

# Advanced Programming Concepts with C++ CSI2372

Mohammad Alnabhan  
EECS

Université d'Ottawa | University of Ottawa



uOttawa

L'Université canadienne  
Canada's university



[uOttawa.ca](http://uOttawa.ca)

# This lecture

OO

- **Object-oriented design**
  - Polymorphism
    - Virtual Functions, Ch. 15.3, 15.7
    - Abstract classes, Ch. 15.4
    - Dynamic cast, Ch. 19.2.1

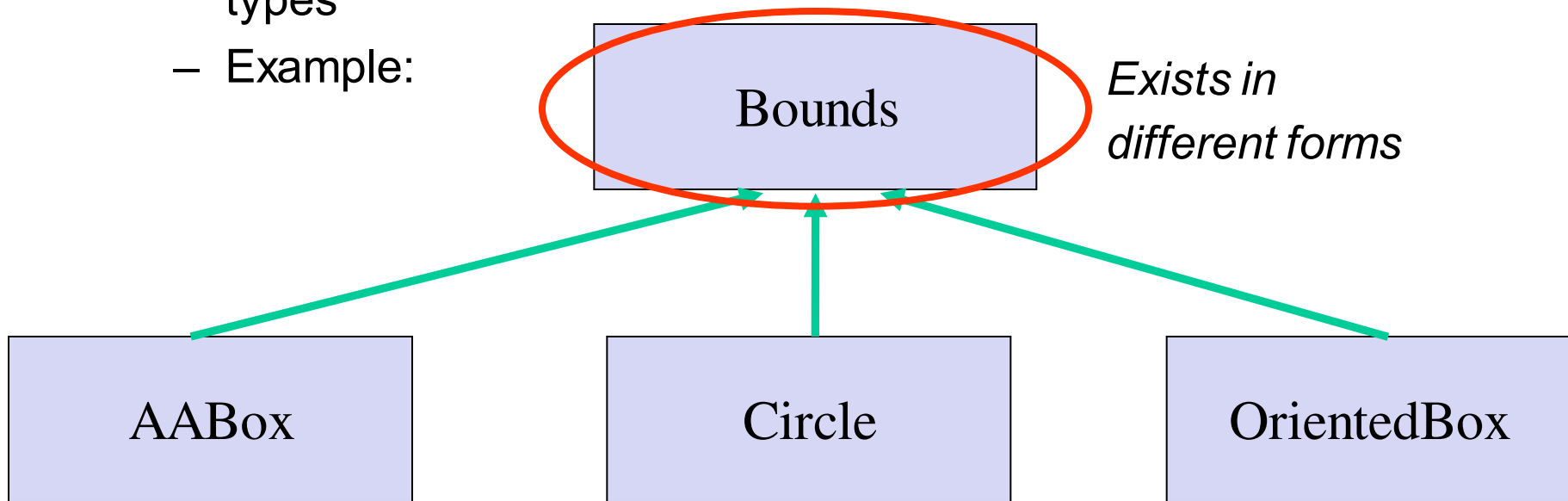
# Polymorphism and Inheritance

- **Goal: Abstraction**

- Base class summarizes the behavior of all derived classes.

- **Concept: Polymorphism**

- A base class handle may give access to different derived types
- Example:

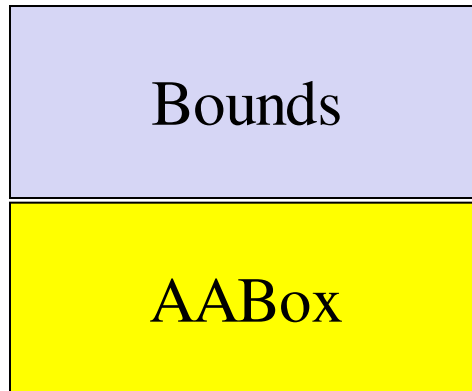


# Example: Polymorphism

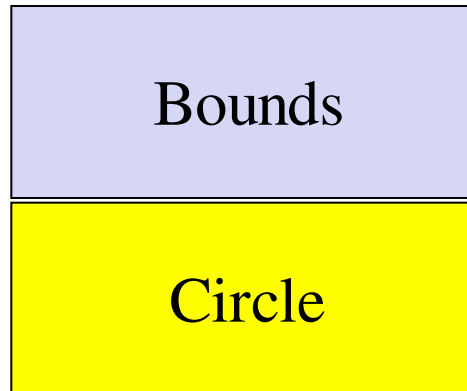
- Call function defined for all bounding primitives
- Execute different code depending on derived class type of Bounds

- **Example: Enclose boundary samples**

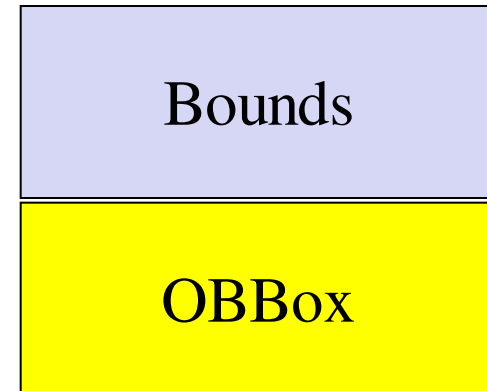
enclose with AAB



enclose with circle



enclose with OBB



## Example in C++: AABox, Circle and OBBBox are Bounds with different properties

```
// Define class hierarchy
class Bounds { ... };
class AABox: public Bounds { ... };
class Circle : public Bounds { ... };
class OBBBox : public Bounds { ... };

// Handles to different bounds
Bounds *boundsA = new AABox();
Bounds *boundsB = new Circle();
OBBBox objOBB; Bounds& boundsC = objOBB;

// Each bounds should behave differently
Point2D pts[4];
boundsA->enclose( pts, 4 ); // use algorithm for AABox
boundsB->enclose( pts, 4 ); // use algorithm for Circle
boundsC.enclose( pts, 4 ); // use algorithm for OBBBox
```

# Dynamic Binding and Virtual Methods

- **In Java all methods are virtual**
  - Virtual machine decides at run-time what method to execute
- **In C++ only methods declared virtual are virtual**
  - Dynamic binding is only invoked when pointer or reference to an object is used
  - Type of object is not known until run time but base object is guaranteed to be part of object

# Definition of a Virtual Function

```
class Bounds {    ...
public:
    virtual bool isInside( const Point2D& _qPt ) const;
    virtual bool enclose( Points2D _extrema[], int _size ) {
        return false; }
};

class AABox : public Bounds { ...
public:
    virtual bool isInside( const Point2D& _pt ) const {
        if ( _pt.isSmaller( d_upperRight ) &&
            _pt.isGreater( d_lowerLeft ) )
            return true; else return false;
    }
};
```

- **Keyword virtual in derived class optional; once a base class defined function virtual**

# Override

- **As soon as we define a function with the same signature than a virtual function in the base class, we override the function: In C++11, we can use `override`**

```
class Bounds {    ...
public:
    virtual bool isInside( const Point2D& _qPt ) const;
    virtual bool enclose( Points2D _extrema[], int _size );
    virtual double area();
};

class AABox : public Bounds { ...
public:
    virtual bool isInside( const Point2D& _pt ) const;
    bool enclose( Points2D _extrema[], int _size );
    double area() override;
};
```

All virtual functions overriding functions in Bounds





# Call of a Virtual Function

```
// use a reference to base class (could change for pointer)
bool isInside( const Bounds& _bounds, const Point2D& pt ) {
    auto res = _bounds.isInside( pt );
    return res;
}

int main() {
    AABox aab;
    Circle circ;
    Point2D pts[4];
    aab.enclose( pts, 4 ); circ.enclose( pts, 4 );
    Point2D pt( 2.0, 3.0 );
    cout << "Inside AA_Box? " << isInside( aab, pt );
    cout << "Inside Circle? " << isInside( circ, pt );
}
```

# Example: Virtual Destructors

- **Base class Bounds**
  - No need for a dtor by itself; built-in is just fine
  - But we don't know about derived classes.
- **Derived class**
  - Virtual dtor provides a hook for dtor of derived class

```
class Bounds {  
public: ...  
    virtual ~Bounds() {};  
};  
class AABox : public Bounds { ...  
    ~AABox();  
};
```

# Virtual Destructors (Dtors)

- **Base classes require a virtual destructor**
  - Consider what destructor needs to be called:

```
int main() {  
    Bounds* boundShape = new Bounds();  
    Bounds* boxShape = new AABox();  
    delete boundShape; // delete a Bounds  
    delete boxShape; // delete a AABox by deleting a Bounds  
}
```

- A virtual destructor is needed in the base class, even if it does not need a destructor itself. Because an object of a derived class may be destructed through a pointer to the base class!

# Abstract Classes

- **Familiar Concept from Java**
  - No objects of an abstract class
  - Abstract classes serve purely as a base class
- **Differences to Java**
  - No keyword abstract
  - Methods are non-virtual by default
  - No interfaces; can use abstract classes with only pure virtual methods instead

# Pure Virtual Functions and Abstract Classes

- One pure virtual functions make a class abstract
- Abstract classes similar to Java
- Objects of abstract classes can not be generated
- Abstract classes with only pure virtual functions serve similar role than Java interfaces

```
class myAbstractBase {  
public:  
    virtual float myVirtualFunc() = 0;  
}
```

# Improvements for Implementing Class Hierarchies in C++11

- **Inheritance of constructors**
  - Inherent parent constructors for the derived class
  - Constructors “change” their name
  - Default and copy constructor are not inherited but synthesized in the usual way
  - Access level remains the same

```
class GoalPoint2D : public Point2D {  
public:  
    using Point2D::Point2D; // All constructors from Point2D  
};  
...  
GoalPoint2D( 2.0, 4.0 );
```

# Dynamic Cast

- **Run-time cast**
  - Not a compile-time cast than other named casts
  - No old-style cast equivalent
  - Works with handles: object pointers or object references
- **Uses**
  - Down-casts
    - turning a base-class into a sub-class
  - Cross-casts
    - multiple inheritance, casting between different parent hierarchies

# “Down-Casts”

- Abstraction often implies using a base class to represent a sub-class in interfaces
- However: Would like to interact with sub-class
- **Solution: Cast the object down the hierarchy**

```
Bounds *getBounds() {  
    Bounds* res = new AABox();  
    return res;  
}  
Bounds *myBounds = getBounds();  
AABox *myAABox = dynamic_cast<AABox*>( getBounds() );
```



# Errors using Dynamic Casts

- **Dynamic casts can produce run-time errors**
  - Pointers returned by cast is 0
  - Reference cast will throw exception `bad_cast`

```
Bounds *getBounds() {  
    return new Circle();  
}  
Bounds& getBoundsAsRef() {  
    AABox *res = new AABox();  
    return *res;  
}  
// myObb will be 0  
OBBox *myObb = dynamic_cast<OBBox *>( getBounds() );  
// Line below will throw  
OBBox &myObbRef = dynamic_cast<OBBox &>( getBoundsAsRef() );
```

# Next

## Text is beautiful

- **Input and output streams**
  - Relevant classes for STL Stream I/O
  - File handling
  - Overloading the insertion and extraction operators
  - String streams