Chapter Six

Class Definitions and Member Functions

Summing Up

Functions hide details, can be called many times, can be reused in other programs, and help in the design of larger programs. Each function performs a well-defined service.

Coming Up

When a function belongs to a class, it becomes a class member function. Class member functions have a lot in common with their nonmember cousins. Chapter 6 presents an introduction to C++ class definition and member function implementations. You will learn to read and understand classes by their definitions—the collection of member function headings (the interface) and data members (the state). In the second part of this chapter, you will learn to implement class member functions. You will also see a few appropriate object-oriented design heuristics that help explain why classes are designed the way they are. After studying this chapter, you will be able to

- read and understand class definitions (interface and state)
  Exercises and projects to reinforce reading new class interfaces
- implement class member functions using existing class definitions
- apply some object-oriented design heuristics
  Exercises and projects to reinforce member function implementation

---

This chapter could be studied after Chapter 10, "Vectors." Some projects and a few subsections marked with a Chapter 6 prerequisite would have to be skipped. This provides the flexibility to learn classes early, late, or not at all. This textbook is designed to accommodate all three approaches.

6.1 The Interface

Abstraction refers to the practice of using and understanding something without full knowledge of its implementation. Abstraction allows the programmer using a class to concentrate on the data characteristics and the messages that manipulate state. For example, a programmer using the `string` class need not know the details of the internal data representation nor how those operations are implemented in the hardware and software. The programmer can concentrate on the set of allowable messages—the *interface*.

This chapter presents some implementation issues that so far have been hidden. In the first part of this chapter, the `bankAccount` class will be studied at the implementation-detail level. However, before examining the physical side of class design let's consider some of the design decisions that were made for this textbook's `bankAccount` class.

6.1.1 Design Decisions with the `bankAccount` Class

All `bankAccount` objects have four allowable operations: `deposit`, `withdraw`, `balance`, and `name`. There could have been more, or there could have been less. The member functions for `bankAccount` were chosen to keep the class simple and to provide a collection of operations that are relatively easy to relate to. A compromise was made. The design decisions were influenced by the context—a first example of a C++ class used in a particular domain—the area of banking.

The `bankAccount` member functions that make up the interface are only a subset of the operations named by students who were asked this question: What should we be able to do with bank accounts? The data members are also a subset of the operations named by students who were asked this question: What should bank accounts know about themselves?

Many additional operations that were recognized by students—`transfer`, `applyInterest`, `printMonthlyStatement`—and many additional data members—type of account, record of transactions, address, social security number, and mother's maiden name—were not included. The design of these classes was affected by the intention of keeping these objects as simple as possible while retaining some realism. However, a group of object-oriented designers developing large-scale applications in the banking domain would likely retain many of the operations and attributes recognized by students. There is rarely one single design that is correct for all circumstances.

Designing anything requires making decisions in an effort to make the thing "good." Good might mean having a software component that is easily maintainable; it might mean classes that can be reused in other applications; or it might mean a system that is very robust—one that can recover from almost any disastrous event. Good might mean a design that results in something that is easier to use, prettier, etc. There is rarely ever a single perfect design. There are usually trade-offs. Design is an iterative process that evolves with time. Design is influenced by personal opinion, evolving research, the domain (banking, information systems, process control, engineering, for example), and a variety of other influences. Fortunately, there are design heuristics (guidelines) to show the way, a few of which are presented later.

Let's now turn to the construct that captures many of these design decisions in object-oriented software development—the class definition.

6.2 Class Definitions

The classes of objects under study—`ostream` (`cout`), `istream` (`cin`), `string`, `int`, and `double`—are building blocks of larger programs. However, programs typically require many other classes. They may be standard classes, classes that are bought off the shelf, or other classes that must be designed and implemented by the programming team.

Because it is difficult to have mastery of all classes in a large project, this section provides some general techniques for understanding unfamiliar classes. The knowledge attained here also provides experience with the major component of object-oriented software development—the class.

This process begins with learning to read class definitions. You will also implement member functions and add new operations to existing classes. This approach has the added benefit of making it easier to design and implement new classes of your own. You will not be asked to design new classes until Chapters 12 and 13, when you design classes after discovering the need for them during analysis.

A *class definition* lists member functions after the keyword `public:`. This set of operations represents the class interface. The class definition also lists the *data members*—the object declarations after `private:`. This set of data members represents the state of the objects.

A class definition provides a lot of information. A class definition stresses the *what,* not the *how.* It lists the messages understood by the objects. It specifies the number and type of arguments required when sending a message to one of the objects. When documented with preconditions, postconditions, and example messages,

a class definition also explains how to use instances of the class. The documentation may provide other pertinent information. All of these things allow the programmer to use objects of the class without knowing the details of the implementation.

GENERAL FORM 6.1. *Class definition*

```
class class-name {
public:   // MEMBER FUNCTIONS (the interface)
//--constructors
   class-name() ;                    // Default constructor
   class-name(parameter-list) ;      // Another constructor
//--modifiers
  function-heading ;                 // Member function that modifies the state
  function-heading ;                 // Member function that modifies the state
//--accessors
  function-heading const;            // Member function that accesses the state
  function-heading const;            // Member function that accesses the state
  ...
private:   // STATE
  object-declaration                 // Data member
  object-declaration                 // Data member
  ...
} ;
```

6.2.1 An Example Definition: The `bankAccount` Class

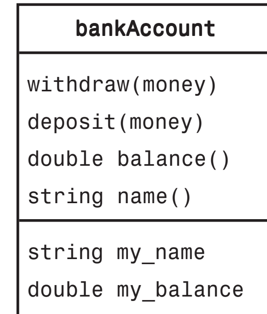
Now, let's get down to a concrete, familiar example. You are probably familiar with your own bank account. Recall that a `bankAccount` object stores the data necessary to manipulate one account at a bank. Each `bankAccount` object stores some unique identification (`string my_name;` below) and an account balance (`double my_balance;` below). Operations in `bankAccount` include making deposits, making withdrawals, and accessing the current balance.

The data members in the private section represent the state. The data members maintained by every `bankAccount` object include `my_name` (a string) and `my_balance` (a double). Every instance of this `bankAccount` class stores its own private name and balance data (the object's state). In other words, every `bankAccount` object knows its own name and current balance.

CLASS DEFINITION: *bankAccount*

File baccount.h | Class Diagram

```
class bankAccount {
public:  // OPERATIONS

//--constructors

  bankAccount();
  // post: Construct a default bankAccount object

  bankAccount(string initName, double initBalance);
  // post: Construct with two arguments:
  //       bankAccount anAcct("Hall", 100.00);

//--modifiers

  void deposit(double depositAmount);
  // post: Credit depositAmount to the balance

  void withdraw(double withdrawalAmount);
  // post: Debit withdrawalAmount from the balance

//--accessors

  double balance() const;
  // post: Return this account's current balance

  string name() const;
  // post: Return this account's name

private:  // STATE
  string my_name;
  double my_balance;
};
```



Example messages in the context of a program provide a valuable resource for understanding the behavior of unfamiliar objects written by other programmers. The messages could be gathered together within the context of a program that uses every operation (class member function) listed in the class definition. Such a test driver acts as a summary of the entire interface.

The following program sends every possible message to anAcct. Additionally, a default object is constructed to illustrate the default state of bankAccount objects. Now there are two different classes of objects that represent the state of a bankAccount—a string name and a numeric balance.

```
// Send every possible message to one bankAccount object and show
// the default bankAccount state

#include <iostream>
using namespace std;
#include "baccount"  // For the bankAccount class

int main()
{
  bankAccount anAcct("Moss", 500.00);  // Construct an object
  bankAccount defaultAccount;  // Default name is "?name?".
                               // Default balance is 0.00.

  anAcct.withdraw(100.00);                          // Modify state
  anAcct.deposit(40.00);                            // Modify state
  cout << "Name: " << anAcct.name() << endl;        // Access state
  cout << "Balance: " << anAcct.balance() << endl;  // Access state

  cout << "Default Name: " << defaultAccount.name() << endl;
  cout << "Default Balance: " << defaultAccount.balance() << endl;

  return 0;
}
```

To test the success of the documentation, your ability to read class definitions, and the use of descriptive identifiers to properly represent the operations, you will be asked questions similar to these later on but with new classes. In fact, the first three programming projects require you to read new class definitions.

Self-Check

6-1 Write the output generated by the preceding program.

6-2 List the operations that make up the interface of the `bankAccount` class.

6-3 List the data members of this `bankAccount` class.

6-4 Use the class definition to answer the questions that follow:

-a What happens during a withdraw message when an argument of `-20.00` is passed to `withdrawalAmount`?

-b What happens during a withdraw message when an argument of `100.00` is passed to `withdrawalAmount` and the balance is `50.00`?

-c What happens during a withdraw message when an argument of `100.00` is passed to `withdrawalAmount` and the balance is `300.00`?

-d What happens during a deposit message when an argument of `-20.00` is passed to `deposit`?

-e What happens during a deposit message when an argument of `20.00` is passed to `deposit`?

6.2.2 Constructors

Most `bankAccount` function headings are similar to the nonmember function headings discussed earlier—they usually have return types and parameters. However, two member functions do not fall into this category. Did you notice something different about the two function headings named `bankAccount`?

First of all, the two `bankAccount::bankAccount` member functions have no return type. They also have the same name as the class! These special member functions are dubbed *constructors* because they are used to "build" objects. Constructors associate the object name with a portion of memory and initialize the data members of the object. Here are some examples of messages that result in execution of a class constructor:

```
grid aGrid(10, 10, 5, 4, east);
string name;
string aString("The initial value of this object");
bankAccount a;
bankAccount b("Katey Jo Delong", 10.00);
```

Object construction is different from other messages, both in semantics and in syntax. There is no "dotting" and you don't need parentheses for the default constructor.

General Form 6.2. *Calling a constructor (initializing objects)*

class-name *object-name* **;**

- or -

class-name *object-name* **(** *initial-state* **) ;**

The *class-name* is the name of its class. The *object-name* is any valid identifier and *initial-state* is represented by the arguments supplied in the client code. When

a constructor is encountered with arguments, the *initial-state* is passed to the object's data members. The state of the object is initialized. For example, the following code constructs a `bankAccount` object with an initial name of `"Patricia Patterson"` and an initial balance of `507.34`:

```
bankAccount one("Patricia Patterson", 507.34);
```

When another object is constructed like this:

```
bankAccount another("Bob Zimmerman", 437.05);
```

there exists a separate `bankAccount` object with its own name of `"Bob Zimmerman"` and its own initial balance of `437.05`. So the return values of these two messages would be `507.34` followed by `437.05`.

```
cout << one.balance() << " " << another.balance() << endl;
```

When the default constructor is used—as in the following two lines of code—the object is initialized to the default state appropriate for that class of objects:

```
bankAccount aDefaultAccount;
string name;
```

For example, the default `string` state is the null string, and for `bankAccount` the default name is `"?name?"` and the default initial balance is `0.00`—guaranteed, every time.

Self-Check

Use this class definition to answer the self-check questions that follow:

```
class libraryBook {
public:
//--constructors
  libraryBook();
  // post: Create a default libraryBook object
  // ex: libraryBook aBook;

  libraryBook(string initTitle, string initAuthor);
  // post: Initialize a libraryBook object
  // ex: libraryBook aBook("Patriot", "Clancy");

//--modifiers
  void borrowBook(string borrowersName);
  // post: Records the borrower's name
```

libraryBook
borrowBook (borrower) returnBook() string borrower
string my_author string my_title string my_borrower

```
  // ex:   aBook.borrowBook("Wilma Riveria");

  void returnBook();
  // post: The book becomes available
  // ex:   aBook.returnBook();

//--accessor
  string borrower() const;
  // post: Returns "not currently borrowed" if the book is
  //       available or the borrower's name if the book is out
  // ex:   aBook.returnBook();

private:
  string my_author;
  string my_title;
  string my_borrower;
};
```

6-5 What is the name of the class shown above?

6-6 Except for constructors, name all the operations.

6-7 What type of value is returned by `libraryBook::borrower`?

6-8 What type of value is returned by `libraryBook::borrowBook`?

6-9 What class of argument must be part of all `libraryBook::borrowBook` messages?

6-10 How many arguments are required to initialize one `libraryBook` object?

6-11 Initialize one `libraryBook` object using your favorite book and author.

6-12 Send the message that borrows your favorite book. Use your own name as the argument.

6-13 Write the message that reveals the borrower's name of your favorite book.

6-14 Without knowing the implementation of the class member functions, write the output generated by the following program. Assume the class is defined by including the file named `libbook`.

```
// Send every possible message to a libraryBook object
#include <iostream>   // For cout
using namespace std;
#include "libbook"    // For the libraryBook class
```

```
int main()
{
  libraryBook aBook("Little Drummer Girl", "John LeCarre");

  cout << aBook.borrower() << endl;
  aBook.borrowBook("Chris Miller");
  cout << aBook.borrower() << endl;
  aBook.returnBook();
  cout << aBook.borrower() << endl;

  return 0;
}
```

6.3 The State Object Pattern

Even though quite different in specific operations and state, `string`, `bankAccount`, and `libraryBook` objects have the following common characteristics:

- private data members store the state of the object
- constructors initialize the state
- some messages modify the state
- other messages allow access to the current state of the object

These commonalities guide the effective use of these and similar classes of objects. Later on, these commonalities will also help guide effective implementation of C++ classes. These common traits are a pattern, or more specifically they represent an object pattern as described by Eugene Wallingford [Wallingford 96]. An *object pattern* is a guide for designing and implementing new classes. In his paper, Wallingford describes several object patterns, the first of which is State Object, a pattern that describes objects that "maintain a body of data and provide suitable access to it by other objects and human users."

6.3.1 Using Constructors, Modifiers, and Accessors

Object patterns also help programmers understand how to use new objects. The constructors, modifiers, and accessors in the `public:` section of a class definition are the operations available to all instances of the class.

6.3.1.1 Constructors

Constructors are present for many reasons, including initializing the state of any instance of the class. As shown earlier, objects are initialized like this:

```
string aString("initial string");
// assert: State of aString is "initial string", length of 14

string aDefaultString;
// assert: State of aDefaultString is ""

bankAccount anAcct("Early Grey", 2150.67);
// assert: Early Grey has a starting balance of 2,150.67

bankAccount aDefaultAccount;
// assert: aDefaultAccount.name() would return "?name?" and
//         aDefaultAccount.balance() would return 0.0

libraryBook aBook("Pride and Prejudice", "Jane Austen");
// assert: aBook represents a classic book written by Jane Austen
```

6.3.1.2 Modifiers

Modifiers[1] modify the state of an object. Modifiers are part of the State Object pattern for a variety of reasons. Perhaps it's best to simply show some example messages that modify the state of an object.

```
aString.replace(1, 3, "NEW");
// assert: s2 is "iNEWial string"

g.move(5);
// assert: The mover is five spaces forward

anAcct.withdraw(50.00);
// assert: The balance of anAcct is 50.00 less

aBook.borrowBook("Fred Featherstone");
// assert: aBook's borrower has become Fred Featherstone
```

Sending a modifier message results in a change of state. Modifiers are not declared with const after the function heading—accessors are.

6.3.1.3 Accessors

Accessors are part of a state object simply because programmers often need to access the state of an object. An accessor message returns information related to the state of an object. An accessor may simply return the value of a data member as with `libraryBook::borrower` and `bankAccount::balance`. Accessors may also need to do some internal processing using the state of an object to return the information

1. Some computer scientists and software engineers use the word *mutator* rather than *modifier.*

(`employee::incomeTax`, for instance). Here are some example messages that access the state of objects:

```
s2.length()          // Return the number of characters in s2
g.row()              // Return the mover's current row
anEmployee.incomeTax() // Return income tax based on IRS tax tables
anAcct.balance()     // Return the current balance of anAcct
aBook.borrower()     // Return the borrower's name of aBook
```

6.3.2 Naming Conventions

Modifying operations are typically given a name that indicates the message will change the state of the object. This is easily accomplished if the designer of the class simply gives a descriptive name to the operation. The name should describe—as best as possible—what the operation actually does. Another way to help programmers who use a class to distinguish modifiers from accessors is to give the modifiers names that can be used as verbs:[2] `withdraw`, `deposit`, `borrowBook`, and `returnBook`, for example. The accessors are given names that can be used as nouns: `borrower` and `balance`. Considering that the constructor has the same name as the class, some guidelines are now established (as a pattern) for designing and reading class definitions. These three categories of operations—typical of state objects—can be distinguished by following these naming conventions:

Operation	Name
Constructor	Same name as the class
Modifier	Identifier name that could be used as a verb
Accessor	Identifier name that could be used as a noun

Above all, always try to use identifiers that describe what the object is. For example, don't use `x` as the name of the operation to withdraw money from a `bankAccount` (or `turnRight` to make the mover `turnLeft`).

Self-Check

6-15 What is meant when the `const` keyword is part of the function heading in a class definition?

2. Because many words can be used as verbs and nouns (*balance,* for instance), some programmers precede modifiers with `set` and accessors with `get`.

6-16	Which member functions have the same name as the class?
6-17	What do accessors do?
6-18	What do modifiers do?
6-19	What do constructors do?
6-20	What are the data members for?

6.4 public: or private:

One of the considerations in the design of a class is the placement of member functions and data members under the most appropriate access mode, either `public:` or `private:`. Whereas public members of a class can be called by a client (outside of the class), the scope of private members is limited to the class member functions. For example, the `bankAccount` data member named `my_balance` is only known to the member functions of the class. On the other hand, any member declared in the `public:` section of a class is known everywhere in the class and also in the block of source code where the object is declared (or globally, if constructed outside of a block).

Access Mode	Where Is the Member Known?
`public:`	In all class member functions and in the block of the client code where the object has been declared (in `main`, for instance)
`private:`	Only in class member functions

Although the data members representing state could have been declared under `public:`, it is highly recommended that all data members be declared under the `private:` access mode. There are several reasons for this.

The consistency helps simplify some design decisions. More importantly, when data members are made `private:`, the state can be modified only through a member function. This prevents client code from indiscriminately changing the state of objects. For example, it's impossible to accidentally make a credit like this:

```
bankAccount myAcct("Mine", 100.00);

// An error occurs: attempting to modify private data
myAcct.my_balance = myAcct.my_balance + 100000.00;  // <- ERROR
```

or a debit like this:

```
// An error occurs: attempting to modify private data
myAcct.my_balance = myAcct.my_balance - 100.00;
```

6.4.1 Separating Interface from Implementation

The practice of studying a class through its interface represents a principle in software engineering. It allows one to separate the interface from the implementation (the details of how the operations actually work). In C++, the completed member function implementations are often separated from the interface (the class definition) by placing them in separate files. Historically, class definitions have been kept in `.h` files with member function implementations in `.cpp` files (or `.cc` files in Unix). However, this is changing. Some programmers implement the member functions directly in the same file as the class definitions.

The convention used for the programming projects in this textbook is to separate the definition from the implementation by storing the class definition in a `.h` file and the member function implementation in a `.cpp` file. To simplify things, this textbook's disk contains additional `#include` files that `#include` the class definition from the proper `.h` file and the member function implementations from the proper `.cpp` file. For example, when a program includes a file like this,

```
#include "baccount"  // For the bankAccount class
```

additional *preprocessor directives* come into effect:

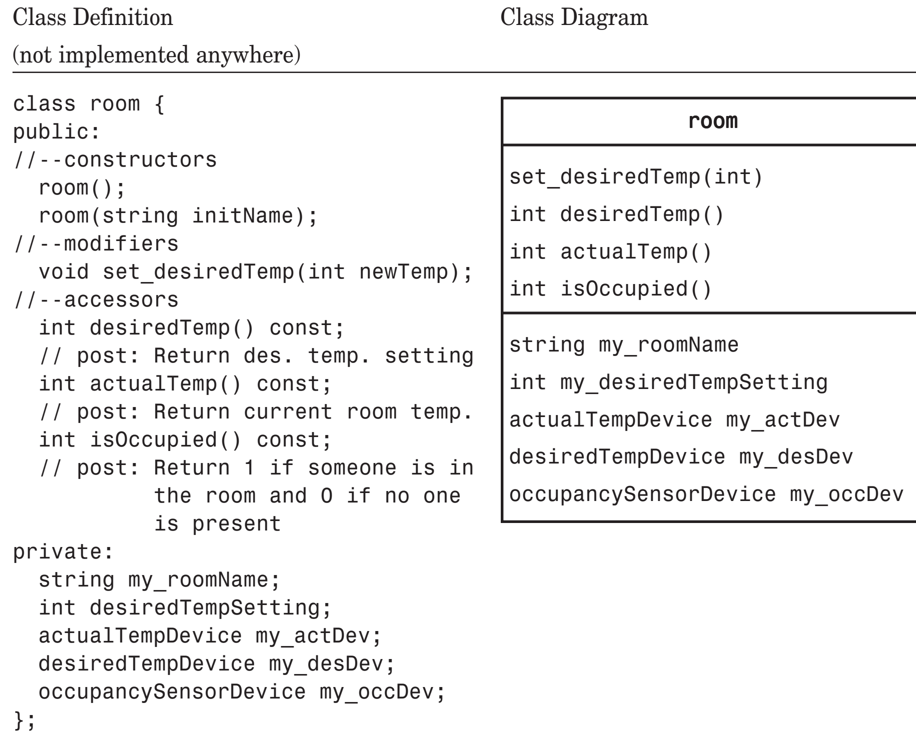
```
// File name: baccount
#ifndef _BACCOUNT_  // The first two lines mean the two #includes
#define _BACCOUNT_  // that follow will only be performed once
#include "baccount.h"    // For the bankAccount class definition
#include "baccount.cpp"  // For the member function implementations
#endif
```

Therefore, one `#include` actually `#includes` both parts of a class—the class definition and the member function implementations (discussed in the second part of this chapter). The first two lines, `#ifndef _BACCOUNT_` and `#define _BACCOUNT_`, ensure that the files are included only once. It reads like this: If the identifier `_BACCOUNT_` is not defined, define it. Then, at an attempt to `#include baccount` a second or third time, everything is skipped down to `#endif`. This prevents errors such as "class/function already defined."

Exercises

Use the `room` class definition from the domain of a home heating system to answer exercises 1–11.

Class Definition (not implemented anywhere)	Class Diagram



```
class room {
public:
//--constructors
  room();
  room(string initName);
//--modifiers
  void set_desiredTemp(int newTemp);
//--accessors
  int desiredTemp() const;
  // post: Return des. temp. setting
  int actualTemp() const;
  // post: Return current room temp.
  int isOccupied() const;
  // post: Return 1 if someone is in
           the room and 0 if no one
           is present
private:
  string my_roomName;
  int desiredTempSetting;
  actualTempDevice my_actDev;
  desiredTempDevice my_desDev;
  occupancySensorDevice my_occDev;
};
```


1. What is the name of the class?
2. Name all the operations besides the constructors.
3. How many data members are there?
4. `room::desiredTemp` returns what type of value?
5. Which operations can modify any `room` object?
6. Which operations do not modify `room` objects?
7. Construct an object named `kitchen` that has a `room` name of `kitchen`.

8. Set the desired temperature to 70 degrees Fahrenheit.

9. Write a statement that displays the `room`'s desired temperature.

10. Write a statement that displays the `room`'s actual temperature.

11. Write a message that returns the integer 1 if someone is in the `room`.

Use this class definition to answer exercises 12–19:

Class Definition
(not implemented anywhere)

```
class ATM {
public:
//--constructors
  ATM();
  ATM(int dollars);
//--modifiers
  void startUp();
  void getEnvelope();
  void dispenseCash();
//--accessor
  int dollarLevel() const;
private:
  cardReader my_cardReader;
  keyPad my_keyPad;
  cashDispenser my_cashDispenser;
  receiptPrinter my_receiptPrinter;
  transactionList my_transactionList;
};
```

Class Diagram

ATM
startUp() getEnvelope() dispenseCash() int dollarLevel()
cardReader my_cardReader keyPad my_keyPad cashDispenser my_cashDispenser receiptPrinter my_receiptPrinter transactionList my_transactionList

12. What is the name of the class?

13. How may constructors are there?

14. Name all operations other than the constructors named `ATM`.

15. How many data members does this class have?

16. What type of value is returned by `ATM::dollarLevel`?

17. Which operations modify any `ATM` object?

18. Which operations only access the state of any `ATM` object?

19. Write a test driver that sends every possible message to one `ATM` object. Assume the class becomes available with `#include "ATM"`.

Programming Tips

1. Be sure to `#include "weekemp"` for the first three programming projects.

The first three programming projects (6A, 6B, and 6C) require the definition of the `weeklyEmp` class from the file `weekemp.h`. They also require the member functions from `weekemp.cpp`. If these two files are in your working folder (directory), simply place this at the beginning of your program:

```
#include "weekemp"  // #include weekemp.h and weekemp.cpp
```

2. Here is your chance to read a new class definition.

The first three programming projects (6A, 6B, and 6C) require the `weeklyEmp` class. It is repeated here for your convenience, followed by a summary class diagram.

```
// You can include this class definition in your program with
// #include "weekemp" which includes the class definition of weekemp.h
// and member function implementations of weekemp.cpp
class weeklyEmp {
public:

//--constructors
  weeklyEmp();

  weeklyEmp(string initName,
            double initHours,
            double initRate,
            int initExemptions,
            string initFilingStatus);
  // post: A weeklyEmp object is initialized with five arguments:
  //       weeklyEmp anEmp("Hall, Rob", 40.0, 9.75, 3, "M");
  //       The fourth argument must be in the range of 0 to 99. The
  //       last argument is either "M" for married or "S" for single.

//--modifiers

  void set_hours(double thisWeeksHours);
  // post: Set the hours worked for a given week

  void set_rate(double thisWeeksRate);
  // post: Change the employee's hourly rate of pay
```

```
//--accessors

  double grossPay() const;
  // post: Return gross pay with overtime

  double incomeTax() const;
  // post: Return the federal income tax

  double FICATax() const;
  // post: Return the social security tax

  string name() const;
  // post: Return the employee's name

 private:
  string my_name;
  double my_hours;
  double my_rate;
  int my_exemptions;
  string my_filingStatus;
};
```

weeklyEmp
set_hours(newHours) set_rate(newRate) double grossPay() double incomeTax() double FICATax() string name()
string my_name double my_hours double my_rate int my_exemptions string my_filingStatus

Programming Projects

6A weeklyEmp

Write a complete C++ program that will

- initialize one `weeklyEmp` object (see the `weeklyEmp` class definition above)
- display the `weeklyEmp` object's gross pay
- display the `weeklyEmp` object's income tax

6B Payroll for One

Write a payroll program that produces a simple paycheck for one weeklyEmp object. The program must input the employee's name, filing status (S or M), number of exemptions, hourly rate of pay, and hours worked. The paycheck should be formatted to always show currency amounts rounded to the nearest hundredth. The output must include the employee's name, the gross pay, all taxes, and the net pay. Here is one sample dialogue and an example paycheck with headings:

```
Name: Lucas
Hourly Rate: 10.00
Hours Worked: 40
```

```
Exemptions: 2
S)ingle M)arried: M

Employee: Lucas
  Gross  Income    FICA      Net
    Pay     Tax     Tax      Pay
 ======  ======  ======  =======
 400.00   ??.??   ??.??  ????.??
```

6C Raise and Overtime

Write a C++ program that will:

- initialize a `weeklyEmp` object who is married (use `"M"` as the fifth argument in the constructor), has three exemptions (the fourth argument), and worked 40.0 hours at $10.00 per hour
- display the gross pay and income tax
- give the employee a raise to $11.50 per hour
- display the gross pay and income tax
- change hours worked to 42.0 (overtime)
- display the gross pay and income tax

Your output must look *exactly* like this (consider using a `void` function named `show` that displays the output. Use `cout.width(9)` and the `decimals` function from `"compfun"`):

```
Gross Pay  Income Tax
=========  ==========
   400.00       32.51
   460.00       41.51
   494.50       46.69
```

6.5 Implementing Class Member Functions

Class member function implementations are similar to those of their nonmember relatives—with these differences:

- Class member functions implemented outside of the class definition must be qualified with the class name and the scope resolution operator `::`. This tells the compiler they are member functions of a particular class and as such they are allowed to directly reference the private data members.
- The constructors are class member functions with the same name as the class and they do not have a return type.

The relatively familiar `bankAccount` class will be used to demonstrate member function implementations.

6.5.1 Implementing Constructors

A constructor is a special member function that always has the same name as the class. It never has a return type. Although member functions can be defined within a class definition, this textbook uses the software engineering principle of separating interface from implementation by implementing the member functions in a separate file. In this case, the member functions must begin with *class-name* `::`.

6.5.1.1 Default Constructor

The programmer can specify whatever default state seems appropriate in the default constructor. The following code implements the default constructor—a constructor with zero parameters:

```
// Member function implementations are in the separate file named
// baccount.cpp

bankAccount::bankAccount()
{ // post: This object has the default state
  my_name = "?name?";
  my_balance = 0.0;
}
```

Why have default constructors like this? There are several reasons.

- They are required to have collections of objects (see Chapter 10, "Vectors").
- They guarantee initialization to a specific state. Programmers always know what to expect (more vivid examples are yet to come).

This default constructor is the class member function that is executed whenever a bankAccount is declared and initialized like this:

```
bankAccount anAcct;
                                    // Output:
cout << anAcct.name() << endl;     // ?name?
cout << anAcct.balance() << endl;  // 0
```

6.5.1.2 A Constructor with Parameters

The following code implements the two-parameter constructor:

```
bankAccount::bankAccount(string initName,  double initBalance)
{
  my_name = initName;
  my_balance = initBalance;
}
```

This is the function that executes whenever a bankAccount is initialized with two arguments (a string followed by a number).

In the following code, the account name `"Stein"` is passed to the parameter `initName`, which in turn is assigned to the private data member `my_name`. The starting balance of `250.55` is also passed to the parameter named `initBalance`, which in turn is assigned to the private data member `my_balance`.

```
// Call the two-parameter constructor to initialize
// anInitializedAccount

bankAccount anInitializedAccount("Stein", 250.55);
                                                  // Output:
cout << anInitializedAccount.name() << endl;      // Stein
cout << anInitializedAccount.balance() << endl;   // 250.55
```

After an object is constructed, the object knows how to respond to its accessor messages. That's because every object is responsible for knowing its own name and its own balance.

The major difference between implementing class member functions and their nonmember cousins is this: Class member functions must be preceded with the class name and the `::` operator. For example, the bankAccount constructor is preceded with `bankAccount::` to inform the compiler that it is a member function and as such, has access to the object's private data members. Failure to add `bankAccount::` results in a nonmember function that can reference neither `my_balance` nor `my_name`. For example, the compiler will generate error messages at both attempts to use the private data members (`my_name` and `my_balance`) because `bankAccount::` is missing:

```
bankAccount(string initName, double initBalance)  // <-- WHOOPS
{
  my_name = initName;        // ERROR: my_name is not known
  my_balance = initBalance;  // ERROR: my_balance is not known
}
```

This scope error is due to the following fact:

SCOPE RULE FOR C++ CLASSES

The scope of private members is limited to the class member functions.

So remember to precede a class member function implementation with the class to which it belongs and `::`. This defines the function as a class member function that can access the private data members. A member function can do whatever it has to do with the state.

6.5.1.3 Function Overloading

You may be wondering how there could be two constructors with the same name. Through a technique known as *function overloading,* more than one function with the same name is allowed to exist. However, there has to be something that distinguishes two functions with the same name. One of these distinguishing characteristics is having a different number of parameters. Function overloading allows the programmer to have a default constructor with zero parameters at the same time as constructors with one or more parameters. This is allowed if the second function of the same name has a different number of arguments. In other words C++ can distinguish between these two constructor function headings inside the class definition:

```
class bankAccount {
public:
  bankAccount();  // Zero parameters
  bankAccount(string initName, double initBalance);  // Two
                                                      // parameters
  // . . .
private:
  // . . .
};
```

Both constructors (named `bankAccount`) are used in the following code:

```
bankAccount a;  // Initialize object with the default constructor
bankAccount b("Bob", 678.99);  // Initialize two parameters
```

6.5.2 Implementing Modifiers

A member function may either modify the state or access the state of an instance of the class. For example, consider `bankAccount::deposit`, which modifies the private data member named `my_balance`.

```
void bankAccount::deposit(double depositAmount)
{
```

```
  my_balance = my_balance + depositAmount;
}
```

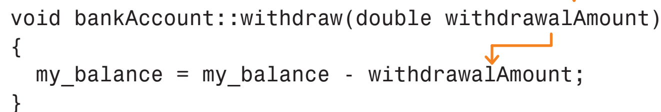
When the following `deposit` message is sent, the argument (`157.42`) is copied by value to the parameter `depositAmount`, which is then added to this object's balance:

```
anAcct.deposit(157.42);
```

Notice that the function heading matches the heading in the class definition.[3] Specifically, the return type of `bankAccount::deposit` is `void` and there is one double argument.

```
class bankAccount {
public:
// . . .
//--modifiers
  void deposit(double depositAmount);
  void withdraw(double withdrawalAmount);
private:
// . . .
};
```

The `bankAccount::withdraw` function is another modifying member function that changes the state of a bankAccount object. Specifically, a withdraw message deducts `withdrawalAmount` from `my_balance`:



```
void bankAccount::withdraw(double withdrawalAmount)
{
  my_balance = my_balance - withdrawalAmount;
}
```

When the following `withdraw` message is sent, the argument (`50.00`) is copied by value to the parameter `withdrawalAmount`, which would then be subtracted from `anAcct`'s balance:

```
anAcct.withdraw(50.00);
```

As you are implementing class member functions, make sure all function headings match the appropriate function heading in the class definition. Your implemen-

---

3. Once again, semicolon placement can create havoc. Remember that the class definition (in `.h` files) terminates function headings with the semicolon. However, the member function implementations (in `.cpp` files) do not. Also, the qualifications of *class-name*`::` must be added to implement the class member function.

tations, usually stored in a different file, must have the same exact return type, function name, and number and class of parameters as shown in the class definition. And don't forget to eliminate the semicolon and to add *class-name*`::` to the implementation.

It should be noted here that there could be much more processing within a class member function. For example, programming project 7N, "Maintain`weeklyEmp`," asks you to update `weeklyEmp::incomeTax`. This class member function does not return any of the values stored in the `private:` access section. Instead, some of the state is used to help compute the U.S. income tax based on the Internal Revenue Service tax tables. Incidentally, the code is approximately 38 lines long and fairly complex. The member function implementations in this chapter have been kept intentionally simple during this introduction to class definitions and member function implementations.

6.5.3 Implementing Accessors

The state pattern indicates that functions are provided to allow access to the state of the objects. Some accessor functions simply return the value of individual data members.

```
string bankAccount::name() const
{
  return my_name;
}

double bankAccount::balance() const
{
  return my_balance;
}
```

Because these accessing functions in the class definition have the keyword `const`, the implementation must also include `const` after the function heading and before the block beginning with `{`. The keyword `const` denotes a member function that does not modify state. If you examine the accessor implementations above, you'll notice nothing is changed in the block. `name` and `balance` simply return values. On the other hand, go back to the modifiers `withdraw` and `deposit` to observe that both modifiers modify the balance.

Also remember to make sure all member function headings exactly match the headings in the class definitions (without `;`). And remember to type the class name and `::` before the class member function name in the `.cpp` files.

To summarize, here is the implementation of all of the `bankAccount` member functions:

```
// File name: baccount.cpp
#include "baccount.h"       // Allows for separate compilation

//--constructors
bankAccount::bankAccount()  // Default constructor
{
  my_name = "?name?";
  my_balance = 0.0;
}

bankAccount::bankAccount(string initName, double initBalance)
{
  my_name = initName;
  my_balance = initBalance;
}

//--modifiers
void bankAccount::deposit(double depositAmount)
{
  my_balance = my_balance + depositAmount;
}

void bankAccount::withdraw(double withdrawalAmount)
{
  my_balance = my_balance - withdrawalAmount;
}

//--accessors
double bankAccount::balance() const
{
  return my_balance;
}

string bankAccount::name() const
{
  return my_name;
}
```

Self-Check

6-21 How does a function implementation become a member of a class?

6-22 Can class member functions reference the private data members?

6-23 Can nonmember functions reference private data members?

6.6 Object-Oriented Design Heuristics

One particular object-oriented design decision involves determining where to place the data members that store object state. More specifically, since this text uses C++ as the implementation language, the designer has to decide if data member functions go in the `public:` or the `private:` section of a C++ class. The following design heuristic[4] states that a good design protects object state from the outside world:

OBJECT-ORIENTED DESIGN HEURISTIC 6.1

All data should be hidden within its class.

Although data members could be `public:`, the convention used in this text—and in any well-designed class—is this: Hide the data members. C++ data members are easily hidden when declared in the `private:` section of the class definition. This simplifies some design decisions that must be made in any new classes that you develop.

A `private:` data member can only be modified or accessed through messages. This prevents users of the class from indiscriminately changing certain data such as an account balance. The state of an object can be protected from accidental or improper alteration. With data members declared in the `private:` section, the state of any object can only be altered through a message. It becomes impossible to accidentally make a false debit like this:

```
// Compiletime error: attempt to modify private data
// If my_balance is public:, what is the new balance?

myAcct.my_balance = myAcct.my_balance - myAcct.my_balance;
```

However, if `my_balance` had been declared in the `public:` section, the compiler would not protest. The resulting program would allow you to destroy the state of any object. The hidden balance is more properly modified only when the transaction is allowed according to some policy. What happens, for instance, if a withdrawal amount exceeds the account balance in a withdraw message? Some accounts allow this by transferring money from a savings account. Other bank accounts may generate loans in increments of $100.00.

---

4. A design heuristic is a guideline intended to help produce good object-oriented programs. Some design heuristics (including this one) are from the book *Object-Oriented Design Heuristics*, by Arthur J. Riel [Riel 96], who catalogued and/or developed 60 design heuristics in all.

With my_balance declared in the private: access section, users of the class must instead send a withdraw message. The client code relies on the bankAccount to determine if the withdrawal is to be allowed. Perhaps the bankAccount object will ask some other object if the withdrawal is to be allowed. Perhaps it delegates authority to some unseen bankManager object. Perhaps the bankAccount object itself can decide what to do. Although this text's implementation of bankAccount doesn't do much, real-world withdrawals do.

By hiding data and other details, all credits and debits must "go through the proper channels." This might be quite complex. For example, each withdrawal or deposit may be recorded in a transaction file to help prepare monthly statements for each bankAccount. The withdraw and deposit operations may have additional processing to prevent unauthorized credits and debits. Part of the hidden red tape might include manual verification of a deposit or a check-clearing operation at the host bank; there may be some sort of human or computer intervention before any credit is actually made. Such additional processing and protection within the deposit and withdraw operations help give bankAccount a "safer" design. Because all hidden processing and protection is easily circumvented when data members are exposed in the public: section, the object designer must enforce proper object use and protection by hiding the data members.

6.6.1 Cohesion within a Class

This set of messages described in the class interface should be strongly related. A class stores data. The data should be strongly related. In fact, all elements of a class should have a persuasive affiliation with each other. These ideas relate to the preference for tight cohesion (solidarity, hanging together, adherence, unity) within a class. For example, don't expect a bankAccount object to understand the message areYouPreheated? This may be an appropriate message for an oven object, but certainly not for a bankAccount object. Here is one heuristic related to the desirable attribute of cohesion.

Object-Oriented Design Heuristic 6.2

Keep related data and behavior in one place.

The bankAccount class should hide certain policies such as handling withdrawal requests greater than the balance. The system's design improves when behavior and data combine to accomplish the withdrawal algorithm. This makes for nice clean messages from the client code, like this:

```
anAccount.withdraw(withdrawalAmount);
```

This client code relies on the bankAccount object to determine what should happen. The behavior should be built into the object that has the necessary data. Perhaps the algorithm allows a withdrawal amount greater than the balance—with the extra cash coming as a loan or as a transfer from a savings account. Even though the bankAccount class of this textbook does very little, a real bank account class might have eight different actions that are triggered for every withdrawal—all behind the scenes.

6.6.2 Why Are Accessors Const and Modifiers Not?

You may be wondering why const is added to function headings intended to access, rather than modify, the object's state. The answer has to do with the three different parameter modes.

When an object is passed by value or by reference to a function, that function can send any and all possible messages to that object. However, when the const reference parameter mode is utilized, the function promises not to change that object. In fact, it cannot.[5] To illustrate, consider the following function that will not compile—there is a compiletime error at the attempt to withdraw from the const object b. This is actually a good thing. The reason for using const parameters is to avoid accidental modification of the associated argument.

```
// Illustrate connection between member functions tagged as const
// functions and passing an object of that class as const parameter

#include <iostream>  // For cout and endl
using namespace std;
#include "baccount"  // For the bankAccount class

void display(const bankAccount & b)
{
// OKAY to send name and balance messages--they are declared const
   cout << "{ bankAccount: " << b.name()
        << ", $" << b.balance() << " }" << endl;

  // This modifying message to a nonconst member function was not
  // tagged as a const member, so this should be an error:
  b.withdraw(234.56);  // ERROR
```

5. This is true for most compilers; however, Turbo/Borland compilers allow const objects to be modified! Here is what Borland says: "This is an error, but was reduced to a warning to give existing programs a chance to work." Be careful with Borland compilers; const does not work as it should.

```
}

int main()
{
  bankAccount anAcct("Rita Jupe", 1234.56);

  display(anAcct);
  return 0;
}
```

This protection works fine for standard classes such as `string`. The same protection will only work with your new classes if care is taken to tag the accessors as const and leave the modifiers as nonconst.

A consistent use of const accessors allows the accessing messages to be sent to the const parameters. At the same time, by not using const with modifiers, a const parameter disallows modifying messages.

Only const messages are allowed on const parameters.

On the other hand, it is okay to send messages that do not modify the object. This safety net is possible only when the programmer diligently tags accessing class member functions with const and always remembers not to tag a modifier that way.

```
class bankAccount {
public:
//--modifiers
  void deposit(double depositAmount);  // No const for modifiers
  void withdraw(double withdrawalAmount);
//--accessors
  double balance() const;  // Use const on accessors
  string name() const;
// . . .
```

This leads to another design heuristic.

Object-Oriented Design Heuristic 6.3

Always declare accessor member functions as const.

Perhaps the biggest problem with this guideline is in remembering the heuristic. It is easily violated. You'll never know the ramifications until an instance of your class is passed as a const reference parameter. As another example, consider the `grid` class modifiers, which are nonconst, and some accessors, which are declared as const functions.

```
class grid {
public:
. . .
//--modifiers
  void move(int nMoves);
 . . .
//--accessors
  int row() const;
  int column() const;
  . . .
};
```

The presence of `const` tells the compiler to allow the message to be sent even for objects passed by const reference (`g` here):

```
void foo(const grid & g);
{
  cout << g.row() << endl;       // OKAY
  cout << g.nColumns() << endl;  // OKAY
  g.display();                   // OKAY
  g.move(1);                     // Compiletime ERROR
  g.pickUp();                    // Compiletime ERROR
}
```

On the other hand, the attempt to send nonconst function messages such as `grid::move` results in a compiletime error like these (more cryptic errors exist):

```
nonconst member function 'grid::move()' called for const object
```

- or -

```
attempt to modify a const object
```

Declaring accessors as const functions allows existing objects to be safely passed to a const parameter. However, it takes diligence to maintain the same safety net for the new classes that you write. Remember these two class design guidelines.

1. Modifiers should *not* be declared const so the compiler can catch attempts to modify const objects.
2. Accessors should be declared const so objects can be safely passed to const parameters and still allow nonmodifying messages.

It would be easier to completely ignore these rules, but the only way to get away with it would be to never pass objects to const parameters. This textbook uses const in a member function because it says something about whether or not a function

modifies the state of an object. And this is something object-oriented programmers must know about. The designer of the class must still decide if the message will modify an instance of the class or not.

Self-Check

6-24 Using the class definition of the `bankAccount` class, list the lines that cause errors in a standard C++ compiler (1, 2, 3, and/or 4).

```
#include <iostream>
using namespace std;
#include "baccount"  // For the bankAccount class

void check(const bankAccount & b, double amount)
{
  cout << b.name() << endl;    // 1
  b.deposit(45.00);            // 2
  b.withdraw(12344.00);        // 3
  cout << b.balance() << endl  // 4
}

int main()
{
  bankAccount myAcct("Me", 12345.00);
  check(myAcct, 50.00);
  return 0;
}
```

6-25 Does the interface of a class refer to its member functions or its data members?

6-26 Does the client code need to know the names of data members to use objects of the class?

6-27 Does client code need to know the names of member functions to use objects?

6-28 Describe the scope of the public members of a class.

6-29 Describe the scope of the private members of a class.

6-30 Give one justification for making the data members of a class private.

6-31 If the designer of a class changed the name `my_balance` to `myBalance`, would programs using `bankAccount` need to be changed?

6-32 If a designer changed the name of the `withdraw` message to `withdrawThisAmount` after the class was already in use by dozens of programs, would these dozens of programs need to be changed?

6-33 Who is responsible for knowing if a particular `libraryBook` is available for lending, the `libraryBook` or the program using `libraryBook`?

6-34 Should a `bankAccount` object understand the message `isThisBrakeLockingUp`?

6-35 Should a thermostat object know the current room temperature?

6-36 If an object is passed by value, which set of messages can be sent—modifiers, accessors, or both?

6-37 If an object is passed by reference (with `&`), which set of messages can be sent—modifiers, accessors, or both?

6-38 If an object is passed by const reference (`const grid & aGrid`), which set of messages can be sent—modifiers, accessors, or both?

6-39 Can a function change the state of an argument passed by value?

Chapter Summary

- Chapter 6 showed class definitions with a collection of function headings that represent the class interface. These are the message names that any object of the class will understand.
- A class definition lists
  - the class member functions with parameters and return types, collectively known as the interface
  - the data members, known collectively as the state
- Each object of a class may store many values, which may be of different classes. For example, each `bankAccount` object stores string data for the name and numeric data for the balance.
- The state pattern guides class design when the primary need for the object is to store state and provide adequate access to it. The state pattern in C++ recommends that the following items be included in a class definition:
  - a default constructor
  - a constructor to initialize objects with programmer-supplied state

    - modifying functions
    - accessor functions
    - private data members to store the state of every object
- Class constructors declare and initialize objects like this:

```
bankAccount anotherAccount("Calissario", 4320.10);
```

- Modifying class member functions changes the state of the object.
- Accessor functions provide access to the state of an object.
- Accessors have the keyword `const` attached at the end of the function heading.
- Ramifications of adhering to Object-Oriented Design Heuristic 6.1, "All data should be hidden within its class," include:
    - Good: Can't mess up the state (compiler complains).
    - Bad: Need to implement additional accessors (`balance`, for example).
- The ramifications of adhering to Design Heuristic 6.2, "Keep related data and behavior in one place," include:
    - Good: Results in a more intuitive design.
    - Good: Easier to maintain.
- The ramifications of adhering to Design Heuristic 6.3, "Always declare accessor member functions as const," include:
    - Good: Helps the user distinguish between modifiers and accessors.
    - Good: Adheres to the principle that objects passed as const reference parameters cannot be accidentally modified by the function while allowing the function to send const messages.
    - Bad: It is easy to forget to use `const` and the error will not show up until the object is passed in the three different modes—the result is more extensive testing to ensure the safety of const and the efficiency of const reference parameters.
- Class member functions are implemented in a manner similar to nonmember functions. However, class member functions must be qualified with the class name and `::` (the scope resolution operator). This gives the function access to the private data members.
- The class definitions have historically been stored in `.h` files.
- The member function implementations have historically been stored in `.cpp` files.
- A class should be designed to exhibit high cohesion.
    - The data should be related to the operations.
    - The messages should be related to each other.

Exercises

20. Use the `one` class and the `two` class definitions that follow to list:

    a. the data members of class `one`

    b. the member functions of class `one`

    c. the data members of class `two`

    d. the member functions of class `two`

```
class one {
public:
//--constructors
  one();
  // post: Set both operands to 0
  one(double initOp1, double initOp2);
  // post: Initialize object
//--accessor
  double sum() const;
  // Return the sum of the data
private:
  double op1;
  double op2;
};
```

```
class two {
public:
//--constructors
  two();
  // post: Data members are set to 0
  two(double initOp1, double initOp2);
  // post: Initialize object
//--accessor
  double product() const;
  // post: Return product of operands
private:
  double op1;
  double op2;
};
```

21. Using the class definitions and documentation for the `one` class and the `two` class above, write the output generated by the following program, assuming both classes are completely included by `one_two`:

```
#include <iostream>
using namespace std;
#include "one_two"  // Includes one.h, two.h, one.cpp, and two.cpp
int main()
{
  one a(1.9, 2.8);
  one b(5.0, -7.0);
  two c(4.5, 6.0);
  two d(1.4, 2.1);
  cout << a.sum() << "  " << b.sum() << endl;
  cout << c.product() << "  " << d.product() << endl;
  return 0;
}
```

22. With pencil and paper, implement the default constructor for class `one` such that both `op1` and `op2` are set to 0.0.

23. With pencil and paper, implement the other constructor for class `one` such that both `op1` and `op2` can be initialized to client-supplied arguments.

24. With pencil and paper, implement `one::sum` that returns the sum of the private data `op1 + op2`. Do not forget to place `const` after the parentheses and before the left curly brace `{`. Don't forget to add `one::`.

25. With pencil and paper, implement all three member functions for class `two` including both constructors.

26. Given this definition for a `counter` class, predict the output generated by the test driver below:

Definition

```
class counter {
public:
  counter(int startingValue, int maxValue);
  // post: Initialize counter to startingValue
  //       and set the maximum count

  void click();
  // post: If count is at maximum, set count
  //       to 0, otherwise add 1 to the count

  void reset();
  // post: Resets the counter to 0

  int count() const;
  // post: Return the current count

private:
  int my_count;
  int my_maxValue;
};
```

Diagram

counter
click() int count() int my_count int my_maxValue

TEST DRIVER

```
#include <iostream>
using namespace std;
#include "counter"  // For the counter class definition and member
                    // functions
```

```
int main()
{ // Test drive counter class
  counter aCounter(0, 2);
  aCounter.click();
  cout << aCounter.count() << endl;
  aCounter.reset();
  cout << aCounter.count() << endl;
  aCounter.click();
  cout << aCounter.count() << endl;
  aCounter.click();
  cout << aCounter.count() << endl;
  aCounter.click();
  cout << aCounter.count() << endl;
  return 0;
}
```

27. With paper and pencil, write the implementation for the constructor of the `counter` class. Make sure all postconditions will be met.

28. With paper and pencil, write the implementation of `counter::click`. Make sure all postconditions will be met, assuming the client satisfies all preconditions. You will need the `%` operator.

29. With paper and pencil, implement the `counter::reset` member function with zero parameters.

30. With paper and pencil, send an `add` message to the object named `aCounter`.

Programming Tips

3. The following programming projects require some author-supplied files.

These files are available on the disk and also at this textbook's Web site. The convention used in these programming projects is to have your program `#include` a file that `#includes` the correct class definition from the proper `.h` file and also the correct member function implementations from the correct `.cpp` file. For example, `#include "libbook"`, that is required in project 6D, should be placed before `int main()` like this:

```
#include "libbook"  // Includes the class definition (libbook.h) and
                    // member function implementations (libbook.cpp)
```

```
int main()
{
  libraryBook aBook( "The Old Man and the Sea", "Ernest Hemingway" );
  // . . .
  return 0;
}
```

4. Here is what happens when you `#include libbook` and other files.

This file named `libbook` `#includes` the file with the class definition (`libbook.h`). It also includes the file that has the member function implementations (`libbook.cpp`). It looks something like this:

```
// File name: "libbook"
#ifndef _LIBBOOK_       // Avoid duplicate compilation
#define _LIBBOOK_       // This will be defined the next time
#include "libbook.h"    // For the libraryBook class definition
#include "libbook.cpp"  // For the member function implementations
#endif
```

5. Working with three files is more difficult than working with one.

Some programming projects will now require that you work with three files, not just one. This takes a little patience as you grow accustomed to working with multiple files. Remember, the `.h` file contains the class definition; the `.cpp` file contains the member function implementations. The third file has the `main` function.

6. There is a variety of ways to make classes available.

Even though the convention of having one file include the `.h` and `.cpp` files is atypical, it makes things easier and matches the standard (many `#include` files do not have `.h` anymore). However, someday you may be asked to create object files or project files to compile and link programs using author-supplied classes. Then your program may just include the `.h` file so it can compile. Linking comes later.

```
#include "baccount.h"  // Other steps required to link
int main()
{ // . . .
```

7. The nonmember function syntax applies to member function headings also.

The function heading in the implementation must match the function heading in the class definition in terms of

- return type (none for constructors)
- function name
- number of parameters
- class of parameters
- use of `const` in both places

8. Be aware of these differences.

The function heading in the implementation differs in the following ways:

- add the class name and `::`
- do not write the semicolon after the member function heading; instead, replace it with the function body: `{ }`

```
// Part of a CD class definition file in cd.h
class CD {
public:
  // . . .
  CD(string initArtist, string initTitle, int initCdNumber);
  // Initialize a CD with zero tracks

  string artist() const;
  // . . .
private:
  // . . .
};

// Member function implementations in cd.cpp
CD::CD(string initArtist, string initTitle, int initCdNumber)
{
}

string CD::artist() const
{
  // . . .
}
```

Programming Projects

6D Author/Title

Add two accessor member functions to the `libraryBook` class named `author` and `title` so they return the book's author and title, respectively. You may use the following activities to complete this project:

- Add both function headings for `author` and `title` to the class definition in the file `libbook.h`. Since these are accessor functions, remember to write `const` after both function headings. Remember to add the semicolon.
- Save `libbook.h`.
- Implement both member functions inside the file `libbook.cpp`. Remember to include `const` in the implementation. Do not add the semicolon.
- Save `libbook.cpp`.
- Test your changes with this test driver (file name: `test6d.cpp`):

```
// File name: test6d.cpp
#include <iostream>
using namespace std;
#include "libbook"  // For libraryBook class definition (libbook.h) and
                    // member function implementations (libbook.cpp)

int main()
{ // Test drive libraryBook
  libraryBook aBook("The Mythical Man Month", "Fred Brooks");
  cout << "borrower at initialization: " << aBook.borrower() << endl;
  cout << "Author: " << aBook.author() << endl;
  cout << "Title: " << aBook.title() << endl;
  return 0;
}
```

OUTPUT

```
borrower at initialization: not currently borrowed
Author: Fred Brooks
Title: The Mythical Man Month
```

6E Transaction Count

Allow `bankAccount` objects to keep track of and report the number of transactions—number of deposits and withdrawals—made since initialization for any `bankAccount` object.

- Open the file `baccount.h`. Add a private data member named `my_transactionCount`.
- Save `baccount.h`.
- Open the file `baccount.cpp` and modify both constructors (the default with zero parameters and the single parameter constructor) so `my_transactionCount` always begins at 0, no matter which constructor is used to initialize the object.
- Save `baccount.cpp`.
- Use this test driver and ensure your output matches (file name: `test6e.cpp`):

```
// File name: test6e.cpp
#include <iostream>
using namespace std;
#include "baccount"

int main()
{
  bankAccount anAcct;
  // assert: Transaction count should start at 0

  cout << "transaction count = " << anAcct.transactionCount() << endl;
  anAcct.deposit(10.00);
  anAcct.deposit(20.00);
  anAcct.deposit(30.00);
  cout << "after three transactions: " << anAcct.transactionCount() << endl;

  bankAccount another("Bob", 100.00);
  // assert: Transaction count should be 0

  another.deposit(25.00);
  cout << "Should be 1: " << another.transactionCount() << endl;

  return 0;
}
```

Output

```
transaction count = 0
after three transactions: 3
Should be 1: 1
```

6F The `counter` Class

Using the definition for the `counter` class in exercise 26, completely implement and test the class by performing four major activities:

- Define the class in `counter.h` by retyping the `counter` class definition exactly as shown in exercise 26.

```
class counter {
public:
```

```
  // . . .
private
  // . . .
} ;  // Do NOT forget this semicolon!
```

- Implement the `counter` class member functions in a new file named `counter.cpp`. The postconditions of `counter::click` indicate the number will turn over (like the odometer on a car or a 100,000 mile click counter). To simulate this, use the `%` operator to make sure the counter resets to 0 when a click operation occurs and the counter is at the maximum value (see output that follows).
- Make sure you include `const` after the accessors.

```
int counter::count() const  // Keep the const, get rid of the ;
{
  // You fill this in
}
```

- Test your member functions with the following test driver (file name: `test6f.cpp`):

```
// File name: test6f.cpp
#include <iostream>
using namespace std;
// The file named "counter" is included on the disk
#include "counter"  // For the counter class definition and member
                    // functions. You have to create both files.

int main()
{ // Test drive counter class
  counter aCounter(0, 3);

  cout << aCounter.count() << endl;  // 0
  aCounter.click();

  cout << aCounter.count() << endl;  // 1
  aCounter.click();

  cout << aCounter.count() << endl;  // 2
  aCounter.click();

  cout << aCounter.count() << endl;  // 3
  aCounter.click();

  cout << aCounter.count() << endl;  // 0
  aCounter.click();

  cout << aCounter.count() << endl;  // 1
  aCounter.reset();
```

```
  cout << aCounter.count() << endl;  // 0

  return 0;
}
```

6G turnAround/turnRight

Add the following operations to the definition of the `grid` class in the file named `grid.h`:

```
// You add turnAround and turnRight to the class definition

void turnAround();
// post: The mover is facing the opposite direction

void turnRight();
// post: The mover is facing 90 degrees clockwise
```

Also add both class member functions at the top of the file named `grid.cpp`. Please try to ignore all the other stuff in that rather long file. You will find it easier to use the existing member function `grid::turnLeft` to implement these new ones. Feel free to retype this implementation for `grid::turnAround`:

```
void grid::turnAround()
{ // post: The mover is facing the opposite direction
  turnLeft();  // Note: The object name and dot are not needed before
  turnLeft();  //       turnLeft because turnLeft is a member of grid
               //       and the message is sent from a member of grid
}
```

Test these new functions with `test6g.cpp`:

```
// File name: test6g.cpp
#include "grid"

int main()
{
  grid g(10, 10, 8, 8, north);

  g.display();
  g.move(5);
  g.turnAround();
  g.move(3);
  g.turnRight();
  g.move(6);
  g.display();

  return 0;
}
```