Chapter 7:
Introduction to Classes and Objects

Starting Out with C++
Early Objects
Global Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley
is an imprint of
PEARSON

Topics

Topics (Continued)

7.9 Passing Objects to Functions
7.10 Object Composition
7.11 Separating Class Specification, Implementation, and Client Code
7.12 Structures
7.13 More About Enumerated Data Types
7.14 Home Software Company OOP Case Study
7.15 Introduction to Object-Oriented Analysis and Design
7.16 Screen Control (study by yourself)

7.1 Abstract Data Types

- **Programmer-created data types** that specify
  - legal values that can be stored
  - operations that can be done on the values
- The user of an abstract data type (ADT) does not need to know any implementation details (*e.g.*, how the data is stored or how the operations on it are carried out)

Abstraction in Software Development

- Abstraction allows a programmer to design a solution to a problem and to use data items without concern for how the data items are implemented
- This has already been encountered in the book:
  - To use the **`pow`** function, you need to know what inputs it expects and what kind of results it produces
  - You do not need to know how it works

Abstraction and Data Types

- **Abstraction**: a definition that captures general characteristics without details

  ex: An abstract triangle is a 3-sided polygon. A specific triangle may be scalene(不等邊), isosceles(等腰), or equilateral(正)

- **Data Type**: defines the kind of values that can be stored and the operations that can be performed on it

7.2 Object-Oriented Programming

- **Procedural programming** uses variables to store data, and focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.

- **Object-oriented programming** is based on objects that encapsulate the data and the functions that operate on it.

Object-Oriented Programming Terminology

- **object**: software entity that combines data and functions that act on the data in a single unit
- **attributes**: the data items of an object, stored in **member variables**
- **member functions (methods)**: procedures/ functions that act on the attributes of the class

More Object-Oriented Programming Terminology

- **data hiding**: restricting access to certain members of an object. The intent is to allow only member functions to directly access and modify the object’s data
- **encapsulation**: the bundling of an object’s data and procedures into a single entity

Object Example

Square

Member variables (attributes)

```
int side;
```

Member functions

```
void setSide(int s)
{   side = s;     }

int getSide()
{   return side;  }
```

Square object's data item: **`side`**

Square object's functions: **`setSide`** - set the size of the side of the square, **`getSide`** - return the size of the side of the square

Why Hide Data?

- **Protection** – Member functions provide a layer of protection against inadvertent or deliberate data corruption
- **Need-to-know** – A programmer can use the data via the provided member functions. As long as the member functions return correct information, the programmer needn't worry about implementation details.

7.3 Introduction to Classes

- **Class**: a programmer-defined data type used to define objects
- It is a pattern for creating objects

  ex:

  `string fName, lName;`

  creates two objects of the **`string`** class

Introduction to Classes

- Class declaration format:

```
class className
{
    declaration;
    declaration;
};
```

Notice the required ;

Access Specifiers

- Used to control access to members of the class.
- Each member is declared to be either

  `public`: can be accessed by functions outside of the class

  or

  `private`: can only be called by or accessed by functions that are members of the class

Class Example

Access specifiers

```
class Square
{
 private:
   int side;
 public:
   void setSide(int s)
   { side = s; }
   int getSide()
   { return side; }
};
```

More on Access Specifiers

- Can be listed in any order in a class **`(Book P447, ePP413)`**
- Can appear multiple times in a class
- If not specified, the default is **`private`**

7.4 Creating and Using Objects

- An object is an instance of a class
- It is defined just like other variables

```
Square sq1, sq2;
```

- It can access members using dot operator

```
sq1.setSide(5);
cout << sq1.getSide();
```

(P449 Program 7-1, eP415)

Types of Member Functions

- **Accessor, get, getter function**: uses but does not modify a member variable

  ex: `getSide`

- **Mutator, set, setter function**: modifies a member variable

  ex: `setSide`

7.5 Defining Member Functions

- Member functions are part of a class declaration
- Can place entire function definition inside the class declaration

  or
- Can place just the prototype inside the class declaration and write the function definition after the class

Defining Member Functions Inside the Class Declaration

- Member functions defined inside the class declaration are called **inline functions**
- Only very short functions, like the one below, should be inline functions

```
int getSide()
{ return side; }
```

Inline Member Function Example

```
class Square
{
  private:
    int side;
  public:
    void setSide(int s)
    { side = s; }
    int getSide()
    { return side; }
};
```

inline functions

Defining Member Functions After the Class Declaration

- Put a function prototype in the class declaration
- In the function definition, precede the function name with the class name and scope resolution operator (::)

```
int Square::getSide()
{
   return side;
}
```

(P452 Program 7-2, eP418)

Conventions and a Suggestion

Conventions:

- Member variables are usually **`private`**
- Accessor and mutator functions are usually **`public`**
- Use 'get' in the name of accessor functions, 'set' in the name of mutator functions

(P453 Program 7-3, eP419)

Suggestion: calculate values to be returned in accessor functions when possible, to minimize the potential for stale data

Tradeoffs of Inline vs. Regular Member Functions

- When a regular function is called, control passes to the called function
  - the compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into the program in place of the call when the program is compiled
  - This makes a larger executable program, but
  - There is less function call overhead, and possibly faster execution

7.6 Constructors

- A **constructor** is a member function that is often used to initialize data members of a class (P459 Program 7-5, eP425)
- Is called automatically when an object of the class is created (P458 Program 7-4, eP424)
- It must be a **`public`** member function
- It must be named the same as the class
- It must have no return type

Constructor – 2 Examples

Inline:	Declaration outside the class:

`class Square`
`{`
`   . . .`
`   public:`
`      Square(int s)`
`      { side = s; }`
`   . . .`
`};`
	`Square(int);   //prototype`
	`               //in class`
	
	`Square::Square(int s)`
	`{`
	`   side = s;`
	`}`

Overloading Constructors

- A class can have more than 1 constructor
- Overloaded constructors in a class must have different parameter lists

```
Square();
Square(int);
```

(P461 Program 7-6, eP427)

The Default Constructor

- Constructors can have any number of parameters, including none
- A **default constructor** is one that takes no arguments either due to
  - No parameters or
  - All parameters have default values
- If a class has any programmer-defined constructors, it must have a programmer-defined default constructor

Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square()          // default
    { side = 1; }     // constructor

    // Other member
    // functions go here
};
```

Has no parameters

Another Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square(int s = 1) // default
    { side = s; }     // constructor

    // Other member
    // functions go here
};
```

Has parameter but it has a default value

Invoking a Constructor

- To create an object using the default constructor, use no argument list and no **`()`**

```
Square square1;
```

- To create an object using a constructor that has parameters, include an argument list

```
Square square1(8);
```

7.7 Destructors

- Is a public member function automatically called when an object is destroyed
- The destructor name is ~*className*, *e.g.*,

  `~Square`

- It has no return type
- It takes no arguments
- Only 1 destructor is allowed per class

(*i.e.*, it cannot be overloaded) (P463 Program 7-7, eP429)

7. 8 Private Member Functions

- A `private` member function can only be called by another member function of the same class
- It is used for internal processing by the class, not for use outside of the class
- (P466 Program 7-8, eP432)

7.9 Passing Objects to Functions

- A class object can be passed as an argument to a function
- When passed by value, function makes a local copy of object. Original object in calling environment is unaffected by actions in function
- When passed by reference, function can use ‘set’ functions to modify the object.
- (P470 Program 7-9, eP436)

Notes on Passing Objects

- Using a value parameter for an object can slow down a program and waste space

- Using a reference parameter speeds up program, but allows the function to modify data in the parameter

Notes on Passing Objects

- To save space and time, while protecting parameter data that should not be changed, use a **`const`** reference parameter

```
void showData(const Square &s)
                         // header
```

- In order to for the showData function to call **`Square`** member functions, those functions must use **`const`** in their prototype and header:

```
int Square::getSide() const;
```

Returning an Object from a Function

- A function can return an object

```
Square initSquare();    // prototype
s1 = initSquare();      // call
```

- The function must define an object
  – for internal use
  – to use with **`return`** statement

Returning an Object Example

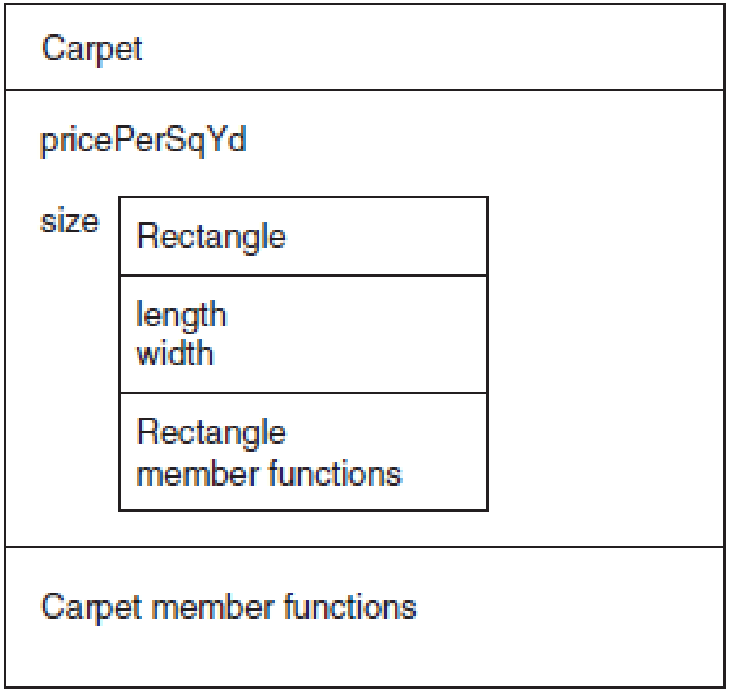
```
Square initSquare()
{
  Square s;     // local variable
  int inputSize;
  cout << "Enter the length of side: ";
  cin >> inputSize;
  s.setSide(inputSize);
  return s;
}
```

(P473 Program 7-10, eP439)

7.10 Object Composition

- Occurs when an object is a member variable of another object.
- It is often used to design complex objects whose members are simpler objects
- ex. (from book): Define a rectangle class. Then, define a carpet class and use a rectangle object as a member of a carpet object.

Object Composition, cont.



(P477 Program 7-11, eP443)

7.11 Separating Class Specification, Implementation, and Client Code

Separating class declaration, member function definitions, and the program that uses the class into separate files is considered good design

Using Separate Files

- Place class declaration in a header file that serves as the **class specification file**. Name the file ***`classname.h`*** （for example, **`Square.h`**）
- Place member function definitions in a **class implementation file**. Name the file ***`classname.cpp`*** (for example, **`Square.cpp`**）This file should **`#include`** the class specification file.
- A **client program** (client code) that uses the class must **`#include`** the class specification file and be compiled and linked with the class implementation file.

Include Guards

- Used to prevent a header file from being included twice
- Format:

```
#ifndef symbol_name
#define symbol_name
. . .      (normal contents of header file)
#endif
```

- ***symbol_name*** is usually the name of the header file, in all capital letters:

```
#ifndef SQUARE_H
#define SQUARE_H
. . .
#endif
```

Example

- P481 Rectangle.h (eP447)
- P482 Rectangle.cpp (eP448)
- P484 pr7-12.cpp (client code,eP450)
- P485 Table 7-1 + Figure 7-5
- Project (eP451)

Advantage of Using Multiple Files

- Abstraction
- Just provide a copy of the specification file and the compiled object file for the class implementation
- Easy to handle when the class member functions must be modified

What Should Be Done Inside *vs.* Outside the Class

- Class should be designed to provide functions to store and retrieve data
- In general, input and output (I/O) should be done by functions that use class objects, rather than by class member functions

7.12 Structures

- Structure: Programmer-defined data type that allows multiple variables to be grouped together
- Structure Declaration Format:

```
struct structure name
{
  type1 field1;
  type2 field2;
     …
  typen fieldn;
};
```

Example `struct` Declaration

```
struct Student
{
   int studentID;
   string name;
   short year;
   double gpa;
};
```

structure name

structure members

`Notice the required ;`

`struct` Declaration Notes

- **`struct`** names commonly begin with an uppercase letter
- The structure name is also called the tag
- Multiple fields of same type can be in a comma-separated list

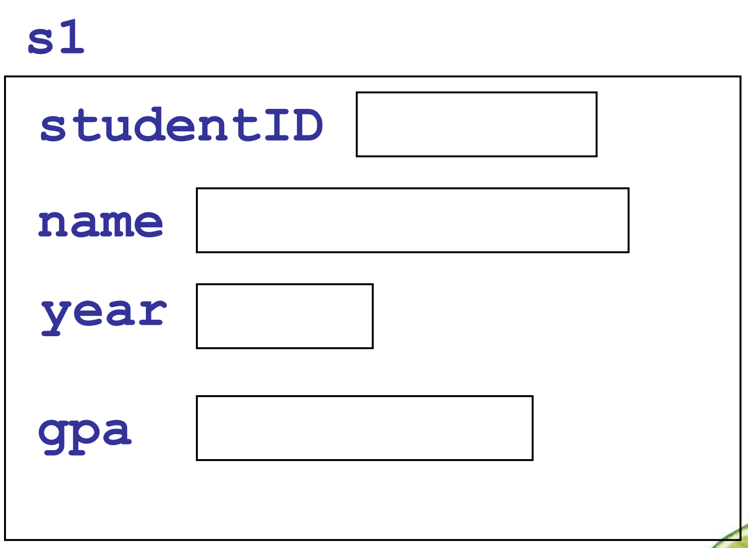
```
string name,
       address;
```

- Fields in a structure are all public by default

Defining Structure Variables

- **`struct`** declaration does not allocate memory or create variables
- To define variables, use structure tag as type name

```
Student s1;
```



Accessing Structure Members

- Use the dot `(.)` operator to refer to members of **`struct`** variables

```
getline(cin, s1.name);
cin >> s1.studentID;
s1.gpa = 3.75;
```

- Member variables can be used in any manner appropriate for their data type

Displaying `struct` Members

To display the contents of a **`struct`** variable, you must display each field separately, using the dot operator

Wrong:

```
cout << s1; // won't work!
```

Correct:

```
cout << s1.studentID << endl;
cout << s1.name << endl;
cout << s1.year << endl;
cout << s1.gpa;
```

Comparing `struct` Members

- Similar to displaying a **`struct`**, you cannot compare two **`struct`** variables directly:

```
if (s1 >= s2) // won't work!
```

- Instead, compare member variables:

```
if (s1.gpa >= s2.gpa) // better
```

Initializing a Structure

Cannot initialize members in the structure declaration, because no memory has been allocated yet

```
struct Student          // Illegal
{                       // initialization
  int studentID = 1145;
  string name = "Alex";
  short year = 1;
  float gpa = 2.95;
};
```

Initializing a Structure (continued)

- Structure members are initialized at the time a structure variable is created
- Can initialize a structure variable's members with either
  - an initialization list
  - a constructor

Using an Initialization List

An initialization list is an ordered set of values, separated by commas and contained in **{ }**, that provides initial values for a set of data members

```
{12, 6, 3}  // initialization list
            // with 3 values
```

More on Initialization Lists

- Order of list elements matters: First value initializes first data member, second value initializes second data member, etc.
- Elements of an initialization list can be constants, variables, or expressions

```
{12, W, L/W + 1} // initialization list
                 // with 3 items
```

Initialization List Example

Structure Declaration

```
struct Dimensions
{ int length,
      width,
      height;
};
```

Structure Variable

box

length	12
width	6
height	3

```
Dimensions box = {12,6,3};
```

Partial Initialization

Can initialize just some members, but cannot skip over members

```
Dimensions box1 = {12,6};  //OK
Dimensions box2 = {12,,3}; //illegal
```

Problems with Initialization List

- Can’t omit a value for a member without omitting values for all following members
- Does not work on most modern compilers if the structure contains any string objects
  - Will, however, work with C-string members

Using a Constructor to Initialize Structure Members

- Similar to a constructor for a class:
  - name is the same as the name of the struct
  - no return type
  - used to initialize data members
- It is normally written inside the **`struct`** declaration

A Structure with a Constructor

```
struct Dimensions
{
   int length,
       width,
       height;
   // Constructor
   Dimensions(int L=0, int W=0, int H=0)
   {length = L; width = W; height = H;}
};
```

Nested Structures

A structure can have another structure as a member.

```
struct PersonInfo
{   string name,
           address,
           city;
};
struct Student
{   int            studentID;
    PersonInfo     pData;
    short          year;
    double         gpa;
};
```

Members of Nested Structures

Use the dot operator multiple times to access fields of nested structures

```
Student s5;
s5.pData.name = "Joanne";
s5.pData.city = "Tulsa";
```

(P493 Program 7-14, eP459)

Structures as Function Arguments

- May pass members of **`struct`** variables to functions

```
computeGPA(s1.gpa);
```

- May pass entire **`struct`** variables to functions

```
showData(s5);
```

- Can use reference parameter if function needs to modify contents of structure variable

Notes on Passing Structures

- Using a value parameter for structure can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure
- To save space and time, while protecting structure data that should not be changed, use a `const` reference parameter

```
void showData(const Student &s)
                                   // header
```

Returning a Structure from a Function

- Function can return a **`struct`**

```
Student getStuData();   // prototype
s1 = getStuData();      // call
```

- Function must define a local structure variable
  - for internal use
  - to use with **`return`** statement

Returning a Structure Example

```
Student getStuData()
{ Student s;     // local variable
  cin >> s.studentID;
  cin.ignore();
  getline(cin, s.pData.name);
  getline(cin, s.pData.address);
  getline(cin, s.pData.city);
  cin >> s.year;
  cin >> s.gpa;
  return s;
}
```

7.13 More About Enumerated Data Types

Additional ways that enumerated data types can be used:

- Data type declaration and variable definition in a single statement:

```
enum Tree {ASH, ELM, OAK} tree1, tree2;
```

- Assign an int value to an enum variable:

```
enum Tree {ASH, ELM, OAK} tree1;
tree1 = static_cast<Tree>(1);  // ELM
```

More About Enumerated Data Types

- Assign the value of an enum variable to an int:

```
enum Tree {ASH, ELM, OAK} tree1;
tree1 = ELM;
int thisTree = tree1;  // assigns 1
int thatTree = OAK;    // assigns 2
```

More About Enumerated Data Types

- Assign the result of a computation to an enum variable

```
enum Tree {ASH, ELM, OAK} tree1, tree2;
tree1 = ELM;
tree2 = static_cast<Tree>(tree1 + 1);
                         // assigns OAK
```

More About Enumerated Data Types

- Using enumerators in a switch statement

```
enum Tree {ASH, ELM, OAK} tree1;
switch(tree1)
{
 case ASH: cout << "Ash";
           break;
 case ELM: cout << "Elm";
           break;
 case OAK: cout << "Oak";
           break;
}
```

More About Enumerated Data Types

- Using enumerators for loop control:

```
enum Tree {ASH, ELM, OAK};
for (Tree tree1 = ASH; tree1 <= OAK;
      tree1=static_cast<Tree>(tree1+1))
{
  .
  .
  .
}
```

Strongly Typed enums (C++ 11)

- Enumerated values (names) cannot be re-used in different enumerated data types that are in the same scope.
- A strongly typed enum (an enum class) allows you to do this

```
enum class Tree {ASH, ELM, OAK};
enum class Street {RUSH, OAK, STATE};
```

- Note the keyword **`class`** in the declaration

More on Strongly Typed enums (C++ 11)

- Because enumerators can be used in multiple enumerated data types, references must include the name of the strongly typed enum followed by : :

```
enum class Tree : char {ASH, ELM, OAK};
Tree tree1 = Tree::OAK;
```

More on Strongly Typed enums (C++ 11)

- Strongly typed enumerators are stored as ints by default.
- To choose a different integer data type, indicate the type after the enum name and before the enumerator list:

```
enum class Tree : char {ASH, ELM, OAK};
```

More on Strongly Typed enums (C++ 11)

- To retrieve the integer value associated with a strongly-typed enumerator, cast it to an int:

```
enum class Tree : char {ASH, ELM, OAK};
int val = static_cast<int>(Trees::ASH);
```

Example

```
// This program uses two strongly typed enumerated data types.
#include <iostream>
using namespace std;

enum class Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };

int main()
{
        Presidents prez = Presidents::ROOSEVELT;
        VicePresidents vp1 = VicePresidents::ROOSEVELT;
        VicePresidents vp2 = VicePresidents::SHERMAN;

        cout << static_cast<int>(prez) << "  " << static_cast<int>(vp1)
            << "  " << static_cast<int>(vp2) << endl;
        return 0;
}
```

7.14 Home Software Company OOP Case Study

P 503 Case study eP467

7.15 Introduction to Object-Oriented Analysis and Design

- Object-Oriented Analysis: that phase of program development when the program functionality is determined from the requirements
- It includes
  - identification of objects and classes
  - definition of each class's attributes
  - identification of each class's behaviors
  - definition of the relationship between classes

Identify Objects and Classes

- Consider the major data elements and the operations on these elements
- Candidates include
  - user-interface components (menus, text boxes, *etc.*)
  - I/O devices
  - physical objects (vehicles, machines,…)
  - historical data (employee records, transaction logs, *etc.*)
  - the roles of human participants

Define Class Attributes

- Attributes are the data elements of an object of the class

- They are necessary for the object to work in its role in the program (ex: menuItem, CustomerOrder,…)

Define Class Behaviors

- For each class,
  - Identify what an object of a class should do in the program
- The behaviors determine some of the member functions of the class

Relationships Between Classes

Possible relationships

- Access ("uses-a")
- Ownership/Composition ("has-a")
- Inheritance ("is-a")

Finding the Classes

Technique:

- Write a description of the problem domain (objects, events, etc. related to the problem)
- List the nouns, noun phrases, and pronouns. These are all candidate objects
- Refine the list to include only those objects that are relevant to the problem (P513~519, eP477)

Determine Class Responsibilities

Class responsibilities:

- What is the class responsible to know?
- What is the class responsible to do?

Use these to define some of the member functions

Object Reuse

- A well-defined class can be used to create objects in multiple programs
- By re-using an object definition, program development time is shortened
- One goal of object-oriented programming is to support object reuse

Object-Oriented vs. Object-Based Programming

- Using classes and objects might more correctly be referred to as object-based programming. When we add the ability to define relationships among different classes of objects, to create classes of objects from other classes (inheritance) and to determine the behavior of a member function depending on which object calls it (polymorphism), it becomes true object-oriented programming.

Homework 3 (1/2)

TEXT BOOK CH. 7

Programming Challenges 19. Patient Fees (P.539, eP.502)

Group Project (1、2或4人)（禁止3人）

要求：

作業總共四個部分，需要平均分配工作量，

並請在你負責的部份註解上你的學號姓名，

最後必須要在有 **main**的檔案開頭

註解小組成員的 學號、姓名、負責部份、開發中遇到的問題＆解決方案（最少兩個，獨立開發則不用）

Homework 3 (2/2)

19. Patient Fees

This program should be designed and written by a team of students. Here are some suggestions:

- One or more students may work on a single class.
- The requirements of the program should be analyzed so each student is given about the same workload.
- The names, parameters, and return types of each function and class member function should be decided in advance.
- The program will be best implemented as a multifile program.

Write a program that computes a patient's bill for a hospital stay. The different components of the program are

- The `PatientAccount` class will keep a total of the patient's charges. It will also keep track of the number of days spent in the hospital. The group must decide on the hospital's daily rate.
- The `Surgery` class will have stored within it the charges for at least five types of surgery. It can update the `charges` variable of the `PatientAccount` class.
- The `Pharmacy` class will have stored within it the price of at least five types of medication. It can update the `charges` variable of the `PatientAccount` class.
- The `main` program.

The student who designs the main program will design a menu that allows the user to enter a type of surgery, enter one or more types of medication, and check the patient out of the hospital. When the patient checks out, the total charges should be displayed.

Homework 4

TEXT BOOK CH. 7 (P.513, eP.477)

課本513頁(電子書477頁)有一段對於Joe's Automotive Shop的需求描述，以此為基礎再加上manager可隨時刪除、查詢估價單(service quote)，查詢時可以"客戶名字"或"估價單日期"作為關鍵字，列出所有符合關鍵字的估價單。

請依上述需求，實作一object-based program，並製作使用說明文件，說明如何使用你的程式，若有特殊功能亦請詳述。