Object-Oriented Programming: <u>Inheritance</u>

CommissionEmployee class

```
#ifndef COMMISSION_H
#define COMMISSION_H

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

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
```

CommissionEmployee class

```
   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

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

#endif
```

```cpp
#include <iomanip>
#include <stdexcept>
#include <sstream>
#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) {
   firstName = first; // should validate
   lastName = last; // should validate
   socialSecurityNumber = ssn; // should validate
   setGrossSales(sales); // validate and store gross sales
   setCommissionRate(rate); // validate and store commission rate
}

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

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

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

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

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

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

// 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;
}
```

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

// calculate earnings
double CommissionEmployee::earnings() const {
   return commissionRate * grossSales;
}

// return string representation of CommissionEmployee object
string CommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2); // two digits of precision
   output << "commission employee: " << firstName << " " << lastName
      << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate;
   return output.str();
}
```

```cpp
#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

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

```cpp
// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string& first, const string& last, const string& ssn,
   double sales, double rate, double salary)
   // explicitly call base-class constructor
   : 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");
   }
```

```cpp
   baseSalary = salary;
}

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

// calculate earnings
double BasePlusCommissionEmployee::earnings() const {
   // derived class cannot access the base class's private data
   return baseSalary + (commissionRate * grossSales);
}

// returns string representation of BasePlusCommissionEmployee object
string BasePlusCommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2); // two digits of precision

   // derived class cannot access the base class's private data
   output << "base-salaried commission employee: " << firstName << ' '
      << lastName << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate
      << "\nbase salary: " << baseSalary;
   return output.str();
}
```

Compilation Errors from the Clang/LLVM Compiler in Xcode 7.2

```
BasePlusCommissionEmployee.cpp:34:25:
   'commissionRate' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:34:42:
   'grossSales' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:42:55:
   'firstName' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:43:10:
   'lastName' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:43:54:
   'socialSecurityNumber' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:44:31:
   'grossSales' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:45:35:
   'commissionRate' is a private member of 'CommissionEmployee'
```

The compiler generates errors because base-class CommissionEmployee's data members are **private**.
C++ rigidly enforces restrictions on accessing private data members, so that *even a derived class cannot access the base class's private data.*

The error can be prevented by the following two ways

- By using the **get member functions** inherited from class CommissionEmployee.
- By using **protected** data.

Protected access specifier

A **base class's public members** are accessible within its body and anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that class or one of its derived classes, including in derived classes.

A **base class's private members** are accessible only within its body and to the friends of that base class.

A **base class's protected members** can be accessed within the body of that base class, by members and friends of that base class, and by members and friends of any classes ***derived*** from that base class.

Objects of a derived class also can access **protected** members in *any* of that derived class's ***indirect*** base classes.

To enable class BasePlusCommissionEmployee to *directly access* CommissionEmployee data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate, we can declare those members as **protected** in the base class.

```cpp
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
   double getCommissionRate() const; // return commission rate

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

Protected access specifier

Inheriting protected data members **slightly improves performance**, because we can directly access the members without incurring the overhead of calls to *set* or *get* member functions.

Using protected data members creates two serious problems:

1. The derived-class object **does not have to use a member function** to set the value of the base class's protected data member. An invalid value can easily be assigned to the protected data member.
2. Derived-class member functions are more likely to be written so that they **depend on the base-class implementation**. Derived classes should depend **only on the base-class services** (i.e., nonprivate member functions) and not on the base-class implementation.

- ✓ You should be able to change the base-class implementation while still providing the same services to derived classes.
- ✓ It's appropriate to use the protected access specifier when a **base class should provide a service** (i.e., a non-private member function) **only to its derived classes and friends**.
- ✓ Declaring base-class data members private (as opposed to declaring them protected) enables you to change the base-class implementation without having to change derived-class implementations.

Using private data and set/get methods

We now re-examine our hierarchy once more, this time using *best software engineering practices*.
We **replace the protected data by private data** and access them **through set/get methods**.

```
// constructor
CommissionEmployee::CommissionEmployee(const string &first,
    const string &last, const string &ssn, double sales, double rate)
    : firstName(first), lastName(last), socialSecurityNumber(ssn) {
    setGrossSales(sales); // validate and store gross sales
    setCommissionRate(rate); // validate and store commission rate
}
```

```cpp
// calculate earnings
double CommissionEmployee::earnings() const {
   return getCommissionRate() * getGrossSales();
}

// return string representation of CommissionEmployee object
string CommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2); // two digits of precision
   output << "commission employee: "
      << getFirstName() << ' ' << getLastName()
      << "\nsocial security number: " << getSocialSecurityNumber()
      << "\ngross sales: " << getGrossSales()
      << "\ncommission rate: " << getCommissionRate();
   return output.str();
}
```

If we decide to change the data member **names**, the earnings and toString definitions will *not* require modification - only the definitions of the *get* and *set* member functions that directly manipulate the data members will need to change.
These changes occur solely within the base class—no changes to the derived class are needed.

❖ Using a member function to access a data member's value can be slightly slower than accessing the data directly. However, today's optimizing compilers perform many optimizations **implicitly** (such as **inlining** set and get member-function calls). **You should write code that adheres to proper software engineering principles, and leave optimization to the compiler.**

```
// calculate earnings
double BasePlusCommissionEmployee::earnings() const {
   return getBaseSalary() + CommissionEmployee::earnings();
}

// return string representation of BasePlusCommissionEmployee object
string BasePlusCommissionEmployee::toString() const {
   ostringstream output;
   output << "base-salaried " << CommissionEmployee::toString()
      << "\nbase salary: " << getBaseSalary();
   return output.str();
}
```

Note the syntax used to invoke a redefined base-class member function from a derived class — place the **base-class name** and the **scope resolution operator (::)** before the base-class member-function name.

This member-function invocation is a good software engineering practice: **we avoid duplicating the code** and **reduce code-maintenance problems**.

- When a base-class member function is redefined in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the **::** operator prefixed with the name of the base class when referencing the base class's member function causes **infinite recursion**, because the derived-class member function would then call itself.

Inheritance

So, what is the good software engineering design with inheritance?

- Declaring base-class data members private (as opposed to declaring them protected) enables you to change the base-class implementation without having to change derived-class implementations.

- Declare the base-class data members as **private** and access them through set/get methods **everywhere**.

- It's appropriate to use the protected access specifier when a base class should provide a service (i.e., a non-private member function) only to its derived classes and friends.

Constructors/Destructors in Derived classes

Instantiating a derived-class object begins a *chain* of constructor calls in which the derived-class constructor, **before** performing its own tasks, **invokes** its direct base class's constructor either *explicitly* (via a base-class member initializer) or *implicitly* (calling the base class's default constructor).

Similarly, if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on.

The last constructor **called** in this chain is the one of the class at the base of the hierarchy, whose body actually finishes **executing** *first.*

The most-derived-class constructor's body finishes **executing** *last*.

When a program creates a derived-class object,

1. *the derived-class constructor immediately calls the base-class constructor,*
2. *the base-class constructor's body executes,*
3. *then the derived class's member initializers execute,*
4. *and finally the derived-class constructor's body executes.*

This process cascades up the hierarchy if it contains more than two levels.

Constructors/Destructors in Derived classes

When a derived-class object is destroyed, the program calls that object's destructor.

This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in ***reverse*** of the order in which **the constructors executed**.

When a derived-class object's destructor is called,

1. the destructor performs its task, then invokes the destructor of the next base class up the hierarchy.

2. This process repeats until the destructor of the final base class at the top of the hierarchy is called.

3. Then the object is removed from memory.

Constructors/Destructors in Derived classes

Suppose that we create an object of a derived class where both the base class and the derived class contain (via composition) objects of other classes.

When an object of that derived class is created,

1. the derived-class constructor immediately **calls** the base-class constructor,
2. first the constructors for the base class's member objects **execute**,
3. then the base class constructor body executes,
4. then the constructors for the derived class's member objects execute,
5. then the derived class's constructor body executes.
6. Destructors for derived-class objects are called in the **reverse** of the order in which their corresponding constructors are called.

Constructors/Destructors in Derived classes

```
class BaseMember {
public:
   BaseMember() {
      cout << "BaseMember::BaseMember" << endl;
   }
   ~BaseMember() {
      cout << "BaseMember::~BaseMember" << endl;
   }
};

class DerivedMember {
public:
   DerivedMember() {
      cout << "DerivedMember::DerivedMember" << endl;
   }
   ~DerivedMember() {
      cout << "DerivedMember::~DerivedMember" << endl;
   }
};
```

Constructors/Destructors in Derived classes

```
class Base {
public:
   Base() {
      cout << "Base::Base" << endl;
   }

   ~Base() {
      cout << "Base::~Base" << endl;
   }

private:
   BaseMember b;
};
```

Constructors/Destructors in Derived classes

```
class Derived : public Base {
public:
   Derived() : Base() { // we can skip Base() call since it is a default constructor call
      cout << "Derived::Derived" << endl;
   }
   ~Derived() {
      cout << "Derived::~Derived" << endl;
   }

private:
   DerivedMember d;
};


int main()
{
   Derived d;
   return 0;
}
```

<u>Result:</u>

BaseMember::BaseMember
Base::Base
DerivedMember::DerivedMember
Derived::Derived
Derived::~Derived
DerivedMember::~DerivedMember
Base::~Base
BaseMember::~BaseMember

Constructors/Destructors in Derived classes

By default, base-class constructors, destructors and overloaded assignment operators are ***not*** **inherited by derived classes**.
Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class versions.

```
class Base {
public:
    Base() {
        cout << "Base::Base" << endl;
    }
    Base(int i) {
        cout << "Base::Base(int i)" << endl;
    }
    Base(int i, double d) {
        cout << "Base::Base(int i, double d)" << endl;
    }
    Base(const Base& base) {
        cout << "Base::Base(const Base&)" << endl;
    }
};
```

Constructors/Destructors in Derived classes

```
class Derived : public Base {
};

int main()
{
   Derived d(1);
   return 0;
}
```

Compile error:

No matching function constructor for initialization of 'Derived'.
Candidate constructors: <u>copy constructor</u> and <u>default constructor.</u>

Constructors/Destructors in Derived classes

Sometimes a derived class's constructors simply *specify the same parameters as the base class's constructors* and simply pass the constructor arguments to the base-class's constructors.

For such cases, **C++11** allows you to specify that a **derived class should inherit** a base class's constructors.

To do so, *explicitly* include a **using** declaration of the form

using *BaseClass*::*BaseClass*;

anywhere(access specifier doesn't matter) in the derived-class definition.

With a few exceptions, for each constructor in the base class, the compiler generates a derived-class constructor that **calls** the corresponding base-class constructor.

Each generated constructor has the same name as the derived class.

The generated constructors perform only *default initialization* for the derived class's additional data members.

Constructors/Destructors in Derived classes

```
class Derived : public Base {
    using Base::Base;
};

int main()
{
   Derived d(1);
   return 0;
}
```

Result:
Base::Base(int i)

Constructors/Destructors in Derived classes

When you inherit constructors:

- Each generated constructor has the **same access specifier** (public, protected or private) as its corresponding base-class constructor.
- The **default** and **copy** are not inherited.
- If the derived class does not explicitly define constructors, the compiler still **generates a default constructor** in the derived class.
- A given base-class constructor is not inherited if a constructor that you explicitly define in the derived class **has the same parameter list**.
- A base-class constructor's **default arguments are not inherited**. Instead, the compiler generates overloaded constructors in the derived class. For example, if the base class declares the constructor

  BaseClass(int = 0, double = 0.0);
- The compiler generates the following derived-class constructors **without default arguments**

  DerivedClass();
  DerivedClass(int);
  DerivedClass(int, double);

These each **call** the *BaseClass* constructor that **specifies** the default arguments.

Function Overloading in Derived classes

```
class Base {
public:
   void f() {
      cout << "Base::f" << endl;
   }
   void f(int i) {
      cout << "Base::f(int i)" << endl;
   }
   void f(int i, double d) {
      cout << "Base::f(int i, double d)" << endl;
   }

};
class Derived : public Base {
public:
   void f(char c) {
      cout << "Derived::f(char)" << endl;
   }
};
```

Function Overloading in Derived classes

```
int main()
{
   Derived d;
   d.f(); // Error! No matching function to call Derived::f()
   return 0;
}
```

- Functions do not overload across scopes: **Base** class scope **f** functions were not able to overload the **Derived** class scope **f** functions. So, the **Base** class **f** functions were **hidden** in **Derived** class.

- **using-declarations** can be used to add a function to Derived class scope:

  using Base::f;

- A name brought into a derived class scope by a using-declaration has its **access** determined by the **placement** of the using-declaration.

- We cannot use using-directives to bring all members of a base class into a derived class.

Function Overloading in Derived classes

```
class Derived : public Base {
   using Base::f;
public:
   void f(char c) {
      cout << "Derived::f(char)" << endl;
   }
};

int main()
{
   Derived d;
   d.f(); // Error! f is a private member of derived
   return 0;
}
```

Function Overloading in Derived classes

```
class Derived : public Base {
public:
   using Base::f;
public:
   void f(char c) {
      cout << "Derived::f(char)" << endl;
   }
};

int main()
{
   Derived d;
   d.f();
   return 0;
}
```

Result:
Base::f

Types of Inheritance

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Object-Oriented Programming: Polymorphism

Polymorphism

Polymorphism enables you to "**program in the general**" rather than "**program in the specific**."

In particular, you can write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class.

With polymorphism, you can design and implement systems that are easily ***extensible***—new classes can be added with little or no modification to the general portions of the program.

The **same message** sent to a variety of objects has many forms of results—hence the term **polymorphism**.

Polymorphism enables you to deal in **generalities** and **let the execution-time environment** concern itself with the specifics.

You can direct a variety of objects to behave in manners appropriate to those objects **without even knowing their types**—as long as those objects belong to **the same inheritance hierarchy** and are being accessed off a **common base-class pointer or a common base-class reference**.

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

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 commissionEmployee and basePlusCommissionEmployee
   cout << "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:\n"
      << commissionEmployee.toString() // base-class toString
      << "\n\n"
      << basePlusCommissionEmployee.toString(); // derived-class toString
```

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

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

   // aim base-class pointer at derived-class object
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCALLING TOSTRING WITH BASE-CLASS POINTER TO "
      << "DERIVED-CLASS OBJECT\nINVOKES BASE-CLASS TOSTRING "
      << "FUNCTION ON THAT DERIVED-CLASS OBJECT:\n"
      << commissionEmployeePtr->toString() // base version
      << endl;
}
```

```
DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:
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: 300.00

CALLING 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 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: 300.00

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

Polymorphism

The output of each toString member-function invocation in this program reveals that the invoked functionality depends on the **type of the pointer** (or **reference**, as you'll soon see) used to invoke the function, NOT the **type of the object for which the member function is called**.

When we introduce **virtual** functions, we demonstrate that it's possible to invoke the **object type's functionality**, rather than **invoke the pointer type's** (or reference type's) functionality.

Polymorphism

```
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main() {
7     CommissionEmployee commissionEmployee{
8        "Sue", "Jones", "222-22-2222", 10000, .06};
9
10    // aim derived-class pointer at base-class object
11    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
12    BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
13       &commissionEmployee};
14 }
```

Microsoft Visual C++ compiler error message:

```
c:\examples\ch12\fig12_02\fig12_02.cpp(13):
   error C2440: 'initializing': cannot convert from 'CommissionEmployee *'
   to 'BasePlusCommissionEmployee *'
```

```cpp
int main() {
   BasePlusCommissionEmployee basePlusCommissionEmployee{
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300};

   // aim base-class pointer at derived-class object (allowed)
   CommissionEmployee* commissionEmployeePtr{&basePlusCommissionEmployee};

   // invoke base-class member functions on derived-class
   // object through base-class pointer (allowed)
   string firstName{commissionEmployeePtr->getFirstName()};
   string lastName{commissionEmployeePtr->getLastName()};
   string ssn{commissionEmployeePtr->getSocialSecurityNumber()};
   double grossSales{commissionEmployeePtr->getGrossSales()};
   double commissionRate{commissionEmployeePtr->getCommissionRate()};

   // attempt to invoke derived-class-only member functions
   // on derived-class object through base-class pointer (disallowed)
   double baseSalary{commissionEmployeePtr->getBaseSalary()};
   commissionEmployeePtr->setBaseSalary(500);
}
```

GNU C++ compiler error messages:

```
fig12_03.cpp:26:45: error: ‘class CommissionEmployee’ has no member named
‘getBaseSalary’
    double baseSalary{commissionEmployeePtr->getBaseSalary()};
                                             ^
fig12_03.cpp:27:27: error: ‘class CommissionEmployee’ has no member named
‘setBaseSalary’
    commissionEmployeePtr->setBaseSalary(500);
```

Virtual Functions

Recall that the type of **the handle determines** which class's functionality to invoke.

In that case, the **CommissionEmployee** pointer invoked the **CommissionEmployee** member function **toString** on the **BasePlusCommissionEmployee** object, <u>even though the pointer was aimed at a **BasePlusCommissionEmployee** object</u> that has its own custom **toString** function.

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*** <u>***return type***</u> (i.e., *prototype*) as the function it overrides in its base class.

If we <u>*do not* declare</u> 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 override keyword** to the prototype of every derived-class function 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 an error.

Not only does this ensure that you override the base-class function with the appropriate signature, it also prevents you from accidentally hiding a base-class function 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
```

```
   // 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 <u>process dynamically allocated objects of a class hierarchy</u>.

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 <u>C++ standard</u> specifies that the behavior is <u>*undefined*</u>**.

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.