Overview

- Inheritance
  - Base class and sub class
  - Access specifiers
  - Constructors
  - Multiple Inheritance
- Polymorphism
  - Static vs. Dynamic Type
  - Generic Methods and Types
  - Virtual Methods
- Abstract Classes/Interfaces

Inheritance

- Inherit to create hierarchy of classes
  - Bases class provides basic/shared functionality
  - Subclass extends base class for specific functionality

- Subclass inherits
  - Member variables
  - Implemented methods and interfaces

Inheritance

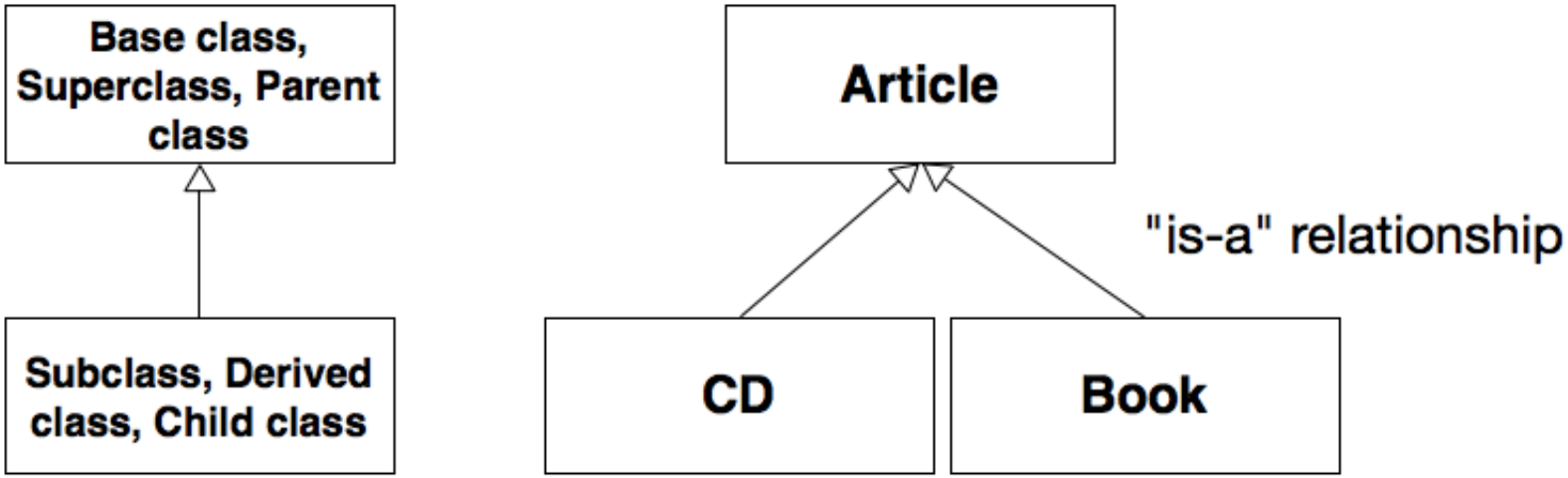
- Subclass may
  - use existing members and methods
  - add new members and methods
  - override members and methods

- Inheritance is transitive (a derived class inherits from all its ancestor classes)
  If `C` is derived from `B` , and `B` is derived from `A` , then `C` inherits the members declared in `B` and `A`
- Subclass can not remove inherited members

Inheritance

- Subclass extends the base class
- Subclass is a "specialization" of the base class
- Subclasses are compatible with the base class

Example



Base Class

Common functionality should be in a base class!

Example articles

- Price
- Number
- Title/Name

```
class Article {
private:
 string m_number;
 string m_title;
 double m_price;
public:
 Article() { ... }
 string getNumber() const { return m_number; }
 string getTitle() const { return m_title; }
 double getPrice() const { return m_price; }
};
```

Derived Class

Adds special functionality for specific use cases

Example book

- Author
- Publisher
- ISBN Number

```
class Book: public Article {  // Book is derived from Article
private:
 string m_author;
 string m_publisher;
 double m_isbn;
public:
 Book() { ... }
 string getAuthor() const { return m_author; }
 string getPublisher() const { return m_publisher; }
 string getIsbn() const { return m_isbn; }
};
```

Derived Class

Class `Book` inherits all features of `Article`

```
Book book;
book.getAuthor();
book.getPublisher();
book.getIsbn();

// Method from super class
book.getNumber();
book.getTitle();
book.getPrice();
```

Inheritance Access Specifiers

Base class may be provided with an access specifier

```
class Derived: public Base { ... };
class Derived: protected Base { ... };
class Derived: private Base { ... };
class Derived: Base { ... };
```

Restricts access to `Base` for the children of `Derived`

- public: all access specifiers in `Base` preserve meaning
- protected: public members of `Base` become protected (visible for subclasses)
- private: all members of `Base` become private (default)

Typically, simply public inheritance is applied

Inheritance Access Specifiers - Example

```
class Book : Article {};

book.getAuthor(); // Compiler error
```

```
class Book : protected Article {};

book.getAuthor(); // Compiler error
```

```
class Book : private Article {};

book.getAuthor(); // Compiler error
```

Constructors

- The constructors of a base class are not inherited
  - Derived class is responsible for initializing data members of base class

- A derived class must define its own constructor
  - May call (in its initalization list) a constructor of the base class
  - Otherwise, the default constructor of base class is called first

Constructors - Example

```
class Book: public Article {
public:
 Book(string number, string title, double price, string author,
string publisher, string isbn)
  : Article(number, title, price), m_author(author),
m_publisher(publisher), m_isbn(isbn) {}

};
```

Multiple Inheritance

In C++ a class may have multiple parents

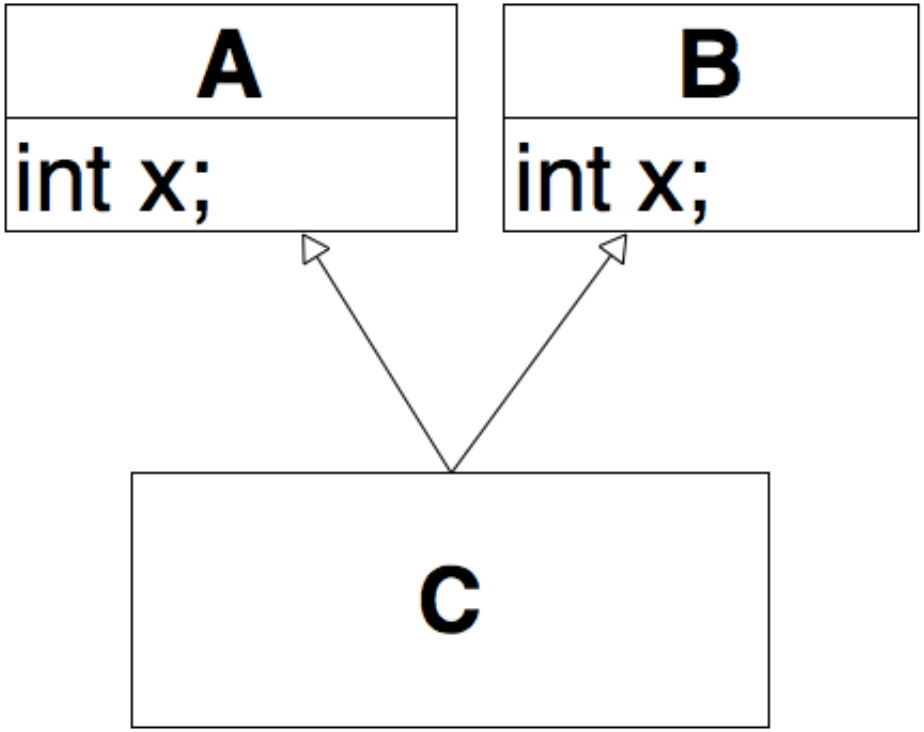
```
class Derived : public Base1, public Base2{
 ...
};
```

- `Derived` inherits from `Base1` and `Base2`
  - Object contains separate "subobjects" for each base class

- Hierarchy is a graph (not a tree)
  - Name conflicts possible

Multiple Inheritance

Name clashes have to be resolved by qualification



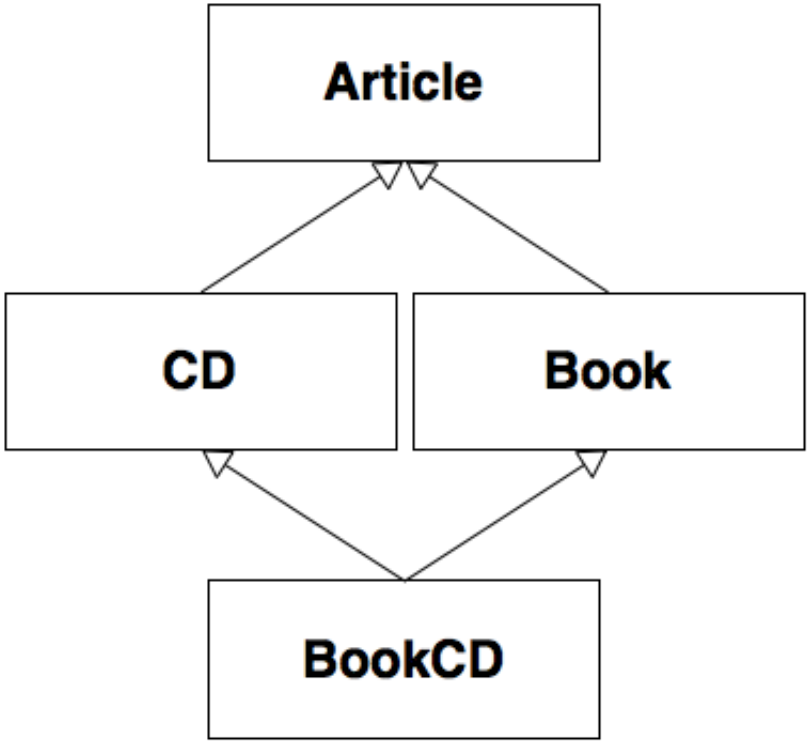
Resolve ambiguity in `C` with scope operator `(A::x, B::x)`

Share base class object - Diamond Problem

```
class Book: public Article
{}
class Cd: public Article
{}
class BookCD: public Book,
public Cd {}

BookCD bookCD;
bookCD.getPrice(); //
Ambiguous
bookCD.Book::getPrice();
bookCD.Cd::getPrice();
```

Class



`BookCD` contains two separate subobjects of `Article`

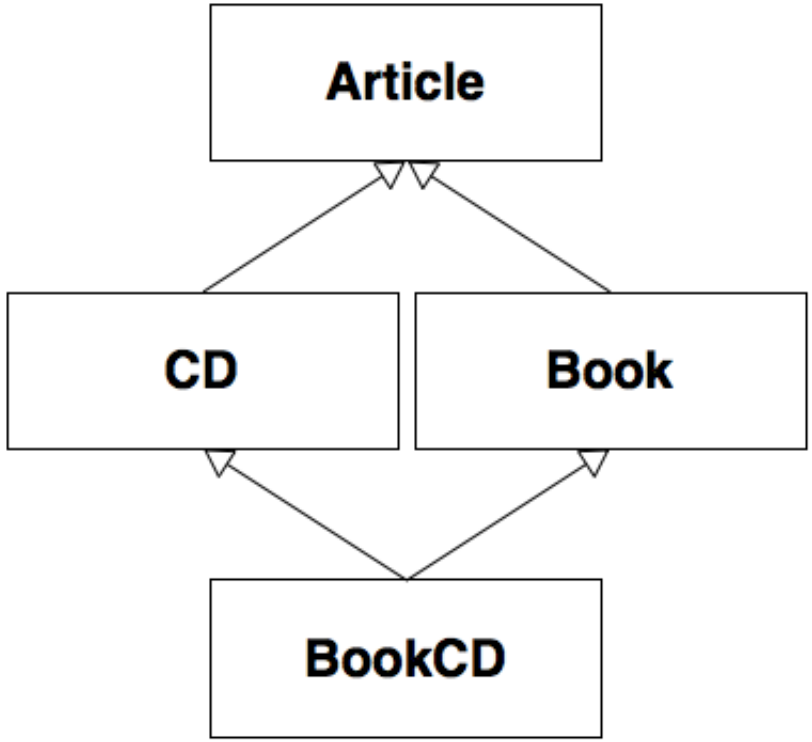
Method `getPrice();` is available in both super classes

This situation is sometimes referred to as diamond inheritance because the inheritance diagram is in the shape of a diamond. Virtual inheritance can help to solve this problem.

Share base class object - Virtual Access Specifier

```
class Book: public virtual
Article {}
class Cd: public virtual
Article {}
class BookCD: public Book,
public Cd {}

BookCD bookCD;
bookCD.getPrice(); //
Unambiguous
```



Specifier `virtual` lets corresponding subobjects be merged to one

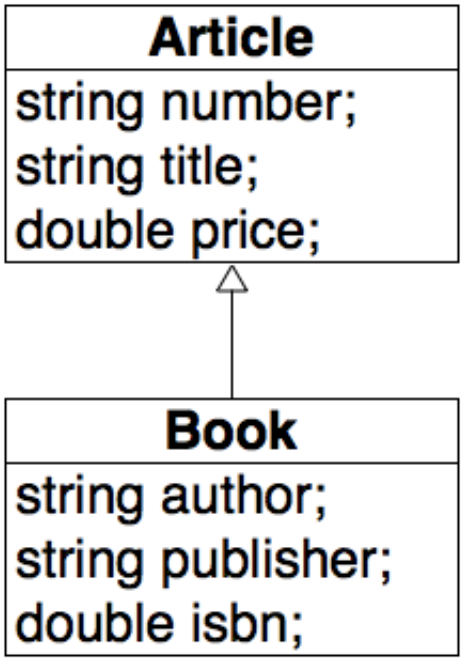
Polymorphism

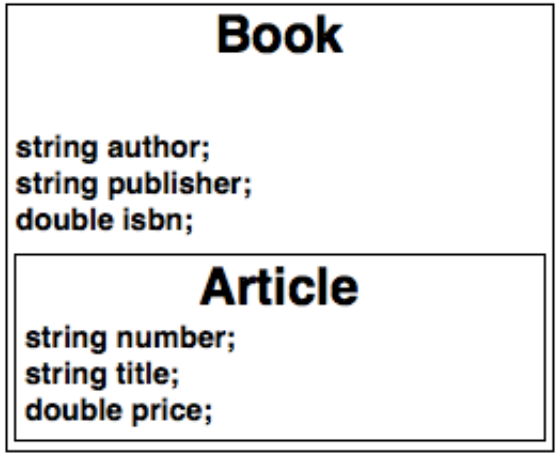
- Objects of the subclass type can be used instead of objects of the baseclass type
- Subclass is compatible with the base class
- Code that works with objects of a certain class also works with objects of subclasses (e.g. collection classes)
- In C++ solved with pointers

```
void printArticle(Article* a) {
 cout << a->getTitle();
}

Article* pArticle = new Article("100", "Article1", 9.90);
Book* pBook = new Book("200", "C++", 24.90, "Stroustrup", "", "1-
23");
printArticle(pArticle);
printArticle(pBook);
```

Polymorphism






```
Article* pArticle; // Static type "Article"
Book* pBook; // Static type "Book"

Article* pArticle = new Book(); // Static type "Article", dynamic
type "Book"
```

Static vs. Dynamic Types

Static Type

- The type appearing in the declaration
- Is known at compile time
- Determines which members can be accessed

Dynamic type

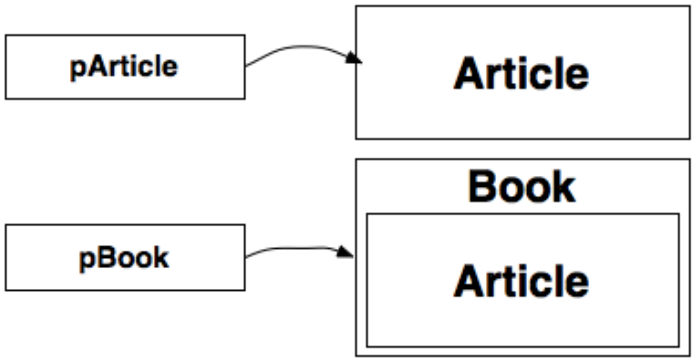
- The type of the object stored at runtime (can change)
- May be (directly or indirectly) derived from the static type
- Determines which virtual member functions are called (see later)

A polymorph variable (class needs a virtual function) may refer to objects of the static type or of any subclass

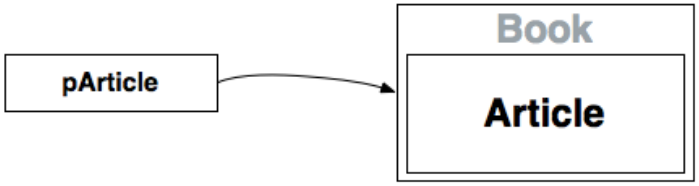
Static vs. Dynamic Types

```
Article* pArticle = new
Article("100", "Article1",
9.90);
Book* pBook = new Book("200",
"C++", 24.90, "Stroustrup", "",
"1-23");

pArticle->setTitle("Basic
Article"); // ok
pBook->setPublisher("Addison-
Wesley"); // ok
```




```
pArticle = pBook;
pArticle-
>setPublisher("dpunkt"); //
compiler error
pArticle->setPrice(29.90); //
ok
```



Dynamic Casts

A pointer variable of a subclass type can always be assigned to a pointer variable of the baseclass

Inverse assignment is not possible - Cast is needed

```
Article* pArticle;
Book* pBook = new Book("123", "C++", 24.90, "Stroustrup", "", "1-
23");
pArticle = pBook; // ok
pBook = pArticle; // error
pBook = dynamic_cast<Book*>(pArticle); // ok
```

dynamic_cast: Checks whether pointer points to an object of class `Book` (or subclass), If yes, it returns a pointer of type `Book*` to the object; if not, a `NULL` pointer is returned

Generic Methods

Generic methods can operate on arguments of multiple dynamic types

```
void printInfo(Article* a) {
 cout << "Article " << a->getTitle();
 cout << " (" << a->getNumber() << "); ";
 cout << a->getPrice() << " Euro" << endl;
}

Book* pBook = new Book(...);
Cd* pCd = new Cd(...);
printInfo(pBook);
printInfo(pCd);
```

Generic Types

Generic containers can contain elements of multiple dynamic types

```
class ShoppingCart {
 ...
 void add(Article* a);
 Article* getArticle(int index);
};

ShoppingCart cart(...);
Book* pBook = new Book(...);
C* pCd = new Cd(...);
cart.add(pBook);
cart.add(pCd);
Article* article = cart.getArticle(0); // may be book or Cd
```

Override methods

Inherited methods can be overridden

- Same interface (same method signature)
- 2 forms of override
    - Static override: Static Binding
    - Dynamic override: Dynamic Binding

Inherited methods can be overloaded

- Different interface

Static Override

```
class Article{
public:
 void printInfo(){
  cout << "Article: " << m_title;
 }
};

class Book: public Article{
public:
 void printInfo(){
  cout << "Book: " << m_title << "|" << m_author;
 }
};
```

```
void main(){
 Book* pBook = new Book("123", "C++", 24.90, "Stroutrup", "", "1-
23");
 Article* pArticle = pBook;
 pArticle->printInfo(); // what is printed out?
```

Virtual Methods

- In C++, methods are statically linked per default
- Keyword virtual is necessary for dynamically linked methods
- Only non-static methods can be virtual
- When a virtual function is called on an object, the function of the object's dynamic type is executed
- Buzzword: Dynamic Binding
- Dynamic Binding works only with polymorph variables (class needs min. one virtual method)
- Base function may be still called
  `object->Base::func(...)`
- In Java all methods are virtual and dynamically linked

Virtual Methods

```
class Article{
public:
 virtual void printInfo(){
  cout << "Article: " << m_title;
 }
};

class Book: public Article{
public:
 void printInfo(){
  cout << "Book: " << m_title << "|" << m_author;
 }
};
```

```
void main(){
 Book* pBook = new Book("123", "C++", 24.90, "Stroutrup", "", "1-
23");
 Article* pArticle = pBook;
 pArticle->printInfo(); // what is printed out?
```

Generic Types/Methods

```
class ShoppingCart {
private:
 int _number;
 Article* _articles[10];
public:
 void add(Article* a){...}
 void printArticles(){
  for(int i = 0; i<_number;i++){
   _articles[i]->printInfo();
  }
 }
};

ShoppingCart cart(...);
Book* pBook = new Book(...); cart.add(pBook);
C* pCd = new Cd(...); cart.add(pCd);
cart.printArticles();
```

Core of object-oriented programming: generic types/methods call the methods associated to the dynamic types of their elements/arguments.

Virtual Destructor

By default the destructor of a class is not virtual

- If an object is deleted, the destructor of its static type is called
- In most situations, this is not what is wanted/expected

A destructor can be declard as virtual in the base class

- Then the destructor of the dynamic type is called
- The destructors of derived classes automatically get virtual

Virtual Destructor

```
class Base {
 virtual ~Base() { ... }
};

Base* object = new Derived();
delete object; // Derived::~Derived() is called
```

A class with virtual functions should also have a virtual destructor

Abstract Classes

- Classes which can't be instantiated (no object can be created)
- May serve as static types but not as dynamic ones
- A class is abstract if at least one method is declared as pure virtual (= abstract method)
  ```
  virtual void draw() = 0;
  ```
- Abstract method may contain default implementation
- Define an interface (no interface keyword in C++)
- Requires implementation of parts or the whole interface in the subclasses

Abstract Classes

```
class GraphicObjects{
public:
 virtual void draw() = 0;
};

class Rectangle : public GraphicObject{
public:
 virtual void draw(){
 ...
 }
};

GraphicObject* pGraphicObject = new GraphicObject(); // Compiler
error
GraphicObject* pGraphicObject = new Rectangle();
pGraphicObject->draw();
```

Interfaces

An interface

- is an abstract class with only pure virtual functions
- defines an abstract data type
    - Defines only the signature (operations) of a data type

A concrete class

- Represents an implementation of the interface (data type)
- Defines the concrete representation and the concrete realization of the operations on the type

> Interfaces make your software more flexible and modular!

Interfaces

Java

```
public interface Serializable{
 void writeObject(ObjectOutputStream out);
 void readObject(ObjectInputStream in);
}
```

C++

```
class Serializable{
 virtual void writeObject(ObjectOutputStream &out) = 0;
 virtual void readObject(ObjectInputStream &in) = 0;
};
```

Interface implementation

Example

```
class Rectangle : public Serializable {
 virtual void writeObject(ObjectOutputStream &out) {
  out << _x << _y << _width << _height;
 }
 virtual void readObject(ObjectInputStream &in) {
  in >> _x >> _y >> _width >> _height;
 }
};

class Triangle : public Serializable {
 virtual void writeObject(ObjectOutputStream &out) {
  out << p1.x << p1.y << p2.x << p2.y << p3.x << p3.y;
 }
 virtual void readObject(ObjectInputStream &in) {
  in >> p1.x >> p1.y >> p2.x >> p2.y >> p3.x >> p3.y;
 }
};
```

Interfaces

```
class SaveToFileOperation{
private:
 vector<Serializable*> _serializeAbleObjects;
public:
 void save(string filename){
  ObjectOutputStream out; // get output stream
  for(int i = 0; i < _serializeAbleObjects.size(); i++){
   _serializeAbleObjects.at(i)->writeObject(out);
  }
 }
};
```

Structs vs Classes

Structs are very similar to classes

Main difference is the visibility of the members and inheritance

- Members of structs are public as default
- Members of classes are private as default

- Inheritance between structs is public as default
- Inheritance between classes is private as default

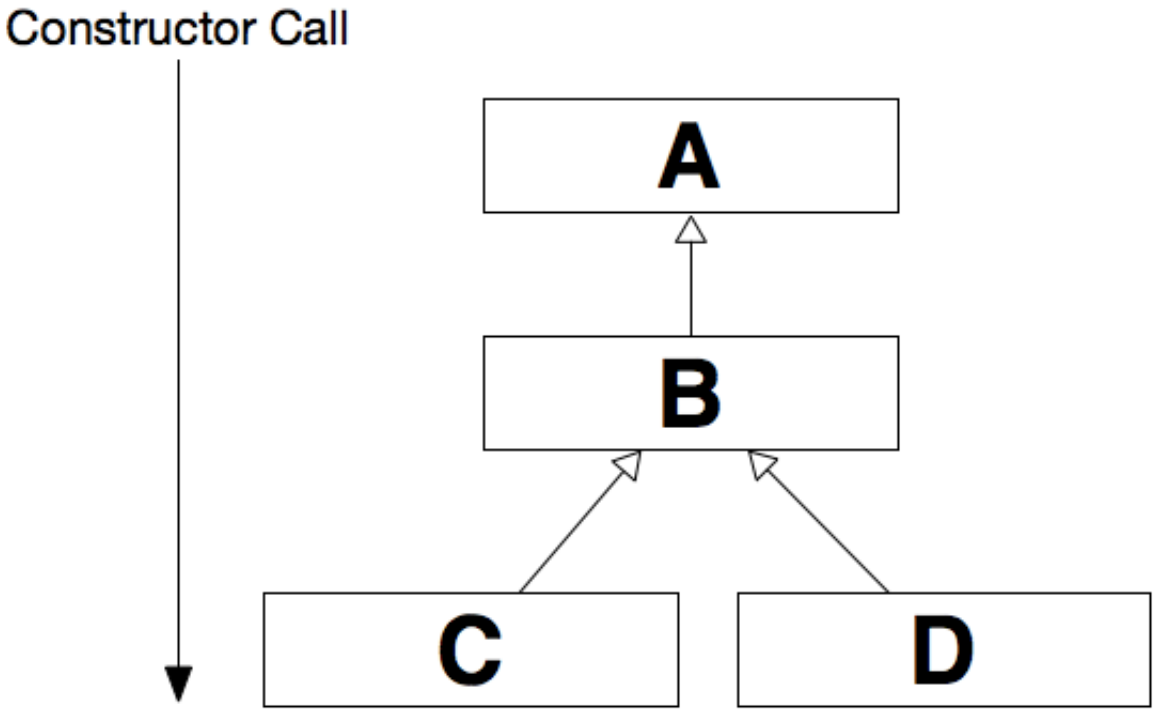
Structs vs Classes

```
struct Rectangle : GraphicsObject{
 // Members are public
 int _x;
 int _y;
};

class Rectangle : GraphicsObject{
 // Members are private
 int _x;
 int _y;
};
```

Constructor Call Order

Constructors are called from the top to bottom



Destructor Call Order

Destructor are called from the bottom to top