Crossroads
The ACM Student Magazine

Object-Orientation and C++

Part II of II

G. Bowden Wise

Object-orientation has become a buzzword with many mean- ings. It is a design methodology, a paradigm (a way of thinking about problems and finding solutions), and a form of program- ming. As a design methodology, we can use object-oriented tech- niques to design software systems. But it is more than a design methodology, it is a whole new way of thinking about prob- lems. Object-oriented design allows us to think about the actual real-world entities of the problem we are attempting to provide a solution for. Beginning the design with concepts from the real- world problem domain allows the same concepts to be carried over to implementation, making the design and implementation cycle more seamless.

Once a design has been conceived, a programming language can be chosen for implementation. By factoring out the inherit- ance relationships from the object hierarchies discovered during design, one can even implement the system in a traditional, non- object-oriented language. However, using a object-oriented lan- guage, such as C++, makes it easier to realize the design into an implementation because the inherent relationships among objects can be directly supported in the language.

Languages such as C++ are considered hybrid languages because they are multi- paradigm languages. C++ is an object- oriented extension of C and can be used as a

procedural language or as an object-oriented language. In this issue, we continue our tour of the object-oriented features of C++.

The Object-Oriented Features of C++

In the last issue, we learned how C++ provides support for encapsulation, the first of the three fundamental characteristics of the object-oriented paradigm. C++ provides the class construct to allow programmers to define new data types which can be used as naturally as the types built-in to the language. Classes allow programmers to shield implementation details from the class users through data hiding and abstraction.

We now examine how C++ provides for the remaining two characteristics of the object-oriented paradigm: inheritance and polymorphism. These two characteristics bring the most power to object-oriented programming.

INHERITANCE in C++. One of the major strengths of any object-oriented programming language is the ability to build other classes from existing classes, thereby reusing code. Inheritance allows existing types to be extended to an associated collection of sub-types.

Recall that one of the key actions of object-oriented design is to identify real-world entities and the relationships among them. When a software system is designed, a variety of objects arise, which may be related in one way or another. Some classes may not be related at all. Many times it makes sense to organize the object classes into an inheritance hierarchy. Organizing a set of classes into a class hierarchy requires that we understand the relationships among the classes in detail. Not all class relationships dictate that inheritance be used.

C++ provides three forms of inheritance: public, private, and protected. These different forms are used for different relation- ships between objects. To illustrate these different types of inher- itance, let's look at several different class relationships.

The first relationship is the IS-A relationship. This type of relationship represents a specialization between types or classes. IS-A inheritance holds for two classes if the objects described by one class belongs to the set of objects described by the other more general class. The IS-A relationship is the traditional form of inheritance called subtyping. The subtype is a specialization of some more general type known as the supertype. In C++, the supertype is called the base class and the subtype the derived

class.

Figure 1 shows an inheritance diagram for a hierarchy of digital logic gates. At the root of the hierarchy is the class gate, which defines the public interface common to all gates. At the next level of the hierarchy are the classes uni_gate and binary_gate which are both derived from gate. These represent base classes for gates with one and two inputs respectively. Finally, at the bottom level are classes for the individual gates. The classes follower and inverter are single input gates derived from class uni_gate. The classes and_gate, or_gate, and xor_gate are derived from binary_gate since they all have two inputs.

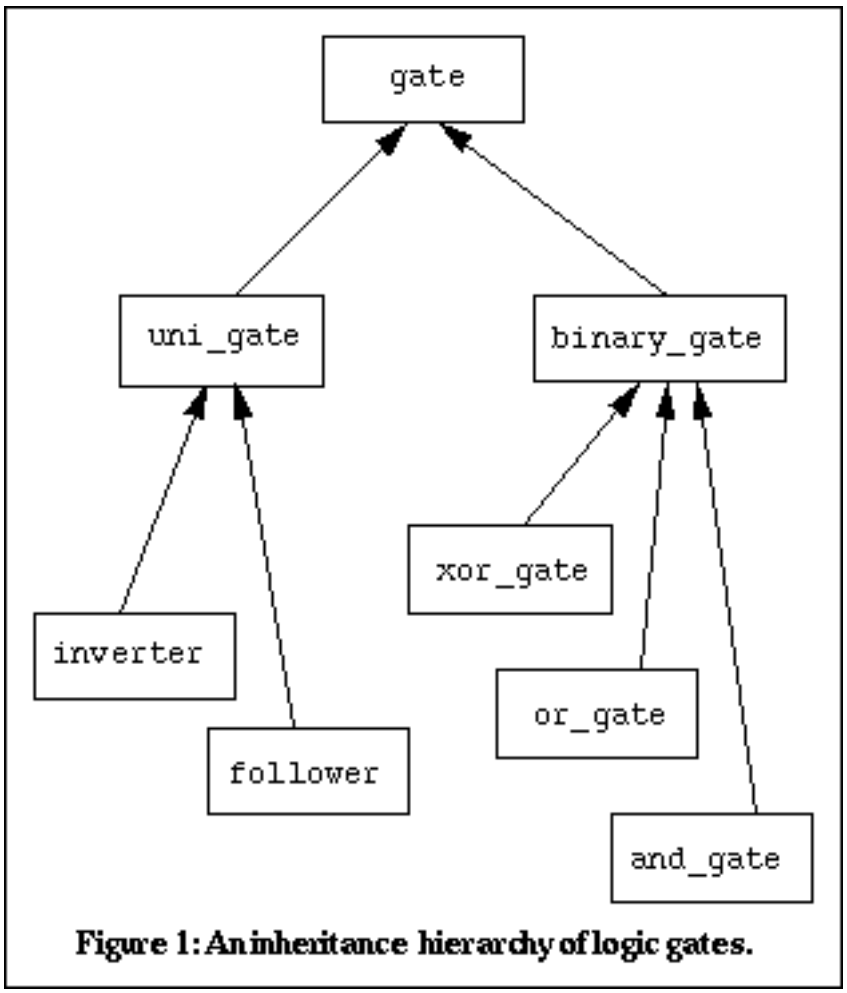

Figure 1: An inheritance hierarchy of logic gates.

To implement the IS-A relationship in C++ we use public inheritance. When public inheritance is used the public parts of the base class become public in the derived class and the protected parts of the base class become protected in the derived class. Figure 2 shows how public inheritance is used to derive the class and_gate from the class binary_gate. The public inter- face of binary_gate becomes publicly accessible to users of the and_gate class.

Figure 2: Public inheritance

```
class gate
{
public:
   gate() {}
   virtual unsigned value () = 0;
   virtual void printon (ostream& os) = 0;
   virtual void printtable (ostream& os) = 0;

   friend ostream& operator<<
                  (ostream& os, gate& g);
};


class binary_gate : public gate
{
   unsigned  a : 1;
   unsigned  b : 1;
public:
   binary_gate (unsigned in1=0,unsigned in2=0)
   const unsigned A() const;
   const unsigned B() const;
   virtual unsigned setA (unsigned in1);
   virtual unsigned setB (unsigned in2);
   virtual void printon (ostream& os);
   virtual void printtable (ostream &os);
   virtual unsigned value
          (unsigned v1, unsigned v2) = 0;
};
```

Sometimes the relationship between two objects denotes containership or a part-of relationship. Neither object is a specialization of the other, but one of them is part of or contained in the other. This relationship is called the HAS-A relationship.

To implement the HAS-A relationship in C++ we use either composition or private inheritance. For example, a stack can be implemented using an array. We can either use the stack as a data member (composition) or derive the stack class from the array class using private inheritance. Figure 3 shows the definition of a stack class using

composition. Note that the data array is a private data member of the class.

Figure 3: HAS-A using composition

```
class Stack
{
private:
   IArray* data;
   int  top;

public:
   enum { STACK_MAX       = 10 };
   enum { STACK_UNDERFLOW = -111 };

   Stack ();
   ~Stack();

   void push (const int i);
   int  pop ();
   int  empty() const;

   friend ostream& operator<<
                  (ostream&os, const Stack& s);
};
```

It is also possible to use inheritance to achieve a containership relationship between two classes. Private inheritance is used when the inheritance is not part of the interface; the base class is an implementation detail. Under private inheritance, the public and protected parts of the base class become part of the private part of the derived class. Users of the derived class cannot access any of the base class interface. However, member functions of the derived class are free to use the public and private parts of the base class. When used this way, users cannot write code that depends on the inheritance. This is a powerful way of preserving your ability to change the implementation to a different base class. Figure 4 shows the same stack class, but uses private inheritance instead of composition. In most cases, composition is preferred over private inheritance. Private inheritance is rarely used because it is always easier to use composition. However, if the derived class must override a virtual function of the base class, then private inheritance must be used.

Figure 4: HAS-A using private inheritance

```
class Stack : private IArray
{
private:
   int  top;

public:
   enum { STACK_MAX       = 10 };
   enum { STACK_UNDERFLOW = -111 };

   Stack ();
   ~Stack();

   void push (const int i);
   int  pop ();
   int  empty() const;

   friend ostream& operator<<
                (ostream&os, const Stack& s);
};
```

One other form of inheritance, which is very rarely used is protected inheritance. Protected inheritance is also used to implement HAS-A relationships. When protected inheritance is used, the public and protected parts of the base class become protected in the derived class. So, you may wish to use protected inheritance when the inheritance is part of the interface to derived classes, but not part of the interface to the users. A protected base class is almost like a private base class, except the interface is known to derived classes.

It is best to use composition where possible. In cases where you must override functions in a base class then by all means use inheritance. Only use public inheritance if your derived class is indeed a specialization of the base class, otherwise, private inheritance should be used. Needlessly using inheritance makes your system harder to understand.

In summary, a class specifies two interfaces: one to the users of the class (the public interface) and another to implementors of derived classes (the union of public and

protected parts). Inheritance works almost identically. When the inheritance is public, the public interface of the base class becomes part of the public interface to users of the derived class. When the inheritance is protected, the public and protected parts of the base class are accessible to the member functions (the implementation) of the derived classes, but not to general users of the derived classes. Finally, when inheritance is private, the public and protected parts of the base class are only accessible to the implementor of the class, but not to users or derived classes.

POLYMORPHISM in C++. Polymorphism is the last of the three fundamental primitives of object-oriented programming and the most important. Together with inheritance, polymorphism brings the most power, in terms of run-time flexibility, to object-oriented programming. Polymorphism, which means many forms, provides a generic software interface so that a collection of different types of objects may be manipulated uniformly. C++ provides three different types of polymorphism: virtual functions, function name overloading, and operator overloading.

Virtual functions enable an inheritance hierarchy of related objects to be manipulated in the same way. Consider the collection of digital logic gates in Figure 1, which all compute a value based on their inputs. A circuit simulation program does not know what kinds of gates will be used in a circuit until run-time. However, if all gates respond to the same message value(), the virtual function mechanism ensures that the correct value function is called based on the type of gate encountered dynamically at run-time.

A normal member function of a class (not a virtual) is statically bound to a single procedure call at compile-time. There is only one set of instructions for that member function that is used for all instances of the class.

Figure 5(a) shows the definitions of classes A and B. B has been derived from class A. Class A has two members an integer a and a void function v(). Class B also has two members an integer b, and a void member v(). Because v() contains the same signature as v() from the derived class A, it overrides the A definition. The data member a of A, however, is inherited by class B.

Figure 5(b) shows what the layout of objects of classes A and B might look like in memory. Each A object contains enough memory to hold a single integer, while each B objects can hold two integers. Note that an object does not contain the code for the member functions. Only a single copy of the machine code for a member function is

generated by the compiler. Because the functions are not virtual, the compiler knows which procedure v() is called for every object.

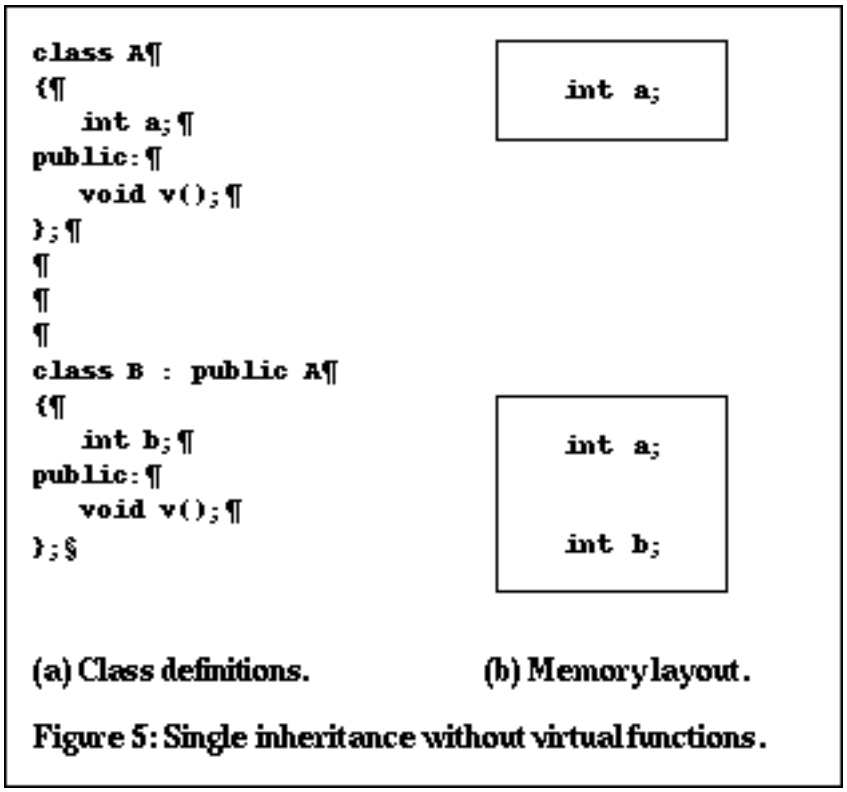

Figure 5: Single inheritance without virtual functions.

When a member function is made virtual by prefixing its definition with the keyword virtual, the actual procedure that is called depends on the type of the object instance through which the call is invoked. This is called dynamic binding. The usual static form of binding is sometimes called early binding.

Figure 6 shows the definitions and memory layouts of classes A and B when the member function v() is made virtual. When a class definition contains a virtual member, the C++ compiler creates a special table in memory, called the vtable, to hold the procedure addresses for the virtual functions. In addition to the space needed for the interger variables, each object also needs a pointer to the vtable. When a call to a virtual function is encountered at run-time, the vtable for the particular object is consulted to determine which function is to be called.

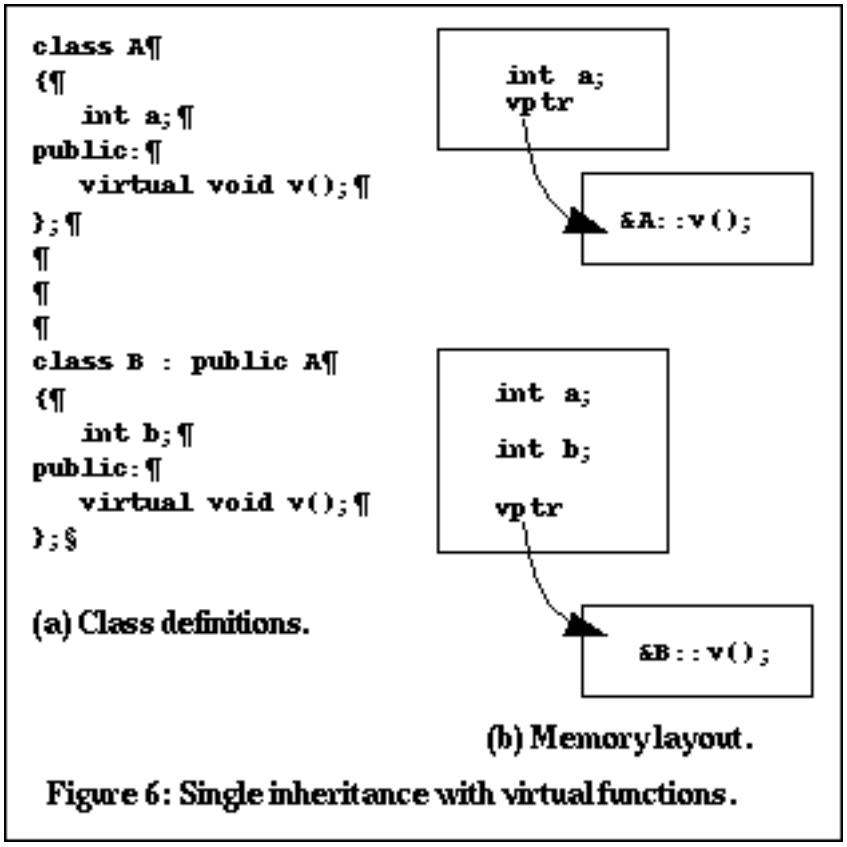

Figure 6: Single inheritance with virtual functions.

The virtual function mechanism can only be invoked through the use of a base class reference or pointer. Suppose we have the following code:

```
A theA;
B theB;
A* ptr;
```

Recall that a base class pointer can point to an object of the base type or an object of any type that is derived from the base class so ptr may point to an A or a B object. Because v() is virtual, when we invoke v() by dereferencing the pointer, the virtual function table tells the compiler which function definition to use:

```
ptr = &theA;
A->v();  // Calls A::v()

ptr = &theB;
A->v();  // Calls B::v()
```

We can also use a reference, as follows:

```
A& refA = theA;
```

```
refA.v();  // Calls A::v()

A& refB = theB;
refB.v();  // Calls B::v()
```

Using the dot operator does not invoke the virtual function mechanism. Intead, the functions are invoked statically:

```
  theA.v();  // Always calls A::v()
  theB.v();  // Always calls B::v()
```

Virtual functions are also used to implement the logic gate hierarchy shown in Figure 1. The class gate is an abstract base class at the root of the inheritance hierarchy. The class definition of gate can be found in Figure 2. A class is considered abstract when some of its virtual member functions do not have an implementation. These functions are assigned to be zero in the class definition.The methods value(), printon(), and printtable() are all virtual functions. Derived classes must provide implementations for them.

Another form of polymorphism found in C++ is function overloading. A function is said to be overloaded when it is declared more than once in a program. Overloading allows a set of functions that perform a similar operation to be collected under the same name. When there are several declarations of the same function, the compiler determines which function should be called by examining the return type and argument signature of the function call.

Suppose we wish to provide a lookup facility for ints, chars, and doubles. We would define implementations for each of these arrays and overload the function Lookup:

```
int Lookup (int    v);
int Lookup (char   v);
int Lookup (double v);
```

The compiler will call the correct Lookup routine based on the type of the arguments in the call.

When the types of arguments found in the call do not match any of those found in the definitions, the compiler will try converting the arguments to other types in an attempt

to find a match. First, type promotions are applied. A promotion is the implicit conversion of a type to a wider representation (i.e., convert an int to a double). If no promotions result in a match, standard type conversions are performed. If a match is still not found, a compile error results. For example, when the following call is made:

```
long l = 1L;
Lookup (l);
```

The variable l is implicitly converted to an int so that a call to Lookup (int) is made.

When we define new data types, it is often useful to define standard operations that are found in similar types. For example, a complex type also has addition and subtraction defined for it. We can use operator overloading so that the addition (`+') and subtraction (`-') operators work for complex objects just as they do for ints and floats.

Operators are defined in much the same was as normal C++ functions and can be members or non-members of a class. Operators take one or two arguments and are called unary and binary operators accordingly. In C++, a member operator function is defined like an ordinary member function, but the name is prefixed with the keyword operator.

C++ places a number of restrictions on operator overloading. Only the pre-defined set of C++ operators may be overloaded. It is illegal to define a new operator and then overload it. The arity of an operator must also be preserved. You cannot turn a unary operator into a binary operator or vice versa. Only the operators +, -, *, &, ++, and -- have both postfix and prefix forms. Also, the following operators cannot be overloaded: scope operator (`::'), member object selection operator (`.*'), class object selector operator (`.'), and the arithmetic if operator (`?:'). Operators are also not allowed to have default arguments.

When the compiler encounters an operator it is converted into an appropriate function call. However, non-member functions are interpreted differently than member functions. For any binary operator @, x @ y is interpreted as x.operator@(y) as a member and as operator@(x, y) as a non-member.

Suppose we define a nom-member addition operator for complex numbers:

```
class Complex
{
   // ...
public:
   Complex (double re=0.0,
            double im= 0.0);
   // ...
   Complex operator+(const Complex& a);
   // ...
};
```

As a non-member, the addition of two Complex numbers b and c, b + c, is interpreted as b.operator+ (c). This is fine because the left operand is of Complex type. However, what happens when we try to add a double with a Complex number:

```
Complex c(0.0);
Complex d = c    + 10.0;   // OK
Complex e = 10.0 + c;      // Error
```

The first assignment, is interpreted as c.operator+(10.0) which compiles because the Complex constructor has default arguments and enables the compiler to implicitly convert the 10.0 from a double to a Complex before the addition is performed. However, the compiler will treat the second assignment as (10.0).operator+(c). This fails to compile because the compiler will not implicitly convert the left operand (10.0) to a Complex before applying the operator. When the addition operator is made a non-member, as:

```
friend Complex operator+
   (const Complex& a, const Complex& b);
```

type conversions can occur on both sides of the operator and will enable both statements to compile. Remember that non-member functions are defined with two arguments, because they do not have the implicit this pointer that member functions have.

For any prefix unary operator @, @x is interpreted as x.operator@() as a member and as operator@(x) as a non-member. A postfix operator @, x@ is interpreted just like @x, but the compiler generates the code so that the operator is applied after x is evaluated. Unary operators are almost always made members because we do not want

type conversions to occur. As an example, we define unary minus for the Complex class a member:

```
class Complex
{
   double re, im
   // ...
public:
   // ...
   Complex operator-() const
      { return Complex(-re, -im); }
   // ...
};
```

Sometimes it is necessary to have a postfix and prefix form of a unary operator. We have all seen how the increment operator supports both prefix (e.g., ++x) and postrix (e.g., x++). Suppose we have our own class of integers called MyInt and that we have defined a prefix increment operator as a member and a prefix decrement operator as a non-member:

```
class MyInt
{
   // ...
public:
   // prefix operators
   MyInt operator++();
   friend MyInt operator--(MyInt&);
};
```

How do we define a postfix version of the increment and decrement operators as well? After all they both have the same signature. Different functions must have different signatures (arguments and return types) to be distinguishable by the compiler. C++ uses an int as the second argument to distinguish between prefix and postfix forms:

```
class MyInt
{
public:
   // prefix operators
   MyInt operator++(int);
```

```
   friend MyInt operator--(MyInt&,int);
};
```

The additional int argument is automatically supplied by the compiler and is transparent to users.

When should an operator be made a member or non-member function? Here are some guidelines to help determine whether to make an operator a member or non-member function:

- The assignment (`='), subscript (`[]'), and call (`()') oper- ators must be defined as class member functions. C++ makes it illegal to make these operators non-members.
- If the left-operand of an operator must be an object of the class type, then the operator should be a member.
- If the left-operand of an operator may not be an object of the class type, then the operator must be a non-member.
- Most binary arithmetic operators are best made non-members in order to gain the advantage of having implicit type con- versions performed on the left operand.
- For most unary operators, it is desirable to suppress implicit type conversion, so they are made member functions.

In the last two issues of ObjectiveViewPoint we have looked at how C++ supports the object-oriented paradigm. You now have all the tools you need to begin implementing your own class hierarchies. So, give it a shot! If you have been hesitant to use object-oriented techniques because you don't want to learn yet another programming language, remember that object-orientation provides a whole new way of thinking about problems. You may wish to begin by learning object-oriented analysis and design. I promise you will find many new exciting ways to design your software systems. As always send questions and comments to wiseb@cs.rpi.edu for possible inclusion in the next issues Grab Bag.