Inheritance

- Derived Classes
- Constructor Base Initializer List
- Constructors in Derived Classes
- The protected Qualifier
- Redefinition of Member Functions
- Redefining Versus Overloading
- Polymorphism
  - Late Binding
  - Virtual Functions in C++
  - Virtual Functions and Extended Type Compatibility

Object oriented programming (OOP) is a popular and powerful programming technique. OOP provides the abstraction technique called inheritance. There are several definitions of an Object Oriented Language. Most definitions disagree in some detail. All agree that inheritance is necessary. Inheritance provides:
- A very general form of a class/type can be defined and compiled.
- Later, another version of the class that can use (inherit) the already defined features of the first class, that can add new features not present in the first class, and can redefine (in at least two senses) features from the first class.
15.1 Inheritance Basics

- The process of inheritance allows the programmer to create a new class -- called the derived (sub-) class -- from another -- called the base (super-) class.
- The derived class automatically has all the member variables and functions present in the base class.
- The derived class can define additional member functions, variables, or both.
- Class D is derived from class B means that class D has all the features of class B and some extra, added features as well.
- Sometimes D is called the child class and B the parent class.
- Recall that \texttt{ifstream} is derived from \texttt{istream} by adding extra features such as \texttt{open} and \texttt{close}.

Inheritance: Point, Circle, Cylinder

- We now consider the capstone exercise for Inheritance topic. We consider a point, circle, cylinder hierarchy. First, we develop and use class Point. Then we present an example in which we derive class Circle from class Point. Finally, we present an example in which we derive class Cylinder from class Circle.
//Inheritance: Point, Circle, Cylinder

//Set radius
void Circle::setRadius(double r)
{ radius=(r>=0 ? r : 0 ); }

//Get radius
double Circle::getRadius()
{ return radius; }

//Calculate area of Circle
double Circle::area()
{ return 3.14159 *radius *radius; }

//Output a circle in the form:
//Center=[x,y ];Radius =#.##
ostream &Circle::operator<<(ostream &output,const Circle &c)
{ output <<"Center=" << static_cast<Point >(c) <<" "; Radius=" <<setiosflags(ios::fixed | ios::showpoint) <<setprecision(2) << c.radius;
 return output; //enables cascaded calls }

//Driver for class Cylinder
int main()
{ 
 Cylinder cyl(5.7, 2.5, 12.23);
 //use get functions to display the Cylinder
 cont ""X coordinate is " << cyl.getX() << " y coordinate is " << cyl.getY() << " Radius is " << cyl.getRadius() << " area is " << cyl.area() << " height is " << cyl.getHeight() << endl;

 return 0; }
}
Inheritance: Point, Circle, Cylinder

//Output of main() driver

X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7
The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.00
The area of cyl is:
306.53
Cylinder printed as a Point is: [2, 2]
Cylinder printed as a Circle is:
Center = [2, 2]; Radius = 4.25
Area: 56.74

Inheritance

So far, we have discussed single inheritance in which each class is derived from exactly one base class.

A class may be derived from more than one base class; such derivation is called multiple inheritance.

Multiple inheritance means that a derived class inherits the members of several base classes. This powerful capability encourages interesting forms of software reuse, but can cause a variety of ambiguity problems.

Go to polymorphism, sec. 15.2, slide 52.

Sub- Classes (1 of 6)

- There is a hierarchy for classifying employees.
- We may think of employees in terms of:
  - A general class of employee, of which there is
  - A subset of employees that are paid an hourly wage
  - A subset of employees that are paid a fixed wage
    (Employee subsets such as administrative, permanent, temporary, part-time, or a catch-all “other” may be added.)
  - A notion of general employee may not be essential to the program but it can be useful in thinking about the program.
  - All kinds of employees have names, employee numbers, and perhaps member functions that are the same.

Sub- Classes (2 of 6)

- These can be put into the base class so each of these is inherited by other classes derived from the general employee class.
- We define an undifferentiated class Employee (Display 15.1) to enable definition of derived classes for different kinds of employees.
- The reason for an (undifferentiated) base class Employee is to encapsulate the common behavior and data for all employees so we can derive classes for different employees from this.
- Each derived class inherits all member functions of the base class Employee: print_check, get_name, change_name, give_raise, and so on.
- Each derived class redefines the functions print_check and give_raise in a way that is meaningful for an employee of that class.

Sub- Classes (3 of 6)

- It makes little sense to call the base class Employee print_check function, so it is implemented to display an error message and stop the program.
- A class that is derived from class Employee will automatically have the data members of the class Employee: (name, ssn, net_pay).
- Notice there is the keyword protected where you may be accustomed to seeing private.
- A protected member is the same as private to any function that is except a member function of a class derived from the base class (or a class that is derived from a class derived from the base class, that is by any chain of derivations).
Sub- Classes (4 of 6)

- A sub-class inherits all data members and all function members of the super-class. The public members of the base class are accessible to any function. The protected members are directly accessible by any function that is a member of any class derived directly or indirectly by an inheritance chain from the super-class, regardless of length.
- Interface files for classes derived from class Employee are given in Display 15.3 (HourlyEmployee) and in Display 15.4 (SalariedEmployee).
- Because these classes are related, we have placed them in one namespace.
Sub-Classes (5 of 6)

- Interface files for classes derived from class Employee are given in Display 15.3 (HourlyEmployee) and in Display 15.4 (SalariedEmployee).
- These classes are related so they are placed in one namespace.
- Syntax for inheritance:
  
  ```
  class HourlyEmployee : public Employee 
  { ... } ;
  ```
- The definition of a sub-class begins in the usual way: with the keyword `class`, followed by the name of the class.
- Then there is a colon separator, followed by an access specifier `public`, then the name of the super-class, `Employee`.
- This is called public inheritance. There is private and protected inheritance as well, but we will not deal with that in this course.

Sub-Classes (6 of 6)

- The definition of class HourlyEmployee does not mention member variables `name` and `net_pay`, nor is there any need to do so. These variables along with the super-class’s member functions are automatically supplied by inheritance.
- When a sub-class is defined, all prototypes of `new` member functions must be given. For example, in class `HourlyEmployee` the prototype for the replacement for `print_check` must be given.
- You do not give the prototypes of `inherited` member functions that are not being replaced when a derived class is defined.
Sub-Classes (7 of 7)

Inherited Members
A derived class automatically has all the functions and member variables of the base class. These members from the base class are said to be inherited. These inherited member functions and inherited member variables are not mentioned in the definition of the derived class, but they are automatically member of the derived class. (As we will see, you do mention an inherited member function in the definition of the derived class if you want to change the definition of the inherited member function.)

An Object of a Sub-Class is also an Object of the Super-Class
In everyday experience an hourly employee is an employee. In C++ the same sort of thing holds. Since HourlyEmployee is a derived class of class Employee, every object of the class HourlyEmployee can be used any place a class Employee can be used. In particular, you can use an argument of type HourlyEmployee when a function requires an argument of type Employee. You can assign an object of class HourlyEmployee to a variable of type Employee. (But be warned: You cannot assign a plain old Employee object to a variable of type HourlyEmployee. After all, an Employee object is not necessarily an HourlyEmployee.) Of course, the same remarks apply to any base class and its derived class. You can use an object of a derived class any place that an object of its base class is allowed. This relationship between a derived class and its base class is often referred to as the “Is-A” relationship. An HourlyEmployee is a Employee.

Constructor Base Initialization List (1 of 3)

- When we use inheritance, the inherited members must be initialized as well as new variables defined in the derived class.
- When defining a constructor you can initialize member variables in a base initialization list.
- The base initialization list is part of the heading of the constructor definition.

```cpp
class Rational {
public:
    Rational( );
    Rational(int t, int b);
    Rational(int w);
    // other members
private:
    int top;
    int bottom;
};
```

Constructor Base Initialization List (2 of 3)

- The base initialization list as part of the heading of the constructor definition.
- Rational::Rational( ): top(0), bottom(0) { /* empty body */ }
- Rational::Rational(int t, int b): top(t), bottom(b) { /* empty body */ }
- Rational::Rational(int w): top(w), bottom(1) { /* empty body */ }
- These examples illustrate the rule: The initialization goes in the constructor implementation header following a colon that in turn follows the parenthesis that closes the parameter list, and before the opening brace of the function block.
- The initialization list is a comma separated list of initializers.
- The initializers consist of the name of the member variable with its initializing value enclosed in parentheses.

Constructor Base Initialization List (3 of 3)

- These examples recall an alternate initialization we discussed briefly in Chapter 2 for simple variables:
  ```cpp
  int x(2); // is a variant of int x = 2;
  ```
- There need not be an initialization section, but when we define a derived class constructor, most of the time we must invoke a base class constructor to initialize the inherited variables.
- Use of initializer lists in constructor is a good practice, and will help you in some obscure cases where you must use initializer lists. (See the footnote on page 849 of the text.)
- There are times when the logic necessary to confirm that a constructor’s argument is legal will be difficult or impossible to squeeze into the parentheses of an initializer. Place such logic in the body of the constructor.

Constructors in Derived Classes (1 of 2)

- A constructor for a derived class uses the constructor for the base class to initialize the inherited data members.
- A derived class constructor calls the base class constructor first. The call is placed in the implementation.
- An example of a constructor for a derived class is illustrated in Display 15.5.
- The syntax for invoking a base class constructor is:
  ```cpp
  HourlyEmployee::HourlyEmployee() : Employee() {
  // code for the default HourlyEmployee initialization.
  }
  ```
- The portion, : Employee(), is the initialization section for the default constructor call to initialize the inherited data members.
- Always include a call to the super-class constructor in the initialization section of sub-class constructor.
We considered a Point<Circle<Cylinder hierarchy.

```cpp
ostream &operator<<(ostream &output, const Circle &c )
{ output <<"Center ="<< static_cast<Point >(c ) <<
"; Radius ="<<setiosflags(ios::fixed |ios::showpoint )
<<setprecision(2 ) << c.radius;
return output; //enables cascaded calls
}
class Cylinder : public Circle {
public: //default constructor
Cylinder(double h =0.0,double r =0.0, int x =0,int y =0 );
void setHeight(double ); //set height
double getHeight()const; //return height
double area(); //calculate and return area
double volume(); //calculate and return volume
protected:
double height; //height of the Cylinder
};
```

Employee and HourlyEmployee/SalariedEmployee class organization

You can assign an object of class HourlyEmployee to a variable of type Employee. But you cannot assign a plain Employee object to a variable of type HourlyEmployee. HourlyEmployee is an Employee but is Employee not an HourlyEmployee.)

- base initialization list.
- Rational::Rational(int t, int b): top(t), bottom(b) { /* empty body */ }
- Heterogeneous Linked Lists: See online class notes for examples.

```cpp
namespace savitchemployees
{
HourlyEmployee::HourlyEmployee( ) : Employee( )
{ cout <<"Enter HourlyEmployee wage rate, followed by return.:
"; cin >> wage_rate;
cout <<"Enter number of hours worked, followed by return:
"; cin >> hours;
}
HourlyEmployee::HourlyEmployee(string new_name, string new_number,
double new_wage_rate, double new_hours)
: Employee(new_name, new_number),
wage_rate(new_wage_rate), hours(new_hours)
{
// deliberately empty
}
void HourlyEmployee::set_rate(double new_wage_rate)
{ wage_rate = new_wage_rate; }
void HourlyEmployee::get_rate()
{ return wage_rate; }
void HourlyEmployee::set_hours(double hours_worked)
{ hours = hours_worked; }
double HourlyEmployee::get_hours()
{ return hours; }
```

Constructors in Derived Classes (2 of 2)

- This is the parameterized constructor for HourlyEmployee constructor.
  ```cpp
  HourlyEmployee::
  HourlyEmployee(string new_name, string new_number,
  double new_wage_rate, double new_hours)
  : Employee(new_name, new_number),
  wage_rate(new_wage_rate), hours(new_hours)
  {
  // deliberately empty.
  }
  ```

- Notice that all the work of the constructor is done in the initializer list, so the body is empty.
- It is good practice to comment a deliberately empty block.
We used the keyword protected where you might expect to see private.
A protected member is the same as private to any function that is except a member function of a class derived from the base class (or a class that is derived from a class derived from the base class, that is by any chain of derivations).
The meaning of the protected qualifier is illustrated in the definition of the function print_check of the derived class HourlyEmployee in Display 15.5.
The inherited member variables net_pay, name and ssn are marked protected in the base class, these variables can be accessed by name in the definition of member functions of all derived classes.
If the inherited members had been marked private in the base class definition, direct use of these variables in the derived class would be illegal.

Redefinition of Member Functions (1 of 2)

The protected Qualifier

In HourlyEmployee we gave prototypes for the new member functions set_rate, get_rate, set_hours, and get_hours.
We gave prototypes for some but not all member functions inherited from class Employee.
The inherited member functions whose prototypes were not given have the same definition in the derived class HourlyEmployee as they do in the base class Employee.
The only prototypes from the base class that are included in the derived class are the functions whose definitions are to be changed in the derived class definition.
The class HourlyEmployee gave new definitions for print_check and give_raise, definitions that are different from the base class definitions.
We say that the functions, print_check and give_raise are redefined (like overloading, but not quite) in the sub-class.

Redefinition of Member Functions (2 of 2)

We can derive another class for company officers from SalariedEmployee, which itself is a class derived from Employee.
A class that does nothing more than add a title is possible.
The only changes needed are
• a change in constructor to add set the new information
• redefinition of change_name to insert a title.
All other member functions are inherited unchanged from the base class SalariedEmployee, and its base class, Employee.

Redefining an Inherited Function

A derived class inherits all of the member function (and data members as well) that belong to the base class. However, if a derived class requires a different implementation for an inherited member function, the function may be redefined in the derived class. When a member function is redefined, you must list its prototype in the definition of the derived class, even though the prototype is the same as in the base class. If you do not wish to redefine a member function that is inherited from the base class, then it is listed in the definition of the derived class.
15.2 Polymorphism

- The term polymorphism is made up of "poly", meaning many; "morph", meaning form; and "ism", an action suffix. The most general meaning of the term is "having many forms".
- In the most general sense in programming, the word refers to the association of multiple meanings with one function name.
- Polymorphism is also used in a more restrictive sense.
- When used in this more restrictive sense, overloading refers to the ability to associate multiple meanings with one function name by means of the mechanism of late binding.
- Remember, in C++, the name of a function is more than just the function identifier. In C++, the name of a function is the function's identifier together with the sequence of types in the parameter list.
- Polymorphism is more than conventional function identifier overloading.

Late Binding (1 of 4)

- A virtual function is a function that, in some sense, may be used prior to definition.
- Consider a graphics program that has several kinds of figures: rectangles, circles, ovals, etc.
- There is a base class Figure.
- Each figure might be an object of a different class, derived from class Figure:
  - A rectangle class has width, breadth and center point.
  - An oval class has a large width, a short width and center point.
  - A circle class has a radius and center point.
- Each of these classes needs a draw member function, and each implemented differently.
- What does this have to do with "virtual functions" and use before definition?

Late Binding (2 of 4)

- The class Figure may have functions that apply to all figures.
- Suppose Figure has a member function Figure::center that moves a figure to the center of the screen by two steps:
  - erasing the existing figure,
  - redrawing the figure at the center of the screen
- The Figure::center member function uses the draw function to redraw the figure at the center.
- Suppose the class Figure is already written. In some later time we add a class for a new kind of figure, perhaps Triangle.
- Triangle is a derived class of base class Figure.
- The member function center is inherited by each derived class from base class Figure, and should perform correctly for all Triangle class objects.
- We have some trouble here. Why?

Late Binding (3 of 4)

- There is trouble here. Why?
- The inherited function center (unless something special is done) will use the definition of draw in class Figure, and that version of draw won't work for Triangle figures.
- We want the inherited function draw to use Triangle::draw, not Figure::draw.
- However, class Triangle and Triangle::draw were not written at the time Figure was written.
- If the function draw is declared virtual (abstract) function in class Figure, then things work correctly. How?
- By declaring a member function virtual in a super-class, we are telling the compiler to wait until this function is used in a program to decide what implementation to use.
- This is called late binding or dynamic binding.
Late Binding (4 of 4)

- Technically:
  - Late or dynamic binding is carried out with a table, called the virtual table, that is hidden from the programmer.
  - If you have a virtual function for which there is no implementation, you may get an error message that mentions a virtual table.
  - When a derived class needs a virtual base class function, the system decides at run time which of several available functions to use.
  - We will show how the system makes these decisions later.

Virtual Functions in C++ (1 of 5)

- In an automobile parts store point of sales program:
  - We cannot account for all possible types of sales but
  - We want to make the program versatile enough to be sure it is possible to account for all possibilities in the future.
  - Initially, there are only retail sales of single parts.
  - Later we include support for
    - volume discount,
    - mail order sales that include shipping costs
  - This version must report the sum of gross sales daily.
  - Additional features intended for the future:
    - the largest and smallest sales, and
    - average
  - The additional features these can be computed from individual sales.

Virtual Functions in C++ (2 of 5)

- Only a stub, bill(), for comparing sales will be provided to allow delay of implementation until types of sales are known.
- To enable this, the sales computation function will be virtual.
- Display 15.8 contains the interface and Display 15.9 contains the implementation for the base class Sale.
- In 15.8, the keyword virtual is used with the prototype of the function bill.
- In 15.9, both the member function savings and the overloaded operator< use the function bill.
- The function bill is declared to be virtual, we can define derived classes of the class Sale each of which defines its own versions of the function bill.
- The version of bill in the derived classes will use the version of savings, and overloaded operator< that correspond to the object of the derived class.

Virtual Functions in C++ (3 of 5)

- Display 15.10 shows a derived class DiscountSale.
- The class DiscountSale requires a new definition for the function bill.
- The function savings and operator< will use the new version of bill given with DiscountSale.
- Note that the new version of the function bill had not been written at the time the functions savings and operator< were written.
- How can these functions know to use the DiscountSale version of bill?
- In C++ you assume that it happens automatically.
- When you define a function to be virtual, you are telling the C++ environment to create the necessary machinery so it can wait until the program is being run to decide how to get the correct implementation.
Virtual Functions in C++ (4 of 5)

- Display 15.11 contains a sample program illustrating how the virtual function `bill` and the functions that use `bill` work in a complete program.
- Some useful technical details:
  1. If a function will have a different definition in a derived class than in the base class, and you want it to be a virtual function, you place the `virtual` keyword in front of the function prototype in the base class.
  2. The property of being virtual is inherited. That means if the base class declares a function to be virtual, then a function with the same signature in a derived class will automatically be virtual. (It is a good practice to use the virtual keyword on function the derived class that are already virtual, though this is not required.)
  3. The virtual keyword is used in the prototype (in the class) but not in the function definition.
  4. You do not get a virtual function and you do not get the benefits of a virtual function unless you use the keyword virtual.

Some languages make all functions virtual.

Why doesn't C++ do that?

There is a small but significant cost in the use of virtual functions.

A C++ design rule says "If you don't use a feature, you don't pay for that feature."

C++ provides both late binding (at a cost) with virtual functions and early binding (at no cost) for functions that are not.

Rule: Use virtual functions only when you need them.

Virtual Functions in C++ (5 of 5)

Some languages make all functions virtual.

Why doesn't C++ do that?

There is a small but significant cost in the use of virtual functions.

A C++ design rule says "If you don't use a feature, you don't pay for that feature."

C++ provides both late binding (at a cost) with virtual functions and early binding (at no cost) for functions that are not.

Rule: Use virtual functions only when you need them.

Overriding

When a function definition is changed in a derived class, programmers often say the function definition is overridden. In C++ literature a distinction is made between the terms redefined and overridden. Both terms refer to changing the definition of the function in a derived class. If the function is a virtual function then this is called overriding. If the function is not a virtual function then it is called a redefinition.

This may seem silly -- a distinction without a difference, but these are treated differently by the compiler: One case, virtual functions and overriding, involves significant overhead while the other, redefinition, does not.
Polymorphism

The term polymorphism is made up of "poly", meaning many; "morph", meaning form; and "ism", an action suffix. The most general meaning of the term is "having many forms". In the most general sense in programming, the word refers to the association of multiple meanings with one function name.

Polymorphism is also used in a more restrictive sense. When used in this more restrictive sense, overloading refers to the ability to associate multiple meanings with one function name by means of the mechanism of late binding.

Remember, in C++, the name of a function is more than just the function identifier. In C++, the name of a function is the function's identifier together with the sequence of types in the parameter list.

Polymorphism is more than conventional function identifier overloading.

When we use polymorphism in the more restricted sense, polymorphism, late binding, and virtual functions all really are the same topic.

### Consider the declarations:

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

class Dog : public Pet {
  public:
    virtual void print( ); // virtual not needed, but
    // provides clarity and is good style
    string breed;
};

Dog vdog;
Pet vpet;
```

Virtual Functions and Extended Type Compatibility (1 of 5)

- Further consequences of declaring a class member function to be virtual and one example of use of some of these features follows:
  - C++ is strongly typed but relaxes this by providing automatic casts, called coercion, enabling assignment of a value of one type to variable of another type, such as int to double.
  - On the other hand, you cannot assign a double to any of the integer types (int, char, short, long).
  - However, strong typing prevents assignments between base and derived class objects.

### Virtual Functions and Extended Type Compatibility (2 of 5)

- Consider the declarations:

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

class Dog : public Pet {
  public:
    virtual void print( ); // virtual not needed, but
    // provides clarity and is good style
    string breed;
};

Dog vdog;
Pet vpet;
```

Virtual Functions and Extended Type Compatibility (3 of 5)

- We concentrate on data members name and breed.
  - Note that this is an example. In a real example, the data would be private or protected, and functions would be provided to manipulate them.
  - Anything that is a dog is also a pet. Thus these assignments should be reasonable and legal.
    - vdog.name = "Tiny";
    - vpet.breed = "Great Dane";
  - This is legal, but there are problems. The breed field is lost! The variable vpet has type Pet, and only members of Pet are available to the object vpet, regardless of what was assigned to it. This is the "slicing problem". Here is an attempted access and an error message:
    ```
    cout << vpet.breed;
    // Error: class Pet has no member named breed.
    ```
Virtual Functions and Extended Type Compatibility (4 of 5)

- It is arguable that this is reasonable, that vpet should be an ordinary Pet object though it was set from a Dog of type Pet.
- Interesting discussion — but little assistance with programming.
- The dog is named Tiny, it is a Great Dane, and we would like for it to retain its breed even if we treat it as a Pet along the way.
- C++ provides a mechanism to treat Dog as a Pet without slicing away the derived class attributes of breed and the member function associated with class Dog. Consider the declarations:

```cpp
Pet *vpet;
Dog *vdog;
```

- Using pointers and dynamic variables we can treat Tiny as a Pet yet keep the breed:

```cpp
vdog = new Dog;
vdog->name = "Tiny";
vdog->breed = "Great Dane";
```

- vpet cannot be a calling object for a member function introduced in Dog, and the data member Dog::breed is lost.
- Why? The base class doesn’t know about derived class extensions made in the inheritance process.

Virtual Functions and Extended Type Compatibility (5 of 5)

- We can still access the breed data member of the node pointed to by vpet, but not directly:
- Suppose Dog::print();
- is defined:

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

The statement

```cpp
vpet->print(); // (*ppet).print();
```

will cause the following to be printed:

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

by virtue of the fact that print() is a virtual member of Pet and Dog.

Bottom line: The Slicing Problem

- It is legal to assign a derived class object to a base class variable, because such an assignment slices off data belonging exclusively to the derived class object.
- Any data members in the derived class but not in the base class will be lost in the assignment and any member functions that are not defined in the base class are similarly unavailable to the resulting base class object.
- If we make the assignments
  - Dog vdog;
  - Pet vpet;
  - vdog.name = "Tiny";
  - vdog.breed = "Great Dane";
  - vpet = vdog;
  then vpet cannot be a calling object for a member function introduced in Dog, and the data member Dog::breed is lost.
- Why? The base class doesn’t know about derived class extensions made in the inheritance process.

Pitfall: Not Using virtual Member Functions (1 of 3)

- To get the benefit of extended type compatibility discussed in above, you must use virtual member functions.
- Example: If we have not used virtual member functions in Display 15.7, and suppose in place of

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

we had used

```cpp
cout << "name: " << vpet->name << endl;
cout << "breed: " << vpet->breed << endl;
```

This would have precipitated an error message to the effect that there is no member variable breed for an object of class Pet.

- The type of the variable vpet pointer to class Pet, thus the expression *vpet has type class Pet, which has no member variable breed.

Pitfall: Not Using virtual Member Functions (2 of 3)

- The function, print() was declared virtual in class Pet, so when the system sees the call

```cpp
vpet->print();
```

it looks in the virtual table for classes Pet and Dog, sees that vpet points to an object of type Dog, and calls the function

```cpp
Dog::print();
```

instead of

```cpp
Pet::print();
```

Pitfall: Not Using virtual Member Functions (3 of 3)

- Object oriented programming with dynamic variables is very different way of viewing programming. This can be bewildering.
- These rules will help:

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

```cpp
p_ancestor = p_descendant; // POINTER ASSIGNMENT!!!
```

Moreover, none of the data members or member functions of the dynamic variable being pointed to by p_descendant will be lost.

2. Although all the extra fields of the dynamic variable are there, you will need virtual member functions to access them.
We have advocated developing incrementally. This means code a little and test a little, then code a little more then test some more. Usually, you can ignore implementations of member functions you do not call.

This is definitely not the case for virtual member functions. An attempt to compile a program with a class that has even one virtual function that does not have an implementation, results in some very hard-to-understand error messages, even if you do not call the undefined member functions! An error message that one compiler gives is “undefined reference to Class_Name virtual table.” Such an error message results even if there is no derived class and there is only one virtual member, but that member is not defined.

It gets worse: For the functions declared virtual, there will be further error messages complaining about an undefined reference to a default constructor, even if these constructors are already defined.

The moral is clear: You should implement all virtual functions prior to compiling, even if it is just a dummy, as in

```cpp
class A
{
    public:
        // constructors
        virtual void foo();
        // the rest of the class members
    }

    void A::foo()
    {
        // dummy implementation to prevent hard to understand error messages.
    }
};
```

Inheritance provides a tool for code reuse by deriving one class from another, adding features to the derived. Derived class object inherit all the members of the base class, and may add members. Late binding means that the decision of which version of a member function is appropriate is decided at runtime. Virtual functions are what C++ uses to achieve late binding. A protected member in the base class is directly available to a publicly derived class’s member functions.