Data Abstraction and ADT

孫雅麗
台灣大學 資管所
Outline

- What is an abstract data type (ADT)?
- How to specify ADT List?
- How to implement an ADT?
  - C++ classes
  - array implementation
Modular Design

- To break a solution into a number of small, manageable modules

- To reduce the complexity of the problem-solving

- To control the interaction between modules in a systematic fashion – API (application Programming Interface)

- To facilitate “piecing together” existing software components with yet-to-be-developing software - reusability

- Only need a clear specification of the existing software (What it does. NOT how it does.)
Data Abstraction

- To identify important data (entities, objects) in the problem domain.
- To focus on what operations can do on the identified data instead of how to implement them.

Examples:
- Class enrollment system
- Windowing system
Data Types in C++ (1/2)

- Basic data types, e.g., `char`, `int`, `float` and `double`
- Modifiers of basic data types,
  - e.g., `short`, `long`, `unsigned`, and `signed`
- Pointer data type, e.g., `int i`, `*pi`
- `array`, `struct` and `class`, e.g., `int a[20]` and

```c
struct student{
    char name;
    int student_id;
    char grade;
}
```
User defined data type, e.g., `typedef structure student…`

```cpp
typedef student_record {
    char student_name[25];
    char student_id[8];
    int departmentID;
    int grade;
} 

student_record old_student, new_student;
```
What is a Data Type?

- Example data types:
  - int: +, -, *, / and % (operators)
  - float: +, -, *, / and % (operators)
  - (type casting, overloading, etc.)

- Data types are programming language level representation of the modeling and abstraction of the “things” in the real world!!
Abstract Data Type (1/3)

- An abstract data type (ADT) is a specification (definition) of a collection of data and a set of operations on that data.

- Abstracting is to conceptualize and summarize "things" of interest!!

- An "abstract data type (ADT)" is a data type organized in a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.
Wall of ADT operations

Program

Data Structure

ADD
Remove
Find
Display
Violating the Wall of ADT operations

Program

Data Structure

ADD
Remove
Find
Display
Figure 3-4

A wall of ADT operations isolates a data structure from the program that uses it.
Abstract Data Types (2/3)

- An operation specification consists of
  - the name of the function (or operation)
  - the type of its arguments
  - the type of its result (function prototype)

Boolean makeCall (phoneNumber dialNumber);

- a description of what the function does.
- Do not appeal to internal representation or implementation details, i.e. “implementation-independent”
Abstract Data Types (3/3)

- The functions (operations) of an abstract data type are classified into three categories:
  - *Creator/Constructor* – to create a new instance of the designated type
  - *Transformers* – to create an instance of the designated type by using one or more other instances (e.g., copy constructor)
  - *Observers/reporters* – to provide information about an instance of type without modifying it.
Benefits of ADT in Software Development

- Simplification of software development
  - Decomposition of a complex task of developing a software system into a number of simpler subtasks

- Testing and Debugging
  - Each subtask can be tested and debugged separately

- Reusability

- Modification to the representation of a data type
Specifying ADTs (1/2)

- Defining ADTs are to **model** and **abstract** (conceptualize and summarize) “things” of interest!!
- A specification (definition) of a collection of data and a set of operations on that data

Example: Manipulation of a grocery list

- milk
- eggs
- butter
- apples
- bread
- chicken
Specifying ADTs (2/2)

Example: Manipulation of a grocery list

- **Data-items**: list
- **Operations**
  - To **add** a new item to the beginning of the list
  - To **add** a new item to the end of the list
  - To **add** a new item to a proper position so that the list is in alphabetical or chronological order
  - To **remove** an item from the list
  - To determine the **length** of the list
  - To **search** an item in the list
  - To **retrieve** first item—head of the list
  - To **retrieve** last item—the end (tail) of the list
  - **Retrieve**: Unique predecessor (Previous) and successor (Next)
Example: ice maker in a refrigerator

Out-of-ice indicator

Chilled water
Crushed ice
Ice cubes

Water
ADT List
Definition

- A list is an ordered sequence of elements
- \( L = \{X_0, X_1, \ldots, X_{n-1}\} \)
- Length of \( L \) is \( n \)
The ADT List

- Basic Operations
- Example:
  - operations on a grocery list
  - implementation-independent application of the ADT list
- The ADT sorted list
  - operations
ADT List: Operations (1/4)

- `createList() // Creates an empty list.`

- `destroyList() // Destroys a list.`

- `isEmpty(): boolean {query} // Determines whether a list is empty.`

- `getLength(): integer {query} // Returns the number of items that are in a list.`
ADT List Operations (2/4)

- `insert` (in `index` : integer,
  in `newItem` : ListItemType,
  out `success` : boolean)

  // Inserts `newItem` at position `index` of a list, if
  // 1 <= index <= getLength()+1.
  // If index <= getLength(), items are renumbered
  // as follows: The item at index becomes the item
  // at index+1, the item at index+1 becomes
  // the item at index+2, and so on. The `success` flag
  // indicates whether the insertion was successful.
**ADT List Operations (3/4)**

- **remove** (in `index`:integer, out `success`: boolean)

  // removes the item at position index of a list, if
  // 1 <= index <= getLength(). If index <
  // getLength(), items are renumbered as follows: The item
  // at index+1 becomes the item at index, the item at
  // index+2 becomes the item at index+1, and so on.
  // The `success` flag indicates whether the deletion was successful.
ADT List Operations (4/4)

- `retrieve` (in `index`: integer, out `dataItem`: ListItemType, out `success`: boolean)

  // Copies the item at position index of a list into `dataItem`, if 1 <= index <= getLength().
  
The list is left unchanged by this operation. The `success` flag indicates whether the retrieval was successful.
Example: grocery list (1/3)

- The grocery list
  - milk, eggs, butter, apples, bread, chicken

- Construct the list by using the ADT list operations.

  *First create an empty list $L$ and then use a series of insertion operations to successively append the item's to the list:*

  - `aList.CreateList()`
  - `aList.ListInsert(1, milk, Success)`
  - `aList.ListInsert(2, eggs, Success)`
  - `aList.ListInsert(3, butter, Success)`
  - `aList.ListInsert(4, apples, Success)`
  - `aList.ListInsert(5, bread, Success)`
  - `aList.ListInsert(6, chicken, Success)`
Example: grocery list (2/3)

- The list \( L \) becomes.
  
  \[ \textit{milk, eggs, butter, apples, bread, chicken} \]

- The operation
  
  \[ \texttt{aList.insert(4, nuts, success)} \]

- The list \( L \) becomes.
  
  \[ \textit{milk, eggs, butter, \textbf{nuts}, apples, bread, chicken} \]

- All items that were at position numbers greater than or equal to 4 before the insertion are \underline{moved} and are \underline{now} at the next higher position number after the insertion.
Example: grocery list (3/3)

- The list $L$
  
  \[ \text{milk, eggs, butter, nuts, apples, bread, chicken} \]

- The operation
  
  `aList.remove(5, success)`

- The list becomes
  
  \[ \text{milk, eggs, butter, nuts, bread, chicken} \]
The wall between “displayList” and the implementation of the ADT list
List Example: grocery list implementation

displayList(in aList:List)
// Displays the item on the list L.
for (position = 1 through aList.getLength())
{
    aList.retrieve(position, dataItem, success)
    Display dataItem
}

replace (in aList:List, in i:integer,
in newItem:ListItemType, out success: boolean)
// Replace the i\textsuperscript{th} item on the list aList with newItem.
// The success flag indicates whether the replacement was successful.
{ aList.remove(i, success)
    if (success)
        aList.insert(i, newItem, success)
The ADT “Sorted” List Operations (1/3)

- `createSortedList()`  
  // Creates an empty sorted list.

- `destroySortedList()`  
  // Destroys a sorted list.

- `sortedIsEmpty()` : boolean  
  // Determines whether a sorted list is empty.

- `sortedGetLength()` : integer  
  // Returns the number of items that are in a sorted list.
The ADT Sorted List Operations (2/3)

- `sortedInsert (in newItem: ListItemType, out success: boolean)`
  
  // Inserts newItem into its proper sorted position in a sorted list. Success indicates whether the insertion was successful.

- `sortedRemove (in anItem: ListItemType, out success: boolean)`
  
  // Deletes anItem from a sorted list. Success indicates whether the deletion was successful.
ADT Sorted List Operations (3/3)

- **sortedRetrieve** (in `index:integer`, out `dataItem:ListItemType`, out `success: boolean`) {query}

  // Sets `dataItem` to the item at position `index` of a sorted list, if 1 <= `index` <= `SortedGetLength()`.
  // The list is left unchanged by this operation.
  // `success` indicates whether the retrieval was successful.

- **locatePosition** (in `anItem:ListItemType`, out `isPresent:boolean`): integer {query}

  // Sets `position` to the position where `anItem` belongs or exists in a sorted list. `IsPresent` indicates whether `anItem` is currently in the list. `anItem` and the list are unchanged.
Design an ADT: An Appointment Book

- An appointment book that spans a one-year period

- To identify an ADT appointment-book
  - **Data**: appointment (date, time, purpose)
  - **Operations specification**
ADT “appointment book” Operations (1/2)

- `createAppointmentBook()`
  // Creates an empty appointment book.

- `destroyAppointmentBook()`
  // Destroys the appointment book.

- `isAppointment (in appDate:Date, in appTime:Time) : boolean`
  // Returns true if an appointment exists for the appDate and appTime specified; otherwise returns false.
ADT “appointment book” Operations
(2/2)

- **makeAppointment** (in appDate:Date, in appTime:Time, in purpose:String) : boolean
  // Inserts an appointment for the appDate, appTime, and Purpose specified, if an appointment at appDate/Time does not exist already.

- **cancelAppointment** (in appDate:Date, in appTime:Time) : boolean
  // Deletes the appointment for the appDate and appTime specified.

- **checkAppointment** (in appDate:Date, in appTime:Time, out purpose:String) : boolean
  // Retrieves into Purpose the purpose of the appointment at the given appDate/appTime, if one exists.
Using ADT “appointment book”

(1/2)

// change the date or time of an appointment within apptBook
change (oldDate, oldTime, newDate, newTime)
{
   // 1. get purpose of appointment
   apptBook.checkAppointment (oldDate, oldTime, oldPurpose)

   if (oldPurpose is not null) // 2. a scheduled appointment?
      {
         // 3. see if new date/time is available
         if ( apptBook.isAppointment(newDate, newTime) )
            // new date/time is booked
            write ("You already have an appointment at ", newTime,
                   " on ", newDate)
Using ADT appointment book

(2/2)

else  // new date/time is available
  { apptBook.cancelAppointment(oldDate, oldTime)
    if (apptBook.makeAppointment(newDate, newTime, oldPurpose)

      write ("Your appointment has been rescheduled to", newTime, "on", newDate)

    }
  }
} // end if
else
  write ("You do not have an appointment: at ", oldTime,

    " on ", oldDate)

}
ADTs that suggest other ADTs: Example (1/7)

- Problem: Design a data base of recipes (a container of objects)
  - A container itself is an object (objects in a container object)

- Consider this database as an ADT recipe-data-base
  - data-recipe (an object)
  - operations (insertRecipe, deleteRecipe, retrieveRecipe)
ADT “recipe database” (2/7)

ADT recipe-data-base

- insertRecipe (in aRecipe:Recipe, out success: boolean)
  // inserts recipe into the data base.

- deleteRecipe (in aRecipe:Recipe, out success: boolean)
  // Deletes recipe from the data base.

- retrieveRecipe (in name: string, out aRecipe: Recipe, out success: boolean)
  // Retrieves the named recipe from the data base.
ADTs that suggest other ADTs:

Example (3/7)

- Problem: want to write a function to scale a recipe retrieved from the DB
  - (e.g., initially it is for \( n \) people; but now you want to resize it to serve \( m \) people)

- “Measurements” in a recipe
  - contains measurement of coefficient quantities
  - e.g., 1 \( \frac{1}{2} \) cups, 1 table-spoon, and \( \frac{1}{4} \) teaspoon
  - integers and fractions in units
ADTs that suggest other ADTs:
Example (4/7)

- Design an ADT **measurement**
  - **data-measure** *(unit, quantity)*
  - **Operations**
    - *(getMeasure, setMeasure, scaleMeasure, convertMeasure)*
ADT “recipe database” (5/7)

ADT **measurement**

- **getMeasure()**: measurement
  // Returns the measure.

- **setMeasure(in m: measurement)**
  // Sets a measure.

- **scaleMeasure (in scaleFactor: float; out newMeasure: measurement)**
  // Multiplies measure by a fractional scaleFactor, which
  // has no units, to obtain newMeasure.

- **convertMeasure(in oldUnits: measureUnit, in newUnits: measureUnit; out newMeasure: measurement)**
  // Converts measure from its old units to newMeasure in new units.
ADTs that suggest other ADTs:

Example (6/7)

- Perform exact fractional arithmetic (In C++, floating-point arithmetic is not exact).

- an ADT fraction
  - data-fraction
  - operations
    (addition, subtraction, multiplication and division)
ADTs that suggest other ADTs: Example (7/7)

- addFraction (in first: fraction, in second: fraction): fraction
  // adds two fractions and returns the sum reduced to lowest terms.
Summary

- ADT recipe-data-base
  - data-recipe (an object)
  - operations (insertRecipe, deleteRecipe, retrieveRecipe)

- ADT measurement
  - data-measure (unit, quantity)
  - Operations (getMeasure, setMeasure, scaleMeasure, convertMeasure)

- ADT fraction
  - data-fraction
  - operations (addition, subtraction, multiplication and division)
ADT Implementation

- To choose a data structure for the ADT’s data
- To write functions that access the data in accordance with the ADT operation
  - Use top-down design approach to design an algorithm for each of the operations
- Encapsulation
  - no direct access to data structures
**Figure 3-8**

A wall of ADT operations isolates a data structure from the program that uses it.
ADT List Implementation using Array
Array-based ADT List

- Implement the ADT list as a class.
- The ADT list operations:
  - createList( )
  - destroyList( )
  - isEmpty( )
  - getLength( )
  - insert (in index, in newItem, out success)
  - remove (in index, out success)
  - retrieve (in index, out dataItem, out success)
### An array-based implementation of the ADT list

<table>
<thead>
<tr>
<th>Array indexes</th>
<th>MAX_List-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-1</td>
<td>K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>ADT list position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>K</td>
<td>MAX_List</td>
</tr>
</tbody>
</table>

#### Figure 3-11  Shifting items for insertion at Position 3

<table>
<thead>
<tr>
<th>Array indexes</th>
<th>New item</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>ADT list position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MAX_List</td>
<td>MAX_List-1</td>
</tr>
</tbody>
</table>

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
(a) Deletion causes a gap; (b) Fill gap by shifting

(a) Array indexes

<table>
<thead>
<tr>
<th>K</th>
<th>12</th>
<th>3</th>
<th>44</th>
<th>100</th>
<th>...</th>
<th>10</th>
<th>18</th>
<th>...</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>K</td>
<td>K+1</td>
<td>MAX_List</td>
<td>MAX_List-1</td>
</tr>
<tr>
<td>ADT list position</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Array indexes

<table>
<thead>
<tr>
<th>K</th>
<th>12</th>
<th>3</th>
<th>44</th>
<th>100</th>
<th>...</th>
<th>10</th>
<th>18</th>
<th>...</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>K</td>
<td>MAX_List</td>
<td>MAX_Hist</td>
<td>MAX_List-1</td>
<td></td>
</tr>
<tr>
<td>ADT list position</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Array-based ADT List: List.h

// **************************************************
// Header file List.h for the ADT list.
// Array-based implementation.
// **************************************************
const int MAX_LIST = maximum-size-of-list;
typedef desired-type-of-list-item listItemType;

class listClass
{
    public:
        listClass (); // default constructor

        // destructor is supplied by compiler
Array-based ADT List:

List.h (cont’d)

// list operations:

```cpp
boolean isEmpty() const;
```

// Determines whether a list is empty.
// Precondition: None.
// Postcondition: Returns true if the list is empty, otherwise returns false.

// tagged with const is a fail-safe technique that ensures the operation will not change the data of the object instance.

```cpp
int getLength() const;
```

// Determines the length of a list
// Precondition: None.
// Postcondition: Returns the number of items that are currently in the list.
Array-based ADT List: List.h (cont’d)

insert (int index, listItemType newItem, bool & success)

// Inserts an item into a list.
// Precondition: index indicates where the
// insertion should occur.
// newItem is the item to be inserted.
// Postcondition: If insertion was successful, newItem is at position index in the
// list, other Items are renumbered accordingly, and success is true;
// otherwise success is false.
// Note: Insertion will not be successful if index < 1 or > ListLength()+1.
Array-based ADT List: List.h (cont’d)

- **remove** (int index, bool & success)

  // Deletes an item from a list.
  // Precondition: index indicates where the deletion
  // should occur.
  // Postcondition: If 1 <= index <= ListLength(),
  // the item at position index in the list is
  // deleted, other items are renumbered accordingly,
  // and success is true; otherwise success is false.
Array-based ADT List: 
List.h (cont’d)

void retrieve(int index, listItemType dataItem, 
bool & success) const;

// Retrieves a list item by position number.
// Precondition: index is the number of the item to 
// be retrieved.
// Postcondition: If 1<= index <= ListLength(), 
// DataItem is the value of the desired item and 
// success is true; otherwise success is false.
Array-based ADT List: List.h

private:

listItemType items [MAX_LIST];   // array of list items
int size;                                    // number of items in list
int translate(int index) const; // a private function

// Converts the position of an item in a list to the
// correct index with in its array representation.

};   // end class

// End of header file.
// *********************************************
// Implementation file ListA.cpp for the ADT list.
// Array-based implementation.
// *********************************************

#include "ListA.h " // header file

listClass::listClass() : size(0)
{
}
} // end default constructor
bool listClass::listIsEmpty() const
{
    return bool(size == 0);
}  // end ListIsEmpty

int listClass::listLength() const
{
    return size;
} // end ListLength
void listClass::insert(int index, listItemType newItem, bool & success)
{
    success = (index >= 1) && (index <= size+1) && (size < MAX_LIST); //
    if (success)
    {
        // make room for new item by shifting all items at
        // position >= index toward the end of the
        // list (no shift if index== size+1)
        for (int position = size; position >= index; --position)
            items[translate(position+1)] = items[translate(position)];

        // insert new item
        items[translate(index)] = newItem;
        ++size;
    } // end if
} // end insert
ADT List: ListA.cpp (cont'd)

void listClass::remove(int index, bool & success)
{
    success = (index >=1) && (index <= size);
    if (success)
    {
        // delete item by shifting all items at positions > position toward the beginning of the list
        // (no shift if position == size)
        for (int fromPosition = position+1; fromPosition <= size; ++fromPosition)
            items[translate(fromPosition-1)] = items[translate(fromPosition)];
        --size;
    } // end if
}  // end ListDelete
void listClass::retrieve(int index, listItemType & dataItem, bool & success) const
{
    success = (index >= 1) && (index <= size);
    if (success)
        dataItem = items[translate(index)];
} // end ListRetrieve

int listClass::translate(int index) const
{
    return index-1;
} // end

// End of implementation file.
The end. 😊
Homework

- page 167: #5, #7.
- Page 170: #6, #8 and #10.
C++ Classes
C++ Classes (1/2)

- A C++ program is a collection of objects that interact.
- A class contains data members and member functions
- Public and private
- An example C++ class: Sphere
A C++ Class: Sphere.h

// *********************************************
// Header file Sphere.h for the class sphereClass.
// *********************************************
const double PI = 3.14159;
class Sphere
{
    public:
        Sphere();
        // Default constructor: Creates a sphere and
        // initializes its radius to a default value.
        // Precondition: None.
        // Postcondition: A sphere of radius 1 exists.
A C++ Class: Sphere.h (cont’d)

Sphere (double initialRadius);
// Constructor: Creates a sphere and initializes
// its radius.
// Precondition: initialRadius is the desired radius.
// Postcondition: A sphere of radius initialRadius exists.

void setRadius (double NewRadius);
// Sets (alters) the radius of an existing sphere.
// Precondition: NewRadius is the desired radius.
// Postcondition: The sphere's radius is NewRadius
A C++ Class : Sphere.h (cont’d)

double getRadius () const;
// Determines a sphere's radius.
// Precondition: None.
// Postcondition: Returns the radius.

double getDiameter () const;
// Determines a sphere's diameter.
// Precondition: None.
// Postcondition: Returns the diameter.
double getCircumference() const;
// Determines a sphere's circumference.
// Precondition: PI is a named constant.
// Postcondition: Returns the circumference.

double getArea() const;
// Determines a sphere's surface area.
// Precondition: PI is a named constant.
// Postcondition: Returns the surface area.

double getVolume() const;
// Determines a sphere's volume.
// Precondition: PI is a named constant.
// Postcondition: Returns the volume.
A C++ Class: Sphere.h

```cpp
void displayStatistics() const;
// Displays statistics of a sphere.
// Precondition: None.
// Postcondition: Displays the radius, diameter,
// circumference, area, and volume.

private:
    double theRadius;  // the sphere's radius
};  // end class
// End of header file.
```
sphereClass (cont’d)

// *********************************************
// Implementation file Sphere.cpp for the class Sphere.
// *********************************************
#include "Sphere.h"  // header file
#include <iostream.h>
using namespace std;

Sphere::Sphere (): theRadius(1.0)
{}  // end default constructor

Sphere::Sphere (double initialRadius)
{ if (initialRadius > 0)
    theRadius = initialRadius;
  else
    theRadius = 1.0;
}  // end constructor
Sphere::setRadius (double newRadius)
{
    if (newRadius > 0)
        theRadius = newRadius;
    else
        theRadius = 1.0;
} // end setRadius

Sphere::getRadius () const
{
    return theRadius;
} // end getRadius
sphereClass (cont’d)

double Sphere::getDiameter() const
{
    return 2.0 * theRadius;
}  // end getDiameter

double Sphere::getCircumference() const
{
    return PI * getDiameter();
}  // end getCircumference
sphereClass (cont’d)

double Sphere::getArea() const
{
    return 4.0 * PI * theRadius * theRadius;
} // end getArea

double Sphere::getVolume() const
{
    double radiusCubed = theRadius * theRadius * theRadius;
    return (4.0 * PI * radiusCubed )/3.0;
} // end getVolume
sphereClass

void Sphere::displayStatistics() const
{
    cout << "\nRadius = " << getRadius()
        << "\nDiameter = " << getDiameter()
        << "\nCircumference = " << getCircumference()
        << "\nArea = " << getArea()
        << "\nVolume = " << getVolume() << endl;
} // end displayStatistics

// End of implementation file.
Using Class Sphere

#include <iostream.h.>
#include "Sphere.h"
using namespace std;

int main()
{
    Sphere unitSphere; // radius is 1.0
    Sphere mySphere (5.1); // radius is 5.1
    unitSphere.displayStatisticst();
    mySphere.setRadius(4.2); // resets radius to 4.2
    cout << mySphere.getDiameter() << endl;
    return 0;
} // end main
C++ Namespaces

- Namespace is used to logically group declarations and definitions of a problem solution into a common declarative region, i.e., namespace.
- The contents of a namespace can be accessed by code inside or outside the namespace.
  - From inside: access directly, e.g.,
    - using namespace smallNamespace
    - count +1;
  - From outside: by using the scope resolution operator, e.g.,
    - smallNamespace:: count +1;
- “using namespace std”
  - The statement indicates that one can use the elements in the Standard Library.