Polymorphism and Virtual Functions

Chapter 10

Objectives

- **Explain the features of virtual functions and dynamic binding.**
- **Describe pointer conversion in C++ inheritance and use pointers in connection with virtual functions.**
- **Use polymorphism in C++ to write better structured, more maintainable code.**
- **Provide virtual destructors for classes using virtual functions.**
- **Specify abstract classes using pure virtual functions.**
- **Gain experience through code walk-throughs and lab exercises.**

  - The example programs are in the **chapter directory.**
  - Labs located in Labs/Lab10

A Case for Polymorphism

- **Consider the problem of generating a payroll for different types of employees.**
  - Wage and salary employees have pay calculated by different algorithms.
  - A traditional approach is to maintain a type field in an employee structure and to calculate pay in a switch statement, with cases for each type.
  - Such switch statement type code is error prone, and requires much maintenance when adding a new type (e.g., sales employee, where pay is based on commission).
- **A complete example of this case is located in the Employee folder.**

A Case for Polymorphism (continued)

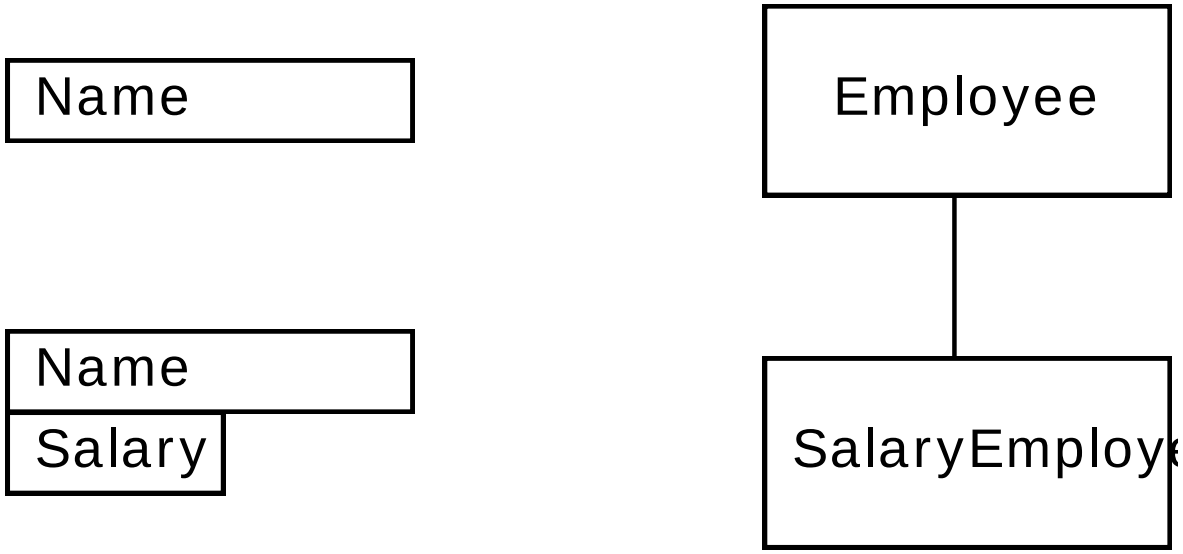
- **An alternative is to localize the intelligence to calculate pay in each employee class, which will support its own *GetPay* function.**
  - Generic payroll code can then be written that will handle different types of employees and will not have to be modified to support an additional employee type.
  - Provide a **GetPay** function in the base class, and an override of this function in each derived class.
  - Call **GetPay** through a pointer to a general **Employee** object. Depending on the actual **Employee** class pointed to, the appropriate **GetPay** function will be called.

Dynamic Binding

- **This use of "switch" statement must be replicated wherever Employee objects are manipulated.**
  - It is error prone.
  - Much duplicate code must be maintained.
- **Virtual functions and dynamic binding remove this problem.**
- **The member function *GetPay* in each class uses the appropriate algorithm for calculating pay.**
- **Declare *GetPay* as a virtual function.**
- **Then the compiler resolves the class type at runtime using the internal mechanism of dynamic binding.**

Pointer Conversion in Inheritance

- **Pointers can be converted up in an inheritance hierarchy but not down.**



Pointer Convergence in Inheritance

- **Consider pointers to *Employee* and *SalaryEmployee*:**

```
Employee* pEmp = new Employee("John");

SalaryEmployee* pSalEmp = new SalaryEmployee("Bill", 1500);

pEmp = pSalEmp;   // legal

pSalEmp = pEmp;   // illegal
```

  - A salary employee “is a” employee. It is safe to access the fields of *Employee* through **pSalEmp**, because the object pointed to by **pSalEmp** contains all the fields of *Employee* and possibly some additional fields.
  - An employee is not necessarily a salary employee. If not, there will be an error in accessing the additional field salary through **pEmp**.

Polymorphism Using Dynamic Binding

- **A generic pointer to a base class can be changed at run time to point to an object belonging to a derived class.**
- **A virtual function in the base class is overridden in the derived class.**
- **The virtual function is called through a generic pointer to the base class. Which override of the function that gets invoked is determined at runtime by the class of object referenced by the pointer.**
  - This runtime determination of which function is called is referred to as *dynamic binding*.
  - The ability for the same function call to result in different behavior depending on the object through which the function is invoked is referred to as *polymorphism*.
- **Polymorphic functions are declared as virtual by the programmer.**
- **Dynamic binding mechanism is carried out by the compiler. User need not know any details.**

Virtual Function Specification

- **A member function is declared as virtual within the class declaration as:**

```
class Employee {
public:
    virtual int GetPay() const;
    ...
};
```

Virtual Function Specification (continued)

- **The function is overridden in derived classes, keyword virtual not required :**

```
class SalaryEmployee : public Employee
{
public:
    int GetPay() const
    { return m_salary; }
        ...
};
```

```
class WageEmployee : public Employee
{
public:
    int GetPay() const
    { return m_hours * m_wage; }
        ...
};
```

Invoking Virtual Functions

- **To use dynamic binding, a virtual function must be invoked through a pointer (or a reference).**

```
int pay;
Employee* pEmp;
WageEmployee Joe;
SalaryEmployee Mary;

pEmp = &Joe;
pay = pEmp->GetPay();      // wage version

pEmp = &Mary;
pay = pEmp->GetPay();      // salary version
```

Disabling Virtual Binding

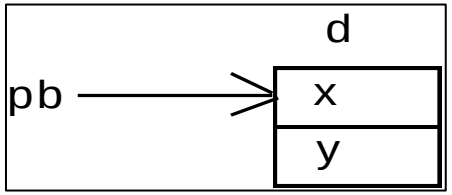
- **The use of class scope operator disables dynamic binding:**
  - pWageEmp->Employee::GetPay(); *// always resolves to Employee::GetPay()*

Virtual Functions Demo

- **The folder VirtDemo/Step1 contains a program that you can build and run and then modify.**
- **Before running it, predict the output.**
- **How can you change the definition of the base class to get the "expected" output from the following after the pointer has been reassigned to point to a D instance?**
  - pb->f()
- **The folder VirtDemo/Step2 contains the changes to get the "expected" output .**

Virtual Functions Demo (Cont'd)

- **There is static binding, and *pb->f()* will always call the "B" (base) version of the function.**

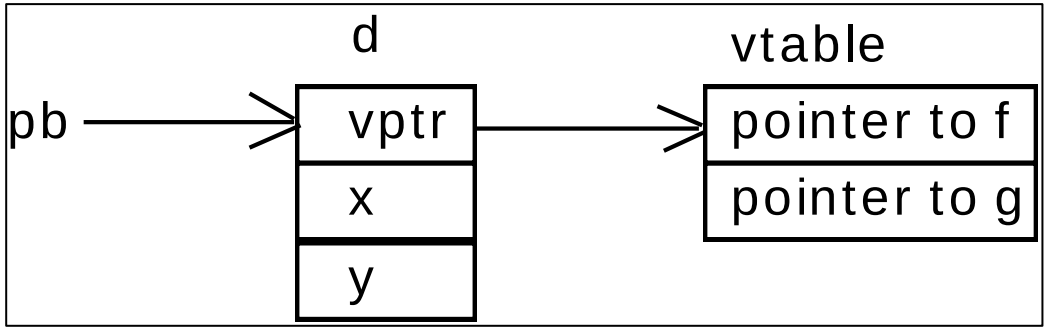


- **Now declare the functions virtual in the base class and run again.**

```
class B {
long x;
public:
  virtual void f();
  virtual void g();
};
```

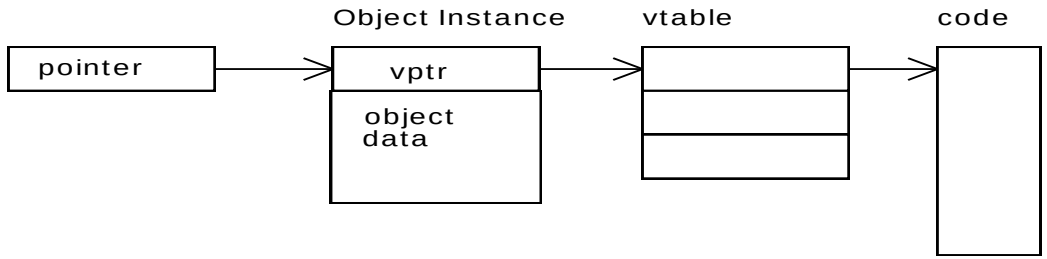
Virtual Functions Demo (Cont'd)

- **There is dynamic binding, and *pb->f()* will call the "B" version of the function if the pointer has been assigned to point to a D object**
- **The size of the objects is increased by 4 or 8 bytes, because each object instance now holds a "vptr" (pointer to a vtable).**



vtable

- Virtual functions are accessed through a pointer (or reference, which is implemented by a pointer).
- The vtable contains an array of function pointers, which point to code implementing the member functions of the interface.
- The pointer points to an area of memory of the object instance which contains a pointer to the object's vtable.
- The vtable is associated with the "class" corresponding to the object -- there is a single vtable for all object instances.



Virtual Destructors

- **Suppose *Employee* classes have a destructor (e.g., private storage of *name* is changed to use heap):**

```
Employee* pEmp;
SalaryEmployee* pSalEmp = new SalaryEmployee("John", 1500);
pEmp = pSalEmp;
delete pEmp;
```

- **Which destructors are involved? Destructor for *Employee*, *SalaryEmployee*, or both?**
  - Answer is only *Employee*, even though it was intended to destroy *pSalEmp*.

Virtual Destructors (continued)

- **To destroy *pSalEmp*, sequence should be destructor for *SalaryEmployee* and then for *Employee*.**
- **In general, it is a good idea to declare destructors of base classes as virtual:**

```
class Employee {
public:
    virtual ~Employee();
    ...
};
```

Pure Virtual Function

- **Often it is desirable to have a base class as a protocol for deriving implementations in the derived classes, without the base class having to implement all the specified functions itself.**
  - A virtual function specified but not implemented in the base class is referred to as a **pure virtual function.**
  - Notation for a pure virtual function is **= 0** after its prototype:
  - virtual int GetPay() = 0;

Pure Virtual Function (continued)

- **For a pure virtual function, only its signature is specified while its definition is deferred to derived classes**
- **A class that contains at least one pure virtual function is referred to as an *abstract class*.**
  - An abstract class cannot be instantiated.
  - For any derived class to be non-abstract, it must define all inherited pure virtual functions.

Employee as an Abstract Class

- **The *GetPay* function is not meaningful in the Employee class -- more information about an employee is need to calculate pay:**
  - Declare **GetPay** as a pure virtual function in **Employee**
  - **Employee** becomes an abstract class

```
class Employee {
public:
    virtual int GetPay() const = 0;
    ...
};
```

Heterogeneous Collections

- **A heterogeneous collection can be constructed using pointers to a base class.**
  - A pointer to a base class is generic, and at run time can be assigned to point to different derived classes.

```
Employee* pEmp[10];


WageEmployee Joe("Joe",40,15);
SalaryEmployee Mary("Mary",1500);


pEmp[0] = &Joe;
pEmp[1] = &Mary;
int nNumEmp = 2;
```

Polymorphic Code

- **We can now write generic, polymorphic code to calculate pay for a group of Employees.**
  - This code is general and won't change, even if new classes of employees are defined, provided each derived employee class implements ***GetPay*** function.

```
for (int i = 0; i < nNumEmp; ++i)
        payroll[i] = pEmp[i]->GetPay();
```

Final Consideration: Fragile Base Class Problem

- The Fragile Base Class problem is when base class inheritance is destabilized by the parent / child class hierarchy.
- The *override* and *final* keywords help resolve the fragile base class problem. Allows developers to state their intention.
  - Either keyword must be applied only to virtual methods
- The *final* specifier prevents a method from being overridden in a derived class.
- The *override* specifier stipulates that a method is intentionally overriding a method in the base class.

Demo

- The program in folder Employee demonstrates polymorphism.

Summary

- **Virtual functions and dynamic binding support the concept of polymorphism.**
- **Polymorphic code is cleaner and easier to maintain, eliminating forests of switch statements.**
- **Dynamic binding rests on accessing functions through pointers, determining the function invoked at runtime by the class of the object pointed to.**
- **In C++ inheritance hierarchies it is safe to cast a pointer to a class higher in the hierarchy but not to a class that is lower.**
- **Virtual destructors are essential for using delete on a class with virtual functions.**
- **Abstract classes are defined using pure virtual functions and cannot be instantiated.**
- **Heterogeneous collections can be constructed in C++ by using pointers to a base class.**