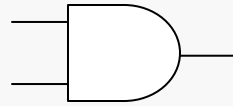
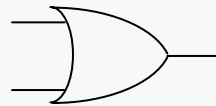


andGate: accepts two binary inputs  $x$  and  $y$ , emits  $x \ \& \ y$



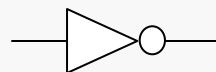
x	y	Output
0	0	0
0	1	0
1	0	0
1	1	1

orGate: accepts two binary inputs  $x$  and  $y$ , emits  $x \ | \ y$



x	y	Output
0	0	0
0	1	1
1	0	1
1	1	1

notGate: accepts one binary input  $x$ , emits  $!y$



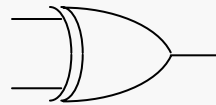
x	Output
0	1
1	0

nandGate: accepts two binary inputs  $x$  and  $y$ ,  
emits  $\neg(x \ \& \ y)$



x	y	Output
0	0	1
0	1	1
1	0	1
1	1	0

xorGate: accepts two binary inputs  $x$  and  $y$ ,  
emits  $x \ \wedge \ y$



x	y	Output
0	0	0
0	1	1
1	0	1
1	1	0

The basic logic gates can be combined to form more complex digital circuits of all types.

In fact, the NOT and AND gates alone are sufficient, but that does not really concern us...

Consider the fundamental characteristics of the logic gates presented:

- ability to connect to one or two input wires
- ability to connect to one output wire
- ability to receive a value from a connected input wire
- ability to transmit a value to a connected output wire
- ability to compute correct output value, given current input value(s)

Now, the notGate is something of an anomaly since it is the only one that takes a single input wire rather than two. For now, we will restrict our attention to the two-input gates.

The remaining (two-input) gates differ only in how they calculate the correct output value.

Note that in order to build circuits it appears we must also model wires used to connect logic gates.

It is sensible to view each of the 2-input logic gates as a specialized sub-type of a generic logic gate (a base type) which has 2 input wires and transmits its output to a single output wire.

The base type gate doesn't actually need to define a calculation for the output value, since each of the sub-types must specialize the calculation.

The base type gate is actually an example of an *abstract type*.

An *abstract type* is a type of which no actual instances ever exist.

Abstract types play important roles in the design of inheritance hierarchies, even though no objects of those types will ever be created.

We also note that each 2-input gate must be capable of supporting associations to two input wire objects and one output wire object.

One possible C++ representation of the 2-input gate type appears on the following slide:

```
class Wire;

class Gate {
protected:           // note use of protected access
    Wire *Left, *Right; // input wire links
    Wire *Out;        // output wire link
    bool Evaluate() const; // calculate output value

public:
    Gate(Wire* const L = NULL, Wire* const R = NULL,
          Wire* const O = NULL);

    bool addIn(Wire* const pW = NULL); // add next input wire
    bool addOut(Wire* const pW = NULL); // add next output wire
    void Act(Wire* const Source); // xmit output value
};
```

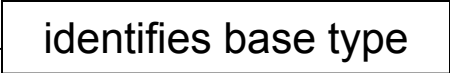
Note that the internal gate logic is symmetric with respect to its inputs, so we can be fairly loose about which is which.

Now, the andGate may be implemented by deriving it from the base type.

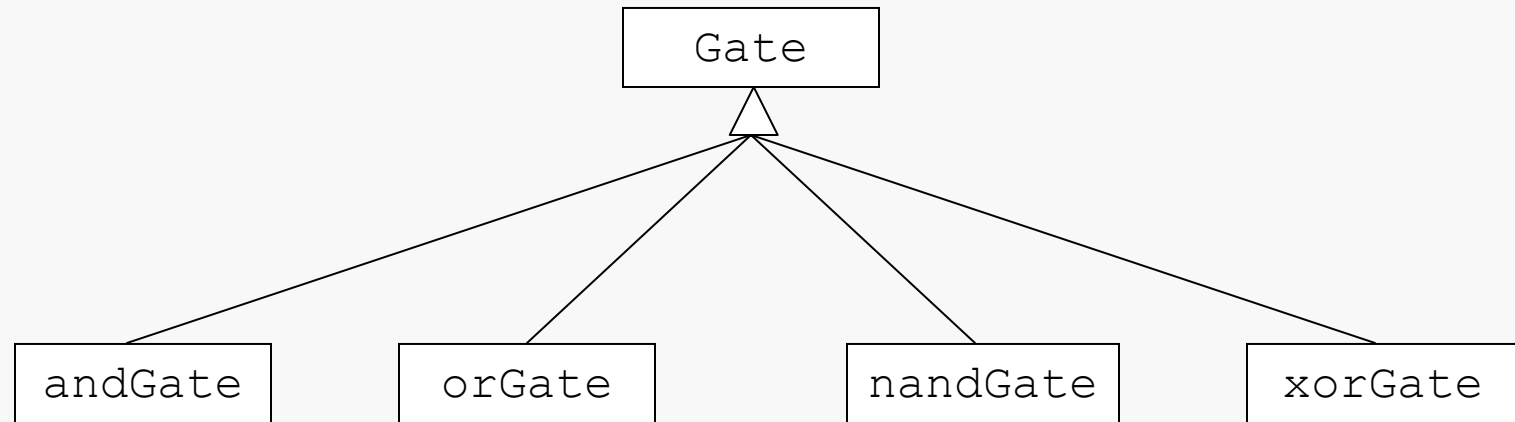
In C++, a derived type (sub-type) automatically includes all the data members and most of the function members of its base type. (Constructors and destructors are special...)

The derived type only needs to declare any new data and function members it needs, supply constructors and destructor (if needed), and to possibly implement new versions of inherited member functions to modify behavior:

```
class andGate : public Gate {  
protected:  
    bool Evaluate() const;  
  
public:  
    andGate(Wire* const L = NULL, Wire* const R = NULL,  
            Wire* const O = NULL);  
};
```



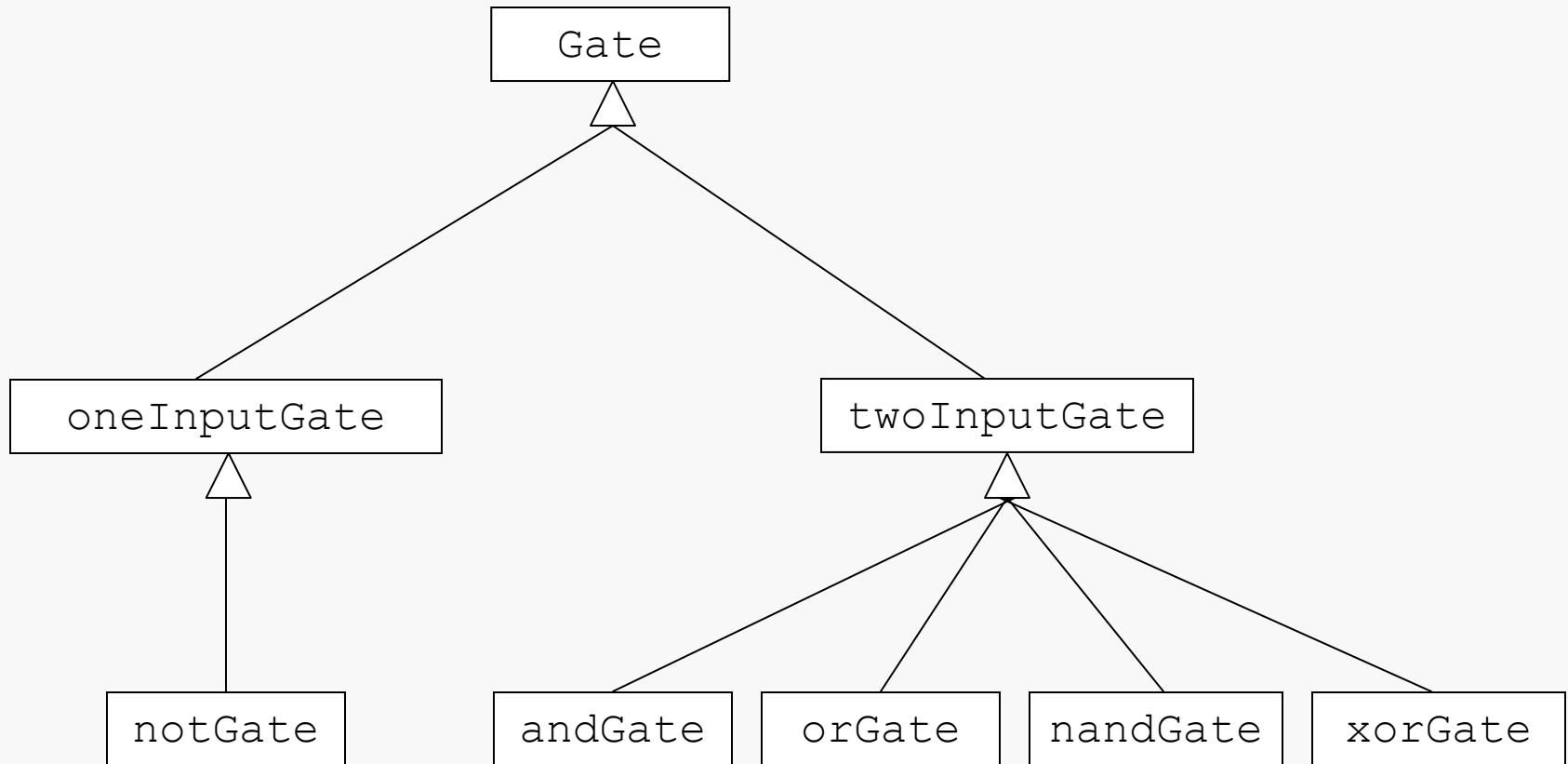
We can repeat this approach to derive the remaining 2-input sub-types from `Gate`:



Each of these sub-types would have an implementation virtually identical to the one already shown for the `andGate` class.

But... how does with the `notGate` fit into this hierarchy? Or other gate types that don't take two inputs? One possibility is to rethink the current hierarchy to incorporate a more general base type and intermediate sub-types of that:

Here's a refinement that would allow extension to include a variety of specific gate types:



Additional sub-types (like 3-input gates) can easily be added, but it would obviously get cumbersome if lots of such additions were made. There is a better way... but we will pursue this design for now.



```
class Wire;

class Gate {
protected:
    // omit data members completely
    bool Evaluate() const;    // calculate output value

public:
    Gate();
    bool addIn(Wire* const pW = NULL);    // add next input wire
    bool addOut(Wire* const pW = NULL);    // add next output wire
    void Act(Wire* const Source);    // xmit output value
};
```

None of the member functions do anything (except return a value if necessary).

This is still an abstract type, so we don't ever intend to create instances of it.

```
class Wire;

class twoInputGate : public Gate {
protected:
    Wire *Left, *Right;           // declare necessary data members
    Wire *Out;                    // input wire links
                                // output wire link

public:
    twoInputGate(Wire* const L = NULL, Wire* const R = NULL,
                 Wire* const O = NULL);

    bool addIn(Wire* const pW = NULL); // add next input wire
    bool addOut(Wire* const pW = NULL); // add next output wire
    void Act(Wire* const Source);      // calc and xmit value
};
```

Here, we will implement meaningful member functions to add wires, and a generic function to trigger evaluation and transmission of an output value.

We still won't implement evaluation code, so we'll just keep the inherited function for that.

The other intermediate class (`oneInputGate`) is very similar to `twoInputGate`.

The *concrete* gate types don't really change from the `andGate` example presented earlier, aside from specifying a different base class.

Each of the concrete types need only implement its own version of `Evaluate()` to provide the correct output value.

There are some questions that need to be answered:

- when a sub-type re-implements a function it inherits from its base, what happens?
- is there any way to actually make it impossible for the client to create an instance of an abstract class (since they aren't fully functional)?
- are there any gotcha's when we use the derived types?

What happens if a sub-type declares and implements a function whose interface is exactly the same as a function it inherited from its base class?

When a client calls that function directly, it will get the sub-type's implementation of it rather than the base type's implementation.

```
twoInputGate myGate;  
  
myGate.Act(...);    // calls twoInputGate.Act() !!
```

So, from the client's perspective, the overriding function appears to have replaced the version that was inherited from the base class.

This allows a derived type to modify behavior inherited from its base... which is very useful.

We can't really proceed much further without modeling a wire. Fortunately, there is only one type of wire:

```
class Gate;

class Wire {
private:
    bool Val;           // wires store a value
    Gate *In, *Out;    // support connections to two gates

public:
    Wire(bool V = 0, Gate* const I = NULL, Gate* const O = NULL);
    bool addIn(Gate* const I);
    bool addOut(Gate* const O);
    void Act(bool V);
    bool Output() const; // return stored value
    ~Wire();
};
```

`Wire` objects store a value... that isn't entirely necessary but it is convenient during testing to have current values stored somewhere.

The stored values are represented using `bool` variables. That saves space since the only possible values are 0 and 1.

A bigger issue is that `Wire` objects use pointers to `Gate` objects, but if we assemble a circuit the targets of those pointers will be objects of the concrete sub-types.

Just how will that work?

Syntactically, there is no problem, because a base type pointer can always store the address of an object of any sub-type.

But are there any other problems... yes and no.

*polymorphism*: automatically obtaining the correct behavior from an object even in situations where we do not know precisely what kind of object we're dealing with.

Consider the following situation:

```
Gate *p;                // make a base-type pointer
p = gateFactory.Make(); // give it a derived-type target

p->Act(. . .);          // call a member function of the gate object
```

Now, the member function `Act ()` has been implemented in several places:

- the base class `Gate`
- the sub-types `oneInputGate` and `twoInputGate`

Also, `Act ()` will call `Evaluate ()` ...

and the correct version of `Evaluate ()` is implemented in the class `andGate`.

Our problem is that there is no way for the compiler to know exactly what kind of object  $p$  is pointing to after the function is called:

```
p = gateFactory.Make(); // give it a derived-type target
```

This may seem contrived at the moment, but we will soon see that it is actually a very common situation. All the compiler can be sure of is that the target of  $p$  is an object of one of the sub-types of `Gate`.

```
p->Act(. . .); // call a member function of the gate object
```

The compiler's job is to determine what function declaration matches the call, but that is now impossible. There are at least three relevant implementations of `Act()`.

So how do we achieve polymorphic behavior?

The compiler's only clue is that the pointer is a `Gate*`, and that's just not sufficient.

**There is simply no way for the compiler to make the correct decision...**



The compiler's dilemma is resolved by using *virtual* member functions.

To make a member function virtual, just precede its declaration with `virtual`. The declaration of Gate could become:

```
class Gate {
protected:                                // omit data members completely
    virtual bool Evaluate() const;
public:
    Gate();

    virtual bool addIn(Wire* const pW = NULL);
    virtual bool addOut(Wire* const pW = NULL);
    virtual void Act(Wire* const Source);
};
```

When:

- a member function is declared to be `virtual`, and
- the function is called by dereferencing a pointer, and
- the function is a member of a class within an inheritance hierarchy

then the effect is that the decision of exactly what function is actually being called is delayed until the program is actually running.

This is called *runtime binding* or *late binding*.

We will discuss the mechanism that makes this possible later.

For now, it is enough to know that:

- runtime binding is the key to making polymorphism work in C++
- runtime binding is enabled by combining inheritance, call-by-pointer, and virtual member functions

What functions do you make virtual?

The ones for which you need to achieve polymorphic behavior. Essentially, this means the ones that you expect to override in some derived class.

So, it's important to consider the entire hierarchy design at once.

If a base class does not declare the right member functions to be virtual, that limits what can be achieved by deriving sub-types from that base class.

In the latest revision of the class `Gate`, all the member functions except constructor and destructor have been made virtual.

That maximizes the flexibility of the derived sub-types.

However, it's still not entirely ideal...

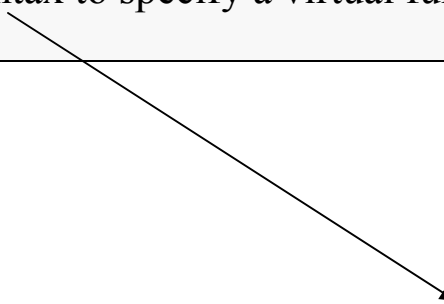
The `Gate` class declares a number of member functions for which it cannot provide meaningful implementations. (All of them, in fact.)

That's ugly. But, we can't just omit those member function declarations from `Gate`, because we need them to make polymorphism work in the derived sub-types.

Is there any way to eliminate the need to provide useless, silly implementations of those functions in the `Gate` class?

Yes. A *pure virtual function* requires no implementation. Basically, a pure virtual function is simply a function declaration with the promise that the function will eventually be overridden and implemented in a subtype. Here's the syntax to specify a virtual function is pure:

```
class Gate {  
    . . .  
public:  
    . . .  
    virtual bool addIn(Wire* const pW = NULL) = 0;  
    . . .  
};
```



A class that declares one or more pure virtual functions cannot ever have any instances.

Such a class is called an *abstract class*.

In our next revision of the logic gates hierarchy, `Gate` and its two immediate sub-types will all become abstract classes. That improves the model, since there should not, in fact, ever be instances of those types.

Only in the lowest-level types, such as `andGate`, will we finally override all of the pure virtual functions declared in `Gate`.

The idea is to not provide useless non-functional implementations in the higher-level classes.

The nice side effect is that this also prevents a client from attempting to use objects of those abstract types.

An attempt to declare an object of an abstract class leads to a compile-time error.

The abstract base Gate class simply provides the conceptual platform for deriving useful subtypes:

```
class Gate {
protected:
    virtual bool Evaluate() const = 0;

public:
    Gate();
    virtual bool addIn(Wire* const Input = NULL) = 0;
    virtual bool addOut(Wire* const Output = NULL) = 0;
    virtual void Act(Wire* const Source) = 0;
};
```

The only member function that is implemented is the default constructor, and it doesn't really do anything and could be omitted.

The abstract `twoInputGate` class provides the data and function members to associate to wire objects, and a generic function to trigger the gate's action:

```
class twoInputGate : public Gate {
protected:
    Wire *Left, *Right, *Out;

public:
    twoInputGate(Wire* const L = NULL, Wire* const R = NULL,
                Wire* const O = NULL);
    virtual bool addIn(Wire* const Input = NULL);
    virtual bool addOut(Wire* const Output = NULL);
    virtual void Act(Wire* const Source);
};

void twoInputGate::Act(Wire* const Source) {

    if ( (Left == NULL) || (Right == NULL) || (Out == NULL) )
        return;

    if ( Source == Left || Source == Right )
        Out->Act( Evaluate() );
}
```

The concrete andGate class only needs to provide the evaluation function:

```
class andGate : public twoInputGate {
protected:
    virtual bool Evaluate() const;

public:
    andGate(Wire* const L = NULL, Wire* const R = NULL,
            Wire* const O = NULL);
};
```

```
bool andGate::Evaluate() const {
    return ( Left->Output() && Right->Output() );
}
```

What's interesting here is that the `Act()` function implemented in the base class will actually call the `Evaluate()` function implemented in the derived class... as long as we make the call to `Act()` via a base-type pointer. **That's** polymorphism.



Virtuality is inherited.

In other words, since `Gate` declared `Evaluate()` as virtual, when `twoInputGate` inherits `Gate::Evaluate()` it's still virtual (still pure virtual in fact). And the same holds true when `andGate` is derived from `twoInputGate`.

Virtuality is persistent.

When `twoInputGate` overrides `Gate::Act()`, the overriding function `twoInputGate::Act()` is automatically virtual, whether `twoInputGate` declares it to be or not.