Object-Oriented Programming: Polymorphism

Virtual Functions

Suppose that shape classes such as **Circle**, **Triangle**, **Rectangle** and **Square** are all derived from base class **Shape**.

Each of these classes has a member function **draw()** to *draw itself.*

In a program that draws a set of shapes, it would be useful to be able to **treat all the shapes *generally as objects of the base class Shape***.

Then, to draw any shape, we could simply use a base-class **Shape pointer** to invoke function draw and let the program determine ***dynamically*** (i.e., at runtime) which derived-class draw function to use, **based on the type of the object to which the base-class Shape pointer *points at any given time***. This is ***polymorphic behavior***.

- With **virtual functions**, the **type** of the **object**—**NOT** the **type of the handle** used to invoke the object's member function—determines which version of a virtual function to invoke.

Virtual Functions

To enable this behavior, we declare draw in the base class as a **virtual function**, and we **override** draw in *each* of the derived classes to draw the appropriate shape.

From an implementation perspective, *overriding* a function is no different than *redefining* one (which is the approach we've been using until now). An overridden function in a derived class has the ***same signature*** ***and*** ***return type*** (i.e., *prototype*) as the function it overrides in its base class.

If we *do not* declare the base-class function as virtual, we can ***redefine*** that function.

By contrast, if we *do* declare the base-class function as virtual, we can *override* that function to enable *polymorphic behavior*.

virtual void **draw**() const;

Virtual Functions

- Once a function is declared virtual, **it remains virtual all the way down the inheritance hierarchy from that point**, *even if that function is not explicitly declared virtual when a derived class overrides it*.

- Even though certain functions are implicitly virtual because of a declaration made higher in the class hierarchy, **for clarity explicitly declare these functions virtual** at every level of the class hierarchy.

- When a derived class chooses not to override a virtual function from its base class, *the derived class simply inherits its base class's virtual function implementation.*

Virtual Functions

If a program invokes a **virtual** function through
- a base-class pointer to a derived-class object (e.g., **shapePtr->draw()**)
- or a base-class reference to a derived-class object (e.g., **shapeRef.draw()**),

the program will choose the correct derived-class function **dynamically** (i.e., at execution time) **based on the object type**—**not** the pointer or reference type.

*Choosing the appropriate function to call at **execution** time (rather than at **compile** time) is known as **dynamic binding**.*

When a virtual function is called by referencing a specific object by name and using the dot member-selection operator (e.g., **squareObject.draw()**), the function invocation is resolved at **compile time** (this is called **static binding**) and the **virtual function** that's called is the one defined for (or inherited by) **the class of that particular object**—this is NOT polymorphic behavior.

Dynamic binding with virtual functions occurs only off *pointers* and *references.*

Virtual Functions

To help prevent errors, apply **[C++11]'s <u>override</u> keyword** to the <u>*prototype* of every derived-class function</u> that overrides a base-class virtual function.

This enables the compiler to **check** whether the **base class has a virtual member function with the same signature**.

If not, the compiler generates <u>an error</u>.

Not only does this ensure that you override the base-class function with the appropriate signature, <u>it also prevents you from accidentally hiding a base-class function</u> that has the same name and a different signature (*demonstrate an example*).

```
class CommissionEmployee {
public:
   CommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0);

   void setFirstName(const std::string&); // set first name
   std::string getFirstName() const; // return first name

   void setLastName(const std::string&); // set last name
   std::string getLastName() const; // return last name

   void setSocialSecurityNumber(const std::string&); // set SSN
   std::string getSocialSecurityNumber() const; // return SSN

   void setGrossSales(double); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   void setCommissionRate(double); // set commission rate (percentage)
   double getCommissionRate() const; // return commission rate

   virtual double earnings() const; // calculate earnings
   virtual std::string toString() const; // string representation
private:
   std::string firstName;
   std::string lastName;
   std::string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
};
```

```cpp
// Fig. 12.5: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration

class BasePlusCommissionEmployee : public CommissionEmployee {
public:
   BasePlusCommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0, double = 0.0);

   void setBaseSalary(double); // set base salary
   double getBaseSalary() const; // return base salary

   virtual double earnings() const override; // calculate earnings
   virtual std::string toString() const override; // string representation
private:
   double baseSalary; // base salary
};

#endif
```

```cpp
int main() {
   // create base-class object
   CommissionEmployee commissionEmployee{
      "Sue", "Jones", "222-22-2222", 10000, .06};

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee{
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300};

   cout << fixed << setprecision(2); // set floating-point formatting

   // output objects using static binding
   cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND DERIVED-CLASS "
      << "\nOBJECTS WITH STATIC BINDING\n"
      << commissionEmployee.toString() // static binding
      << "\n\n"
      << basePlusCommissionEmployee.toString(); // static binding

   // output objects using dynamic binding
   cout << "\n\nINVOKING TOSTRING FUNCTION ON BASE-CLASS AND "
      << "\nDERIVED-CLASS OBJECTS WITH DYNAMIC BINDING";

   // natural: aim base-class pointer at base-class object
   const CommissionEmployee* commissionEmployeePtr{&commissionEmployee};
   cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER"
      << "\nTO BASE-CLASS OBJECT INVOKES BASE-CLASS "
      << "TOSTRING FUNCTION:\n"
      << commissionEmployeePtr->toString(); // base version

   // natural: aim derived-class pointer at derived-class object
   const BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
      &basePlusCommissionEmployee}; // natural
   cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS "
      << "POINTER\nTO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
      << "TOSTRING FUNCTION:\n"
      << basePlusCommissionEmployeePtr->toString(); // derived version
```

```cpp
    // aim base-class pointer at derived-class object
    commissionEmployeePtr = &basePlusCommissionEmployee;
    cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER"
       << "\nTO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
       << "TOSTRING FUNCTION:\n";

    // polymorphism; invokes BasePlusCommissionEmployee's toString
    // via base-class pointer to derived-class object
    cout<< commissionEmployeePtr->toString() << endl;
}
```

```
INVOKING TOSTRING FUNCTION ON BASE-CLASS AND DERIVED-CLASS
OBJECTS WITH STATIC BINDING
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO BASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING FUNCTION:
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS TOSTRING FUNCTION:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS TOSTRING FUNCTION:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300
```

Virtual Destructors

A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy.

If a derived-class object with a **non-virtual destructor** is **destroyed by applying the delete operator to a *base-class pointer* to the object**, **the** C++ standard **specifies that the behavior is *undefined***.

The simple solution to this problem is to create a **public virtual destructor** in the **base class**.

If a base-class destructor is declared virtual, the destructors of **any derived classes are *also* virtual**.

For example, in class CommissionEmployee's definition, we can define the virtual destructor as follows

virtual ~CommissionEmployee() {};

Virtual Destructors

Now, if an object in the hierarchy is destroyed explicitly by applying the delete operator to a *base-class pointer*, the destructor for the *appropriate class* is called, **based on the object to which the base-class pointer points**.

Remember, when a derived-class object is destroyed, the base-class part of the derived-class object is also destroyed, so it's important for the destructors of *both* the derived and base classes to execute.

The base-class destructor automatically executes after the derived-class destructor.

- If a class has virtual functions, always provide a virtual destructor, even if one is not required for the class. This ensures that a custom derived-class destructor (if there is one) will be invoked when a derived-class object is deleted via a base-class pointer.

- **Constructors cannot be virtual**. Declaring a constructor virtual is a compilation error.

Virtual Destructors

The preceding destructor definition also may be written as follows:

virtual ~CommissionEmployee() = **default**;

In **C++11**, you can tell the compiler to explicitly generate the default version of a *default constructor*, *copy constructor*, *move constructor*, *copy assignment operator*, *move assignment operator* or *destructor* by following the special member function's prototype with **= default.**

This is useful, for example, when you explicitly define a constructor for a class and still want the compiler to generate a default constructor as well

ClassName() = **default**;

Virtual Destructors

Prior to C++11, a derived class could override ***any*** of its base class's virtual functions. In C++11, a base-class virtual function that's declared **final** in its prototype, as in

virtual *someFunction*(*parameters*) **final**;

cannot be overridden in any derived class.

- This guarantees that the base class's final member function definition will be used by all *base-class objects* and by all objects of the base class's *direct and indirect derived classes*.

Similarly, prior to C++11, *any* existing class could be used as a base class in a hierarchy. As of C++11, you can declare a class as **final** to *prevent it from being used as a base class*, as in

class *MyClass* **final** { // this class cannot be a base class
// class body
};

- Attempting to override a final member function or inherit from a final base class results in a *compilation error.*

Type fields and Switch statement

```
class Base {
public:
   virtual void draw() const {
      cout << "Base::draw function called" << endl;
   }
};
class Derived : public Base {
public:
   virtual void draw() const override {
      cout << "Derived::draw function called" << endl;
   }
};
int main()
{
   Derived d1, d2;
   Base b;
   Base* arr[3] = {&d1, &d2, &b};
   for (int i = 0; i < 3; ++i) {
      arr[i]->draw();
   }
   return 0;
}
```

Result:
Derived::draw function called
Derived::draw function called
Base::draw function called

Type fields and Switch statement

One way to determine an object's type is to use a **switch** statement to check the value of a field in the object.

```
class Base {
public:
…
int _type = 0;
};

class Derived : public Base {
public:
…
   int _type = 1;
};
```

```
…
for (int i = 0; i < 3; ++i) {
        switch (arr[i]->_type) {
        case 0:
            arr[i]->draw();
            break;
        case 1:
            ((Derived*)arr[i])->draw();
            break;
        }
    }
…
```

- Polymorphic programming can eliminate the need for switch logic. By using the polymorphism mechanism to perform the equivalent logic, you can avoid the kinds of errors typically associated with switch logic.
- An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and simpler sequential code.

Virtual Functions with default arguments

```
class Base {
public:
   virtual void draw(int i = 0) const {
      cout << "Base::draw function called: " << i << endl;
   }
};
class Derived : public Base {
public:
   virtual void draw(int i = 45) const override {
      cout << "Derived::draw function called: " << i << endl;
   }
};
int main()
{
   Derived d1, d2;
   Base b;
   Base* arr[3] = {&d1, &d2, &b};
   for (int i = 0; i < 3; ++i) {
      arr[i]->draw();
   }
   return 0;
}
```

Result:
Derived::draw function called: **0**
Derived::draw function called: **0**
Base::draw function called: 0

Note!
The Derived::draw body was executed with Base::draw signature.

Abstract Classes

When we think of a class as a type, we assume that programs will create objects of that type.

However, there are cases in which it's useful to define *classes from which you never intend to instantiate any objects*. Such classes are called **abstract classes.**

Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as **abstract base classes**.

These classes cannot be used to instantiate object since there are ***incomplete***—derived classes must define the "missing pieces" before objects of these classes can be instantiated.

Classes that can be used to instantiate objects are called **concrete classes**.

Abstract Classes

A class is made ***abstract*** by declaring one or more of its virtual functions to be "pure."

A **pure virtual function** is specified by placing "**= 0**" in its declaration:

virtual void draw() const **= 0**; // pure virtual function

The "= 0" is a **pure specifier.**

- Pure virtual functions do *not* provide implementations.
- Each *concrete* derived class *must override all* base-class pure virtual functions with concrete implementations of those functions; otherwise, the derived class is also abstract.
- Pure virtual functions are used when it does *not* make sense for the base class to have an implementation of a function, but you want to force all concrete derived classes to implement the function.
- Although we *cannot* instantiate objects of an abstract base class, we *can* use the abstract base class to declare *pointers* and *references* that can refer to objects of any *concrete* classes derived from the abstract class

Abstract Classes

- An abstract class defines a common public interface for the various classes that derive from it in a class hierarchy. An abstract class contains one or more pure virtual functions that *concrete* derived classes must override.

- Failure to override a pure virtual function in a derived class makes that class **abstract**. Attempting to instantiate an object of an abstract class causes a compilation error.

- An abstract class has at least one pure virtual function. An abstract class also can have data members and concrete functions (including constructors and destructors), which are subject to the normal rules of inheritance by derived classes.

Payroll System using Polymorphism

A company pays its employees weekly.

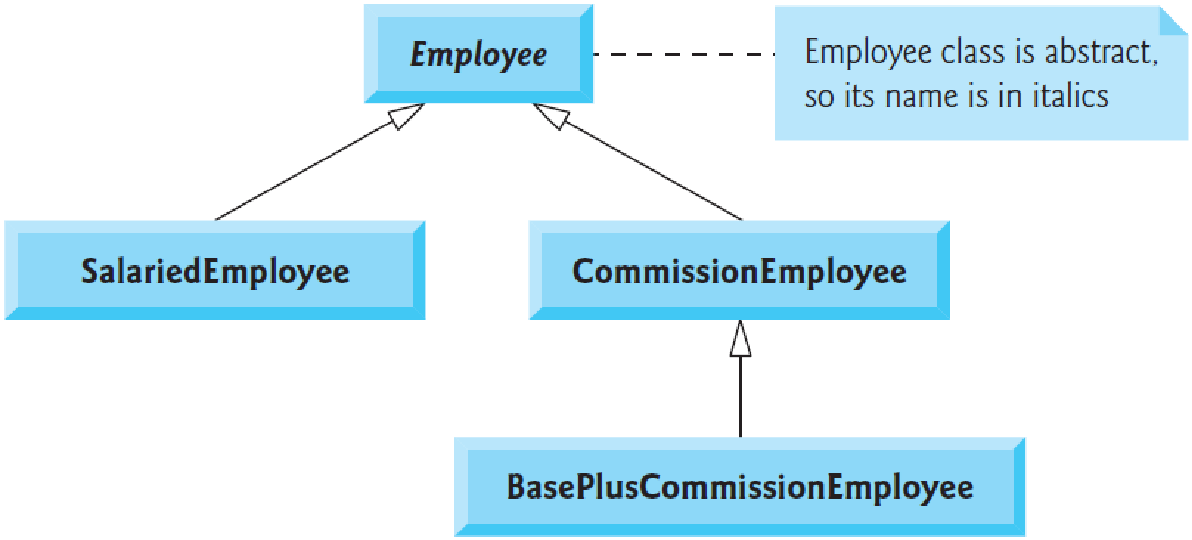
The employees are of three types:

1. <u>Salaried employees</u> are paid a fixed weekly salary regardless of the number of hours worked,
2. <u>Commission employees</u> are paid a percentage of their sales
3. <u>Base-salary-plus-commission employees</u> receive a base salary plus a percentage of their sales.

For the current pay period, the company has decided to <u>reward base-salary-plus-commission</u> employees by <u>adding 10 percent</u> to their base salaries.

The company wants to implement a C++ program that performs its <u>payroll calculations polymorphically</u>.

Payroll System using Polymorphism



A derived class can inherit **interface** and/or **implementation** from a base class.

- Hierarchies designed for **implementation inheritance** tend to have their functionality high in the hierarchy—each derived class inherits one or more member functions from a base class, and the derived class uses the base-class definitions.
- Hierarchies designed for **interface inheritance** tend to have their functionality lower in the hierarchy—a base class specifies one or more functions that should be defined by every derived class, but the individual derived classes provide their own implementations of the function(s).

```cpp
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string> // C++ standard string class

class Employee {
public:
   Employee(const std::string&, const std::string&, const std::string &);
   virtual ~Employee() = default; // compiler generates virtual destructor

   void setFirstName(const std::string&); // set first name
   std::string getFirstName() const; // return first name

   void setLastName(const std::string&); // set last name
   std::string getLastName() const; // return last name

   void setSocialSecurityNumber(const std::string&); // set SSN
   std::string getSocialSecurityNumber() const; // return SSN

   // pure virtual function makes Employee an abstract base class
   virtual double earnings() const = 0; // pure virtual
   virtual std::string toString() const; // virtual
private:
   std::string firstName;
   std::string lastName;
   std::string socialSecurityNumber;
};

#endif // EMPLOYEE_H
```

```cpp
#include "Employee.h" // Employee class definition
using namespace std;

// constructor
Employee::Employee(const string& first, const string& last,
   const string& ssn)
   : firstName(first), lastName(last), socialSecurityNumber(ssn) {}

// set first name
void Employee::setFirstName(const string& first) {firstName = first;}

// return first name
string Employee::getFirstName() const {return firstName;}

// set last name
void Employee::setLastName(const string& last) {lastName = last;}

// return last name
string Employee::getLastName() const {return lastName;}

// set social security number
void Employee::setSocialSecurityNumber(const string& ssn) {
   socialSecurityNumber = ssn; // should validate
}

// return social security number
string Employee::getSocialSecurityNumber() const {
   return socialSecurityNumber;
}

// toString Employee's information (virtual, but not pure virtual)
string Employee::toString() const {
   return getFirstName() + " "s + getLastName() +
      "\nsocial security number: "s + getSocialSecurityNumber();
}
```

```cpp
#ifndef SALARIED_H
#define SALARIED_H

#include <string> // C++ standard string class
#include "Employee.h" // Employee class definition

class SalariedEmployee : public Employee {
public:
   SalariedEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0);
   virtual ~SalariedEmployee() = default; // virtual destructor

   void setWeeklySalary(double); // set weekly salary
   double getWeeklySalary() const; // return weekly salary

   // keyword virtual signals intent to override
   virtual double earnings() const override; // calculate earnings
   virtual std::string toString() const override; // string representation
private:
   double weeklySalary; // salary per week
};

#endif // SALARIED_H
```

```cpp
#include "SalariedEmployee.h" // SalariedEmployee class definition
using namespace std;

// constructor
SalariedEmployee::SalariedEmployee(const string& first,
   const string& last, const string& ssn, double salary)
   : Employee(first, last, ssn) {
   setWeeklySalary(salary);
}

// set salary
void SalariedEmployee::setWeeklySalary(double salary) {
   if (salary < 0.0) {
      throw invalid_argument("Weekly salary must be >= 0.0");
   }

   weeklySalary = salary;
}

// return salary
double SalariedEmployee::getWeeklySalary() const {return weeklySalary;}

// calculate earnings;
// override pure virtual function earnings in Employee
double SalariedEmployee::earnings() const {return getWeeklySalary();}

// return a string representation of SalariedEmployee's information
string SalariedEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2);
   output << "salaried employee: "
      << Employee::toString() // reuse abstract base-class function
      << "\nweekly salary: " << getWeeklySalary();
   return output.str();
}
```

```cpp
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
#include "Employee.h" // Employee class definition

class CommissionEmployee : public Employee {
public:
   CommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0);
   virtual ~CommissionEmployee() = default; // virtual destructor

   void setCommissionRate(double); // set commission rate
   double getCommissionRate() const; // return commission rate

   void setGrossSales(double); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   // keyword virtual signals intent to override
   virtual double earnings() const override; // calculate earnings
   virtual std::string toString() const override; // string representation
private:
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
};

#endif // COMMISSION_H
```

```cpp
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee(const string &first,
   const string &last, const string &ssn, double sales, double rate)
   : Employee(first, last, ssn) {
   setGrossSales(sales);
   setCommissionRate(rate);
}

// set gross sales amount
void CommissionEmployee::setGrossSales(double sales) {
   if (sales < 0.0) {
      throw invalid_argument("Gross sales must be >= 0.0");
   }

   grossSales = sales;
}

// return gross sales amount
double CommissionEmployee::getGrossSales() const {return grossSales;}

// set commission rate
void CommissionEmployee::setCommissionRate(double rate) {
   if (rate <= 0.0 || rate > 1.0) {
      throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
   }

   commissionRate = rate;
}

```

```
// return commission rate
double CommissionEmployee::getCommissionRate() const {
   return commissionRate;
}

// calculate earnings; override pure virtual function earnings in Employee
double CommissionEmployee::earnings() const {
   return getCommissionRate() * getGrossSales();
}

// return a string representation of CommissionEmployee's information
string CommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2);
   output << "commission employee: " << Employee::toString()
      << "\ngross sales: " << getGrossSales()
      << "; commission rate: " << getCommissionRate();
   return output.str();
}
```

```cpp
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class definition

class BasePlusCommissionEmployee : public CommissionEmployee {
public:
   BasePlusCommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0, double = 0.0);
   virtual ~BasePlusCommissionEmployee() = default; // virtual destructor

   void setBaseSalary(double); // set base salary
   double getBaseSalary() const; // return base salary

   // keyword virtual signals intent to override
   virtual double earnings() const override; // calculate earnings
   virtual std::string toString() const override; // string representation
private:
   double baseSalary; // base salary per week
};

#endif // BASEPLUS_H
```

```cpp
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string& first, const string& last, const string& ssn,
   double sales, double rate, double salary)
   : CommissionEmployee(first, last, ssn, sales, rate) {
   setBaseSalary(salary); // validate and store base salary
}

// set base salary
void BasePlusCommissionEmployee::setBaseSalary(double salary) {
   if (salary < 0.0) {
      throw invalid_argument("Salary must be >= 0.0");
   }

   baseSalary = salary;
}

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const {
    return baseSalary;
}

```

```cpp
// calculate earnings;
// override virtual function earnings in CommissionEmployee
double BasePlusCommissionEmployee::earnings() const {
    return getBaseSalary() + CommissionEmployee::earnings();
}

// return a string representation of a BasePlusCommissionEmployee
string BasePlusCommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2);
   output << "base-salaried " << CommissionEmployee::toString()
      << "; base salary: " << getBaseSalary();
   return output.str();
}
```

```cpp
#include "Employee.h"
#include "SalariedEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

void virtualViaPointer(const Employee* const); // prototype
void virtualViaReference(const Employee&); // prototype

int main() {
   cout << fixed << setprecision(2); // set floating-point formatting

   // create derived-class objects
   SalariedEmployee salariedEmployee{
      "John", "Smith", "111-11-1111", 800};
   CommissionEmployee commissionEmployee{
      "Sue", "Jones", "333-33-3333", 10000, .06};
   BasePlusCommissionEmployee basePlusCommissionEmployee{
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300};

   // output each Employee's information and earnings using static binding
   cout << "EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING\n"
      << salariedEmployee.toString()
      << "\nearned $" << salariedEmployee.earnings() << "\n\n"
      << commissionEmployee.toString()
      << "\nearned $" << commissionEmployee.earnings() << "\n\n"
      << basePlusCommissionEmployee.toString()
      << "\nearned $" << basePlusCommissionEmployee.earnings() << "\n\n";
```

```cpp
   // create and initialize vector of three base-class pointers
   vector<Employee *> employees{&salariedEmployee, &commissionEmployee,
      &basePlusCommissionEmployee};

   cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING\n\n";

   // call virtualViaPointer to print each Employee's information
   // and earnings using dynamic binding
   cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS POINTERS\n";

   for (const Employee* employeePtr : employees) {
      virtualViaPointer(employeePtr);
   }

   // call virtualViaReference to print each Employee's information
   // and earnings using dynamic binding
   cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS REFERENCES\n";

   for (const Employee* employeePtr : employees) {
      virtualViaReference(*employeePtr); // note dereferencing
   }
}

// call Employee virtual functions toString and earnings off a
// base-class pointer using dynamic binding
void virtualViaPointer(const Employee* const baseClassPtr) {
   cout << baseClassPtr->toString()
      << "\nearned $" << baseClassPtr->earnings() << "\n\n";
}

// call Employee virtual functions toString and earnings off a
// base-class reference using dynamic binding
void virtualViaReference(const Employee& baseClassRef) {
   cout << baseClassRef.toString()
      << "\nearned $" << baseClassRef.earnings() << "\n\n";
}
```

```
EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00
base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

```
EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING

VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS POINTERS
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

```
VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS REFERENCES
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Dynamic Binding "Under the Hood"

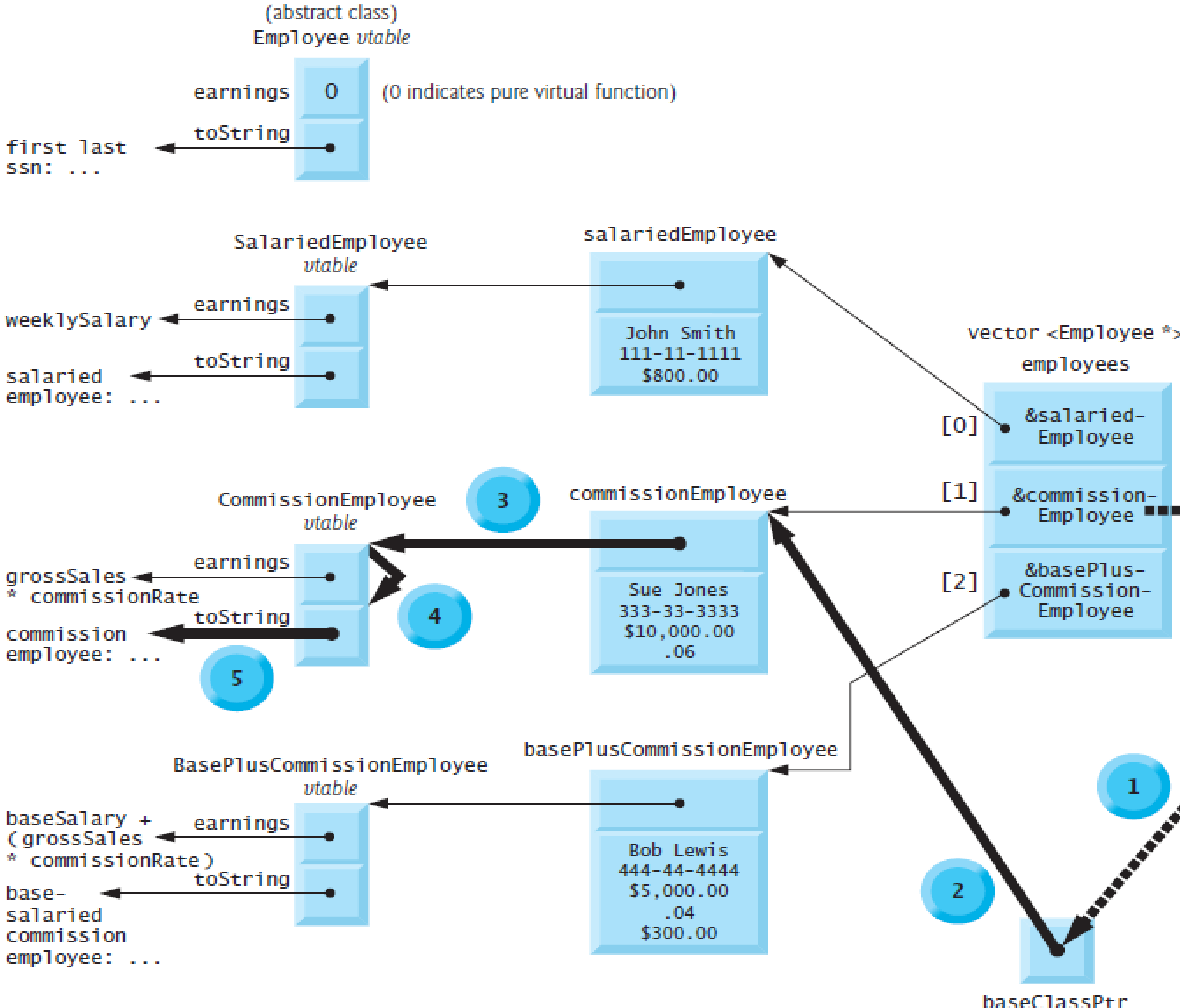
When C++ compiles a class that has one or more virtual functions, it builds a **virtual function table (*vtable*)** for that class.

The ***vtable*** *contains* pointers to the class's virtual functions—a **pointer to a function** contains the starting address in memory of the code that performs the function's task.

An executing program uses the ***vtable*** to select the proper function implementation each time a virtual function of that class is called on any object of that class.

Whenever an object of a class with one or more virtual functions is instantiated, the compiler attaches to the object a pointer to the vtable for that class.

Polymorphism is accomplished through three levels of pointers, i.e., *triple indirection*:
1. Function pointers in vtable
2. vtable pointer
3. pointer to the objects that receives the virtual function call



Flow of Virtual Function Call baseClassPtr->toString()
When baseClassPtr Points to Object commissionEmployee

1. pass &commissionEmployee to baseClassPtr
2. get to commissionEmployee object
3. get to commissionEmployee vtable
4. get to toString pointer in vtable
5. execute toString for commissionEmployee

Dynamic Binding “Under the Hood”

Consider the **baseClassPtr->toString()** call in virtualViaPointer function where baseClassPtr=employees[1].

- Compiler determines that the call is indeed being made via a *base-class pointer* and that toString is a virtual function.
- Compiler determines that toString is the *second* entry in each of the *vtables.*
- Compiler compiles an **offset** or **displacement** into the table of machine-language object-code pointers to find the code that will execute the virtual function call.

Dynamic Binding "Under the Hood"

The compiler generates code that performs the following operations (**possible implementation**):

1. Select the i-th entry of employees (employees[1]) and pass it as an argument to function virtualViaPointer. This sets parameter baseClassPtr to point to commissionEmployee.

2. Dereference (*) that pointer to get to the commissionEmployee object—which, as you recall, begins with a pointer to the CommissionEmployee **vtable**.

3. Dereference commissionEmployee's **vtable** pointer to get to the CommissionEmployee vtable.

4. **Skip the offset** of 4 bytes (or 8 depending on machine) to select the toString function pointer.

5. Dereference the toString function pointer to form the "name" of the actual function to execute, and use the function-call operator **()** to execute the appropriate toString function.

Object size

```
class Base {
};


int main()
{
   Base b;
   cout << sizeof(b) << endl;
}
```

Result:
1

- To ensure that the addresses of two different objects will be different, the size of an empty class is **1 byte**.

Object size

```
class Base {
public:
   void f() {}
};


int main()
{
   Base b;
   cout << sizeof(b) << endl;
}
```

Result:
1

Object size

```
class Base {
public:
    virtual void f() {}
};

int main()
{
   Base b;
   cout << sizeof(b) << endl;
}
```

Result:
4

- Whenever an object of a class with one or more virtual functions is instantiated, the compiler attaches to the object a **pointer to the vtable** for that class.

Runtime Type Information (RTTI)

To increase the base salaries of BasePlusCommissionEmployees by 10%, we have to determine the *specific type* of each Employee object ***at execution time***.

This is achieved by **runtime type information (RTTI)** and **dynamic casting**, which enable a program to determine an object's type at execution time.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <typeinfo>
#include "Employee.h"
#include "SalariedEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main() {
   // set floating-point output formatting
   cout << fixed << setprecision(2);

   // create and initialize vector of three base-class pointers
   vector<Employee*> employees{
      new SalariedEmployee("John", "Smith", "111-11-1111", 800),
      new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06),
      new BasePlusCommissionEmployee(
         "Bob", "Lewis", "444-44-4444", 5000, .04, 300)};
```

```
    // polymorphically process each element in vector employees
    for (Employee* employeePtr : employees) {
       cout << employeePtr->toString() << endl; // output employee

       // attempt to downcast pointer
       BasePlusCommissionEmployee* derivedPtr =
          dynamic_cast<BasePlusCommissionEmployee*>(employeePtr);

       // determine whether element points to a BasePlusCommissionEmployee
       if (derivedPtr != nullptr) { // true for "is a" relationship
          double oldBaseSalary = derivedPtr->getBaseSalary();
          cout << "old base salary: $" << oldBaseSalary << endl;
          derivedPtr->setBaseSalary(1.10 * oldBaseSalary);
          cout << "new base salary with 10% increase is: $"
             << derivedPtr->getBaseSalary() << endl;
       }

       cout << "earned $" << employeePtr->earnings() << "\n\n";
    }

    // release objects pointed to by vector's elements
    for (const Employee* employeePtr : employees) {
       // output class name
       cout << "deleting object of "
          << typeid(*employeePtr).name() << endl;

       delete employeePtr;
    }
 }
```

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

Runtime Type Information (RTTI)

For downcast operation, **dynamic_cast** operator is used to determine whether the current Employee's type is BasePlusCommissionEmployee.

If employeePtr points to an object that ***is a*** BasePlusCommissionEmployee object,

- then that object's ***address*** is assigned to derived-class pointer derivedPtr;
- otherwise, **nullptr** is assigned to derivedPtr.

We *must* use **dynamic_cast** here, rather than **static_cast**, to perform type checking on the underlying object—a static_cast would **simply cast** the Employee* to a BasePlusCommissionEmployee* **regardless** of the underlying object's type.

With a *static_cast*, the program would attempt to increase ***every*** Employee's base salary, resulting in undefined behavior for each object that's not a BasePlusCommissionEmployee.

Operator **typeid** returns a *reference* to an object of class **type_info** that contains the information about the type of its operand, including the name of that type.

- To use **typeid**, the program must include header **<typeinfo>.**
- The string returned by **type_info** member function **name** may vary by compiler.