Chapter 11: Introduction to Classes

C++ FOR ENGINEERS AND SCIENTISTS

Objectives

- In this chapter you will learn about:
  - Classes
  - Basic class functions
  - Adding class functions
  - A case study involving the construction of a `Date` object
  - Unified Modeling Language (UML) class and object diagrams
  - Common programming errors

Classes

- A procedural program consists of one or more algorithms that have been written in computer-readable language
  - Input and display of program output take a back seat to processing
    - Clear emphasis on formulas and calculations
- An object-oriented approach fits graphically windowed environments
- Abstract data types: Central to creation of objects; a user defined rather than built-in data type

Refer to page 618 for more explanations and examples

Abstract Data Types

- **Data type**: Combination of data and associated operations
- A data type defines *both* the types of data and the types of operations that can be performed on the data
  - *Data type = Allowable Data Values + Operational Capabilities*
- Operations in C++ are an inherent part of each data type

Abstract Data Types (continued)

Capability	Example
Define one or more variables of the data type	`int a, b;`
Initialize a variable at definition	`int a = 5;`
Assign a value to a variable	`a = 10;`
Assign one variable's value to another variable	`a = b;`
Perform mathematical operations	`a + b`
Perform relational operations	`a > b`
Convert from one data type to another	`a = int(7.2);`

Table 11.1 C++ Built-In Data Type Capabilities

Abstract Data Types (continued)

- **Abstract data type (ADT)**: User defined type that specifies both a type of data and the operations that can be performed on it
  - User defined types are required when you want to create objects that are more complex than simple integers and characters
- **Data structure**: How data is stored
- **Class**: C++ name for an abstract data type

Refer to pages 619-620 for more explanations and examples

Class Construction

- A class is usually constructed in two parts:
  - **Declaration section**
    - Declares both the data types and functions for the class
  - **Implementation section**
    - Defines the functions whose prototypes have been declared in the declaration section

Class Construction (continued)

- **Class members**: Both the variables and the functions listed in the declaration section
- **Data members** or **instance variables**: Variables listed in the declaration section
- **Member functions**: Functions listed in the declaration section
- When a function is part of a class it is referred to as a **method** to denote class membership

Class Construction (continued)

- Initial uppercase letter in class name `Date` not required but followed by convention
- Keywords `public` and `private` are access specifiers that define access rights
  - `private`: Indicates that class member can only be accessed by class functions
- Restricting user from access to data storage implementation details is called **data hiding**
- After a class category like `private` is designated, it remains in force until a new category is specified

Refer to pages 620-622 for more explanations and examples

Class Construction (continued)

- `public` functions *can* be called from outside the class
- In general, all class functions should be `public` so that they provide capabilities to manipulate class variables from outside the class
- The function with same name as class is the class's **constructor function**
  - Used to initialize class data members with values

Refer to page 623 for more explanations and examples

Class Construction (continued)

- Implementation section: member functions declared in the declaration section are written
- General form for functions written in the implementation section is the same as all C++ functions with the addition of the class name and the scope resolution operator `::`

Refer to pages 623-625 for more explanations and examples

Class Construction (continued)

- Variables of a user-declared class must:
  - Be defined before use in a program
  - Are referred to as **objects**
- An object name's attribute is referenced with the **dot operator**
  - *objectName.attributeName*

    objectName is the name of a specific object

    attributeName is the name of a data member defined for the object's class

Refer to page 626 for more explanations and examples

Class Construction (continued)

- The syntax for referring to an object's method is:
  - *objectName.methodName(parameters)*
    - *objectName* is the name of the specific object
    - *methodName* is name of a function defined for the object's class

Refer to pages 627,628 for more explanations and examples

Terminology

- **Class**: Programmer defined data type from which objects can be created
- **Objects**: Created from classes
  - Referred to as **instances** of a class
- Process of creating a new object is called **instantiation** of the object
- Each time a new object is instantiated a new set of data members belonging to the object is created
  - Values contained in these data members determine the object's **state**

Refer to page 629 for more explanations and examples

Basic Class Functions

- Constructor: A function used to initialize an object's data members when the object is created
- Accessor: A function that reports information about an object's state
- Mutator: A function that modifies the values stored in an object's data members

Constructor Functions

- A **constructor function** is any function with the same name as its class
- Multiple constructors can be defined for each class as long as they can be distinguished by number and types of their parameters
- A constructor's intended purpose is to initialize a new object's data members
- If no constructor function is written, the compiler supplies a default constructor
- In addition to initialization, a constructor can perform other tasks when it is called

Constructor Functions (continued)

- General format of a constructor includes:
  - The same name as the class to which it belongs
  - No return type (not even `void`)
  - A constructor that does not require arguments is called the **default constructor**

```
className::className(parameter list)
{
   // function body
}
```

Refer to page 633 for more explanations and examples

Calling Constructors

- Constructors are called when an object is created
- Declaration can be made in a variety of ways

```
Date c(4,1,2013);
Date c = Date(4,1,2013);
Date c = 8; // similar to above with
            // defaults for day and
            // year
```

- An object should never be declared with empty parentheses

```
Date a();
```

  - Not the same as the declaration `Date a;`
  - Does not result in an object being created

Refer to page 635 for more explanations and examples

Overloaded and Inline Constructors

- Primary difference between a constructor and other user-written functions is how the constructor is called
  - Constructors are called automatically each time an object is created
  - Most other functions must be called explicitly by name
- Inline functions are functions defined in the class declaration section

Destructors

- **Destructor functions**: Counterpart to the constructor functions
- Destructors:
  - Are functions with the same name as constructors but are preceded with a tilde (~)
    - For the `Date` class the destructor name is `~Date()`
  - Take no parameters and return no values
- There can only be one destructor per class

Destructors (continued)

- Destructors:
  - Called automatically when an object goes out of existence
  - Clean up any undesirable effects the object might leave, such as releasing memory stored in a pointer

Accessor Functions

- An **accessor function** provides a means for reporting on an object's state
- Conventionally called **get()** functions
- Each class should provide a complete set of accessor functions
- Accessor functions are extremely important because they provide a means of retrieving and displaying an object's private data values

```
double getReal() {return realPart;}              // inline accessor
double getImaginary() {return imaginaryPart;}    // inline accessor
```

Mutator Functions

- A **mutator function** provides a means for changing an object's data member
- Conventionally called **set()** functions
- A class can contain multiple mutators, as long as each one has a unique name or parameter list

```
void setReal(double rl) {realPart = rl;}            // inline mutator
void setImaginary(double im) {imaginaryPart = im;}  // inline mutator
```

Refer to page 638 for more explanations and examples

Sharing Functions

- Memory locations are allocated to an object only when the object is declared
- In this way, each object receives its own set of data members
- In contrast, only one copy of a member function is created, which comes into existence when the function is defined

Sharing Functions (continued)

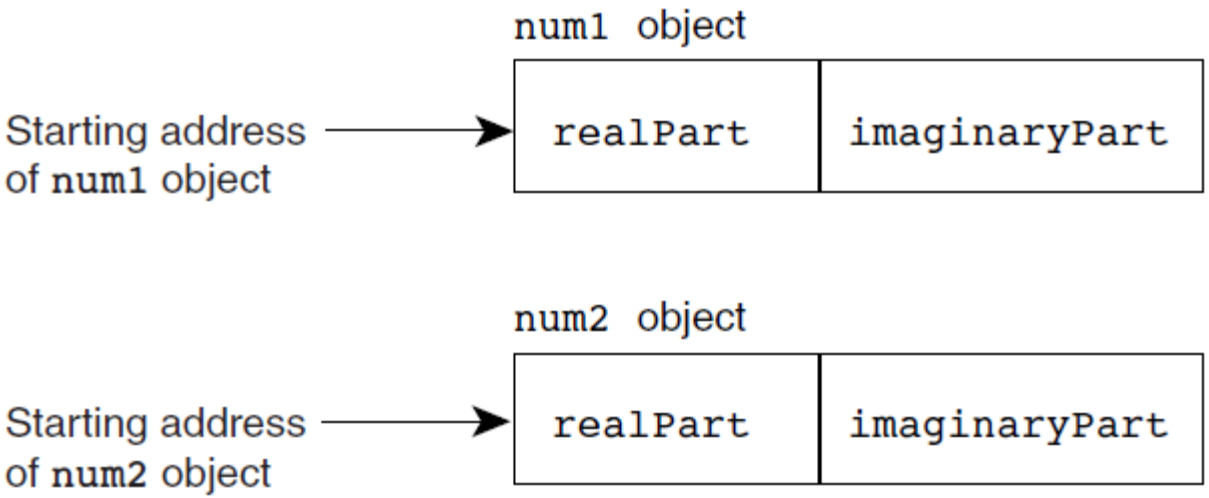

Figure 11.1 Storing two `Complex` objects in memory

Sharing Functions (continued)

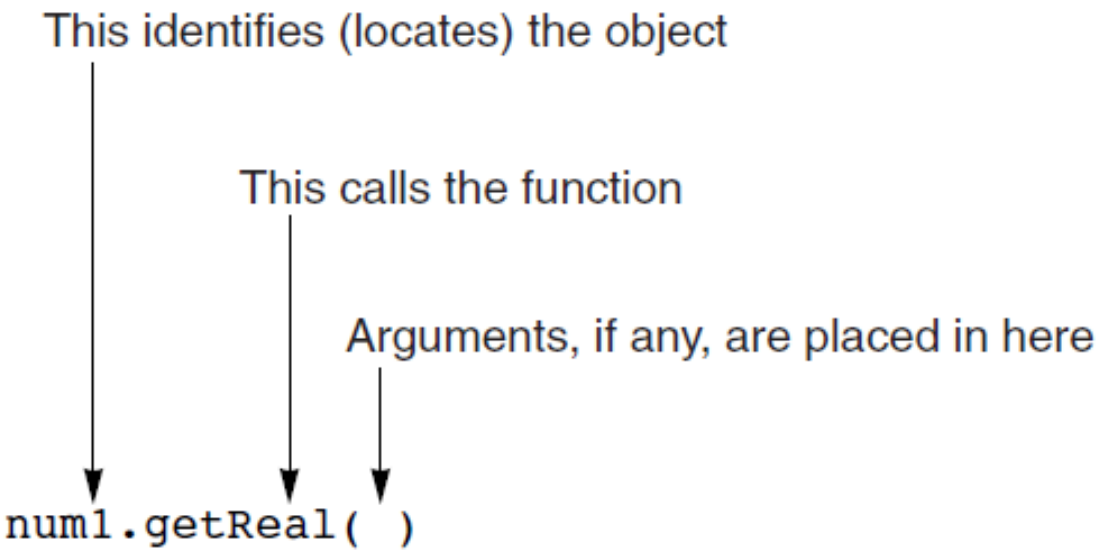

Figure 11.2 Calling a member function

Refer to page 640 for more explanations and examples

The `this` Pointer

- Two questions at this point are as follows:
  - How is this address passed to `getReal()`?
  - Where is this address stored?
- Each member function actually receives an extra argument that's the address of an object
- When a function is called, the calling object's address is passed to it and stored in the function's `this` pointer

The `this` Pointer (continued)

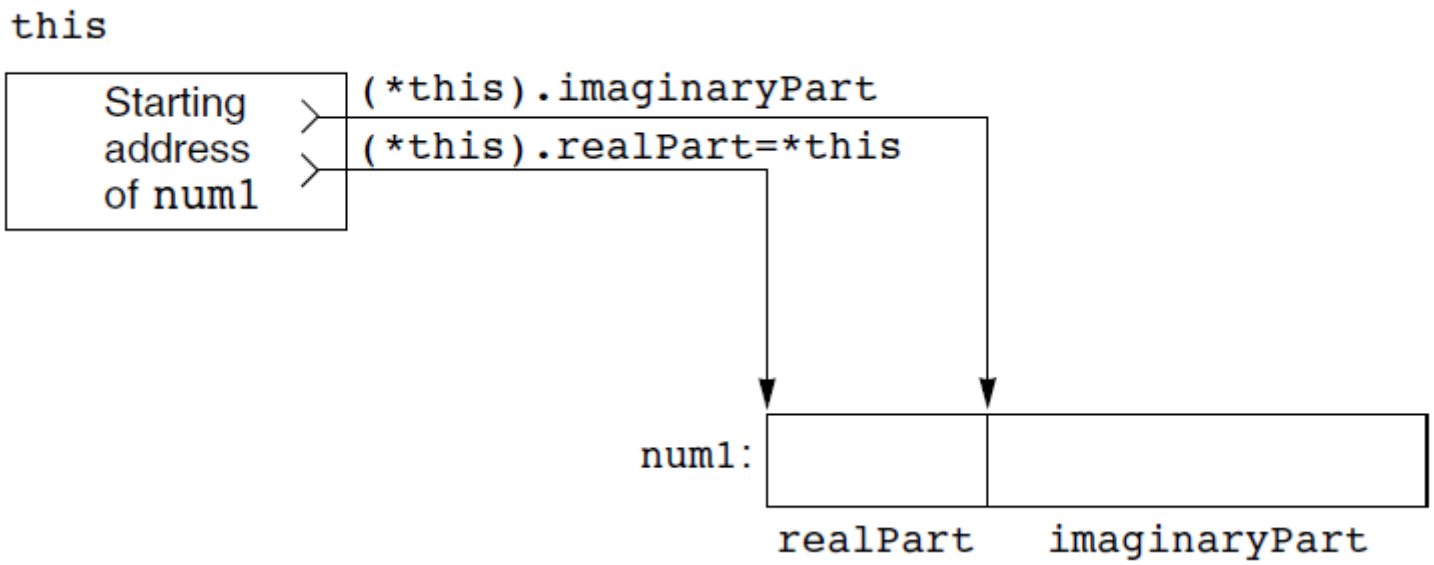

Figure 11.3 A pointer can be used to access object members

Refer to page 641 for more explanations and examples

The `this` Pointer (continued)

- The expression *`(*pointer).dataMember`* can always be replaced with the notation *`pointer->dataMember`*

```
void assignNewValues(double real, double imag)
  {realPart = real; imaginaryPart = imag;}
```

can be written as follows:

```
void assignNewValues(double real, double imag)
  {this->realPart = real; this->imaginaryPart = imag;}
```

Base/Member Initialization

- True initialization has no reliance on assignment
- C++ makes initialization possible with base/member initialization list syntax
- Initialization list syntax is only available in constructor functions

```
ClassName(argument list) : list of data
members( initializing values) {}
```

Refer to pages 642,643 for more explanations and examples

Adding Class Functions

- Most classes typically require additional functions
- In C++, there are two basic means of supplying these additional capabilities:
  - Construct class functions in a similar manner as mutator and accessor functions
  - Construct class functions that use conventional operator symbols, such as =, *, ==, >=, which are known as operator functions
- Both approaches can be implemented as member or friend functions

Member Functions

- Member functions can be added to a class by including their prototypes in the declaration section and providing code for the function in the implementation section or as an inline function
- The general syntax for each function header is:

```
returnType className::functionName(parameter list)
```

Refer to pages 646-649 for more explanations and examples

Operator Functions

- You can also use the operators C++ provides for built-in data types, such as +, −, ==, >=, and so on to construct class functions
  - These are referred to as **operator functions**
  - They are declared and implemented in the same manner as all functions, except the function name must use the syntax: `operator<symbol>`

Operator Functions

- Only the symbols in Table 11.1 can be used for user-defined purposes
- New operator symbols cannot be created
- Neither the precedence nor the associativity of the C++ operators can be modified

Refer to pages 650-656 for more explanations and examples

Assignment Operator

- The assignment operator, =, is the one operator that works with all classes without requiring an operator function
- For example, if `a` and `b` are objects constructed from the `Complex` class, the statement `a = b;` sets the values in `a`'s data members to their equivalent values in `b`'s data members
- This type of assignment is referred to as memberwise assignment

Refer to page 656 for more explanations and examples

Memberwise Assignment with Pointers

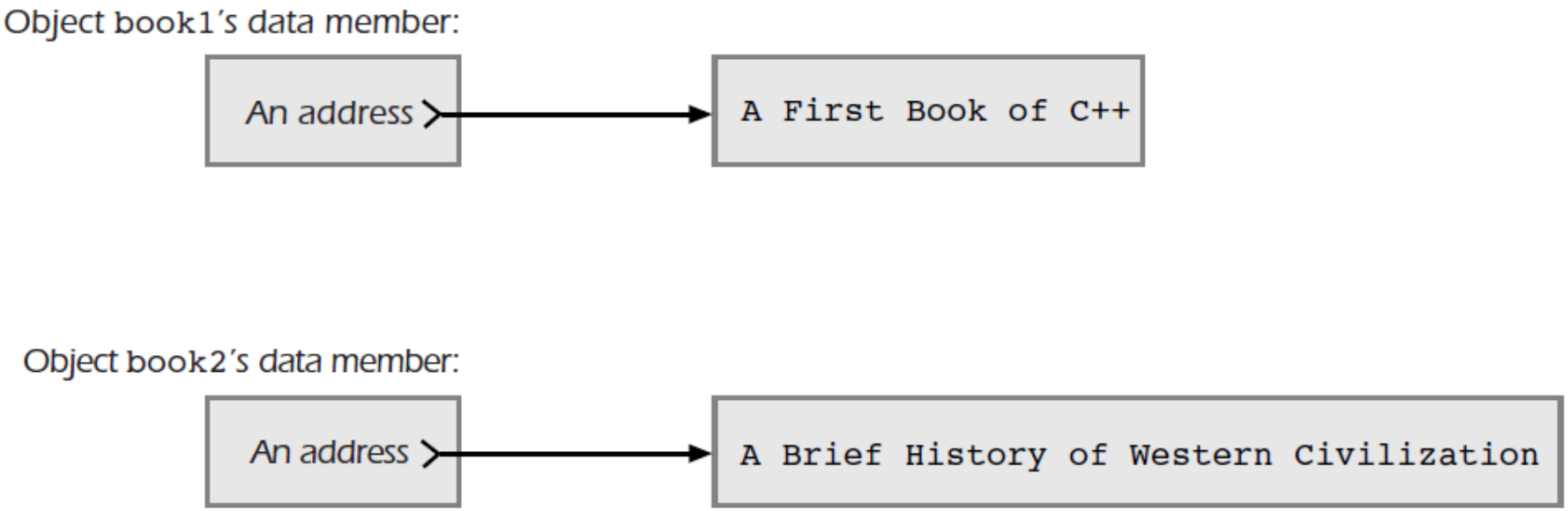

Figure 11.4 Two objects containing pointer data members

Refer to pages 657-659 for more explanations and examples

Copy Constructors

- One type of initialization that closely resembles assignment occurs in C++ when one object is initialized by using another object of the same class
- Examples:

```
Complex b = a;
Complex b(a);
```

  - The `b` object is initialized to the previously declared a object
  - The constructor performing this type of initialization is called a copy constructor

Refer to pages 660,661 for more explanations and examples

Friend Functions

- Private variables can be accessed and manipulated through a class's member functions

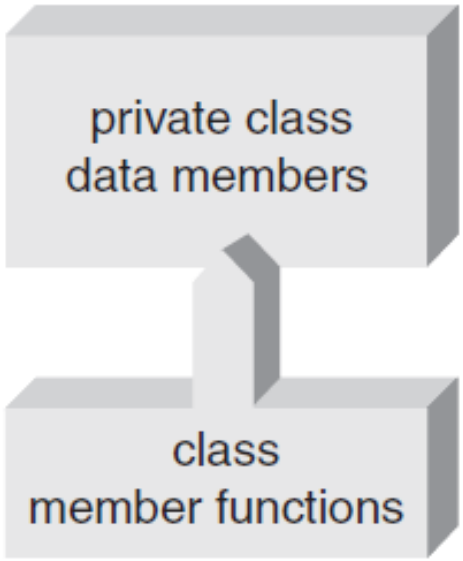

Figure 11.7a Direct access provided to member functions

Friend Functions (continued)

- External access to private functions can be granted through the friends list mechanism
- The friends list members are granted the same privileges as a class's member functions
- Nonmember functions in the list are called friend functions
- Friends list: Series of function prototype declarations preceded with the `friend` keyword

Refer to pages 662-665 for more explanations and examples

A Case Study: Constructing a `Date` Class

- Step 1: Analyze the problem: define operations
- Step 2: Develop a solution: define classes and data members
- Step 3: Code the solution: as seen in Class 11.1
- Step 4: Test and correct the program: testing the `Date` class entails testing and verifying each class function and operator function

Refer to pages 669-670 for more explanations and examples

A Closer Look: UML Class and Object Diagrams

- When solving any problem, it is often helpful to start by creating a diagram or map or devising a theoretical analogy for the problem you are trying to solve
- The first step in constructing an object-based program is developing an object-based model of the program

A Closer Look: UML Class and Object Diagrams (continued)

- **Unified Modeling Language (UML)** is a widely accepted technique for developing object oriented programs
  - A program-modeling language
  - Uses diagrams and techniques that are easy to understand and support all the features required to implement an object-oriented design

Class and Object Diagrams

- **Class diagrams** are used to describe classes and their relationships
- **Object diagrams** are used to describe objects and their relationships
- Both classes and objects are represented with a diagram consisting of a box
- In class diagrams, the class name is in bold text and centered at the top of the box
- In object diagrams, the object's name is also centered at the top of the box, underlined

Class and Object Diagrams (continued)

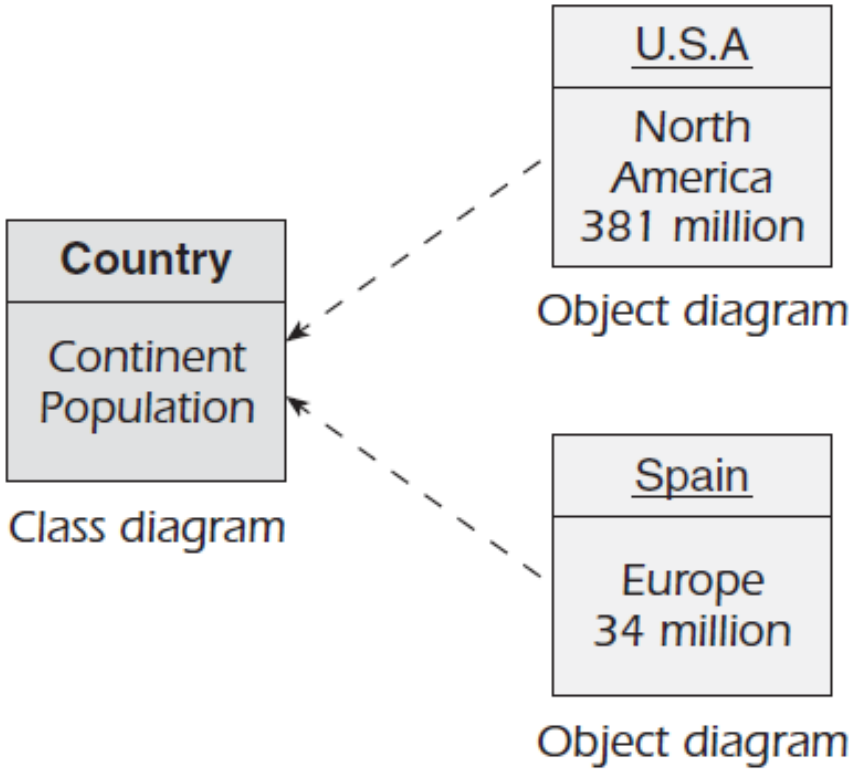

Figure 11.10 Including attributes in UML class and object diagrams

Class and Object Diagrams (continued)

- Visibility defines where an attribute can be seen
  - Private
    - Can be used on in its defining class
    - Cannot be accessed by other classes directly
    - Indicated by a minus (-) sign in front of attribute name
  - Public
    - Used in any other class
    - Indicated by a plus (+) sign in front of attribute name
  - Protected
    - Available to derived classes
    - Neither plus nor minus sign in front of attribute name

Class and Object Diagrams (continued)

- **Operations** are transformations that can be applied to attributes and are coded as C++ functions
- Operation names are listed below attributes and separated from them by a line

Class and Object Diagrams (continued)

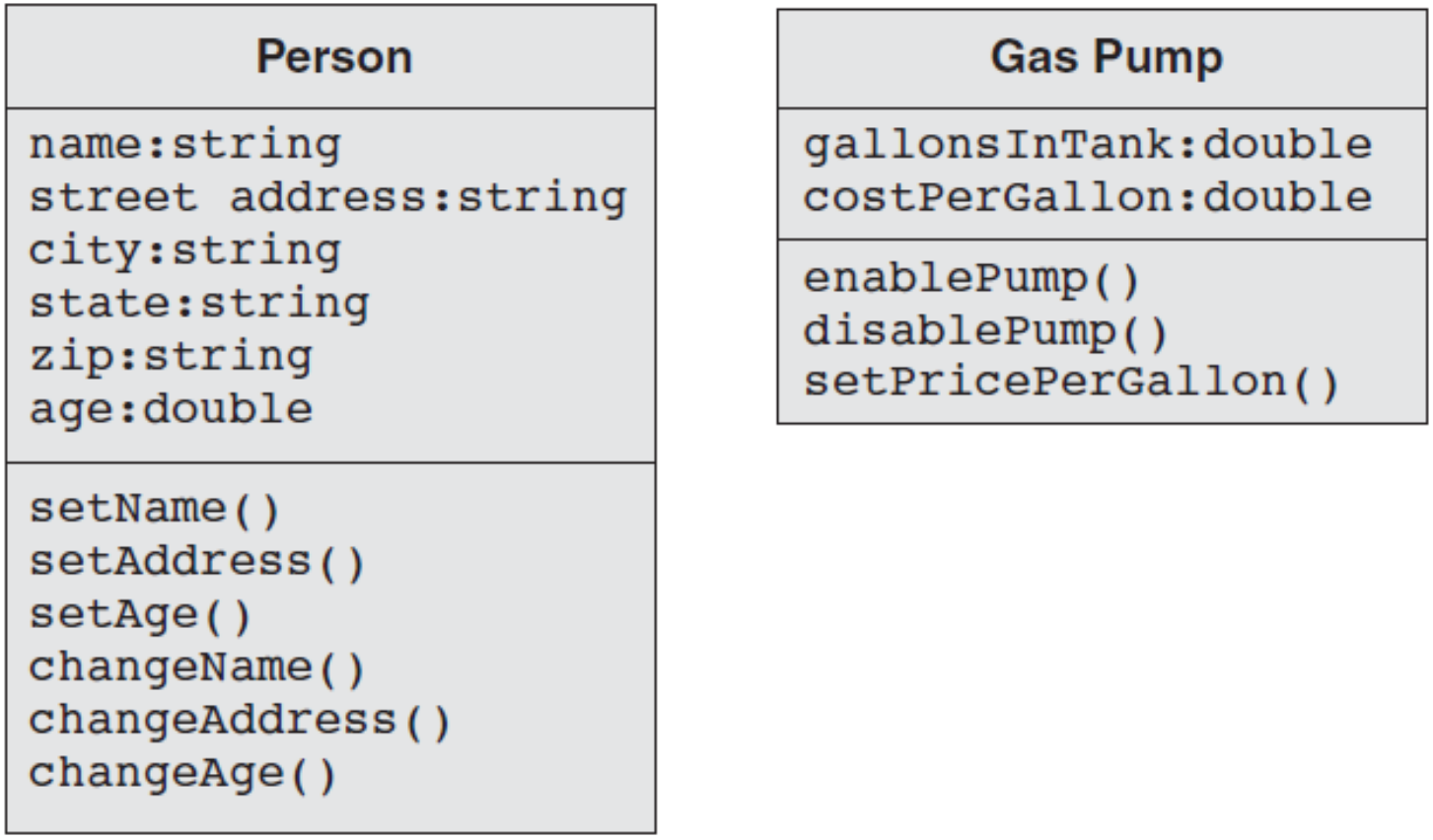

Figure 11.12 Including operations in class diagrams

Common Programming Errors

- Failing to terminate class declaration section with a semicolon
- Including the return type with the constructor's prototype or failing to include the return type with other the functions' prototypes
- Using same name for a data member as for a member function
- Defining more than one default constructor
- Forgetting to include the class name and scope operator, `::`, in the function header

Common Programming Errors (continued)

- Declaring an object with empty parentheses, as in `Complex a();`
  - The correct declaration is `Complex a;`
- Not defining an operator function's parameter as a reference to an object
- Redefining an overloaded operator to perform a function not indicated by its conventional meaning

Summary

- A class
  - Is a programmer-defined data type
  - Consists of a declaration and implementation section
- Class functions can be written inline or included in the class implementation section
- A constructor function is a special function that is called automatically each time an object is declared
  - If no constructor is declared, the compiler supplies a default

Summary (continued)

- Default constructor is the term for any constructor that does not require arguments
  - Each class can have only one default constructor
- Constructors can be overloaded
- A destructor function is called each time an object goes out of scope
- User-defined operators can be constructed for classes by using operator functions
- A nonmember function can access a class's private data members if it is granted friend status by the class

Homework

- P681 exercises 2, 3