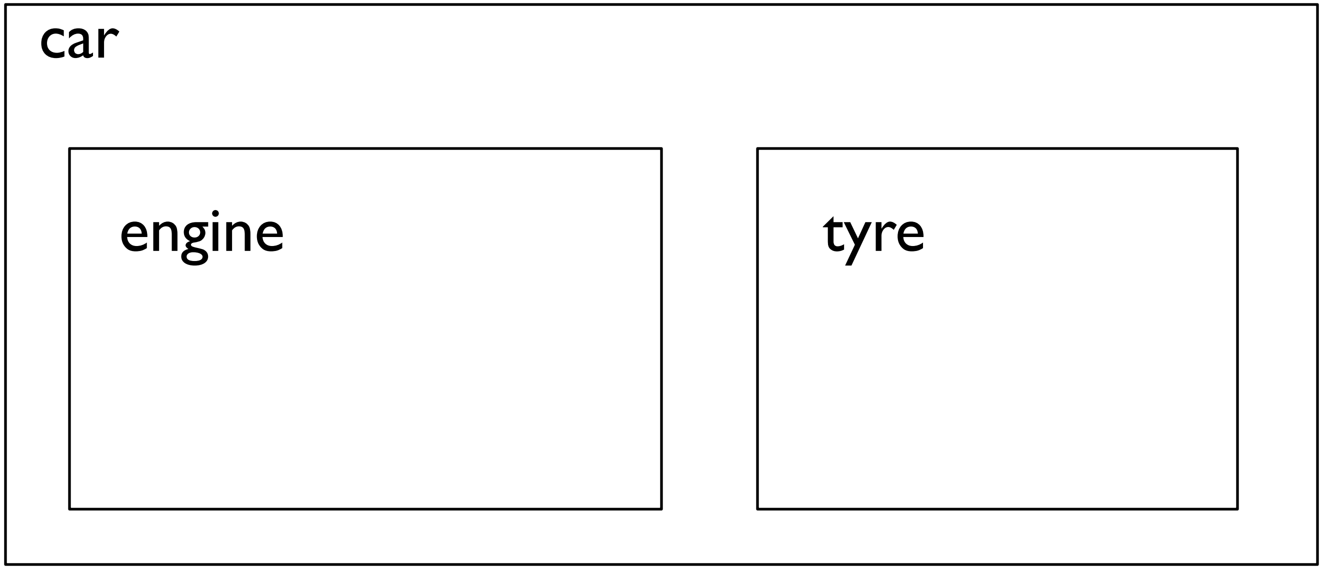
Composition & Inheritance

Object-Oriented Programming with C++

Reusing the implementation

- Composition: construct new object with existing objects
- It is the relationship of “has-a”



Composition

- Objects can be used to build up other objects
- Ways of inclusion
  - Fully
  - By reference
- Inclusion by reference allows sharing

- For example, an Employee has a
  - Name
  - Address
  - Health Plan
  - Salary History
    - Collection of Raise objects
  - Supervisor
    - Another Employee object!

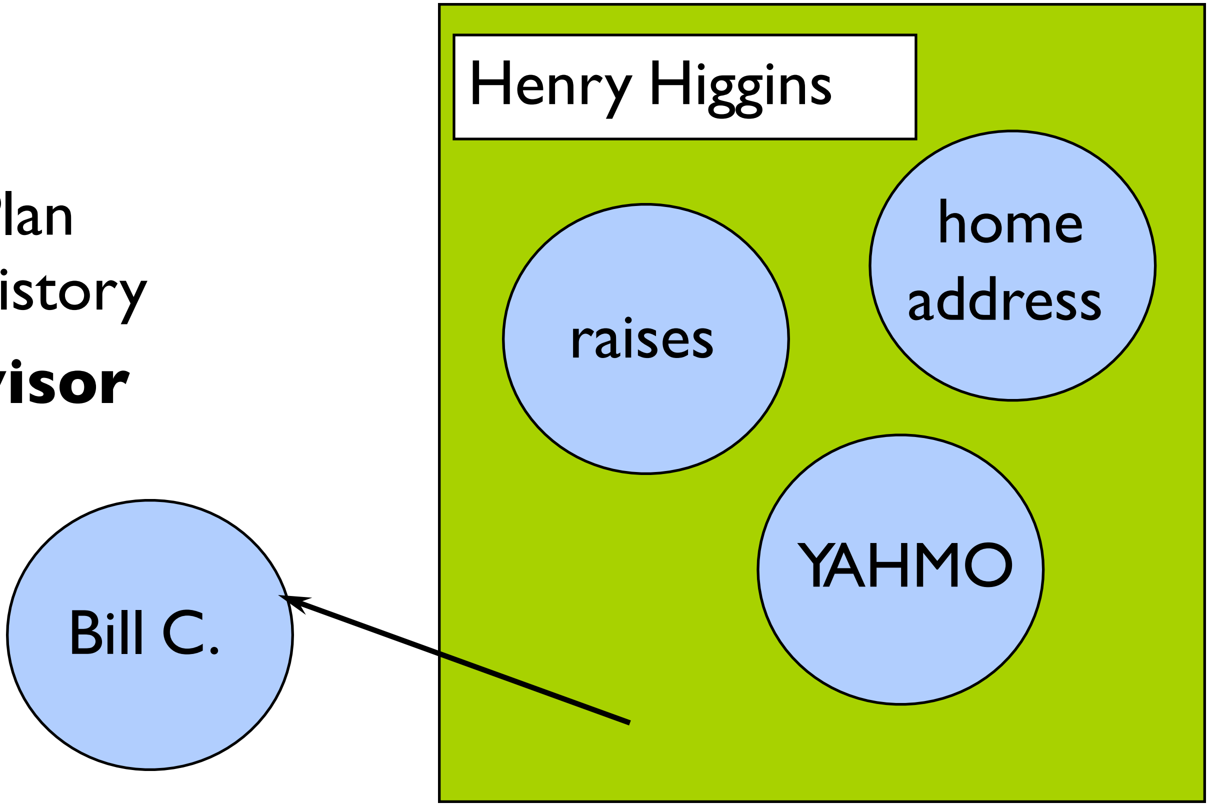
Composition in action

Classes

Employee

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

Instances



Example

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

Example...

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

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

Embedded objects

- All embedded objects are initialized
  - The default constructor is called if
    - you don’t supply the arguments, and there is a default constructor (or one can be built)
- Constructors can have initializer list
  - any number of objects separated by commas
  - is optional
  - Provide arguments to sub-constructors
- Syntax:

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

Question

- If we wrote the constructor as (assuming we have the set accessors for the sub-objects):

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

- Default constructors would be called

Public vs. Private

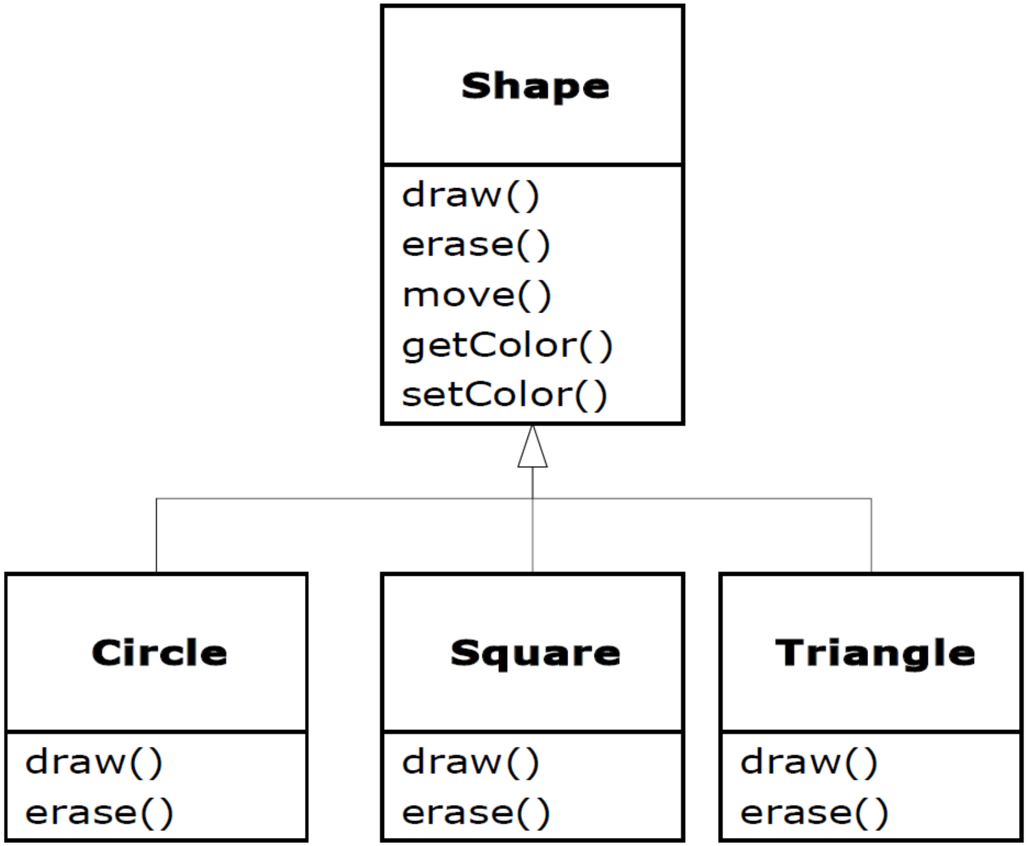
- It is common to make embedded objects private:
  - they are part of the underlying implementation
  - the new class only has part of the public interface of the old class
- Can embed as a public object if you want to have the entire public interface of the sub-object available in the new object:

```
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 is to take the existing class, clone it, and then make additions and modifications to the clone.

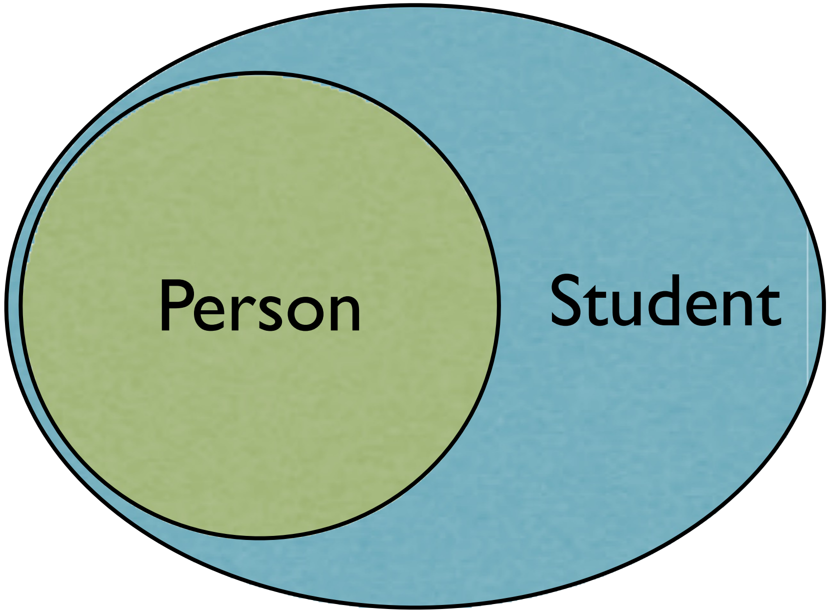


Inheritance

- Language implementation technique
- Also an important component of the OO design methodology
- Allows sharing of design for
  - Member data
  - Member functions
  - Interfaces
- Key technology in C++

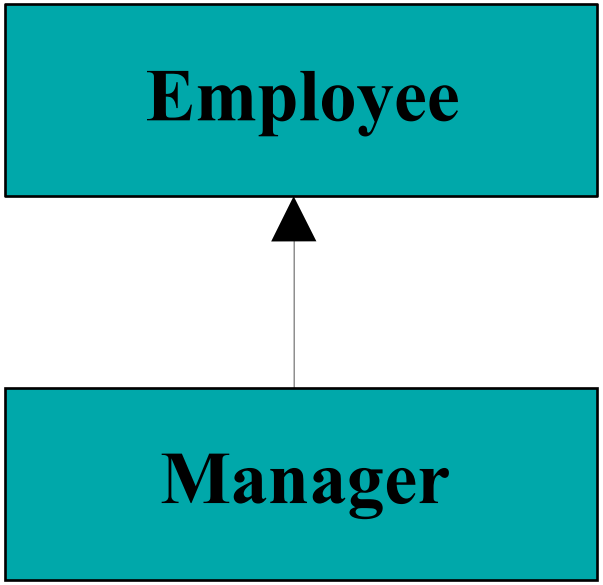
Inheritance

- The ability to define the behavior or implementation of one class as a ***derived one*** of another ***base*** class



Inheritance

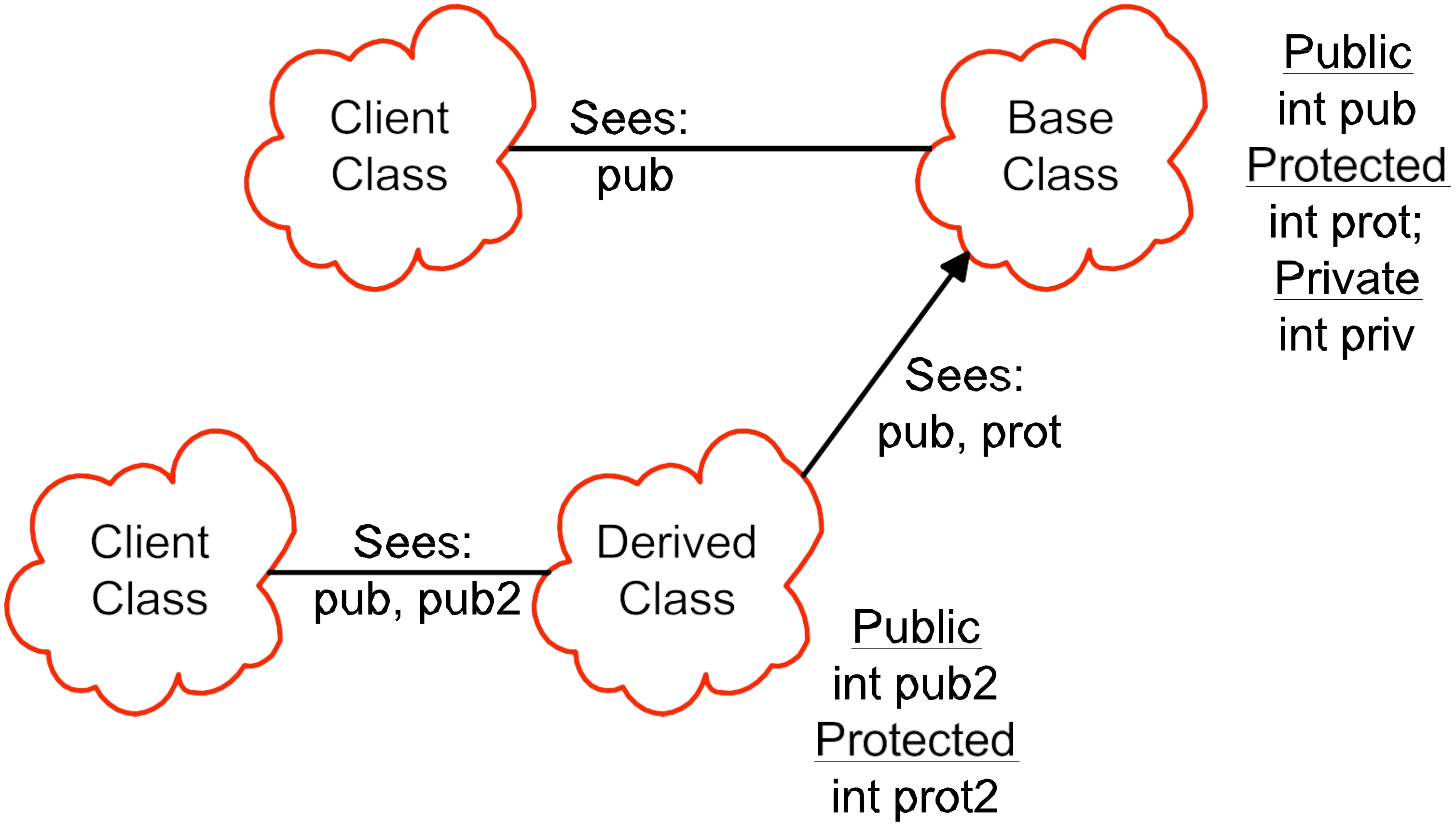
- **Class relationship: Is-A**



Base Class
Super
Parent

Derived Class
Sub
Child

Scopes and access in C++



Client
Class

Sees:
pub

Base
Class

Public
int pub
Protected
int prot;
Private
int priv

Sees:
pub, prot

Client
Class

Sees:
pub, pub2

Derived
Class

Public
int pub2
Protected
int prot2

Declare an Employee class

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

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

Now add Manager

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

Inheritance and constructors

- Think of inherited traits as an embedded object
- Base class is mentioned by 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
  - Default constructor will be called
- Destructors are called in exactly the reverse order of the constructors.

Manager member functions

```
inline void Manager::print( std::ostream& out ) const
{
      Employee::print( out ); //call the base class print
      out << m_title << endl;
}

inline const std::string& Manager::get_title() const
{
      return m_title;
}

inline const std::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" );

      string name = bill.get_name(); // okay Manager
      inherits Employee
      //string title = bob.get_title(); // Error --
      bob is an Employee!
      cout << bill.title_name() << '\n' << endl;
      bill.print(cout);
      bob.print(cout);
      bob.print(cout, "Employee:");
     //bill.print(cout, "Employee:"); // Error hidden!
}
```

Name Hiding

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

Access protection

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 as a ***friend***, as well as 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
  - Public: `class Derived :` **`public`** `Base ...`
  - Protected: `class Derived :` **`protected`** `Base ...`
  - Private: `class Derived :` **`private`** `Base ...`

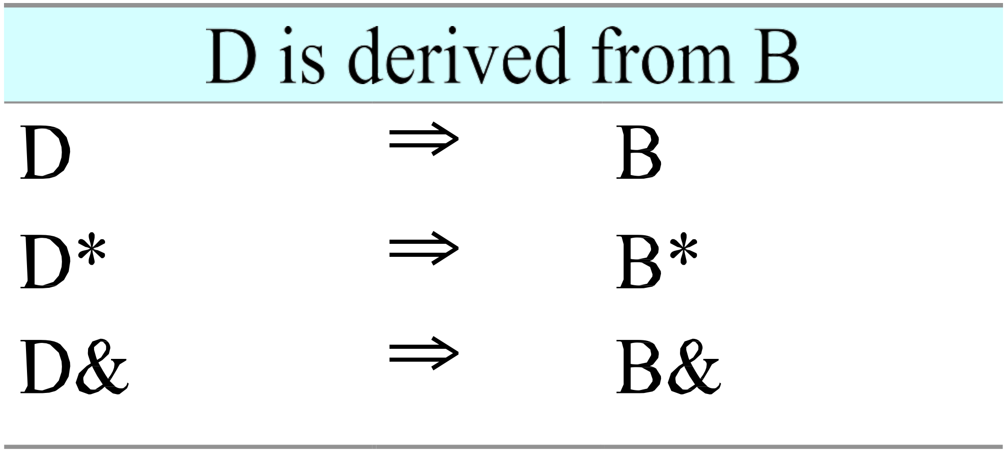
How inheritance affects access

Suppose class B is derived from A. Then:

	Base class member access specifier		
Inheritance Type (B is)	*public*	*protected*	*private*
`:public A`	public in B	protected in B	private
`:private A`	private in B	private in B	private
`:protected A`	protected in B	protected in B	private

Conversions

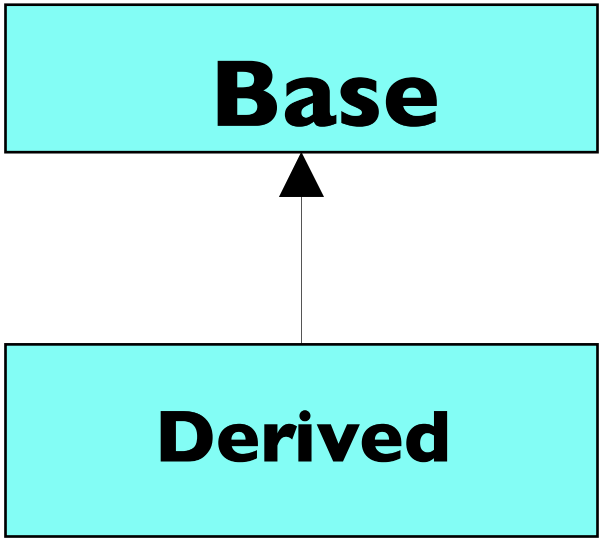
- Public Inheritance should imply substitution
  - If B *is-a* A, you can use a B anywhere an A can be used.
    - if B *is-a* A, then everything that is true for A is also true of B.
  - Be careful if the substitution is not valid!



D is derived from B		
D	⇒	B
D*	⇒	B*
D&	⇒	B&

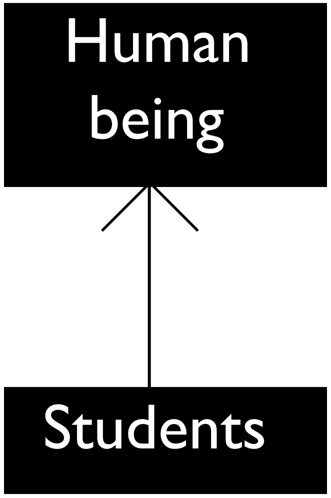
Up-casting

- Upcasting is the act of converting from a derived reference or pointer to a base class reference or pointer.



Up-casting

- Is to regard an object of the derived class as an object of the base class.
- It is to say: Students are human beings. You are students. So you are human being.



Up-casting 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);  // prints base class version
```