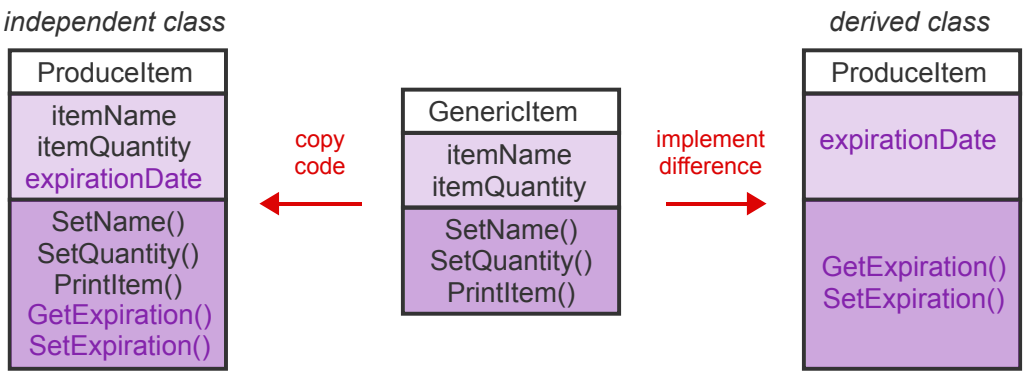
13.1 Derived classes

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 13.1.1: Creating a ProduceItem from GenericItem.



Animation content:

Static figure:
Class GenericItem and class ProduceItem are displayed. Class GenericItem has two data members and three member functions. Class ProduceItem has all of GenericItem's data members, as well as an additional data member that GenericItem does not have. Class ProduceItem also has all of GenericItem's member functions, as well as two additional member functions that GenericItem does not have.

Step 1: A GenericItem has data members itemName and itemQuantity and 3 member functions.

Step 2: A ProduceItem is very similar to a GenericItem, but a ProduceItem also needs an expirationDate data member.

Step 3: A ProduceItem needs the same data member functions as a GenericItem plus functions to get and set the expiration date. Member functions GetExpiration() and SetExpiration() are added only to ProduceItem.

Step 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.

Step 5: If ProduceItem is implemented as a derived class, ProduceItem need only implement what is different between a GenericItem and ProduceItem. As a derived class, ProduceItem needs to only declare data member expirationDate and only define member functions GetExpiration() and SetExpiration().

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 13.1.2: 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 13.1.1: Class ProduceItem is derived from class GenericItem.

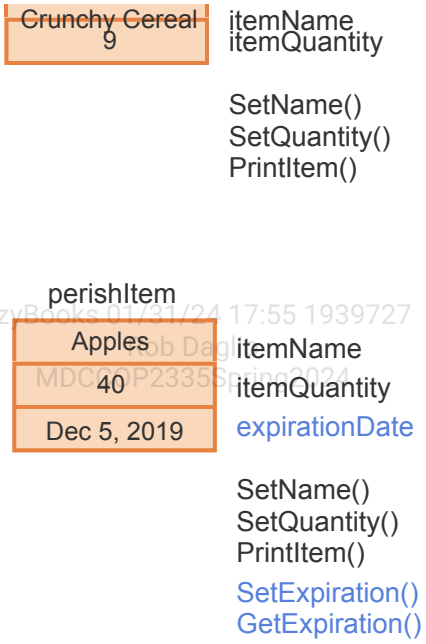

```
// Base class
class GenericItem {
   public:
      void SetName(string newName) {
          itemName = newName;
      }

      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 13.1.3: Using GenericItem and ProduceItem objects.

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

miscItem

```
using namespace std;

// See figure above for class details
class GenericItem { ... };
class ProduceItem : public GenericItem { ... };

int main() {
   GenericItem miscItem;
   ProduceItem perishItem;

   miscItem.SetName("Crunchy Cereal");
   miscItem.SetQuantity(9);
   miscItem.PrintItem();

   perishItem.SetName("Apples");
   perishItem.SetQuantity(40);
   perishItem.SetExpiration("Dec 5, 2019");
   perishItem.PrintItem();
   cout << "  (Expires: " << perishItem.GetExpiration()
        << ")" << endl;

   return 0;
}
```




```
Crunchy Cereal 9
Apples 40
  (Expires: Dec 5, 2019)
```


Animation content:

Static figure:
Begin C++ code:
#include <iostream>
#include <string>
using namespace std;

// See figure above for class details
class GenericItem { ... };
class ProduceItem : public GenericItem { ... };

int main() {
   GenericItem miscItem;
   ProduceItem perishItem;

   miscItem.SetName("Crunchy Cereal");
   miscItem.SetQuantity(9);
   miscItem.PrintItem();

   perishItem.SetName("Apples");
   perishItem.SetQuantity(40);

```
   perishItem.SetExpiration("Dec 5, 2019");
   perishItem.PrintItem();
   cout << "  (Expires: " << perishItem.GetExpiration() << ")" << endl;

   return 0;
}
```

End C++ code.

GenericItem object miscItem and ProduceItem object perishItem are displayed.

Step 1: The base class GenericItem and derived class ProduceItem are declared. Statement class GenericItem { ... }; declares base class GenericItem and statement class ProduceItem : public GenericItem { ... }; delares derived class ProduceItem.

Step 2: miscItem is a GenericItem, which has 2 data members and 3 member functions. GenericItem miscItem has itemName and itemQuantity as data members. miscItem also has SetName(), SetQuantity(), and PrintItem() as member functions.

Step 3: Since ProduceItem is derived from GenericItem, ProduceItem inherits GenericItem's members and adds new members implemented in ProduceItem. ProduceItem perishItem inherits data members itemName and itemQuantity from GenericItem, but has expirationDate as an additional data member. perishItem also inherits member functions SetName(), SetQuantity(), and PrintItem() from GenericItem, but has SetExpiration() and GetExpiration() as additional member functions.

Step 4: miscItem's name and quantity are set, and miscItem is printed to the screen. miscItem's itemName is set to "Crunchy Cereal" and miscItem's itemQuantity is set to 9.

Step 5: perishItem's name, quantity, and expiration are set. Then perishItem is printed to the screen. perishItem's itemName is set to "Apples", perishItem's itemQuantity is set to 40, and perishItem's expirationDate is set to "Dec 5, 2019".

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 13.1.4: 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:

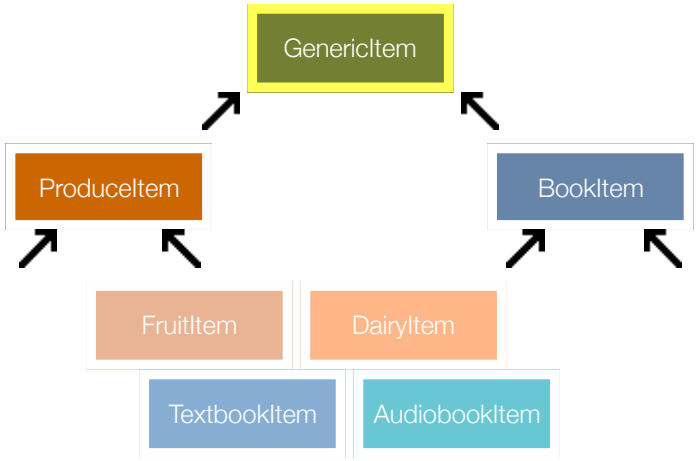
- A derived class can serve as a base class for another class. Ex: `class FruitItem: public ProduceItem {...}` creates a derived class FruitItem from ProduceItem, which was derived from GenericItem.
- A class can serve as a base class for multiple derived classes. Ex: `class FrozenFoodItem: public GenericItem {...}` 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 13.1.5: 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 13.1.6: 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 13.1.2: 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
```

PARTICIPATION ACTIVITY 13.1.7: 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 { ... };`

©zyBooks 01/31/24 17:55 1939727
Rob Daglio
MDCCOP2335Spring2024

Exploring further:

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

CHALLENGE ACTIVITY 13.1.1: Derived classes.

539740.3879454.qx3zqy7

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 << "Speed: ";
         PrintSpeed();
      }
};

int main() {
   Car myCar;
   myCar.SetSpeed(50);

   myCar.PrintCarSpeed();

   return 0;
}
```

```
Speed: 50
```

1 2 3

Check Next

CHALLENGE ACTIVITY 13.1.2: 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 courseStudent's data. End with a

newline. Sample output from the given program:

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

Learn how our autograder works

539740.3879454.qx3zqy7

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Person {
6     public:
7        void SetName(string userName) {
8           lastName  = userName;
9        }
10
11       void SetAge(int numYears) {
12          ageYears = numYears;
13       }
14
15       // Other parts omitted
```

Run

13.2 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 13.2.1: 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 13.2.1: 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 13.2.2: 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;
}
```


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 13.2.1: 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 13.2.2: 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 13.2.3: 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

13.3 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 13.3.1: Overriding member function example.

```
class Business {
   public:
      ...
      string GetDescription() const {
         return name + " -- " + address;
      }

   protected:
      string name;
      string address;
};

class Restaurant : public Business {
   public:
      ...
      string GetDescription() const {
         return name + " -- " + address +
            "\n  Rating: " + to_string(rating);
      }

   private:
      int rating;
};

int main() {
   Restaurant favoritePlace;
   favoritePlace.SetName("Friends Cafe");
   favoritePlace.SetAddress("500 W 2nd Ave");
   favoritePlace.SetRating(5);
   cout << favoritePlace.GetDescription() << endl;

   return 0;
}
```

same

```
Friends Cafe -- 500 W 2nd Ave
  Rating: 5
```

Animation content:

Static figure:
Begin C++ code:

```
class Business {
  public:
    ...
    string GetDescription() const {
      return name + " -- " + address;
    }

  protected:
    string name;
    string address;
};

class Restaurant : public Business {
  public:
    ...
    string GetDescription() const {
      return name + " -- " + address +
        "\n  Rating: " + to_string(rating);
    }

  private:
    int rating;
};

int main() {
  Restaurant favoritePlace;
  favoritePlace.SetName("Friends Cafe");
  favoritePlace.SetAddress("500 W 2nd Ave");
  favoritePlace.SetRating(5);
  cout << favoritePlace.GetDescription() << endl;

  return 0;
}
```

End C++ code.

©zyBooks 01/31/24 17:55 1939727
Rob Daglio
MDCCOP2335Spring2024

Step 1: The Business class defines a GetDescription() member function that returns a string with the business name and address. The output console is empty.

Step 2: Restaurant derives from Business and defines a member function with the same name, return

type, and parameters as the base class function GetDescription(). The return statement for the Business class is name + " -- " + address. The return statement for the Restaurant class is the same as the Business class, except "\n Rating: " + to_string(rating) is concatenated to the variable address.

Step 3: The Restaurant object favoritePlace calls the Restaurant's GetDescription(), which overrides the base class's GetDescription(). The output console now contains two lines of output:
Friends Cafe -- 500 W 2nd Ave
  Rating: 5

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, number of parameters, or return values. In overriding, a derived class member function must have the same parameter types, number of parameters, and return value as the base class member function with the same name. Overloading is performed if derived and base member functions have different parameter types; the member function of the derived class does not hide the member function of the base class.

PARTICIPATION ACTIVITY 13.3.2: 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 13.3.1: 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 13.3.3: 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()

CHALLENGE ACTIVITY 13.3.1: Overriding member function.

539740.3879454.qx3zqy7

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 << "Internet: " << internet << endl;
      cout << "CPU: " << cpuUsage << 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("20%", "connected");
   myLaptop.SetWiFiStatus("bad");

   myLaptop.PrintStatus();

   return 0;
}
```



```
WiFi: ba
CPU: 20%
```

1 2

Check Next

CHALLENGE ACTIVITY 13.3.2: 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's PrintAll() function.

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

Learn how our autograder works

539740.3879454.qx3zqy7

```
#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() {
```


Run

13.4 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 13.4.1: Compile-time polymorphism vs. runtime polymorphism.

compile-time polymorphism

```
void DriveTo(string restaurant) {
   cout << "Driving to " << restaurant << endl;
}

void DriveTo(Restaurant restaurant) {
   cout << "Driving to " << restaurant.GetDescription() << endl;
}

int main() {
   DriveTo("Big Mac's");  // Call string version
}
```


runtime polymorphism

```
void DriveTo(Business* businessPtr) {
   cout << "Driving to " << businessPtr->GetDescription() << endl;
}

int main() {
   int index;
   vector<Business*> businessList;
   Business* businessPtr;
   Restaurant* restaurantPtr;
   ...
   businessList.push_back(businessPtr);
   businessList.push_back(restaurantPtr);

   index = rand() % businessList.size();
   DriveTo(businessList.at(index));
}
```

Calls Restaurant's GetDescription() for Restaurant pointer

Animation content:

Static figure:
compile-time polymorphism
Begin C++ code:

```
void DriveTo(string restaurant) {
   cout << "Driving to " << restaurant << endl;
}

void DriveTo(Restaurant restaurant) {
   cout << "Driving to " << restaurant.GetDescription() << endl;
}

int main() {
   DriveTo("Big Mac's");  // Call string version
```

```
}
End C++ code.
runtime polymorphism
Begin C++ code:
void DriveTo(Business* businessPtr) {
   cout << "Driving to " << businessPtr->GetDescription() << endl;
}

int main() {
   int index;
   vector<Business*> businessList;
   Business* businessPtr;
   Restaurant* restaurantPtr;
   ...
   businessList.push_back(businessPtr);
   businessList.push_back(restaurantPtr);

   index = rand() % businessList.size();
   DriveTo(businessList.at(index));
}
End C++ code.
```


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 13.4.2: 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 13.4.3: Adding a virtual function to the base class.

```
class Business {
public:
   virtual string GetDescription() const {

   }
   ...
};

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

int main() {
   Business* businessPtr;
   Restaurant favoriteCafe;
   favoriteCafe.SetName("Friends Cafe");
   favoriteCafe.SetAddress("500 2nd Ave");
   favoriteCafe.SetRating(5);

   // Point to favoriteCafe
   businessPtr = &favoriteCafe;

   cout << businessPtr->GetDescription();
}
```

No virtual function

Friends Cafe -- 500 2nd Ave

With virtual function

Friends Cafe -- 500 2nd Ave
Rating: 5

Animation content:

Static figure:
Begin C++ code:
class Business {
public:
   string GetDescription() const {

   }
   ...

```
};

class Restaurant : public Business {
public:
   string GetDescription() const {
      return name + " -- " + address + "\n  Rating: " + to_string(rating);
   }
   ...
};

int main() {
   Business* businessPtr;
   Restaurant favoriteCafe;
   favoriteCafe.SetName("Friends Cafe");
   favoriteCafe.SetAddress("500 2nd Ave");
   favoriteCafe.SetRating(5);

   // Point to favoriteCafe
   businessPtr = &favoriteCafe;

   cout << businessPtr->GetDescription();
}
End C++ code.
```


Step 1: Restaurant overrides GetDescription() from the base class Business without using the "override" keyword.

Step 2: businessPtr points to favoriteCafe. businessPtr->GetDesciption() calls Business GetDescription() instead of Restaurant GetDescription() because runtime polymorphism does not work without virtual functions. A monitor labeled "No virtual function" with the text "Friends Cafe -- 500 2nd Ave" is displayed.

Step 3: The "virtual" keyword makes GetDescription() a virtual function. The "override" keyword indicates that Restaurant GetDescription() overrides the base class GetDescription(). The "virtual" keyword is prepended to base class GetDescription(), virtual string GetDescription() const. The "override" keyword is appended to Restaurant GetDescription(), string GetDescription() const override.

Step 4: Running the program with an overridden virtual function causes runtime polymorphism to call Restaurant GetDescription(). Another monitor labeled "With virtual function" with the text "Friends Cafe -- 500 2nd Ave\n Rating: 5" is displayed.

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 13.4.1: 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);
```


©zyBooks 01/31/24 17:55 1939727
Rob Daglio
MDCCOP2335Spring2024

```
   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
```


PARTICIPATION ACTIVITY 13.4.4: 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 13.4.2: 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 13.4.5: 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 13.4.1: Polymorphism and virtual member functions.

539740.3879454.qx3zqy7

Start

Type the program's output

©zyBooks 01/31/24 17:55 1939727
Rob Daglio
MDCCOP2335Spring2024

```
#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;
   SmartWatch* watch3;

   vector<Watch*> watchList;
   unsigned int i;

   watch1 = new Watch();
   watch1->SetHours(6);
   watch1->SetMins(46);

   watch2 = new SmartWatch();
   watch2->SetHours(11);
   watch2->SetMins(38);
   watch2->SetPercentage(59);

   watch3 = new SmartWatch();
   watch3->SetHours(12);
   watch3->SetMins(32);
   watch3->SetPercentage(51);

   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;
}
```



```
11:38
12:32
6:46
```

1 2

Check Next

CHALLENGE ACTIVITY 13.4.2: 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 virtual to make PrintItem() a virtual function.

Learn how our autograder works

539740.3879454.qx3zqy7

```
#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  */

```

Run

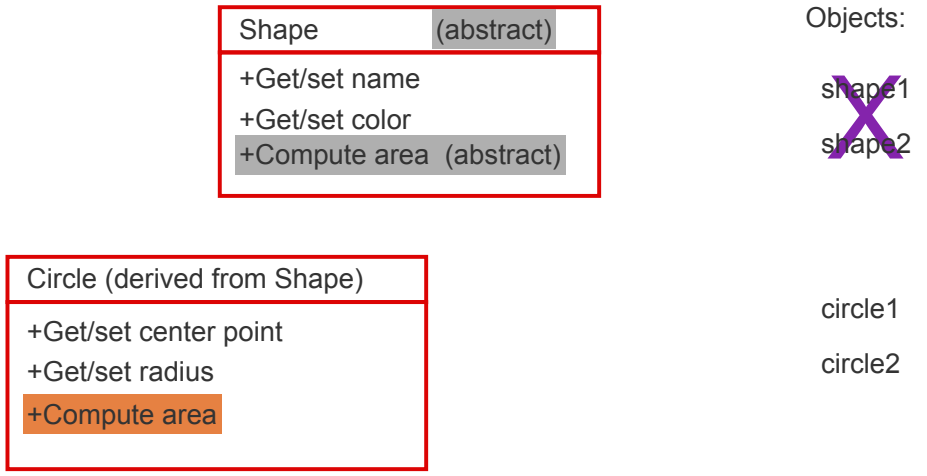
13.5 Abstract classes: Introduction (generic)

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 13.5.1: Classes, inheritance, and abstract classes.



Animation content:

Static figure:
Two boxes representing classes are displayed. The first box is labeled Shape. The text (abstract) is highlighted grey and is present adjacent to the label Shape. A box containing a list of text, +Get/set name, +Get/set color, and +Compute area (abstract) is connected to the box labeled Shape. The text +Compute area (abstract) is highlighted grey. The second box is labeled Circle (derived from Shape). A box containing a list of text, +Get/set center point, +Get/set radius, and +Compute area is connected to the box labeled Circle (derived from Shape). The text +Compute area is highlighted orange. The text Objects:, shape1, and shape2 is displayed next to the box labeled Shape. A large purple x covers the text shape1 and shape2. The text circle1 and circle2 are displayed next to the box labeled Circle (derived from Shape).
Step 1: A class provides data/behaviors for objects.
The box labeled Shape with the list of data/behavior of the Shape class, +Get/set name, +get/set color, and the text Objects: are displayed. The newly created Shape objects listed as shape1 and shape2 appear.

Step 2: Inheritance creates a Circle class that implements behaviors specific to a circle.

The box labeled Circle (derived from Shape) appears along with the list of data/behavior of the Circle class, +Get/set center point, +Get/set radius appear. The newly created Circle objects listed as circle1 and circle2 appear.

Step 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.
The text (abstract) is added next to the Shape class label, highlighted grey. The new behavior, +Compute area (abstract), is added to the list of data/behavior of the Shape class. +Compute area (abstract) is highlighted grey. A large purple x appears over the objects shape1 and shape2, indicating that the Shape object cannot be created.

Step 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.
The new behavior +Compute area is added to the list of data/behavior of the Circle class. +Compute area is highlighted orange.

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 13.5.2: 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

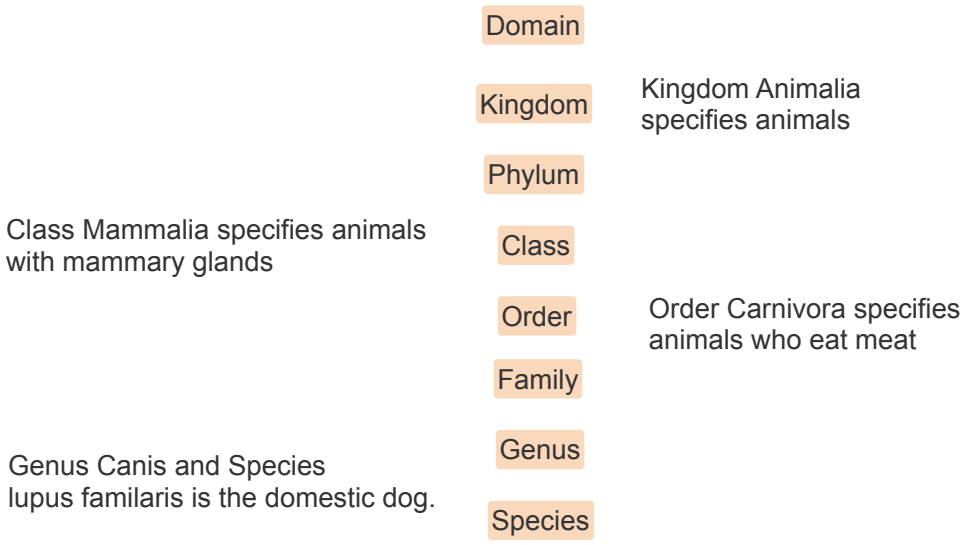
4) The Circle class must implement the ComputeArea() function.

- 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 13.5.3: Biological classification uses abstract classes.



Animation content:

Static Figure:
Eight boxes, each with a different level of biological hierarchy, in the following order: Domain, Kingdom, Phylum, Class, Order, Family, Genus, Species.

The static figure has the following text:

Kingdom Animalia specifies animals
Class Mammalia specifies animals with mammary glands
Order Carnivora specifies animals who eat meat
Genus Canis and Species lupus familaris is the domestic dog.
The hierarchy of biological classification is an example of abstract classes

Step 1: Each level of the biological hierarchy specifies behaviors common to that level.
The level Domain appears.
The level Kingdom appears. The text appears: Kingdom Animalia specifies animals.
The level Phylum appears.
The level Class appears. The text appears: Class Mammalia specifies animals with mammary glands.

Step 2: At this level of the hierarchy, a lot of behavior for the organism is known but the organism is not yet specified.
The level Order appears. The text appears: Order Carnivora specifies animals who eat meat.
The level Family appears.

Step 3: At the final level (species), the organism can be fully described, just as a concrete class can be fully instantiated.
The level Genus appears.
The level Species appears. The text appears: Genus Canis and Species lupus familaris is the domestic dog.

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 13.5.4: 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

13.6 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 13.6.1: A Shape class with a pure virtual function is an abstract 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);
      }
};

int main(int argc, const char* argv[]) {
   Shape* shape = new Shape();
   ...
}
```

Error

Animation content:

Static figure:
Begin C++ code:

```
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);
    }
};

int main(int argc, const char* argv[]) {
  Shape* shape = new Shape();
  ...
}
```

End C++ code.

Step 1: The Shape class has the pure virtual ComputeArea() function. The Shape class is abstract due to having a pure virtual function. The line of code, virtual double ComputeArea() const = 0;, is highlighted.

Step 2: An abstract class cannot be instantiated. The line of code, Shape* shape = new Shape();, is highlighted and labeled as "Error".

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 13.6.2: Shape class.

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

2) The Shape class defines and provides code for non-pure-virtual functions.
   - 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 13.6.1: 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 class

```
#ifndef SHAPE_H
#define SHAPE_H
#include "Point.h"

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);
      }
};

#endif
```

Point.h
A Point object holds the x, y coordinates for a point

```
#ifndef POINT_H
#define POINT_H

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;
      }
};

#endif
```

Circle.h

Defines a Circle class

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include <cmath>
#include "Shape.h"

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));
      }
};

#endif
```

Rectangle.h

Defines a Rectangle class

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

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);
      }
};

#endif
```


TestShapes.cpp

Implements the main function to test the Shape classes

```
#include <iostream>
#include "Circle.h"
#include "Rectangle.h"
using namespace std;

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: 3.14159
Area of circle 2 is: 12.5664
Area of rectangle is: 1

Circle 1 is at: (1, 1)
Rectangle is at: (0, 1)

Circle 1 is at: (3, 1)
Rectangle is at: (1, 2)
```

©zyBooks 01/31/24 17:55 1939727
Rob Daglio
MDCCOP2335Spring2024

PARTICIPATION ACTIVITY 13.6.3: 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 13.6.1: Abstract classes.

539740.3879454.qx3zqy7

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("Mike", 11, 2.3);
   Teacher myTeacher = Teacher("Orwell", 25, "English");

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

```
Mike, 1
GPA: 2.

Orwell,
Subject
```


1

Check Try again

13.7 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 13.7.1: 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 13.7.2: 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 13.7.1: 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

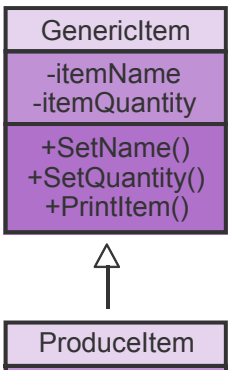
3) Dog / Owner
   - Is-an
   - Has-an

4) Mug / Cup
   - Is-a
   - Has-a

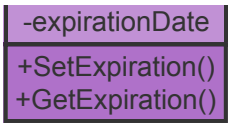
UML diagrams

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

PARTICIPATION ACTIVITY 13.7.2: UML derived class example: ProduceItem derived from GenericItem.



Member access
- means private
+ means public
# means protected



Animation content:

Static Figure:
Two class diagrams are shown.
The first class diagram has the class name, “GenericItem”. The “GenericItem” class has two data members, “-itemName” and “-itemQuantity”, and three functions, “+SetName()“, “+SetQuantity()“, and “+Printitem()“.
The second class diagram has the class name, “ProduceItem”. The “ProduceItem” class has one data member, “-expirationDate” and two functions, “+SetExpiration()” and “+GetExpiration()”. An arrow with a solid line and a closed, unfilled arrowhead points from the second class “ProduceItem” points to the first class, “GenericItem”.
A diagram key is shown:
Member access
- means private
+ means public
# means protected

Step 1: A class diagram depicts a class' name, data members, and functions.
A class diagram with three separate sections are shown. The first sections is shown to be the class name, “GenericItem”. The second section is shown to be the class’s data members, “-itemName” and “-itemQuantity”. The third section is shown to be the class’s functions, “+SetName()“, “+SetQuantity()“, and “+Printitem()“.

Step 2: A solid line with a closed, unfilled arrowhead indicates a class is derived from another class.
Another diagram is shown, only with one section. The one section is shown to be the class name, “ProduceItem”. An arrow with a solid line and a closed, unfilled arrowhead points from the second class’s name “ProduceItem” points to the first class, “GenericItem” indicating the class is derived from the other.

Step 3: The derived class only shows additional members.
Two additional sections are added to the, “ProduceItem” class diagram. The first additional section is shown to be the class’s data member, “-expirationDate”. The second additional section is shown to be the class’s functions, “+SetExpiration()” and “+GetExpiration()“.

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.

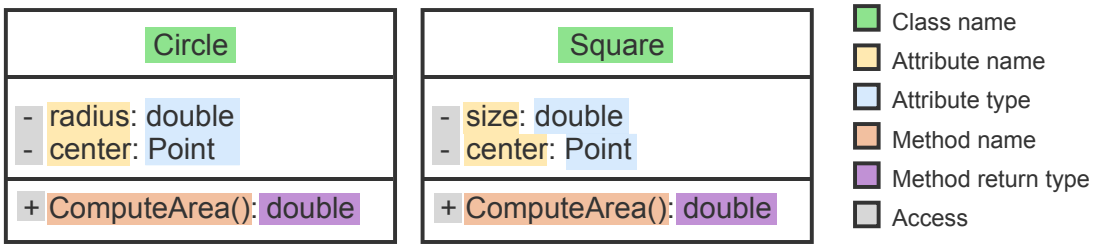
13.8 UML

UML class diagrams

The ***Unified 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 attributes (variables) and operations (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 13.8.1: UML class diagrams show class names, members, types, and access.



Animation content:

Static Figure:
There are two boxes. Each box has a header on the top.

The box on the left contains the following:

The header reads "Circle" and is highlighted in green which means "Circle" is a class name.

The next line reads: "- radius: double".
"-" is highlighted in grey which means "-" describes the access.
"radius:" is highlighted in yellow which means "radius:" is a member variable name.
"double" is highlighted in blue which means "double" is a member variable type.

The next line reads: "- center: Point".
"-" is highlighted in grey which means "-" describes the access.
"center:" is highlighted in yellow which means "center:" is a member variable name.
"Point" is highlighted in blue which means "Point" is a member variable type.

The next line reads: "+ computeArea(): double".
"+" is highlighted in grey which means "+" describes the access.
"computeArea():" is highlighted in orange which means "computeArea()" is a function name.
"double" is highlighted in purple which means "purple" is a function return type.

The box on the right contains the following:

The header reads "Square" and is highlighted in green which means "Square" is a class name.

The next line reads: "- size: double".
"-" is highlighted in grey which means "-" describes the access.
"size:" is highlighted in yellow which means "size:" is an attribute name.
"double" is highlighted in blue which means "double" is an attribute type.

The next line reads: "- center: Point".
"-" is highlighted in grey which means "-" describes the access.
"center:" is highlighted in yellow which means "center:" is an attribute name.
"Point" is highlighted in blue which means "Point" is an attribute type.

The next line reads: "+ computeArea(): double".
"+" is highlighted in grey which means "+" describes the access.
"computeArea():" is highlighted in orange which means "computeArea()" is a function name.
"double" is highlighted in purple which means "purple" is a function return type.

Step 1: One box exists for each class. The class name is centered at the top.
Two boxes appear, each representing a different class. Each box contains the class name, attributes, and methods. The class names are located at the top center of the box. This first is class is name Circle and the second is named Square. Both class names are highlighted in Green.

Step 2: The middle section contains class attributes. Each of a C++ class's member variables is a UML class attribute. An attribute has a name followed by a colon and the type.
The attributes for the classes are shown in the middle section of each class box. The attributes for the Circle class are “- radius: double” and “- center: Point". The attributes for the Square class are “- size: double” and “- center: Point". For both classes, the attributes names are highlighted in yellow, “radius”, “size” and “center”. For both classes, the attribute types are highlighted in grey, “radius”, “size and “center”.

Step 3: The bottom section contains class operations. Each of a C++ class's member functions is a

UML class operation. Each function's name and return type is listed similarly.
The methods for the classes are shown in the bottom section of each class box. Both the method fo the Circle class and Square class are the same, which is, "ComputeArea(): double". For both classes, the method name, "+ ComputeArea()", is highlighted in orange and the method type, "double", is highlighted in purple.

Step 4: Private and public access is noted to the left of each member. A minus (-) indicates private and a plus (+) indicates public.
All the attributes in both the Circle and Square class are private which is indicated with a minus (-) in front of each attribute. All the methods in both the Circle and Square class are public which is indicated with a plus (+) in front of each method. The plus (+) and minus (-) are highlighted in grey.

Animation captions:

1. One box exists for each class. The class name is centered at the top.
2. The middle section contains class attributes. Each of a C++ class's member variables is a UML class attribute. An attribute has a name followed by a colon and the type.
3. The bottom section contains class operations. Each of a C++ class's member functions is a UML class operation. Each 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 13.8.2: 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.

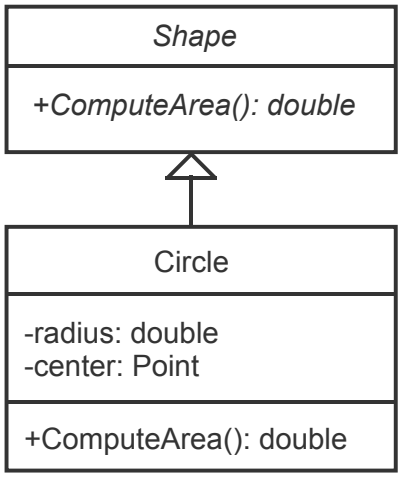
- True
- False

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 13.8.3: UML uses italics for abstract classes and functions.



Animation content:

Static Figure:
There are two boxes.

The top box contains the following:

The header reads "Shape".
The text below the header reads:
"+ComputeArea(): double"

The bottom box contains the following:
The header reads "Circle".
The text below the header reads:
"-radius: double
-center: Point
+ComputeArea(): double"

A solid line with a closed, unfilled arrowhead is shown which connects from the Circle class box to the Shape class box.

Step 1: Shape is an abstract class, so the class name and abstract function are in italics.
A box appears with the class name “Shape” and a function, “+ComputeArea(): double”

Step 2: The solid-lined arrow with an unfilled arrow head indicates that the Circle class inherits from Shape.
Another box appears with a class name “Circle”, two attributes, “-radius: double” and “-center: Point”, and a function “+ComputeArea(): double”.
A solid line with a closed, unfilled arrowhead appears and connects from the Circle class box to the Shape class box. The label, “Circle inherits from Shape” is shown next to the arrow.

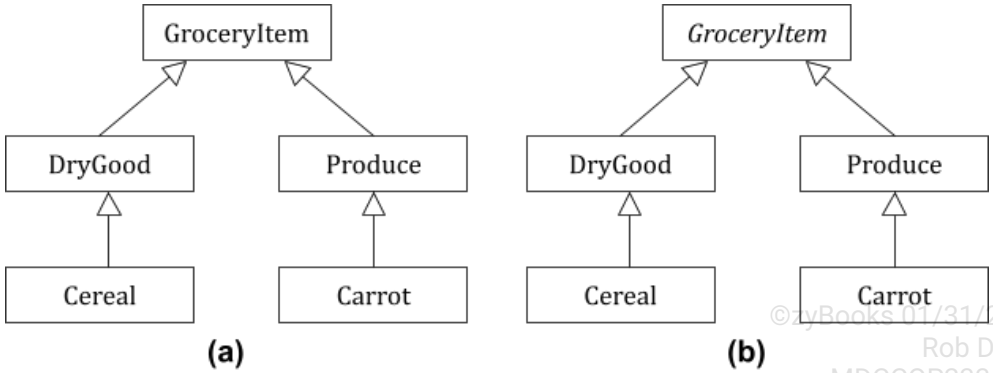
Step 3: Circle is a concrete class, so the class name is shown in regular font. Note that Circle implements ComputeArea().
The class name “Circle” and the Circle class function, “+ComputeArea(): double” are highlighted yellow to indicate the difference to the function in the Shape class.

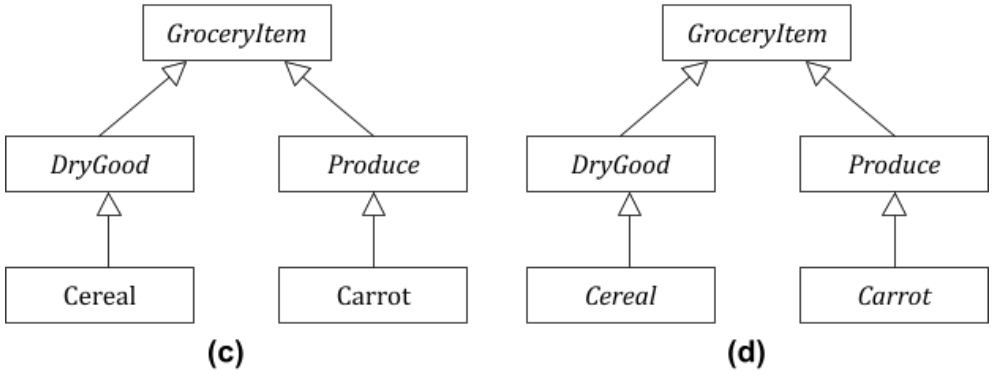
Animation captions:

1. Shape is an abstract class, so the class name and abstract function 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 13.8.4: 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.





If unable to drag and drop, refresh the page.

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

	GroceryItem is abstract and requires all subclasses to implement specific functions. DryGood and Produce can be created as classes.
	GroceryItem is abstract and requires all subclasses to implement specific functions. 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

13.9 C++ example: Employees and overriding class functions

zyDE 13.9.1: Inheritance: Employees and overriding a class function.

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

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

©zyBooks 01/31/24 17:55 1939727
Rob Daglio
MDCCOP2335Spring2024

Current file: EmployeeMain.cpp Load default template...

```
#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 cla
   manager.SetData("Michele", "Sales", "03-03-1975",
   staff1.SetData ("Bob",     "Sales", "02-02-1980",
```


Run

zyDE 13.9.2: 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 Person classes are unchanged.

Current file: EmployeeMain.cpp Load default template...

```
#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 cla
   manager.SetData("Michele", "Sales", "03-03-1975",
   staff1.SetData ("Bob",     "Sales", "02-02-1980",
```


Run

13.10 C++ example: Employees using an abstract class

zyDE 13.10.1: 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 and 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 override EmployeePerson's GetAnnualBonus() pure virtual function but simply return 0.

2. Modify the EmployeeManager and EmployeeStaff GetAnnualBonus functions to return the correct bonus rather than just returning 0. A manager's bonus is 10% of the annual salary, and a staff's bonus is 7.5% of the annual salary.

Current file: EmployeeMain.cpp Load default template...


```
#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 EmployeePe
   manager.SetData("Michele", "Sales", "03-03-1975",
   staff1.SetData ("Bob",     "Sales", "02-02-1980",

   // Print the objects
```

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

Run

zyDE 13.10.2: 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 template...

```
#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 EmployeePe
   manager.SetData("Michele", "Sales", "03-03-1975",
   staff1.SetData ("Bob",     "Sales", "02-02-1980",

   // Print the objects
```


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

Run

13.11 LAB: Pet information (derived classes)

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

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

Ex. If the input is:

```
Dobby
2
Kreacher
```

```
3
Scottish Fold
```

the output is:

```
Pet Information:
   Name: Dobby
   Age: 2
Pet Information:
   Name: Kreacher
   Age: 3
   Breed: Scottish Fold
```


LAB ACTIVITY 13.11.1: LAB: Pet information (derived classes) 0 / 10

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

```
1  #include <iostream>
2  #include<string>
3  #include "Cat.h"
4
5  using namespace std;
6
7  int main() {
8
9     string petName, catName, catBreed;
10    int petAge, catAge;
11
12    Pet myPet;
13    Cat myCat;
14
15    getline(cin, petName);
```

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) → Output

Program output displayed here

Coding trail of your work What is this?

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

13.12 Project 2 - Pet information (derived classes)

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

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

Ex. If the input is:

```
Dobby
2
Kreacher
3
Scottish Fold
```

the output is:


```
Pet Information:
   Name: Dobby
   Age: 2
Pet Information:
   Name: Kreacher
   Age: 3
   Breed: Scottish Fold
```

LAB ACTIVITY | 13.12.1: Project 2 - Pet information (derived classes) 0 / 10

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


```
1  #include <iostream>
2  #include<string>
3  #include "Cat.h"
4
5  using namespace std;
6
7  int main() {
8
9     string petName, catName, catBreed;
10    int petAge, catAge;
11
12    Pet myPet;
13    Cat myCat;
14
15    getline(cin, petName);
```

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) → Output

Program output displayed here

Coding trail of your work What is this?

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

13.13 LAB: Instrument information

Given main() and the `Instrument` class, define a derived class, `StringInstrument`, for string instruments, with functions to set and get private data members of the following types:

- int to store the number of strings
- int to store the number of frets
- bool to store whether the instrument is bowed

Ex. If the input is:

```
Drums
Zildjian
2015
2500
Guitar
Gibson
2002
1200
6
19
0
```

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
   Is bowed: false
```

LAB ACTIVITY | 13.13.1: 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, stringMa
8        int yearBuilt, cost, stringYearBuilt, stringCost, numStrings, numFrets;
9        bool bowed;
10
11       cin >> instrumentName;
12       cin >> manufacturerName;
13       cin >> yearBuilt;
14       cin >> cost;
15
```


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) → Output

Program output displayed here

Coding trail of your work What is this?

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

13.14 LAB: Course information (derived classes)

Given main(), define a `Course` base class with functions to set and get the private data members of the following types:

- string to store the course number
- string to store the course title

Define `Course`'s PrintInfo() function that outputs the course number and title.

Then, define a derived class `OfferedCourse` with functions to set and get the private data members of the following types:

- string to store the instructor name
- string to store the location
- string class time

Ex. If the input is:

```
ECE287
Digital Systems Design
ECE387
Embedded Systems Design
Mark Patterson
Wilson Hall 231
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
   Location: Wilson Hall 231
   Class Time: WF: 2-3:30 pm
```


LAB ACTIVITY | 13.14.1: LAB: Course information (derived classes) | 0 / 10

File is marked as read only

Current file: main.cpp

```
#include "OfferedCourse.h"

int main() {
   Course myCourse;
   OfferedCourse myOfferedCourse;

   string courseNumber, courseTitle;
   string oCourseNumber, oCourseTitle, instructorName, location, classTime

   getline(cin, courseNumber);
   getline(cin, courseTitle);

   getline(cin, oCourseNumber);
   getline(cin, oCourseTitle);
   getline(cin, instructorName);
```


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) → Output

Program output displayed here

Coding trail of your work What is this?

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

13.15 LAB: Book information (overriding member functions)

Given main() and a base **Book** class, define a derived class called **Encyclopedia** with member functions to get and set private class data members of the following types:

- string to store the edition
- int to store the number of pages

Within the derived **Encyclopedia** class, define a PrintInfo() member function that overrides the **Book** class' PrintInfo() function by printing the title, author, publisher, publication date, edition, and number of pages.

Ex. If the input is:

```
The Hobbit
J. R. R. Tolkien
George Allen & Unwin
21 September 1937
The Illustrated Encyclopedia of the Universe
Ian Ridpath
Watson-Guptill
2001
2nd
384
```

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: Ian Ridpath
   Publisher: Watson-Guptill
   Publication Date: 2001
   Edition: 2nd
   Number of Pages: 384
```


Note: Indentations use 3 spaces.

LAB ACTIVITY | 13.15.1: 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() {
8     Book myBook;
9     Encyclopedia myEncyclopedia;
10
11    string title, author, publisher, publicationDate;
12    string eTitle, eAuthor, ePublisher, ePublicationDate, edition;
13    int numPages;
14
15    getline(cin, title);
```

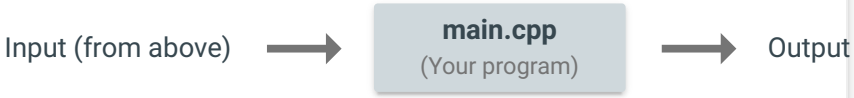

Develop mode | Submit mode

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

Coding trail of your work What is this?

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

13.16 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), add 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 1 Information:
   Plant name: Spirea
   Cost: 10

Plant 2 Information:
   Plant name: Hydrangea
   Cost: 30
   Annual: false
   Color of flowers: lilac

Plant 3 Information:
   Plant name: Rose
   Cost: 6
   Annual: false
   Color of flowers: white

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


LAB ACTIVITY | 13.16.1: 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 a vector of plant (or flow
10
11 int main() {
12    // TODO: Declare a vector called myGarden that can hold object of type pla
13
14    // TODO: Declare variables - plantName, plantCost, flowerName, flowerCost,
15    //       colorOfFlowers, isAnnual
```


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) → Output

Program output displayed here

Coding trail of your work What is this?

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