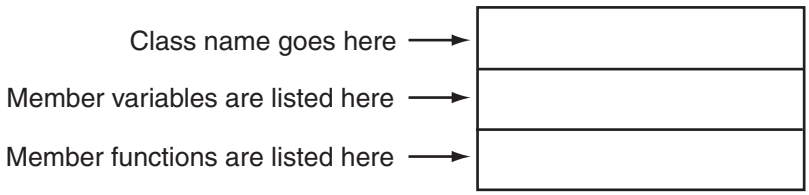
Appendix D: Using UML in Class Design

When designing a class it is often helpful to draw a UML diagram. *UML* stands for *Unified Modeling Language*. The UML provides a set of standard diagrams for graphically depicting object-oriented systems. Figure D-1 shows the general layout of a UML diagram for a class. Notice that the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's member variables. The bottom section holds a list of the class's member functions.

Figure D-1



For example, in Chapter 13, Introduction to Classes, you studied a `Rectangle` class that could be used in a program that works with rectangles. The class has the following member variables:

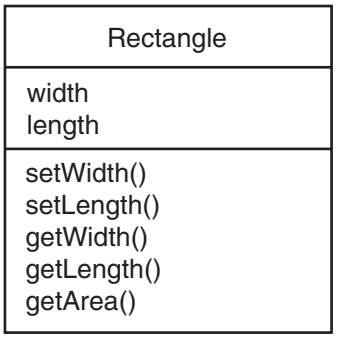
- `width`
- `length`

The class also has the following member functions:

- `setWidth`
- `setLength`
- `getWidth`
- `getLength`
- `getArea`

From this information alone we can construct a simple UML diagram for the class, as shown in Figure D-2.

Figure D-2



The UML diagram in Figure D-2 tells us the name of the class, the names of the member variables, and the names of the member functions. Compare this diagram to the actual C++ class declaration, which follows.

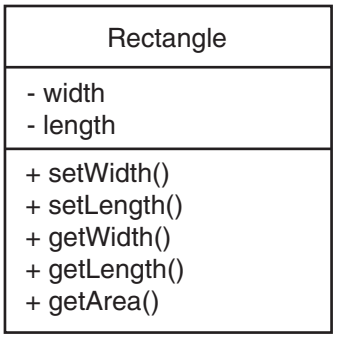
```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

The UML diagram in Figure D-2 does not convey many of the class details, such as access specification, member variable data types, parameter data types, and function return types. The UML provides optional notation for these types of details.

Showing Access Specification in UML Diagrams

The UML diagram in Figure D-2 lists all of the members of the `Rectangle` class but does not indicate which members are private and which are public. In a UML diagram you may optionally place a **-** character before a member name to indicate that it is private, or a **+** character to indicate that it is public. Figure D-3 shows the UML diagram modified to include this notation.

Figure D-3



Data Type and Parameter Notation in UML Diagrams

The Unified Modeling Language also provides notation that you may use to indicate the data types of member variables, member functions, and parameters. To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable. For example, the `width` variable in the `Rectangle` class is a `double`. It could be listed as follows in the UML diagram:

```
- width : double
```

NOTE: In UML notation the variable name is listed first, then the data type. This is opposite of C++ syntax, which requires the data type to appear first.

The return type of a member function can be listed in the same manner: After the function's name, place a colon followed by the return type. The `Rectangle` class's `getLength` function returns a `double`, so it could be listed as follows in the UML diagram:

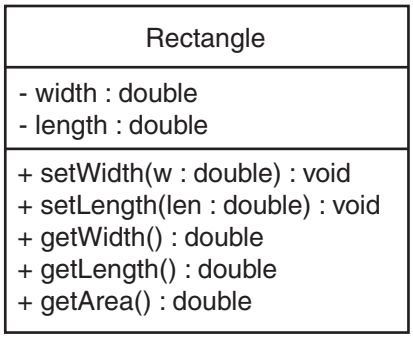
```
+ getLength() : double
```

Parameter variables and their data types may be listed inside a member function's parentheses. For example, the `Rectangle` class's `setLength` function has a `double` parameter named `len`, so it could be listed as follows in the UML diagram:

```
+ setLength(len : double) : void
```

Figure D-4 shows a UML diagram for the `Rectangle` class with parameter and data type notation.

Figure D-4



Showing Constructors and Destructors in a UML Diagram

There is more than one accepted way of showing a class's constructor in a UML diagram. In this text we show a constructor just as any other method, except we list no return type. Figure D-5 shows a UML diagram for the first version of the `InventoryItem` class, which was discussed in Chapter 13. This version of the class has one constructor, which accepts three arguments.

Figure D-5

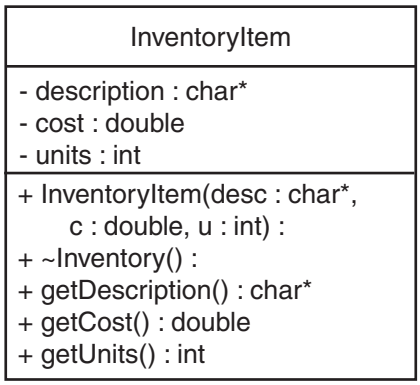
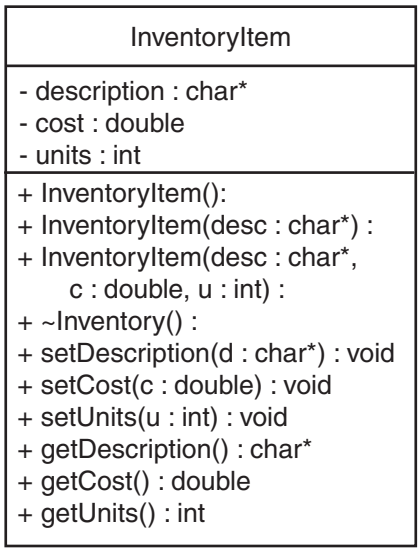


Figure D-6 shows a UML diagram for the second version of the `InventoryItem` class, which has three overloaded constructors, as well as some additional member functions.

Figure D-6

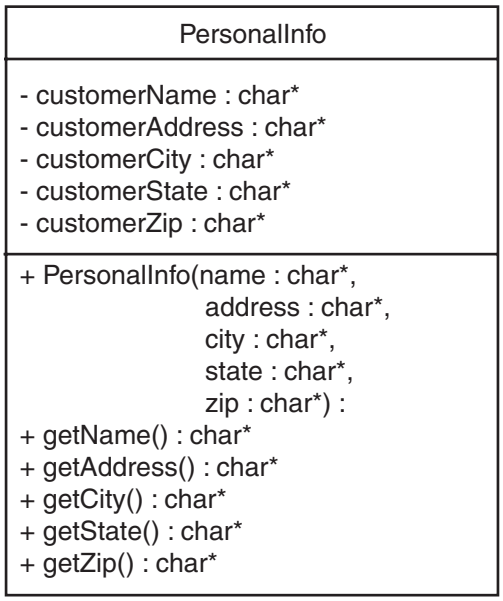


Aggregation UML Diagrams

Aggregation occurs when a class contains an instance of another class as a member. You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class that contains instances of other classes.

For example, suppose we have the `PersonalInfo` class shown in Figure D-7, which holds information about a person.

Figure D-7



Suppose we also have the `BankAccount` class shown in Figure D-8, which holds the balance of a bank account, and can perform operations such as making deposits and withdrawals.

Figure D-8

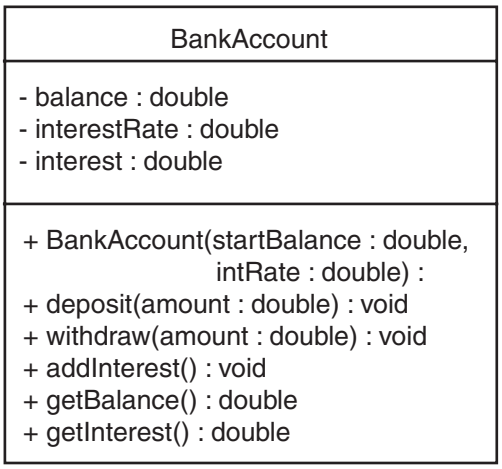


Figure D-9 shows a UML diagram for another class, `BankCustomer`, which contains instances of the `PersonalInfo` and `BankAccount` classes as members. The relationship between the classes is shown by the connecting lines with the open diamond. The open diamond is closest to the `BankCustomer` class because it contains instances of the other classes as members.

Inheritance in UML Diagrams

You show inheritance in a UML diagram by connecting two classes with a line that has an open arrowhead at one end. The arrowhead points to the base class. For example, Figure E-10 shows a UML diagram depicting the relationship between the `GradedActivity` and `FinalExam` classes that you studied in Chapter 15. The arrowhead points toward the `GradedActivity` class, which is the base class.

Showing Protected Members

Protected class members may be denoted in a UML diagram with the `#` symbol. In the second version of the `GradedActivity` class, in Chapter 15, the `score` member variable was declared `protected`. Figure D-11 shows a UML diagram for the class.

Figure D-9

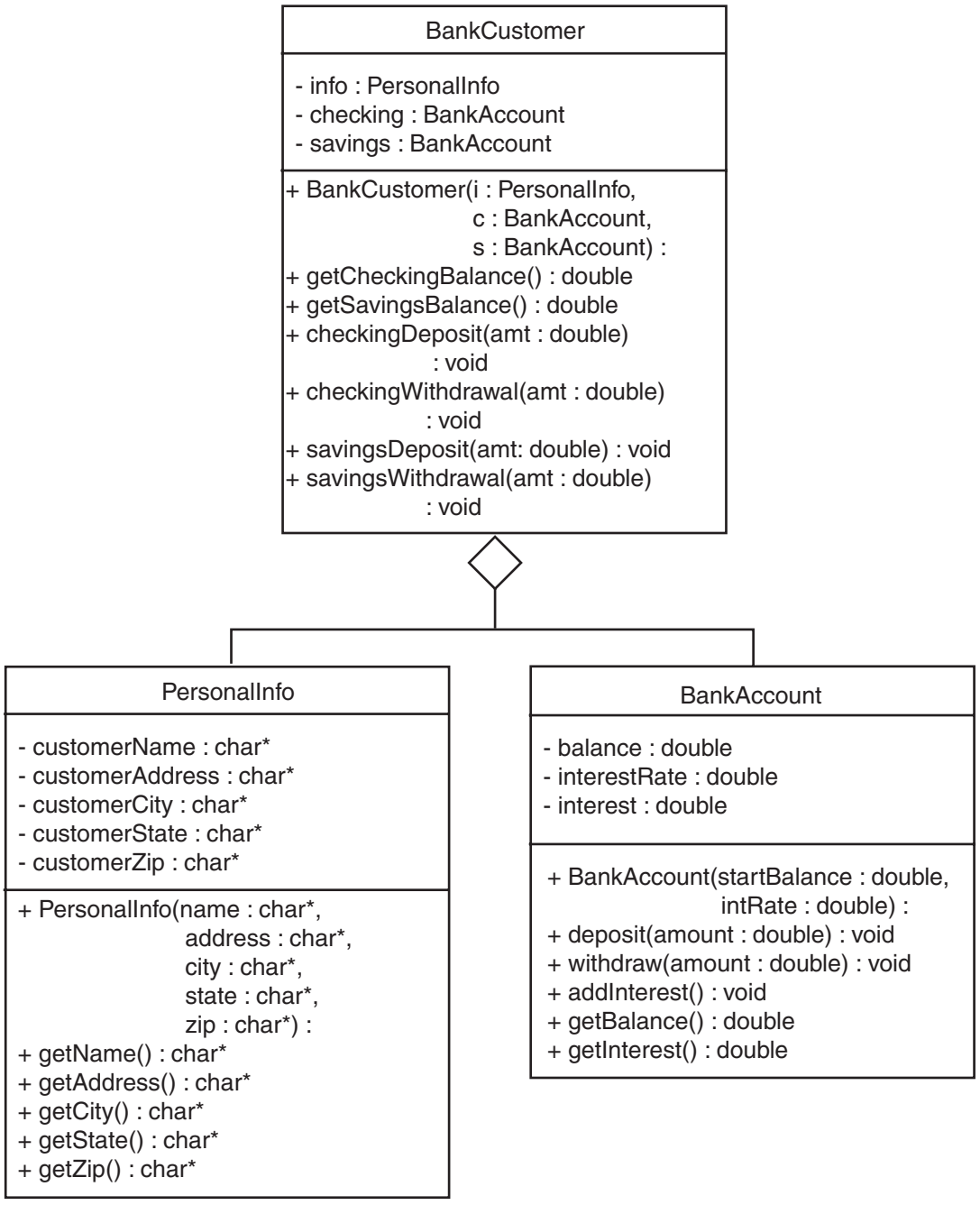

BankCustomer
- info : PersonalInfo
- checking : BankAccount
- savings : BankAccount
+ BankCustomer(i : PersonalInfo,
c : BankAccount,
s : BankAccount) :
+ getCheckingBalance() : double
+ getSavingsBalance() : double
+ checkingDeposit(amt : double)
: void
+ checkingWithdrawal(amt : double)
: void
+ savingsDeposit(amt: double) : void
+ savingsWithdrawal(amt : double)
: void
PersonalInfo
- customerName : char*
- customerAddress : char*
- customerCity : char*
- customerState : char*
- customerZip : char*
+ PersonalInfo(name : char*,
address : char*,
city : char*,
state : char*,
zip : char*) :
+ getName() : char*
+ getAddress() : char*
+ getCity() : char*
+ getState() : char*
+ getZip() : char*
BankAccount
- balance : double
- interestRate : double
- interest : double
+ BankAccount(startBalance : double,
intRate : double) :
+ deposit(amount : double) : void
+ withdraw(amount : double) : void
+ addInterest() : void
+ getBalance() : double
+ getInterest() : double

Figure D-10

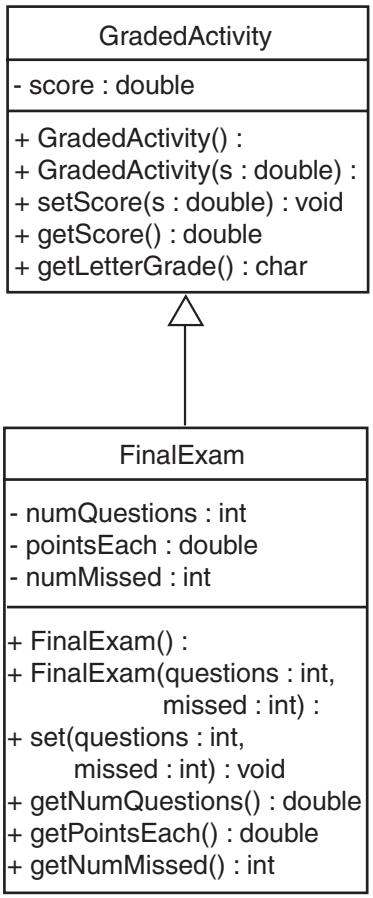


Figure D-11