

# Data Structures Using C++ 2E

## *Chapter 1* *Software Engineering Principles and* *C++ Classes*

# Textbook Programs Download

- [www.cengage.com](http://www.cengage.com)
- Search the book using keywords “Malik Data Structure C++”
- You will get to this link to download the self extracting .exe file (for Windows)
- [http://www.wadsworth.com/cgi-wadsworth/course\\_products\\_wp.pl?fid=M20b&product\\_isbn\\_issn=9780324782011&token=](http://www.wadsworth.com/cgi-wadsworth/course_products_wp.pl?fid=M20b&product_isbn_issn=9780324782011&token=)

# Objectives

- Learn about software engineering principles
- Discover what an algorithm is and explore problem-solving techniques
- Become aware of structured design and object-oriented design programming methodologies
- Learn about classes
- Become aware of `private`, `protected`, and `public` members of a class

# Objectives (cont'd.)

- Explore how classes are implemented
- Become aware of Unified Modeling Language (UML) notation
- Examine constructors and destructors
- Become aware of an abstract data type (ADT)
- Explore how classes are used to implement ADTs

# Software Life Cycle

- Program life cycle
  - Many phases between program conception and retirement
  - Three fundamental stages
    - Development, deployment (use), and maintenance
- Program retirement
  - Program too expensive to maintain
    - No new version released
- Software development phase
  - First and most important software life cycle phase

# Software Development Phase

- Four phases
  - Requirements and Analysis
  - Design
  - Implementation
  - Testing and debugging
- Requirements and Analysis
  - First and most important step
  - Analysis requirements
    - Thoroughly understand the problem
    - Understand the problem requirements
    - Divide problem into subproblems (if complex)
    - Use cases

# Software Development Phase (cont'd.)

- Design
  - Design an algorithm to solve the problem or subproblem
  - Algorithm
    - Step-by-step problem-solving process
    - Solution obtained in finite amount of time
  - Structured design
    - Dividing problem into smaller subproblems
    - Also known as: top-down design, stepwise refinement, and modular programming

# Software Development Phase (cont'd.)

- Design (cont'd.)
  - Object-oriented design (OOD)
    - Identifies components called objects
    - Determines how objects interact with one another
    - Object specifications: relevant data; possible operations performed on that data
  - Object-oriented programming (OOP) language
    - Programming language implementing OOD
  - Object-oriented design principles
    - Encapsulation, inheritance, and polymorphism
    - Reuse and extend



# Object-oriented Design Principles

- Encapsulation
  - Ability to combine data and operations
- Inheritance
  - Encourages code reuse
- Polymorphism
  - Occurs through function overloading, operator overloading, and templates

# Software Development Phase (cont'd.)

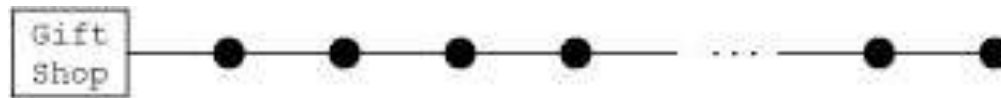
- Implementation
  - Write and compile programming code
    - Implement classes and functions discovered in the design phase
  - Final program consists of several functions
    - Each accomplishes a specific goal
  - Precondition
    - Statement specifying condition(s)
    - Must be true before function called
  - Postcondition
    - Statement specifying true items after function call completed

# Software Development Phase (cont'd.)

- Testing and debugging
  - Testing
    - Testing program correctness
    - Verifying program works properly
  - Increase program reliability
    - Discover and fix errors before releasing to user
  - Test case
    - Set of inputs, user actions, other initial conditions, and the expected output
    - Document properly
  - Black-box testing and white-box testing

# Algorithm Analysis: The Big-O Notation

- Analyze algorithm after design
- Example
  - 50 packages delivered to 50 different houses
  - 50 houses one mile apart, in the same area



**FIGURE 1-1** Gift shop and each dot representing a house

# Algorithm Analysis: The Big-O Notation (cont'd.)

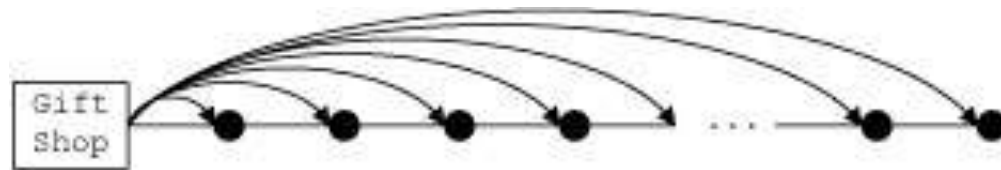
- Example (cont'd.)
  - Driver picks up all 50 packages
  - Drives one mile to first house, delivers first package
  - Drives another mile, delivers second package
  - Drives another mile, delivers third package, and so on
  - Distance driven to deliver packages
    - $1+1+1+\dots +1 = 50$  miles
  - Total distance traveled:  $50 + 50 = 100$  miles



**FIGURE 1-2** Package delivering scheme

# Algorithm Analysis: The Big-O Notation (cont'd.)

- Example (cont'd.)
  - Similar route to deliver another set of 50 packages
    - Driver picks up first package, drives one mile to the first house, delivers package, returns to the shop
    - Driver picks up second package, drives two miles, delivers second package, returns to the shop
  - Total distance traveled
    - $2 * (1+2+3+\dots+50) = 2550$  miles



**FIGURE 1-3** Another package delivery scheme

# Algorithm Analysis: The Big-O Notation (cont'd.)

- Example (cont'd.)
  - $n$  packages to deliver to  $n$  houses, each one mile apart
  - First scheme: total distance traveled
    - $1+1+1+\dots+n = 2n$  miles
    - Function of  $n$
  - Second scheme: total distance traveled
    - $2 * (1+2+3+\dots+n) = 2*(n(n+1) / 2) = n^2+n$
    - Function of  $n^2$

# Algorithm Analysis: The Big-O Notation (cont'd.)

- Analyzing an algorithm
  - Count number of operations performed
    - Not affected by computer speed

**TABLE 1-1** Various values of  $n$ ,  $2n$ ,  $n^2$ , and  $n^2 + n$

$n$	$2n$	$n^2$	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000



# Algorithm Analysis: The Big-O Notation (cont'd.)

- Example 1-1
  - Illustrates fixed number of executed operations
  - Total number of operation is 8

```
cout << "Enter two numbers";           //Line 1

cin >> num1 >> num2;                   //Line 2

if (num1 >= num2)                       //Line 3
    max = num1;                         //Line 4
else                                   //Line 5
    max = num2;                         //Line 6

cout << "The maximum number is: " << max << endl; //Line 7
```

# Algorithm Analysis: The Big-O Notation (cont'd.)

- Example 1-2
  - Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl; //Line 1

count = 0; //Line 2
sum = 0; //Line 3

cin >> num; //Line 4

while (num != -1) //Line 5
{
    sum = sum + num; //Line 6
    count++; //Line 7
    cin >> num; //Line 8
}

cout << "The sum of the numbers is: " << sum << endl; //Line 9

if (count != 0) //Line 10
    average = sum / count; //Line 11
else //Line 12
    average = 0; //Line 13

cout << "The average is: " << average << endl; //Line 14
```

# Algorithm Analysis: The Big-O Notation (cont'd.)

- Lines 1 through 4 before while have 5 operations
- Line 5 has one operation, and 4 operations within the while loop
- 9 or 8 operations after the while loop
- If while loop executes 10 times, total number of operations is:
  - $10*5 + 1 + 5 + 9$  or  $10*5 + 1 + 5 + 8$
- Generalize the case when the while loop executes  $n$  times
  - $5n + 15$  or  $5n + 14$
- For very large value of  $n$ , the term  $5n$  becomes the dominating term and the terms 15 and 14 become negligible

# Algorithm Analysis: The Big-O Notation (cont'd.)

- Search algorithm
  - $n$ : represents list size
  - $f(n)$ : count function
    - Number of comparisons in search algorithm
  - $c$ : units of computer time to execute one operation
  - $cf(n)$ : computer time to execute  $f(n)$  operations
  - Constant  $c$  depends computer speed (varies)
  - $f(n)$ : number of basic operations (constant)
  - Determine algorithm efficiency
    - Knowing how function  $f(n)$  grows as problem size grows

# Algorithm Analysis: The Big-O Notation (cont'd.)

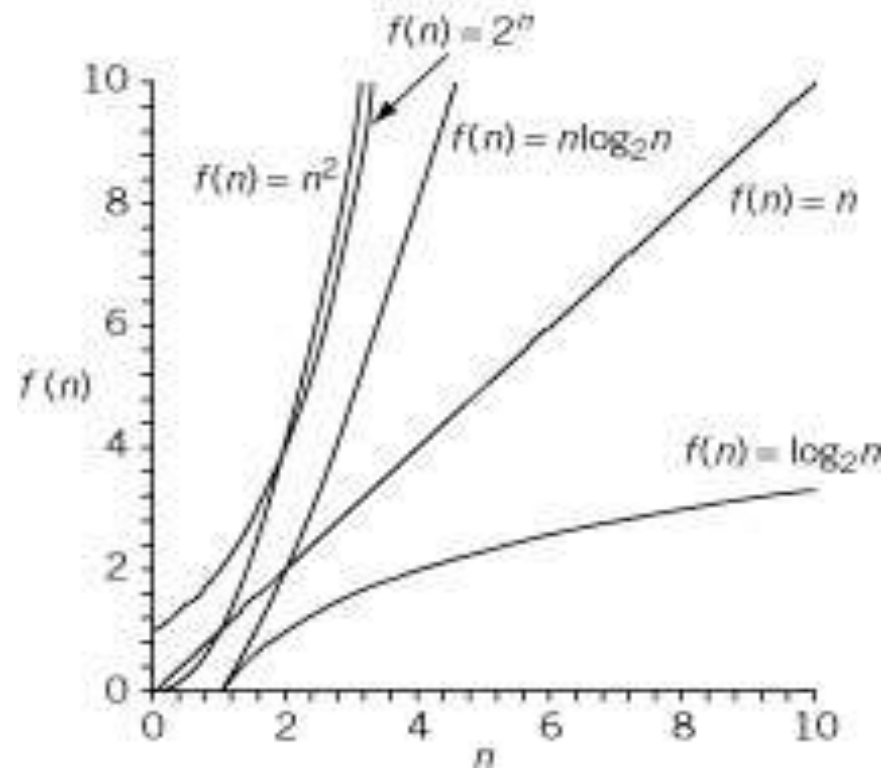
**TABLE 1-2** Growth rates of various functions

$n$	$\log_2 n$	$n \log_2 n$	$n^2$	$2^n$
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

# Algorithm Analysis: The Big-O Notation (cont'd.)

**TABLE 1-3** Time for  $f(n)$  instructions on a computer that executes 1 billion instructions per second

$n$	$f(n) = n$	$f(n) = \log_2 n$	$f(n) = n \log_2 n$	$f(n) = n^2$	$f(n) = 2^n$
10	0.01 $\mu$ s	0.003 $\mu$ s	0.033 $\mu$ s	0.1 $\mu$ s	1 $\mu$ s
20	0.02 $\mu$ s	0.004 $\mu$ s	0.086 $\mu$ s	0.4 $\mu$ s	1 ms
30	0.03 $\mu$ s	0.005 $\mu$ s	0.147 $\mu$ s	0.9 $\mu$ s	1 s
40	0.04 $\mu$ s	0.005 $\mu$ s	0.213 $\mu$ s	1.6 $\mu$ s	18.3min
50	0.05 $\mu$ s	0.006 $\mu$ s	0.282 $\mu$ s	2.5 $\mu$ s	13 days
100	0.10 $\mu$ s	0.007 $\mu$ s	0.664 $\mu$ s	10 $\mu$ s	$4 \times 10^{13}$ years
1000	1.00 $\mu$ s	0.010 $\mu$ s	9.966 $\mu$ s	1 ms	
10,000	10 $\mu$ s	0.013 $\mu$ s	130 $\mu$ s	100ms	
100,000	0.10ms	0.017 $\mu$ s	1.67ms	10s	
1,000,000	1 ms	0.020 $\mu$ s	19.93ms	16.7m	
10,000,000	0.01s	0.023 $\mu$ s	0.23s	1.16 days	
100,000,000	0.10s	0.027 $\mu$ s	2.66s	115.7 days	



**Figure 1-4** Growth rate of functions in Table 1-3

# Algorithm Analysis: The Big-O Notation (cont'd.)

- Notation useful in describing algorithm behavior
  - Shows how a function  $f(n)$  grows as  $n$  increases without bound
- Asymptotic
  - Study of the function  $f$  as  $n$  becomes larger and larger without bound
  - Examples of functions
    - $g(n)=n^2$  (no linear term)
    - $f(n)=n^2 + 4n + 20$

# Algorithm Analysis: The Big-O Notation (cont'd.)

- As  $n$  becomes larger and larger
  - Term  $4n + 20$  in  $f(n)$  becomes insignificant
  - Term  $n^2$  becomes dominant term
  - $f(n)$  and  $g(n)$  are asymptotically equivalent as  $n \rightarrow \infty$   $\square$

**TABLE 1-4** Growth rate of  $n^2$  and  $n^2 + 4n + 20$

$n$	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	100	160
50	2500	2720
100	10,000	10,420
1000	1,000,000	1,004,020
10,000	100,000,000	100,040,020



# Algorithm Analysis: The Big-O Notation (cont'd.)

- Algorithm analysis
  - If function complexity can be described by complexity of a quadratic function without the linear term
    - We say the function is of  $O(n^2)$  or Big-O of  $n^2$
- Let  $f$  and  $g$  be real-valued functions
  - Assume  $f$  and  $g$  nonnegative
    - For all real numbers  $n$ ,  $f(n) \geq 0$  and  $g(n) \geq 0$
- $f(n)$  is Big-O of  $g(n)$ : written  $f(n) = O(g(n))$ 
  - If there exists positive constants  $c$  and  $n_0$  such that  $|f(n)| \leq |cg(n)|$  for all  $n \geq n_0$

# Algorithm Analysis: The Big-O Notation (cont'd.)

**TABLE 1-5** Some Big-O functions that appear in algorithm analysis

Function $g(n)$	Growth rate of $f(n)$
$g(n) = 1$	The growth rate is constant and so does not depend on $n$ , the size of the problem.
$g(n) = \log_2 n$	The growth rate is a function of $\log_2 n$ . Because a logarithm function grows slowly, the growth rate of the function $f$ is also slow.
$g(n) = n$	The growth rate is linear. The growth rate of $f$ is directly proportional to the size of the problem.
$g(n) = n \log_2 n$	The growth rate is faster than the linear algorithm.
$g(n) = n^2$	The growth rate of such functions increases rapidly with the size of the problem. The growth rate is quadrupled when the problem size is doubled.
$g(n) = 2^n$	The growth rate is exponential. The growth rate is squared when the problem size is doubled.

# In Class Exercises

- What is the best big O notation to describe the computing times

$$T(n) = n^3 + 100n \log_2 n$$

$$T(n) = 2^n + n^{99} + 7$$

$$T(n) = \frac{n^2 - 1}{n + 1} + 8 \log_2 n$$

$$T(n) = 1 + 2 + 4 + \cdots + 2^{n-1}$$

# In Class Exercises (cont'd)

- What is the worst case computing time of the following in big O notation

```
// bubble sort
for (int i=0; i < n-1; i++) {
    for (int j=0; j<n-1; j++) {
        if (x[j] > x[j+1]) {
            temp = x[j];
            x[j]=x[j+1];
            x[j+1]=temp;
        }
    }
}
```

```
// Matrix multiplication
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++) {
        c[i][j]=0;
        for (int k=0; k<n; k++)
            c[i][j]+=a[i][k]*b[k][j];
    }
```

```
// Calculate mean
n=0;
sum=0;
cin >> x;
while (x !=-999) {
    n++;
    sum+=x;
    cin >> x;
}
```

# Classes

- OOD first step: identify components (objects)
- Encapsulation: object combines data and data operations in a single unit
- Class: collection of a fixed number of components
  - Class members: class components
  - Class member categories
    - `private`, `public`, `protected`

```
class classIdentifier
{
    class members list
};
```

```

class clockType
{
public:
    void setTime(int hours, int minutes, int seconds);
        //Function to set the time
        //The time is set according to the parameters
        //Postcondition: hr = hours; min = minutes; sec = seconds
        //    The function checks whether the values of hours,
        //    minutes, and seconds are valid. If a value is invalid,
        //    the default value 0 is assigned.

    void getTime(int& hours, int& minutes, int& seconds) const;
        //Function to return the time
        //Postcondition: hours = hr; minutes = min; seconds = sec

    void printTime() const;
        //Function to print the time
        //Postcondition: Time is printed in the form hh:mm:ss.

    void incrementSeconds();
        //Function to increment the time by one second
        //Postcondition: The time is incremented by one second.
        //    If the before-increment time is 23:59:59, the time
        //    is reset to 00:00:00.

    void incrementMinutes();

    void incrementHours();
        //Function to increment the time by one hour
        //Postcondition: The time is incremented by one hour.
        //    If the before-increment time is 23:45:53, the time
        //    is reset to 00:45:53.

    bool equalTime(const clockType& otherClock) const;
        //Function to compare the two times
        //Postcondition: Returns true if this time is equal to
        //    otherClock; otherwise, returns false

private:
    int hr; //stores the hours
    int min; //store the minutes
    int sec; //store the seconds
};

```

# Classes (cont'd.)

- Constructors
  - Declared variable not automatically initialized
  - With parameters or without parameters (default constructor)
  - Properties
    - Constructor name equals class name
    - Constructor has no type
    - All class constructors have the same name
    - Multiple constructors: different formal parameter lists
    - Execute automatically: when class object enters its scope
    - Execution: depends on values passed to class object

```

class clockType
{
public:
    //Place the function prototypes of the functions setTime,
    //getTime, printTime, incrementSeconds, incrementMinutes,
    //incrementHours, and equalTime as described earlier, here.

    clockType(int hours, int minutes, int seconds);
        //Constructor with parameters
        //The time is set according to the parameters.
        //Postconditions: hr = hours; min = minutes; sec = seconds
        //    The constructor checks whether the values of hours,
        //    minutes, and seconds are valid. If a value is invalid,
        //    the default value 0 is assigned.

    clockType();
        //Default constructor with parameters
        //The time is set to 00:00:00.
        //Postcondition: hr = 0; min = 0; sec = 0

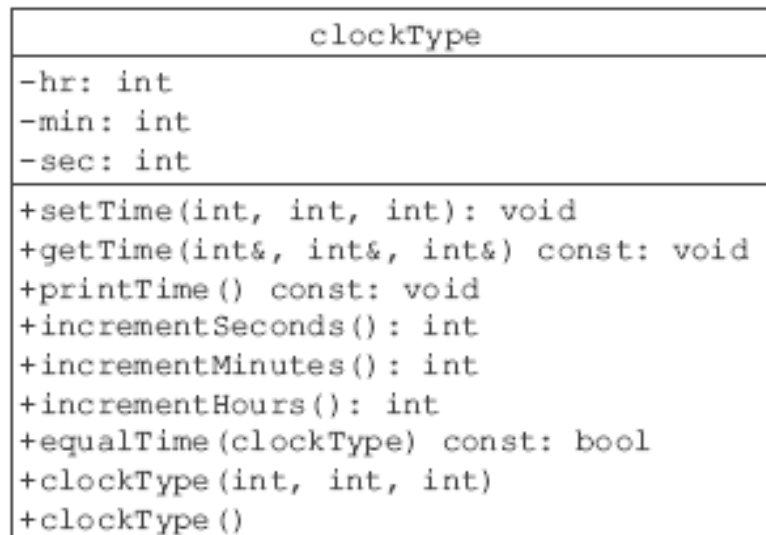
private:
    int hr; //stores the hours
    int min; //store the minutes
    int sec; //store the seconds
};

```



# Classes (cont'd.)

- Unified Modeling Language diagrams
  - Graphical notation describing a class and its members
  - Private and public members



**FIGURE 1-5** UML class diagram of the class `clockType`

# Classes (cont'd.)

- Variable (object) declaration
  - Once class defined
    - Variable declaration of that type allowed
  - Class variable
    - Called class object, class instance, object in C++
  - A class can have both types of constructors
  - Upon declaring a class object
    - Default constructor executes or constructor with parameters executes

```
className classObjectName;
```

```
className classObjectName(argument1, argument2, ...);
```

# Classes (cont'd.)

- Accessing class members
  - When an object of a class is declared
    - Object can access class members
  - Member access operator
    - The dot, . (period)
  - Class members accessed by class object
    - Dependent on where object declared

```
classObjectName.memberName
```

# Classes (cont'd.)

```
clockType myClock;  
clockType yourClock(5, 12, 40);  
  
myClock.setTime(5, 2, 30);  
myClock.printTime();  
  
If (myClock.equalTime(yourClock)) {  
...  
}
```

# Classes (cont'd.)

- Implementation of member functions
  - Reasons function prototype often included for member functions
    - Function definition can be long, difficult to comprehend
    - Providing function prototypes hides data operation details
  - Writing definitions of member functions
    - Use scope resolution operator, :: (double colon), to reference identifiers local to the class

# Classes (cont'd.)

- Implementation of member functions (cont'd.)
  - Example: definition of the function `setTime`

```
void clockType::setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

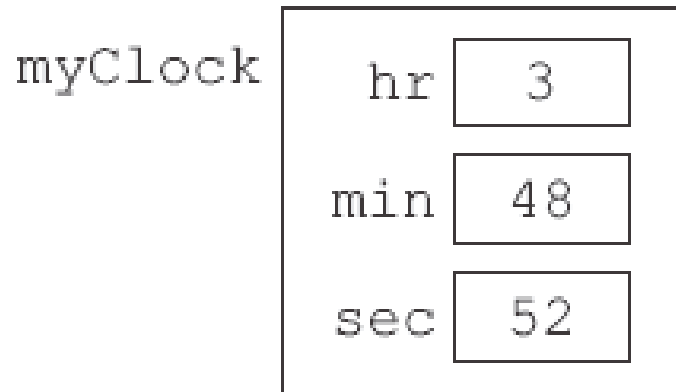
    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

# Classes (cont'd.)

- Implementation of member functions (cont'd.)

- Execute statement

```
myClock.setTime(3, 48, 52);
```



**FIGURE 1-6** Object `myClock` after the statement `myClock.setTime(3, 48, 52);` executes

# Classes (cont'd.)

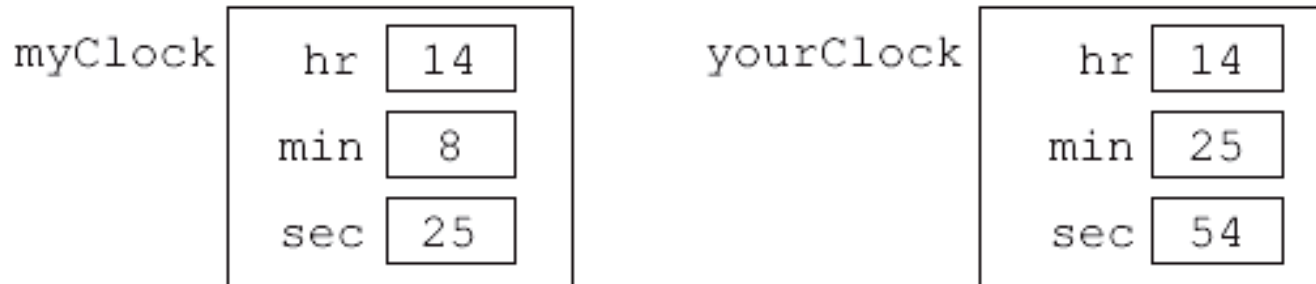
- Implementation of member functions (cont'd.)
  - Example: definition of the function `equalTime`

```
bool clockType::equalTime(const clockType& otherClock) const
{
    return (hr == otherClock.hr
            && min == otherClock.min
            && sec == otherClock.sec);
}
```



# Classes (cont'd.)

- Implementation of member functions (cont'd.)
  - Objects of type `clockType`
    - `myClock` and `yourClock`



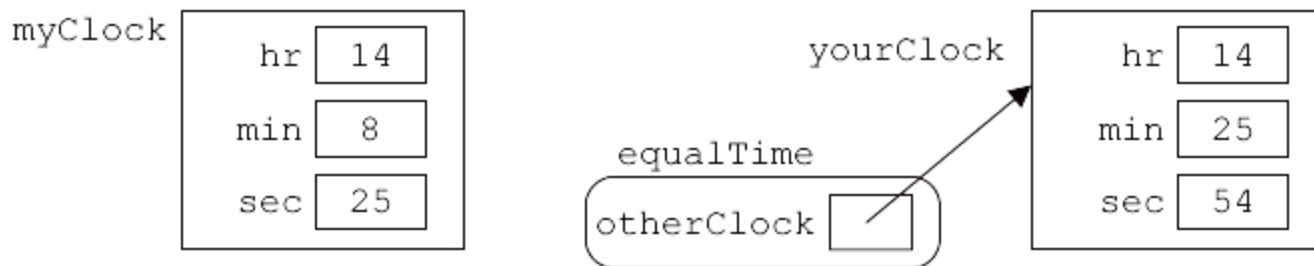
**FIGURE 1-7** Objects `myClock` and `yourClock`

# Classes (cont'd.)

- Implementation of member functions (cont'd.)

```
if (myClock.equalTime(yourClock)) ...
```

- **Object** `myClock` **accesses** member function `equalTime`
- `otherClock` is a reference parameter
- Address of actual parameter `yourClock` passed to the formal parameter `otherClock`



**FIGURE 1-8** Object `myClock` and parameter `otherClock`

# Classes (cont'd.)

- Implementation of member functions (cont'd.)
  - `equalTime` execution
  - Variables `hr`, `min`, `sec` in `equalTime` function body
    - Instance variables of variable `myClock`
  - Once class properly defined, implemented
    - Can be used in a program
  - Client
    - Program or software using and manipulating class objects
  - Instance variables
    - Have own instance of data

# Classes (cont'd.)

- Reference parameters and class objects (variables)
  - Variable passed by value
    - Formal parameter copies value of the actual parameter
  - Variables requiring large amount of memory and needing to pass a variable by value
    - Corresponding formal parameter receives copy of the data of the variable
  - Variable passed by reference
    - Corresponding formal parameter receives only the address of the actual parameter

# Classes (cont'd.)

- Reference parameters and class objects (variables) (cont'd.)
  - Disallow changing values of actual parameter
    - Declare as a reference parameter using the keyword `const`
  - If the formal parameter is a value parameter
    - Can change the value within function definition
  - If formal parameter is a constant reference parameter
    - Cannot change value within the function
    - Cannot use any other function to change its value

# Classes (cont'd.)

- Reference parameters and class objects (variables) (cont'd.)
  - Two built-in operations
    - Member access (.)
    - Assignment (=)
- Assignment operator and classes
  - Assignment statement performs a memberwise copy
  - **Example:** `myClock = yourClock;`
    - Values of the three instance variables of `yourClock` copied into corresponding instance variables of `myClock`

# Classes (cont'd.)

- Class scope
  - Automatic
    - Created each time control reaches declaration
    - Destroyed when control exits surrounding block
  - Static
    - Created once when control reaches declaration
    - Destroyed when program terminates
  - Can declare an array of class objects: same scope
  - Member of a class: local to the class
  - Access (public) class member outside the class
    - Use class object name, member access operator (.)

# Classes (cont'd.)

- Functions and classes
  - Rules
    - Class objects passed as parameters to functions and returned as function values
    - Class objects passed either by value or reference as parameters to functions
    - Class objects passed by value: instance variables of the actual parameter contents copied into the corresponding formal parameter instance variables



# Classes (cont'd.)

- Constructors and default parameters

- Constructor can have default parameters

```
clockType clockType(int = 0, int = 0, int =0);
```

- Rules declaring formal parameters

- Same as declaring function default formal parameters

- Actual parameters passed with default parameters

- Use rules for functions with default parameters

- Default constructor

- No parameters or all default parameters

```
clockType clock1;  
clockType clock2(5);
```

# Classes (cont'd.)

- Destructor
    - Function
    - No type
    - Neither value-returning nor void function
    - One destructor per class
      - No parameters
    - Name
      - Tilde character (~) followed by class name
- ```
clockType ~clockType();
```
- Automatically executes
    - When class object goes out of scope

# Classes (cont'd.)

- Structs
  - Special type of classes
  - All struct members `public`
  - C++ defines structs using the reserved word `struct`
  - If all members of a class are public, C++ programmers prefer using `struct` to group the members
  - Defined like a class

# Data Abstraction, Classes, and Abstract Data Types

- Abstraction
  - Separating design details from use
- Data abstraction
  - Process
    - Separating logical data properties from implementation details
- Abstract data type (ADT)
  - Data type separating logical properties from implementation details
  - Includes type name, domain, set of data operations

# Data Abstraction, Classes, and Abstract Data Types (cont'd.)

- ADT
  - Example: defining the `clockType` ADT

```
dataTypeName  
    clockType
```

```
domain
```

```
    Each clockType value is a time of day in the form of hours,  
    minutes, and seconds.
```

```
operations
```

```
    Set the time.
```

```
    Return the time.
```

```
    Print the time.
```

```
    Increment the time by one second.
```

```
    Increment the time by one minute.
```

```
    Increment the time by one hour.
```

```
    Compare the two times to see whether they are equal.
```

# Data Abstraction, Classes, and Abstract Data Types (cont'd.)

- Implementing an ADT
  - Represent the data; write algorithms to perform operations
  - C++ classes specifically designed to handle ADTs

# Identifying Classes, Objects, and Operations

- Object-oriented design
  - Hardest part
    - Identifying classes and objects
- Technique to identify classes and objects
  - Begin with problem description
  - Identify all nouns and verbs
    - From noun list: choose classes
    - From verb list: choose operations

# Summary

- Algorithms
- Big O notation
- Object-oriented design principles
  - Encapsulation
  - Inheritance
  - Polymorphism
- Class: collection of fixed number of components
  - Member variables + member functions
  - Constructors: initialize class instance variables
  - Destructor: cleanup, only one per class



# Summary (cont'd.)

- UML diagrams: graphical notation describing class and its members
- Abstract data type (ADT)

# Self Exercises

- Programming Exercises: 1, 2, 6