11.1 Derived classes

Derived class concept

Commonly, one class is similar to another class but with some additions or variations. Ex: A store inventory system might use a class called GenericItem that has itemName and itemQuantity data members. But for produce (fruits and vegetables), a ProduceItem class with data members itemName, itemQuantity, and expirationDate may be desired.

PARTICIPATION ACTIVITY 11.1.1: Creating a ProduceItem from GenericItem.

Animation content:

undefined

Animation captions:

1. A GenericItem has data members itemName and itemQuantity and 3 member functions.
2. A ProduceItem is very similar to a GenericItem, but a ProduceItem also needs an expirationDate data member.
3. A ProduceItem needs the same data member functions as a GenericItem plus functions to get and set the expiration date.
4. If ProduceItem is implemented as an independent class, all the data/function members from GenericItem must be copied into ProduceItem, creating lots of duplicate code.
5. If ProduceItem is implemented as a derived class, ProduceItem need only implement what is different between a GenericItem and ProduceItem.

PARTICIPATION ACTIVITY 11.1.2: Derived class concept.

1) Creating an independent class that has the same members as an existing class creates duplicate code.
   - ○ True
   - ○ False

2) Creating a derived class is generally less work than creating an independent class.
   - ○

True

False

Inheritance

A ***derived class*** (or ***subclass***) is a class that is derived from another class, called a ***base class*** (or ***superclass***). Any class may serve as a base class. The derived class is said to inherit the properties of the base class, a concept called ***inheritance***. An object declared of a derived class type has access to all the public members of the derived class as well as the public members of the base class.

A derived class is declared by placing a colon ":" after the derived class name, followed by a member access specifier like public and a base class name. Ex: `class DerivedClass: public BaseClass { ... };`. The figure below defines the base class GenericItem and derived class ProduceItem that inherits from GenericItem.

Figure 11.1.1: Class ProduceItem is derived from class GenericItem.

```
// Base class
class GenericItem {
   public:
      void SetName(string newName) {
          itemName = newName;
      }

      void SetQuantity(int newQty) {
          itemQuantity = newQty;
      }

      void PrintItem() {
          cout << itemName << " " << itemQuantity << endl;
      }

   private:
      string itemName;
      int itemQuantity;
};

// Derived class inherits from GenericItem
class ProduceItem : public GenericItem {
   public:
      void SetExpiration(string newDate) {
          expirationDate = newDate;
      }

      string GetExpiration() {
          return expirationDate;
      }

   private:
      string expirationDate;
};
```

PARTICIPATION ACTIVITY 11.1.3: Using GenericItem and ProduceItem objects.

Animation content:

undefined

Animation captions:

1. The base class GenericItem and derived class ProduceItem are declared.
2. miscItem is a GenericItem, which has 2 data members and 3 member functions.
3. Since ProduceItem is derived from GenericItem, ProduceItem inherits GenericItem's members and adds new members implemented in ProduceItem.
4. miscItem's name and quantity are set, and miscItem is printed to the screen.
5. perishItem's name, quantity, and expiration are set. Then perishItem is printed to the screen.

PARTICIPATION ACTIVITY 11.1.4: Derived classes.

1) A class that can serve as the basis for another class is called a _____ class.

Check Show answer

2) In the figure above, how many total class members does GenericItem contain?

Check Show answer

3) In the figure above, how many total class members are unique to ProduceItem?

Check Show answer

4) Class Dwelling has data members door1, door2, door3. A class House is derived from Dwelling and has data

members wVal, xVal, yVal, zVal. How many data members does `House h;` create?

Check Show answer

Inheritance scenarios

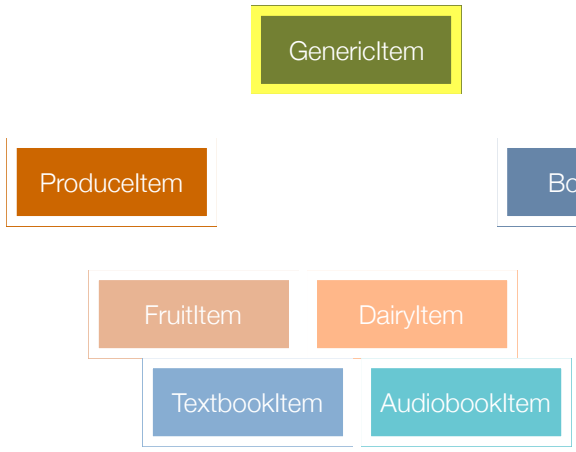
Various inheritance variations are possible:

- A derived class can serve as a base class for another class. Ex: `class FruitItem: public ProduceItem {...}` creates a derived class FruitItem from ProduceItem, which was derived from GenericItem.
- A class can serve as a base class for multiple derived classes. Ex: `class FrozenFoodItem: public GenericItem {...}` creates a derived class FrozenFoodItem that inherits from GenericItem, just as ProduceItem inherits from GenericItem.
- A class may be derived from multiple classes. Ex: `class House: public Dwelling, public Property {...}` creates a derived class House that inherits from base classes Dwelling and Property.

PARTICIPATION ACTIVITY 11.1.5: Interactive inheritance tree.

Click a class to see available functions and data for that class.

Inheritance tree	Selected class pseudocode




```
public:
    void SetName(string newName)
    void SetQuantity(int newQty)
    void PrintItem()

private:
    string itemName;
    int itemQuantity;
```


Selected class code

```
class GenericItem {
   public:
      void SetName(string newName)
         { itemName = newName; };
      void SetQuantity(int newQty)
         { itemQuantity = newQty; };
      void PrintItem() {
         cout << itemName << " "
              << itemQuantity
              << endl;
      };

   private:
      string itemName;
      int itemQuantity;
};
```


PARTICIPATION ACTIVITY 11.1.6: Inheritance scenarios.

Refer to the interactive inheritance tree above.

1) The BookItem class acts as a derived class and a base class.
   - True
   - False

2) ProduceItem and BookItem share some of the same class members.
   - True
   - False

3) DairyItem and TextbookItem share some of the same class members.
   - True
   - False

4) AudiobookItem inherits the data member called readerName from BookItem.
   - True
   - False

5) AudiobookItem inherits the member function GetTitle() from BookItem.

- True
- False

Example: Business and Restaurant

The example below defines a Business class with data members name and address. The Restaurant class is derived from Business and adds a rating data member with a getter and setter.

Figure 11.1.2: Inheritance example: Business and Restaurant classes.

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

class Business {
   public:
      void SetName(string busName) {
         name = busName;
      }

      void SetAddress(string busAddress) {
         address = busAddress;
      }

      string GetDescription() const {
         return name + " -- " + address;
      }

   private:
      string name;
      string address;
};

class Restaurant : public Business {
   public:
      void SetRating(int userRating) {
         rating = userRating;
      }

      int GetRating() const {
         return rating;
      }

   private:
      int rating;
};

int main() {
   Business someBusiness;
   Restaurant favoritePlace;

   someBusiness.SetName("ACME");
   someBusiness.SetAddress("4 Main St");

   favoritePlace.SetName("Friends Cafe");
   favoritePlace.SetAddress("500 W 2nd Ave");
   favoritePlace.SetRating(5);

   cout << someBusiness.GetDescription() << endl;
   cout << favoritePlace.GetDescription() << endl;
   cout << "  Rating: " << favoritePlace.GetRating() << endl;

   return 0;
}
```



```
ACME -- 4 Main St
Friends Cafe -- 500 W 2nd Ave
  Rating: 5
```

©zyBooks 04/25/21 07:37 488201
xiang zhao
BAYLORCSI14301440Spring2021

PARTICIPATION ACTIVITY 11.1.7: Inheritance example.

Refer to the code above.

1) How many member functions are defined in Restaurant?

- 2
- 3
- 5

2) How many member functions can a Restaurant object call?

- 2
- 3
- 5

3) Which function call produces a syntax error?

- `someBusiness.SetRating(4);`
- `favoritePlace.GetRating();`
- `favoritePlace.SetRating(4);`

4) What is the best way to declare a new DepartmentStore class?

- ```
  class DepartmentStore {
  ... };
  ```
- ```
  class DepartmentStore :
  public Restaurant { ...
  };
  ```
- ```
  class DepartmentStore :
  public Business { ... };
  ```

Exploring further:

- Inheritance (C++) from msdn.microsoft.com.

CHALLENGE ACTIVITY 11.1.1: Derived classes.

Start

Type the program's output
  ```

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

class Vehicle {
   public:
      void SetSpeed(int speedToSet) {
         speed = speedToSet;
      }

      void PrintSpeed() {
         cout << speed;
      }

   private:
      int speed;
};

class Car : public Vehicle {
   public:
      void PrintCarSpeed() {
         cout << "Driving at: ";
         PrintSpeed();
      }
};

int main() {
   Car myCar;
   myCar.SetSpeed(45);

   myCar.PrintCarSpeed();

   return 0;
}
```


1 2 3

Check Next

CHALLENGE ACTIVITY 11.1.2: Basic inheritance.

Assign courseStudent's name with Smith, age with 20, and ID with 9999. Use the PrintAll() member function and a separate cout statement to output courseStudents's data. End with a newline. Sample output from the given program:

```
Name: Smith, Age: 20, ID: 9999
```



```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
```

```
class PersonData {
   public:
      void SetName(string userName) {
         lastName  = userName;
      }

      void SetAge(int numYears) {
         ageYears = numYears;
      }

      // Other parts omitted

      void PrintAll() {
```


Run

11.2 Access by members of derived classes

Member access

The members of a derived class have access to the public members of the base class, but not to the private members of the base class. This is logical—allowing access to all private members of a class merely by creating a derived class would circumvent the idea of private members. Thus, adding the following member function to the Restaurant class yields a compiler error.

Figure 11.2.1: Member functions of a derived class cannot access private members of the base class.

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

class Business {
   private:
      string name;
      string address;

   public:

       ...

};

class Restaurant : public Business {
   private:
      int rating;

   public:

       ...

       void DisplayRestaurant() {
         cout << name << endl;
         cout << address << endl;
         cout << " Rating: " << rating << endl;
       }
};
```



```
$ g++ -Wall Restaurant.cpp
Restaurant.cpp: In member function 'void Restaurant::DisplayRestaurant()':
Restaurant.cpp:14: error: 'std::string Business::name' is private
Restaurant.cpp:25: error: within this context
Restaurant.cpp:15: error: 'std::string Business::address' is private
Restaurant.cpp:25: error: within this context
```

PARTICIPATION ACTIVITY 11.2.1: Access by derived class members.

Assume `class Restaurant: public Business{...}`

1) Business's public member function can be called by a member function of Restaurant.

- ○ True
- ○ False

2) Restaurant's private data members can be accessed by Business.

- ○ True

False

Protected member access

Recall that members of a class may have their access specified as *public* or *private*. A third access specifier is ***protected***, which provides access to derived classes but not by anyone else. The following illustrates the implications of the protected access specifier.

In the following example, the member called name is specified as protected and is accessible anywhere in the derived class. Note however that the name member is not accessible in main() -- the protected specifier only applies to derived classes; protected members are private to everyone else.

Figure 11.2.2: Access specifiers -- Protected allows access by derived classes but not by others.

Code contains intended errors to demonstrate protected accesses.

```
class Business {
   protected: // Members accessible by self and derived classes
      string name;

   private:   // Members accessible only by self
      string address;

   public:    // Members accessible by anyone
      void PrintMembers();
};

class Restaurant : public Business {
   private:
      int rating;

   public:

      ...

      void DisplayRestaurant() {
         // Attempted accesses
         PrintMembers();             // OK
         name = "Gyro Hero";         // OK    ("protected" above made this possible)
         address = "5 Fifth St";     // ERROR
      }

      // Other class members ...

};

int main() {
   ...

   Business business;
   Restaurant restaurant;

   ...

   // Attempted accesses
   business.PrintMembers();          // OK
   business.name  = "Gyro Hero";     // ERROR (protected only applies to derived classes)
   business.address = "5 Fifth St";  // ERROR

   restaurant.PrintMembers();         // OK
   restaurant.name  = "Gyro Hero";    // ERROR
   restaurant.rating = 5;  // ERROR

   return 0;
}
```


©zyBooks 04/25/21 07:37 488201
xiang zhao
BAYLORCSI14301440Spring2021

To make Restaurant's DisplayRestaurant() function work, we merely need to change the private members to protected members in class Business. Business's data members name and address thus become accessible to a derived class like Restaurant, but not elsewhere. A programmer may often want to make some members protected in a base class to allow access by derived classes, while making other members private to the base class.

The following table summarizes access specifiers.

Table 11.2.1: Access specifiers.

Specifier	Description
private	Accessible by self.
protected	Accessible by self and derived classes.
public	Accessible by self, derived classes, and everyone else.

PARTICIPATION ACTIVITY 11.2.2: Protected access specifier.

Assume `class Restaurant: public Business{...}`.

Suppose a new class, `class SkateShop{...}`, is defined.

1) Business's protected data members can be accessed by a member function of Restaurant.
   - True
   - False

2) Business's protected data members can be accessed by a member function of SkateShop.
   - True
   - False

Class definitions

Separately, the keyword "public" in a class definition like `class DerivedClass: public BaseClass {...}` has a different purpose, relating to the kind of inheritance being carried out:

- *public* : "public-->public, protected-->protected" -- public members of BaseClass are accessible as public members of DerivedClass, and protected members of BaseClass are accessible as protected members of DerivedClass.

- *protected* : "public-->protected, protected-->protected" -- public and protected members of BaseClass are accessible as protected members of DerivedClass.
- *private* : "public-->private, protected-->private" -- public and protected members of BaseClass are accessible as private members of DerivedClass. Incidentally, if the specifier is omitted as in "class DerivedClass: BaseClass {...}", the default is private.

Most derived classes created when learning to program use public inheritance.

PARTICIPATION ACTIVITY 11.2.3: Access specifiers for class definitions.

Suppose a public data member `string telephone;` is added to the Business class, and a new class `class FoodTruck` is derived from Restaurant.

For the following cases, which specifier should be used for `class Restaurant: _____ Business{...}`?

1) Restaurant's telephone data member cannot be accessed by main() but can be accessed by a FoodTruck member function .
   - public
   - protected
   - private

2) Restaurant's telephone data member can be accessed by main() and by a FoodTruck member function .
   - public
   - protected
   - private

3) Restaurant's telephone data member cannot be accessed by main() or by a FoodTruck member function .
   - public
   - protected
   - private

Exploring further:

- More on Protected from msdn.microsoft.com

11.3 Overriding member functions

Overriding

When a derived class defines a member function that has the same name and parameters as a base class's function, the member function is said to ***override*** the base class's function. The example below shows how the Restaurant's GetDescription() function overrides the Business's GetDescription() function.

PARTICIPATION ACTIVITY 11.3.1: Overriding member function example.

Animation content:

```
undefined
```

Animation captions:

1. The Business class defines a GetDescription() member function that returns a string with the business name and address.
2. Restaurant derives from Business and defines a member function with the same name, return type, and parameters as the base class function GetDescription().
3. The Restaurant object favoritePlace calls the Restaurant's GetDescription(), which overrides the base class's GetDescription().

Overriding vs. overloading

Overriding differs from overloading. In overloading, functions with the same name must have different parameter types. In overriding, a derived class member function takes precedence over a base class member function with the same name, regardless of the parameter types. Overloading is not performed if derived and base member functions have different parameter types; the member function of the derived class hides the member function of the base class.

PARTICIPATION

ACTIVITY 11.3.2: Overriding.

Refer to the code above.

1) If a Restaurant object calls GetDescription(), the Restaurant's GetDescription() is called instead of Business's GetDescription().
   - True
   - False

2) If a Business object calls GetDescription(), the Restaurant's GetDescription() is called instead of Business's GetDescription().
   - True
   - False

3) Removing Business's GetDescription() function in the example above causes a syntax error.
   - True
   - False

4) Changing Restaurant's `string GetDescription()` to `string GetDescription(int num)` and not changing the function call, `favoritePlace.GetDescription()`, causes an error.
   - True
   - False

5) Changing Business's name and address data members from protected to private in the example above causes a syntax error.
   - True
   - False

Calling a base class function

An overriding function can call the overridden function by prepending the base class name. Ex: `Business::GetDescription()`.

Figure 11.3.1: Function calling overridden function of base class.

```
class Restaurant : public Business {

   ...

   string GetDescription() const {
      return Business::GetDescription() + "\n  Rating: " + to_string(rating);
   };

   ...
};
```


A common error is to leave off the prepended base class name when wanting to call the base class's function. Without the prepended base class name, the call to GetDescription() refers to itself (a *recursive* call), so GetDescription() would call itself, which would call itself, etc., never actually printing anything.

PARTICIPATION ACTIVITY 11.3.3: Override example.

Choose the correct replacement for the missing code below so ProduceItem's PrintItem() overrides GenericItem's PrintItem().

```
class GenericItem {
   public:
      ...
      void PrintItem() const {
         cout << itemName << " " << itemQuantity << endl;
      }

   __(A)__:
      string itemName;
      int itemQuantity;
};

class ProduceItem : public GenericItem {
   public:
      void SetExpiration(string newDate) {
         expirationDate = newDate;
      }

      string GetExpiration() const {
         return expirationDate;
      }

      __(B)__ {
         __(C)__;
         cout << "  (expires " << expirationDate << ")" << endl;
      }

   private:
      string expirationDate;
};
```


1) (A)

- private
- public

2) (B)

- void printItem(int numItem) const
- void PrintItem() const
- void PrintItem()

3) (C)

- PrintItem()
- ProduceItem::PrintItem()
- GenericItem::PrintItem()

©zyBooks 04/25/21 07:37 488201
xiang zhao
BAYLORCSI14301440Spring2021

CHALLENGE ACTIVITY 11.3.1: Overriding member function.

Start

Type the program's output

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

class Computer {
public:
   void SetComputerStatus(string cpuStatus, string internetStatus) {
      cpuUsage = cpuStatus;
      internet = internetStatus;
   };

   void PrintStatus() {
      cout << "CPU: " << cpuUsage << endl;
      cout << "Internet: " << internet << endl;
   };

protected:
   string cpuUsage;
   string internet;
};

class Laptop : public Computer {
public:
   void SetWiFiStatus(string wifiStatus) {
      wifiQuality = wifiStatus;
   };

   void PrintStatus() {
      cout << "WiFi: " << wifiQuality << endl;
      cout << "CPU: " << cpuUsage << endl;
   };

private:
   string wifiQuality;
};

int main() {
   Laptop myLaptop;

   myLaptop.SetComputerStatus("25%", "connected");
   myLaptop.SetWiFiStatus("bad");

   myLaptop.PrintStatus();

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 11.3.2: Basic derived class member override.

Define a member function PrintAll() for class PetData that prints output as follows with inputs "Fluffy", 5, and 4444. Hint: Make use of the base class' PrintAll() function.

```
Name: Fluffy, Age: 5, ID: 4444
```

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

class AnimalData {
public:
   void SetName(string givenName) {
      fullName = givenName;
   };
   void SetAge(int numYears) {
      ageYears = numYears;
   };
   // Other parts omitted

   void PrintAll() {
      cout << "Name: "  << fullName;
      cout << ", Age: " << ageYears;
   };
```


Run

11.4 Polymorphism and virtual member functions

Polymorphism

Polymorphism refers to determining which program behavior to execute depending on data types. Two main types of polymorphism exist:

- ***Compile-time polymorphism*** is when the compiler determines which function to call at compile-time.
- ***Runtime polymorphism*** is when the compiler is unable to determine which function to call at compile-time, so the determination is made while the program is running.

Function overloading is an example of compile-time polymorphism where the compiler determines which of several identically-named functions to call based on the function's arguments.

One scenario requiring runtime polymorphism involves derived classes. Programmers commonly create a collection of objects of both base and derived class types. Ex: The statement

`vector<Business*> businessList;` creates a vector that can contain pointers to objects of type Business or Restaurant, since Restaurant is derived from Business. Similarly, polymorphism is also used for references to objects. Ex: `Business& primaryBusiness` declares a reference that can refer to Business or Restaurant objects.

PARTICIPATION ACTIVITY 11.4.1: Compile-time polymorphism vs. runtime polymorphism.

Animation content:

`undefined`

Animation captions:

1. The DriveTo() function is overloaded. The first version has a string parameter, and the second version a Restaurant parameter.
2. With compile-time polymorphism, the compiler knows at compile-time to call the first DriveTo() because the argument "Big Mac's" is a string.
3. The DriveTo() function has a Business pointer parameter and calls the GetDescription() function.
4. businessList is a vector of Business pointers. Several pointers to Business and Restaurant objects are pushed onto businessList.
5. index is assigned with a randomly chosen integer. The compiler cannot determine at compile-time if DriveTo() is called with a Business pointer or Restaurant pointer.
6. With runtime polymorphism, the decision is made at runtime to call Restaurant GetDescription() instead of Business GetDescription() for a Restaurant pointer.

The program above uses a C++ feature called ***derived/base class pointer conversion***, where a pointer to a derived class is converted to a pointer to the base class without explicit casting. The above statement `businessList.push_back(restaurantPtr);` uses derived/base class pointer conversion to convert the Restaurant pointer to a Business pointer (businessList is a vector of Business pointers).

PARTICIPATION ACTIVITY 11.4.2: Polymorphism.

Refer to the code above.

1) An item of type `Restaurant*` may be added to a vector of type `vector<Business*>`.
   - ○ True
   - ○ False

2) An item of type `Business*` may be added to a vector of type `vector<Restaurant*>`.
   - True
   - False

3) If DriveTo() in the bottom code segment is called with a `Business*`, runtime polymorphism executes the Restaurant GetDescription() function.
   - True
   - False

4) If only `Restaurant*` objects are added to the businessList vector, the compiler could use compile-time polymorphism to call Restaurant GetDescription().
   - True
   - False

Polymorphism word origin

Polymorphism is composed of two Greek words: "poly" means many, and "morphe" means form. A polymorphic object like a Restaurant can take on other forms, like a Business object.

Virtual functions

Runtime polymorphism only works when an overridden member function in a base class is virtual. A ***virtual function*** is a member function that may be overridden in a derived class and is used for runtime polymorphism. A virtual function is declared by prepending the keyword "virtual". Ex: `virtual string GetDescription() const`. At runtime, when a virtual function is called using a pointer, the correct function to call is dynamically determined based on the actual object type to which the pointer or reference refers.

The ***override*** keyword is an optional keyword used to indicate that a virtual function is overridden in a derived class. Good practice is to use the override keyword when overriding a virtual function to avoid accidentally misspelling the function name or typing the wrong parameters.

PARTICIPATION ACTIVITY 11.4.3: Adding a virtual function to the base class.

Animation content:

undefined

Animation captions:

1. Restaurant overrides GetDescription() from the base class Business without using the "override" keyword.
2. businessPtr points to favoriteCafe. businessPtr->GetDesciption() calls Business GetDescription() instead of Restaurant GetDescription() because runtime polymorphism does not work without virtual functions.
3. The "virtual" keyword makes GetDescription() a virtual function. The "override" keyword indicates that Restaurant GetDescription() overrides the base class GetDescription().
4. Running the program with an overridden virtual function causes runtime polymorphism to call Restaurant GetDescription().

Virtual table

*To implement virtual functions, the compiler creates a **virtual table** that allows the computer to quickly lookup which function to call at runtime. The virtual table contains an entry for each virtual function with a function pointer that points to the most-derived function that is accessible to each class. Looking up which function to call makes runtime polymorphism slower than compile-time polymorphism.*

The program below illustrates how runtime polymorphism is used with a vector. businessList is a vector of Business pointers but holds Business and Restaurant pointers. A for loop iterates over businessList and calls each Business pointer's GetDescription() function. Restaurant GetDescription() is called when a Restaurant pointer is accessed because GetDescription() overrides the base class's virtual function.

Figure 11.4.1: Runtime polymorphism via a virtual function.

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

class Business {
public:
   void SetName(string busName) {
      name = busName;
   }

   void SetAddress(string busAddress) {
      address = busAddress;
   }

   virtual string GetDescription() const {
      return name + " -- " + address;
   }

protected:
   string name;
   string address;
};

class Restaurant : public Business {
public:
   void SetRating(int userRating) {
      rating = userRating;
   }

   int GetRating() const {
      return rating;
   }

   string GetDescription() const override {
      return name + " -- " + address +
         "\n  Rating: " + to_string(rating);
   }

private:
   int rating;
};

int main() {
   unsigned int i;
   vector<Business*> businessList;
   Business* businessPtr;
   Restaurant* restaurantPtr;

   businessPtr = new Business;
   businessPtr->SetName("ACME");
   businessPtr->SetAddress("4 Main St");

   restaurantPtr = new Restaurant;
   restaurantPtr->SetName("Friends Cafe");
   restaurantPtr->SetAddress("500 2nd Ave");
   restaurantPtr->SetRating(5);

   businessList.push_back(businessPtr);
   businessList.push_back(restaurantPtr);

   for (i = 0; i < businessList.size(); ++i) {
      cout << businessList.at(i)->GetDescription() << endl;
   }

   return 0;
}
```



```
ACME -- 4 Main St
Friends Cafe -- 500 2nd Ave
  Rating: 5
```

©zyBooks 04/25/21 07:37 488201
xiang zhao
BAYLORCSI14301440Spring2021

PARTICIPATION ACTIVITY 11.4.4: Polymorphism with virtual functions.

Refer to the code above.

1) For Restaurant GetDescription() to be called when a Restaurant pointer is accessed, what function must be a virtual function?

   - Business GetDescription() only
   - Restaurant GetDescription() only
   - Both Business and Restaurant GetDescription() functions

2) What does the code below output?

```
void PrintDescription(Business&
business) {
   cout << business.GetDescription()
<< endl;
}

int main() {
   Restaurant favoritePlace;
   favoritePlace.SetName("Friends
Cafe");
   favoritePlace.SetAddress("500 2nd
Ave");
   favoritePlace.SetRating(5);
   PrintDescription(favoritePlace);
}
```

   - `Friends Cafe -- 500 2nd Ave`
`   Rating: 5`
   - `Friends Cafe -- 500 2nd Ave`
   - Syntax error results.

3) What happens if the "virtual" keyword is removed from the program above?

   - The program runs the same as before.
   - The program displays different output.

- Compiling the program generates a syntax error.

4) If the "virtual" and "override" keywords are removed from the program above, what does the code below output?

```
cout << businessList.at(1)-
>GetDescription();
```

- `Friends Cafe -- 500 2nd Ave`
`Rating: 5`
- `Friends Cafe -- 500 2nd Ave`
- Runtime error results.

5) If the "virtual" and "override" keywords are removed from the program above, what does the code below output?

```
Restaurant favoritePlace;
favoritePlace.SetName("Friends
Cafe");
favoritePlace.SetAddress("500 2nd
Ave");
favoritePlace.SetRating(5);
cout <<
favoritePlace.GetDescription();
```

- `Friends Cafe -- 500 2nd Ave`
`Rating: 5`
- `Friends Cafe -- 500 2nd Ave`
- Runtime error results.

Pure virtual functions

Sometimes a base class should not provide a definition for a member function, but all derived classes must provide a definition. Ex: A Business may require all derived classes to define a GetHours() function, but the Business class does not provide a default GetHours() function.

A ***pure virtual function*** is a virtual function that provides no definition in the base class, and all derived classes must override the function. A pure virtual function is declared like a virtual function with the "virtual" keyword but is assigned with 0. Ex: `virtual string GetHours() const = 0;` declares a pure virtual function GetHours().

A class that has at least one pure virtual function is known as an ***abstract base class***. An abstract base class object cannot be declared. Ex: The variable declaration `Business someBusiness;` generates a syntax error if Business is an abstract base class.

Figure 11.4.2: Business is an abstract base class.


```
class Business {
public:
   void SetName(string busName) {
      name = busName;
   }

   void SetAddress(string busAddress) {
      address = busAddress;
   }

   virtual string GetDescription() const {
      return name + " -- " + address;
   }

   virtual string GetHours() const = 0;     // pure virtual function

protected:
   string name;
   string address;
};
```

In the above example, the programmer may intend to create several classes derived from Business, such as Restaurant, LawnService, and CoffeeShop. The abstract base class Business implements functionality common to all derived classes, thus avoiding redundant code in all derived classes, and supporting uniform treatment of a collection (e.g., vector) of objects of derived classes via polymorphism. Not overriding the pure virtual function in a derived class makes the derived class an abstract base class too.

PARTICIPATION ACTIVITY 11.4.5: Pure virtual functions.

Refer to the code below.

```
class GenericItem {
public:
   void SetName(string newName) {
      itemName = newName;
   }

   void PrintItem() const {
      cout << itemName << endl;
   }

protected:
   string itemName;
};

class ProduceItem : public GenericItem {
public:
   void SetExpiration(string newDate) {
      expirationDate = newDate;
   }

   void PrintItem() const {
      cout << itemName << " (expires " << expirationDate << ")" << endl;
   }

private:
   string expirationDate;
};

void PrintAllItems(const vector<GenericItem*> &items) {
   unsigned int i;
   for (i = 0; i < items.size(); ++i) {
      // Missing code
   }
}
```


1) Write all the code necessary to declare GenericItem PrintItem() as a pure virtual function.

Check Show answer

2) Change ProduceItem PrintItem() to override GenericItem PrintItem().

```
void                          {
   cout << itemName << "
(expires " << expirationDate
         << ")" << endl;
}
```


Check Show answer

3) Write the line of code missing from the for loop to call each item's PrintItem().

```
for (i = 0; i < items.size();
i++) {
    ;
}
```

Check Show answer

4) If GenericItem PrintItem() is a pure virtual function, which class is an abstract base class?

Check Show answer

Possible warning messages when using virtual functions

The following are possible warning messages when using virtual functions. The reason for the warnings is that the base class may have pointer data members that may not be destroyed. Newer compilers may not generate warning messages unless the base class actually contains pointer data members.

```
polydemo.cpp:6: warning: 'class Business' has virtual functions but non-virtual
destructor
polydemo.cpp:19: warning: 'class Restaurant' has virtual functions but non-
virtual destructor
```

Exploring further:

- More on Polymorphism from cplusplus.com

CHALLENGE ACTIVITY 11.4.1: Polymorphism and virtual member functions.

Start

Type the program's output

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

class Watch {
public:
   void SetHours(int watchHours) {
      hours = watchHours;
   }

   void SetMins(int watchMins) {
      mins = watchMins;
   }

   virtual void PrintItem() {
      cout << hours << ":" << mins << endl;
   }

protected:
   int hours;
   int mins;
};

class SmartWatch : public Watch {
public:
   void SetPercentage(int watchPercentage) {
      batteryPercentage = watchPercentage;
   }

   void PrintItem() {
      cout << hours << ":" << mins << " " << batteryPercentage << "%" << endl;
   }

private:
   int batteryPercentage;
};

int main() {
   Watch* watch1;
   SmartWatch* watch2;
   Watch* watch3;

   vector<Watch*> watchList;
   unsigned int i;

   watch1 = new Watch();
   watch1->SetHours(6);
   watch1->SetMins(54);

   watch2 = new SmartWatch();
   watch2->SetHours(7);
   watch2->SetMins(11);
   watch2->SetPercentage(58);

   watch3 = new Watch();
   watch3->SetHours(8);
   watch3->SetMins(53);

   watchList.push_back(watch2);
   watchList.push_back(watch3);
   watchList.push_back(watch1);

   for (i = 0; i < watchList.size(); ++i) {
      watchList.at(i)->PrintItem();
   }

   return 0;
}
```

1 2

Check Next

CHALLENGE ACTIVITY 11.4.2: Basic polymorphism.

Write the PrintItem() function for the base class. Sample output for below program:

```
Last name: Smith
First and last name: Bill Jones
```

Hint: Use the keyword const to make PrintItem() a virtual function.

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

class BaseItem {
public:
   void SetLastName(string providedName) {
      lastName = providedName;
   };

   // FIXME: Define PrintItem() member function

   /* Your solution goes here  */

protected:
   string lastName;
};
```

Run

11.5 Abstract classes: Introduction (generic)

Abstract classes

Object-oriented programming (OOP) is a powerful programming paradigm, consisting of several features. Three key features include:

- ***Classes:*** A class encapsulates data and behavior to create objects.
- ***Inheritance:*** Inheritance allows one class (a subclass) to be based on another class (a base class or superclass). Ex: A Shape class may encapsulate data and behavior for geometric shapes, like setting/getting the Shape's name and color, while a Circle class may be a subclass of a Shape, with additional features like setting/getting the center point and radius.
- ***Abstract classes:*** An ***abstract class*** is a class that guides the design of subclasses but cannot itself be instantiated as an object. Ex: An abstract Shape class might also specify that any subclass must define a ComputeArea() function.

PARTICIPATION ACTIVITY 11.5.1: Classes, inheritance, and abstract classes.

Animation content:

undefined

Animation captions:

1. A class provides data/behaviors for objects.
2. Inheritance creates a Circle subclass that implements behaviors specific to a circle.
3. The abstract Shape class specifies "Compute area" is a required behavior of a subclass. Shape does not implement "Compute area", so a Shape object cannot be created.
4. The Circle class implements "Compute area". The Circle class is a non abstract, which is also called a concrete class, and Circle objects can be created.

PARTICIPATION ACTIVITY 11.5.2: Classes, inheritance, and abstract classes.

Consider the example above.

1) The Shape class is an abstract class, and the Circle class is a concrete class.
   - ○ True
   - ○ False

2) The Shape class can be instantiated as an object.
   - ○ True
   - ○ False

3) The Circle class can be instantiated as an object.
   - ○ True

False

4) The Circle class must implement the ComputeArea() function.

- True
- False

Example: Biological classification

An example of abstract classes in action is the classification hierarchy used in biology. The upper levels of the hierarchy specify features in common across all members below that level of the hierarchy. As with concrete classes that implement all abstract methods, no creature can actually be instantiated except at the species level.

PARTICIPATION ACTIVITY 11.5.3: Biological classification uses abstract classes.

Animation content:

undefined

Animation captions:

1. Each level of the biological hierarchy specifies behaviors common to that level.
2. At this level of the hierarchy, a lot of behavior for the organism is known but the organism is not yet specified.
3. At the final level (species), the organism can be fully described, just as a concrete class can be fully instantiated.

PARTICIPATION ACTIVITY 11.5.4: Abstract classes.

1) Consider a program that catalogs the types of trees in a forest. Each tree object contains the tree's species type, age, and location. This program will benefit from an abstract class to represent the trees.

- True
- False

2) Consider a program that catalogs the types of trees in a forest. Each tree

object contains the tree's species type, age, location, and estimated size based on age. Each species uses a different formula to estimate size based on age. This program will benefit from an abstract class.

- ○ True
- ○ False

3) Consider a program that maintains a grocery list. Each item, like eggs, has an associated price and weight. Each item belongs to a category like produce, meat, or cereal, where each category has additional features, such as meat having a "sell by" date. This program will benefit from an abstract class.

- ○ True
- ○ False

11.6 Abstract classes

Abstract and concrete classes

A ***pure virtual function*** is a virtual function that is not implemented in the base class, thus all derived classes must override the function. A pure virtual function is declared with the "virtual" keyword and is assigned with 0. Ex: `virtual double ComputeArea() const = 0;` declares a pure virtual function ComputeArea().

An ***abstract class*** is a class that cannot be instantiated as an object, but is the superclass for a subclass and specifies how the subclass must be implemented. Any class with one or more pure virtual functions is abstract.

A ***concrete class*** is a class that is not abstract, and hence *can* be instantiated.

PARTICIPATION ACTIVITY 11.6.1: A Shape class with a pure virtual function is an abstract class.

Animation content:

```
undefined
```

Animation captions:

1. The Shape class has the pure virtual ComputeArea() function. The Shape class is abstract due to having a pure virtual function.
2. An abstract class cannot be instantiated.

PARTICIPATION ACTIVITY 11.6.2: Shape class.

1) Shape is an abstract class.
   - True
   - False

2) The Shape class defines and provides code for non-pure-virtual function.
   - True
   - False

3) Any class that inherits from Shape must implement the ComputeArea(), GetPosition(), SetPosition(), and MovePositionRelative() functions.
   - True
   - False

Ex: Shape classes

The example program below manages sets of shapes. Shape is an abstract class, and Circle and Rectangle are concrete classes. The Shape abstract class specifies that any derived class must define a function computeArea() that returns type double.

Figure 11.6.1: Shape is an abstract class. Circle and Rectangle are concrete classes that extend the Shape class.

Shape.h Shape is declared as an abstract base	Point.h A Point object holds the x, y coordinates for a

class

```
class Shape {
   protected:
      Point position;

   public:
      virtual ~Shape() { }
      virtual double ComputeArea()
const = 0;

      Point GetPosition() const {
         return position;
      }

      void SetPosition(Point
newPosition) {
         position = newPosition;
      }

      void MovePositionRelative(Point
otherPosition) {
         double x = position.GetX() +
otherPosition.GetX();
         double y = position.GetY() +
otherPosition.GetY();

         position.SetX(x);
         position.SetY(y);
      }
};
```

point

```
class Point {
   private:
      double x;
      double y;

   public:
      Point() {
         x = 0;
         y = 0;
      }

      Point(double x, double y) {
         this->x = x;
         this->y = y;
      }

      double GetX() const {
         return x;
      }

      double GetY() const {
         return y;
      }

      void SetX(double x) {
         this->x = x;
      }

      void SetY(double y) {
         this->y = y;
      }
};
```


Circle.h

Defines a Circle class

```
class Circle : public Shape {
   private:
      double radius;

   public:
      Circle(Point center, double
radius) {
         this->radius = radius;
         this->position = center;
      }

      double ComputeArea() const {
         return (M_PI * pow(radius,
2));
      }
};
```

Rectangle.h

Defines a Rectangle class

```
class Rectangle : public Shape {
   private:
      double length, height;

   public:
      Rectangle(Point upperLeft, double
length, double height) {
         this->position = upperLeft;
         this->length = length;
         this->height = height;
      }

      double ComputeArea() const {
         return (length * height);
      }
};
```


TestShapes.cpp

Implements the main function to test the Shape classes

```
int main() {
   Circle* circle1 = new Circle(Point(1.0, 1.0), 1.0);
   Circle* circle2 = new Circle(Point(1.0, 1.0), 2.0);

   Rectangle* rectangle = new Rectangle(Point(0.0, 1.0), 1.0, 1.0);

   // Print areas
   cout << "Area of circle 1 is: " << circle1->ComputeArea() << endl;
   cout << "Area of circle 2 is: " << circle2->ComputeArea() << endl;
   cout << "Area of rectangle is: " << rectangle->ComputeArea() << endl;
   cout << endl;

   // Print positions
   cout << "Circle 1 is at: (" << circle1->GetPosition().GetX();
   cout << ", " << circle1->GetPosition().GetY() << ")" << endl;

   cout << "Rectangle is at: (" << rectangle->GetPosition().GetX();
   cout << ", " << rectangle->GetPosition().GetY() << ")" << endl;
   cout << endl;

   // Move shapes
   circle1->SetPosition(Point(3.0, 1.0));
   rectangle->MovePositionRelative(Point(1.0, 1.0));

   // Print positions
   cout << "Circle 1 is at: (" << circle1->GetPosition().GetX();
   cout << ", " << circle1->GetPosition().GetY() << ")" << endl;

   cout << "Rectangle is at: (" << rectangle->GetPosition().GetX();
   cout << ", " << rectangle->GetPosition().GetY() << ")" << endl;
   cout << endl;

   delete circle1;
   delete circle2;
   delete rectangle;

   return 0;
}
```



```
Area of circle 1 is: 3.141592653589793
Area of circle 2 is: 12.566370614359172
Area of rectangle is: 1.0

Circle 1 is at: (1.0, 1.0)
Rectangle is at: (0.0, 1.0)

Circle 1 is at: (3.0, 1.0)
Rectangle is at: (1.0, 2.0)
```

©zyBooks 04/25/21 07:37 488201
xiang zhao
BAYLORCSI14301440Spring2021

PARTICIPATION ACTIVITY 11.6.3: Shape classes.

1) Since the Circle and Rectangle classes

both implement the ComputeArea() function, Circle and Rectangle are both abstract.

- ○ True
- ○ False

2) An instance of the ____ class cannot be created.

- ○ Shape
- ○ Point
- ○ Circle

3) The GetPosition() function of the Circle class is implemented in the _____ class.

- ○ Circle
- ○ Rectangle
- ○ Shape

4) If the Circle class omitted the ComputeArea() implementation, could Circle objects be instantiated?

- ○ Yes
- ○ No

CHALLENGE ACTIVITY 11.6.1: Abstract classes.

Start

Type the program's output

TestPerson.cpp | Person.h | Student.h | Teacher.h

```
#include "Person.h"
#include "Student.h"
#include "Teacher.h"
using namespace std;

int main() {
   Student myStudent = Student("Gus", 12, 3.4);
   Teacher myTeacher = Teacher("Byron", 33, "English");

   myStudent.printInfo();
   cout << endl;
   myTeacher.printInfo();
}
```


1

Check Try again

11.7 Is-a versus has-a relationships

The concept of inheritance is commonly confused with the idea of composition. Composition is the idea that one object may be made up of other objects, such as a MotherInfo class being made up of objects like firstName (which may be a string object), childrenData (which may be a vector of ChildInfo objects), etc. Defining that MotherInfo class does *not* involve inheritance, but rather just composing the sub-objects in the class.

Figure 11.7.1: Composition.

The 'has-a' relationship. A MotherInfo object 'has a' string object and 'has a' vector of ChildInfo objects, but no inheritance is involved.

```
class ChildInfo {
   string firstName;
   string birthDate;
   string schoolName;

   ...
};

class MotherInfo {
   string firstname;
   string birthDate;
   string spouseName;
   vector<ChildInfo> childrenData;

   ...
};
```

In contrast, a programmer may note that a mother is a kind of person, and all persons have a name and birthdate. So the programmer may decide to better organize the program by defining a PersonInfo class, and then by creating the MotherInfo class derived from PersonInfo, and likewise for the ChildInfo class.

Figure 11.7.2: Inheritance.

The 'is-a' relationship. A MotherInfo object 'is a' kind of PersonInfo. The MotherInfo class thus inherits from the PersonInfo class. Likewise for the ChildInfo class.

```
class PersonInfo {
   string firstName;
   string birthDate;

   ...
};

class ChildInfo : public PersonInfo {
   string schoolName;

   ...
};

class MotherInfo : public PersonInfo {
   string spouseName;
   vector<ChildInfo> childrenData;

   ...
};
```

PARTICIPATION ACTIVITY 11.7.1: Is-a vs. has-a relationships.

Indicate whether the relationship of the everyday items is an is-a or has-a relationship. Derived classes and inheritance are related to is-a relationships, not has-a relationships.

1) Pear / Fruit

- Is-a
- Has-a

2) House / Door

- Is-a
- Has-a

3) Dog / Owner

- Is-an
- Has-an

4) Mug / Cup

- Is-a
- Has-a

UML diagrams

Programmers commonly draw class inheritance relationships using ***Unified Modeling Language (UML)*** notation (IBM: UML basics).

PARTICIPATION ACTIVITY 11.7.2: UML derived class example: ProduceItem derived from GenericItem.

Animation content:

undefined

Animation captions:

1. A class diagram depicts a class' name, data members, and functions.
2. A solid line with a closed, unfilled arrowhead indicates a class is derived from another class.
3. The derived class only shows additional members.

11.8 UML

UML class diagrams

The ***Universal Modeling Language*** (***UML***) is a language for software design that uses different types of diagrams to visualize the structure and behavior of programs. A ***structural diagram*** visualizes static elements of software, such as the variables and functions used in the program. A ***behavioral diagram*** visualizes dynamic behavior of software, such as the flow of an algorithm.

A UML ***class diagram*** is a structural diagram that can be used to visually model the classes of a computer program, including member variables and functions.

PARTICIPATION ACTIVITY 11.8.1: UML class diagrams show class names, members, types, and access.

Animation content:

undefined

Animation captions:

1. One box exists for each class. The class name is centered at the top.
2. Class members are listed in the box below. Member variables have a name followed by a colon and the type.
3. Each member function's name and return type is listed similarly.
4. Private and public access is noted to the left of each member. A minus (-) indicates private and a plus (+) indicates public.

PARTICIPATION ACTIVITY 11.8.2: UML class diagrams.

Refer to the animation above.

1) The Square class' size member is _____.
   - public
   - private

2) The computeArea() function takes a double as a parameter.
   - True
   - False

3) Both the Circle and Square class have a member variable named center.
   - True
   - False

4) A UML class diagram is a behavioral diagram.
   - True
   - False

5) A UML class diagram describes everything that is needed to implement a class.
   - True
   - False

UML for inheritance

UML uses an arrow with a solid line and an unfilled arrow head to indicate that one class inherits from another. The arrow points toward the parent class.

UML uses italics to denote abstract classes. In particular, UML uses italics for the abstract class' name, and for each pure virtual function in the class. As a reminder, a parent class does not have to be abstract. Also, any class with one or more pure virtual functions is abstract.

PARTICIPATION ACTIVITY 11.8.3: UML uses italics for abstract classes and methods.

Animation content:

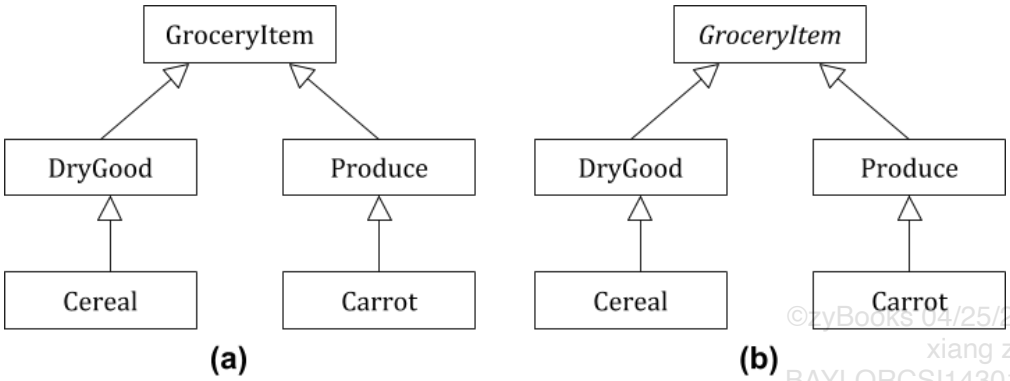
undefined

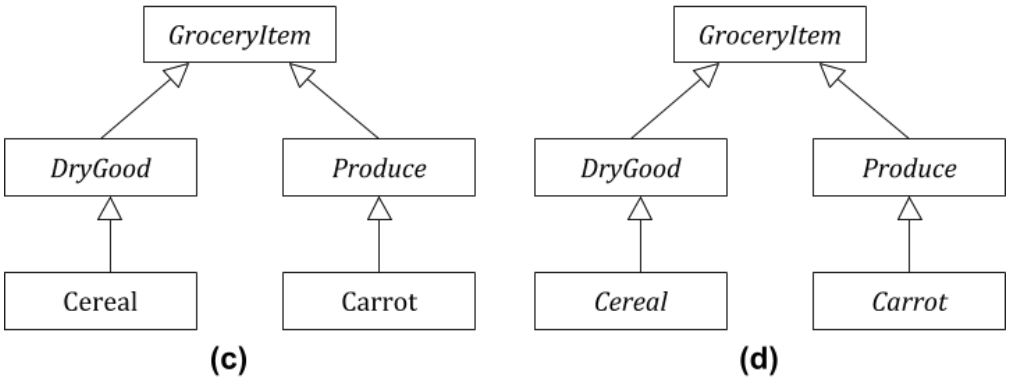
Animation captions:

1. Shape is an abstract class, so the class name and abstract method are in italics.
2. The solid-lined arrow with an unfilled arrow head indicates that the Circle class inherits from Shape.
3. Circle is a concrete class, so the class name is shown in regular font. Note that Circle implements computeArea().

PARTICIPATION ACTIVITY 11.8.4: UML for inheritance.

Match the UML diagram to the best description for that diagram. Each of the questions concerns a different implementation of a grocery store inventory system. The figures may look the same, but note the use of italics.





(c) (b) (d) (a)

	GroceryItem is abstract and requires all subclasses to implement specific methods. DryGood and Produce can be created as classes.
	GroceryItem is abstract and requires all subclasses to implement specific methods. DryGood and Produce are also abstract, as they require subclasses to implement functions specific to each class.
	All classes are concrete.
	All classes are abstract.

Reset

11.9 C++ example: Employees and overriding class functions

zyDE 11.9.1: Inheritance: Employees and overriding a class function.

The classes below describe a superclass named EmployeePerson and two derived clas EmployeeManager and EmployeeStaff, each of which extends the EmployeePerson clas main program creates objects of type EmployeeManager and EmployeeStaff and prints objects.

1. Run the program, which prints manager data only using the EmployeePerson clas printInfo function.
2. Modify the EmployeeStaff class to override the EmployeePerson class' printInfo fu and print all the fields from the EmployeeStaff class. Run the program again and v the output includes the manager and staff information.
3. Modify the EmployeeManager class to override the EmployeePerson class' printIn function and print all the fields from the EmployeeManager class. Run the progran and verify the manager and staff information is the same.

Current file: **EmployeeMain.cpp** Load default templ

```
#include <iostream>
#include "EmployeePerson.h"
#include "EmployeeManager.h"
#include "EmployeeStaff.h"
using namespace std;

int main() {
   // Create the objects
   EmployeeManager manager(25);
   EmployeeStaff staff1("Michele");

   // Load data into the objects using the Person class function
   manager.SetData("Michele", "Sales", "03-03-1975", 70000);
   staff1.SetData ("Bob",     "Sales", "02-02-1980", 50000);

   // Display the objects
   manager.PrintInfo();
   staff1.PrintInfo();
```


Run

zyDE 11.9.2: Employees and overriding a class function (solution).

Below is the solution to the problem of overriding the EmployeePerson class' printInfo() function in the EmployeeManager and EmployeeStaff classes. Note that the Main and P classes are unchanged.

Current file: **EmployeeMain.cpp** Load default templ

```
#include <iostream>
#include "EmployeePerson.h"
#include "EmployeeManager.h"
#include "EmployeeStaff.h"
using namespace std;

int main() {

   // Create the objects
   EmployeeManager manager(25);
   EmployeeStaff staff1("Michele");

   // Load data into the objects using the Person class function
   manager.SetData("Michele", "Sales", "03-03-1975", 70000);
   staff1.SetData ("Bob",     "Sales", "02-02-1980", 50000);

   // Display the objects
   manager.PrintInfo();
```

Run

11.10 C++ example: Employees using an abstract class

zyDE 11.10.1: Employees example: Abstract classes and pure virtual functions.

The classes below describe an abstract class named EmployeePerson and two derived

classes, EmployeeManager and EmployeeStaff, both of which are derived from the EmployeePerson class. The main program creates objects of type EmployeeManager a EmployeeStaff and prints them.

1. Run the program. The program prints manager and staff data using the EmployeeManager's and EmployeeStaff's PrintInfo functions. Those classes overr EmployeePerson's GetAnnualBonus() pure virtual function but simply return 0.
2. Modify the EmployeeManager and EmployeeStaff GetAnnualBonus functions to re the correct bonus rather than just returning 0. A manager's bonus is 10% of the an salary, and a staff's bonus is 7.5% of the annual salary.

Current file: **EmployeeMain.cpp** Load default templ

```
#include <iostream>
#include "EmployeeManager.h"
#include "EmployeeStaff.h"
#include <iostream>
using namespace std;

int main() {
   EmployeeManager manager(25);
   EmployeeStaff staff1("Michele");

   // Load data into the objects using the EmployeePerson class's functions
   manager.SetData("Michele", "Sales", "03-03-1975", 70000);
   staff1.SetData ("Bob",     "Sales", "02-02-1980", 50000);

   // Print the objects
   manager.PrintInfo();
   cout << "Annual bonus: " << manager.GetAnnualBonus() << endl;
```

Pre-enter any input for program, then press run.

Run

zyDE 11.10.2: Employees example: Abstract class and pure virtual functions (solution).

Below is the solution to the above problem. Note that the EmployeePerson class is unchanged.

Current file: **EmployeeMain.cpp** Load default templ

```
#include <iostream>
#include "EmployeeManager.h"
#include "EmployeeStaff.h"
#include <iostream>
using namespace std;

int main() {
   EmployeeManager manager(25);
   EmployeeStaff staff1("Michele");

   // Load data into the objects using the EmployeePerson class's functions
   manager.SetData("Michele", "Sales", "03-03-1975", 70000);
   staff1.SetData ("Bob",     "Sales", "02-02-1980", 50000);

   // Print the objects
   manager.PrintInfo();
   cout << "Annual bonus: " << manager.GetAnnualBonus() << endl;
   staff1.PrintInfo();
```


Pre-enter any input for program, then press run.

Run

11.11 LAB: Pet information (derived classes)

The base class `Pet` has private data members petName, and petAge. The derived class `Dog` extends the `Pet` class and includes a private data member for dogBreed. Complete main() to:

- create a generic pet and print information using PrintInfo().
- create a `Dog` pet, use PrintInfo() to print information, and add a statement to print the dog's breed using the GetBreed() function.

Ex. If the input is:

```
Dobby
2
Kreacher
3
German Schnauzer
```

the output is:

```
Pet Information:
   Name: Dobby
   Age: 2
Pet Information:
   Name: Kreacher
   Age: 3
   Breed: German Schnauzer
```


LAB ACTIVITY 11.11.1: LAB: Pet information (derived classes) 0 / 10

Current file: **main.cpp** Load default template...

```
1  #include <iostream>
2  #include<string>
3  #include "Dog.h"
4
5  using namespace std;
6
7  int main() {
8
9     string petName, dogName, dogBreed;
10    int petAge, dogAge;
11
12    Pet myPet;
13    Dog myDog;
14
15    getline(cin, petName);
16    cin >> petAge;
17    cin.ignore();
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.cpp** (Your program) → Outp

Program output displayed here

Signature of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


11.12 LAB: Instrument information

Given main() and the **Instrument** class, define a derived class, **StringInstrument**, for string instruments.

Ex. If the input is:

```
Drums
Zildjian
2015
2500
Guitar
Gibson
2002
1200
6
19
```

the output is:

```
Instrument Information:
   Name: Drums
   Manufacturer: Zildjian
   Year built: 2015
   Cost: 2500
Instrument Information:
   Name: Guitar
   Manufacturer: Gibson
   Year built: 2002
   Cost: 1200
   Number of strings: 6
   Number of frets: 19
```


LAB

LAB ACTIVITY | 11.12.1: LAB: Instrument information 0 / 10

File is marked as read only Current file: main.cpp

```
1  #include "StringInstrument.h"
2
3  int main() {
4        Instrument myInstrument;
5        StringInstrument myStringInstrument;
6
7        string instrumentName, manufacturerName, stringInstrumentName, stringManufacturer,
8           cost, stringYearBuilt, stringCost, numStrings, numFrets;
9
10       getline(cin, instrumentName);
11       getline(cin, manufacturerName);
12       getline(cin, yearBuilt);
13       getline(cin, cost);
14
15       getline(cin, stringInstrumentName);
16       getline(cin, stringManufacturer);
17       getline(cin, stringYearBuilt);
18       getline(cin, stringCost);
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → main.cpp (Your program) → Outp

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

11.13 LAB: Course information (derived classes)

Given main(), define a **Course** base class with functions to set and get the courseNumber and courseTitle. Also define a derived class **OfferedCourse** with functions to set and get instructorName, term, and classTime.

Ex. If the input is:


```
ECE287
Digital Systems Design
ECE387
Embedded Systems Design
Mark Patterson
Fall 2018
WF: 2-3:30 pm
```

the output is:

```
Course Information:
   Course Number: ECE287
   Course Title: Digital Systems Design
Course Information:
   Course Number: ECE387
   Course Title: Embedded Systems Design
   Instructor Name: Mark Patterson
   Term: Fall 2018
   Class Time: WF: 2-3:30 pm
```

LAB ACTIVITY | 11.13.1: LAB: Course information (derived classes) 0 / 10

File is marked as read only Current file: **main.cpp**

```
1  #include "OfferedCourse.h"
2
3  int main() {
4        Course myCourse;
5        OfferedCourse myOfferedCourse;
6
7        string courseNumber, courseTitle;
8        string oCourseNumber, oCourseTitle, instructorName, term, classTime;
9
10       getline(cin, courseNumber);
11       getline(cin, courseTitle);
12
13       getline(cin, oCourseNumber);
```

```
14      getline(cin, oCourseTitle);
15      getline(cin, instructorName);
16      getline(cin, term);
17      getline(cin, classTime);
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Outp

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

11.14 LAB: Book information (overriding member functions)

Given main() and a base **Book** class, define a derived class called **Encyclopedia**. Within the derived **Encyclopedia** class, define a PrintInfo() function that overrides the **Book** class' PrintInfo() function by printing not only the title, author, publisher, and publication date, but also the edition and number of volumes.

Ex. If the input is:

```
The Hobbit
J. R. R. Tolkien
George Allen & Unwin
21 September 1937
The Illustrated Encyclopedia of the Universe
```

```
James W. Guthrie
Watson-Guptill
2001
2nd
1
```

the output is:


```
Book Information:
   Book Title: The Hobbit
   Author: J. R. R. Tolkien
   Publisher: George Allen & Unwin
   Publication Date: 21 September 1937
Book Information:
   Book Title: The Illustrated Encyclopedia of the Universe
   Author: James W. Guthrie
   Publisher: Watson-Guptill
   Publication Date: 2001
   Edition: 2nd
   Number of Volumes: 1
```

Note: Indentations use 3 spaces.

LAB ACTIVITY | 11.14.1: LAB: Book information (overriding member functions) 0 / 10

File is marked as read only Current file: main.cpp

```
1  #include "Book.h"
2  #include "Encyclopedia.h"
3  #include <string>
4  #include <iostream>
5  using namespace std;
6
7  int main(int argc, char* argv[]) {
8     Book myBook;
9     Encyclopedia myEncyclopedia;
10
11    string title, author, publisher, publicationDate;
12    string eTitle, eAuthor, ePublisher, ePublicationDate, edition;
13    int numVolumes;
14
15    getline(cin, title);
16    getline(cin, author);
17    getline(cin, publisher);
18    getline(cin, publicationDate);
```


Develop mode | Submit mode Run your program as often as you'd like, before submitting

for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → main.cpp (Your program) → Outp

Program output displayed here

Signature of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

11.15 LAB: Plant information (vector)

Given a base `Plant` class and a derived `Flower` class, complete main() to create a vector called **myGarden**. The vector should be able to store objects that belong to the `Plant` class or the `Flower` class. Create a function called PrintVector(), that uses the PrintInfo() functions defined in the respective classes and prints each element in **myGarden**. The program should read plants or flowers from input (ending with -1), adding each `Plant` or `Flower` to the **myGarden** vector, and output each element in **myGarden** using the PrintInfo() function.

Ex. If the input is:

```
plant Spirea 10
flower Hydrangea 30 false lilac
flower Rose 6 false white
plant Mint 4
-1
```


the output is:

```
Plant Information:
   Plant name: Spirea
```

```
   Cost: 10

Plant Information:
   Plant name: Hydrengea
   Cost: 30
   Annual: false
   Color of flowers: lilac

Plant Information:
   Plant name: Rose
   Cost: 6
   Annual: false
   Color of flowers: white

Plant Information:
   Plant name: Mint
   Cost: 4
```


LAB ACTIVITY | 11.15.1: LAB: Plant information (vector) | 0 / 10

Current file: **main.cpp** Load default template...

```
1  #include "Plant.h"
2  #include "Flower.h"
3  #include <vector>
4  #include <string>
5  #include <iostream>
6
7  using namespace std;
8
9  // TODO: Define a PrintVector function that prints an vector of plant (or flower) object
10
11 int main(int argc, char* argv[]) {
12    // TODO: Declare a vector called myGarden that can hold object of type plant pointer
13
14    // TODO: Declare variables - plantName, plantCost, flowerName, flowerCost,
15    //       colorOfFlowers, isAnnual
16    string input;
17
18    cin >> input;
```

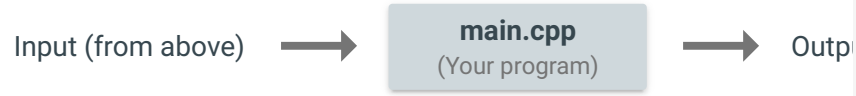

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Signature of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

11.16 LAB: Library book sorting

Two sorted lists have been created, one implemented using a linked list (**`LinkedListLibrary linkedListLibrary`**) and the other implemented using the built-in Vector class (**`VectorLibrary vectorLibrary`**). Each list contains 100 books (title, ISBN number, author), sorted in ascending order by ISBN number.

Complete main() by inserting a new book into each list using the respective **`LinkedListLibrary`** and **`VectorLibrary`** InsertSorted() methods and outputting the number of operations the computer must perform to insert the new book. Each InsertSorted() returns the number of operations the computer performs.

Ex: If the input is:

```
The Catcher in the Rye
9787543321724
J.D. Salinger
```

the output is:


```
Number of linked list operations: 1
Number of vector operations: 1
```

Which list do you think will require the most operations? Why?

LAB 11.16.1: LAB: Library book sorting

ACTIVITY | 11.10.1: LAB: Library book sorting

0 / 10

Current file: main.cpp ▾ Load default template...

```
1  #include "LinkedListLibrary.h"
2  #include "VectorLibrary.h"
3  #include "BookNode.h"
4  #include "Book.h"
5  #include <fstream>
6  #include <iostream>
7  using namespace std;
8
9  void FillLibraries(LinkedListLibrary &linkedListLibrary, VectorLibrary &vectorLibrary) {
10    ifstream inputFS; // File input stream
11    int linkedListOperations = 0;
12    int vectorOperations = 0;
13
14    BookNode* currNode;
15    Book tempBook;
16
17    string bookTitle;
18    string bookAuthor;
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Outp

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

11.17 Lab 4 - The Big Four, Inheritance, and Dynamically Resizing Arrays

Don't worry about matching the output completely. zyBooks requires an output test. Do try to test the same way.

In class we have just completed a discussion of the big four and some inheritance. Hopefully each of you now understand why you were required to implement the big 4 and some of the benefits of inheritance. In this lab we are going to inherit from a class and implement all of the members of the big four along with dynamically resizing data.

IntArray

I have provided a class named LabArray in the file lab4-LabArray.h. Your class named IntArray must inherit from the LabArray class. I have used some inheritance ideas described in section 11.6. Don't worry if you haven't read that section yet. From that section, you only need to understand that the functions described in the LabArray class must be implemented (defined) in your IntArray class as long as it inherits for the LabArray class. Don't forget that you do not need to re-define the private members. Inheritance will give you data, size, and capacity. Also, do not repeat the assignment to the functions (= 0;) in your definition of the IntArray class. Your class IntArray should be defined in lab4-IntArray.h and each funtion should be implemented in lab4-IntArray.cpp.

Test Driver

Then, develop your own test driver (lab4-testmain.cpp) to test your code. Make sure that you test the following circumstances in your test driver.

- instantiate at least two instances of an IntArray
- instantiate one IntArray with another IntArray (after you have inserted values onto one array)
- push values onto your IntArray (create and use a data file - write a quick program to produce the file)
- write a function that passes an IntArray by value
- assigns an IntArray to another IntArray (after you have pushed values onto both arrays)
- test each method individually, if possible

Dynamic Memory Requirements

Your IntArray needs to adhere to these two memory requirements. It should start with enough space to store 5 integers. And, it should increase by 5 integers when more space is needed.


```
class LabArray {
  protected:
   int *data;
   int size, capacity;

  public:
```

```
    virtual int getVal(int) = 0;
    virtual int getCapacity() = 0;
    virtual int getSize() = 0;
    virtual void insertVal(int) = 0;
    virtual int removeVal() = 0;
};
```

A little help with lab4-IntArray.h


```
class IntArray /* need to add inheritance stuff here */ {
  public:
  IntArray();
  // other members of the Big 4

  int getVal(int);  // notice loss of virtual and = 0
}
```

LAB ACTIVITY | 11.17.1: Lab 4 - The Big Four, Inheritance, and Dynamically Resizing Arrays 0 / 1

Current file: **lab4-IntArray.cpp** Load default template...

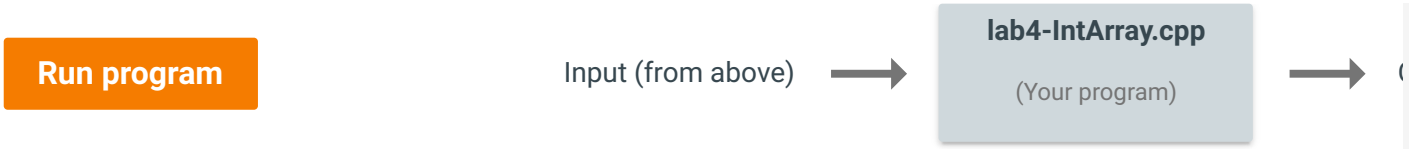
```
1 #include "lab4-IntArray.h"
2
3 IntArray::IntArray() {
4    // implement the default constructor
5 }
6
7 int IntArray::getVal(int ndx) {
8    // implement getVal
9 }
```


Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.



Program output displayed here

Signature of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

11.20 Lab 5 - Doggies

CSI 1440

Lab 5

"Doggies”

This week’s lab will call to your attention the benefits of inheritance and polymorphism using the Dog class. From the most basic viewpoint, inheritance allows us to reuse the code of the base class. We will see from the following discussion, potentially, how much code reuse can reduce the amount of work that is required. In addition, we will see how polymorphism will allow us to present a single public interface to our users yet provide unique actions for each derived class. This will allow us to write more generic code in our driver while allowing unique interactions with our different derived types.

You will need to write each one of the files on your local computer for testing. At the end of the lab, you can upload all the files you have created in order to submit your work. Don't worry about the grade assigned by zyBooks. Your TA or I will grade your lab submission and record the proper grade in Canvas.

Code Reuse Introduction

Let’s assume that we are tasked with writing a software package that will be used to store information for the Westminster Dog Show. The show has dogs broken up into seven different groups and contestants are graded based on not only characteristics of these seven different groups, but are also graded based on specific breed characteristics.

For us to accurately store information pertaining to the contestants, we would most likely want to create classes to store information about the contestants. Now consider that 63 different breeds are represented each year which would require us to create 63 different classes to store all of our

contestant data. Obviously, each of these classes would need to store information about the contestant like height, weight, name, etc…

```
class GermanShepard {
private:
  string name;
  double height, weight;
  ...

class Dachshund {
private:
  string name;
  double height, weight;
  ...

class Chihuahua {
private:
  string name;
  double height, weight;
  ...
```


At this point you should recognize that each of the classes will store the same information as the next class. So, 63 classes will store the same information like height, weight, name, etc… Seems like a lot of work being repeated (You should also recognize, in general, computer scientists don't like doing extra work just for work's sake). There has to be an easier way to get this done.

There is. It's called inheritance. With inheritance, we can create a general Dog class to store all the identical information between all the different breeds, like height, weight, name, etc… using inheritance to take advantage of code reuse. But, it also allows us to store additional unique information for a particular breed. (I'm not a huge dog person. So, I'm making this stuff up.) Like beard length for terriers, or strong webbing between toes of retrievers.

```
class Dog {
private:
  string name;
  double height, weight;
  ...
};


class ScottishTerrier : public Dog {
private:
  double beardLength;
  ...
};
```

Code Reuse in Action

Let's leave the Westminster stuff behind and start writing some simple examples. We are going to stick with the concept of a dog, but we're not going to worry about as many breed characteristics for brevity's sake. We are going to focus on the sound of the dog's bark (At least, an onomatopoeia will be used. We may even make up a few new ones along the way). Type in the following code into **Dogs1.h**. Write a driver, **Driver1.cpp**, that instantiates an instance of each class and tests the object's bark function.

Dogs1.h

```
#include <iostream>
#include <string>

using namespace std;

class Dog {
private:
  string name;

public:
  Dog( string n ) {
    name = n;
  }

  string getName() {
    return name;
  }

  void Bark() {
    cout << "Bow-Wow" << endl;
  }
};

class BostonTerrier {
private:
  string name;

public:
  BostonTerrier( string n ) {
    name = n;
  }

  string getName() {
    return name;
```

```
  }

  void Bark() {
    cout << "Arf-Arf" << endl;
  }
};

class Heinz57 {
private:
  string name;

public:
  Heinz57( string n ) {
    name = n;
  }

  string getName() {
    return name;
  }

  void Bark() {
    cout << "Bow-Wow" << endl;
  }
};
```


Driver1.cpp

```
#include "Dogs1.h"

int main(){
  Dog aDog("Fido");
  cout << aDog.getName() << " says ";
  aDog.Bark();

  BostonTerrier aBoston("Dixie");
  cout << aBoston.getName() << " says ";
  aBoston.Bark();

  Heinz57 aHeinz("Bones");
  cout << aHeinz.getName() << " says ";
  aHeinz.Bark();

  return 0;
}
```

You will need to test and submit both **Dogs1.h** and **Driver1.cpp** to zyBooks. Make sure you include file heading and all other comments for each file.

As was illustrated in the Westminster example, this example is a little silly. It too repeats the same information. And, we can simplify it using inheritance. Type in the next set of code using inheritance for code reuse. Name the new file **Dogs2.h**. A completely new driver is not necessary, but you will need to include the new **Dogs2.h** file and submit the "new" driver as **Driver2.cpp**.


```
class BostonTerrier : public Dog {
public:
  BostonTerrier( string n ) : Dog( n ) { }
};

class Heinz57 : public Dog {
public:
  Heinz57( string n ) : Dog( n ) { }
};
```

Submit and test **Dogs2.h** and **Driver2.cpp**. Make sure you include file heading and all other comments. Add your changes in the file comments.

We are now using inheritance, but we may have lost the ability to have breed specific barks. Obviously, we didn't specify the breed characteristic bark in the new .h file. So, each of your different dog instnaces should have said "Bow-Wow".

We will solve this problem by overriding the Bark method in the BostonTerrier class. Type in the next set of code overriding the Bark method. Just like the previous changes, a completely new driver should not be necessary just include the new header file. Name the new files **Dogs3.h** and **Driver3.cpp**.

```
class BostonTerrier : public Dog {
public:
  BostonTerrier( string n ) : Dog( n ) { }

  void Bark() {
    cout << "Arf-Arf" << endl;
  }
};
```


Submit and test **Dogs3.h** and **Driver3.cpp**. Make sure you include file heading and all other comments. Add your changes in the file comments.

Polymorphism Introduction

Let's add some more functionality to our driver and our objects. We need to allow our dogs to be walked and allow our dogs to beg for food. Add the following functionalities to our Dog class and to our driver. First, make some additions to the Dog class. Make sure to include the changes in your change log in your file comments.

```
class Dog {
private:
  string name;

public:
  Dog( string n ) {
    name = n;
  }

  string getName() {
    return name;
  }

  void Bark() {
    cout << "Bow-Wow" << endl;
  }

  void Wimper() {
    cout << "Rrrrrrrrr" << endl;
  }

 void Pant() {
    cout << "Ah-ha-ha-ha-ha" << endl;
  }
};
```


Then add the following functions in your driver. Make sure to call the functions with each object.

```
void WalkTheDog( Dog &aDog ) {
  aDog.Pant();
  aDog.Bark();
  aDog.Pant();
}

void BegForFood( Dog &aDog ) {
  aDog.Wimper();
  aDog.Bark();
}
```


Submit and test out both **Dogs4.h** and **Driver4.cpp**. Make sure you include file heading and all other comments.

Notice that our Boston Terrier doesn't act like a Boston Terrier within the two new functions. It would be nice if a derived class would always act like the derived class even when passed around as a reference to the base class.

Polymorphism in Action

Hopefully, you can see the desire for a single public interface for all our classes. A single public interface will allow us to write generic code for all our breeds like the functions WalkTheDog() and BegForFood() - functions that can be defined once with the base class in mind and used for all our different derived classes.

We can get this behavior by making the Bark method a virtual method. Declaring the Bark method as virtual changes the behavior by allowing for dynamic binding of a method instead of static binding. Dynamic binding allows the method to be chosen at runtime instead of at compile-time. Virtual methods allow us to write Polymorphic code.

Add the keyword virtual immediately before the return type of the Bark method in the Dog class to allow for polymorphic behavior.

```
class Dog {
private:
  string name;

public:
  Dog( string n ) {
    name = n;
  }

  string getName() {
    return name;
  }

  virtual void Bark() {
    cout << "Bow-Wow" << endl;
  }

  void Wimper() {
    cout << "Rrrrrrrrr" << endl;
  }

 void Pant() {
    cout << "Ah-ha-ha-ha-ha" << endl;
  }
};
```


Afer adding the virtual keyword, submit and test out both Dogs5.h and Driver5.cpp.

Now let's add a few more breeds to see if things continue to work the way we currently have them working. Add the following classes and override the Bark method to print the bark onomatopoeia.

```
1. HuckleberryHound - "Oh, my darlin"
2. Poodle - "Oui-Oui"
3. ScoobyDoo - "Ruh-Roh"
4. GermanSheperd - "I see nothing!"
```

Test your new classes like you did with the BostonTerrier and Heinz57 classes.

Submit and test out both **Dogs6.h** and **Driver6.cpp**.

Dyanmic Dog instances

The driver is getting a little long. We are going to shorten the driver using dynamic memory by placing all the differnent class instances into a single array of Dog pointers.

```
int main() {
  Dog *doggies[] = { new Dog("Fido"), new BostonTerrier("Dixie"), new
Heinz57("Bones"),
            new HuckleberryHound("Huck"), new Poodle("Frenchy"),
            new ScoobyDoo("Scooby"), new GermanSheperd("SgtShultz") };

  for( int i = 0; i < 7; i++ ) {
    cout << "Walking " << doggies[i]->getName() << "..." << endl;
    WalkTheDog( *doggies[i] );

    cout << endl << doggies[i]->getName() << " is begging again..." <<
endl;

    BegForFood( *doggies[i] );
    cout << endl;
  }

  return 0;
}
```

Submit and test your **Dogs7.h** and **Driver7.cpp**. Make sure you include all the proper comments.

The new driver code that follows doesn't show it, but each of you should see how you could create a dynamically resizing pointer array (Yes, a Dog**. I know it's scary looking). This dynamic array of pointers could have the different breeds, held by a single reference - doggies in our instance, specified in a file. And, it would allow us to build our dog show more easily than we could without inheritance and polymorphism. I'm not going to ask you to write a driver like this, but I do encourage each of you to think about how this could be done. Design of such a system can become overwhelming quickly. Tools used in software engineering can be helpful in communicating your ideas as a designer to a programmer. We've learned on one such tool already - UML's Class Diagrams.

LAB

11.20.1: Lab 5 - Doggies

ACTIVITY 11.20.1: Lab 6 - Doggies 0 / 1

Current file: **Dogs1.h**

```
1
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **Dogs1.h** (Your program) → Outp

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.