C++ Programming: Session 7

Anupama Chandrasekhar (Sep 13 2018)

Derived Classes

- The notion of derived class is provided to express hierarchical relationships and to express commonality between classes.
- The language features support building new classes from existing one:
  - **Implementation inheritance**: To save implementation effort by sharing facilities provided by a base class
  - **Interface inheritance**: To allow different derived classes to be used interchangeably through the interface provided by a common base (run-time polymorphism)
  - **Compile time polymorphism (static polymorphism):** Think templates.

Derived Classes : Why?

```
struct Employee {
        string first_name, family_name;
        char middle_initial;
        Date hiring_date;
        short department;
         //...
};

//Solution 1
struct Manager {
        Employee emp; // manager's employee record
        list<Employee*> group;  // people managed
        short level;
         //...
};
```

A **Manager** is also an **Employee**, the **Employee** data is stored in the emp member of the **Manager** object. There is nothing here though that tells the compiler that a **Manager** is also an **Employee**. A **Manager*** is not an **Employee***

To put a **Manager** into a list of **Employee**s we need to either use explicit type conversion of **Manager*** or put the **Employee** member of the **Manager** object.

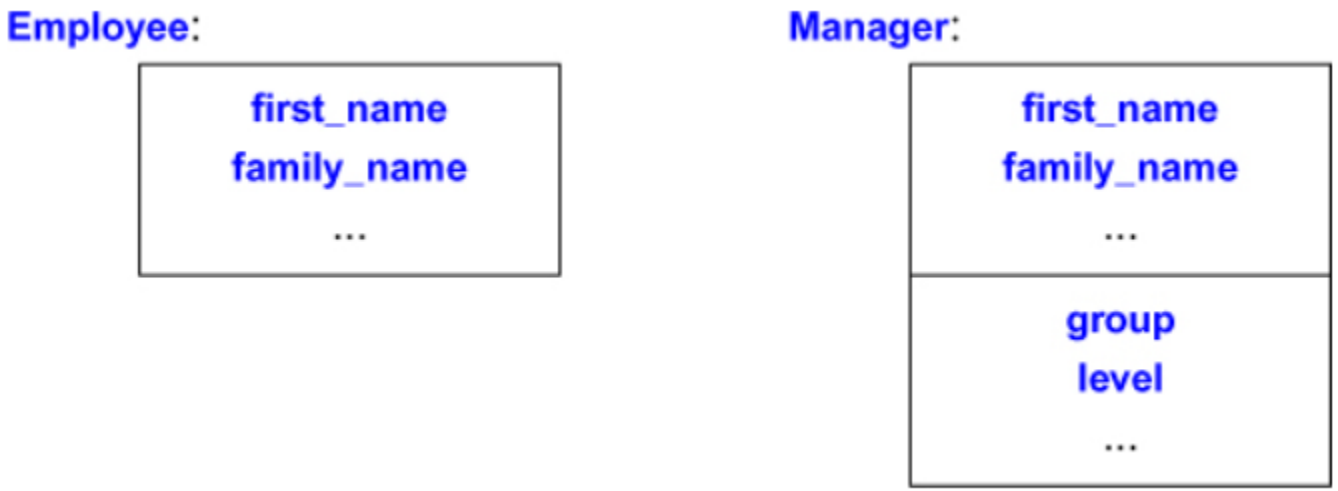
Derived Classes

- The correct approach is to explicitly state that a **Manager** is an **Employee.**

```
struct Manager : public Employee {
        list<Employee*> group;
        short level;
         //...
};
```

- This explicitly states that **Manager** is derived from **Employee**, and conversely **Employee** is a base class for **Manager**.
- You can think of the derived class inheriting properties from the base class, hence the term inheritance.

Derived Classes



- Typically a object of the derived class is implemented as an object of the base class, with the information belonging to the derived class added at the end.

Derived Classes

```
void f(Manager m1, Employee e1)
{
    list<Employee*> elist {&m1,&e1};
     //...
}
```

- This way of derivation allows creating a list of employees where some employees are managers.
- In general if a **Derived** class has a public base class **Base**, then a **Derived*** can be assigned to a variable of type **Base*** without the use of explicit type conversion.

Derived Classes

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;                 // OK: every Manager is an Employee
    Manager* pm = &ee;                  // error: not every Employee is a Manager
    pm->level = 2;                      // disaster: ee doesn't have a level
    pm = static_cast<Manager*>(pe);     // brute force: works because pe points
                                        // to the Manager mm
    pm->level = 2;                      // fine: pm points to the Manager mm that
                                        // has a level
}
```

In other words, an object of a derived class can be treated as an object of its base class when manipulated through pointers and references. The opposite is not true. The use of **static_cast** and **dynamic_cast** is discussed in subsequent slides.

Derived Classes

- A class must be defined before it is used as a base class

```
class Employee;       // declaration only, no definition
class Manager : public Employee {  // error: Employee not defined
    //...
};
```

- To make classes and inheritance more interesting we need member functions
- A member of a derived class can use the public members of a base class but not the private members.

Constructors and Destructors

- **Constructors:** Objects are constructed from the bottom up (base before member and member before derived)
- **Destructors:** Objects are destroyed top-down (derived before member and member before base)
- Each class can initialize its members and bases.
- Typically destructors in a hierarchy need to be **virtual**.
- The hierarchy of constructor/destructor calls brings up an interesting dilemma. What happens if you're inside a constructor/detructor and you call a virtual function? Inside an ordinary member function you can imagine what will happen – the virtual call is resolved at runtime because the object cannot know whether it belongs to the class the member function is in or some class derived from it. For consistency you might think this is what should happen inside constructors/destructors.
- 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/destructor.

Class Hierarchies

- A derived class can be a base class

```
class Employee { /* ... */ };

class Manager : public Employee { /* ... */ };

class Director : public Manager { /* ... */ };
```

- Or more generally

```
class Temporary { /* ... */ };

class Assistant : public Employee { /* ... */ };

class Temp : public Temporary, public Assistant { /* ... */ };

class Consultant : public Temporary, public Manager { /* ... */ };
```

Problem of choosing the correct member function

- 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***.
- But with polymorphism a base class pointer can point to an object of the derived class, but what if you want the correct member function to be called .. that is the point of polymorphism right?
- 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

Solution: Virtual Functions

- **virtual** functions overcome the problems with the type-field solution by allowing the programmer to declare functions in a base class that can be redefined in each derived class. The compiler and linker will guarantee the correct correspondence between objects and the functions applied to them.
- See example 2_virtualfunction.cpp
- Note: A **virtual** function *must* be defined for the **class** in which it is first declared (unless it is declared to be a pure virtual function)
- A **virtual** function can be used even if no class is derived from its class, and a derived class that does not need its own version of a virtual function need not provide one. When deriving a class, simply provide an appropriate function if it is needed.

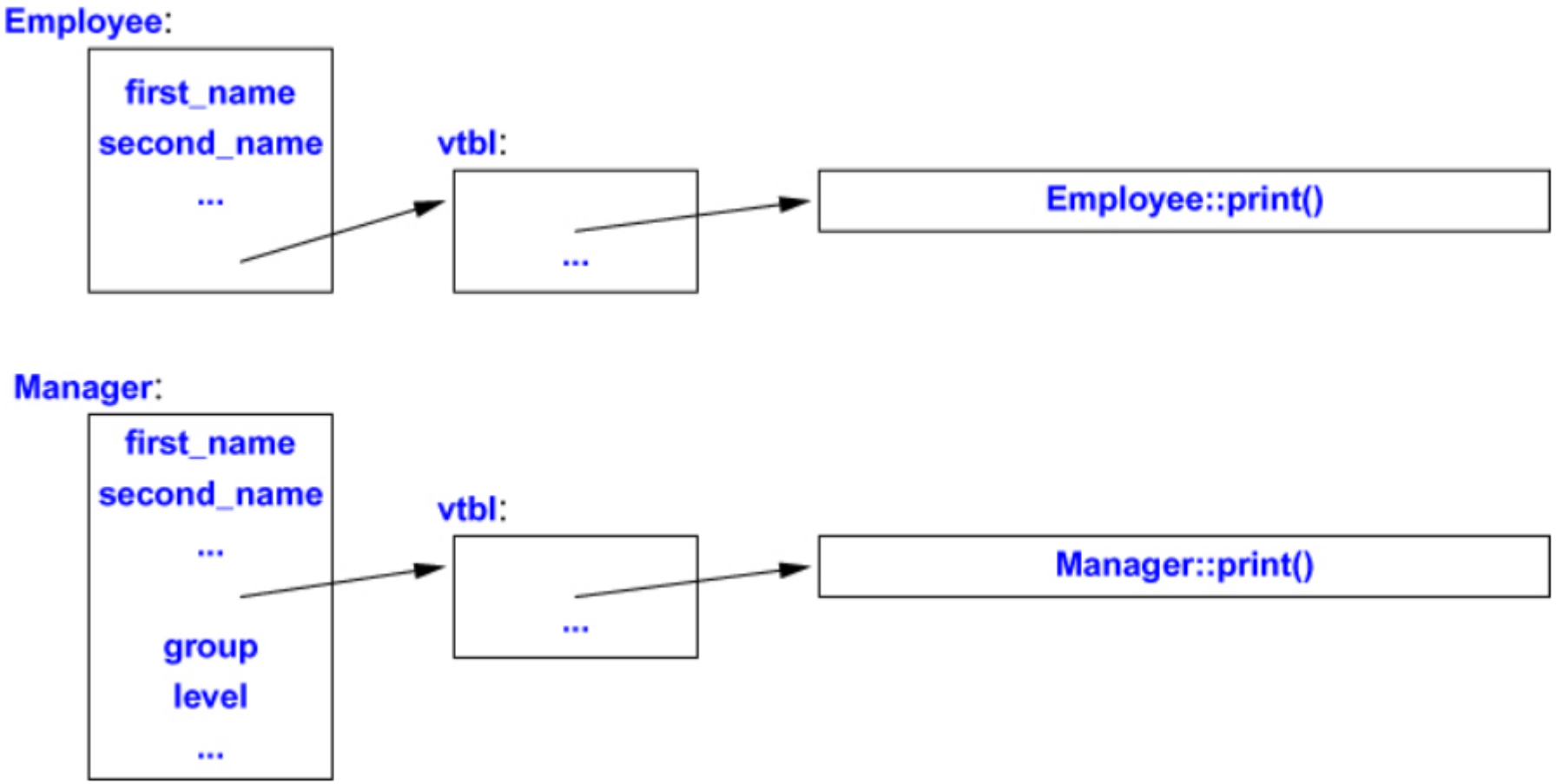
Virtual Functions

- A function from a derived class with the same name and the same set of argument types as a virtual function in a base is said to *override* the base class version of the virtual function.
- Except where we explicitly say which version of a virtual function is called (as in the call **Employee::print()**), the overriding function is chosen as the most appropriate for the object for which it is called. Independently of which base class (interface) is used to access an object, we always get the same function when we use the virtual function call mechanism.
- Note that this will work even if **print_list()** was written and compiled before the specific derived class **Manager** was even conceived of! This is a key aspect of classes. When used properly, it becomes the cornerstone of object-oriented designs and provides a degree of stability to an evolving program.

Run Time Polymorphism

- Getting “the right” behavior from **Employee**’s functions independently of exactly what kind of **Employee** is actually used is called *polymorphism*. A type with virtual functions is called a *polymorphic type* or (more precisely) a *run-time polymorphic type*. To get runtime polymorphic behavior in C++, the member functions called must be **virtual** and objects must be manipulated through pointers or references. When manipulating an object directly (rather than through a pointer or reference), its exact type is known by the compiler so that run-time polymorphism is not needed.
- By default, a function that overrides a virtual function itself becomes **virtual**. We can, but do not have to, repeat **virtual** in a derived class. If you want to be explicit, use **override.**

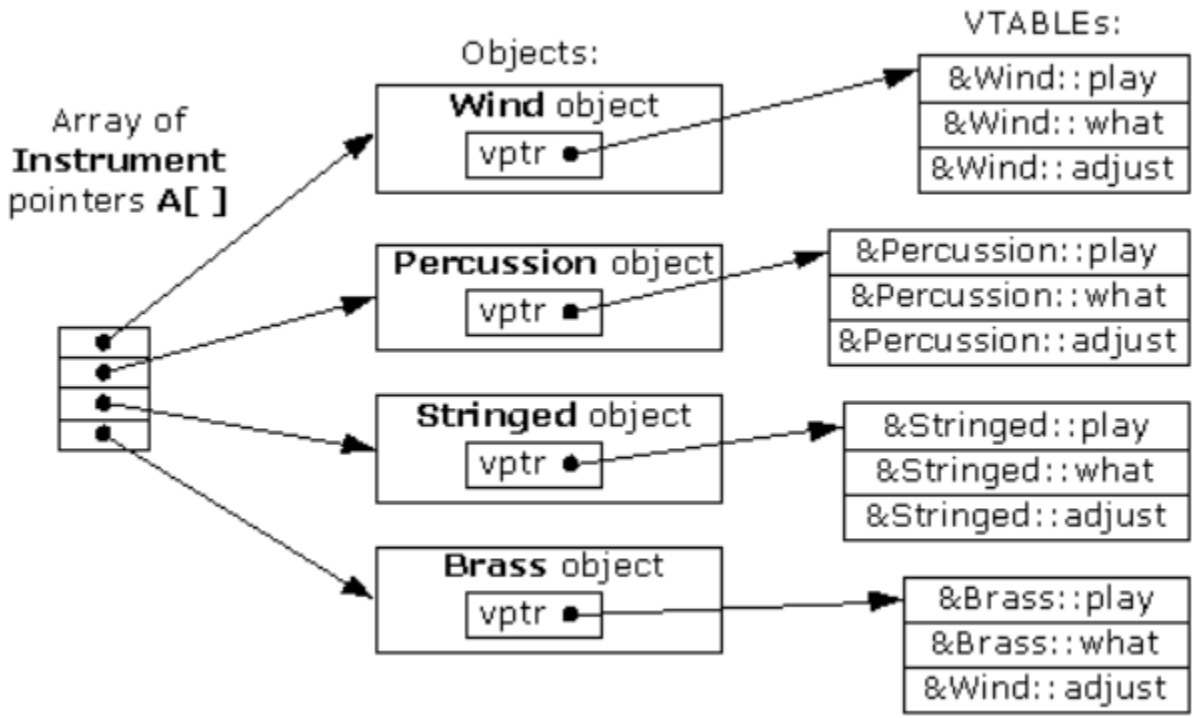
Implementation Detail: vTable



Implementation Detail : vTable

- To implement polymorphism, the compiler must store some kind of type information in each object of class **Employee** and use it to call the right version of the virtual function **print()**. In a typical implementation, the space taken is just enough to hold a pointer, the usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That table is usually called *the virtual function table* or simply the **vtbl**. Each class with virtual functions has its own **vtbl** identifying its virtual functions.
- A virtual function invoked from a constructor or a destructor reflects that the object is partially constructed or partially destroyed. It is therefore typically a bad idea to call a virtual function from a constructor or a destructor.
- **Explicit Qualification**: Calling a function using the scope resolution operator, **::**, as is done in **Manager::print()** ensures that the **virtual** mechanism is not used

Implementation Detail: Vtable



Override Controls

- If you declare a function in a derived class that has exactly the same name and type as a virtual function in a base class, then the function in the derived class overrides the one in the base class. That's a simple and effective rule. However, for larger class hierarchies it can be difficult to be sure that you actually override the function you meant to override.
- See example 4_Overridecontrol.cpp
  - **B0::f()** is not **virtual**, so you can't override it, only hide it.
  - **D::g()** doesn't have the same argument type as **B0::g()**, so if it overrides anything it's not the virtual function **B0::g()**. Most likely, **D::g()** just hides **B0::g()**.
  - There is no function called **h()** in **B0**, if **D::h()** overrides anything, it is not a function from **B0**. Most likely, it is introducing a brand-new virtual function.

Override Controls : Best Practice

- Use more specific controls while overriding:
  - • **virtual**: The function may be overridden
  - **=0**: The function must be **virtual** and must be overridden
  - The function is meant to override a virtual function in a base class
  - **final**: The function is not meant to be overridden
- In a large or complicated class hierarchy with many virtual functions, it is best to use **virtual** only to introduce a new virtual function and to use **override** on all functions intended as overriders. Using **override** is a bit verbose but clarifies the programmer's intent.
- The **override** specifier comes last in a declaration, after all other parts.
- **Esoterica: override** is not a keyword; it is what is called a *contextual keyword*. That is, **override**has a special meaning in a few contexts but can be used as an identifier elsewhere. So is **final.**

Override Controls: final

- If we do not want a member function to be overridden further by a derived class or if we do not want a class to be derived further, use the keyword **final**.

- See example **5_finalControl.cpp**

Abstract Classes

- Some classes like a **class Shape** might represent an abstract concept and make sense only if a more concrete class derives from it.
- Instead of using exceptions, make the functions in Shape pure virtual functions using “ **= 0** ”
- A class with one or more pure virtual functions is an *abstract class*, and no objects of that abstract class can be created.
- An abstract class is intended as an interface to objects accessed through pointers and references. An abstract class is intended as an interface to objects accessed through pointers and references. The design style supported by abstract classes is called *interface inheritance* in contrast to the *implementation inheritance* supported by base classes with state and/or defined member functions.

Access Control

- A member of a class can be **private**, **protected**, or **public**:
  - If it is **private**, its name can be used only by member functions and friends of the class in which it is declared.
  - If it is **protected**, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class.
  - If it is **public**, its name can be used by any function.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Multiple Inheritance

- As we saw earlier, multiple inheritance from different base classes can be achieved by separating the base classes by commas.

```
class Employee { /* ... */ };

class Manager : public Employee { /* ... */ };

class Director : public Manager { /* ... */ };

class Temporary { /* ... */ };

class Assistant : public Employee { /* ... */ };

class Temp : public Temporary, public Assistant { /* ... */ };

class Consultant : public Temporary, public Manager { /* ... */ };
```

Types of Inheritance

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private**members of the derived class.

Diamond problem and using virtual inheritance

- What if 2 base classes derive from the same base class?
- See DiamondProblem.cpp

- **Solution:** Virtual base

Runtime Type Identification

- 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.
- Say “Hello!” to **dynamic_cast**<>

BackUp: Details about the diamond Problem

- https://en.wikipedia.org/wiki/Virtual_inheritance
- https://www.cprogramming.com/tutorial/virtual_inheritance.html
- https://hownot2code.com/2016/08/12/good-and-bad-sides-of-virtual-inheritance-in-c/