Composition:

You simply create objects of your existing class inside the new class. This is called composition because the new class is

Composed of objects of existing classes.

Y contains a subobject of X: As per useful.cpp Both member objects and base class storage are referred to as subobjects.

The constructor initializer list:

The solution is simple: Call the constructor for the subobject. C++ provides a special syntax for this, the constructor initializer list. The form of the constructor initializer

list echoes the act of inheritance. With inheritance, you put the base classes after a colon and before the opening brace of the class body. In the constructor initializer list, you put the calls to subobject constructors after the constructor argument list and a colon, but before the opening brace of the function body. For a class MyType, inherited from Bar, this might look like this:

MyType::MyType(int i) : Bar(i) { // ...

if Bar has a constructor that takes a single int argument.

Member object initialization

It turns out that you use this very same syntax for member object initialization when using composition.

For composition, you give the names of the objects instead of the class names. If you have more than one

constructor call in the initializer list, you separate the calls with commas:

MyType2::MyType2(int i) : Bar(i), m(i+1) { // ...

This is the beginning of a constructor for class MyType2, which is inherited from Bar and contains a member

object called m. Note that while you can see the type of the base class in the constructor initializer list,

you only see the member object identifier.

Overriding

It’s worth emphasizing that constructors and destructors are quite unusual in that every one in the hierarchy is called,

whereas with a normal member function only that function is called, but not any of the base-class versions.

If you also want to call the base-class version of a normal member function that you’re overriding,

you must do it explicitly.

NameHiding

If you inherit a class and provide a new definition for one of its member functions, there are two possibilities.

The first is that you provide the exact signature and return type in the derived class definition as in the base class

definition. This is called redefining for ordinary member functions and overriding when the base class member function is a virtual function (virtual functions are the normal case, and will be covered in detail in Chapter 15).

But what happens if you change the member function argument list or return type in the derived class? Here’s an example:

NameHiding.cpp

Functions that don’t automatically inherit//TODO

1.constructors and destructors don’t inherit and must be created specially for each derived class.

2.In addition, the operator= doesn’t inherit because it performs a constructor-like activity. That is, just because you know how to assign all the members of an object on the left-hand side of the = from an object on the right-hand side doesn’t

mean that assignment will still have the same meaning after inheritance.

Inheritance and static member functions

static member functions act the same as non-static member functions:

1. They inherit into the derived class.

2. If you redefine a static member, all the other overloaded functions in the base class are hidden.//TODO

3.If you change the signature of a function in the base class, all the base class versions with that function name are hidden

(this is really a variation of the previous point).

4. Static Member Function can be not be virtual

Choosing composition vs. inheritance

Composition is generally used when you want the features of an existing class inside your new class, but not its

interface. That is, you embed an object to implement features of your new class, but the user of your new class

sees the interface you’ve defined rather than the interface from the original class. To do this, you follow the typical

path of embedding private objects of existing classes inside your new class.

Occasionally, however, it makes sense to allow the class user to directly access the composition of your new class,

that is, to make the member objects public. The member objects use access control themselves, so this is a safe thing

to do and when the user knows you’re assembling a bunch of parts, it makes the interface easier to understand.

A Car class is a good example:car.cpp

private inheritance:

You may wonder what the purpose of private inheritance is, because the alternative of using composition to

create a private object in the new class seems more appropriate. private inheritance is included in the

language for completeness, but if for no other reason than to reduce confusion, you’ll usually want to use

composition rather than private inheritance. However, there may occasionally be situations where you want to

produce part of the same interface as the base class and disallow

the treatment of the object as if it were a base-class object. private inheritance provides this ability.

Publicizing privately inherited members

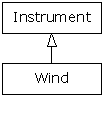
When you inherit privately, all the public members of the base class become private. If you want any of them to

be visible, just say their

names (no arguments or return values) along with the using keyword in the public section of the derived class:

So Private Inheritance is useful if I want to hide a part o functionality of the class

### Upcasting:Why “upcasting?”



Casting from derived to base moves *up* on the inheritance diagram, so it’s commonly referred to as upcasting. Upcasting is always safe because you’re going from a more specific type to a more general type – the only thing that can occur to the class interface is that it can lose member functions, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

Ex CopyConstructor.cpp

## Upcasting

an object can be used as its own type or as an object of its base type. In addition, it can be manipulated through an address of the base type. Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type is called *upcasting* because of the way inheritance trees are drawn with the base class at the top.

Example Upcasting.cpp:Modified

The function **tune( )** accepts (by reference) an **Instrument**, but also without complaint anything derived from **Instrument**. In **main( )**, you can see this happening as a **Wind** object is passed to **tune( )**, with nocast necessary. This is acceptable; the interface in **Instrument** must exist in **Wind**, because **Wind** is publicly inherited from **Instrument**. Upcasting from **Wind** to **Instrument** may “narrow” that interface, but never less than the full interface to **Instrument**.

The same arguments are true when dealing with pointers; the only difference is that the user must explicitly take the addresses of objects as they are passed into the function.

## The problem

The problem with **Instrument2.cpp** can be seen by running the program. The output is **Instrument::play**. This is clearly not the desired output, because you happen to know that the object is actually a **Wind** and not just an **Instrument**. The call should produce **Wind::play**. For that matter, any object of a class derived from **Instrument** should have its version of **play( )** used, regardless of the situation.

The behavior of **Instrument2.cpp** is not surprising, given C’s approach to functions. To understand the issues, you need to be aware of the concept of *binding*.

### 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*. You may not have heard the term before because it’s never been an option with procedural languages: C compilers have only one kind of function call, and that’s early binding.

The problem in the program above is caused by early binding because the compiler cannot know the correct function to call when it has only an **Instrument** address.

The solution is called *late binding*, which means the binding occurs at runtime, based on the type of the object. Late binding is also called *dynamic binding* or *runtime binding*. When a language implements late binding, there must be some mechanism to determine the type of the object at runtime and call the appropriate member function. In the case of a compiled language, the compiler still doesn’t know the actual object type, but it inserts code that finds out and calls the correct function body. The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects. You’ll see how this works later.

## virtual functions

To create a member function as **virtual**, you simply precede the declaration of the function with the keyword **virtual**. 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**.*

## 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.

To accomplish this, the typical compiler[[54]](http://www.drbio.cornell.edu/pl47/programming/TICPP-2nd-ed-Vol-one-html/Chapter15.html" \l "fn54) creates a single table (called the VTABLE) for each class that contains **virtual** functions. The compiler places the addresses of the virtual functions for that particular class in the VTABLE.

 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 – setting up the VTABLE for each class, initializing the VPTR, inserting the code for the virtual function call – happens automatically, so you don’t have to worry about it. With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.

Explain WITH Example : Sizes.cpp

With no virtual functions, the size of the object is exactly what you’d expect: the size of a single **int**. With a single virtual function in **OneVirtual**, the size of the object is the size of **NoVirtual** plus the size of a **void** pointer. It turns out that the compiler inserts a single pointer (the VPTR) into the structure if you have one *or more* virtual functions. There is no size difference between **OneVirtual** and **TwoVirtuals**. That’s because the VPTR points to a table of function addresses. You need only one table because all the virtual function addresses are contained in that single table.

### Installing the vpointer

Because the VPTR determines the virtual function behavior of the object, you can see how it’s critical that the VPTR always be pointing to the proper VTABLE. You don’t ever want to be able to make a call to a virtual function before the VPTR is properly initialized. Of course, the place where initialization can be guaranteed is in the constructor, but none of the **Instrument** examples has a constructor.

This is where creation of the default constructor is essential. In the **Instrument** examples, the compiler creates a default constructor that does nothing except initialize the VPTR. This constructor, of course, is automatically called for all **Instrument** objects before you can do anything with them, so you know that it’s always safe to call virtual functions.

The implications of the automatic initialization of the VPTR inside the constructor are discussed in a later section.

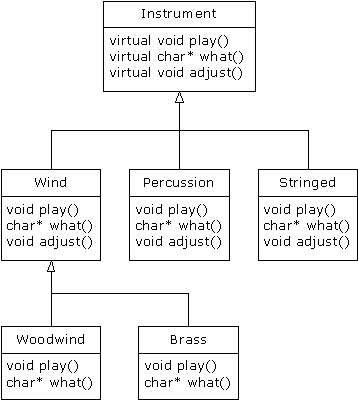
If Compiler knows the type of object then in any case(Virtual Fucntion presence ) it will perform the early binding

## Abstract base classes and pure virtual functions:

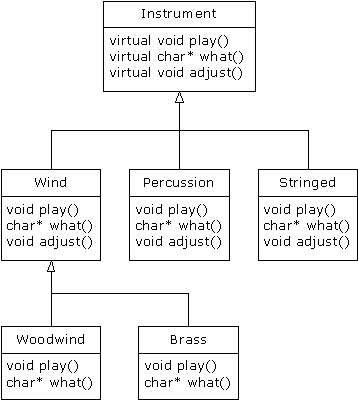
Often in a design, you want the base class to present only an interface for its derived classes. That is, you don’t want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by making that class abstract, which happens if you give it at least one pure virtual function. You can recognize a pure virtual function because it uses the **virtual** keyword and is followed by **= 0**. If anyone tries to make an object of an abstract class, the compiler prevents them. This is a tool that allows you to enforce a particular design.

When an abstract class is inherited, all pure virtual functions must be implemented, or the inherited class becomes abstract as well. Creating a pure virtual function allows you to put a member function in an interface without being forced to provide a possibly meaningless body of code for that member function. At the same time, a pure virtual function forces inherited classes to provide a definition for it.

In all of the instrument examples, the functions in the base class **Instrument** were always “dummy” functions. If these functions are ever called, something is wrong. That’s because the intent of **Instrument** is to create a common interface for all of the classes derived from it.



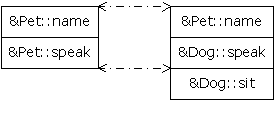
Note that pure virtual functions prevent an abstract class from being passed into a function *by value*. Thus, it is also a way to prevent *object slicing* (which will be described shortly). By making a class abstract, you can ensure that a pointer or reference is always used during upcasting to that class.



## Inheritance and the VTABLE

You can imagine what happens when you perform inheritance and override some of the virtual functions. The compiler creates a new VTABLE for your new class, and it inserts your new function addresses using the base-class function addresses for any virtual functions you don’t override. One way or another, for every object that can be created (that is, its class has no pure virtuals) there’s always a full set of function addresses in the VTABLE, so you’ll never be able to make a call to an address that isn’t there (which would be disastrous).

The class **Pet** contains a two virtual functions: **speak( )** and **name( )**. **Dog** adds a third virtual function called **sit( )**, as well as overriding the meaning of **speak( )**. A diagram will help you visualize what’s happening. Here are the VTABLEs created by the compiler for **Pet** and **Dog**:



Notice that the compiler maps the location of the **speak( )** address into exactly the same spot in the **Dog**VTABLE as it is in the **Pet** VTABLE. Similarly, if a class **Pug** is inherited from **Dog**, its version of **sit( )** would be placed in its VTABLE in exactly the same spot as it is in **Dog**. This is because (as you saw with the assembly-language example) the compiler generates code that uses a simple numerical offset into the VTABLE to select the virtual function. Regardless of the specific subtype the object belongs to, its VTABLE is laid out the same way, so calls to the virtual functions will always be made the same way.

In this case, however, the compiler is working only with a pointer to a base-class object. The base class has only the **speak( )** and **name( )**functions, so those is the only functions the compiler will allow you to call. How could it possibly know that you are working with a **Dog** object, if it has only a pointer to a base-class object? That pointer might point to some other type, which doesn’t have a **sit( )** function. It may or may not have some other function address at that point in the VTABLE, but in either case, making a virtual call to that VTABLE address is not what you want to do. So the compiler is doing its job by protecting you from making virtual calls to functions that exist only in derived classes.

There are some less-common cases in which you may know that the pointer actually points to an object of a specific subclass. If you want to call a function that only exists in that subclass, then you must cast the pointer. You can remove the error message produced by the previous program like this:

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

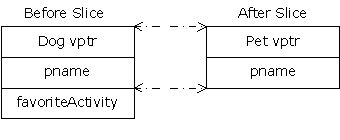
Here, you happen to know that **p[1]** points to a **Dog** object, but in general you don’t know that. If your problem is set up so that you must know the exact types of all objects, you should rethink it, because you’re probably not using virtual functions properly. However, there are some situations in which the design works best (or you have no choice) if you know the exact type of all objects kept in a generic container. This is the problem of run-time type identification (RTTI)**.**

RTTI is all about casting base-class pointers down to derived-class pointers (“up” and “down” are relative to a typical class diagram, with the base class at the top). Casting up happens automatically, with no coercion, because it’s completely safe. Casting down is unsafe because there’s no compile time information about the actual types, so you must know exactly what type the object is. If you cast it into the wrong type, you’ll be in trouble.

### Object slicing

Example: ObjectSlicing.cpp

IMG_256



You’re saved from disaster because the object is being passed by value. Because of this, the compiler knows the precise type of the object because the derived object has been forced to become a base object. When passing by value, the copy-constructor for a **Pet**object is used, which initializes the VPTR to the **Pet** VTABLE and copies only the **Pet** parts of the object. There’s no explicit copy-constructor here, so the compiler synthesizes one. Under all interpretations, the object truly becomes a **Pet** during slicing.

Object slicing actually removes part of the existing object as it copies it into the new object, rather than simply changing the meaning of an address as when using a pointer or reference. Because of this, upcasting into an object is not done often; in fact, it’s usually something to watch out for and prevent. Note that, in this example, if **description( )** were made into a pure virtual function in the base class (which is not unreasonable, since it doesn’t really do anything in the base class), then the compiler would prevent object slicing because that wouldn’t allow you to “create” an object of the base type (which is what happens when you upcast by value). This could be the most important value of pure virtual functions: to prevent object slicing by generating a compile-time error message if someone tries to do it.

## Overloading & overriding

Virtual function restrict Overloading

//TODO:Returning a pointer or reference to a derived// type during overriding

## virtual functions & constructors: ->

### **Order of constructor calls**

### The second interesting facet of constructors and virtual functions concerns the order of constructor calls and the way virtual calls are made within constructors.

All base-class constructors are always called in the constructor for an inherited class. This makes sense because the constructor has a special job: to see that the object is built properly. A derived class has access only to its own members, and not those of the base class. Only the base-class constructor can properly initialize its own elements. Therefore it’s essential that all constructors get called; otherwise the entire object wouldn’t be constructed properly. That’s why the compiler enforces a constructor call for every portion of a derived class. It will call the default constructor if you don’t explicitly call a base-class constructor in the constructor initializer list. If there is no default constructor, the compiler will complain.

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any **public** and **protected** members of the base class. This means you must be able to assume that all the members of the base class are valid when you’re in the derived class. In a normal member function, construction has already taken place, so all the members of all parts of the object have been built. Inside the constructor, however, you must be able to assume that all members that you use have been built. The only way to guarantee this is for the base-class constructor to be called first. Then when you’re in the derived-class constructor, all the members you can access in the base class have been initialized. “Knowing all members are valid” inside the constructor is also the reason that, whenever possible, you should initialize all member objects (that is, objects placed in the class using composition) in the constructor initializer list. If you follow this practice, you can assume that all base class members *and* member objects of the current object have been initialized.

This is not the case. 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.

## Destructors and virtual destructors

You cannot use the **virtual**keyword with constructors, but destructors can and often must be virtual.

The constructor has the special job of putting an object together piece-by-piece, first by calling the base constructor, then the more derived constructors in order of inheritance (it must also call member-object constructors along the way). Similarly, the destructor has a special job: it must disassemble an object that may belong to a hierarchy of classes. To do this, the compiler generates code that calls all the destructors, but in the *reverse* order that they are called by the constructor. That is, the destructor starts at the most-derived class and works its way down to the base class. This is the safe and desirable thing to do because the current destructor can always know that the base-class members are alive and active. If you need to call a base-class member function inside your destructor, it is safe to do so. Thus, the destructor can perform its own cleanup, then call the next-down destructor, which will perform *its* own cleanup, etc. Each destructor knows what its class is derived *from*, but not what is derived from it.

You should keep in mind that constructors and destructors are the only places where this hierarchy of calls must happen (and thus the proper hierarchy is automatically generated by the compiler). In all other functions, only *that* function will be called (and not base-class versions), whether it’s virtual or not. The only way for base-class versions of the same function to be called in ordinary functions (virtual or not) is if you *explicitly* call that function.

Normally, the action of the destructor is quite adequate. But what happens if you want to manipulate an object through a pointer to its base class (that is, manipulate the object through its generic interface)? This activity is a major objective in object-oriented programming. The problem occurs when you want to **delete** a pointer of this type for an object that has been created on the heap with **new**. If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during **delete**. Sound familiar? This is the same problem that virtual functions were created to solve for the general case. Fortunately, virtual functions work for destructors as they do for all other functions except constructors.

Example : VirtualDestructors.cpp

When you run the program, you’ll see that **delete bp** only calls the base-class destructor, while **delete b2p** calls the derived-class destructor followed by the base-class destructor, which is the behavior we desire. Forgetting to make a destructor **virtual** is an insidious bug because it often doesn’t directly affect the behavior of your program, but it can quietly introduce a memory leak. Also, the fact that *some* destruction is occurring can further mask the problem.

Even though the destructor, like the constructor, is an “exceptional” function, it is possible for the destructor to be virtual because the object already knows what type it is (whereas it doesn’t during construction). Once an object has been constructed, its VPTR is initialized, so virtual function calls can take place.

### Virtuals in destructors

If you’re inside an ordinary member function and you call a virtual function, that function is called using the late-binding mechanism. This is not true with destructors, virtual or not. Inside a destructor, only the “local” version of the member function is called; the virtual mechanism is ignored

## Downcasting

 when you upcast you are always clearly derived from an ancestor class (typically only one, except in the case of multiple inheritance) but when you downcast there are usually several possibilities that you could cast to. More specifically, a **Circle** is a type of **Shape** (that’s the upcast), but if you try to downcast a **Shape** it could be a **Circle**, **Square**,**Triangle**, etc. So the dilemma is figuring out a way to safely downcast.

C++ provides a special *explicit cast* (introduced in Chapter 3) called **dynamic\_cast** that is a *type-safe downcast* operation. When you use **dynamic\_cast** to try to cast down to a particular type, the return value will be a pointer to the desired type only if the cast is proper and successful, otherwise it will return zero to indicate that this was not the correct type. Here’s a minimal example:

Example: DynamicCast.cpp

## Summary

Polymorphism – implemented in C++ with virtual functions – means “different forms.” In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the virtual functions.

## 