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

A major objective of OOP: _________________________________ (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:

- __________________: 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:

Good:
— Derived 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 _______________________ in those objects;

3. Where there is commonality:
   - Define ____________________________________________________________: and
   - ________________________________

These last two steps are the most difficult aspects of OOD.
Object-oriented programming (OOP): first used to describe the programming environment for _______________.

the earliest true object-oriented programming language.

Three important properties of OOP languages:

• ____________________
• ____________________
• ____________________, with the related concept of ____________________________

B. Derived Classes

Problem: Create types to model various kinds of licenses.

Critical question: What attributes do all licenses have in common?

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

```cpp
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:

```cpp
class DriversLicense
{
    public:
        ...

    private:
        ...

        int myVehicleType;
        string myRestrictionsCode;
    ...;
};

class HuntingLicense
{
    public:
        ...

    private:
        ...

        string thePrey;
        Date seasonBegin,
            seasonEnd;
    ...;
};
```
class PetLicense
{
public:
    ...
private:
    string myAnimalType;
    ...
};

Inclusion works, but is "clunky" and inefficient.

Worse, it's ____________________ — It should be that a DriversLicense _______ License, not a DriversLicense _______ License

Preferred approach:

Problem:

C++ solution:

Members declared to be ___________________________________, 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 ____________________________:

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

    long myNumber;
    string myLastName, myFirstName;
    char myMiddleInitial;
    int myAge;
    Date myBirthDay;
    ...
};
Now we can derive classes for the more specialized licenses from License:

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

class HuntingLicense ____________________________
{
public:
    ...
protected:
    string thePrey;
    Date seasonBegin,
        seasonEnd;
    ...
};

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

Classes like HuntingLicense, DriversLicense, and BoatingLicense are said to be _______________ (or __________________________), and the class License from which they are derived is called the ________________.

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

```cpp
class MooseLicense ______________________________
{
public:
    ...
protected:
    int theAntlerMaximum;
    int theBullwinkleFactor;
    ...
};
```
This leads to _______________________________ — usually picture as a tree but with arrows is drawn from a derived class to its base class:

```
License
   /    /
  Driver License Hunting License
     /     . . .
   Car License Unicycle License Moose License Dinosaur License
      . . .
   Dog License Hamster License
```

**General form of declaration of a derived class:**

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

kind_of_inheritance is usually the keyword _______________,
but it may be ____________ or ______________

**The Fundamental Property of Derived Classes:**

- ________________
- ________________
- ________________
- ___________________________________________________________ depends on the kind of inheritance specified.
  - public
  - private
  - protected
  - public and protected, respectively
  - private
  - 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 ____________relationship:

If

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

Then

A

This is in contrast to the ___________ relationship (also called the inclusion 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 object.

Design Principle: ________________________________________________________________________

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 relationship between classes is the __________ 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 appropriate 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:

```
Attributes Common to Object 1 thru Object i

Object 1  Object i
```

```
Attributes Common to Object j thru Object n

Object j  Object n
```

3. Then repeat this approach “upwards” as appropriate:

```
Attributes Common to Object 1 thru Object n

Attributes Common to Object 1 thru Object i

Attributes Common to Object j thru Object n
```

Once no more commonality exists, OO implementation then:

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

```
Attributes Common to Object 1 thru Object n
```

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

```
Attributes Common to Object 1 thru Object n

Attributes Common to Object 1 thru Object i

Attributes Common to Object j thru Object n
```
6. Derivations continue until classes for the actual objects in the system are built:

```
Attributes Common
to Object 1 thru Object n

Attributes Common
to Object 1 thru Object i
  . . .
Object 1 Object i
  . . .
Object j Object n
```

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:
        // ... salaried employee operations ...
      protected:
        double mySalary;
      }
      ```
b. An hourly employee class:

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

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

### Reusability:

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

```cpp
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`:

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

and

```cpp
void HourlyEmployee::Print(ostream & out) const
{
    // inherited member
    out << "$" << 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:

```cpp
// 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.

```cpp
// Definition of SalariedEmployee explicit-value constructor
SalariedEmployee::SalariedEmployee(long id, string last, string first, char initial, int dept, double sal)
{
        // initialize the non-inherited members in the usual manner ...
}
```

General form of Member-Initializer List Mechanism:

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

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

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

```cpp
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:

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

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

### D. Polymorphism:

Consider:

```cpp
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

```cpp
cout << aLicense << "\n\n" << aHuntingLicense << "\n\n" << aDogLicense << endl;
```
gives:

```
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____________________________: Don't bind a definition of Print() to a call to Print() until runtime.

Use__________________________:

class License
{
    //--- Function Members
    public:
    ...

    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 ______________________ (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 ____________________________________________ !

So consider a declaration:

```c++
Employee * eptr;
```

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

```c++
eptr = new SalariedEmployee;
```

eptr can point to an HourlyEmployee object:

```c++
eptr = new HourlyEmployee;
```

For the call

```c++
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 ____________________________ :

____________ PrototypeOfFunc ____________;

Then there is no definition of *Func* in the base class — called an __________________ — classes driven from it must provide a definition.
E. Heterogeneous Data Structures

Consider a LinkedList of Employee objects:

```cpp
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:

```
L  emp1  emp2  emp_n
    ^    ^    |
    |    |    
    |    v    ...
```

Such a list is a **homogeneous** structure: Each value in the list must be of the same type (Employee).

Now suppose we make L a LinkedList of Employee pointers,

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

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

```
L  emp1  emp2  emp_n
    ^    ^    |
    |    |    
    |    v    ...
```

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

Now consider:

```cpp
Node * nPtr = L.first;
while (nPtr != 0)
{
    nPtr->data->Print(cout);
    nPtr = nPtr->next;
}
```

For the call

```cpp
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.