12.1 Derived classes

Commonly, one class is similar to another class but with some additions or variations. For example, a store inventory system might use a class called GenericItem having itemName and itemQuantity members. But for produce (fruits and vegetables), a class ProduceItem having itemName, itemQuantity, and expirationDate members may be desired. Note that ProduceItem is really a GenericItem with an additional feature, so ideally a program could define the ProduceItem class as being the same as the GenericItem class but with the addition of an expirationDate member.

Such similarity among classes is supported by indicating that a class is derived from another class, as shown below.

Figure 12.1.1: A derived class example: Class ProduceItem is derived from class GenericItem.

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

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;
};

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

   string GetExpiration() {
      return expirationDate;
   };

private:
   string expirationDate;
};

int main() {
   GenericItem miscItem;
   ProduceItem perishItem;

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

   perishItem.SetName("Apples");
   perishItem.SetQuantity(40);
   perishItem.SetExpiration("May 5, 2012");
   perishItem.PrintItem();
   cout << "  (Expires: " << perishItem.GetExpiration()
   << ")" << endl;

   return 0;
}
```



```
Smith Cereal 9
Apples 40
  (Expires: May 5, 2012)
```

©zyBooks 04/05/18 21:47 261830
Julian Chan
WEBERCS2250ValleSpring2018

A class named GenericItem is defined as normal. In main(), a GenericItem variable miscItem is declared, the item's data fields set to "Smith Cereal" and "9", and the item's PrintItem() member function called. A class named ProduceItem is also defined, that class was *derived* from the

GenericItem class by appending `: public GenericItem` after the name ProduceItem , i.e., `class ProduceItem : public GenericItem {` . As such, declaring the ProduceItem variable perishItem creates an object with data members itemName and itemQuantity (from GenericItem) plus expirationDate (from ProduceItem). Also, ProduceItem has member function SetName(), SetQuantity(), and PrintItem() (from GenericItem) plus SetExpiration() and GetExpiration() (from ProduceItem). So in main(), perishItem has its data fields set to "Apples", "40", and "May 5, 2012", and the item is printed using the PrintItem() member function and using the GetExpiration() member function. (Note: We have written the code unusually concisely to help focus attention on the derivation concepts being learned)

The term derived class (or ***subclass***) refers to a class that is derived from another class that is known as a ***base class*** (or ***superclass***). Any class may serve as a base class; no changes to the definition of that class are required. The derived class is said to inherit the properties of its base class, a concept commonly 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. The following animation illustrates the relationship between a derived class and a base class.

PARTICIPATION ACTIVITY 12.1.1: Derived class example: ProduceItem derived from GenericItem.

Animation captions:

1. GenericItem has its own members.
2. ProduceItem is derived from GenericItem so inherits GenericItem's members, plus has its own members.

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

PARTICIPATION ACTIVITY 12.1.2: UML derived class example: ProduceItem derived from GenericItem.

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.

Various class derivation variations are possible:

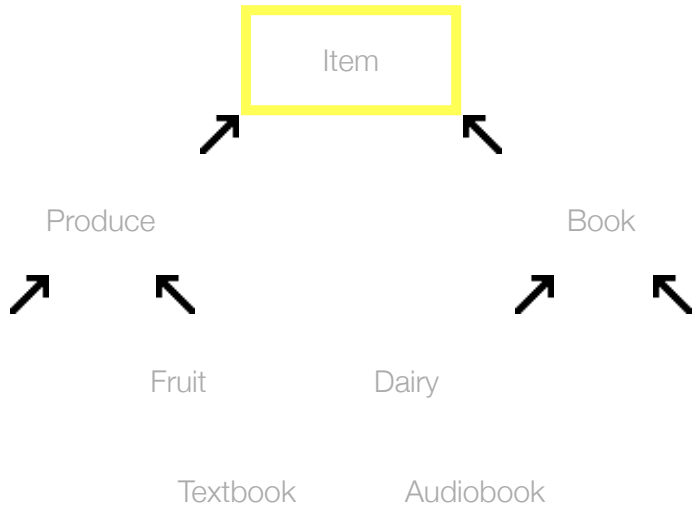
- A derived class can itself serve as a base class for another class. In the earlier example, `class FruitItem: public ProduceItem {...}` could be added.

- A class can serve as a base class for multiple derived classes. In the earlier example, `class ProduceItem: public GenericItem {...}` could be added.
- A class may be derived from multiple classes. For example, `class House: public Dwelling, public Property {...}` could be declared.

PARTICIPATION ACTIVITY 12.1.3: Interactive inheritance tree.

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

Inheritance tree



Selected class psuedocode

```
public:
   void   SetName(string nm)
   void   SetQuantity(int
qnty)
   void   Print()

private:span>
   string name;
   int    quantity;
```

Selected class code

```
class Item {
   public:
      void SetName(string nm)
         { name = nm; };
      void SetQuantity(int qnty)
         { quantity = qnty; };
      void Print() {
         cout << name << " "
              << quantity
              << endl;
      };

   private:
      string name;
```

```
    int     quantity;
};
```

PARTICIPATION ACTIVITY 12.1.4: Derived classes basic.

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

Check Show answer

2) 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

Exploring further:

- Overview of Derived Classes from msdn.microsoft.com.

CHALLENGE ACTIVITY 12.1.1: Derived classes.

Start

Type the program's output.

```
Driving at: 40
```

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

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

   void PrintSpeed() {
      cout << speed;
   };

private:
   int speed;
};

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

int main() {
   Car myCar;
   myCar.SetSpeed(40);

   myCar.PrintCarSpeed();

   return 0;
}
```


1 2 3 4

Check Next

CHALLENGE ACTIVITY 12.1.2: Basic inheritance.

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

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



```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class PersonData {
6 public:
7    void SetName(string userName) {
```

```
   8        lastName  = userName;
   9     };
  10     void SetAge(int numYears) {
  11        ageYears = numYears;
  12     };
  13     // Other parts omitted
  14
  15     void PrintAll() {
  16        cout << "Name: " << lastName;
  17        cout << ", Age: "  << ageYears;
  18     };
```


Run

12.2 Access by members of derived classes

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 earlier ProduceItem class yields a compiler error.

Figure 12.2.1: Member functions of a derived class cannot access private members of the base class.

```
class ProduceItem : public GenericItem {
public:

   ...

   void DisplayProduceItem() {
      cout << itemName << " " << itemQuantity
      << " (Expires: " << expirationDate << ")"
      << endl;
   };

   ...
};
```

```
$ g++ -Wall derived.cpp
derived.cpp: In member function 'void ProduceItem::DisplayProduceItem()':
derived.cpp:14: error: 'std::string GenericItem::itemName' is private
derived.cpp:25: error: within this context
derived.cpp:15: error: 'int GenericItem::itemQuantity' is private
derived.cpp:25: error: within this context
```


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.

Figure 12.2.2: Access specifiers -- Protected allows access by derived classes but not by others.

Code contains intended errors to demonstrate protected accesses.


```
class BaseClass {
public:    // Members accessible by anyone
   void PrintMembers();

protected: // Members accessible by self and derived classes
   string baseName;

private:   // Members accessible only by self
   int baseCount;
};

class DerivedClass : public BaseClass {
   ...

   // Attempted accesses
   PrintMembers();                // OK
   baseName = "Mike";             // OK    ("protected" above made this possible)
   baseCount = 1;                 // ERROR

   ...
};

int main() {
   ...

   BaseClass baseObj;
   DerivedClass derivedObj;

   ...

   // Attempted accesses
   baseObj.PrintMembers();        // OK
   baseObj.baseName  = "Mike";    // ERROR (protected only applies to derived classes)
   baseObj.baseCount = 1;         // ERROR

   derivedObj.PrintMembers();     // OK
   derivedObj.baseName  = "Mike"; // ERROR
   derivedObj.baseCount = 1;      // ERROR

   return 0;
}
```


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

To make ProduceItems DisplayProduceItem() function work, we merely need to change the private members to protected members in class GenericItem. GenericItem's data members itemName and itemQuantity thus become accessible to a derived class like ProduceItem, 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 12.2.1: Access specifiers.

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

Separately, the keyword "public" in a class definition like `class DerivedClass: public BaseClass {...}` has a rather 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 12.2.1: Access by derived class members.

Assume `class DerivedClass : public BaseClass {...}`

1) BaseClass' public member function can be called by a member function of DerivedClass.

- ○ Yes
- ○ No

2) BaseClass' protected member function can be called by a member function of DerivedClass.

- Yes
- No

3) BaseClass' private data members can be accessed by a member function of DerivedClass.

- Yes
- No

4) For `DerivedClass derivedObj;` in main(), derivedObj can access a protected member of BaseClass.

- Yes
- No

Exploring further:

- More on Protected from msdn.microsoft.com

12.3 Overriding member functions

A derived class may define a member function having the same name as the base class. Such a member function ***overrides*** the function of the base class. The following example shows the earlier GenericItem/ProduceItem example where the ProduceItem class has its own PrintItem() member function that overrides the PrintItem() function of the GenericItem class.

Figure 12.3.1: ProduceItem's PrintItem() function overrides GenericItem's PrintItem() function.

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

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

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

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

protected:
   string itemName;
   int itemQuantity;
};

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

   string GetExpiration() {
      return expirationDate;
   };

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

private:
   string expirationDate;
};

int main() {
   GenericItem miscItem;
   ProduceItem perishItem;

   miscItem.SetName("Smith Cereal");
   miscItem.SetQuantity(9);
   miscItem.PrintItem();    // Calls GenericItem's PrintItem()

   perishItem.SetName("Apples");
   perishItem.SetQuantity(40);
   perishItem.SetExpiration("May 5, 2012");
   perishItem.PrintItem(); // Calls ProduceItem's PrintItem()

   return 0;
}
```


©zyBooks 04/05/18 21:47 261830
Julian Chan
WEBERCS2250ValleSpring2018

```
Smith Cereal 9
Apples 40 (Expires: May 5, 2012)
```

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

The overriding function can still call the overridden function by prepending the base class name, as in `GenericItem::PrintItem()`, as follows.

Figure 12.3.2: Function calling overridden function of base class.

```
class ProduceItem : public GenericItem { // Derived from GenericItem

   ...

   void PrintItem() {
      GenericItem::PrintItem();
      cout << " (Expires: " << expirationDate << ")" << endl;
   };

   ...
};
```

Without the prepended base class name, the call to PrintItem() would refer to itself (a *recursive* call), so the function would call itself, and that call would call itself, etc., never actually printing anything (an error in this case).

PARTICIPATION ACTIVITY 12.3.1: Override.

Assume myItem is declared as GenericItem, and myProduce as ProduceItem, with classes GenericItem and ProduceItem defined as above.

1) `myItem.PrintItem()` calls the PrintItem() function for which class?
   - GenericItem
   - ProduceItem

2) `myProduce.PrintItem()` calls the PrintItem() function for which class?
   - GenericItem
   - ProduceItem

3) Provide a statement within PrintItem() function of the ProduceItem class to

call the PrintItem() function of ProduceItem's base class.

- PrintItem();
- base.PrintItem();
- GenericItem::PrintItem();

4) If ProduceItem did NOT have its own PrintItem() function defined, the PrintItem() function of which class would be called?

- GenericItem
- ProduceItem
- A call to PrintItem() yields an error.

CHALLENGE ACTIVITY 12.3.1: Overriding member function.

Start

Type the program's output.

```
WiFi: goo
CPU: 20%
```

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

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

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

protected:
   string cpuUsage;
   string internet;
};

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

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

private:
   string wifiQuality;
};

int main() {
   Laptop myLaptop;

   myLaptop.SetComputerStatus("20%", "connected");
   myLaptop.SetWiFiStatus("good");

   myLaptop.PrintStatus();

   return 0;
}
```


1 2 3

Check Next

CHALLENGE ACTIVITY 12.3.2: Basic derived class member override.

©zyBooks 04/05/18 21:47 261830
Julian Chan
WEBERCS2250ValleSpring2018

Define a member function PrintAll() for class PetData that prints output as follows. Hint: Make use of the base class' PrintAll() function.

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

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

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

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

```


Run

12.4 Polymorphism and virtual member functions

Polymorphism refers to determining which program behavior to execute depending on data types. Function overloading is a form of ***compile-time polymorphism*** wherein the compiler determines which of several identically-named functions to call based on the function's arguments. Another form is ***runtime polymorphism*** wherein the compiler cannot make the determination but instead the determination is made while the program is running.

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<GenericItem*> inventoryList;` creates a vector that can contain pointers to objects of type GenericItem or ProduceItem. ProduceItem derives from GenericItem. Similarly, polymorphism is also used for references to objects. Ex: `GenericItem& saleItem` declares a reference that can refer to objects of ProduceItem or GenericItem.

Figure 12.4.1: Runtime polymorphism via a virtual function.

When the object is a pointer, the indication of virtual causes the program to dynamically determine the correct function to call based on the pointer type.

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

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

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

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

protected:
   string itemName;
   int itemQuantity;
};

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

   string GetExpiration() {
      return expirationDate;
   };

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

private:
   string expirationDate;
};

int main() {
   GenericItem* genericItemPtr = nullptr;
   ProduceItem* produceItemPtr = nullptr;
   vector<GenericItem*> inventoryList;
   int i = 0;

   genericItemPtr = new GenericItem;
   genericItemPtr->SetName("Smith Cereal");
   genericItemPtr->SetQuantity(9);

   produceItemPtr = new ProduceItem;
   produceItemPtr->SetName("Apple");
   produceItemPtr->SetQuantity(40);
   produceItemPtr->SetExpiration("May 5, 2012");

   genericItemPtr->PrintItem();
   produceItemPtr->PrintItem();

   // More common: Collection (e.g., vector) of objs
   // Polymorphism -- Correct Print() called
   inventoryList.push_back(genericItemPtr);
   inventoryList.push_back(produceItemPtr);

   cout << endl << "Inventory:" << endl;
   for (i = 0; i < inventoryList.size(); ++i) {
```

Without the keyword "virtual":

```
Smith Cereal 9
Apple 40 (Expires: May 5,
2012)

Inventory:
Smith Cereal 9
Apple 40
```


With the keyword "virtual":

```
Smith Cereal 9
Apple 40 (Expires: May 5,
2012)

Inventory:
Smith Cereal 9
Apple 40 (Expires: May 5,
2012)
```

```
        inventoryList.at(i)->PrintItem();
    }

    return 0;
}
```

The program uses a C++ feature relating to ***derived/base class pointer conversion*** wherein a pointer to a derived class can be converted to a pointer to the base class (without explicit casting). Such conversion is in contrast to other pointer conversions, such as converting an integer pointer to a character pointer (which is an error unless explicitly cast). Thus, the above statement `inventoryList.push_back(produceItemPtr);` uses this feature, with a ProduceItem pointer being converted to a GenericItem pointer (inventoryList is a vector of GenericItem pointers). The conversion is intuitive; recall in an earlier animation that a derived class like ProduceItem consists of the base class GenericItem plus additional members, so the conversion yields a pointer to the base class part (so really there's no change).

A problem however is that when printing the vector contents, for a given element, how does the program know whether to call GenericItem's PrintItem() or ProduceItem's PrintItem()? If we do nothing special, the program will simply call GenericItem's PrintItem() function because all elements are GenericItem pointers, which is why in the first sample output above, the printing of the vector elements doesn't print the "Expires" part.

The solution is to declare the function as virtual. A ***virtual function*** is a member function that may be overridden in a derived class and for which runtime polymorphism is used. A virtual function is declared by prepending the keyword "virtual". Ex: `virtual void PrintItem()`. At runtime, when a virtual function is called using a pointer or reference to an object, the correct function to call is dynamically determined based on the actual object type to which the pointer or reference refers. The compiler generates code necessary to keep track of the type of object being referred to, so that the appropriate function can be called.

The word "virtual" can be slightly confusing because virtual usually means something doesn't really exist, whereas a virtual function in a base class does exist; the word virtual just relates to how that function gets overridden. In contrast, one can create a truly virtual function, known as a ***pure virtual function***, using the syntax shown below.

Figure 12.4.2: Making GenericItem's PrintItem() function a pure virtual function.

```
class GenericItem {
      ...

      virtual void PrintItem() = 0;

      ...
};
```


The syntax is rather odd but the compiler understands the syntax to indicate that this is a pure virtual function, which means the function must be defined in a derived class. A class that has at

least one pure virtual function is known as an ***abstract base class***, meaning objects cannot be declared of that type. For example, the variable declaration `GenericItem genericItem1;` would generate a compiler error like the following:

Figure 12.4.3: Sample compiler error when trying to declare a variable of an abstract base class type.


```
inventory.cpp: In function 'int main()':
inventory.cpp:35: error: cannot declare variable 'genericItem1' to be of abstract type
'GenericItem'
inventory.cpp:6: note: because the following virtual functions are pure within 'GenericItem':
inventory.cpp:12: note: virtual void GenericItem::PrintItem()
```

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

PARTICIPATION ACTIVITY 12.4.1: Virtual member functions.

Consider the GenericItem and ProduceItem classes defined in an earlier section.

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

2) Prepending the word "virtual" to GenericItem's PrintItem() function allows a derived class like ProduceItem to override with its own PrintItem() function.
   - ○ True
   - ○ False

3) A class having a pure virtual function implies that objects of that class type cannot be declared.
   - ○ True
   - ○

False

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

Figure 12.4.4: Possible warning messages when using virtual functions.

```
inventory.cpp:6: warning: 'class GenericItem' has virtual functions but non-virtual destructor
inventory.cpp:19: warning: 'class ProduceItem' has virtual functions but non-virtual
destructor
```

Exploring further:

- More on Polymorphism from cplusplus.com

CHALLENGE ACTIVITY 12.4.1: Basic polymorphism.

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

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

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4  using namespace std;
5
6  class BaseItem {
7  public:
8     void SetLastName(string providedName) {
9        lastName = providedName;
10    };
11
12    // FIXME: Define PrintItem() member function
13
14    /* Your solution goes here  */
15
16 protected:
17    string lastName;
18 };
19
```

Run

12.5 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 12.5.1: 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 12.5.2: 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 12.5.1: 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

12.6 C++ example: Employees and overriding class

PARTICIPATION ACTIVITY 12.6.1: 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.

Current file: **EmployeeMain.cpp**

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

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

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

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

```


Run

PARTICIPATION ACTIVITY 12.6.2: 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**

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

int main() {

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

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

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

Run

12.7 C++ example: Employees using an abstract c

PARTICIPATION ACTIVITY 12.7.1: 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**

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

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

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

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

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

Run

PARTICIPATION ACTIVITY 12.7.2: 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**

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

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

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

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


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

Run