Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 8

Date: October 21, 2015

What we’re learning in this Slide Set:

- Inheritance – Part II

Textbook Chapters:

Relevant to this slide set:

- Chapter 15

Coming soon:

- Chapter 16
- Chapter 17

We previously talked about some of the basics of inheritance and derived classes (Recall chapter 10)

Basically, inheritance allows you to create a very general form of a class and then later derive more specialized versions of the class that inherit all the properties of the previous class.

Recall, our discussions of the BankAccount classes and derived classes previously.

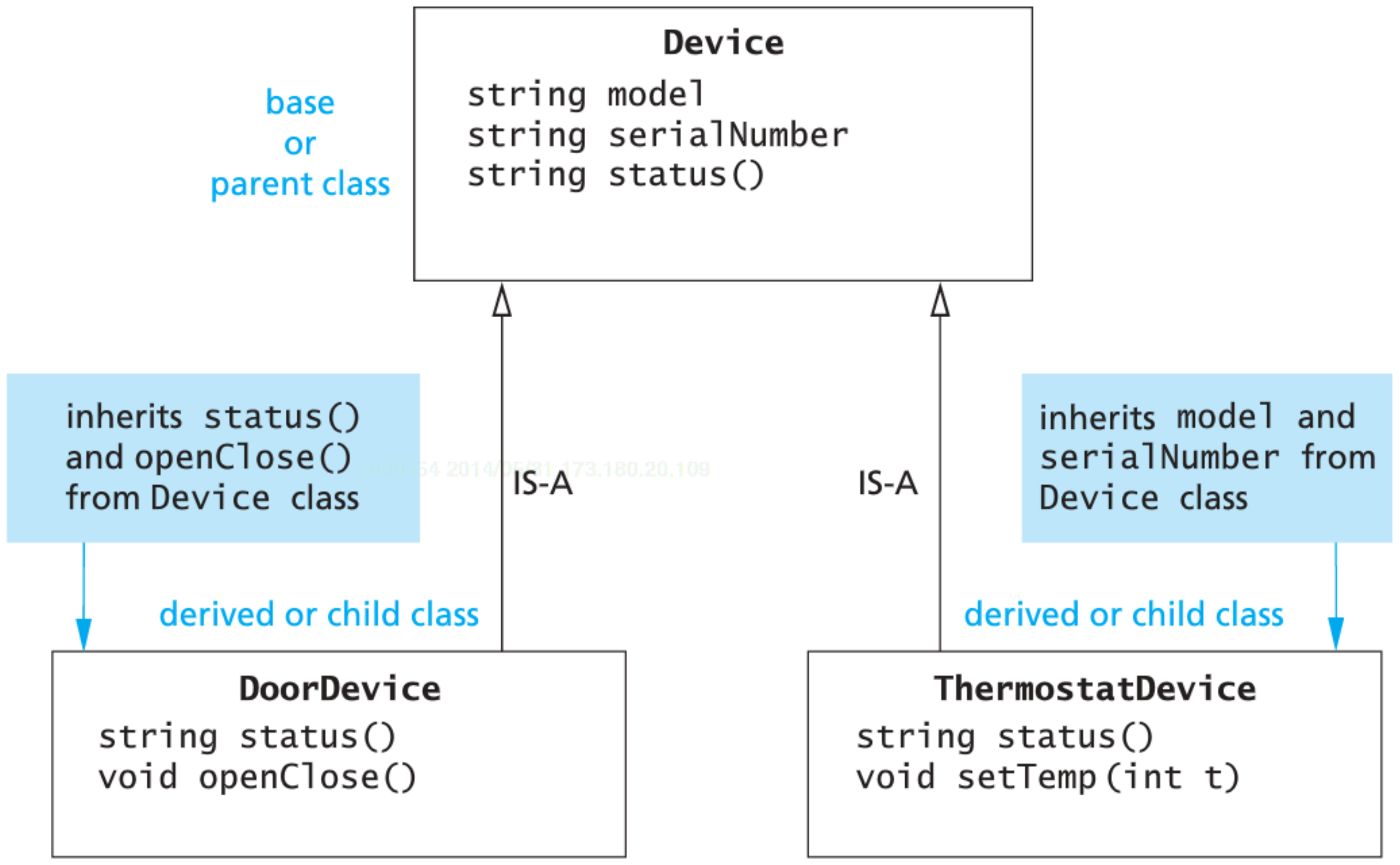
Remember, inheritance is the process whereby a new ***derived class*** is created from a ***base class***

Alternately, the base class may also be called the parent class and the derived class the child class.

Derived classes include all the member variables and functions of the base class

- They can also have additional member functions and/or variables that are particular to the derived class.

Example of Inheritance Hierarchy for Home Automation Devices



Note that a derived class can either redefine or override a base class function (e.g. status()).

Another example:

Think about the predefined classes `ifstream` and `istream`.

- `ifstream` is a derived class from the `istream` class by adding functions such as open and close
- `cin` belongs to the class of all input streams (i.e. `istream`), but not to the input-file streams (`ifstream`).
  - It lacks the member functions open and close for the derived class `ifstream`

Let’s look at an example of a record keeping program with records for employees (both salaried and hourly employees).

The common denominator is that they are all employees, however:

One subset is paid an hourly wage, and

The other subset is paid a fixed wage per month (or week)

Let’s look at an example implementation of the base employee class

Base class:

What is missing from the employee class Interface?

```
//This is the header file employee.h.
//This is the interface for the class Employee.
//This is primarily intended to be used as a base class to derive
//classes for different kinds of employees.
#ifndef EMPLOYEE_H
#define EMPLOYEE H
#include <string>
using namespace std;
namespace employeessavitch
{

    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;
    };

}//employeessavitch
#endif //EMPLOYEE_H
```

```
//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;
    }
```

```
        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);
        }

    }//employeessavitch
```

Note: The print check function prints an error because it does not know how to print a check (cheque) without knowing the employee type

Now our derived classes can inherit the member variables and member functions

The example places the class Employee and the two derived classes in the same namespace.

C++ does not require them to be in the same namespace, but since they are related, it makes sense to keep them together.

The derived class has a similar definition to the base class but adds a colon and the keyword public as well as the name of the base class to the first line of the definition

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

Derived class: Hourly Employee

```
//This is the header file hourlyemployee.h.
//This is the interface for the class HourlyEmployee.
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H

#include <string>
#include “employee.h”

using namespace std;
namespace employeessavitch
{

    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;
    };

}//employeessavitch

#endif //HOURLY EMPLOYEE_H
```

You only list the declaration of an inherited member function if you want to change the definition of the function.

Derived class: Salaried Employee

```
//This is the header file salariedemployee.h.
//This is the interface for the class SalariedEmployee.
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <string>
#include “employee.h”

using namespace std;

namespace employeessavitch
{

    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
    };

}//employeessavitch

#endif //SALARIEDEMPLOYEE_H
```

<u>Some important notes about derived classes (assume public inheritance):</u>

Derived classes automatically access all the member variables and functions of the base class

- However, derived classes can only *directly* access the ***public*** member variables and functions.
- Like all other classes, derived classes are not allowed to directly access private member variables or use private member functions of the base class
  - ***<u>But</u>*** every object of the derived class (HourlyEmployee, SalariedEmployee) has these private member variables. This is why you need accessor and mutator functions

(We’ll talk about protected members later.)

Some important notes about derived classes (assume public inheritance):

Derived classes can also add additional member variables (e.g. salary) and functions (get_salary) to their definition

- All member functions and member variables added directly to the derived class can be accessed directly by that class

You do not give the declarations of the inherited member functions as part of the derived class definition ***unless*** you want to change them.

This is why the print_check function from the base class was repeated in both of our derived classes.

We also have to pay special attention to the constructors for derived classes (but we'll look at that later).

The implementation file for the derived class defines all of the added member functions (and does not include definitions for inherited member functions unless the definition has changed in the derived class).

Some important notes about derived classes (assume public inheritance):

Remember, Base and Derived classes are commonly referred to as ***Parent*** and ***Child*** classes.

If a class is a parent of a parent of a parent of another class, it is often called an ***ancestor*** class.

Furthermore, if class A is an ancestor class of class B, then class B can also be referred to as a ***descendant*** of class A.

Derived classes inherit all the member variables and all ordinary member functions of the base class (some specialized functions such as constructors are not automatically inherited).

As you will see, by having the common base class, Employee, you are saved from having to write a significant amount of identical code for twice for the two derived classes.

As such, inheritance allows you to reuse the code in the Employee class

The definition of an inherited member function can be changed in the definition of a derived class so that it has a different meaning for a derived class than the base class:

This is called ***redefining*** the inherited member function

For example the member function print_check() is redefined for both the derived classes

To redefine a member function, include it in the class definition of the derived class and give it a new definition (as you would with any member function added to the derived class).

Let’s look at the implementation of the derived class Hourly Employee

```
//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;
    }
```

```
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 the 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";
}


}//employeessavitch
```

Constructors in Derived Classes

A constructor in a base class is not inherited in the derived class.

However, you ***can*** invoke a constructor from a base class within the definition of a derived class constructor.

- This will normally meet your needs

A derived class's constructor uses the base class' constructor in a special way.

Specifically, the base class constructor initializes all of the inherited data.

So a derived class constructor begins with an invocation of the constructor for the base class

There is a special syntax for invoking the base class constructor in the derived class shown in these two examples:

```
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
}

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

The portion after the colon is the initialization section of the constructor definition

There is a special syntax for invoking the base class constructor in the derived class shown in these two examples:

```
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
}

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

The portions `Employee(the_name, the_number)` and `Employee()` are invocations of two of the different constructors for the base class Employee

There is a special syntax for invoking the base class constructor in the derived class shown in these two examples:

```
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
}

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

As you can see, the syntax for invoking (calling) the base class constructor is analogous to the syntax used to set member variables: `wage_rate(the_wage_rate)` sets `wage_rate` to `the_wage_rate`

There is a special syntax for invoking the base class constructor in the derived class shown in these two examples:

```
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
}

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

Since all of the work to initialize the object’s member variables is done in the initialization section, the body of the constructors is empty.

There is a special syntax for invoking the base class constructor in the derived class shown in these two examples:

```
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
}

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

The key point: You should always include an invocation of one of the base class constructors in the initialization section of a derived class constructor so that all of the member variables can be initialized.

If a constructor definition for a derived class does not include an invocation of a constructor for the base class, then the default version of the base class constructor will automatically be invoked.

So this:

```
HourlyEmployee::HourlyEmployee( ) : wage_rate(0), hours(0)
{
    //deliberately empty
}
```

is equivalent to this:

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

However, style wise, it is better to be explicit with respect to calling the base class constructor for readability.

Final Notes on constructors for derived classes:

Remember: a derived class object has all the member variables of the base class (even if they are private and cannot be directly accessed).

All of these member variables need to be allocated memory and should be initialized.

This allocation of memory for the inherited member variables must be done by a constructor for the base class

It also provides an easy way to initialize these variables.

The call to the base class constructor (even if it is the default one) is the first action taken by a derived class (If there is no default constructor for the base class, that is an error condition).

If class C is derived from class B, which is derived from class A, then when an object of class C is created, first the constructor for class A is called, then a constructor for class B is called and finally the remaining class C constructor actions are taken.

Object’s of Derived Classes have more than one type:

Object’s of derived classes can be used anywhere an object of a base class can be used. For example:

- You can use the object of a derived class in lieu of an object of the base class as an argument for a function.
- You can assign an object of the derived class type to an object of base class type.

NOTE: You ***cannot*** assign an object of a base class type to an object of a derived class type.

More generally, an object of a class type can be used anywhere that an object of any of its ancestor classes can be used.

Hint: Think of our HourlyEmployee derived class and the Employee base class.

Use of Private Member Variables from the Base Class in a Derived Class

Remember: Even though derived classes include the member variables from the base class, you cannot directly access (by name) a ***private*** member variable in the definition of a member function outside of the base class even if that member function is part of a derived class.

This makes accessor and mutator functions very important. For example, we can initialize the variable joe's name as follows:

```
    HourlyEmployee joe(“Josephine”, “123-45-6789”, 0, 0);
```

But to change the name, we need a mutator function:

```
    joe.set_name(“Mighty-Joe”);
```

Use of Private Member Variables from the Base Class in a Derived Class

This means that if we want to redefine the print_check() function, the following is illegal:

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

Illegal use of net_pay

Instead we need to use the accessor and mutator functions of the base class as shown here:

```
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”;
```

Use of Private Member Variables from the Base Class in a Derived Class

It may seem odd that you cannot directly access private member variables of a base class in a derived class.

However, as part of encapsulation, we want to hide private details of classes and not make them accessible to users.

- This is particularly helpful if the base class gets redefined/updated.

Otherwise, all we would have to do to access the private member variables of a class would be to create a derived class and access it in a member function of the derived class.

- This would make private member variables accessible to anyone.

Besides, intentional subversion and adversarial situations, direct access to private member variables of a class could also result in a user changing the values by mistake or in an inappropriate way

- Mutator and accessor functions prevent this.

Private Member Functions from the Base Class in a Derived Class

Since derived classes cannot directly access any private member of an object, this means that private member functions are effectively not inherited as they are not available for use.

Since private functions should be "helper" functions, this should not be a problem.

- If you really need to use the function in a derived class, then it should be moved to the public section of the base class declaration.

We will now discuss an alternative to labelling class members as private and public that provide some other alternatives.

Using the protected qualifier

Since derived classes cannot directly access any private member of an object, there is an alternate classification for member variables that let them be accessed by name but only in a derived class (and not anywhere else).

Using the qualifier ***protected*** before a member variable or member function in a class has the same effect as labelling them as private members outside derived classes.

- However, inside a derived class, these variables and functions can be accessed by name.

Using the protected qualifier

For example, what would happen if all of the private member variables in the Employee class were labelled as protected instead of private?

How would this effect our definition of the redefined print_check() function?

```
void HourlyEmployee::print_check( )
//Only works if the member variables of Employee are marked
//protected instead of private.
{
    net_pay = hours * wage_rate;

    cout << “\n___________________________________________\n”;
    cout << “Pay to the order of “ << name << endl;
    cout << “The sum of “ << net_pay << “ Dollars\n”;
    cout << “__________________________________________\n”;
    cout << “Check Stub: NOT NEGOTIABLE\n”;
    cout << “Employee Number: “ << ssn << endl;
    cout << “Hourly Employee. \nHours worked: “ << hours
         << “ Rate: “ << wage_rate << “ Pay: “ << net_pay
         << endl;
    cout << “__________________________________________\n”;

}
```

Using the protected qualifier

Also note that if you derive class C from class B from class A and label the member variables of class A as protected instead of private:

- Not only will they be accessible by name in class B’s member function definitions,
- But they will also be accessible by name in class C’s member function definitions.

Using the protected qualifier

Many view using the protected qualifier as bad style and believe that it should not be used.

- It compromises the principle of encapsulation (hiding the class implementation and that all member variables should be marked private).
- Instead they believe that these inherited variables should only be accessed by accessor and mutator functions.

However, this encapsulation comes at a cost, so why would you choose to use the bad style of labelling variables as protected?

Redefining member functions

As previously shown in our example, the HourlyEmployee class and the SalariedEmployee class both redefine the function print_check (differently) to be used as appropriate in their derived classes.

When a function is redefined, its declaration must be listed in the definition of the derived class even though the declaration is the same as the base class.

Let’s look at the implementation of the SalariedEmployee class…

But first, we recall the class definition for the Salaried Employee Class

```
//This is the header file salariedemployee.h.
//This is the interface for the class SalariedEmployee.
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <string>
#include “employee.h”

using namespace std;

namespace employeessavitch
{

    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
    };

}//employeessavitch

#endif //SALARIEDEMPLOYEE_H
```

```
//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;
    }
```

Part 2 of SalariedEmployee Class Implementation:

```
    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";
    }
}//employeessavitch
```

An example program using the two Derived classes:

```
#include <iostream>
#include "hourlyemployee.h"
#include "salariedemployee.h"
using std::cout;
using std::endl;
using namespace employeessavitch;

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;
}
```

The functions **set_name, set_ssn, set_rate, set_hours,** *and* **get_name** *are inherited unchanged from the class* **Employee**. *The function* **print_check** *is redefined. The function* **get_hours** *was added to the derived class* **HourlyEmployee**.

A sample dialogue from the example program:

```
Check for Mighty Joe for 40 hours.

_________________________________________
Pay to the order of Mighty Joe
The sum of 820 Dollars

_________________________________________
Check Stub: NOT NEGOTIABLE
Employee Number: 123-45-6789
Hourly Employee.
Hours worked: 40 Rate: 20.5 Pay: 820

_________________________________________
Check for Mr. Big Shot

_________________________________________
Pay to the order of Mr. Big Shot
The sum of 10500.5 Dollars

_________________________________________
Check Stub NOT NEGOTIABLE
Employee Number: 987-65-4321
Salaried Employee. Regular Pay: 10500.5

_________________________________________
```

Redefining member functions

Note: *Redefining a function definition in a derived class is different from overloading a function name:*

When you redefine a function definition, the new function definition given in the derived class has the same number and types of parameters as in the base class.

If the new function in the derived class were to have a different number of parameters or a parameter of a different type from the function in the base class, then the derived class would have ***both*** functions (that would be overloading).

Redefining member functions

Note: *Redefining a function definition in a derived class is different from overloading a function name:*

For example, if we added the following function declaration to the definition of the class HourlyEmployee:

```
void set_name(string first_name, string last_name);
```

The class would now have this two-argument function set_name and it would inherit the Employee class one argument function:

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

This is overloading.

Redefining member functions

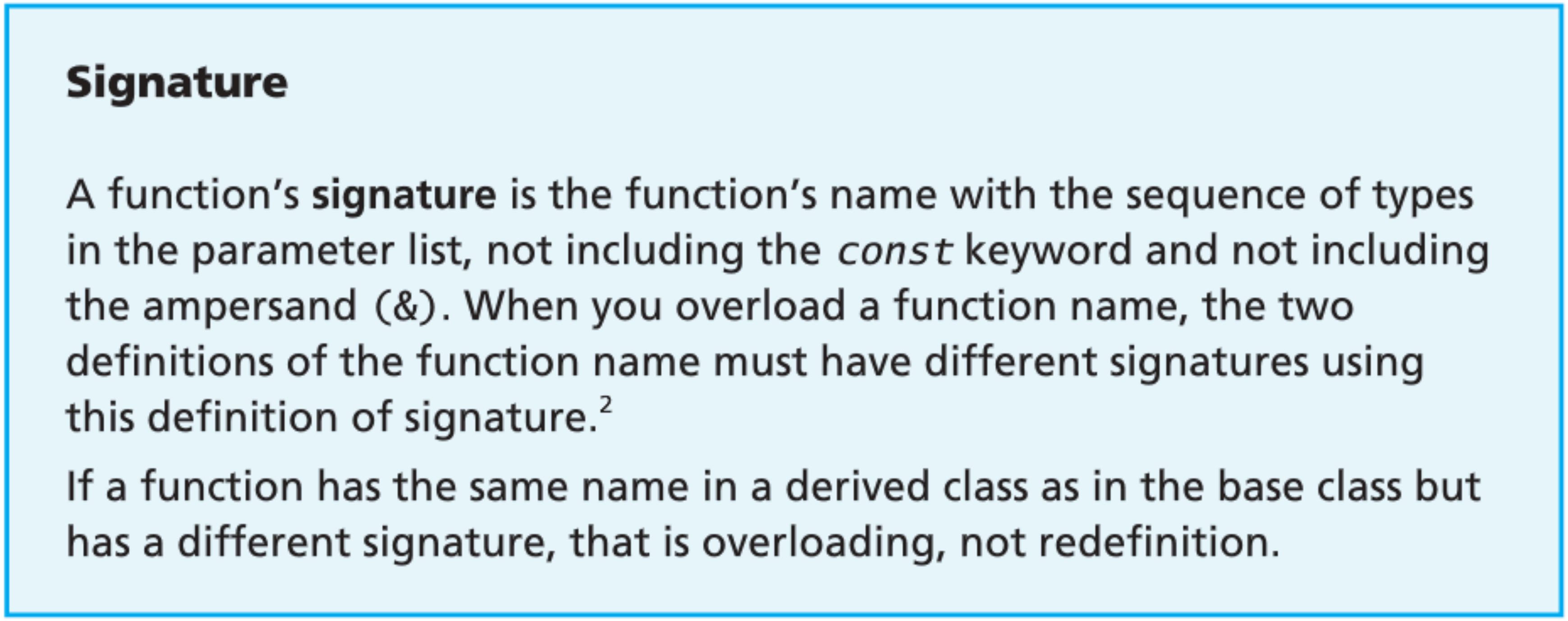
Note: *Redefining a function definition in a derived class is different from overloading a function name:*

Conversely, both the Employee class and the HourlyEmployee class have the same definition:

```
void print_check( );
```

As such, the class HourlyEmployee only has one function named print_check() and its definition of the function is different from the original definition in the base class Employee.

In this case, the function print_check() has been redefined.



Signature

A function's **signature** is the function's name with the sequence of types in the parameter list, not including the *`const`* keyword and not including the ampersand (&). When you overload a function name, the two definitions of the function name must have different signatures using this definition of signature.[2]

If a function has the same name in a derived class as in the base class but has a different signature, that is overloading, not redefinition.

Some compilers may allow overloading on the basis of `const` versus no `const`. For now assume that they don't, (and do not worry about it).

Redefining member functions

What if you redefine a function in a derived class, but at some point want to use the original function defined in the base class for a derived class object?

You are able to do this using the scope resolution operator you use to define class functions. For example, assume the following definitions based on our original examples:

```
Employee jane_e;
HourlyEmployee sally_h;
```

This uses the print_check() from the class Employee

```
jane_e.print_check( );
```

This uses the print_check from the class HourlyEmployee

```
sally_h.print_check( );
```

This uses the print_check given in the base class Employee with the derived class object sally_h

```
sally_h.Employee::print_check( );
```

Functions that are not inherited

If Derived is a derived class and Base is a base class, then all “normal” functions are inherited from the Base to Derived class.

There are some “special” functions that are not inherited (e.g. constructors). Destructors are also (effectively) not inherited (similar to private member functions) in that you cannot call them explicitly in the Derived class.

Copy constructors are not inherited.

- Although the compiler will generate a default copy constructor if you do not create one it will not work correctly for classes with pointers or dynamic data in their member variables.

Therefore, derived classes are no different from any other class:

- If your class member variables involve pointers, dynamic arrays or dynamic data, you should define a copy constructor.

Functions that are not inherited

The assignment operator (‘=‘) is not inherited.

- If the Base class defines an assignment operator and the Derived class does not, the the assignment operator used by the Derived class will be the default assignment operator that C++ generates (unless you define it in the Derived class yourself).

This is because the assignment operator for the Base class is generally insufficient for the Derived class.

- However, for the assignment operator to work properly for the Derived class, generally it needs to incorporate the definitions of the assignment operator from the Base class

Next we will talk about how to design overloaded assignment operators, etc. for derived classes.

Overloading assignment operators in a Derived class

The following code gives you an outline for overloading the assignment operator.

- Remember that an overloaded assignment operator must be defined as a member function of the class (not a friend function).

Assuming Derived is a derived class of Base class, then:

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

The remainder of the code would include the appropriate assignments for the new member variables introduced in the Derived class definition

Overloading copy constructors in a Derived class

The following code gives you an outline for overloading the copy constructor.

Assuming Derived is a derived class of Base class, then:

```
Derived::Derived(const Derived& object)
                 : Base(object), <probably more initializations>
{
           …
```

The invocation of the Base class copy constructor (i.e. `Base(object)`) initializes all the inherited member variables.

- Remember that since an object of type Derived is also of type Base, object is a legal argument of the Base class copy constructor.

Then the remainder of the code would include the appropriate assignments for the new member variables introduced in the Derived class definition

Destructors in a Derived class

Similar to the Copy Constructor and assignment operator, if we assume that the Base class has a correctly functioning destructor, then we can assume that it can be incorporated in to the Derived class destructor.

The Derived class will automatically invoke the Base class destructor, so it need not be explicitly called in the Derived class destructor.

Instead the Derived class destructor only needs to worry about “deleting” all member variables (and any data they point to) that were added in the Derived class.

The Base class destructor will delete the inherited member variables.

The order of these calls is the reverse of the constructor calls:

If class C is derived from class B is derived from class A, then when an object of class C goes out of scope, first the destructor for class C is called, then the destructor for class B, and finally class A’s destructor.

Polymorphism

Polymorphism refers to the ability to associate multiple meanings to one function name.

However, while this general definition applies to overloaded functions, when people generally use the word polymorphism, they are generally referring to ***late binding***- a specific mechanism for associating multiple meanings to one function name.

A virtual function (in some sense) is one that may be used before it is defined.

- This is kind of confusing, so let’s go through a theoretical example and then a practical example.

Late Binding

Let’s assume a set of graphics classes with a Base class Figure and a bunch of derived classes Rectangle, Circle, and Triangle.

Each derived class needs their own draw function because the shapes are different.

Let’s also assume that the parent class Figure has functions that apply to all figures, such as center (which would move a figure to the centre of the screen by erasing and redrawing it.

- To redraw the figure, `Figure::center` might use the draw function, but each derived class would have their own draw function.
- Since the draw functions for the derived classes did not exist when we wrote and compiled Figure’s center function, we need the concept of virtual functions defined in the Figure parent class to make this work.

The solution is to make draw in the base class, Figure, a virtual function.

Late Binding

As discussed on the last slide, we need virtual functions for parent classes as the compiler will not know anything about functions created in child classes.

When you make a function virtual, you are telling the compiler that you do not know the function implementation, but that when the function is used in a program, it should get the implementation for the object instance.

- This technique of waiting until run-time to determine the implementation of a procedure is called ***late binding*** or ***dynamic binding***.

Virtual functions provide C++ with the mechanism for late binding.

Let’s look at another example (with code).

<u>Late Binding</u>

Let’s assume you are designing a record keeping program with a base class Sale.

Initially, the only sales will be to retail customers that buy one particular item.

Later, you may want to add sales with discounts, or mail-order sales with shipping charge (all potential derived classes).

All sales will have a basic price for the item and produce some bill.

- Initially, the bill is just the basic price, but if you add discounts, then the bill will also depend on the size of the discount.

Late Binding

Your program will also need to compute the daily gross sales (which should be the sum of all the individual sales bills).

- You may also want to calculate the smallest, largest, and or average sales for the day

All of these can be calculated from the individual bills, but the functions for computing the bills will not be added until later (as it will depend on the type of sale).

- As such, we make the function for computing the bill a virtual function.

The next slides contain the interface and implementation for the class Sale.

- This corresponds to simple sales of a single item with no added discounts or charges.

Definition of 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
```

Implementation of the base class Sale

```
//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
```

Late Binding

As you can see from the code, the reserved word virtual has been included in the function declaration for bill in the class definition.

You will also see that the member function savings and the overloaded operator ‘<‘ both use the function bill.

Since bill is declared as a virtual function we can define new versions of the function bill in derived classes and the definitions of the member function savings and the overloaded operator ‘<‘ from the Sale class will use the version of the function bill that corresponds with the object’s derived class.

Let’s look at the derived class DiscountSale.

Definition of the derived class DiscountSale

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

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.

Implementation of the derived class DiscountSale

```
//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
```

Notice the different definition of the function bill from the base class Sale.

This new definition will be used when the member function savings or the overloaded operator '<' are used with an object of the DiscountSale class.

Since the class Sale, with the function savings was compiled before the derived class DiscountSale, late binding is needed to help the compiler link the correct version of bill for execution.

Use of a Virtual Function:

```
//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 at 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;
}
```

Important technical details for using virtual functions in C++

If a function has a different definition in a derived class than in the base class and you want it to be virtual, you add the keyword virtual to the function declaration in the base class.

If a function is virtual in the base class, then it is automatically virtual in the derived class.

- You do not need to add the reserved word virtual to the function declaration in the derived class. However, it is a good idea to label the function declaration in the derived class as virtual even though it is not required.

The reserved word virtual is only added to the function declaration (and not the function definition.

You do not have a virtual function (or the benefits of using a virtual function) unless you use the keyword virtual.

Virtual Functions

Why don’t we make all member functions virtual?

- Virtual functions do not run as efficiently as static functions. The compiler and run-time environment need to do much more work for virtual functions (running extra code) making your program less efficient.

Overriding and Polymorphism:

When a virtual function definition is changed in a derived class, programmers often say the function definition is overridden.

We distinguish between redefined and overridden functions.

Although both terms refer to changing the definition of the function in a derived class, they are treated quite differently by the compiler.

If a function is a virtual function, it's called overriding (and uses late binding).

If a function is not a virtual function, it's called redefining (and is done statically at compile time).

Finally, although polymorphism *conceptuall*y means the ability to associate multiple meanings to one function name, it *specifically* refers to the mechanism of late binding.

- Thus people often talk of polymorphism, late binding and virtual functions as the same topic.

Virtual Functions and Extended Type Compatibility:

There are some additional consequences to declaring a class member function to be virtual.

C++ is a reasonably strongly typed language.

- The types of objects/variables/function parameters/etc. are always checked and an error message/warning is issued if there is a type mismatch (assuming there is no conversion that can be automatically invoked).

This means that the value assigned to a variable generally needs to match the type of the variable.

- However in a few well-defined cases, C++ will perform an automatic type cast (called coercion) so that it appears that you can assign a value of one type to a variable of another.
- For example, C++ allows you to assign a value of type char or int to a variable of type double (but not the reverse).

Virtual Functions and Extended Type Compatibility:

Strong typing, however, interferes with the idea of inheritance in object-oriented programming.

Suppose class A and class B and objects aa and bb of these respective types.

You cannot always assign between objects of these types.

Virtual Functions and Extended Type Compatibility:

For example:

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

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

Dog vdog;
Pet vpet;
```

Anything that is a Dog is also a Pet.

Thus, it would seem to make sense to allow programs to consider values of type Dog to also be of type Pet.

Virtual Functions and Extended Type Compatibility:

For example:

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

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

Dog vdog;
Pet vpet;
```

Hence, the following should be allowed:

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

Virtual Functions and Extended Type Compatibility:

As discussed previously, C++ does allow this:

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

You are able to assign a value of a derived class to a variable of a parent type.

- However, you cannot perform the assignment in reverse.

Note that while this is allowed, *the value that is assigned to the variable vpet loses its breed field*.

- This is called the ***slicing problem*** and will result in the following attempted access producing an error message

```
cout << vpet.breed; //Illegal: class Pet has no member named breed
```

Arguably this makes sense, since once a Dog is moved to a variable of type Pet, it should be treated like any other Pet and have any properties particular to Dogs.

Virtual Functions and Extended Type Compatibility:

However, the dog, Tiny, is still a Great Dane and it would be good to still be able to refer to its breed, even if we treated it as a Pet somewhere along the line.

C++ offers a solution, allowing us to treat a Dog as a Pet, without throwing away the name of the breed.

To do this, we use pointers to dynamic object instances, such as:

```
Pet *ppet;
Dog *pdog;
```

If we use pointers and dynamic variables, we can treat Tiny as a Pet without losing his breed as the following is allowed:

```
pdog = new Dog;
pdog->name = “Tiny”;
pdog->breed = “Great Dane”;
ppet = pdog;
```

Virtual Functions and Extended Type Compatibility:

Given:

```
Pet *ppet;
Dog *pdog;
```

and

```
pdog = new Dog;
pdog->name = “Tiny”;
pdog->breed = “Great Dane”;
ppet = pdog;
```

we can still access the breed field of the node pointed to by ppet.

Suppose the following definition:

```
//uses iostream
void Dog::print()
{
    cout << “name: “ << name << endl;
    cout << “breed: “ << breed << endl;
}
```

The statement: `ppet->print();`

prints the following to the screen:

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

by virtue of the fact that print() is a virtual member function.

Recall the Pet and Dog class definitions:

```
//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;
};
```

Recall the Pet and Dog class definitions:

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

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

An application using these classes, could look like:

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

    vdog.name = “Tiny”;
    vdog.breed = “Great Dane”;
    vpet = vdog;

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

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

An example of the output from this program:

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

In summary:

The slicing problem refers to the situation where a derived class object is assigned to a base class variable.

Although the assignment is legal, any data members in the derived class object that are not also in the base class object will be lost in the assignment.

Similarly, any member functions that are not defined in the base class will also be unavailable to the resulting base class object.

From our example:

```
Dog vdog;
Pet vpet;
```

This means that vpet cannot be a calling object for any member function introduced in the class Dog. Let’s see what this means…

Details on why we need virtual member functions :

What if we hadn’t used the member function: `ppet->print();`

and instead had done the following

```
cout << “name: “ << ppet->name
     << “ breed: “ << ppet->breed << endl;
```

This code would have generated an error message as *ppet has its type determined by the pointer type of ppet (i.e. a pointer to type Pet) and type Pet has no field named breed.

However, since `print()` was declared `virtual` by the `Pet` base class, when the compiler sees: `ppet->print();`

it checks the *“virtual”* table for classes `Pet` and `Dog` and sees that `ppet` points to an object of type `Dog`. So it uses the code generated for:

`Dog::print()` instead of `Pet::print()`

Rules for using dynamic variables in object-oriented programming:

1. If the domain type of the pointer `p_ancestor` is a base class for the domain type of the pointer `p_descendant`, the the following assignment of pointers is allowed:

```
    p_ancestor = p_descendant;
```

In the case of the above assignment, none of the data members or member functions of the dynamic variable pointed to by `p_descendant` will be lost.

2. Although all the extra fields of the dynamic variable are there, you need to use virtual member functions to access them.

Attempting to compile class definitions without definitions for every virtual member function:

In this course, I have recommended agile software design, allowing you to develop code incrementally.

However, if you try to compile classes with virtual member functions, but do not implement each member, you may see some difficult to understand error messages (even if you do not call the undefined member functions).

Any virtual member functions that are not implemented before compilation will generate compilation error messages like:

"undefined reference to `Class_Name` virtual table."

Even if there is no derived class and there is only one virtual member, this type of message still occurs if that function does not have a definition.

Attempting to compile class definitions without definitions for every virtual member function:

The reason why this error message may be hard to decipher is that without definitions for the functions declared virtual, there may be *additional error messages* complaining about an undefined reference to default constructors, even if these constructors are already defined.

As such, be sure to include stubs in your class definition that define these functions.

Why make destructors virtual?

It is often a good idea to always make destructors virtual.

Consider the following code where SomeClass is a class with destructor that is not virtual:

```
SomeClass *p = new SomeClass;
    . . .
delete p;
```

When delete is invoked, the destructor of the class SomeClass is automatically invoked.

What happens when a destructor is marked virtual?

Why make destructors virtual?

What happens if we have a `Derived` class derived from `Base` class and have the following code:

```
Base *pBase = new Derived;
    . . .
delete pBase;
```

Which destructor should be called (Base or Derived)?

When the destructor in the class Base is marked virtual and the object pointed to is of type Derived, the destructor for the class Derived is called (which in turns calls the destructor for class Base).

If the destructor in the class Base is not declared as virtual, then only the destructor in the class Base is called.

- This can lead to a memory leak if the Derived class has additional data members (Remember what could happen if these additional data members are dynamic variables).

Another point to remember is that when a destructor is marked as virtual, ***all** destructors of all derived classes are automatically virtual (whether or not they are marked as virtual)*.

Why make destructors virtual?

So why make all destructors virtual?

- It ensures that any dynamic variables are properly returned to the freestore when an object is destroyed, meaning there will be no memory leaks.
- Given that you may not know all of the derived classes from a base class at design time, this keeps your class libraries safe for future usage when they include dynamic variables.
- Alternately, you should avoid the following types of assignments:

```
Base *pBase = new Derived;
    . . .
delete pBase;
```

From an engineering perspective, depending on the application, we may wish to avoid late binding (virtual functions) to reduce runtime overhead.

As such, although this is a safe software design practice it may be impractical for performance critical software – meaning that you should not use virtual functions at all.

Review Questions for Slide Set 8

- Do derived classes include all the member variables and functions of the base class?
- Can derived classes have additional member functions and/or variables that are specific to the derived class?
- Can a derived class redefine or override a base class function?
- Are derived classes allowed to directly access all member variables and functions?
- Do objects of derived classes have the private member variables of the base class?
- What happens if you declare a function in a derived class that has already been declared in the base class?

Review Questions for Slide Set 8

- What is the name for a parent of a parent of a parent class?
- If A is a child of B is a child of C, then A is a _______ of C.
- Which functions are not ordinarily inherited by a derived class?
- What does redefining a function mean? How is it different from overloading a function?
- Can you invoke a constructor from a base class in a derived class?
- If you don't explicitly invoke a constructor from the base class in the derived class, what happens?
- Where does the allocation for the inherited member variables occur?

Review Questions for Slide Set 8

- When is the base class constructor called in the derived class?
- If A is a derived class from class B, which is a derived class from class C, when an object of class A is created, what is the order of the constructor calls?
- What does it mean when I say that “objects of derived classes have more than one type”?
- What does the protected qualifier do?
- Are protected members only accessible in the direct child of a parent class or all descendent classes?
- What are the pros and cons of using the protected qualifier to label member variables?

Review Questions for Slide Set 8

- If you redefine a function, can you use the original function defined in the base class?
- If you overload the assignment operator in the base class and not in the derived class, what assignment operator is used with the derived class object(s)?
- Why does the Base class destructor not need to be explicitly called in the Derived class destructor? What should be in your Derived class destructor?
- What is the order of destructor calls, if class A is derived from class B, which is derived from class C?
- What does Polymorphism mean? What does it generally refer to?

Review Questions for Slide Set 8

- What is late binding? What is another name for late binding?
- What is a virtual function?
- What is the difference between redefining functions and overridden functions? What are the similarities?
- Why don’t we make all member functions virtual?
- What is coercion?
- What happens when you assign a derived class object (or pointer) to a base class object (or pointer)?
- What is the “slicing problem”?
- What are the rules for using dynamic variables with inheritance in object-oriented programming?

Review Questions for Slide Set 8

- Why do you include stubs defining virtual functions in the base class?
- Why might you want to make destructors virtual? Be able to give a concrete example of what might happen if they aren't virtual and explain how being virtual fixes them.
- If the destructor of a Base class is marked as virtual, how does this effect the derived classes?