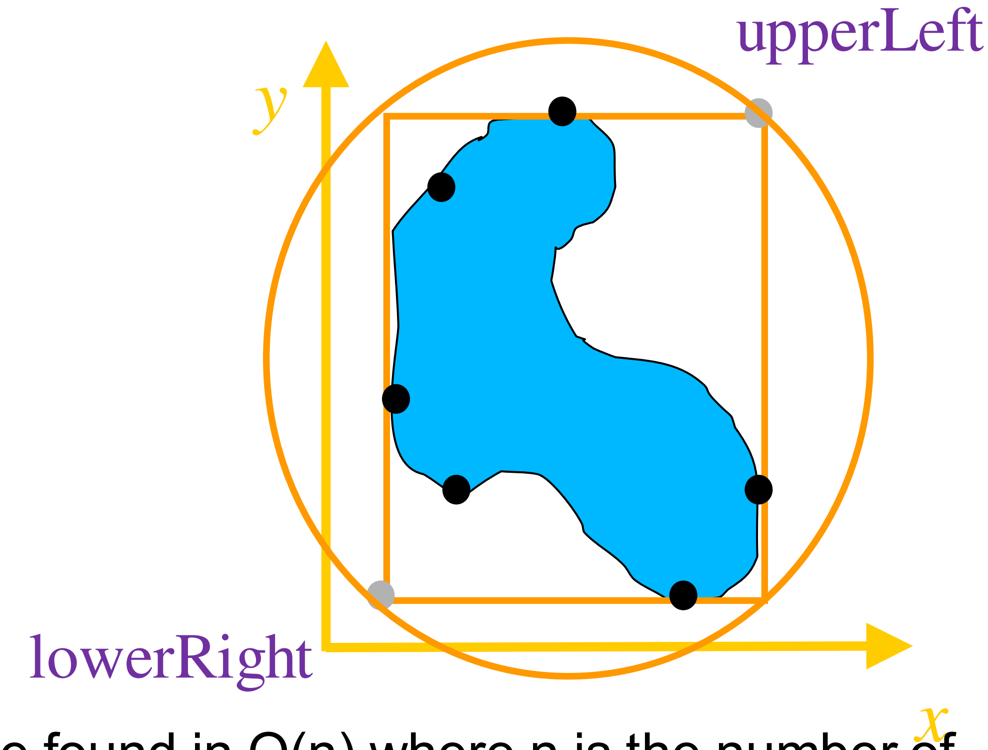
This Lectures

OO

- **Object-oriented design**
  - Class relationships: aggregation, generalization and inheritance, Ch. 15.1, 15.2, 15.5
  - Pointer attributes and this pointer, 13.5
  - Copy construction and assignment, Ch. 13.1

Reminder Example Problem: Bounding a Shape

- **Find bounding primitive that encloses the blue shape**
  - Smallest AABox
  - Circle*



*Note: Smallest circle can be found in O(n) where n is the number of boundary points

Class Example: Point2D and Axis-Aligned Bounding Box (AABox)

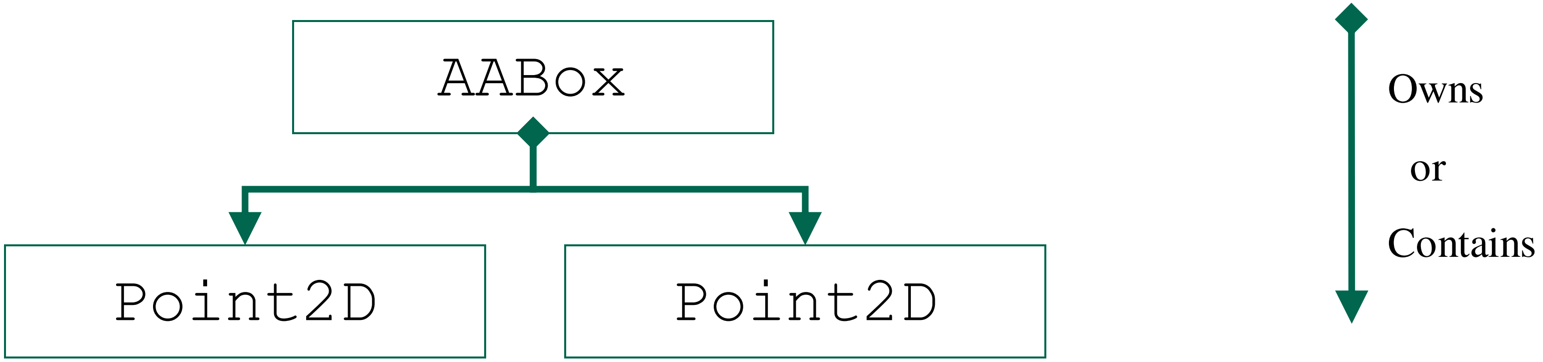
- **Define a `AABox` based on two `Point2D`**

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

class AABox {
    Point2D d_lowerLeft;
    Point2D d_upperRight;
public:
    AABox(const Point2D& _lowerLeft,
          const Point2D& _upperRight );
};
```

Class Relationships – Aggregation

- **The “has a” relationship**
  - Containment relation, e.g., AABox contains two Point2D



```
class AABox {
  Point2D d_lowerLeft;
  Point2D d_upperRight;
public:
  AABox( const Point2D& _lowerLeft,
         const Point2D& _upperRight ); …  };
```

Example

- Make sure all necessary constructors exist
- Make use of initializer lists

```
class AABox {
  AABox( const Point2D& _lowerLeft,
         const Point2D& _upperRight ) :
    d_lowerLeft(_lowerLeft), d_upperRight(_upperRight)
{} …
  bool inside( const Point2D& _pt ) const;
};
```

- What if an object is to be default initialized but has no default argument constructor?

Example Continued

- Assume AABox has no default constructor and no reasonable dummy argument

```
class Triangle {
  Point2D d_vA, d_vB, d_vC;
  AABox d_bbox;
public:
  Triangle( const Point2D& _vA, const Point2D& _vB,
            const Point2D& _vC );
};

Triangle::Triangle( const Point2D& _vA, const Point2D& _vB,
                    const Point2D& _vC )
: d_vA( _vA ), d_vB( _vB ), d_vC( _vC ), ~~d_bbox( ? )~~ {
}
```

Aside: Syntax Pointer to Object

- **Special syntax for accessing attributes and methods through pointer to objects**

```
class Triangle { …
public:
  AABox* d_bbox;
  Triangle( const Point2D& _vA, const Point2D& _vB,
            const Point2D& _vC )
  : d_vA( _vA ), d_vB( _vB ), d_vC( _vC ), d_bbox(0) {}
};

Point2D p2D;
~~d_bbox.inside( p2D );~~
(*d_bbox).inside( p2D );
d_bbox->inside( p2D );
```

Aggregation Summary

  - Contained objects must be initialized in a initializer list or must have a default constructor
    - Pointers must be initialized but not the object pointed to
    - C++11 allows the use of in-class initializers which is preferable to initializer lists for each constructor
- **Internal aggregation**
  - Objects constructs (and destructs) the objects which it owns
- **External aggregation**
  - Contained objects are constructed elsewhere and a reference or pointer is passed in

Reminder: Copy Constructor

- **The compiler automatically creates a shallow copy constructor if none is specified in the source**
- **Constructors always create a new object**
  - Copy constructor makes a new copy of an existing object.
  - The copy will have all the same attributes than the original (with the synthesized copy constructor).

```
Point2D ptA;
Point2D ptB = ptA; // Copy initialization
```

- **This is a call to the copy constructor**
    - Calls the copy constructor for ptB with ptA

Copy Constructor vs. Assignment Operator

- **We have seen**

  = can be used to invoke the copy constructor
- **but**

  = is normally the assignment operator, e.g., an overloaded operator for a class type.

```
Point2D ptA, ptB;
ptB = ptA;
```

  - Copies the content of an existing object ptA to another existing object ptB

Destructor

- **Same name than class but starts with ~**
  - Public method
  - No return value
  - No arguments
    - Only one destructor per class
  - Called whenever an object is destroyed
    - Auto variable gets out of scope (including function arguments at the end of a function)
    - Explicit call to delete for dynamically allocated objects
    - Program terminates
  - Destructor should free all resources associated with an object, e.g., dynamic memory, file descriptors etc.

Aggregation with Pointers

- Internal aggregation means an object constructs the object which it owns during a constructor
- It needs to destruct the owned object in destructor

```
class Triangle { …
  AABox* d_bbox;
public:
  …
  ~Triangle( );    // clean up our objects
};

Triangle::~Triangle() {
  delete d_bbox;
}
```

Defining our own Copy Constructor

- **Default copy constructor makes a shallow copy**

```
class Triangle { …
  AABox* d_bbox;
public:
  …
  Triangle( const Triangle& oTri );
};
// shallow copy ctor – same as default
Triangle::Triangle( const Triangle& oTri )
    : d_bbox( oTri.d_bbox ) {}
```

- **Shallow copy with pointer types is nearly always wrong**
  - Change the copy constructor to make a deep copy

Deep Copy

```
// deep copy ctor - internal aggregation
Triangle::Triangle( const Triangle& oTri )
    : d_bbox( 0 ) {
  d_bbox = new AABox( oTri.d_bbox );
}
```

- **Leads to rule of 3:**
  - if a class needs a non-default copy constructor, it also needs a non-default destructor and assignment operator (to be discussed later)
  - Rule of 3 has become rule of 5 in some cases with C++11 for move constructor and move assignment

Class Relationships – Generalization and Inheritance

- **Generalization and Inheritance**
  - The “is a” relationship
  - Inheritance from a general class to a more specific one
- **Same concept than in Java**
  - Child (or derived) class inherits methods and attributes from the parent (or base) class
- **Example:**
  - Class `Vector2D` is an extension of class `Point2D`

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

- **Difference to Java**
  - Multiple base classes (inheritance)
  - Use of access modifiers

Full Syntax

- **Specification of a base class:**

```
base-spec :
: base-list
base-list :
base-specifier
base-list , base-specifier
base-specifier :
complete-class-name
virtual access-specifier$_{opt}$ complete-class-name
access-specifier virtual$_{opt}$ complete-class-name
access-specifier :
private
protected
public
```

Effect of Access Modifiers

- Default access modifier for inheritance of classes is private
- Default access modifier for inheritance of structures is public

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

Inheritance Example
Initializer List Problem

```
class Point2D {
protected:
  double d_x;
  double d_y;
public:
  Point2D(double _x=0.0, double _y=0.0) : d_x(_x), d_y(_y) {}
  Point2D Point2D::min( const Point2D& _oPoint ) const {
      return Point2D((d_x < _oPoint.d_x)?d_x:_oPoint.d_x,
                     (d_y < _oPoint.d_y)?d_y:_oPoint.d_y); }
};
class Vector2D : public Point2D {
  double d_length;
public:
  Vector2D(double _x=0.0, double _y=0.0) : ~~d_x(_x), d_y(_y)~~
    { d_length = std::sqrt(dot(*this));}
  double dot( const Vector2D& _oVect ) const;
};
```

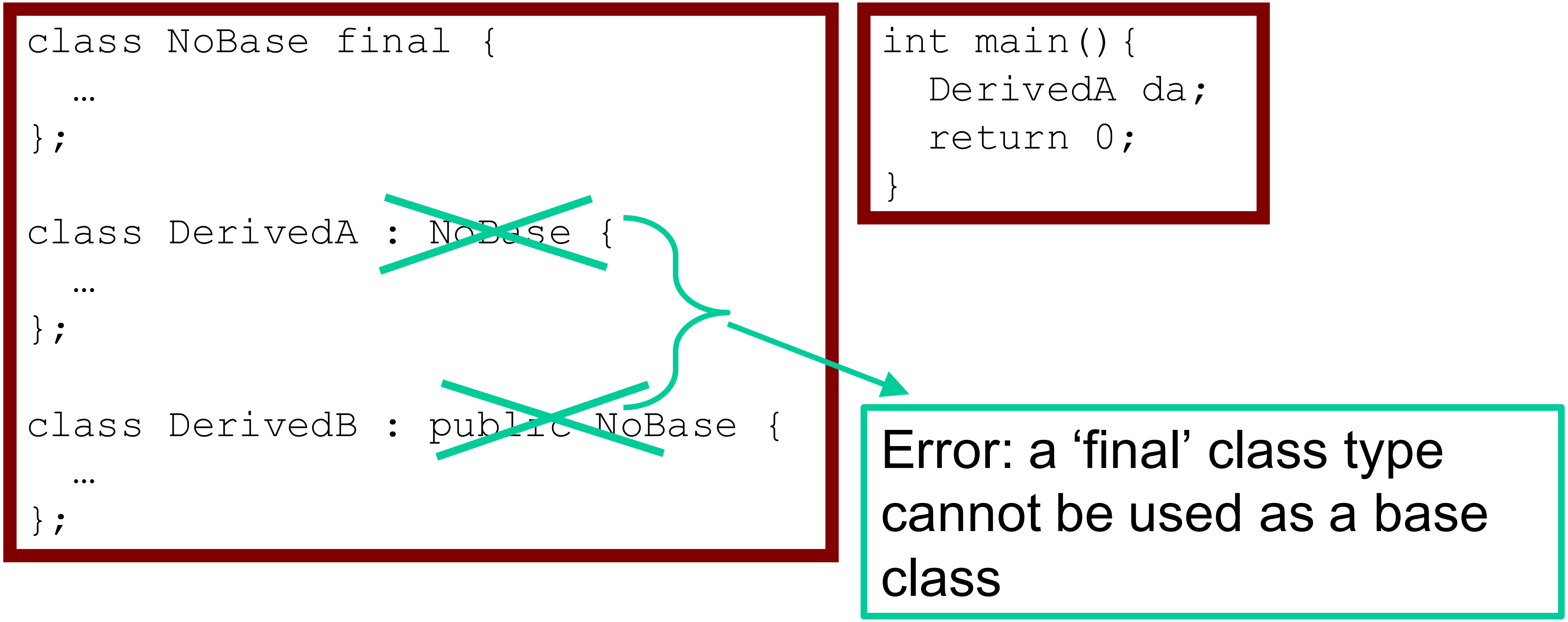
Protected Inheritance Example
Access Problem

```
class Vector2D : protected Point2D {
  double d_length;
public:
  Vector2D(double _x=0.0, double _y=0.0) : d_x(_x), d_y(_y)
    { d_length = std::sqrt(dot(*this));}
  void dot( const Vector2D& _oVec ) const;
};


…
Vector2D v2DA( 3, 2 );
Vector2D v2DB( 1, 1 );
~~v2DB.min( p2D );~~
…
```

Aside: Preventing Class Derivation

- Classes can be declared final in order to prevent the class from being used as a base class



```
class NoBase final {
   …
};

class DerivedA : ~~NoBase~~ {
   …
};

class DerivedB : ~~public~~ NoBase {
   …
};
```

```
int main(){
   DerivedA da;
   return 0;
}
```

Error: a ‘final’ class type cannot be used as a base class

Aside: Preventing Class Derivation

- Sometimes you don’t want to allow derived class to override the base class’ virtual function. C++ 11 allows built-in facility to prevent overriding of virtual function using final specifier.

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

class Base {
public:
     virtual void func() final {
          cout << "The method fun() is from Base class";
        }
};

class Derived : public Base {
public:
     void func() {
          cout << "The method fun() is from Derived class\n";
       }
};
```

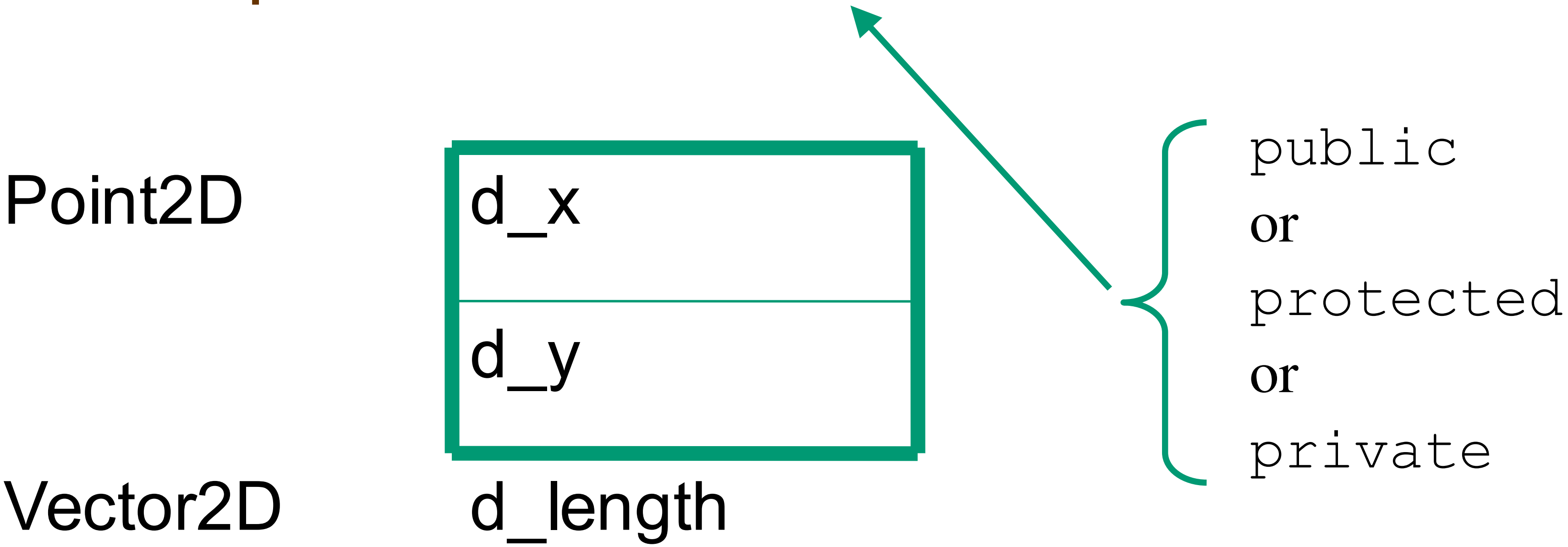
```
int main() {
          Derived d;
          Base &b = d;
          b.func();
          return 0;
}
```

Error: cannot override 'final' function "Base::func“. 'Base::func': function declared as 'final' cannot be overridden by 'Derived::func’.

Layout of a Derived Class

- Object of derived class contains a base class object
- Methods of both classes can be applied (as long as access modifiers are respected)

- **Example:** `class Vector2D : Point2D`



Constructor and Destructor of Derived Class

- **Constructor**
  - Calls the default constructor of the base class
    - Before attributes of the derived class are initialized
  - Use intializer list for use of non-default constructor
    - Base class constructor is always called first independent of order of initializer list
- **Example**

`Point2D()` is called!

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

Constructor and Destructor of Derived Class

- **Constructor**
  - Calls the default constructor of the base class
    - Before attributes of the derived class are initialized
  - Use intializer list for use of non-default constructor
    - Base class constructor is always called first independent of order of initializer list
- **Example**

Can be used instead!

```
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));}
};
```

Copy Constructor

- **Default Copy Constructor**
  - Calls copy constructor of base class first
- **Defined copy constructor**
  - Must explicitly call copy constructor of base class

```
class Vector2D : public Point2D {
int d_length;
public:
  Vector2D(const Vector2D& _oVec ) : Point2D( _oVec ) { … }
};
```

Destructor

- **Base class destructor is always executed after the derived class has been destructed**
  - Overriding the destructor has no effect on the execution of the base class destructor
  - Different then copy constructor and assignment operator
    - Aside: In general can also use default in C++11

```
class Vector2D : public Point2D {
…
public:
  ~Vector2D() {}
    // Point2D part of Vector2D is destructed after Vector2D
 // automatic – no explicit call
};
```

Next Lecture

OO

- **Object-oriented design**
  - Polymorphism: Virtual functions, abstract classes and dynamic cast
  - Exceptions Basics
  - Inline functions, static members