Chapter 1

Software Engineering Principles and C++ Classes

Chapter 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
- Learn about private, protected, and public members of a class
- Explore how classes are implemented

Chapter Objectives

- Become aware of Unified Modeling Language (UML) notation
- Examine constructors and destructors
- Learn about the abstract data type (ADT)
- Explore how classes are used to implement ADT
- Learn about information hiding
- Explore how information hiding is implemented in C++

Software Life Cycle

- Life cycle: the many phases a program goes through from the time it is conceived until the time it is retired
- Three fundamental stages of a program
  - Development
  - Use
  - Maintenance

Software Development Phase

- Analysis
  - Understand problem
  - Requirements analysis
  - Data analysis
  - Divide problem into subproblems and perform analysis

Software Development Phase

- Design
  - Algorithm
  - Structured Design
    - Divide problem into subproblems and analyze each subproblem
  - Object-Oriented Design
    - Identify components (objects) which form the basis of solution
    - Determine how these objects interact with one another

Object-Oriented Design (OOD)

- Three basic principles
  - Encapsulation: ability to combine data and operations in a single unit
  - Inheritance: ability to create new data types from existing data types
  - Polymorphism: ability to use the same expression to denote different operations

Software Development Phase

- Implementation
  - Write and compile programming code to implement classes and functions discovered in design phase

Software Development Phase

- Testing and Debugging
  – Test the correctness of the program to make sure it does what it is supposed to do
  – Find and fix errors
  – Run program through series of specific tests, test cases

Software Development Phase

- Testing and Debugging
  – Black-box testing: test based on inputs and outputs, not the internal working of the algorithm or function
  – White-box testing: relies on the internal structure and implementation of a function or algorithm; ensures that every part of the function or algorithm executes at least once

Algorithm Analysis: Big-O Notation

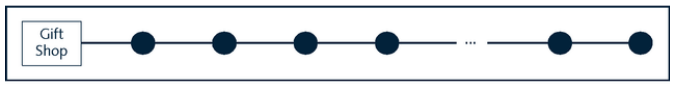


Figure 1-2 Gift shop; each dot represents a house

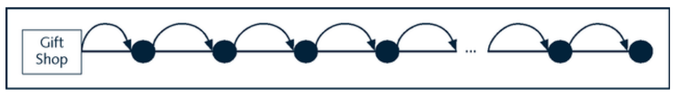


Figure 1-3 Package delivering scheme

Algorithm Analysis: Big-O Notation

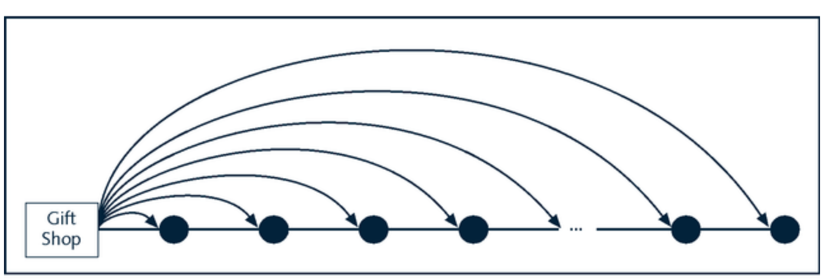


Figure 1-4 Another package delivery scheme

Algorithm Analysis: Big-O Notation

Table 1-1 The 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
1,000	2,000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

Algorithm Analysis: Big-O Notation

Table 1-2 Growth Rate 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	1,024	4,294,967,296

Algorithm Analysis: Big-O Notation

- **Definition:** Let *f* be a function of *n*.The term "**asymptotic**" means the study of the function *f* as *n* becomes larger and larger without bound.

Algorithm Analysis: Big-O Notation

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.

Classes

- class: reserved word; collection of a fixed number of components
- Components: member of a class
- Members accessed by name
- Class categories/modifiers
  - private
  - protected
  - public

Classes

- private: members of class not accessible outside class
- public: members of class accessible outside class
- Class members: can be methods or variables
- Variable members: declared like any other variables

Syntax for Defining a Class

```
class classIdentifier
{
    classMemberList
};
```

```
classVariableName.memberName
```


UML Diagram
class clockType

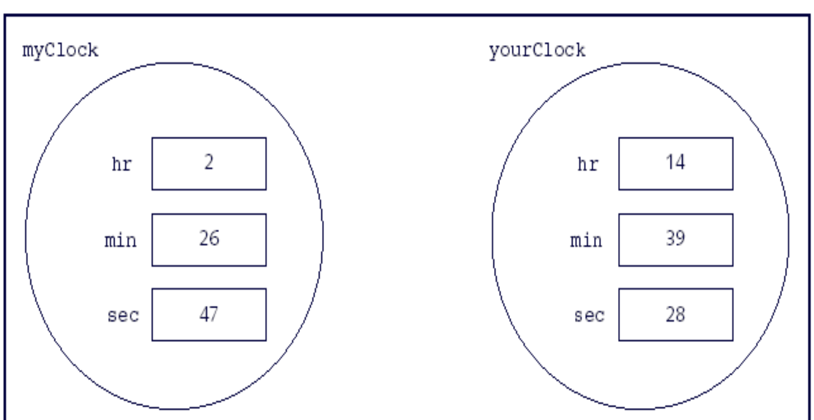
```
clockType
--------------------------------------
-hr: int
-min: int
-sec: int
--------------------------------------
+setTime (int, int, int): void
+getTime (int&, int&, int&): void
+printTime (): void
+incrementSeconds (): int
+incrementMinutes (): int
+incrementHours (): int
+equalTime (const clockType&): bool
```

Figure 1-5 UML diagram of the `class clockType`

UML Diagram

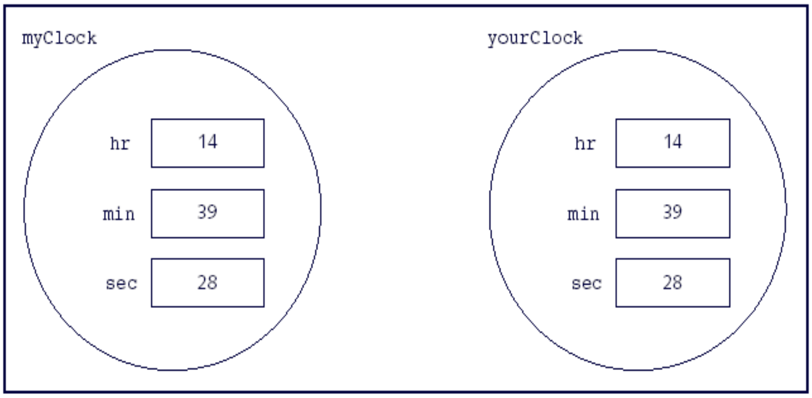
- Top box: Name of class
- Middle box: data members and their data types
- Bottom box: member methods' names, parameter list, return type of method
- + means public method
- - means private method
- # means protected method

Example: class Clock



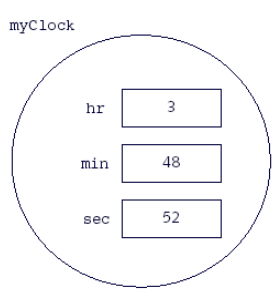
Example: class Clock

After `myClock = yourClock;`



Example: class Clock

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



Constructors

Syntax to invoke the default constructor:

```
className  classVariableName;
```

Syntax to invoke a constructor with a parameter:

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


Destructors and Structs

- Destructors
  - Function like constructors
  - Do not have a type
  - Only one per class
  - Can have no parameters
  - Name of destructor is (-) character followed by name of class
- Structs
  - Special type of class
  - By default all members are public
  - struct is a reserved word
  - Defined just like a class

Abstract Data Types

- Definition

  A data type that specifies the logical properties without the implementation details
- Examples: stacks, queues, binary search trees

Information Hiding

- Implementation file: separate file containing implementation details
- Header file (interface file): file that contains the specification details

Programming Example: Candy Machine (Problem Statement)

A common place to buy candy is from a candy machine. A new candy machine is bought for the gym, but it is not working properly. The machine sells candies, chips, gum, and cookies. You have been asked to write a program for this candy machine so that it can be put into operation.

The program should do the following:

1. Show the customer the different products sold by the candy machine.
2. Let the customer make the selection.
3. Show the customer the cost of the item selected.
4. Accept money from the customer.
5. Release the item.

Programming Example: Candy Machine (Input and Output)

- Input
  - Item Selection
  - Cost of Item
- Output
  - The selected item

Programming Example: Candy Machine

- Components
  - Cash Register
  - Several Dispensers

Programming Example: Candy Machine

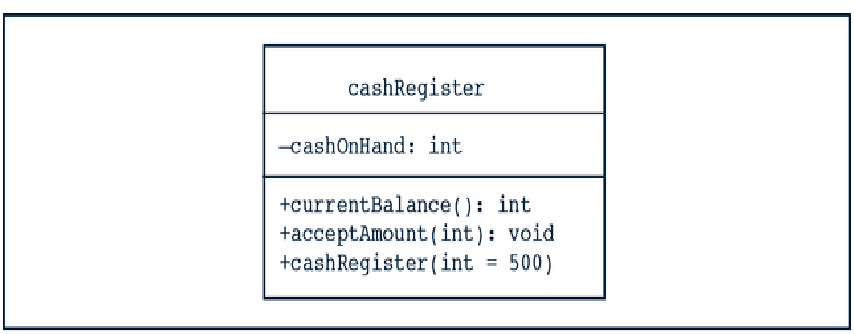


Figure 1-13 UML diagram of the class cashRegister

Programming Example: Candy Machine

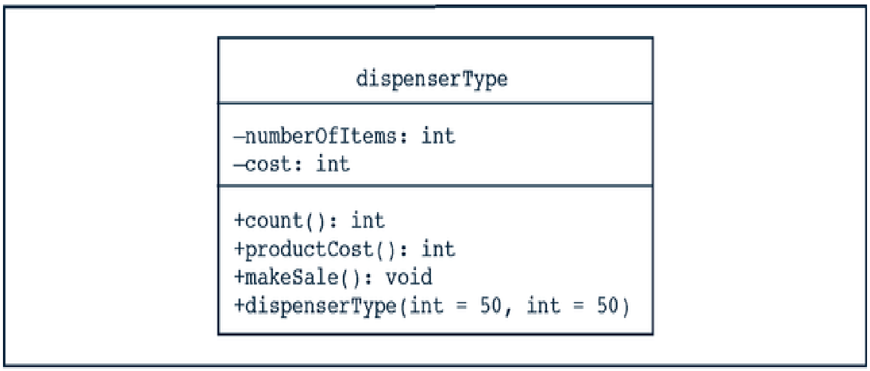


Figure 1-14 14 UML diagram of the class dispenserType

Programming Example: Candy Machine

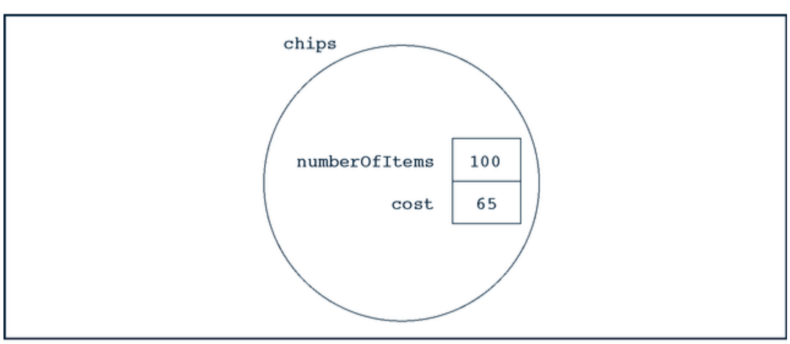


Figure 1-15 Object chips

Object-Oriented Design

- Simplified methodology
  1. Write down detailed description of problem
  2. Identify all (relevant) nouns and verbs
  3. From list of nouns, select objects
  4. Identify data components of each object
  5. From list of verbs, select operations

Object-Oriented Design Example

- Problem Statement
  – Write a program to input the dimensions of a cylinder and calculate and print the surface area and volume
- Nouns: dimensions, surface area, volume, cylinder
- Verbs: input, calculate, print
- Object: cylinder
  – Data members: dimensions (NOT SA or volume)
  – Function members: input, calculate SA, calculate volume, print