<table>
<thead>
<tr>
<th>Generalization versus Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstraction:</strong> simplify the description of something to those aspects that are relevant to the problem at hand.</td>
</tr>
<tr>
<td><strong>Generalization:</strong> find and exploit the common properties in a set of abstractions.</td>
</tr>
<tr>
<td>hierarchy</td>
</tr>
<tr>
<td>polymorphism</td>
</tr>
<tr>
<td>genericity</td>
</tr>
<tr>
<td>patterns</td>
</tr>
</tbody>
</table>
Four-Fold Path to Generalization

**Hierarchy**
Exploitation of an “is-a-kind-of” relationship among kinds of entities to allow related kinds to share properties and implementation.

**Polymorphism**
Exploitation of logical or structural similarities of organization to allow related kinds to exhibit similar behaviors via similar interfaces.

**Genericity**
Exploitation of logical or structural similarities of organization to produce generic objects.

**Patterns**
Exploitation of common relationship scenarios among objects. (e.g., client/server system)
Hierarchy

Represented by generalize/specialize graph

Based on “is-a-kind-of” relationship
  E.g., a Manager is an Employee; a robin is a bird, and so is an ostrich.

Is a form of knowledge representation – a “taxonomy” structures knowledge about nearby entities.

Extendable without redefining everything
  E.g., knowing a robin is a bird tells me that a robin has certain properties and behaviors, assuming I know what a “bird” is.

Specialization can be added to proper subset of hierarchy
A generalization/specialization hierarchy based on “is-a-kind-of” relationships:

- Person
  - Name
  - Address

- Student
  - Name
  - Address
  - ID
  - Major
  - Level

- Employee
  - Name
  - Address
  - ID
  - Department

more general
less details

more specialized
more details
Inheritance

Terminology
- Base type or class (a.k.a. superclass, parent type)
- Derived type or class (a.k.a. subclass, subtype, child type)

Important Aspects
- **Programming**: implement efficiently a set of related classes (mechanical)
- **Design**: organize coherently the concepts in an application domain (conceptual)
- **Software Engineering**: design for flexibility and extensibility in software systems (logical)
class andGate {
private:
    string mName;
    vector<Wire*> In;
    Wire* Out;
    unsigned int mFanIn;
    unsigned int mInputsConnected;

    virtual bool Evaluate() const;

public:
    andGate(const string& ID = "anon", unsigned int fanIn = 0);
    unsigned int fanIn() const;
    string Name() const;
    virtual bool Valid() const;
    virtual bool addIn(Wire* const Input = NULL);
    virtual bool addOut(Wire* const Output = NULL);
    virtual void Act(Wire* const Source);
    virtual ~andGate();
};
class orGate {
private:
    string mName;
    vector<Wire*> In;
    Wire* Out;
    unsigned int mFanIn;
    unsigned int mInputsConnected;

    virtual bool Evaluate() const;

public:
    orGate(const string& ID = "anon", unsigned int fanIn = 0);
    unsigned int fanIn() const;
    string Name() const;
    virtual bool Valid() const;
    virtual bool addIn(Wire* const Input = NULL);
    virtual bool addOut(Wire* const Output = NULL);
    virtual void Act(Wire* const Source);
    virtual ~orGate();
};
What is Common?

Both classes contain the data members

```cpp
    string mName;
    vector<Wire*> In;
    Wire* Out;
    unsigned int mFanIn;
    unsigned int mInputsConnected;
```

and member functions which will have identical implementations

```cpp
    unsigned int fanIn() const;
    string Name() const;
    virtual bool Valid() const;
    virtual bool addIn(Wire* const Input = NULL);
    virtual bool addOut(Wire* const Output = NULL);
    virtual void Act(Wire* const Source);
```

From a coding perspective, this is somewhat wasteful because we must duplicate the declarations and implementations in each class.

From a S/E perspective, this is undesirable since we must effectively maintain two copies of (logically) identical code.
What is Different?

The following member function would have different implementations in the two classes

    virtual bool Evaluate() const;

This would evidently be the only difference between the two classes!
What Do We Want?

Simply put, we want to exploit the fact that \texttt{andGate} and \texttt{orGate} both are "gates".

That is, each shares certain data and function members which logically belong to a more general (more basic) type which we will call a \texttt{Gate}.

We would prefer to NOT duplicate implementation but rather to specify that each of the more specific types will automatically have certain features (data and functions) that are derived from (or inherited from) the general type.

Question: are there any attributes or operations in the overlap that we don't want to include in the base type \texttt{Gate}?
By employing the C++ inheritance mechanism…

Inheritance in C++ is NOT simple, either syntactically or semantically. We will examine a simple case first (based on the previous discussion) and defer explicit coverage of many specifics until later.

Inheritance in C++ involves specifying in the declaration of one class that it is derived from (or inherits from) another class.

Inheritance may be public or private (or protected). At this time we will consider only public inheritance.

It is also possible for a class to be derived from more than one (unrelated) base class. Such multiple inheritance will be discussed later…
The Base Class: Gate

Having identified the common elements shared by both classes, we specify a suitable `base` class:

```cpp
class Gate {
private:
    string mName;
    vector<Wire*> In;
    Wire* Out;
    unsigned int mFanIn;
    unsigned int mInputsConnected;

    virtual bool Evaluate() const = 0;

public:
    Gate(const string& ID = "anon", unsigned int fanIn = 0);
    unsigned int fanIn() const;
    string Name() const;
    virtual bool Valid() const;
    virtual bool addIn(Wire* const Input = NULL);
    virtual bool addOut(Wire* const Output = NULL);
    virtual void Act(Wire* const Source);
    virtual ~Gate();

};
```
A Derived Class: andGate

class andGate : public Gate {

protected:
  virtual bool Evaluate() const;

public:
  andGate(const string& ID = "anon",
          unsigned int fanIn = 0);
  virtual ~andGate();
};
When an object of a derived type is declared, the default constructor for the base type will be invoked BEFORE the body of the constructor for the derived type is executed (unless an alternative action is specified…).

Alternatively:

```cpp
andGate::andGate(const string& ID, unsigned int fanIn) :
            Gate(ID, fanIn) {
        }
```

Alternatively, we can also implement an entirely specialized constructor in the derived type.
Objects of a derived type inherit the data members and function members of the base type. However, the derived object may **not** directly access the **private** members of the base type:

```cpp
bool andGate::Evaluate() const {
    bool Value = In[0]->Output();
    for (unsigned int pos = 1; pos < mFanIn; pos++) {
        Value = Value && In[pos]->Output();
    }
    return Value;
}
```

**Error:** cannot access **private** member declared in class `Gate`.
Protected Access

The restriction on a derived type's access seems to pose a dilemma:

- Having the base type use only public members is certainly unacceptable.
- Having the derived class use the public interface of the base class to access and/or modify private base class data members is clumsy.

C++ provides a middle-ground level of access control that allows derived types to access base members which are still restricted from access by unrelated types.

The keyword protected may be used to specify the access restrictions for a class member:

```cpp
class Gate {  
protected:  
    string mName;  
    vector<Wire*> In;  
    Wire* Out;  
    unsigned int mFanIn;  
    unsigned int mInputsConnected;
};
```
A Sibling Class

class orGate : public Gate {
protected:
    virtual bool Evaluate() const;

public:
    orGate(const string& ID = "anon", unsigned int fanIn = 0);
    virtual ~orGate();
};

Note that, so far as the language is concerned, \texttt{andGate} and \texttt{orGate} enjoy no special relationship as a result of sharing the same base class.
Using Objects of Derived Classes

Objects of a derived class may be declared and used in the usual way:

NEED AN EXAMPLE HERE!!
Extending the Hierarchy

Actually, we would probably want a more complex hierarchy:

There's no restriction on how many levels of inheritance can be designed, nor is there any reason we can't mix inheritance with association and/or aggregation.
Overriding a Base Class Member Function

In some cases, the base class may contain an implemented member function which is not entirely adequate:

```cpp
string Gate::Name() const {
    return mName;
}
```

A derived type may want a more specialized version:

```cpp
string andGate::Name() const {
    return "AND" + mName;
}
```

When `Name()` is called on an `andGate` object, this version, not the base version will be executed.
It is legal to assign a derived type object to a base type object, however:

When a derived object is assigned to a base target, only the data members appropriate to the target type are copied.

This is known as slicing.

Actually, this would seem to be inevitable since the base object would not have any place to store data members that were added in the derived type.
Assigning Base Type to Derived Type

By default, a base type object may **not** be assigned to a derived type object.

After all, the derived type may declare "extra" data members that would not receive appropriate values.

It's possible to legalize this with the right constructor overloading (later).
Parameter Passing Issues

The rules are essentially the same in four situations:

- when passing a derived object **by value** as a parameter.
- when returning a derived object **by value**
- when initializing a base object with a derived object
- when assigning a derived object to a base object

A derived type may be copied when a base type is targeted — however, slicing will occur.

A base type may not be copied when a derived type is targeted — unless a suitable derived type constructor is provided to legalize the conversion.