Object-Oriented Concepts

Lecture Objectives

- Understand object-oriented software development concepts
  - Types, classes, and objects
  - Encapsulation
  - Inheritance
- Understand the relationships between Object-Oriented Programming (OOP) and Object-Oriented Analysis and Design (OOA and OOD)

OO Development

- OO Development is the predominant paradigm for the modern software life-cycle
- Life Cycles phases vary depending on the size and type of project, but can include:
  - Business Analysis - what are the *business processes* and *actors* that the software will interact with (e.g., a University Student Registration system)
  - Requirements Analysis - what are the *actors* and *use cases* for the software (e.g., students, professors, … ; registering for a course, …)
  - Design - what are the objects and classes of the software (e.g., course, subject, transcript, …)
  - Code and unit test
  - System and Integration Testing
- **High quality software requires close attention to analysis and design, not just programming!**

OO Development (cont.)

- In this course, we focus on programming
- However, the connection between programming and design and analysis needs to be emphasized:
  - What is a “good” candidate class or object?
  - How can we identify the operations (methods) and attributes of such classes and objects?
  - How should we define relations between classes and objects?
- **For all but trivial projects, classes and their relations need to be designed before coding commences!**
- Fortunately, there is single international standard for OOA&D
  - The Unified Modelling Language (UML), which is well supported by open-source and commercial software tools.

OO Development (cont.)

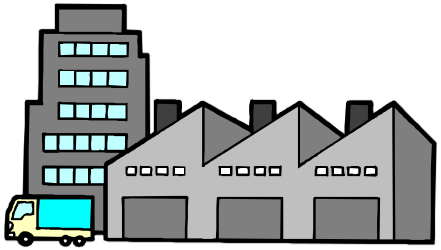
- We will assume that analysis has been done
  - We have written descriptions of what the software should do, either as point-form requirements or use cases
- The focus is on *guidelines* for answering the following design questions:
  - What is a “good” candidate class or object?
  - How can we identify the operations (methods) and attributes of such classes and objects?
  - How should we define relations between classes and objects?
- OOD is very *subjective* - tradeoffs need to be made between simplicity and adaptability based on through domain knowledge and business constraints.

Objects

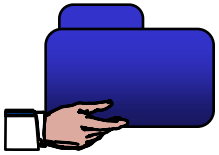
- From Grady Booch, *Object-Oriented Design with Applications*: "An *object* has *state*, *behaviour* and *identity*…"
  - *State* means data: information, attributes, or properties
  - *Behaviour* means operations that query or change the object's state
  - *Identity* means that each object must be unique and precisely defined
- OO systems model real-world systems by simulating real-world objects
  - But only the behaviour and state relevant to the system being implemented

Objects (cont.)

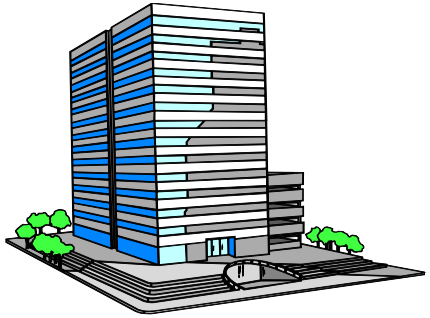
- Which of the real-world entities might correspond to objects in a computer system, and why?



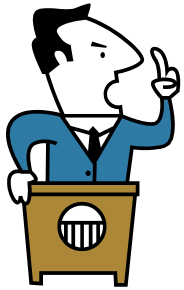
ACME's London Warehouse



Throckmorton's order from ACME



The World Bank



The Government

Objects (cont.)

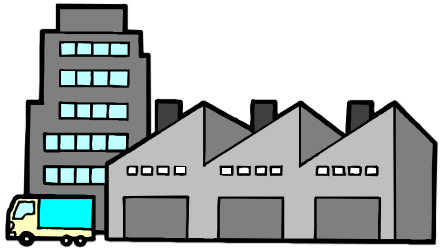
- "The Government" is not a well-defined identity
  - e.g. do we mean State, Federal, Local or overseas governments?
- But the other three examples all have state, behaviour and identity
  - The warehouse has stock that can be shipped
  - The order has items and a price and can be cancelled
  - The world bank has member countries that contribute funds
- *Candidate objects* can be recognized quite easily from written requirements and use cases.
- A safe assumption would be that p**roper nouns are candidate objects.**
  - Proper nouns in English are capitalized, or preceded by "the …"
- Many proper nouns in a requirements document are vague or describe external entities
  - e.g., "the system", "the user", …

Classes

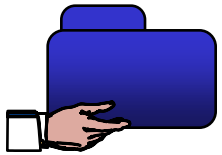
- Similar objects are classified, or grouped, into $classes$
  - All have the same data members, though with different values
  - All share identical behaviour
- An object belongs to only one class
  - Specified when the object is created
  - Inheritance allows classes to have common properties and operations
  - The $\boldsymbol{state}$ design pattern allows an object to appear to change its class

Classes (cont.)

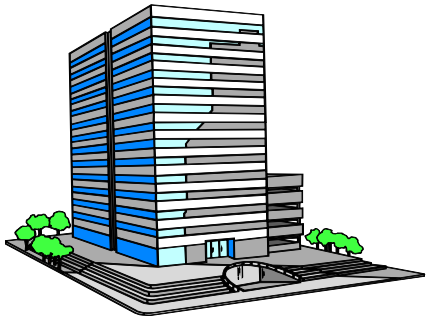
- To what classes in a computer system might the following objects belong?



ACME's London Warehouse



Throckmorton's order from ACME



The World Bank

Classes (cont.)

- Possible answers
  - Acme's London Warehouse - class `Warehouse`
  - Throckmorton's Order from Acme - class `Order`
  - The World Bank - class `WorldBank`
- Sometimes there should only be one instance of a class
  - This is usually known at design time
  - There is only one `World Bank`, in a class of its own
  - At the programming level, the code should enforce this

Classes (cont.)

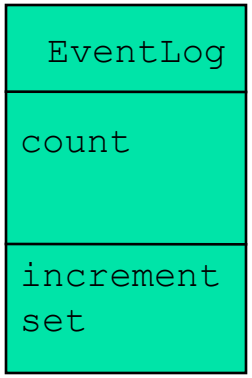
- What programming or design techniques might we use to achieve this in Java or another OOP?
- The *singleton* design pattern will solve the problem
  - A design pattern is just a standard way of solving a common design and coding problem
  - Design patterns were first described in the book "Design Patterns" by Gamma, Helm, Johnson, and Vlissades (the "GoF") in 1995

Classes (cont.)

- Once we have identified classes, it is useful at design time to identify the operations and attributes of those classes
  - Operations (***methods***), are things you can do to that class's objects
  - Attributes are properties of objects of that class (may be constants)
- Candidate attributes, methods, and classes can all be identified from a natural language description of the application or use cases
  - Common nouns attributes or classes
  - Verbs methods

Classes: An example

- Suppose we wish to implement a simple event log
  - Just counting the number of events
  - Later we will develop a more sophisticated event log
    - That record details of the events
- The operations on the EventLog will be set(int) and increment()
  - The increment operation will return the new count of the EventLog



Classes (cont.)

```
// File "EventLog.h"
class EventLog {
  // default is private
  int count;
public:

  int increment() {
    return ++count;
  }
  void set(int value) {
    count = value;
  }
};
```

```
#include "eventlog.h"
#include <iostream>

using namespace std;

int main() {
  EventLog c1, c2;
  c1. set(0);
  cout << c1.increment();
  c2 = c1;
  cout << c1.increment();
  cout << c2.increment();
  return 0;
}
```

Output is:
1 2 2

- Class objects may be assigned, passed as arguments, and returned by functions

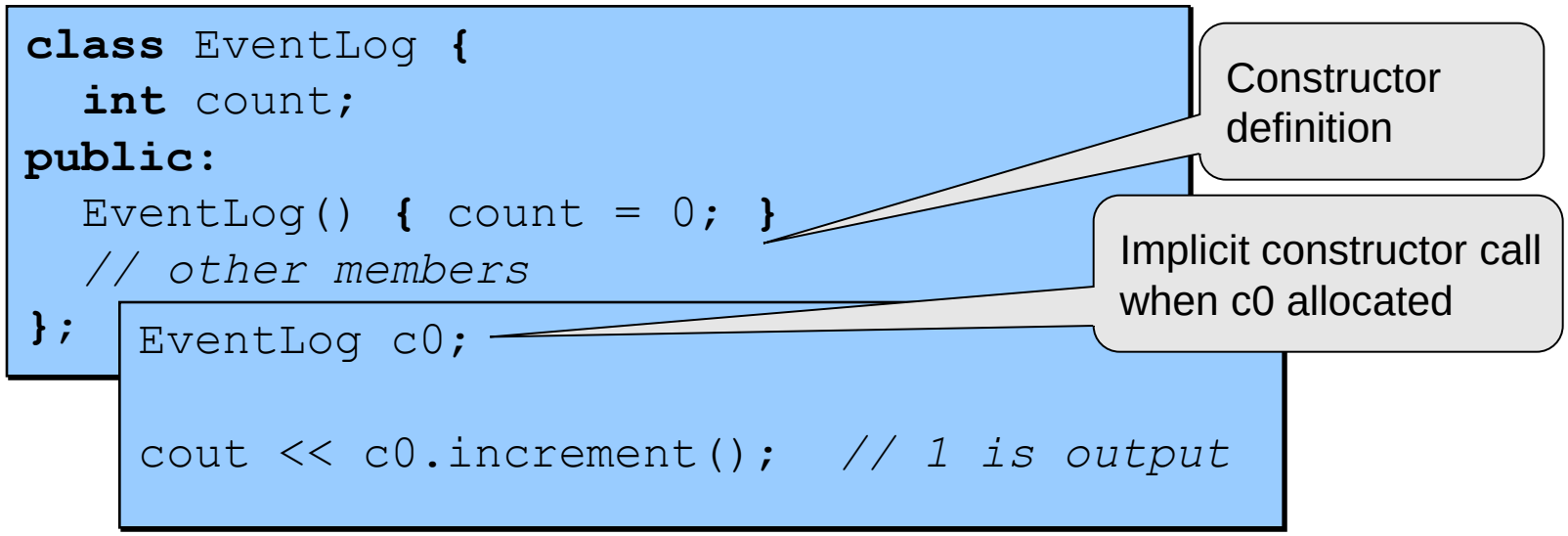
```
c2 = c1;  // 'memberwise copy':
          // same as c2.count = c1.count;IF count were public
```

Classes (cont.)

- Java class variables are very different!
  - Java class variables are references to dynamically allocated objects
    - Assigning one variable to another makes them both refer to the same object
  - What would the program output have been if EventLog was a Java class?
- Our class requires that the user call `set(0)` to initialize an EventLog object before it is used
  - In C/C++ global data is initialized to zero, but it is unsafe to rely on implicit initialization
  - How can we ensure that an object is initialized correctly?

Constructors

- A member function with the same name as the class is a constructor
- Constructors are invoked whenever a class instance is allocated
- **Each class should have a constructor member function**



Constructors (cont.)

- C++ provides another syntax for initializing data members: a member initialiser list

```
class EventLog {
  int count;
public:
  EventLog() :
    count(0) // member initialiser list
  {}         // empty constructor body
  // other methods
};
```

- Member initialiser lists have a more object-oriented style
  - Data members are passed arguments to initialize themselves
- Member initialiser lists must be used for initializing base classes, data members with constructors, and constant data members

Constructors (cont.)

- A constructor cannot be explicitly called as a member function

```
EventLog c0;   // constructor call: c0.count = 0
c0.EventLog(); // compile error
```

- A constructor call by itself causes a temporary object to be allocated

```
EventLog c1;
c1.increment();
c1 = EventLog(); // In this example, c1.set(0)
                 // would be simpler and more readable
```

  - The call `EventLog()` constructs a temporary `EventLog` object, which is then assigned (copied) to `c1`
  - The temporary `EventLog` object is reclaimed automatically at the end of the statement
- A constructor that can be called with no arguments is called a *default constructor*

Constructors (cont.)

- It is illegal to have a return type, or return expression, in a constructor
- It is also illegal to initialize data members outside the constructor
- A constructor has no explicit return type. It effectively "returns" a properly initialized object of its class

```
class EventLog {
  int count = 0;    // illegal initialization
public:
  void EventLog()   // illegal to have void here!
                    // : count(0) SHOULD BE HERE
  {}                // no code needed in body
};
```

Overloading

- Most modern programming languages provide overloading
  - Multiple meanings for one name or symbol in one scope
  - Overloading happens at compile time

- Functions and operators should have the same name if and only if they perform the same operation on different data types.

Overloading (cont.)

- C++ allows overloading of function names and operators

```
point1.move(1,2);
point2.move(point1);
line1.move(1,2);
```

move could be relative or absolute.
moves in the program should behave the same
if they have the same name.
If they are different, rename as: `move_by` and `move_to`

- Methods in the same class can have the same name if their declarations have different parameter types

File "eventlog.h"
two constructors and
two set methods

Default constructor

```
class EventLog {
  int count;
public:
  void set() { set(0); }
  void set(int value) { count = value; }
  EventLog() : count(0) {}
  EventLog(int init)  : count (init) {}
  int increment() { return ++count; }
};
```

Overloading (cont.)

- One member function can call another; the current object is implicit
  - `set()` calls `set(int)` above
- What if one constructor calls another?
  - If below we used: `EventLog() { EventLog(0); }`

```
class EventLog {
  int count;
public:
  void set() { set(0); }
  void set(int value) { count = value; }
  EventLog() : count(0) {}
  EventLog(int init)  : count (init) {}

  int increment() { return ++count; }
};
```

Overloading (cont.)

- A constructor call just creates a temporary object
  - One constructor should not call another!

```
class EventLog {
  int count;
public:
  EventLog() : { EventLog(0); }
  EventLog(int init) : count (init) {}
  // other members
};
```

The call `EventLog(0)` creates a temporary `EventLog` object. The `count` of this object is not initialized

Overloading (cont.)

```
#include "eventlog.h"
#include <iostream>

using namespace std;

int main() {
  EventLog c1(42);
  EventLog c2;

  cout << c2.increment() << '\n';
  cout << c1.increment() << '\n';
  c1.set();
  c2.set(42);

  cout << c2.increment() << '\n';
  cout << c1.increment() << '\n';
  return 0;
}
```

two overloaded constructor calls

Do not declare as: `EventLog c2();`
Causes a syntax error later as this declared `c2` as a C function returning an `EventLog`

Output

```
1
43
43
1
```

Default Arguments

- Overloading is used for arguments with simple defaults
  - Values used if no argument is supplied by the caller
- C++ provides a simple mechanism for declaring functions with default arguments
  - Trailing arguments can have default initialization
- Using default arguments simplifies our `EventLog` class

```
class EventLog {
  int count;
public:
  void set(int value = 0) { count = value; }
  EventLog(int init = 0): count(init) {}
  int increment() { return ++count; }
};
```

Default argument for the set member function

Default argument for constructor

Default Arguments (cont.)

- A function declaration with defaults is equivalent to several separate overloaded function declarations

```
int doit(int, int=0, int=0, int=0);
// equivalent to FOUR functions: doit(int), doit(int, int),
// doit(int, int, int), and doit(int, int, int, int)
```

- If a parameter has default, all remainings must have defaults

```
int doitagain(int, int=0, int, int=0); // compile error
```

- Intermediate arguments in a call cannot use defaults
  - Used only when no arguments remain

```
doit(11, , , 10); // compile error
```

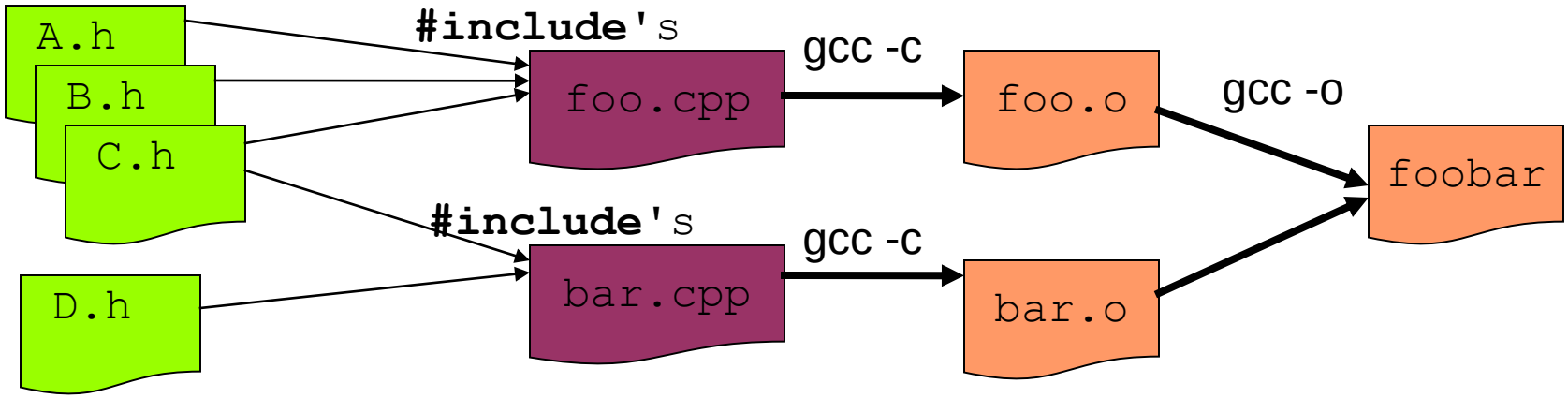
- Overuse of overloading and defaults can cause maintenance headaches

Separate Compilation

- A *compilation unit* consists of a C++ source code file and all the header files that it `#include`s
  - By convention, source code files have an extension `.cpp` or `.cc` and header files have an extension `.h`
- The C++ compiler generates an object file for each compilation unit
  - Object files contain relocatable machine code
  - By convention, object files have an extension `.o` or `.obj`

Separate Compilation (cont.)

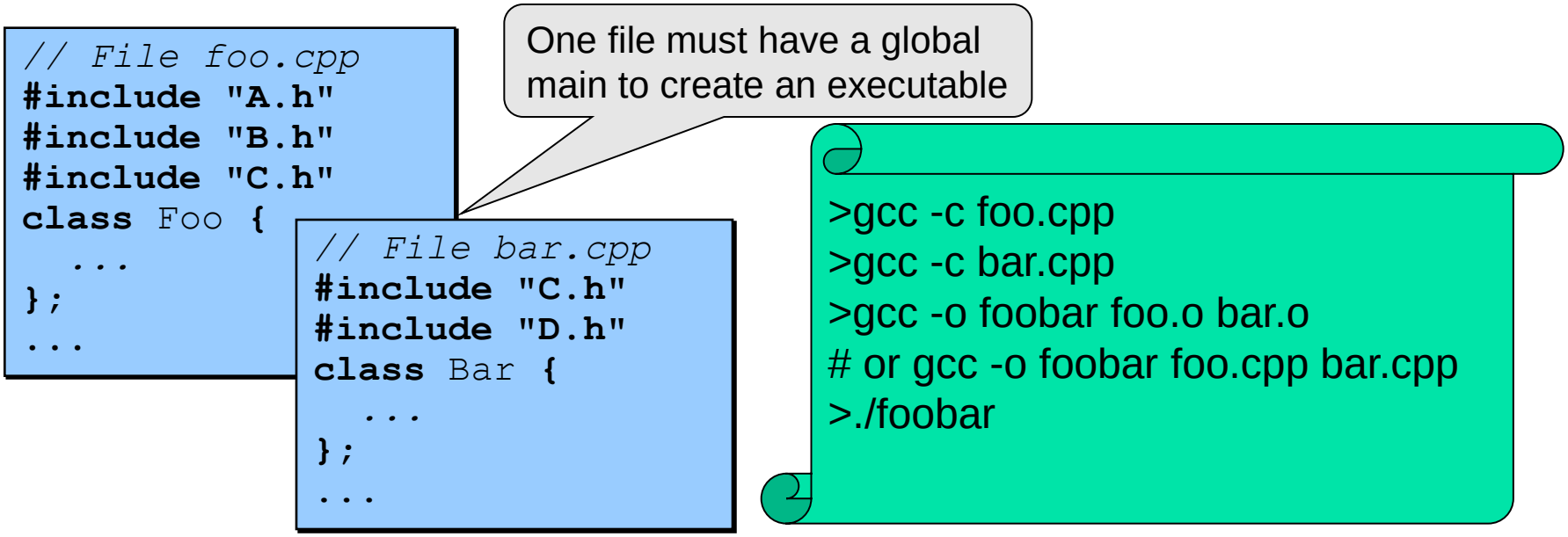
- One or more object files are then linked to form an executable binary file.



- A header file can be `#included` by many source files
  - But should not be included repeatedly within a compilation unit
  - Use of the preprocessor `#ifndef` prevents coincidental repeated includes.

Separate Compilation (cont.)

- Unlike Java, C/C++ imposes no restriction or relation between file names and their contents
- **Good style is one class per file, both with the same name.**



Separate Compilation (cont.)

- Often, some methods are defined in the header file, and some in the '`.cpp`' file
  - Placing short member definitions in the header file is done for efficiency
  - The compiler can *inline* such definitions

```
// File eventlog.h:
class EventLog {
  int count;
public:
  EventLog() : count(0) {}
  int increment();
  void set(int value);
};
```

constructor defined in the `.h` file; `increment` and `set` defined in the `.cpp` file

```
// File: eventlog.cpp
#include "eventlog.h"
int EventLog::increment() {
  return ++count;
}
void EventLog::set(int value) {
  count = value;
}
```

Separate Compilation (cont.)

- An inline function call is replaced by the function code
  - Placing short member definitions in the header file is done for efficiency
  - The compiler can inline such definitions

```
// File eventlog.h:
class EventLog {
  int count;
public:
  inline EventLog()
    : count(0) {}
  int increment();
  void set(int value) {
    count = value;
  }
};
```

inline legal here, but implicit as this is a member function definition in a .h file

```
#include "eventlog.h"
...
Eventlog c1;
int main() {
  ...
  c1.set(42);
  ...
}
```

no code generated, static storage initialization to 0

call replaced by:
`c1.count = 42;`

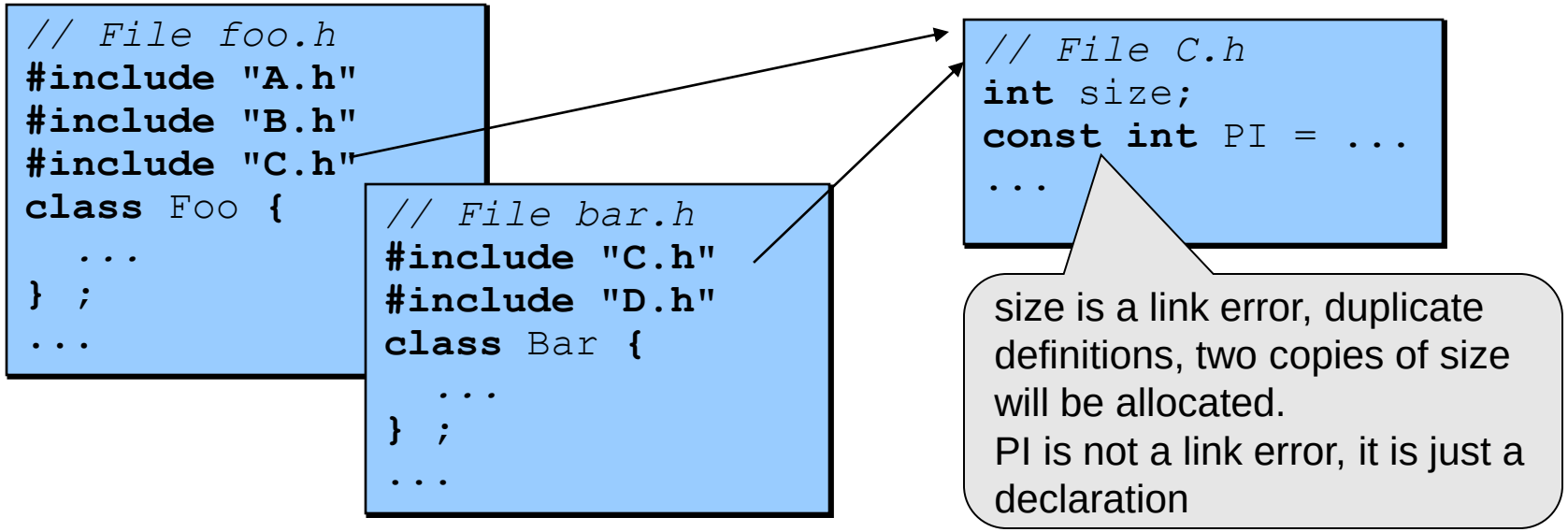
- https://www.youtube.com/watch?v=B2HDcKUT67s

Declarations and Definitions

- In the “compile and link” model that C/C++ use
  - Many compilation units can include references to the *same* library files
  - The executable code for these libraries files needs to be loaded *just once*
  - How to ensure that library code is not duplicated when loaded?

Declarations and Definitions (cont.)

- The solution adopted by C/C++, and many other compiled languages, is that…
  - header files, which can be repeatedly included, should not allocate storage
    - They can contain only *declarations*, not *definitions*



Declarations and Definitions (cont.)

- A declaration (of a variable, function, or class) - in a `.h` file
  - Does not allocate storage
  - Tells the compiler about attributes of the identifier that will be defined elsewhere - in a `.cc` file

```
// some.h or .cpp file
extern int i;                  // variable declaration
int main(int argc, char* argv[]); // function declaration
class Account;                 // class declaration
```

-

class declarations are needed when two class declarations reference each other, e.g.,

```
class B;
// now we can declare a pointer to B
class A { … B* bp; … }
class B { … A* ap; … }
```

Declarations and Definitions (cont.)

- A *definition* of a
  - Variable - allocates storage, specifies initial value (if any)
  - Function - specifies the code to be executed when the function is called
  - Class - declares the members of the class

```
// some .cpp file
int i = 0; // variable definition
int main(int argc, char* argv[]) {
  // function definition
}
class Account {
  // class definition
  // member declarations and definitions
};
```

Declarations and Definitions (cont.)

- Class definitions are needed and allowed in header files in C++
  - The C rule of "no definitions in `.h` files" has had to be relaxed
- Because C++ is downward compatible with C, C++ programs can mix OO and non-OO code:
  - OO code: data and functions members of classes
  - Non-OO (C style) code: global data and functions
- Non-OO code is "unsafe" as all global data is public.
- C++ programs should not mix C style and C++ style programming - it is unsafe and hard to maintain.

UML Notation

- UML is a language that allows modelling object-oriented systems.
- While *class diagrams* are of special interest to this course, it is also important to know how to model requirements!
  - *For this we make use cases!*

Classes (cont.)

- For example, a `create_order` use case:
- `A salesperson enters the customer's number, a new order is created for the customer. The salesperson adds the items and their quantity into the order. ...`
- What classes, operations, and attributes appear in this use case?

Classes (cont.)

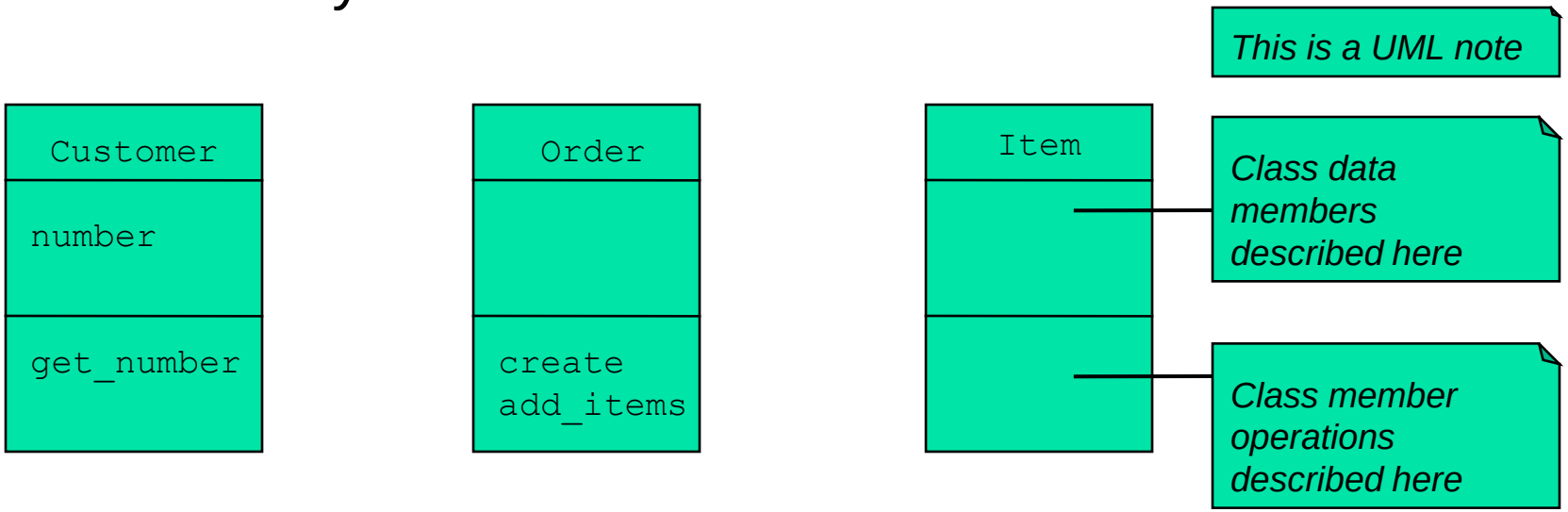
- Possible classes, operations, and attributes for this use case are:
  - class customer - attribute: number
  - class order - operation: create, add_items
  - class item
- salesperson is probably not a class
  - Instead it is an external user of the system, or actor
- Customer number is an attribute as it has no operations - just get and set its value
- item is probably a class as it has attributes and operations

Classes (cont.)

- OOA depends on “domain knowledge”
  - From users and domain experts, not always from written requirements
  - For example, an order might record the id of the salesperson creating the order - then salesperson might need to be an attribute or class

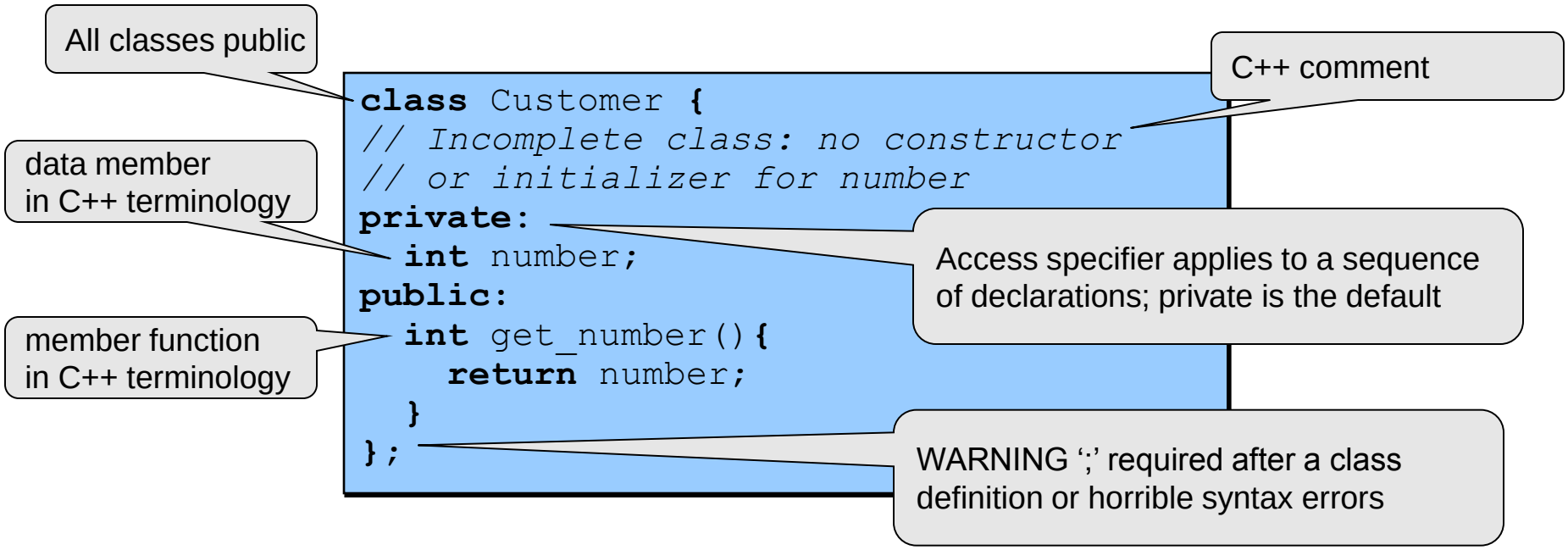
UML Notation - Classes

- UML represents classes as rectangles
  - With the name of the class (required), methods and attributes optional
  - Methods and attributes are added as the design is iteratively refined



C++ Syntax - Classes

- To implement a UML class, we need to know the types of all attributes and methods
- Suppose `number` is a C/C++ `int`, then class `Customer` can be defined as

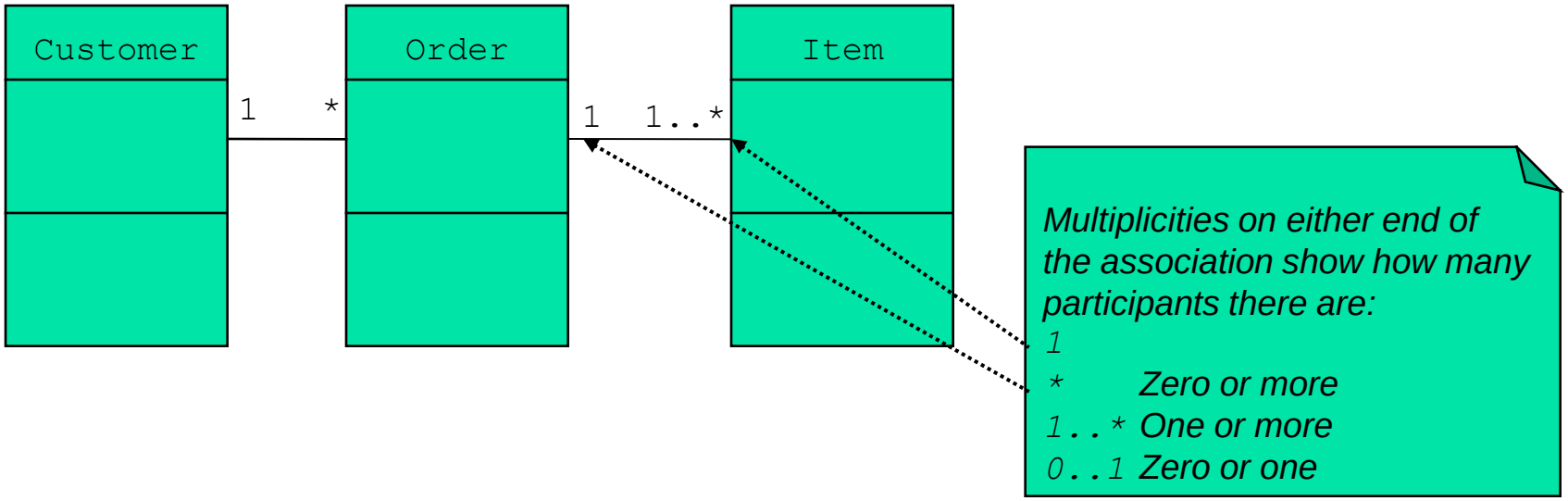


UML Notation - Class Diagrams

- *Class diagrams* show the *static* relationships between classes and objects
  - A `Customer` will have zero or more `Orders`
- There are four types of static relationships in UML class diagrams
  - ***Association - "has a"***
    - ***Aggregation*** - a special form of association
  - ***Inheritance - "is a (kind of)"***
  - ***Dependency - "uses a"***
- These phrases are used in analysis to determine the class relationships
- What verb phrase best describes the relation between an `Order` and an `Item` - "has a", "is a", or "uses a"?

UML Association

- Association models the relation “has” between objects



- What might the attributes of an `Item` be?
- Are any attribute values shared or are they unique to each `Order`?

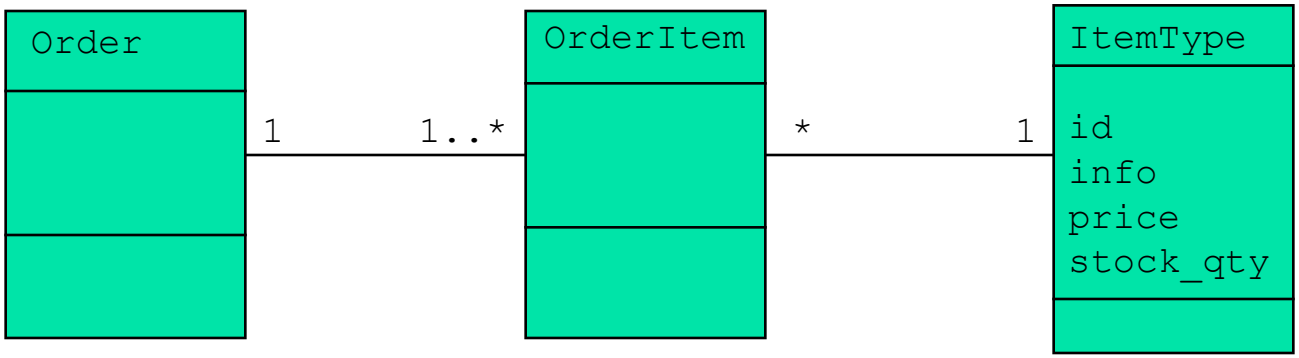
Refactoring Classes

- The attributes of an Item might include:
  - `id (bar code)`,
  - `info`,
  - `price`,
  - `quantity_in_order`,
  - `quantity_in_stock`
- The attribute values of `id`, `info`, `price`, and `quantity_in_stock` are shared by many Items, but `quantity_in_order` is unique to each Order…
- Is such sharing desirable? If not, how can we avoid it?

Refactoring Classes (cont.)

- **Duplication is considered undesirable in any OO design or program**
  - It can be removed by mechanisms including ***inheritance***, ***delegation***
  - In this case, we use ***normalization*** - split classes to remove duplicate data:

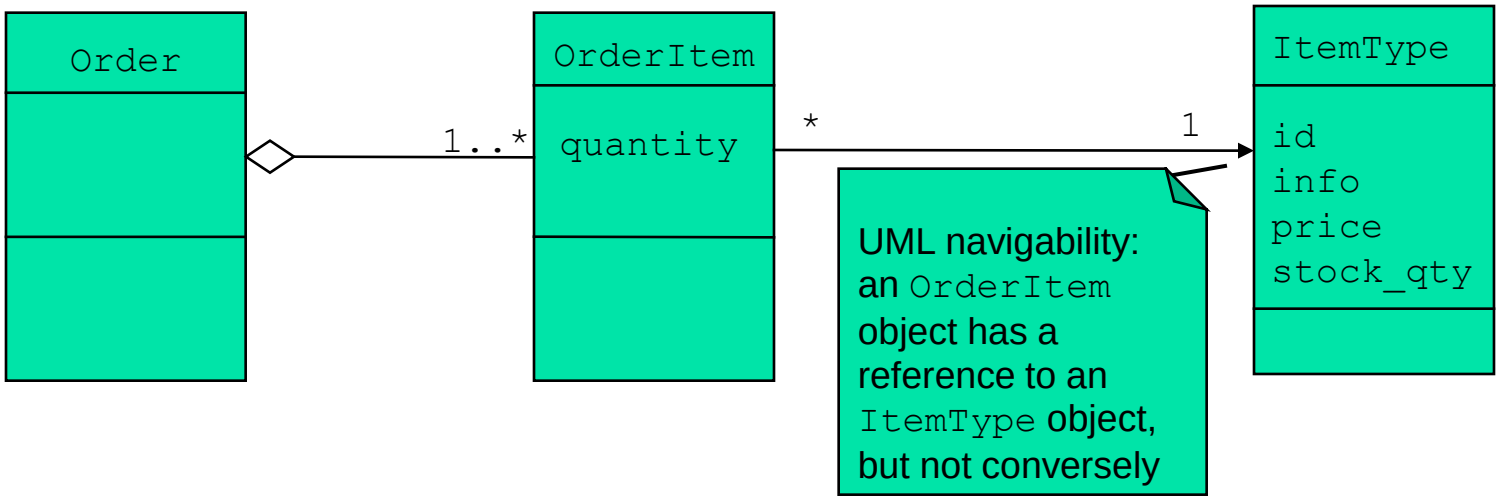
- `Item` **is split into**
  - `ItemType`
  - `OrderItem`



- **This is an example of *refactoring*:**
  - An increment improvement in a design or program
  - This refactoring is closely related to “database normalization”

UML Aggregation

- Aggregation models the object relation “part of,” or “is composed of”
  - A “stronger” class relation than association (“has”)



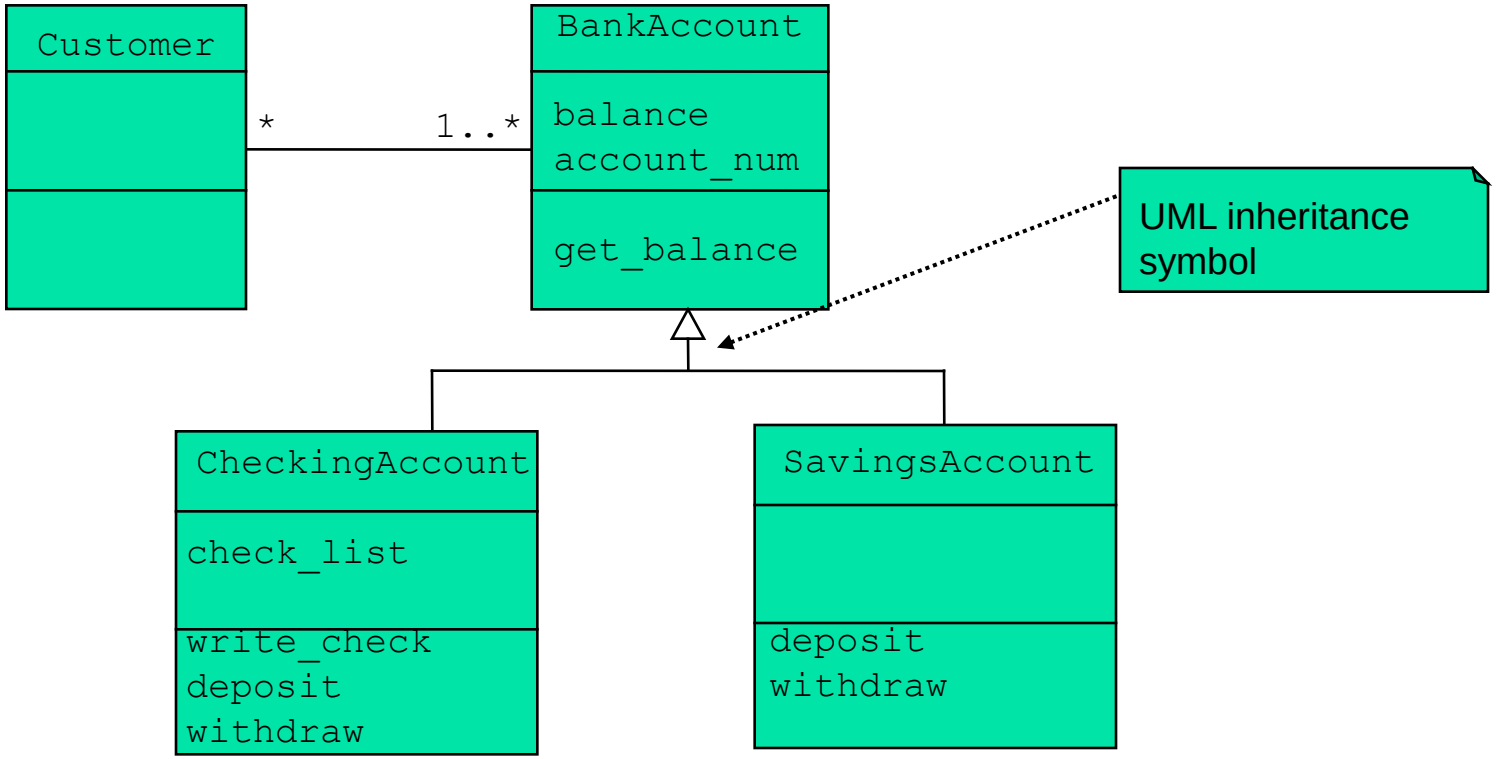
https://www.youtube.com/watch?v=X9yvOl4Ywew

UML aggregation (cont.)

- An `OrderItem` is part of an `Order`
  - An Order owns its `OrderItems` —`OrderItems` are not shared
  - The only way to access an `OrderItem` is through their owner `Order`
  - The `Order` creates and destroys its `OrderItems`
- Aggregation is controversial—different experts define it in different ways
  - What if not all of the above apply? Is it still aggregation?
  - Our recommendation: “When in doubt, leave it out”

Inheritance in UML

- Inheritance models the object relation “is a kind of”
  - A SavingsAccount is a kind of BankAccount
  - Thus any attribute or method of a BankAccount also applies to a SavingsAccount



Inheritance in C++

Money would be typedef'ed in a header file as a decimal type

```
// File: bankaccount.h
class BankAccount {
private:
   Money balance;
   int account_num;
public:
   Money get_balance(){
      return balance;
   }
   // other members,
   // e.g., constructor
};
```

public modifier needed to allow users of the ***derived*** class to access members of the ***base*** class

#include of file containing the base class definition needed.
Double quotes are used as this is a user file, not a system file

```
#include "bankaccount.h"
...
class CheckingAccount : public BankAccount {
private:
   std::list<Check*> check_list;
public:
   void deposit(Money m){ ... }
   void withdraw(Money m){ ... }
   void write_check(Check* cp){ ... }
   // other members, e.g., constructor
};
```