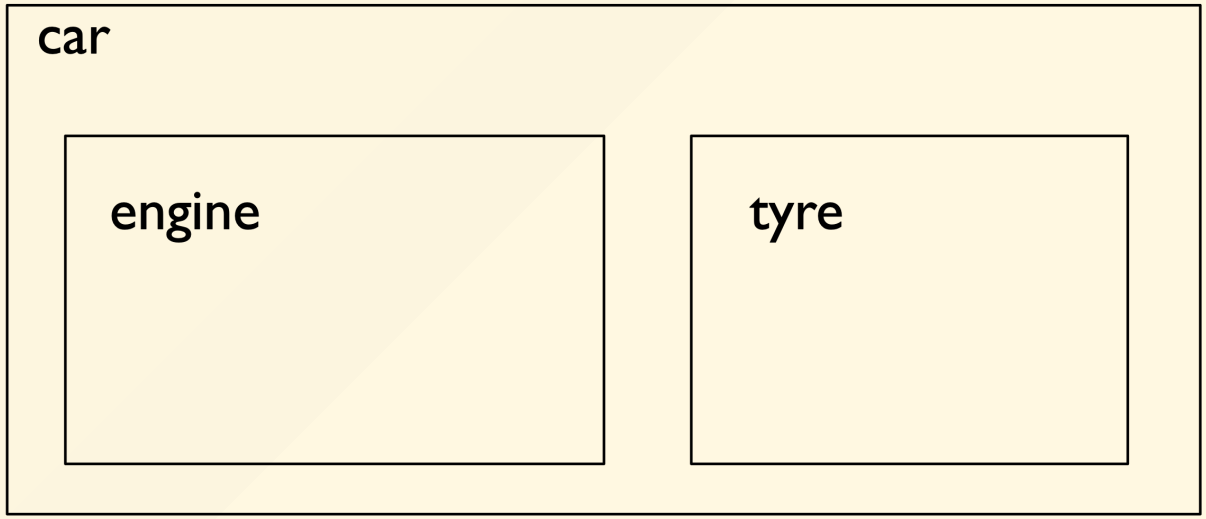
Composition & Inheritance

Object-Oriented Programming with C++

Reusing the implementation

- *Composition*: construct new object with existing objects
- The relationship of *has-a*



Composition

- Ways of inclusion
  - Fully, direct
  - By reference, allows sharing

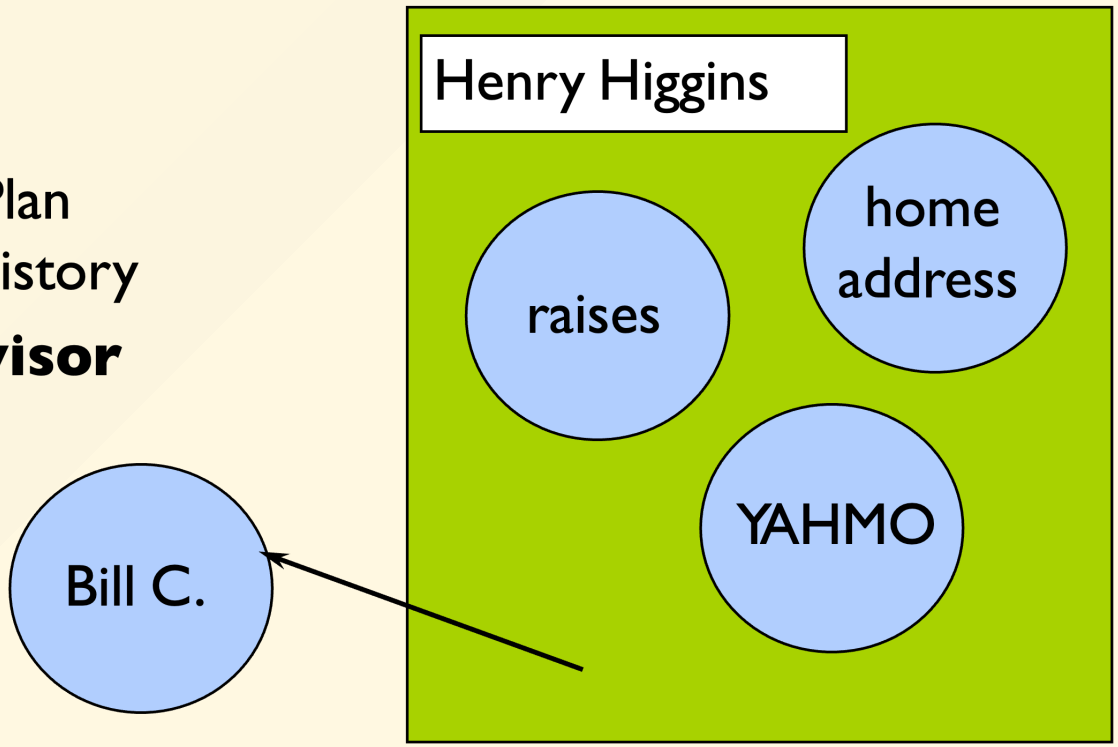
Composition in action

Classes

Employee

- Name
- Address
- Health Plan
- Salary History
- **Supervisor**

Instances



Example

```
class Person { … };
class Currency { … };
class SavingsAccount {
public:
  SavingsAccount(
    const string& name,
    const string& address,
    int cents);
  ~SavingsAccount();
  void print();
private:
  Person m_saver;
  Currency m_balance;
};
```

Example...

```
SavingsAccount::SavingsAccount(
 const string& name, const string& address, int cents)
 : m_saver(name, address),
   m_balance(0, cents)
{}

void SavingsAccount::print()
{
  // how to implement this function?
}
```

Example...

```
SavingsAccount::SavingsAccount(
 const string& name, const string& address, int cents)
 : m_saver(name, address),
   m_balance(0, cents)
{}

void SavingsAccount::print()
{
  m_saver.print();
  m_balance.print();
}
```

Embedded objects

- Must be initialized
  - The default constructor is called if you don't supply the arguments
- Initializer list on Constructor
  - any number of objects separated by commas
  - provide arguments to sub-constructors
- Syntax:

```
name( args ) [':' init-list] '{'
```

A nonobvious problem

- If we implement the constructor as below, then the *default constructors* of the sub-objects would be called.

```
SavingsAccount::SavingsAccount(
 const string& name, const string& address, int cents) {
  m_saver.set_name( name );
  m_saver.set_address( address );
  m_balance.set_cents( cents );
}
```

Public vs. Private

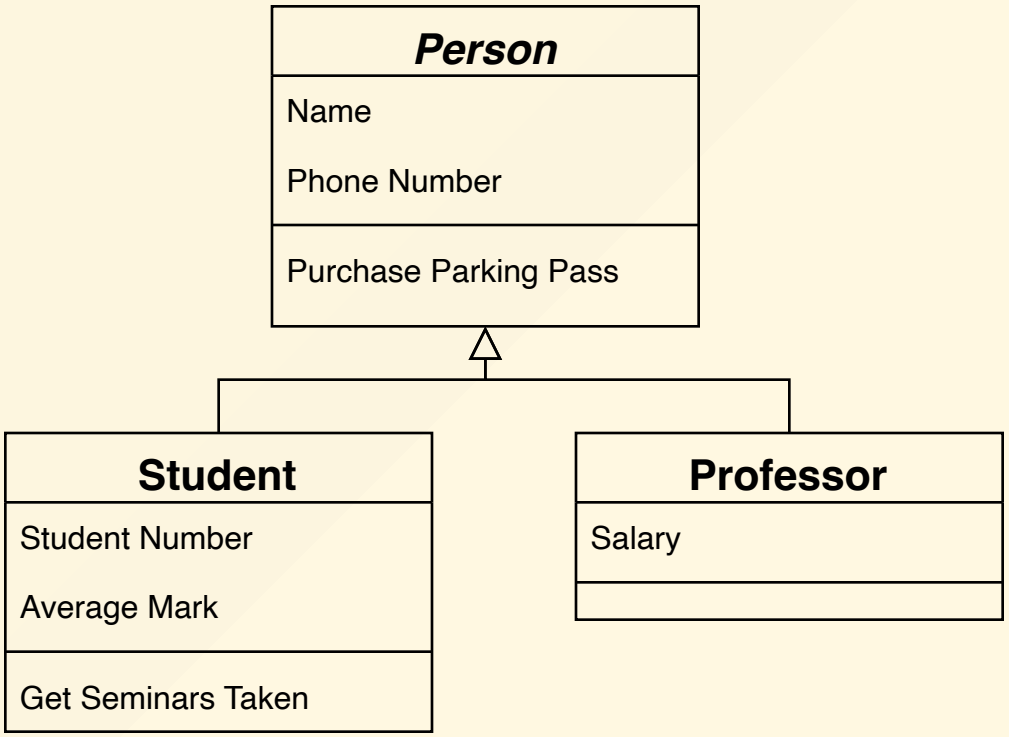
- Usually, we make the embedded objects *private*.
  - as part of the underlying implementation
- Otherwise, enable the entire sub-object interface:

```
class SavingsAccount {
public:
  Person m_saver;
  ...
}; // assume Person class has set_name()
SavingsAccount account;
account.m_saver.set_name("Fred");
```

Inheritance

Reusing the interface

- *Inheritance*: clone an existing class and extend it.
- The relationship of *is-a*

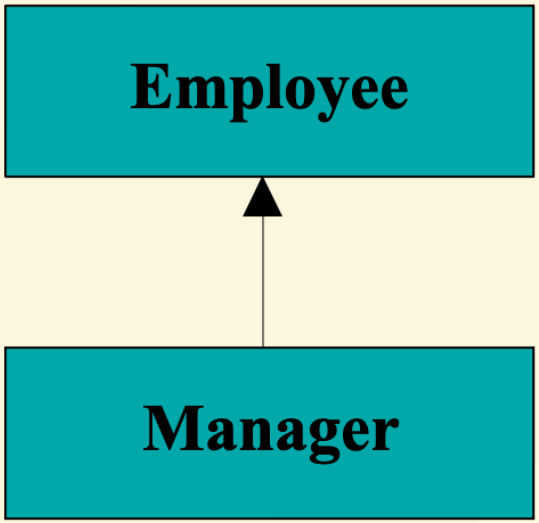


Inheritance

- An important component of the OO design methodology.
- Allows sharing of design for
  - member data
  - member functions
  - interfaces

Inheritance

- Terminology



Base Class
Super
Parent

Derived Class
Sub
Child

Scopes and access in C++

specifiers	within same class	in derived class	outside the class
`private`	Yes	No	No
`protected`	Yes	Yes	No
`public`	Yes	Yes	Yes

Declare an Employee class

```
class Employee {
public:
  Employee(const string& name, const string& ssn);
  const string& get_name() const;
  void print(ostream& out) const;
  void print(ostream& out, const string& msg) const;
protected:
  string m_name;
  string m_ssn;
};
```

Constructor for Employee

```
Employee::Employee(const string& name, const string& ssn)
  : m_name(name), m_ssn(ssn)
{
  // initializer list sets up the values!
}
```

Employee member functions

```
const string& Employee::get_name() const
{
  return m_name;
}
void Employee::print(ostream& out) const
{
  out << m_name << endl;
  out << m_ssn << endl;
}
void Employee::print(ostream& out, const string& msg) const
{
  out << msg << endl;
  print(out);
}
```

Now add Manager

```
class Manager : public Employee {
public:
  Manager(const string& name,
          const string& ssn,
          const string& title);
  const string title_name() const;
  const string& get_title() const;
  void print(ostream& out) const;
private:
  string m_title;
};
```

Inheritance and constructors

- Think of inherited traits as an embedded object
- Base class is mentioned by its class name

```
Manager::Manager(const string& name,
                 const string& ssn,
                 const string& title = "")
  : Employee(name, ssn), m_title(title)
{
}
```

More on constructors

- Base class is always constructed first.
- If no explicit arguments are passed to base class, the default constructor will be called.
- Destructors are called in exactly the reverse order of the constructors.

Manager member functions

```
void Manager::print(ostream& out) const
{
  Employee::print(out); // call the base class print
  out << m_title << endl;
}
const string& Manager::get_title() const
{
  return m_title;
}
const string Manager::title_name() const
{
  return string(m_title + ": " + m_name);
  // access base m_name
}
```

Uses

```
int main () {
  Employee bob( "Bob Jones", "555-44-0000" );
  Manager bill( "Bill Smith", "666-55-1234",
    "ImportantPerson" );
  // okay Manager inherits Employee
  string name = bill.get_name();
  // Error -- bob is an Employee!
  string title = bob.get_title();
  cout << bill.title_name() << '\n' << endl;
  bob.print(cout);
  bob.print(cout, "Employee:");
  bill.print(cout);
  bill.print(cout, "Employee:"); // Error -- hidden!
}
```

Name hiding

- If you *redefine* a member function in the derived class, all the other overloaded functions in the base class are inaccessible.
- We'll see how the keyword *virtual* affects function overloading next time.

Access protection

- Members
  - *public*: visible to all clients
  - *protected*: visible to classes derived from self (and to friends)
  - *private*: visible only to self and to friends!

Friends

- To explicitly grant access to a function that isn't a member of the structure.
- The class itself controls which code has access to its members.
- Can declare a global function, a member function of another class, or even an entire class, as a `friend`.
  - Example: Friend.cpp

class vs. struct

- `class` defaults to `private`
- `struct` defaults to `public`

Access protection

- Inheritance

```
class Derived1 : public Base {}
class Derived2 : protected Base {}
class Derived3 : private Base {}
```

How inheritance affects access

inheritance type (B is)	*public* members	*protected* members	*private* members
`: private A`	private in B	private in B	not accessible
`: protected A`	protected in B	protected in B	not accessible
`: public A`	public in B	protected in B	not accessible

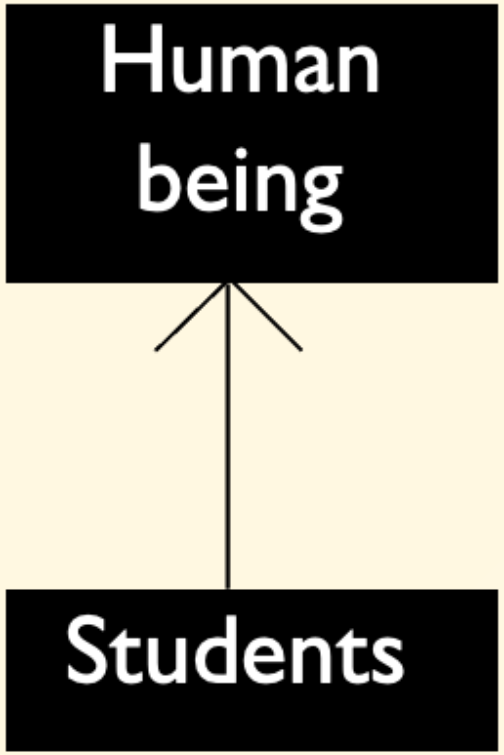
- Suppose class B is derived from class A.

Conversions

- Public Inheritance should imply substitution:
  - If B is-a A, you can use a B anywhere an A can be used.
  - if B is-a A, then everything that is true for A is also true of B.
  - Liskov's Substitution Principle

Upcasting

- Regard an object of the derived class as an object of the base class
  - only valid on reference or pointer.
- Students are human beings. You are students. So you are human being.



Upcasting examples

```
Manager pete("Pete", "444-55-6666", "Bakery");
Employee * ep = &pete; // Upcast
Employee & er = pete;  // Upcast
```

- Lose type information about the object:

```
ep->print(cout); // base class version of print
```

Conversions

D is derived from B
D => B
D* => B*
D& => B&