1. Polymorphism in C++

1.1 Polymorphism and virtual functions

- Polymorphism - many forms
- In C++, polymorphism is implemented through virtual functions. Virtual functions (and so, of course, polymorphism) have a meaning only in the context of inheritance.
- Virtual functions provide the ability to apply true object-oriented programming in C++
- Virtual functions deal with decoupling in terms of *types*.
- Inheritance allows the treatment of an object as its own type *or* its base type. This ability is critical because it allows many types (derived from the same base type) to be treated as if they were one type.
- The virtual function allows one type to express its distinction from another; similar type as long as they're both derived from the same base type.

"If you don't use virtual functions, you don't understand OOP yet."

Bruce Eckel

1.2 Function call binding

- Connecting a function call to a function body is called *binding*. When binding is performed before the program is run (by the compiler and linker), it's called *early binding*.

Early binding presents a problem with upcasting as we saw from a previous example:

```
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument
{
  public:
    void play(note) const { cout << "Instrument::play()"; }
};

class Wind : public Instrument
{
  public:
    void play(note) const { cout << "Wind::play()"; }

};

void tune(Instrument& i)
{
  // ...
  i.play(middleC);

}

void main()
{
  Wind w;
  tune(w) // Upcast, no explicit cast needed
}
```

Here the base-class version of play(), i.e Instrument::play(), will be called, due to early binding.

- The solution is late binding. Late binding, dynamic binding and runtime binding are synonyms and all three of these are often used interchangeably.

- When implementing late binding, there must be some mechanism to determine the type of the object at runtime.

- The late-binding mechanism varies from language to language, but some sort of type information must be "installed" in the objects.

1.3 Virtual functions

- To cause late binding to occur for a particular function, C++ requires that you use the **virtual** keyword when declaring the function in the base class.

- Only the declaration needs the **virtual** keyword, not the definition.

- If a function is declared as **virtual** in the base class, it is **virtual** in all the derived classes.

- The redefinition of a **virtual** function in a derived class is usually called *overriding*.

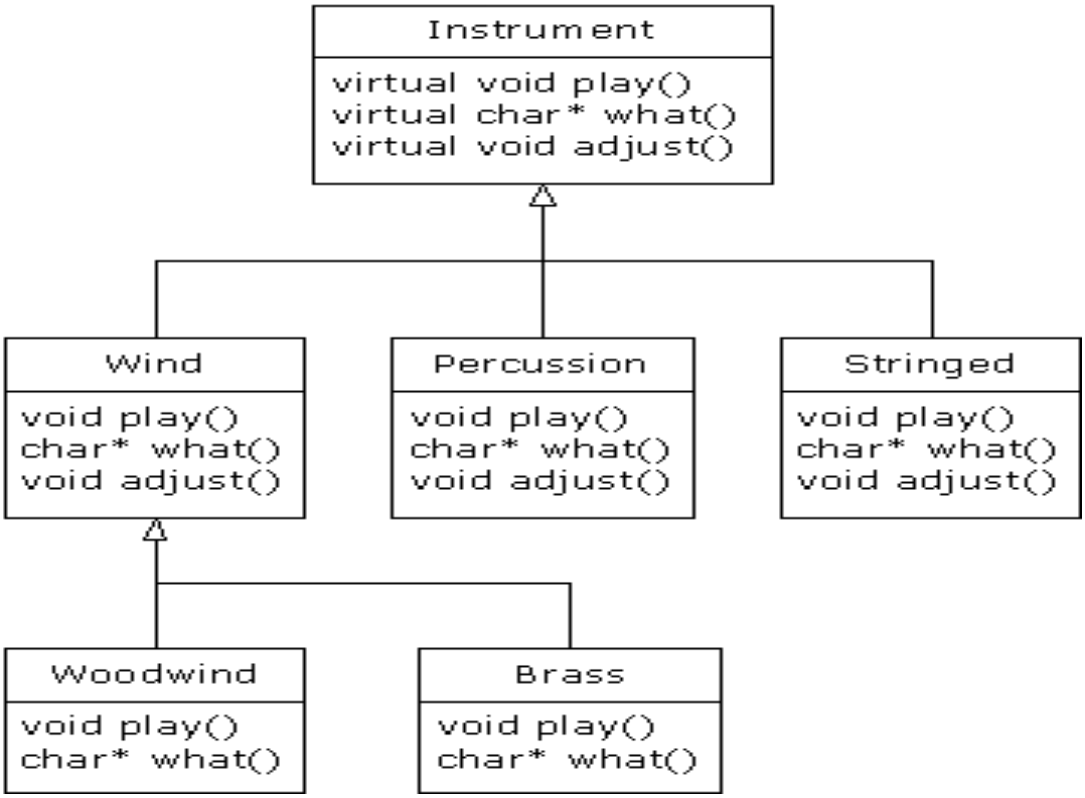
To get the desired behavior from our prevoius example, simply add the **virtual** keyword in the base class before **play(**):

```
class Instrument
{
  public:
    virtual void play(note) const { cout << "Instrument::play()"; }
};
```

- With **play( )** defined as **virtual** in the base class, you can add as many new types as you want without changing the **tune( )** function.

- In a well-designed OOP program, most or all of your functions will follow the model of **tune()** and communicate only with the base-class interface.

- This provides for extensibility because you can add new functionality by inheriting new data types from the common base class.



- The **virtual** function is the lens to use when you're trying to analyze a project: Where should the base classes occur, and how might you want to extend the program?

1.4 How C++ implements late binding

- The keyword **virtual** tells the compiler it should not perform early binding. Instead, it should automatically install all the mechanisms necessary to perform late binding.

- This means that if you call **play( )** for a **Brass** object *through an address for the base-class* **Instrument**, you'll get the proper function.

Compilers may implement virtual behavior any way they want, but the way it's described here is an almost universal approach.

Let's make an array A containing references to different instruments:

```
Instrument* A[] =
{
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};
```

1. The compiler creates a single table (called the VTABLE) for each class that contains **virtual** functions.

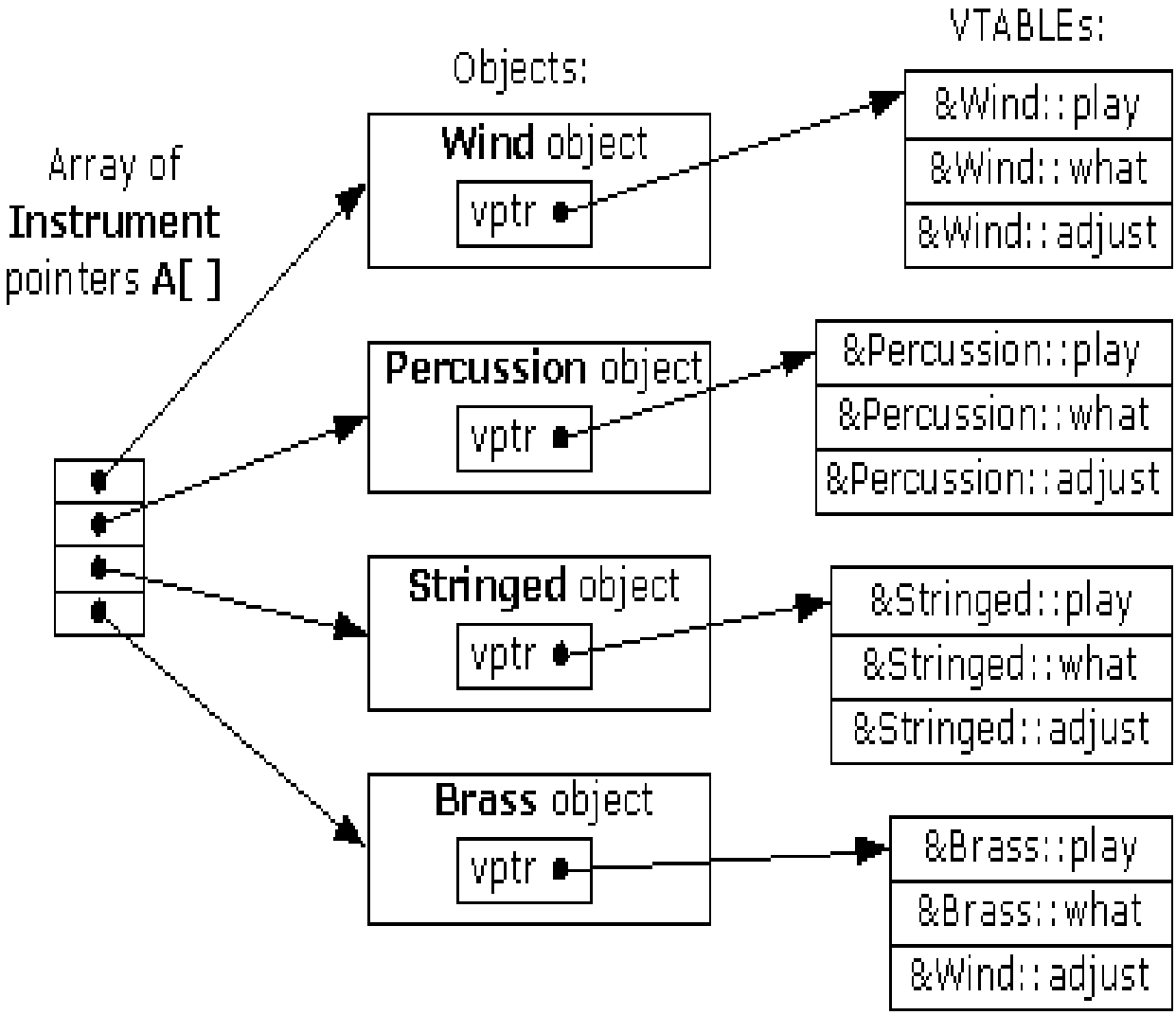
2. The compiler places the addresses of the virtual functions for that particular class in the VTABLE.

3. In each class with virtual functions, it secretly places a pointer, called the *vpointer* (abbreviated as VPTR), which points to the VTABLE for that object.

- When you make a virtual function call through a base-class pointer (that is, when you make a polymorphic call), the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the correct function and causing late binding to take place.

- All of this happens automatically

To make things a bit more visual, below is a drawing of the array of pointers A[]:



- When making a call to a virtual function the compiler begins with the **Instrument** pointer, which points to the starting address of the object. All **Instrument** objects or objects derived from **Instrument** have their VPTR in the same place, so the compiler can pick the VPTR out of the object.

- The VPTR points to the starting address of the VTABLE. All the VTABLE function addresses are laid out in the same order, regardless of the specific type of the object. **play()** is first, **what( )** is second, and **adjust( )** is third.

- For example, the **adjust( )** function is at the location VPTR+2. Thus, instead of saying, "Call the function at the absolute location **Instrument::adjust** " (early binding; the wrong action), it generates code that says, in effect, "Call the function at VPTR+2."

- It's important to realize that utilizing polymorphism through upcasting deals only with addresses. If the compiler has an object, it knows the exact type and therefore (in C++) will not use late binding for any function calls – or at least, the compiler doesn't *need* to use late binding.

1.4.1 Why do I have to know at all about virtual functions if it is such an important feature? Why isn't all this going on automatically behind the scenes making my life easier?

- Well, because it is not quite as efficient. Instead of one simple CALL to an absolute address, there are two – more sophisticated – assembly instructions required to set up the virtual function call.

- Stroustrup stated that his guideline was, “If you don’t use it, you don’t pay for it.” Stoustrup never stated that this has anything to do with keeping C programmers hostile to C++ happy. But, it is not unusual for a fanatic C programmer to take every chance to avoid other languages with the argument, "It's not quite as efficient".

- However, once again, if you are going to use polymorphism, use it everywhere you get the chance. Remember that the penalty in performance is relative to the overall design. Usually, you can afford this penalty in performance.

1.5 Abstract base classes and virtual functions

- Often in a design, you want the base class to present *only* an interface for its derived classes. That is, you don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used.
  This is accomplished by making that class *abstrac*t, which happens if you give it at least one *pure virtual function*.

The syntax for a pure virtual declaration is:

```
virtual void f() = 0;
```

By doing this, you tell the compiler to reserve a slot for a function in the VTABLE, but not to put an address in that particular slot. Even if only one function in a class is declared as pure virtual, the VTABLE is incomplete.

- When an abstract class is inherited, all pure virtual functions must be implemented, or the inherited class becomes abstract as well.

Remember the previously shown inheritance hierarchy using the Instrument class as a baseclass common to all subclasses. The instrument class is there only to create a common interface for all of its subclasses Thus it doesn't make much sense creating an object that is only an instrument, and so you will want to prevent users from doing this.
Below is an implementation of the Instrument inheritance hierarchy. Because the Instrument class has nothing but pure virtual functions, we call it a *pure abstract clas*s:

```
class Instrument
{
  public:
    // Pure virtual functions:
    virtual void play(note) const = 0;
    virtual char* what() const = 0;
    // Assume this will modify the object:
     virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class Wind : public Instrument
{
  public:
    void play(note) const
    {
      cout << "Wind::play" << endl;
    }
    char* what() const { return "Wind"; }
    void adjust(int) {}
};

class Percussion : public Instrument
{
  public:
    void play(note) const
    {
      cout << "Percussion::play" << endl;
    }
    char* what() const { return "Percussion"; }
    void adjust(int) {}
};

class Stringed : public Instrument
{
  public:
    void play(note) const
    {
      cout << "Stringed::play" << endl;
    }
    char* what() const { return "Stringed"; }
    void adjust(int) {}
};

class Brass : public Wind
{
  public:
    void play(note) const
    {
      cout << "Brass::play" << endl;
    }
    char* what() const { return "Brass"; }
};

class Woodwind : public Wind
{
  public:
    void play(note) const
    {
      cout << "Woodwind::play" << endl;
    }
    char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i)
{
  // ...
  i.play(middleC);
}
```

```
// New function:
void f(Instrument& i) { i.adjust(1); }

int main()
{
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

1.6 Pure virtual definitions

It's possible to provide a definition for a pure virtual function in the base class. You're still telling the compiler not to allow objects of that abstract base class, and the pure virtual functions must still be defined in derived classes in order to create objects.

```
class Pet
{
  public:
    virtual void speak() const = 0;
    virtual void eat() const = 0;
    // Inline pure virtual definitions are illegal:
    //! virtual void sleep() const = 0 {}
};

// OK, not defined inline
void Pet::eat() const
{
  cout << "Pet::eat()" << endl;
}

void Pet::speak() const
{
  cout << "Pet::speak()" << endl;
}

class Dog : public Pet
{
  public:
    // Use the common Pet code:
    void speak() const { Pet::speak(); }
    void eat() const { Pet::eat(); }
};

int main()
{
  Dog simba; // Richard's dog
  simba.speak();
  simba.eat();
}
```

Benefits:

- Gathering a common piece of code in one place
- Allows you to change from an ordinary virtual to a pure virtual without disturbing the existing code.

1.7 What happens when you inherit and add new virtual functions in the derived class?

Here's a simple example:

```
class Pet
{
  public:
    Pet(const string& petName) : pname(petName) {}
    virtual string name() const { return pname; }
    virtual string speak() const { return ""; }
  private:
    string pname;
};

class Dog : public Pet
{
  public:
    Dog(const string& petName) : Pet(petName) {}
    // New virtual function in the Dog class:
    virtual string sit() const
    {
      return Pet::name() + " sits";
    }
    string speak() const   // Override
    {
      return Pet::name() + " says 'Bark!'";
    }
  private:
    string name;
};

int main()
{
  Pet* p[] = {new Pet("generic"),new Dog("bob")};
  cout << "p[0]->speak() = " << p[0]->speak() << endl;
  cout << "p[1]->speak() = " << p[1]->speak() << endl;
  cout << "p[1]->sit() = " << p[1]->sit() << endl; // Illegal
}
```

The `p[1]->sit()`, function call can be made possible only through a downcast:

```
((Dog*)p[1])->sit()
```

- If your problem is set up so that you must know the exact types of all objects, you should do some rethinking, because you're probably not using virtual functions (polymorphism) properly. Downcasts are dangerous because of the possibility of turning an object into something that it is not.

1.8 Object slicing

- There is a distinct difference between passing the addresses of objects and passing objects by value when using polymorphism.

- If you upcast to an object instead of a pointer or reference to the object, the object is "sliced" until all that remains is the subobject that corresponds to the destination type of your cast.

- Pure virtual functions prevent an abstract class from being passed into a function *by valu*e. Thus, it is also a way to prevent *object slicing*. By making a class abstract, you can ensure that a pointer or reference is always used during upcasting to that class.

- One of the most important aspects of pure virtual functions is to prevent object slicing by generating a compile-time error message if someone tries to do it.

1.9 Overloading and overriding

This is best illustrated through an example:

```
class Base
{
  public:
    virtual int f() const
    {
      cout << "Base::f()\n";
      return 1;
    }
    virtual void f(string) const { }
    virtual void g() const {}
};

class Derived1 : public Base
{
  public:
    void g() const {}
};

class Derived2 : public Base
{
  public:
    // Overriding a virtual function:
    int f() const
    {
      cout << "Derived2::f()\n";
      return 2;
    }
};

class Derived3 : public Base
{
  public:
    // Cannot change return type:
    // void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base
{
  public:
    // Change argument list: not really overriding
    int f(int) const
    {
      cout << "Derived4::f()\n";
      return 4;
    }
};
```

```
void main()
{
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
  //! d2.f(s); // string version hidden
  Derived4 d4;
  x = d4.f(1);
  //! x = d4.f(); // f() version hidden
  //! d4.f(s); // string version hidden
  Base& br = d4; // Upcast
  //! br.f(1); // Derived version unavailable
  br.f(); // Base version available
  br.f(s); // Base version abailable
}
```

- This example means to illustrate what happens when overriding/redefining functions in a baseclass that are overloaded. Anyone who would use a baseclass like this and manipulate it in these ways in derived classes **should be shot**.

- The example is here for you to understand what is going on when you are sitting down having to understand what some moron-programmer did.

To summarize:

- The compiler will not allow you to change the return type of an overridden function (it will allow it if **f( )** is not virtual). This is an important restriction because the compiler must guarantee that you can polymorphically call the function through the base class, and if the base class is expecting an **int** to be returned from **f(**), then the derived-class version of **f( )** must keep that contract or else things will break.

- If you override/redefine one of the overloaded member functions in the base class, the other overloaded versions become hidden in the derived class.

1.10 Behavior of virtual functions inside constructors

- If you call a virtual function inside a constructor, only the local version of the function is used. That is, the virtual mechanism doesn't work within the constructor.

1.11 Virtual destructors

- Virtual destructors works exactly the same way as an ordinary virtual function
- You should therefore always supply a virtual destructor for a baseclass.

1.12 Pure virtual destructors

- While pure virtual destructors are legal in Standard C++. There is an added constraint when using them: you must provide a function body for the pure virtual destructor.
- All destructors in a class hierarchy are always called. If you *could* leave off the definition for a pure virtual destructor, what function body would be called during destruction?
- The only difference you'll see between the pure and non-pure virtual destructor is that the pure virtual destructor does cause the base class to be abstract, so you cannot create an object of the base class.
- The only distinction between a pure virtual destructor and a virtual destructor happens when the destructor is **the only** pure virtual function.
- Unlike every other pure virtual function, you are *not* required to provide a definition of a pure virtual destructor in the derived class. Why?

1.13 Virtuals in destructors

- Inside a destructor, only the "local" version of the member function is called; the virtual mechanism is ignored.
- Why? The actual function called would (in many cases) rely on portions of an object that have *already been destroye*d!