Today – CS202 Lecture #1

cs.pdx.edu/~Karlaf

1. Welcome! What to expect.
2. Syllabus

3. Approach for

   Programming
4. Introduction to OOP
5. Syntax: Inheritance Preview

Programming Guidelines

1. Each program focusses on OOP
2. This means that each program will have multiple classes.
3. Each class needs to have a well defined purpose or "job"
4. Data Members must be private or protected
5. Consider waiting to integrate the data structure(s) until after the OO design is Known
6. NO Global variables
7. NO use of String class
8. All arrays MUST be dynamically allocated
9. All programs must use inheritance

Sample OO Design

1. Understand the problem from an application perspective.

2. List all nouns (although they won't all be classes

3. Group Nouns together thinking about the job of each resulting class

4. Remember, with OOP classes work together to solve a problem!

Banking Application

Products: Savings, Checking, CD's, Money Market credit cards, Loans (equity, mortgages, car), Student Accounts
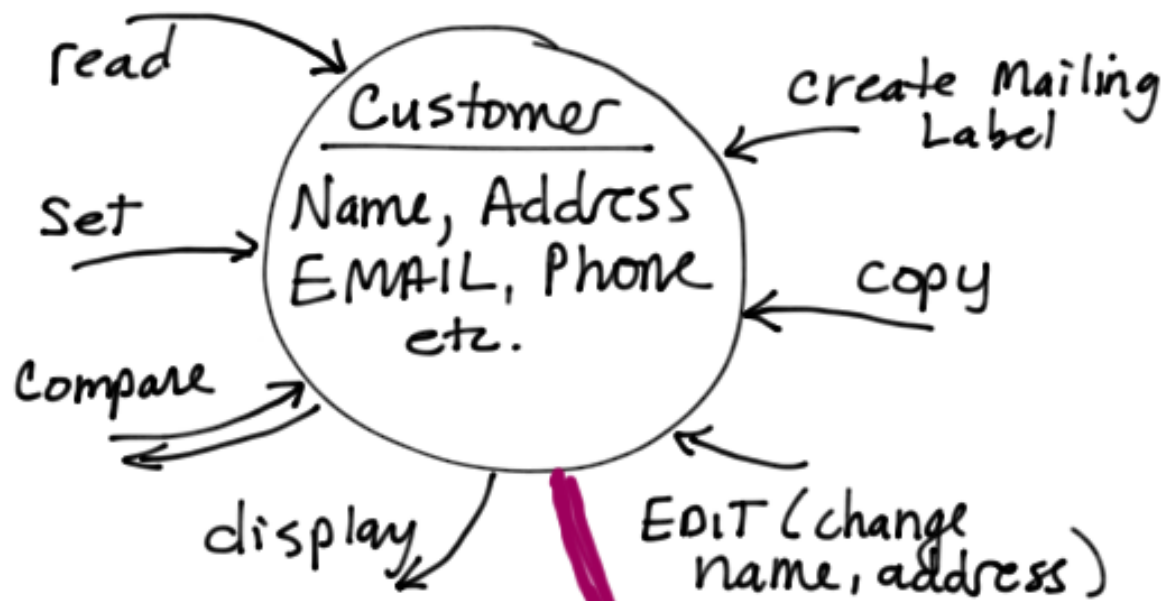
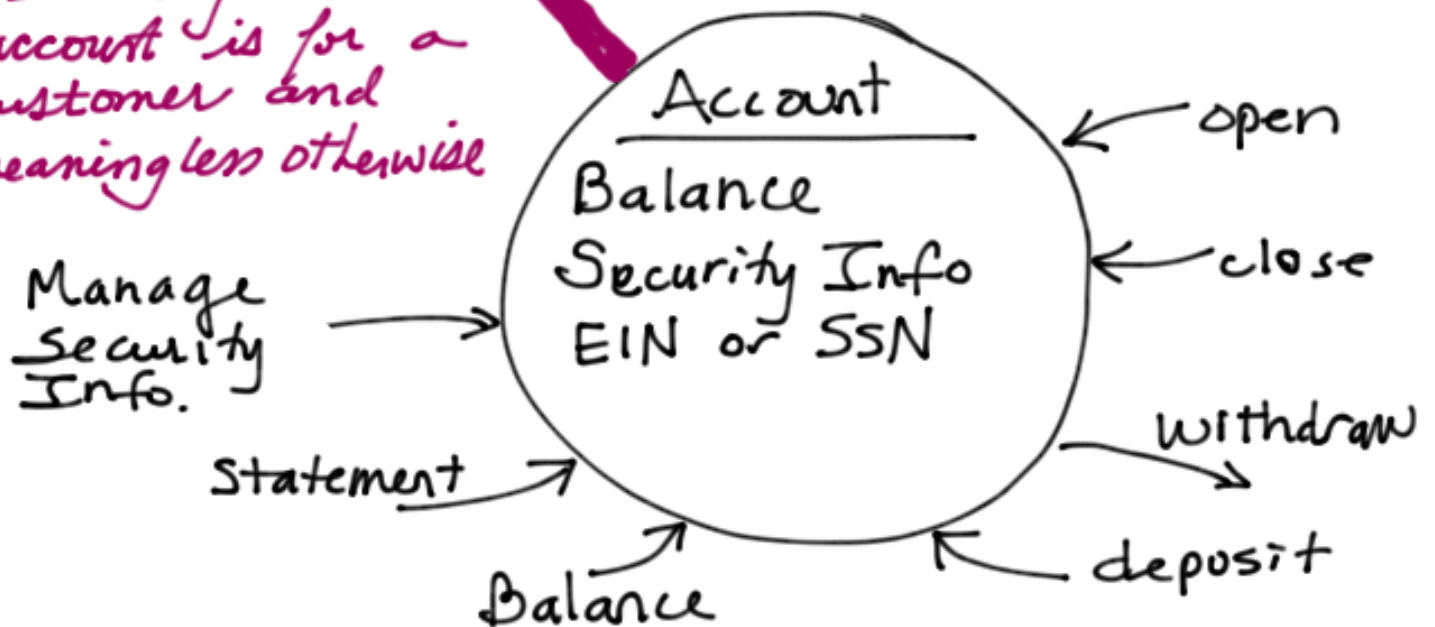**Group Nouns:** Account, Balance, customer, Transactions or history Fees, interest, ...

Customer: Name, Address (Physical, Mailing), email Password, Password questions, Maiden Name credit score, application ...
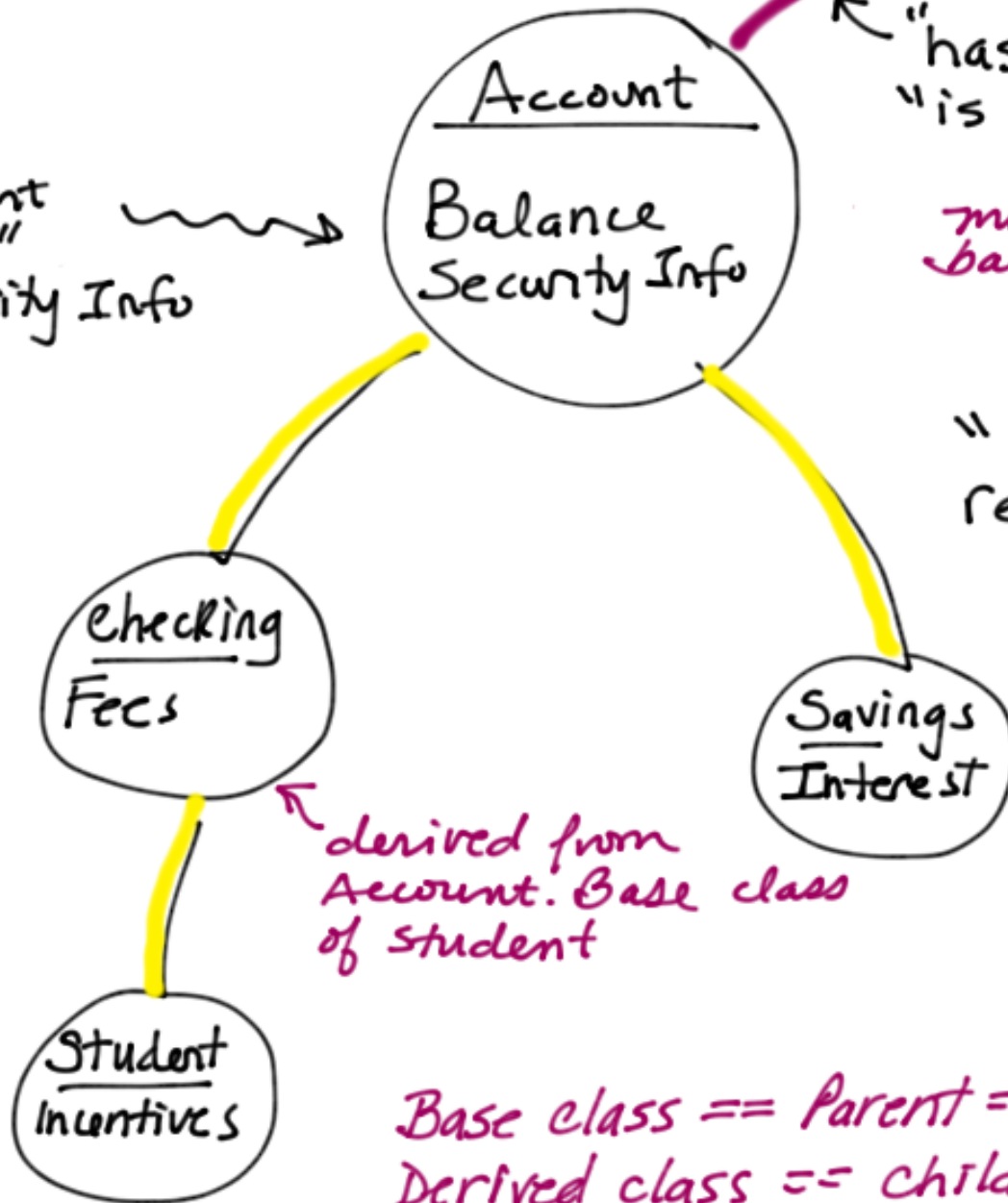
Types of Products

Customer INFO

Account
Balance
Security Info

"has a" or "is a"

most indirect base class

Account "has" Security Info

"is a" relationship

Checking
Fees

Savings
Interest

derived from Account. Base class of Student

Student
Incentives

Base class == Parent == Super class
Derived class == child == Subclass

Syntax for Single Inheritance Hierarchies

```
class Account
{ public:

    protected:          ←——————— Available for subclasses
                                  but not client programs
    private:
       float balance;
};

class Checking : public Account
{
                Derivation list
    public:
                                  "checking is an Account
    protected:                     object plus more"
       // Fees, interest
    private:

};
```
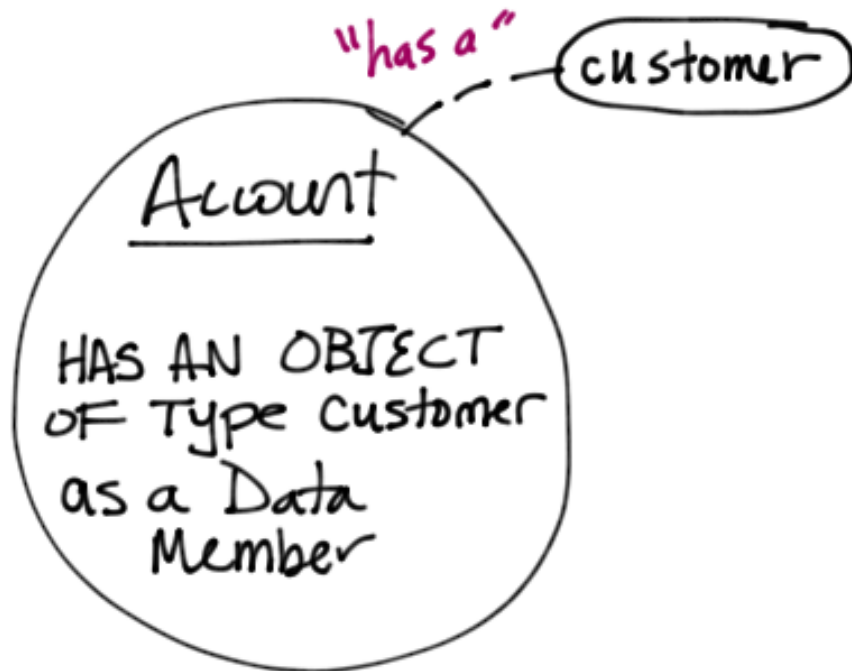
"has a" vs "is a"

"has a" —— (customer)

Account

HAS AN OBJECT
OF Type Customer
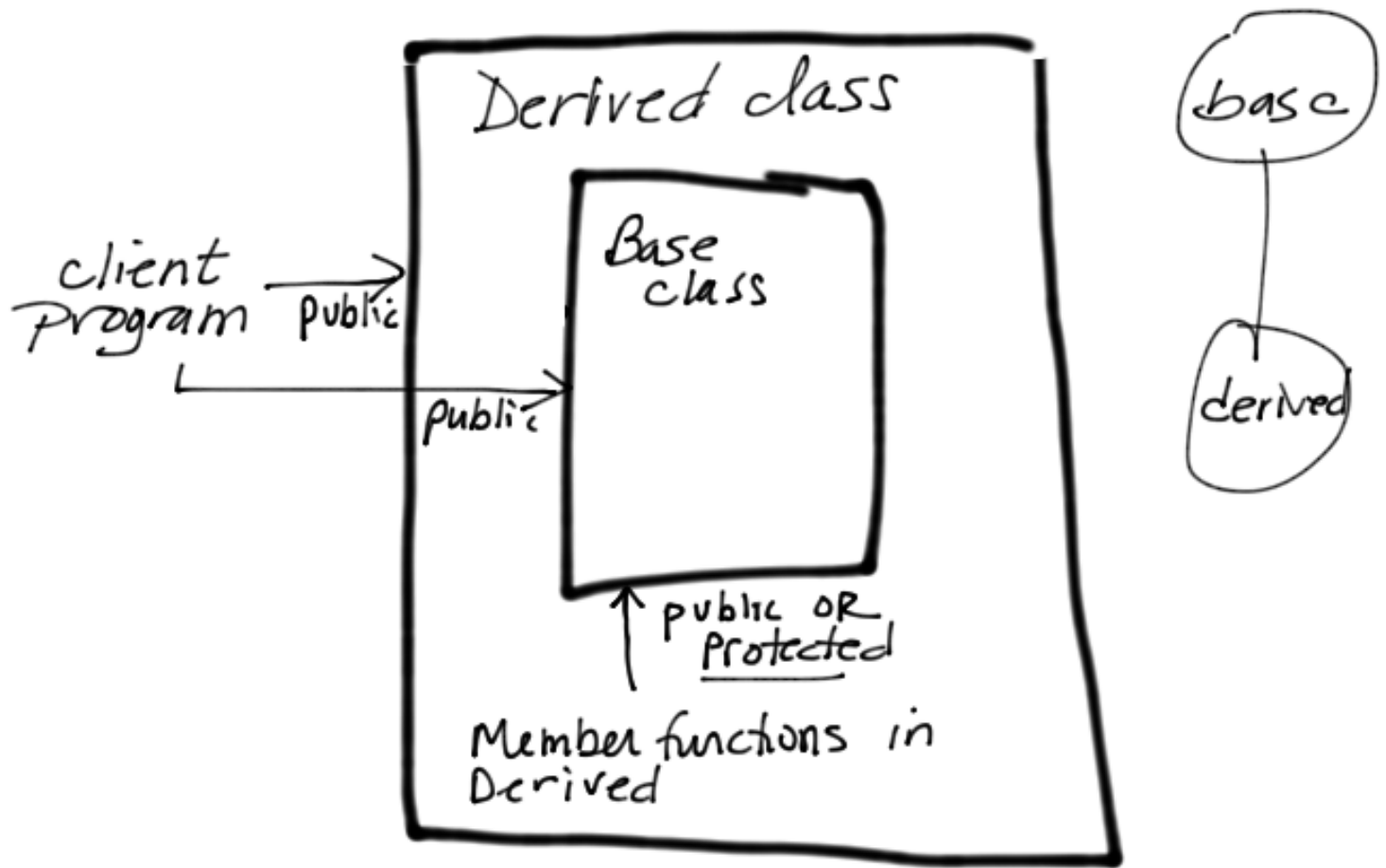as a Data
Member

object. open();
open would have to call
customer member functions
to work w/ name, email, etc
  datamember . input ();
    a customer class
    member function

customer

"is a"

Account
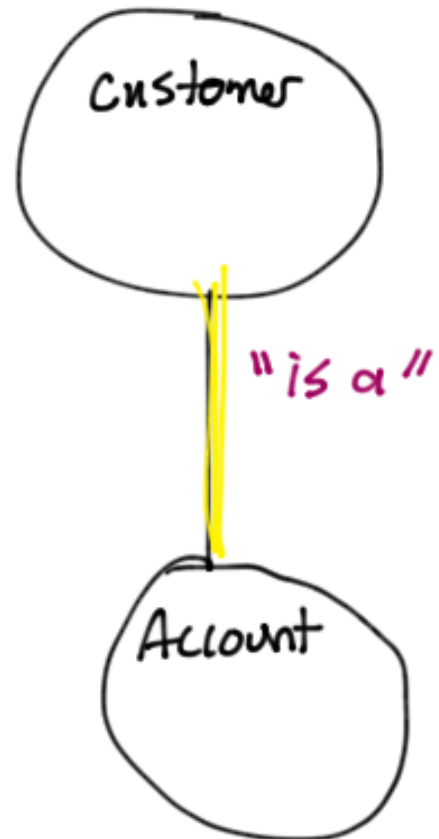
Now, the input
function can be
called directly:
  object. input();

Think of Inheritance:

Derived class

Base class

client program → public

public →

public OR protected

Member functions in Derived

base

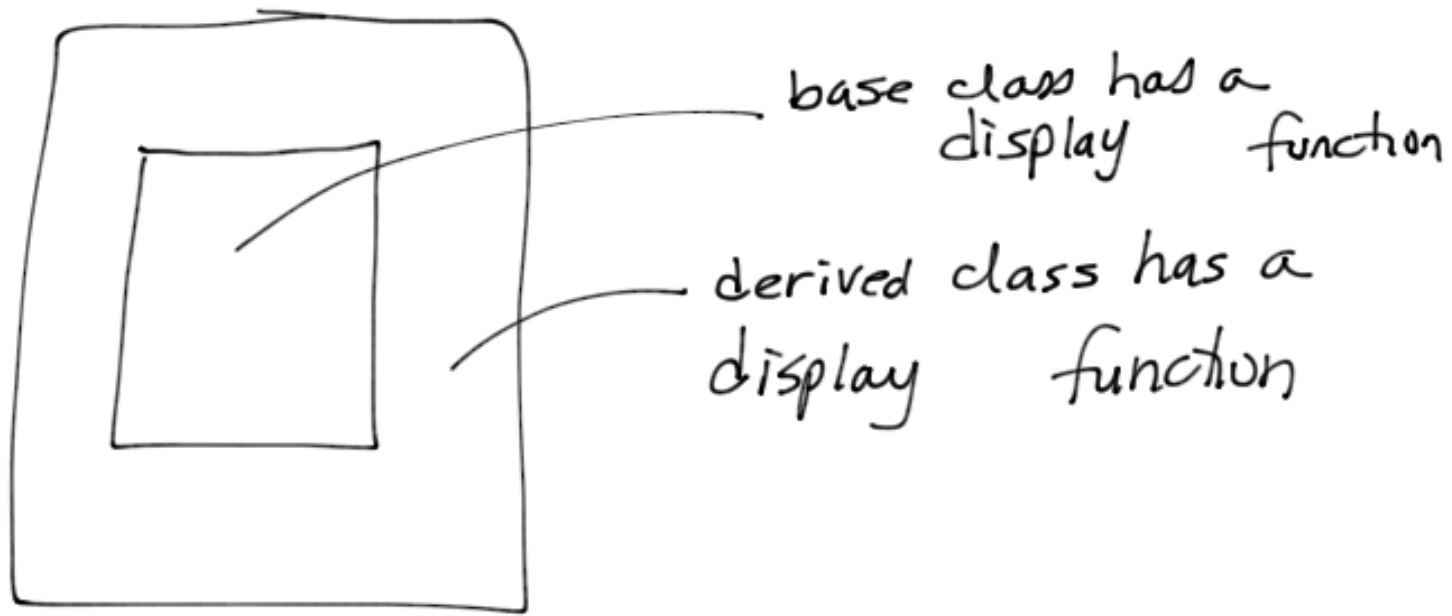derived

Derived object;

Side Note: a member function of a derived class can call a public or protected member function of the base class.

What if.... Same function name?

base class has a
display   function

derived class has a
display   function

Regardless of the argument lists,
Function Overloading does __not__
take place.

The client can call the parent's
public member function via:
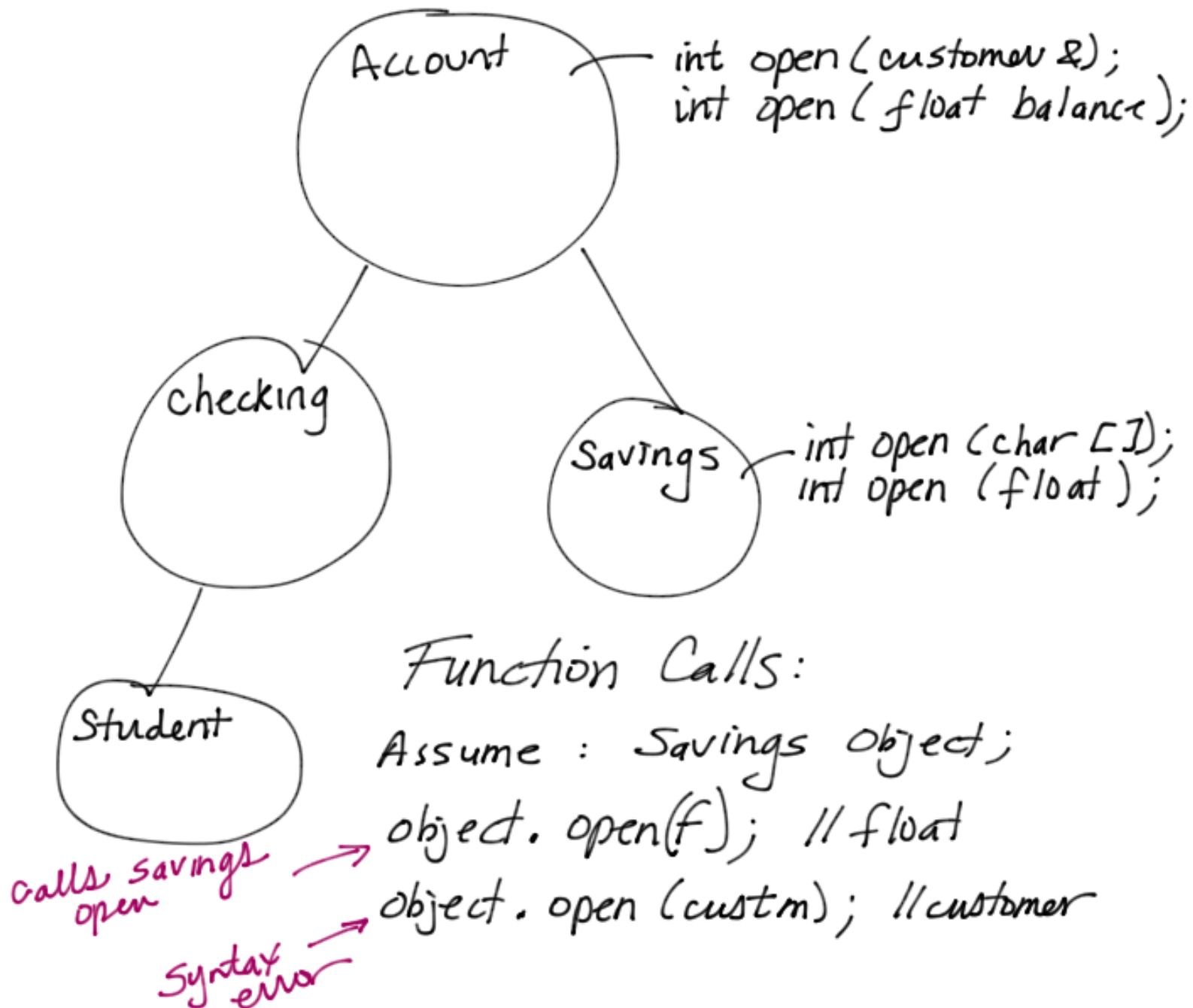
base_class name :: display (args);

↑
Scope resolution operator

derived object;
object. base_class name :: display (args);

Function Overloading...

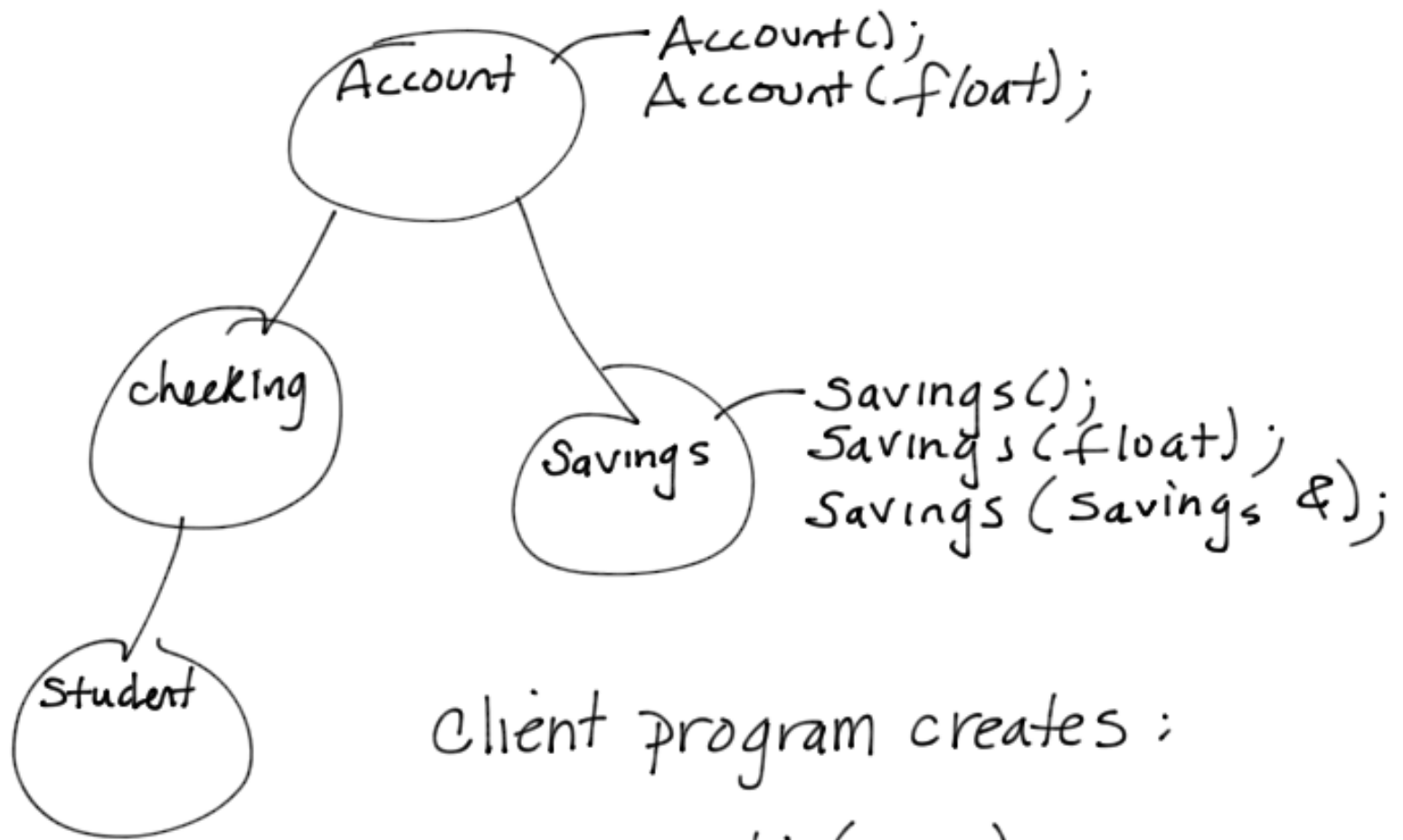only occurs <u>within</u> a given scope $\{ \}$ ← block

Account — int open (customer &);
int open (float balance);

checking

Savings — int open (char []);
int open (float);

Student

Function Calls:

Assume : Savings object;

object. open(f);   // float   ← calls savings open

object. open (custm);   // customer   ← Syntax error

* The derived class functions HIDE the base class
functions of the same name.

Constructors with Arguments

Account
- Account();
- Account(float);

checking

Savings
- Savings();
- Savings(float);
- Savings(Savings &);

Student

Client program creates:

Savings obj(100.0);

1. First the default constructor for the Account class is implicitly called.

2. Then, the Savings constructor with the float argument is called

So, the information is not passed up to the parent........

Initialization Lists

In the implementation of the constructors
we can add initialization lists.

They can be used to kick start the parents'
constructor when arguments are involved

```
Savings :: savings (float val) : Account(val) , fee(0)
{

    ... // body of the function
}
```

initialization list

data member
(Savings)

now the default constructor will not be
involked when an object is created with
a float passed as an argument

What about 1 vs. Many

1. Inheritance is _not_ designed to take 1 instance and turn it into many.

So, if you had a "list", it should _never_ be derived from (a) a node, or (b) an individual item

"has many"

"is a"

Generic list

Item

specialized list

list

No!
≠
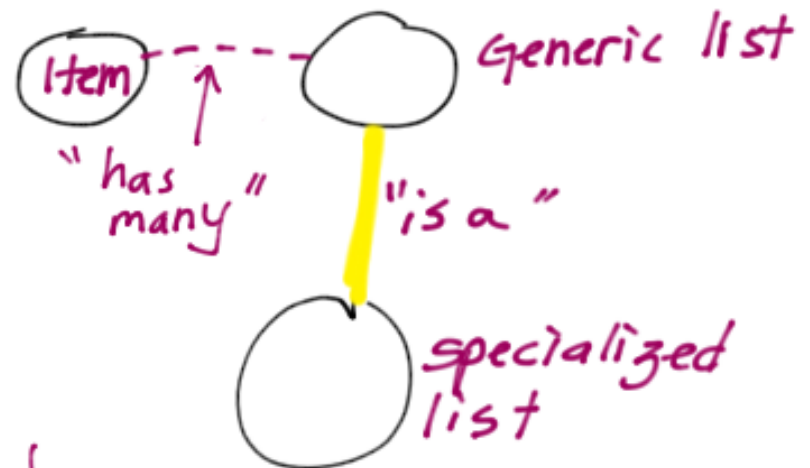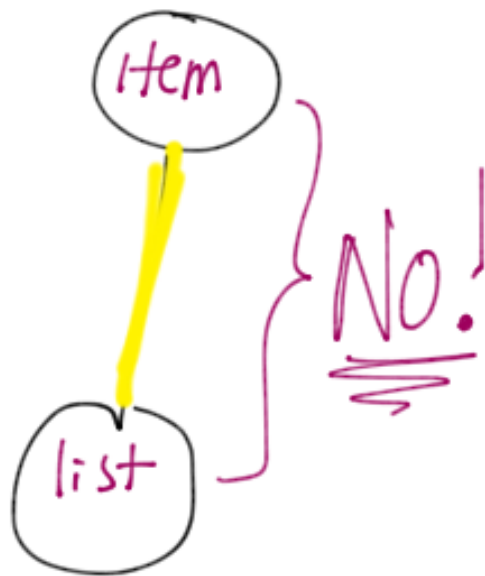
It is common to have _both_ "has a" and "is a" relationships creating the solution.
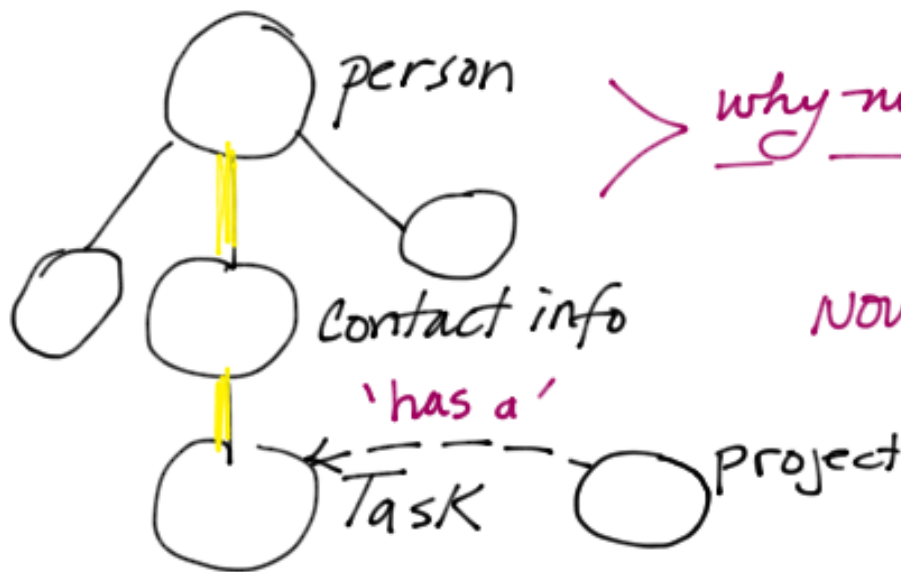
Example: To Do List

1. understand the problem
2. List Nouns, and Group them

Task
person
project
due date
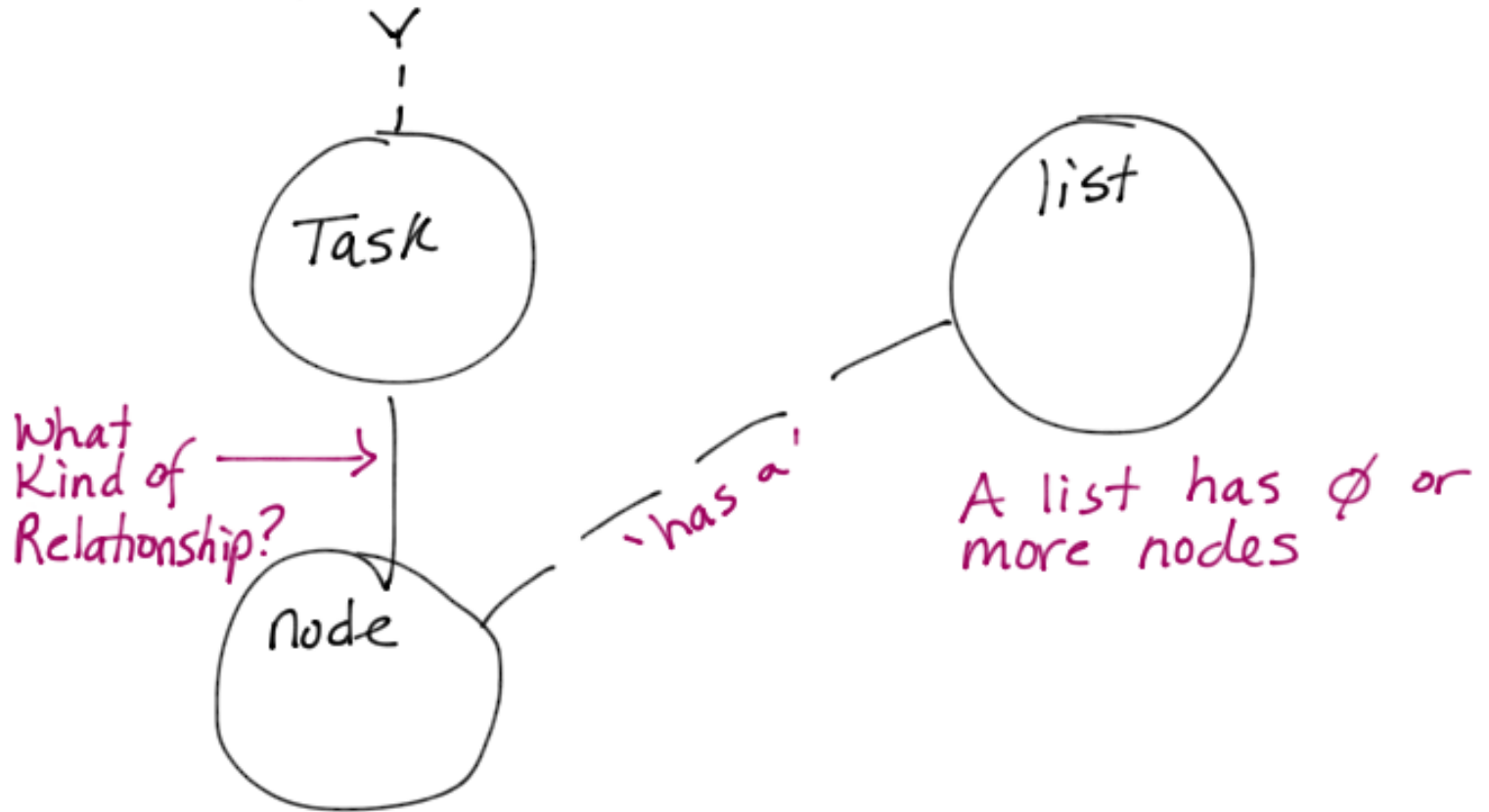priority

Person
name (first, last)
contact information

Contact Information
email address
phone #

person

> why not opposite?

Now, let's create the many

'has a'

Task project

Contact info

One Approach....

Task

list

What Kind of Relationship?

'has a'

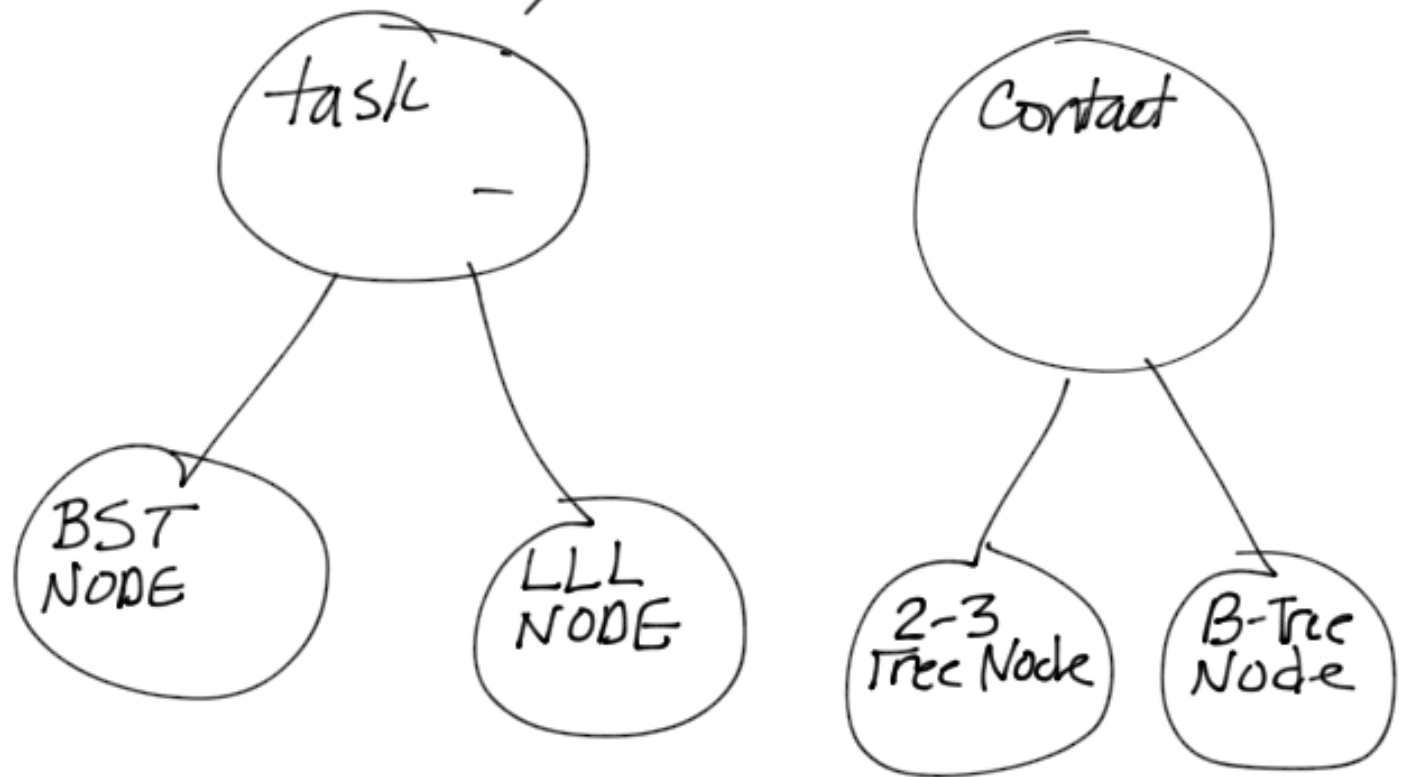A list has $\emptyset$ or more nodes

node

if 'has a" (Task is a data member in a node), what happens if node is a class? It would need "wrapper" functions to pass information from the list class to the task.

if 'is a' (node is derived from task), the list could directly call task public functions

head → displaytask();

Opportunities:

```
task
  |
  +-- BST NODE
  +-- LLL NODE

Contact
  |
  +-- 2-3 Tree Node
  +-- B-Tree Node
```

```cpp
class LLLNode : public Task
{

    :

};
```