

Inheritance

Chapter 15
& additional topics

Overview

- ❑ Inheritance Introduction
- ❑ Three different kinds of inheritance
- ❑ Changing an inherited member function
- ❑ More Inheritance Details
- ❑ Polymorphism

Inheritance Introduction

Motivating Example: Employee Classes

- Design a record-keeping program with records for **salaried** and **hourly employees**
 - **Salaried** and **hourly employees** belong to a class of people who share the property "**employee**"
 - **Salaried employee**
 - A subset of employees with a fixed wage
 - **Hourly employees**
 - Another subset of employees earn hourly wages
- All employees have a name and SSN
 - Functions to manipulate name and SSN are the same for hourly and salaried employees

- ❑ First define a class called `Employee` for all kinds of employees
- ❑ The `Employee` class will be used later to define classes for `hourly` and `salaried` employees

```
class Employee
```

```
{
```

```
public:
```

```
    Employee( );
```

```
    Employee(string the_name, string the_ssn);
```

```
    string get_name( ) const;
```

```
    string get_ssn( ) const;
```

```
    double get_net_pay( ) const;
```

```
    void set_name(string new_name);
```

```
    void set_ssn(string new_ssn);
```

```
    void set_net_pay(double new_net_pay);
```

```
    void print_check( ) const;
```

```
private:
```

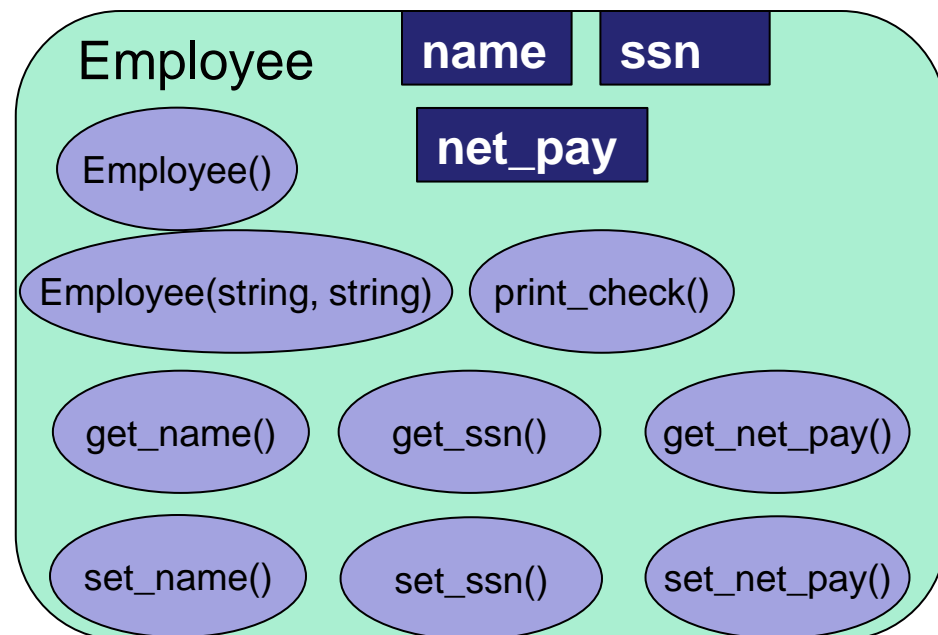
```
    string name;
```

```
    string ssn;
```

```
    double net_pay;
```

```
};
```

employee.h



We now use Employee class to create an
HourlyEmployee class

```
class HourlyEmployee : public Employee
```

```
{
```

```
public:
```

```
    HourlyEmployee( );
```

```
    HourlyEmployee(string the_name, string the_ssn,  
                    double the_wage_rate, double the_hours);
```

```
    void set_rate(double new_wage_rate);
```

```
    double get_rate( ) const;
```

```
    void set_hours(double hours_worked);
```

```
    double get_hours( ) const;
```

```
    void print_check( ) ;
```

```
private:
```

```
    double wage_rate;
```

```
    double hours;
```

```
};
```

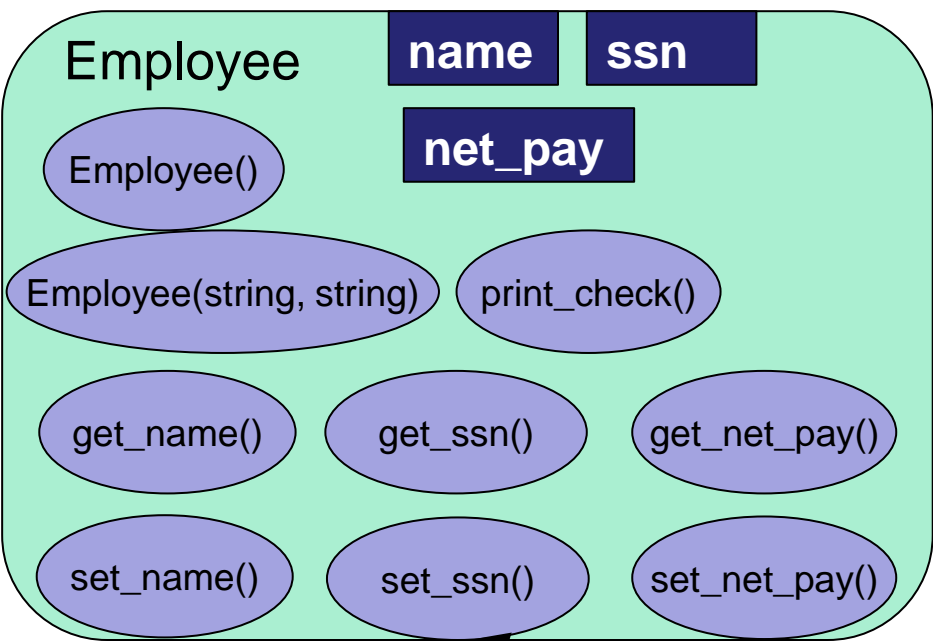
see book
Display 15.3

hourlyemployee.h

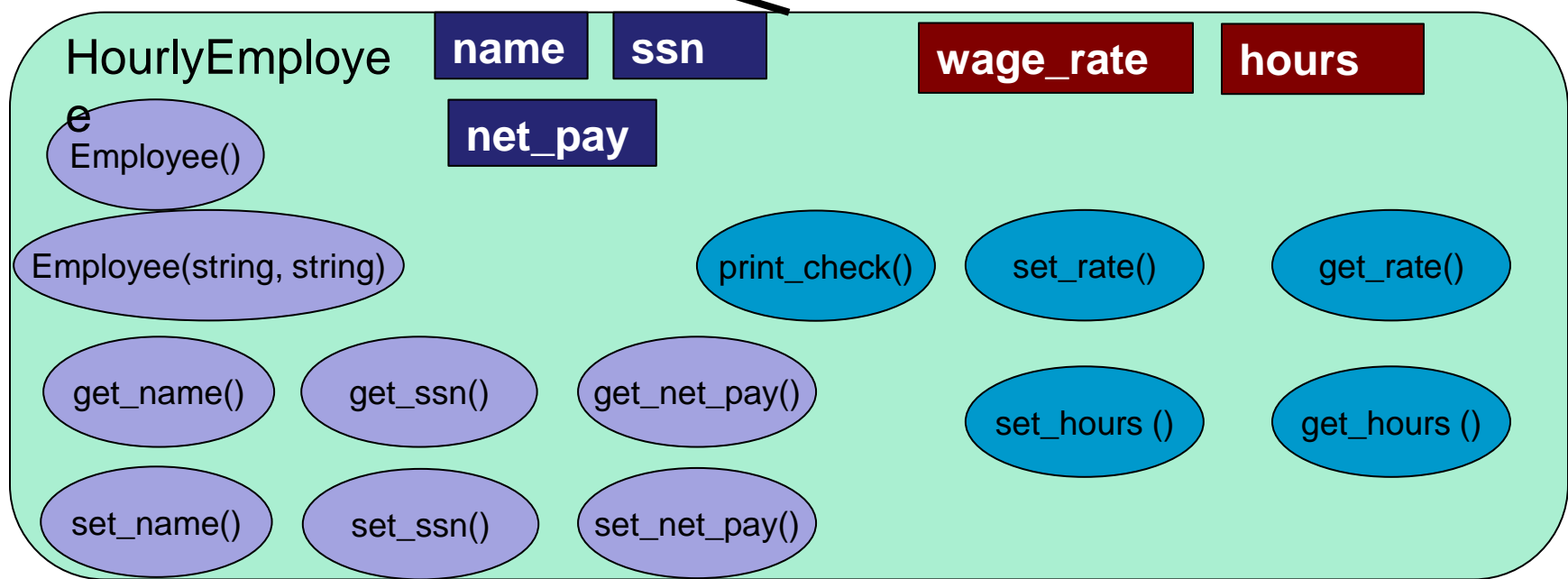
- HourlyEmployee is derived from Class Employee
- HourlyEmployee inherits all member functions and member variables of Employee
 - NOT SHOWN explicitly in HourlyEmployee's defn
- The class definition begins

```
class HourlyEmployee : public Employee
```

 - note that :public Employee shows that HourlyEmployee is derived from class Employee
- HourlyEmployee declares additional member variables wage_rate and hours



is



- Inheritance :
a new class, called a derived class, is created from another class (i.e., the base class)
- A derived class automatically has all the member variables and functions of the base class
- A derived class can have additional member variables and/or member functions

A derived class automatically **has** all the member variables and functions of the base class.

But, the derived class **might not** have the same access rights as the base class when accessing those inherited members! (To be discussed soon...)

Inherited Members

- ❑ A derived class inherits all the members (data members, functions) of the parent class
- ❑ The derived class should **not** re-declare or re-define a member function inherited from the parent **unless** ...
 - The derived class wants to use the inherited member function for doing something different
- ❑ The derived class can add member variables & member functions

Display 15.3

```
class HourlyEmployee : public Employee  
{
```

```
public:
```

```
    HourlyEmployee( );
```

```
    HourlyEmployee(string the_name, string the_ssn,  
                    double the_wage_rate, double the_hours);
```

```
    void set_rate(double new_wage_rate);
```

```
    double get_rate( ) const;
```

```
    void set_hours(double hours_worked);
```

```
    double get_hours( ) const;
```

hourlyemployee.h

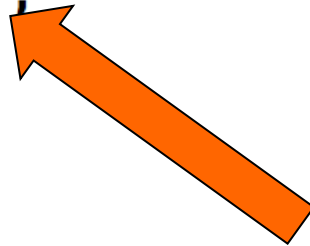
```
    void print_check( ) ;
```

```
private:
```

```
    double wage_rate;
```

```
    double hours;
```

```
};
```



Only list the declaration of an inherited member function if you want to change the defn of the function.

Why re-define print_check() ?

A practical concern here...

- `print_check` will have different definitions to print different checks for each type of employee
 - An `Employee` object lacks sufficient information to print a check
 - Each derived class will have sufficient information to print a check

Implementation for the Base Class Employee (part 1 of 2)

```
//This is the file: employee.cpp.  
//This is the implementation for the class Employee.  
//The interface for the class Employee is in the header file employee.h.  
#include <string>  
#include <cstdlib>  
#include <iostream>  
#include "employee.h"  
using namespace std;  
  
namespace employeessavitch  
{  
    Employee::Employee( ) : name("No name yet"), ssn("No number yet"), net_pay(0)  
    {  
        //deliberately empty  
    }  
  
    Employee::Employee(string the_name, string the_number)  
        : name(the_name), ssn(the_number), net_pay(0)  
    {  
        //deliberately empty  
    }  
  
    string Employee::get_name( ) const  
    {  
        return name;  
    }  
  
    string Employee::get_ssn( ) const  
    {  
        return ssn;  
    }  
}
```

employee.cpp

Implementation for the Base Class Employee (part 2 of 2)

```
double Employee::get_net_pay( ) const
{
    return net_pay;
}

void Employee::set_name(string new_name)
{
    name = new_name;
}

void Employee::set_ssn(string new_ssn)
{
    ssn = new_ssn;
}

void Employee::set_net_pay (double new_net_pay)
{
    net_pay = new_net_pay;
}

void Employee::print_check( ) const
{
    cout << "\nERROR: print_check FUNCTION CALLED FOR AN \n"
        << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
        << "Check with the author of the program about this bug.\n";
    exit(1);
}

} //employeeessavitch
```

employee.cpp

Implementing a Derived Class

- ❑ Any member function added in the derived class are defined in the implementation file for the derived class
 - Definitions are not given for inherited functions that are not to be changed
- ❑ The `HourlyEmployee` class is implemented in `HourlyEmployee.cpp`

Textbook Display 15.5

Implementation for the Derived Class HourlyEmployee (part 1 of 2)

```
//This is the file: hourlyemployee.cpp
//This is the implementation for the class HourlyEmployee.
//The interface for the class HourlyEmployee is in
//the header file hourlyemployee.h.
#include <string>
#include <iostream>
#include "hourlyemployee.h"
using namespace std;

namespace employeessavitch
{
    HourlyEmployee::HourlyEmployee( ) : Employee( ), wage_rate(0), hours(0)
    {
        //deliberately empty
    }

    HourlyEmployee::HourlyEmployee(string the_name, string the_number,
                                   double the_wage_rate, double the_hours)
    : Employee(the_name, the_number), wage_rate(the_wage_rate), hours(the_hours)
    {
        //deliberately empty
    }

    void HourlyEmployee::set_rate(double new_wage_rate)
    {
        wage_rate = new_wage_rate;
    }

    double HourlyEmployee::get_rate( ) const
    {
        return wage_rate;
    }
}
```

Display 15.5 (1/2)

Display 15.5 (2/2)

```
void HourlyEmployee::set_hours(double hours_worked)
{
    hours = hours_worked;
}
```

```
double HourlyEmployee::get_hours( ) const
{
    return hours;
}
```

We have chosen to set net_pay as part of the print_check function since that is when it is used, but in any event, this is an accounting question, not a programming question. But note that C++ allows us to drop the const in the function print_check when we redefine it in a derived class.

```
void HourlyEmployee::print_check( )
{
    set_net_pay(hours * wage_rate);

    cout << "\n_____ \n";
    cout << "Pay to the order of " << get_name( ) << endl;
    cout << "The sum of " << get_net_pay( ) << " Dollars\n";
    cout << "_____ \n";
    cout << "Check Stub: NOT NEGOTIABLE\n";
    cout << "Employee Number: " << get_ssn( ) << endl;
    cout << "Hourly Employee. \nHours worked: " << hours
        << " Rate: " << wage_rate << " Pay: " << get_net_pay( ) << endl;
    cout << "_____ \n";
}
```

We now use `Employee` class to create
an `SalariedEmployee` class

Class SalariedEmployee

```
class SalariedEmployee : public Employee
{
public:
    SalariedEmployee( );
    SalariedEmployee (string the_name,
                      string the_ssn,
                      double the_weekly_salary)
    double get_salary( ) const;
    void set_salary(double new_salary);
    void print_check( );
private:
    double salary;//weekly
};
```

salariedemployee.h

- The class `SalariedEmployee` is also derived from `Employee`
 - Function `print_check` is redefined to have a meaning specific to salaried employees
 - `SalariedEmployee` adds a member variable `salary`

Implementation for the Derived Class SalariedEmployee (part 1 of 2)

```
//This is the file salariedemployee.cpp.  
//This is the implementation for the class SalariedEmployee.  
//The interface for the class SalariedEmployee is in  
//the header file salariedemployee.h.  
#include <iostream>  
#include <string>  
#include "salariedemployee.h"  
using namespace std;  
  
namespace employeessavitch  
{  
    SalariedEmployee::SalariedEmployee( ) : Employee( ), salary(0)  
    {  
        //deliberately empty  
    }  
  
    SalariedEmployee::SalariedEmployee(string the_name, string the_number,  
                                       double the_weekly_salary)  
        : Employee(the_name, the_number), salary(the_weekly_salary)  
    {  
        //deliberately empty  
    }  
  
    double SalariedEmployee::get_salary( ) const  
    {  
        return salary;  
    }  
  
    void SalariedEmployee::set_salary(double new_salary)  
    {  
        salary = new_salary;  
    }  
}
```

Display 15.6 (1/2)

Display 15.6

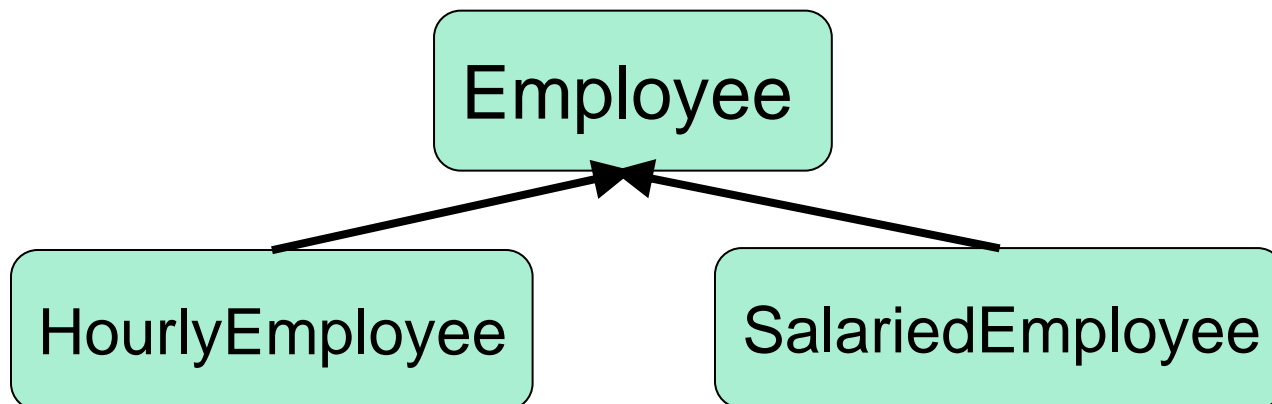
(2/2)

Implementation for the Derived Class SalariedEmployee (*part 2 of 2*)

```
void SalariedEmployee::print_check( )
{
    set_net_pay(salary);
    cout << "\n_____ \n";
    cout << "Pay to the order of " << get_name( ) << endl;
    cout << "The sum of " << get_net_pay( ) << " Dollars\n";
    cout << "_____ \n";
    cout << "Check Stub NOT NEGOTIABLE \n";
    cout << "Employee Number: " << get_ssn( ) << endl;
    cout << "Salaried Employee. Regular Pay: "
        << salary << endl;
    cout << "_____ \n";
}
} //employee savitch
```

Parent and Child Classes

- ❑ Recall that a child class automatically has all the members of the parent class
- ❑ The parent class is an ancestor of the child class
- ❑ The child class is a descendent of the parent class
- ❑ The parent class (**Employee**) contains all the code common to the child classes
 - You do not have to re-write the code for each child



Is-A relationship is one way

- ❑ The child is-a version of the parent, but the parent is **NOT** a version of the child.
- ❑ The child has **ALL** features of the parent but the child has **ADDED** features.
- ❑ Only the child knows about the parent. The parent does not know about the children.

Parent and Child Classes (cont'd)

- An hourly employee *is* an employee
 - An object of type `HourlyEmployee` can be used wherever an object of type `Employee` can be used
 - An object of a class type can be used wherever any of its ancestors can be used
 - An ancestor cannot be used in a place where one of its descendants is expected

```
void fun1(Employee x);  
void fun2(HourlyEmployee y);  
int main()  
{  
    Employee a;  
    HourlyEmployee b;  
    fun1(a); //correct  
    fun1(b); //correct  
    fun2(a); //incorrect  
    fun2(b); //correct  
}
```

public inheritance is an
is-a relationship

Derived Class's Constructors

- ❑ A base class's constructor is **not** inherited in a derived class
- ❑ The base class constructor can be invoked by the constructor of the derived class
- ❑ The constructor of a derived class begins by invoking the constructor of the base class in the initialization section:

```
HourlyEmployee::HourlyEmployee : Employee( ), wage_rate(  
    0), hours(0)  
{ //no code needed }
```



Call a constructor for Employee

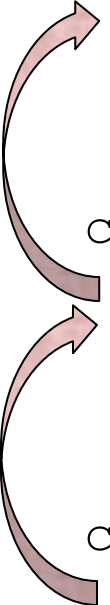
Default Initialization

- If a derived class constructor does not invoke a base class constructor explicitly, the base class **default** constructor will be used automatically
- If class B is derived from class A and class C is derived from class B
 - When an object of class C is created
 - The base class A's constructor is the first invoked
 - Class B's constructor is invoked next
 - C's constructor completes execution

What is the order?

```
int main()  
{  
    C c;  
}
```

```
class A {  
    public:  
        A() { cout << "A()" << " "; }  
};  
class B : public A {  
    public:  
        B(): A() { cout << "B()" << " "; }  
};  
class C : public B {  
    public:  
        C() : B() { cout << "C()" << " "; }  
};  
A() B() C()
```



Private is Private

- ❑ A member variable (or function) that is private in the parent class is **not directly accessible by the member functions in the child class**
- ❑ This code is illegal as `net_pay` is a private member of `Employee`!

```
void HourlyEmployee::print_check( )  
{  
    net_pay = hours * wage_rate;  
}
```

- ❑ The parent class member functions must be used to access the private members of the parent

A member function of a class can NOT directly access its own member variable (inherited, private to its base class)!

The protected Qualifier

- `protected` members of a class appear to be private outside the class, but are directly accessible within a derived classes
- If member variables name, `net_pay`, is listed as `protected` (not `private`) in the `Employee` class, this code becomes legal:

```
HourlyEmployee::print_check( )  
{  
    net_pay = hours * wage_rate;
```

access_specifiers_demo.cpp

Using protected or not?

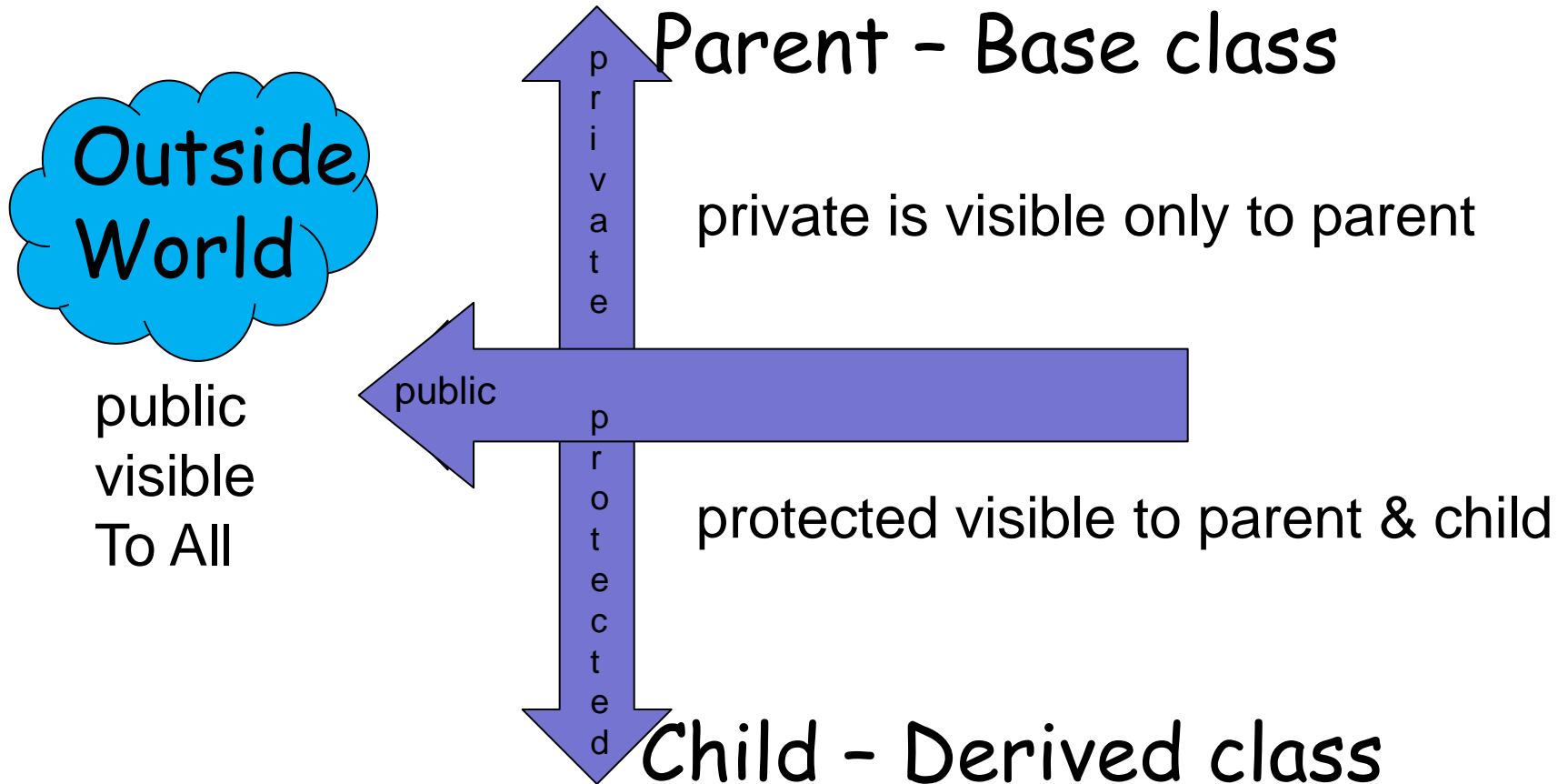
- ❑ Using **protected** members of a class is a convenience to facilitate writing the code of derived classes.
- ❑ Protected members are not necessary
 - Derived classes can use the public methods of their ancestor classes to access private members
- ❑ Many programming authorities consider it bad style to use protected member variables
 - It breaks encapsulation forcing changes in the base class to be handled in children.

Overview

- ❑ Inheritance Introduction
- ❑ Three different kinds of inheritance
- ❑ Changing an inherited member function
- ❑ More Inheritance Details
- ❑ Polymorphism

Three different kinds of inheritance

Inherit Visibility of Access Qualifiers



Three different ways for classes to inherit from other classes: public, private, and protected.

// Inherit from Base publicly

class D1: public Base

{ };

// Inherit from Base privately

class D2: private Base

{ };

// Inherit from Base protectedly

class D3: protected Base

{ };

class D4: Base // Defaults to private inheritance

{ };

If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).

Public inheritance

// Inherit from Base publicly

class D1: public Base

{ };

- All members keep their original access specifications.
- Private members stay private, protected members stay protected, and public members stay public.

public inheritance

Base class access specifier	Derived class access specifier (implicitly given)	Directly accessible in member functions of derived class?	Directly accessible in any other code?
public	public	yes	yes
private	private	no	no
protected	protected	yes	no

Private inheritance

// Inherit from Base privately

class D2: private Base

{ };

- All members from the base class are inherited as private.
 - private members stay private, and protected and public members become private.
- This does not affect that way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

private inheritance

Base class access specifier	Derived class access specifier (implicitly given)	Directly accessible in member functions of derived class?	Directly accessible thru objects of derived class in any other code?
public	private	yes	no
private	private	no	no
protected	private	yes	no

Protected inheritance

// Inherit from Base protectedly

class D3: protected Base

{ };

- Rarely used. The public and protected members become protected, and private members stay private.

protected inheritance

Base class access specifier	Derived class access specifier (implicitly given)	Directly accessible in member functions of derived class?	Directly accessible thru objects of derived class in any other code?
public	protected	yes	no
private	private	no	no
protected	protected	yes	no

- Member functions of a derived classes have access to its inherited members based **ONLY** on the access specifiers of its immediate parent, not affected by the inheritance method used!

Base class access specifier for members	Derived class access specifiier (implicitly given for inherited members)	Directly accessible in member functions of derived class?	Directly accessible in any other code?
public inheritance			
public	public	yes	yes
private	private	no	no
protected	protected	yes	no
private inheritance			
public	private	yes	no
private	private	no	no
protected	private	yes	no
protected inheritance			
public	protected	yes	no
private	private	no	no
protected	protected	yes	no

Access Qualifiers

```
class A
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
```

Outside of class:

public maintains access 'as-is'	// x is private
protected makes it all inaccessible	// y is private
private makes it all inaccessible	// z is not accessible from D

```
class B : public A    // B's children can access x,y
{
    // x is public
    // y is protected
    // z is not accessible from B
};
class C : protected A // C's children can access x,y
{
    // x is protected
    // y is protected
    // z is not accessible from C
};
class D : private A { // defaults to 'private'
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

IMPORTANT NOTE: Classes B, C and D all contain the variables x, y and z. It is just a question of access

Inheritance Qualifiers

- ❑ When derived public, all qualifiers of the parent stay the same in the derived.
 - "As-is" inheritance.
- ❑ When derived protected, both public and protected base members become protected in the derived.
 - Not visible outside, but visible to children
- ❑ When derived private, both public and protected become private in the derived.
 - Not visible outside, not visible to children.

Overview

- ❑ Inheritance Introduction
- ❑ Three different kinds of inheritance
- ❑ Changing an inherited member function
- ❑ More Inheritance Details
- ❑ Polymorphism

Changing an inherited member
function

Redefinition of Member Functions

- ❑ When defining a derived class, list the inherited functions that you wish to change for the derived class
 - The function is declared in the class definition
 - `HourlyEmployee` and `SalariedEmployee` each have their own definitions of `print_check`
- ❑ Next page demonstrates the use of the derived classes defined in `HourlyEmployee.h` and `SalariedEmployee.h`.

Functions defined in Employee class

set_name()

set_ssn()

print_check()

Functions defined in HourlyEmployee class

set_rate()

set_hours()

print_check()

```
int main( )
{
    HourlyEmployee joe;
    joe.set_name("Mighty Joe");
    joe.set_ssn("123-45-6789");
    joe.set_rate(20.50);
    joe.set_hours(40);
    cout << "Check for " << joe.get_name( )
          << " for " << joe.get_hours( ) << " hours.\n";
    joe.print_check( );
    cout << endl;

    SalariedEmployee
        boss("Mr. Big Shot", "987-65-4321", 10500.50);
    cout << "Check for " << boss.get_name( ) << endl;
    boss.print_check( );

    return 0;
}
```

Redefining vs. Overloading

- A function **redefined** in a derived class has the same number and type of parameters
 - The prototype (return value, function name, and parameters) of the function in the derived class must be exactly identical to that in the base class.
 - The derived class has only one function with the same name as the base class
 - An **overloaded** function has a different number and/or type of parameters than the base class
 - For example, the derived class has two functions with the same name as the base class: one is overloading, one is redefining.
- ```
void set_name(string first_name, string last_name); //overloading
void set_name(string new_name); //redefine
```

# A side note: function signatures

- An overloaded function has multiple signatures
  - A function signature is the function's **name** with **the sequence of types** in the parameter list, not including any **const** or **'&'**
  - Some compilers allow overloading based on including const or not including const

# Change access specifier for an inherited member (data or function)

- When re-define a function in a derived class,
  - The re-defined function uses whatever access specifier it is given in the derived class
  - The re-defined function does not inherit the access specifier of the function with the same prototype in the base class.
- Therefore,
  - You can hide an inherited member (originally public in base class) by specifying it as private or protected
  - You can expose an inherited member (originally protected in base class) by specifying it as public
- However, you can only change the access specifiers of base members the class are accessible in the derived class.
  - You can never change the access specifier of a base member from private to protected or public, because derived classes do not have access to private members of the base class.

# Changing Access Qualifiers in Child

- ❑ A base class member that is private
  - CANNOT BE CHANGED!
- ❑ A base class member that is protected
  - Can be changed to public or private in child.
- ❑ A base class member that is public
  - Can be changed to protected or private in child.

# Access to a Redefined Base Function

- When a function of a base class is redefined in a derived class, the base class function can still be used
  - To specify that you want to use the base class version of the redefined function:

```
int main()
{
 HourlyEmployee sally_h;
 sally_h.Employee::print_check();
}
```

# Hierarchy with DayOfYear to Date

```
❑ class DayOfYear {
 public:
 DayOfYear(int mon, int day);
 void set(int mon, int day);
 private:
 int month, day;
};
class Date : public DayOfYear {
 public:
 Date(int mon, int day, int y) : DayOfYear(mon, day), year(y) {}
 using DayOfYear::set; // Make visible in Child
 void set(int mon, int day, int y);
 private:
 int year;
};
```

# Overloaded function between classes

```
int main() {
 Date birthday(4, 19, 2001);

 birthday.set(4, 17); // Can't do this, hidden
 birthday.DayOfYear::set(4,17);

}
```

# Adding new functionality to an inherited member function

- See `derived_changes_inherited_members.cpp` file.

```
void HourlyEmployee::print_check()
{
 set_net_pay(hours*rate);
 Employee::print_check();
 cout << "Hours worked: " << hours;
 ...
}
```

# Hide the functionality of an inherited member function

Hide the functionality of an inherited member function from any other code.

- ❑ Redefine the member function (discussed before)
- ❑ Or, give it a new access specifier `private` when re-defining it in the derived class.
- ❑ Or, simply list it in the private section like this:

```
class Circle : public Shape()
{
private:
```

```
 using Shape::display;
```

```
 //display() is a function defined as public in Shape
```

without even re-defining it.

See  
[derived\\_changes\\_inherited\\_members.cpp](#) file

# Expose an inherited member (originally protected in base class)

- ❑ Read this file  
[derived\\_exposes\\_inherited\\_members.cpp](#)

stopped here 4/4.

# Overview

- ❑ Inheritance Introduction
- ❑ Three different kinds of inheritance
- ❑ Changing an inherited member function
- ❑ More Inheritance Details
- ❑ Polymorphism

# Review what was covered so far...

- ❑ What is the relationship between parent class and child class?
- ❑ What are some names for parent class and some names for child class?
- ❑ What is an access qualifier? Name them.
- ❑ When should a member function that is defined in the parent be included in the child declaration?
- ❑ How do we access member functions in the parent class from child objects?

# More Inheritance Details

- ❑ Some special functions are not inherited by a derived class. They include
  - The assignment operator
  - Copy constructors
  - Destructors
- ❑ Even though they are not inherited derived classes still benefit from the base class.
- ❑ Derived classes only have to implement the big three when they allocate memory.

# The Assignment Operator

- In implementing an assignment operator (`operator=`) in a derived class
  - It is normal to use the assignment operator from the base class in the definition of the derived class's assignment operator
  - Recall that the assignment operator is written as a member function of a class
  - Only implement the assignment operator in the derived class if it is required in the derived class.
  - The default behavior is correct.

# The Operator = Implementation

- This code segment shows how to begin the implementation of the **=** operator for a derived class when required:

```
Derived& Derived::operator= (const Derived& rhs)
{
 Base::operator=(rhs);
```

```
/*
```

Base is the name of the parent class

This line handles the assignment of the inherited member variables by calling the base class assignment operator

The remaining code would assign the member variables introduced in the derived class

```
*/
```

# Operator = and Derived Classes

- If a base class has a defined assignment `operator=` but the derived class does not, then
  - The derived class `operator=` will automatically call the base class version.
  - That's because there is only one assignment `operator=` in any class.

# Copy Constructors and Derived Classes

- ❑ If a copy constructor is not defined in a derived class, C++ will generate a default copy constructor
  - This copy constructor calls the base class copy constructor by default.
  - Since there is only one copy constructor, the derived class calls the one that handles dynamic memory.

# The Copy Constructor

- Implementation of the derived class copy constructor is much like that of any other derived class constructor, **pass up the parameter to the base** in the initializer section:

```
Derived::Derived(const Derived& object)
 :Base(object), <other initializing>
{...}
```

- Invoking the base class copy constructor sets up the inherited member variables
  - Since object is of type `Derived` it is also of type `Base`

# Destructors and Derived Classes

- ❑ A destructor is not inherited by a derived class
- ❑ The derived class should define its own destructor if necessary
- ❑ If the base class defines a destructor, there is no need to define a destructor in the derived class unless it is needed for dynamic member variables declared in the derived class.

# Destructors in Derived Classes

- If the base class has a programmer-defined destructor, then defining the destructor for the derived class is relatively easy
  - When the destructor for a derived class is called, the destructor for the base class is **automatically** called
  - The derived class destructor only need to release memory for the dynamic variables added in the derived class

# Destruction Sequence

- ❑ A is at the root of the hierarchy.
- ❑ If class B is derived from class A and class C is derived from class B...
  - When an object of class C goes out of scope
    - The destructor of class C is called
    - Then the destructor of class B
    - Then the destructor of class A
  - Notice that destructors are called in the reverse order of constructor calls

# Overview

- ❑ Inheritance Introduction
- ❑ Three different kinds of inheritance
- ❑ Changing an inherited member function
- ❑ More Inheritance Details
- ❑ Polymorphism

Polymorphism

# Polymorphism

- ❑ **Polymorphism** refers to the ability to associate multiple definitions with one function declaration using a mechanism called **late binding**
- ❑ Polymorphism is a key component of the philosophy of object oriented programming

# Binding & Early binding

- **Binding**

The process to convert identifiers (such as variable and function names) into machine language addresses.

- **Early binding** (or static binding)

An C++ compiler directly associates an identifier name (such as a function or variable name) with a machine address during compilation process.

Note that all functions have a unique machine address.

When the compiler encounters a function call, it replaces the function call with an instruction that tells the CPU to jump to the address of the function.

- **Late binding** (or dynamic binding)

- To be discussed very soon...

## Functions defined in Employee class

set\_name()  
set\_ssn()  
print\_check()

## Functions defined in HourlyEmployee class

set\_rate()  
set\_hours()  
print\_check()

## Compile-time binding

```
int main()
{
 HourlyEmployee joe;
 joe.set_name("Mighty Joe");
 joe.set_ssn("123-45-6789");
 joe.set_rate(20.50);
 joe.set_hours(40);
 cout << "Check for " << joe.get_name()
 << " for " << joe.get_hours() << " hours.\n";
 joe.print_check();
 cout << endl;

 SalariedEmployee
 boss("Mr. Big Shot", "987-65-4321", 10500.50);
 cout << "Check for " << boss.get_name() << endl;
 boss.print_check();

 return 0;
}
```

# A motivating example

- ❑ Imagine a graphics program with several types of figures
- ❑ Each figure may be an object of a different class, such as a `circle`, `oval`, `rectangle`, etc.
- ❑ Each is a descendant of a class `Figure`
- ❑ Each has a function `draw( )` implemented with code specific to each shape
- ❑ Class `Figure` has functions common to all figures

```
class Figure
```

```
public:
```

```
center()
```

```
{ ...
```

```
draw()
```

```
...
```

```
}
```

```
draw()
```

```
c.center();
```

When a member function is called with a derived class object, the compiler first looks to see if that member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the inherited classes or the top base class. It uses the first one it finds.

```
Circle
```

```
public:
```

```
draw()//re-defined
```

```
//center() is inherited
```

```
Triangle
```

```
public:
```

```
draw()//re-defined
```

```
//center() is inherited
```

```
int man()
```

```
{
```

```
Circle c;
```

```
c.draw(); //which draw() is called?
```

```
c.center(); //which draw() is called inside center()?
```

```
}
```

Look at : [figure\\_demo.cpp](#)

# A Problem

- ❑ Class `Figure` has a function `center`
  - Function `center` moves a figure to the center of the screen by erasing the figure and redrawing it in the center of the screen
- ❑ Function `center` is inherited by each of the derived classes
  - Function `center` SHOULD use each derived object's `draw` function to draw the figure
  - But, `Figure` class does not know about its derived classes, so how can it know how to invoke a derived object's `draw` function?

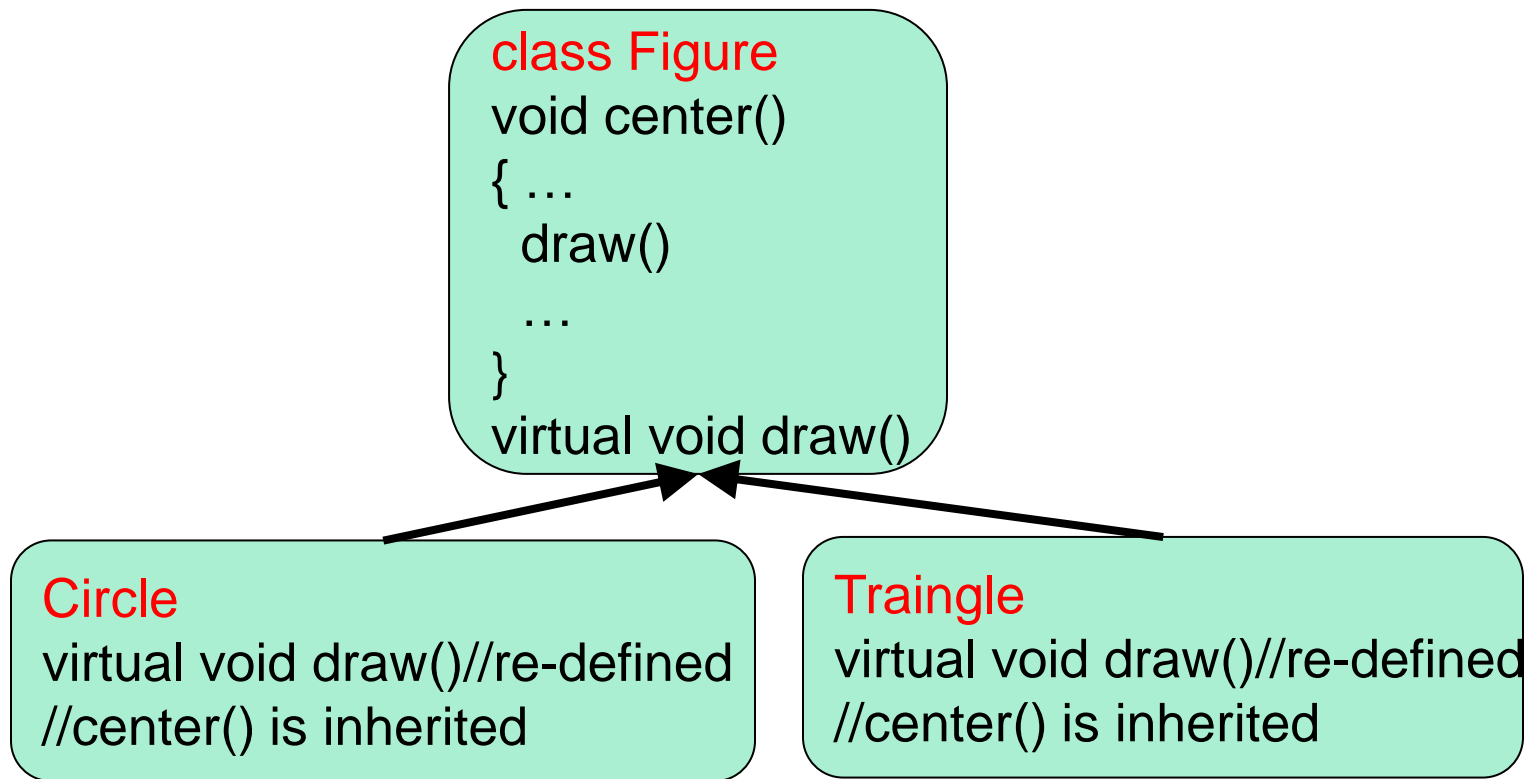
# Virtual Functions

- ❑ **Virtual functions** can be used to address the previous problem.
- ❑ Making a function **virtual** tells the compiler that ...
  - The programmer doesn't know how the function is implemented when the programmer is defining a base class
  - The programmer wants to wait until the function of an object is used in a program. Only at that time, the implementation of the function is clear, i.e., it is given by the class type of the object.
- ❑ This is called **late binding**

# How to use virtual functions?

- ❑ Add keyword **virtual** to a function's declaration in the base class
  - ❑ **virtual** is not added to the function definition
- ❑ Define the function differently in a derived class
  - This is the intention of introducing virtual function
- ❑ **virtual** is not needed for the function declaration in the derived class, but is often included
- ❑ Note that, virtual functions require considerable overhead so excessive use reduces program efficiency
  - However, don't sacrifice design for efficiency.

figure\_demo\_virtual.cpp



```
int main()
{
 Circle c;
 c.draw() //which draw() is called?
 c.center() //which draw() is called inside center()?
}
```

Look at :  
`figure_demo_virtual.cpp`

# Another Example of Virtual Functions

- ❑ As another example, let's design a record-keeping program for an **auto parts store**
- ❑ We want to introduce a **bill** function, and we want a versatile program, and we do not know all the possible types of sales we might have to account for
  - Later we may add **mail-order** and **discount** sales
  - Functions to compute bills will have to be added later when we know what type of sales to add
  - To accommodate the future possibilities, we will make the **bill** function a virtual function

# The Sale Class

- ❑ All sales will be derived from the base class `Sale`
- ❑ The `bill` function of the `Sale` class is virtual
- ❑ The `Sale` class interface and implementation are shown in

**Display 15.8**

**Display 15.9**

## Interface for the Base Class Sale

---

```
//This is the header file sale.h.
//This is the interface for the class Sale.
//Sale is a class for simple sales.
#ifndef SALE_H
#define SALE_H

#include <iostream>
using namespace std;

namespace salesavitch
{

 class Sale
 {
 public:
 Sale();
 Sale(double the_price);
 virtual double bill() const;
 double savings(const Sale& other) const;
 //Returns the savings if you buy other instead of the calling object.
 protected:
 double price;
 };

 bool operator < (const Sale& first, const Sale& second);
 //Compares two sales to see which is larger.

} //salesavitch

#endif // SALE_H
```

## Display 15.8

## The Derived Class DiscountSale

---

```
//This is the interface for the class DiscountSale.
#ifndef DISCOUNTSALE_H
#define DISCOUNTSALE_H
#include "sale.h"
```

# Sale, DiscountSale

```
namespace salesavitch
{
 class DiscountSale : public Sale
 {
 public:
 DiscountSale();
 DiscountSale(double the_price, double the_discount);
 //Discount is expressed as a percent of the price.
 virtual double bill() const;
 protected: ←
 double discount;
 };
} //salesavitch
#endif //DISCOUNTSALE_H
```

*This is the file discountsale.h.*

*The keyword virtual is not required here, but it is good style to include it.*

---

```
//This is the implementation for the class DiscountSale.
#include "discountsale.h"
```

*This is the file discountsale.cpp.*

```
namespace salesavitch
{
 DiscountSale::DiscountSale() : Sale(), discount(0)
 {}

 DiscountSale::DiscountSale(double the_price, double the_discount)
 : Sale(the_price), discount(the_discount)
 {}

 double DiscountSale::bill() const
 {
 double fraction = discount/100;
 return (1 - fraction)*price;
 }
} //salesavitch
```

# DiscountSale::bill

- ❑ Class `DiscountSale` has its own version of virtual function `bill`
  - Even though class `Sale` is already compiled, `Sale::savings( )` and `Sale::operator<` can still use function `bill` from the `DiscountSale` class
  - The keyword `virtual` tells C++ to wait until `bill` is used in a program to get the implementation of `bill` from the calling object

```
//This is the implementation file: sale.cpp
//This is the implementation for the class Sale.
//The interface for the class Sale is in
//the header file sale.h.
#include "sale.h"

namespace salesavitch
{

 Sale::Sale() : price(0)
 {}

 Sale::Sale(double the_price) : price(the_price)
 {}

 double Sale::bill() const
 {
 return price;
 }

 double Sale::savings(const Sale& other) const
 {
 return (bill() - other.bill());
 }

 bool operator < (const Sale& first, const Sale& second)
 {
 return (first.bill() < second.bill());
 }

} //salesavitch
```

# Display 15.9

- Because function **bill** is virtual in class **Sale**, function **savings** and **operator<**, defined only in the base class, can in turn use a version of **bill** found in a derived class
  - When a **DiscountSale** object calls its **savings** function, defined only in the base class, function **savings** calls function **bill**
  - Because **bill** is a virtual function in class **Sale**, C++ uses the version of **bill** defined in the object that called **savings**

## Implementation of the Base Class Sale

**Sale**  
virtual bill()  
savings()

```
//This is the implementation file: sa
//This is the implementation for the class Sale.
//The interface for the class Sale is in
//the header file sale.h.
#include "sale.h"
```

**DiscountSale**  
virtual bill()  
//no re-defined savings()

```
namespace salesavitch
{

 Sale::Sale() : price(0)
 {}

 Sale::Sale(double the_price) : price(the_price)
 {}

 double Sale::bill() const
 {
 return price;
 }

 double Sale::savings(const Sale& other) const
 {
 return (bill() - other.bill());
 }

 bool operator < (const Sale& first, const Sale& second)
 {
 return (first.bill() < second.bill());
 }

} //salesavitch
```

Q:

Since bill() is a virtual function, what will happen in the following code?

If bill() is not a virtual function, what will happen in the following code?

```
Sale simple(10.00);
DiscountSale d1(11.0, 10);
DiscountSale d2(11.0, 10);
if (d1 < simple)
{
 cout << "Saving is $" <<
 simple.savings(d1);
}
if (d1 < d2)
{
 cout << "Saving is $" <<
 d2.savings(d1);
}
```

```
//Demonstrates the performance of the virtual function bill.
#include <iostream>
#include "sale.h" //Not really needed, but safe due to ifndef.
#include "discountsale.h"
using namespace std;
using namespace salesavitch;

int main()
{
 Sale simple(10.00);//One item at $10.00.
 DiscountSale discount(11.00, 10);//One item at $11.00 with a 10% discount.

 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(2);

 if (discount < simple)
 {
 cout << "Discounted item is cheaper.\n";
 cout << "Savings is $" << simple.savings(discount) << endl;
 }
 else
 cout << "Discounted item is not cheaper.\n";

 return 0;
}
```

### Sample Dialogue

Discounted item is cheaper.  
Savings is \$0.10

# Override vs. Redefine

- ❑ Virtual functions whose definitions are changed in a derived class are said to be **overridden**
- ❑ Non-virtual functions whose definitions are changed in a derived class are **redefined**

A potential slicing problem  
if we do not use virtual functions.

# Preliminary: C++'s Type Checking

- C++ carefully checks for type mismatches in the use of values and variables
  - This is referred to as **strong type checking**
  - Generally, the type of a value assigned to a variable must match the type of the variable
    - E.g., `double a = "Hello";` //incorrect. No relationship
    - Recall that some automatic type casting occurs
      - E.g., `int a = 20.34;` //correct, but what happens?
- When a double is assigned to an int, **what happens?**

# Slicing Problem is special to C++

- ❑ C++ is one of the few OOP languages to support objects as automatic variables
- ❑ Automatic variables are local variables or value parameters.
- ❑ Memory for these objects is "automatically" allocated and destroyed on the stack by scope.
- ❑ Java, C#, python only allow primitive types and object references as local variables.
  - No slicing problem when everything is a reference
- ❑ OOP=object-oriented programming

# Type Checking and Inheritance

- Consider

```
class Pet
{
public:
 virtual void print();
 string name;
}
```

```
class Dog : public Pet
{
public:
 virtual void print();
 string breed;
}
```

## **Pet**

Pet::print()  
name

## **Dog**

Dog::print() //overridden  
name  
breed

# Slicing problem: A Sliced Dog is a Pet

- C++ allows the following assignments: `Dog vdog; Pet vpet;`  
`vdog.name = "Tiny";`  
`vdog.breed = "Great Dane";`  
`vpel = vdog;`
- However, `vpel` will lose the `breed` member of `vdog` since an object of class `Pet` has no `breed` member, copying member for member.

- This code would be illegal:

```
cout << vpet.breed;
```

vpel

**Pet**

Pet::print()

name

vdog

**Dog**

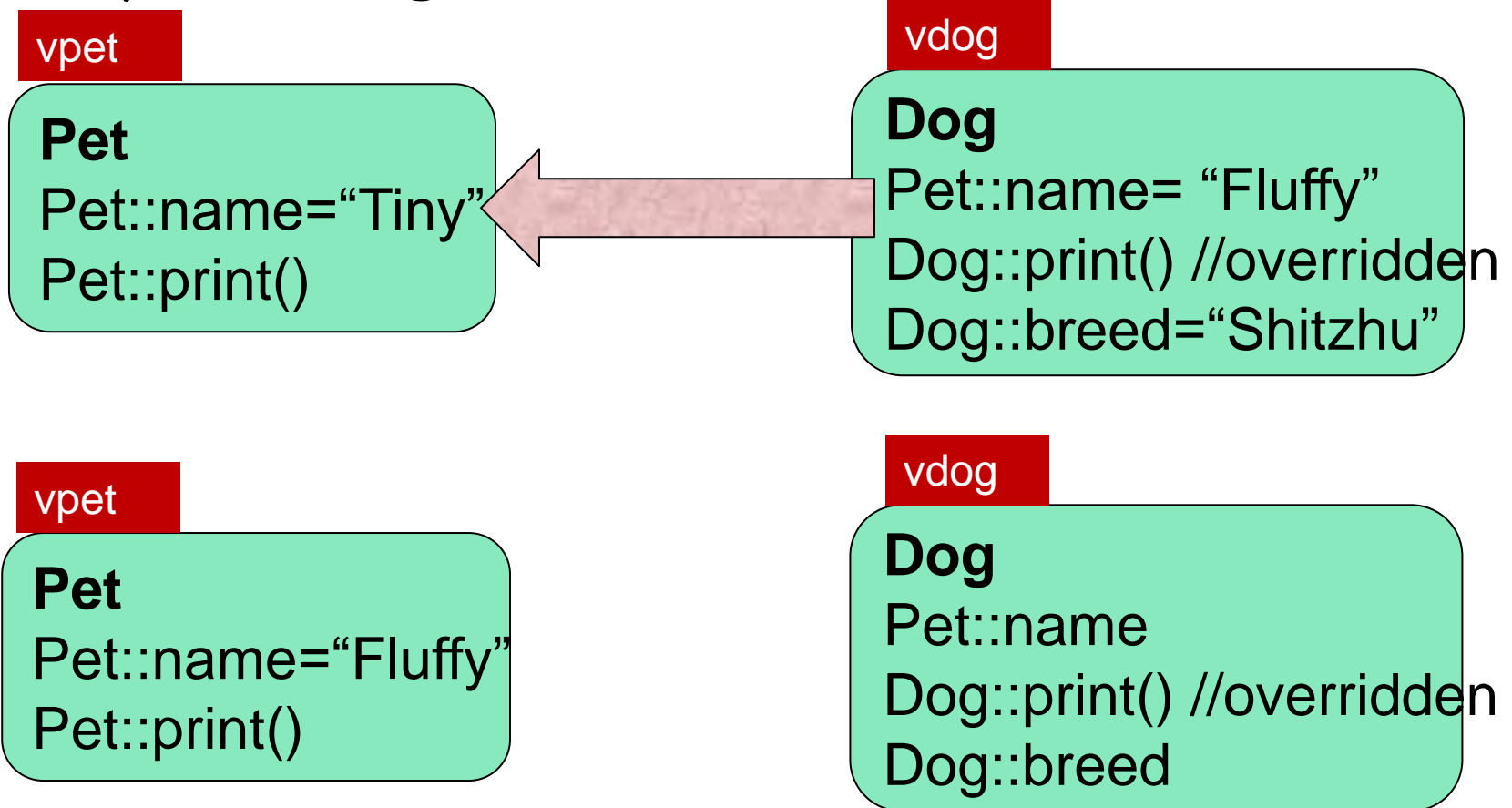
Dog::print() //overridden

name

breed

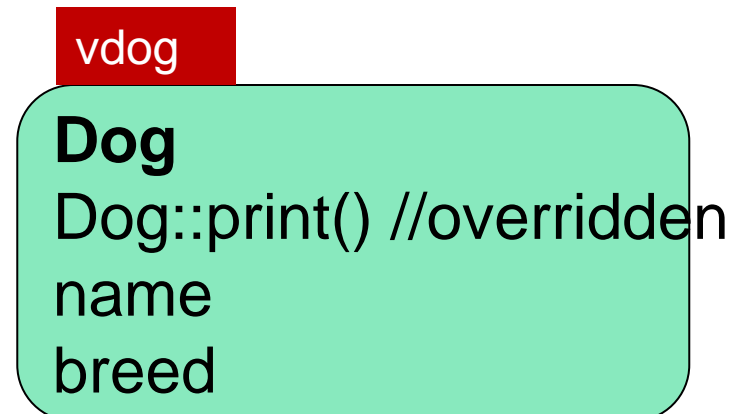
# Only the Pet part gets copied over

❑ `vppet = vdog;`



# The Slicing Problem

- It is **legal** to **assign** a derived class object into a base class **variable** (**not a reference**), however...
  - This slices off data in the derived class that is not also part of the base class
  - Some member functions and member variables are lost



# Extended Type Compatibility

- It is possible in C++ to avoid the slicing problem
  - Using pointers to dynamic variables and virtual functions, we can still access the added members of the derived class object.

# Dynamic Variables and Derived Classes

## □ Example:

```
Pet *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great
 Dane";
ppet = pdog; //or &vdog
```

```
void Dog::print()
{
 cout << "name: "
 << name << endl;
 cout << "breed: "
 << breed << endl;
}
```

`ppet->print( );` is legal and produces:

name: Tiny

breed: Great Dane

**Display 15.12 (1-2)**

## More Inheritance with Virtual Functions (part 1 of 2)

---

```
//Program to illustrate use of a virtual function
//to defeat the slicing problem.
```

```
#include <string>
#include <iostream>
using namespace std;
```

```
class Pet
{
public:
 virtual void print();
 string name;
};
```

```
class Dog : public Pet
{
public:
 virtual void print();//keyword virtual not needed, but put
 //here for clarity. (It is also good style!)
 string breed;
};
```

```
int main()
{
 Dog vdog;
 Pet vpet;

 vdog.name = "Tiny";
 vdog.breed = "Great Dane";
 vpet = vdog;
```

```
//vpert.breed; is illegal since class Pet has no member named breed
```

```
Dog *pdog;
pdog = new Dog;
```

# Display 15.12 (1/2)

## More Inheritance with Virtual Functions (part 2 of 2)

---

```
pdog->name = "Tiny";
pdog->breed = "Great Dane";

Pet *ppet;
ppet = pdog;
ppet->print(); // These two print the same output:
pdog->print(); // name: Tiny breed: Great Dane

//The following, which accesses member variables directly
//rather than via virtual functions, would produce an error:
//cout << "name: " << ppet->name << " breed: "
// << ppet->breed << endl;
//generates an error message: 'class Pet' has no member
//named 'breed' .
//See Pitfall section "Not Using Virtual Member Functions"
//for more discussion on this.

return 0;
}

void Dog::print()
{
 cout << "name: " << name << endl;
 cout << "breed: " << breed << endl;
}

void Pet::print()
{
 cout << "name: " << endl; //Note no breed mentioned
}
```

# Display 15.12 (2/2)

### Sample Dialogue

```
name: Tiny
breed: Great Dane
name: Tiny
breed: Great Dane
```

# Use Virtual Functions

- The previous example:

```
ppet->print();
```

worked because print was declared as a virtual function

- The following code would still produce an error:

```
cout << "name: " << ppet->name
 << "breed: " << ppet->breed;
```

//breed is a **public member**

//but we still cannot use a base class pointer

//to DIRECTLY access it! However,

//We can use **virtual functions to access them.**

# Why?

- `ppet->breed` is still illegal because `ppet` is a pointer to a `Pet` object that has no `breed` member
  - `breed` is just a data member, not a virtual function!
- Function `print( )` was declared virtual by class `Pet`
  - When the computer sees `ppet->print( )`, it checks the virtual table for classes `Pet` and `Dog` and finds that `ppet` points to an object of type `Dog`
  - Because `ppet` points to a `Dog` object, code for `Dog::print( )` is used

# Remember this...

- If the type of the **pointer** `p_ancestor` is a base class for the **pointer** `p_descendant`, the following assignment

```
p_ancestor = p_descendant;
```

- Allows us to use `p_ancestor` to call **virtual functions**. The invoking object is still `p_descendant`, therefore
  - It **will** invoke the derived class function which has access to all derived class data members.
    - (i.e., no data members will be inaccessible)
  - **BUT** we **CANNOT** directly access a derived class member from `p_ancestor`

# A side note on reference (1/3)

- A reference has to be initialized at the time when declared (except as a function parameter)

```
int x=10;
```

```
int& y = x;
```

Use references

```
// Reference y is another name for x, see the address of x
```

```
cout << &x <<endl;
```

```
// Here, the address of y will be exactly the same as x!
```

```
cout << &y <<endl;
```

- A reference **cannot** be redirected to refer to something else once it is defined. Cannot be changed!

## Call-by-Reference Parameters (part 1 of 2)

---

```
//Program to demonstrate call-by-reference parameters.
#include <iostream>

void get_numbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swap_values(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void show_results(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main()
{
 int first_num, second_num;

 get_numbers(first_num, second_num);
 swap_values(first_num, second_num);
 show_results(first_num, second_num);
 return 0;
}

//Uses iostream:
void get_numbers(int& input1, int& input2)
{
 using namespace std;
 cout << "Enter two integers: ";
 cin >> input1
 >> input2;
}

void swap_values(int& variable1, int& variable2)
{
 int temp;

 temp = variable1;
 variable1 = variable2;
 variable2 = temp;
}
```

A side note on  
reference  
(2/3)

Use reference as  
function parameters

Recall this example  
(from the textbook)

# A side note on reference (3/3):

## Variable name, reference, pointer

C++

- reference to a variable (or object) ---- another name for a variable, and it will never be changed to be a reference to a different variable, **immutable**
- pointer to a variable (or object) ---- can be modified to point to different variables, **mutable**
  - imagine it as an erasable address tag

Java

- reference to a variable---- can be modified to refer to different variables
  - imagine it as an erasable name tag, or a named hat that can be given to different persons to wear

# Benefits of virtual functions...

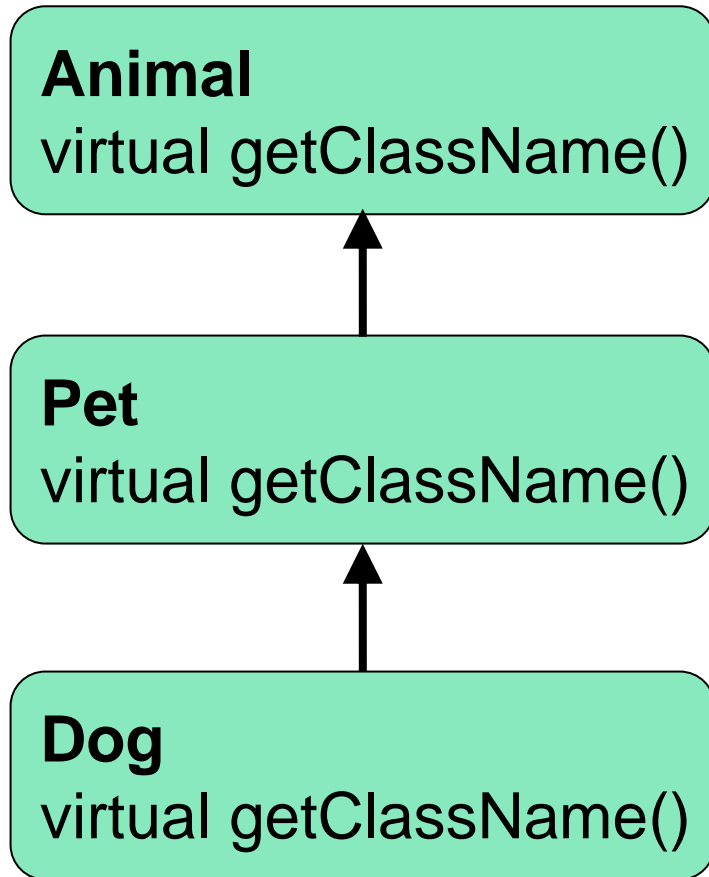
- ❑ Allow us to refer to each derived object by the base class but still behaves like the derived class.
- ❑ Allow us to add derived types as needed.
- ❑ Allow us to use a pass-by-reference for the base class, but pass in derived class arguments to get the desired result.
- ❑ Allow us to define collections of the base class (eg. `vector<base*> list`). We can call a function on every element knowing that the correct one will be called (eg. `employeePtrs[i]->print_check()`, `list[i]->draw()`)

## So far...

- ❑ What is the difference between non-virtual and virtual functions?
- ❑ What is the advantage of virtual functions?
- ❑ What is the slicing problem? How can we avoid it?
- ❑ How do we access derived data members using base class objects?
- ❑ What is the difference between references in C++ and other object-oriented languages like Java?

# More than two classes in a chain of inheritance hierarchy

inheritance hierarchy



```
Dog d ;
Animal &animal_ref = d;
cout << animal_ref.GetClassName();
```

C++ will check every inherited class between Animal and Dog (including Animal and Dog) and use the most-derived version of the function that it finds.

virtual\_functions\_demo\_2.cpp

# More details

- ❑ Pure virtual functions
  - Don't redefine, better to override functions
  - Abstraction - interfaces
- ❑ Virtual destructors
  - When working with base classes, virtual destructors ensure no memory leaks
- ❑ Covariant return types
  - Overriding functions that return an object pointer or reference

Abstract classes

**PURE VIRTUAL FUNCTIONS**

# Refers to the Figure::center example

- ❑ The base class assumes that there is an implementation of all needed functions.
- ❑ If `Figure::center()` function uses `erase()`, `move(int x, int y)` and `draw()` function, all but one must be done in the child.
- ❑ Move just sets `x, y` for the Figure. Can be done in Figure.
- ❑ `Figure::erase()` and `Figure::draw()` must be defined `virtual` because only the child knows how to implement them.

# Pure virtual function - abstract class

- Let's say we need to declare a function in the base class, but only the derived classes know how to implement the function.

Ex. `Shape::draw()`, `Employee::get_net_pay()`

- Make the function pure virtual.
- Classes with pure virtual functions are abstract.
  - No concrete variables can be created
  - Only references and pointers

# Pure virtual function

- If a class has a pure virtual function, then the class cannot be instantiated, and the derived classes of the class have to define these function before they can be instantiated.
  - This ensures the derived classes NOT forget to redefine those pure virtual functions (which is what the base class hopes)
  - Syntax of pure virtual function:

virtual return-type function(type parameter) = 0;

This tells the compiler that the class is abstract so it cannot be used to directly declare object instances.

# Revisit Employee class

- `get_net_pay()` function should be a pure virtual function in the declaration of Employee class.
  - Find the error at compile time instead of waitin until runtime.
  - When a base class contains pure virtual functions, it cannot be used to declare variables. It can only be used as a parameter, a reference or a pointer.

# Calling a base class's virtual function

virtual\_functions\_demo\_4.cpp

# Calling a base class's virtual function

- We've actually seen this in the overridden assignment operator=(const Derived& obj).
- Remember that the overloading operator= in a derived class starts with Base::operator=(obj)

```
Derived& operator=(const Derived& obj) {
 if (this != &obj) {
 Base::operator=(obj);
 // Derived assignment part.
 return *this;
 }
}
```

# Why call base implementation of virtual function?

- ❑ There are many cases where you might want to call the base class implementation of a virtual function inside of a derived class function:
  - Assignment of base member variables
  - Common functionality like `print_check()`, where presentation of base class components could be done first followed by derived class info.
  - Serialization - reading and writing the object into/from a file.
    - Data members of the base class should be handled before the derived class just like `operator=`

# Virtual Destructors

- ❑ Destructors should be made virtual

Example: [destructors.cpp](#)

- ❑ What about copy assignment operators?

- Virtual assignment operators, allow

```
Base* b = &vDerived;
```

```
*b = otherDerived;
```

- Consider `Base *pBase = new Derived;`

...

`delete pBase;`

- If the destructor in `Base` is **virtual**, the destructor for `Derived` is invoked when `pBase` points to a `Derived` object, returning `Derived` members to the freestore
  - The `Derived` destructor in turn **automatically** calls the `Base` destructor
- If the `Base` destructor is not virtual, only the `Base` destructor is invoked
- This leaves `Derived` members, not part of `Base`, in memory

# Covariant return type

virtual\_functions\_demo\_3.cpp

//illustrate covariant return type

# Covariant Return type Motivation

```
class Base {
public:
 virtual Base * clone() const {
 return new Base(*this);
 }
};
class Derived : public Base {
public:
 virtual Base * clone() const override {
 return new Derived(*this);
 }
};
Derived *d1 = new Derived();
Base * b = d1->clone();
Derived *d2 = dynamic_cast<Derived *>(b);
d2->Derived::OtherFunc();
```

- Base class defines a virtual member function `Base::clone` that returns a copy of the given object.
- Derived class overrides this function and ends up returning a pointer of type Base class even though what we really want is a `Derived*`.
- C++ allows the derived type to implement a overridden function with return type that is a sub-type of the return type of the base function.

# Covariant Return Types

```
class Base {
public:
 virtual Base * clone() const {
 return new Base(*this);
 }
};
class Derived : public Base {
public:
 virtual Derived * clone() const override
{
 return new Derived(*this);
 }
};
Derived *d1 = new Derived();
Derived *d2 = d1->clone();
d2->Derived::OtherFunc();
```

- Covariant Return Types help avoid awkward `dynamic_cast`.
- This is one reason that a C++ function signature does not include the return type.
- The `Derived::clone()` function overrides the `Base::clone()` function even though return types are different.

# Design suggestions

- Simple virtual function
  - Inheriting it implies → inherit both interface and a default implementation.
  - In your derive class, you need to support this function, but if you don't want to write your own, you can fall back on the default version in base class.
  - **Danger:** if a derived class might not want to use the default implementation from the base class, but forget to define its own, then it will use the inherited one (which is not what it wants!)

# Design suggestions

- Pure virtual function
  - Inheriting it implies → inherit interface only
  - In your derived class (that can be instantiated), you must define it, but the base class has no idea how you are going to implement it.
  - The danger mentioned for the simple virtual function does not exist.

# Design suggestions

- ❑ Regular non-virtual function
  - Don't redefine an inherited non-virtual function (even though allowed by C++). Make sure the "is-a" relationship always true for public inheritance.

```
class B
{ public:
 void fun1();
}

class D: public B
{public: void fun1(//different implementation);}

//inconsistent, confusing behavior.
//same object D, but different fun1() is called,
// when D is pointed to by different ptr types
// (also true if references used)

D d;
B *pB=&d;
pB ->fun1();// B::fun1() is called!!

D *pD=&d;
pD ->fun1();//D::fun1() is called!!
```

# Interface Class

- ❑ An **interface class** is a class that ...
  - has no members variables,
  - all of the functions are pure virtual!
- ❑ The class is only an interface definition, no actual implementation.
- ❑ Why use interface?
  - When you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

virtual\_functions\_demo\_6.cpp