X. OOP & ADTs: An Introduction to Inheritance
(Chap. 12)

A. Inheritance, OOD, and OOP (§12.1 & 12.2)

A major objective of OOP: writing reusable code (to avoid re-inventing the wheel).

Ways we have done this:

- Write functions
- Build classes
- Store classes and functions in separately compiled libraries
- Convert functions into function templates
- Convert classes into class templates.

Most distinctive way to achieve reusability in OOP:

- **Inheritance**: Derive a class from another class, reusing the work done in building one class to build another class that is just a variation.

Example: Suppose a problem requires stack operations not provided in our `Stack` class.

"Old-fashioned" approach: Add new member functions to this class that implement the needed operations.

Bad: Can easily mess up a tested, operational class, creating problems for other client programs

Object-oriented approach: *Derive a new class* (say, `DerivedStack`) from class `Stack`, which is called the *parent* class of `DerivedStack`.

Good:

— Derived class inherits all of the members of its parent class (including its operations) so need not reinvent the wheel

— Mistakes made in building `DerivedStack` will be local to it — original `Stack` class remains untainted and client programs are not affected

Object-oriented design (OOD) is to engineer one's software as follows:

1. Identify the objects in the problem;
2. Look for <u>*commonality*</u> in those objects;
3. Where there is commonality:
    - Define base classes containing that commonality; and
    - Derive classes that inherit the commonality from the base class.

These last two steps are the most difficult aspects of OOD.

Object-oriented programming (OOP): first used to describe the programming environment for **Smalltalk**, the earliest true object-oriented programming language

Three important properties of OOP languages :

- ***Encapsulation***
- ***Inheritance***
- ***Polymorphism***, with the related concept of ***dynamic*** or ***late binding***

<u>B. Derived Classes</u>

Problem: Create types to model various kinds of licenses .

<u>Critical question</u>: *What attributes do all licenses have in common*?

Then store these common attributes:in a general (base) class `License`:

```
class License
{
public:
   // Function members Display(), Read(), ...

private:        // we'll change this in a minute
   long myNumber;
   string myLastName,
          myFirstName;
   char myMiddleInitial;
   int myAge;
   Date myBirthDay;    // Date is a user-defined type
   ...
};
```

For the various kinds of licenses, we could include a data member of type `License` and then add new members:

```
class DriversLicense
{
public:
   ...
private:
   License common;
   int myVehicleType;
   string myRestrictionsCode;
   ...
};

class HuntingLicense
{
public:
   ...
private:
   License common;
   sring thePrey;
   Date seasonBegin,
        seasonEnd;;
      ...
};


class PetLicense
{
public:
   ...
private:
   License common;
   string myAnimalType;
   ...
};
```

Inclusion works, but is "clunky" and inefficient.

```
HuntingLicense h;

h.common.Display(cout);
```

Worse, it's bad design! It should be that a `DriversLincense` **is a** `License`, not a `DriversLincense` **has a** `License`.

Preferred approach: **inheritance**. Derive more specialized license classes from the base class `License`, and add new members to store and operate on their specialized attributes.

Problem:

Private class members cannot be accessed within derived classes.

C++ solution:

Members declared to be `protected:` can be accessed within a derived class, but they remain inaccessible to programs or non-derived classes that use the class (except for friend functions).

So change the private section in class `License` to a protected section:

```
class License
{
public:
   // Function members Display(), Read(), ...

protected:
   long myNumber;
   string myLastName,
          myFirstName;
   char myMiddleInitial;
   int myAge;
   Date myBirthDay;
   ...
};
```

Now we can derive classes for the more specialized licenses from `License`:

```
class DriversLicense : public License
{
public:
   ...
protected:
   int myVehicleType;
   string myRestrictionsCode;
      ...
};

class HuntingLicense : public License
{
public:
   ...
protected:
   sring thePrey;
   Date seasonBegin,
        seasonEnd;;
      ...
};
```

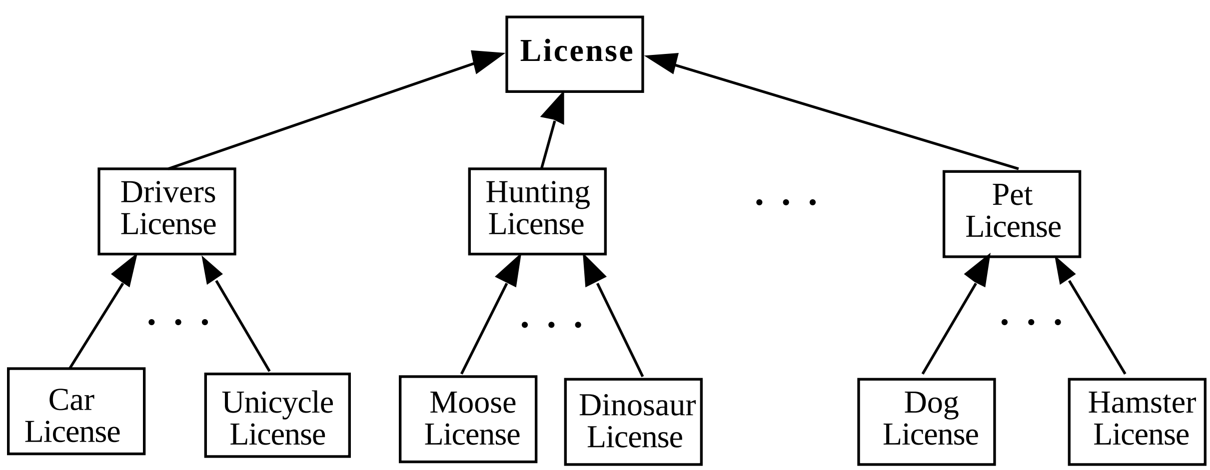
```
class PetLicense : public License
{
public:
   ...
private:
   string myAnimalType;
   ...
};
```

Classes like `DriversLicense`, `HuntingLicense`, and `BoatingLicense` are said to be **derived classes** (or **subclasses**), and the class `License` from which they are derived is called a **base class** or **parent class**

We have used protected sections rather than private ones in these drived classes in case it is necessary to derive "second-level" classes such as:

```
class MooseLicense : public HuntingLicense
{
public:
  ,,,
protected:
   int theAntlerMaximum;
   int theBullwinkleFactor;
   ...
};
```

This leads to **class hierarchies** — usually picture as a tree but with arrows is drawn from a derived class to its base class:



General form of declaration of a derived class:

```
DerivedClassName : kind_of_inheritance BaseClassName
{
   ...
  // new data members and functions for derived class
   ...
}
```

`kind_of_inheritance` is usually the keyword `public`, but it may be `private` or `protected`

The Fundamental Property of Derived Classes:

- Inherit the members of base class (and thus the members of all ancestor classes).
- Cannot access private members of base class
- Kind of access to public and protected members of base class depends on the kind of inheritance specified.

`public`	public and protected, respectively
`private`	private
`protected`	protected

Most common is public inheritance:

Can use public and protected members of base class in base class just as though they were declared within the derived class itself.

It gives rise to the **is-a relationship**:

If

```
class Base : public Derived
{
  // ... members of Beta ...
};
```

Then

A `Derived` object ***is a*** `Base` object.

For example: A `HuntingLicense` is a `License`
A `MooseLicense` is a `HuntingLicense`
A `MooseLicense` is a `License`

This is in contrast to the **has-a relationship** (also called the *inclusiong* or *containment* relationship or *class composition*). This was the situation with our first attempt at modeling licenses. Another example is the relationship between `License` and `Date`: A `License` object *has a* `Date` object, but it is not a `Date` oject.

Design Principle: Don't use public inheritance for the has-a relationship.

For example, it is bad design to do the following just to get the members of one class into another:

```
class BusDriver : public License
{ ... }
```

Rather, we should use:

```
class BusDriver
{
   ...
 private:
   License myLicense;
  ...
}
```

A third relatioship between classes is the **uses relationship**: One class might simply use another class. For example, a `Fee()` member function in a `LicensePlate` class might have a parameter of type `DriversLicense`. But this class simply *uses* the `DriversLicense` class — it *is not* a `DriversLicense` and it *does not have* a `DriversLicense`.

It's not always easy to tell which is the appopriate one to use. Two useful tests in deciding whether to derive Y from X:

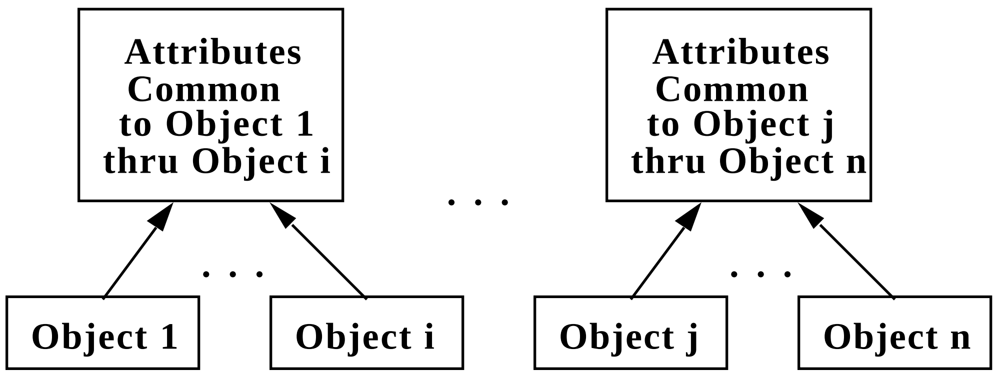
1. Do the operations in X behave properly in Y?

2. (The "need-a use-a" test): If all you need is a Y, can you use an X?

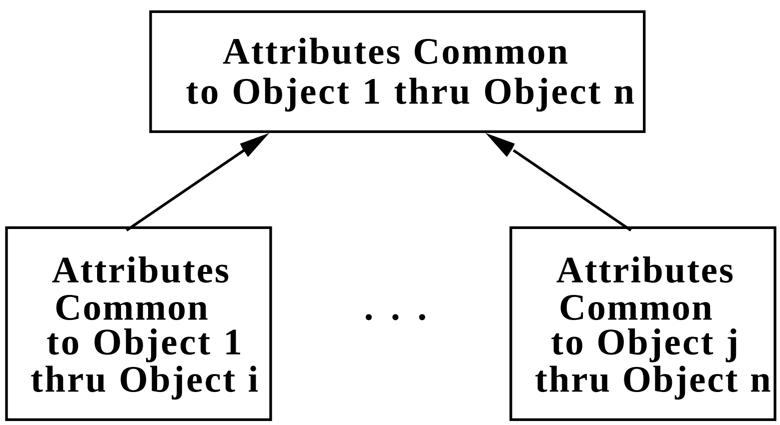
Summary:

The OOP approach to system design is to:

1. Carefully *analyze* the objects in a problem from the bottom up.
2. Where commonality exists between objects, group the common attributes into a base class:



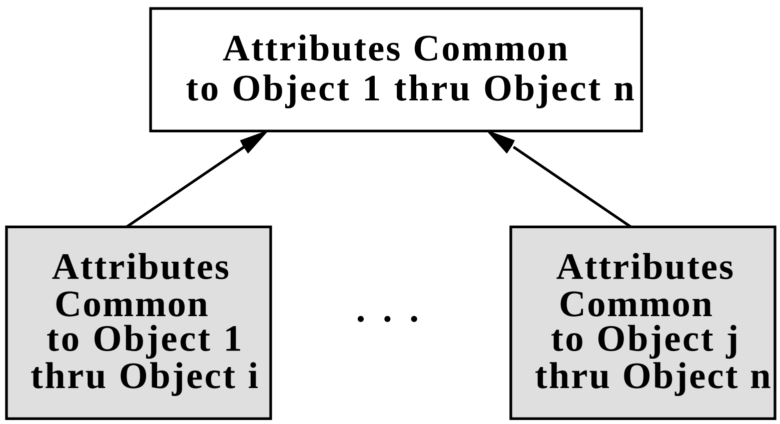
3. Then repeat this approach "upwards" as appropriate:



Once no more commonality exists, OO implementation then:

4. Proceeds from the top down, building the most general base class(es):

5. The less-general classes are then derived (publicly) from that base class(es):



6. Derivations continue until classes for the actual objects in the system are built:

7. These classes can then be used to construct the system’s objects.

C. Another Example:

Suppose we are told to write a payroll program.

Following the four OOD steps, we proceed as follows:

1. Identify the objects in the problem:

   • Salaried employees
   • Hourly employees

2. Look for commonality in those objects: what attributes do they share?

   • Id number
   • Name
   • Department
   • ...

3. Define a base class containing the common data members:

```
class Employee
{
protected:
   long myIdNum;              // Employee's id number
   string myLastName,         //     "      last name
          myFirstName;        //     "      first name
   char myMiddleInitial;      //     "      middle initial
   int myDeptCode;            //     "      department code

   //        ... other members common to all Employees

public:
   // ... various Employee operations ...
};
```

4. From the base class, derive classes containing special attributes:

   a. A salaried employee class:

```
   class SalariedEmployee : public Employee
   {
   public:
      // ... salaried employee operations ...

   protected:
      double mySalary;

   };
```

   b. An hourly employee class:

```
   class HourlyEmployee : public Employee
   {
   public:
      // ... hourly employee operations ...

   protected:
      double  myWeeklyWage,
              myHoursWorked,
              myOverTimeFactor;
   };
```

Reusability:

Suppose `Employee` has an output member function `Print()`:

```
void Employee::Print(ostream & out) const
{
  out << myIdNum << ' ' << myLastName << ", "  << myFirstName << ' '
      << myMiddleInitial << "  " << myDeptCode;
}
```

In derived classes, we can overload `Print()` with new definitions that reuse the `Print()` function of class `Employee`:

```
void SalariedEmployee::Print(ostream & out) const
{
  Employee::Print(out);                      //inherited member
  out << "\n$" << mySalary << endl;          //local member
}
```

and

```
void HourlyEmployee::Print(ostream & out) const
{
  Employee::Print(out);                      //inherited member
  out << "\n$" << myWeeklyWage << endl       //local members
      << myHoursWorked << endl << myOverTimeFactor << endl;
}
```

Note: A class `Deriv` derived from `Base` can call `Base::F()` to reuse the work of the member function `F()` from the base class.

Constructors and Inheritance:

Consider `Employee`'s constructor:

```
// Explicit-Value Constructor
inline Employee::Employee(long id, string last, string first,
                          char initial, int dept)
{
  myIdNum = id;
  myLastName = last;
  myFirstName = first;
  myMiddleInitial = initial;
  myDeptCode = dept;
}
```

A derived class can use a **member-initializer list** to call the base-class constructor to initialize the inherited data members — easier than writing it from scratch.

```
// Definition of SalariedEmployee explicit-value constructor

inline SalariedEmployee::SalariedEmployee(long id, string last, string first,
                                          char initial, int dept, double sal)
: Employee(id, last, first, initial, dept)
{
  mySalary = sal;
}
```

General form of Member-Initializer List Mechanism:

```
Derive::Derive(ParameterList) : Base(ArgList)
{
  // initialize the non-inherited members in the usual manner ...
}
```

Initializations in a member-initializater-list are done first, before those in the body of the constructor function.

Member-initializater list can also be used to initialize local data members in the derived class:

Data member `d` of a derived class can be initialized to an initial value `i` using the unusual function notation `d(i)` in the member-initializer list.

Example:

```
SalariedEmployee::SalariedEmployee(long id, string last, string first,
                                   char initial, int dept, double sal)
: Employee(Id, last, first, initial, dept), mySalary(sal)
{
}
```

Less common, however, than “normal” initialization `d = i;` in the function body:

D. Polymorphism:

Consider:

```
class License
{
//--- Function Members
public:
. . .
void Print(ostream & out) const;
. . .
};  // end of class declaration

// Definition of Print
void License::Print(ostream & out) const
{ . . . }

// Definition of output operator<<
ostream & operator<<(ostream & out, const License & lic)
{
  lic.Print(out);
  return out;
}
```

A statement

```
cout << aLicense << "\n\n"
     << aHuntingLicense << "\n\n"
     << aDogLicense << endl;
12345 Bus Driver
Age:  30
Birthdate:  5/6/1969

00022 Esau of Isaac
Age: 100
Birthdate:  1/2/-6000

31416 Barney the Dinosaur
Age: 0
Birthdate:  1/1/2000
```

not:

```
12345 Bus Driver
Age:  30
Birthdate:  5/6/1969

00022 Esau of Isaac
Age: 100
Birthdate:  1/2/-6000
Prey:   Harts
Season: 1/1 - 12/31
Weapon: Bow & Arrow

31416 Barney the Dinosaur
Age: 0
Birthdate:  1/1/2000
Kind:   Purple
```

Need *dynamic* or *late binding* : Don't bind a definition of `Print()` to a call to `Print()` until runtime.

Use **virtual functions**:

```
class License
{
//--- Function Members
public:
. . .
virtual void Print(ostream & out) const;
. . .
//--- Data Members
protected:
  long myNumber;
  string myLastName,
         myFirstName;
  char myMiddleInitial;
  int myAge;
  ...
};  // end of class declaration

// Definition of Print
void License::Print(ostream & out) const
{ . . . }

// Definition of operator<<()
ostream & operator<<(ostream & out, const License & lic)
{
  lic.Print(out);
  return out;
}
```

This works. The same function call can cause different effects at different times (or have *many forms*), based on the function to which the call is bound. Such calls are described as **polymorphic** (Greek for "many forms"),

Polymorphism is another advantage of inheritance in an OOP language.

Thanks to polymorphism, we can apply `operator<<` to derived class objects without explicitly overloading it for those objects!

Another example:

A base-class pointer can point to any derived class object!

So consider a declaration:

```
Employee * eptr;
```

Since a `SalariedEmployee` *is-an* `Employee`, `ePtr` can point to a `SalariedEmployee` object:

```
eptr = new SalariedEmployee;
```

`eptr` can point to an `HourlyEmployee` object:

```
eptr = new HourlyEmployee;
```

For the call

```
eptr->Print(cout);
```

to work when `ePtr` points at a `SalariedEmployee` object, the function

`SalariedEmployee::Print()` within that object must be called;

but when `ePtr` is a pointer to an `HourlyEmployee`, the function

`HourlyEmployee::Print()` within that object must be called.

Here is another instance where `Print()` must be a virtual function so that this function call can be *bound to different function definitions at different times*..

By preceding a base class member function with the keyword `virtual`,a derived class can overload that function, so that *calls to that function through a pointer or reference* will be bound (at run-time) to the appropriate definition.

Sometimes one may need a **pure virtual function**:

`virtual` *`PrototypeOfFunc`* **`= 0;`**

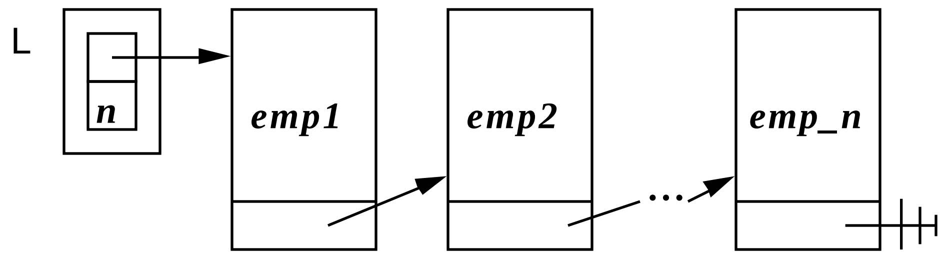
Then there is no definition of *`Func`* in the base class — called an **abstract class** — classes drived from it must provide a definition.

E. Heterogeneous Data Structures

Consider a `LinkedList` of `Employee` objects:

```
LinkedList<Employee> L;
```

Each node of `L` will only have space for an `Employee`, with no space for the additional data of an hourly or salaried employee:

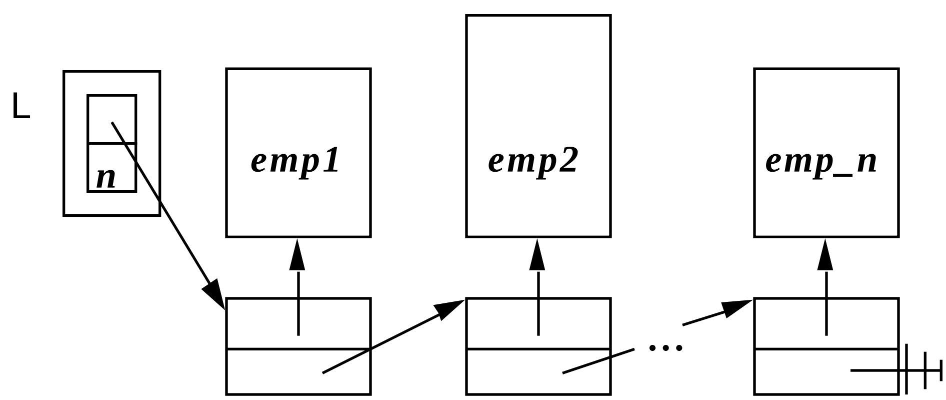


Such a list is a <u>homogeneous</u> structure: Each value in the list must be of the same type (`Employee`).

Now suppose we make `L` a `LinkedList` of `Employee` <u>pointers</u>,

```
LinkedList<Employee *> L;
```

Then each node of `L` can store a pointer to any object derived from class `Employee`:



Thus, salaried and hourly employees can be intermixed in the same list, and we have a <u>*heterogeneous storage structure.*</u>

Now consider:

```
Node * nPtr = L.first;

while (nPtr != 0)
{
   nptr->data->Print(cout);
   nptr = nPtr->next;
}
```

For the call

```
nPtr->data->Print(cout);
```

to work when `nPtr->data` points at a `SalariedEmployee` object, the function `SalariedEmployee::Print()` within that object must be called;

but when `nPtr->Data` is a pointer to an `HourlyEmployee`, the function `HourlyEmployee::Print()` within that object must be called.

Here is another instance where `Print()` must be a virtual function.