Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010
Chapter 4. Using Abstract Data Types
Abstract data type (ADT): A type defined in terms of its behavior. (Rather than its representation, e.g., `char` is represented by codes.)
Abstract data type (ADT): A type defined in terms of its behavior. (Rather than its representation, e.g., `char` is represented by codes.)

Separating behavior from implementation

- Simplicity. Hiding internal representation from the client.
- Flexibility. Implementation can be changed as long as the interface (behavior) remains the same.
- Security. Protect the implementation from the client.
Abstract data type (ADT): A type defined in terms of its behavior. (Rather than its representation, e.g., `char` is represented by codes.)

Separating behavior from implementation

- Simplicity. Hiding internal representation from the client.
- Flexibility. Implementation can be changed as long as the interface (behavior) remains the same.
- Security. Protect the implementation from the client.

Seven classes:
Vector, Grid, Stack, Queue, Map, Lexicon, Scanner.
Outline

1. Vector Class
2. Grid Class
3. Stack Class
4. Queue Class
5. Map Class
6. Lexicon Class
7. Scanner Class
8. Iterators
Vector class

A container class or collection class.

Interface

```
#include "vector.h"
```

A naming convention. For example: `grid.h`, `stack.h`.

Generalization of one-dimensional array type

- Variable size
- Effective size available
- Simple insert and delete
- Bound checking
Vector class (cont.)

Constructor

```
Vector<int> vec;
```

Specify the base type of a vector.
Constructor

```
Vector<int> vec;
```

Specify the base type of a vector.

Methods. Table 4-1, p. 127.

Example.
```
vec.add(10), vec.removeAt(0).
```
Constructor

Vector<int> vec;

Specify the base type of a vector.

Methods. Table 4-1, p. 127.

Example.

vec.add(10), vec.removeAt(0).

Question. How would you remove the last entry?
Idiom: Going through a vector, p. 129

```cpp
void PrintVector(Vector<int> & vec) {
    cout << "["
    for (int i = 0; i < vec.size(); i++) {
        if (i > 0) cout << ", "
        cout << vec[i];
    }
    cout << "]" << endl;
}
```

Note. Passing by reference.
**Idiom: Going through a vector, p. 129**

```cpp
void PrintVector(Vector<int> & vec) {
    cout << "[";
    for (int i = 0; i < vec.size(); i++) {
        if (i > 0) cout << ", ";
        cout << vec[i];
    }
    cout << "]" << endl;
}
```

Note. Passing by reference.

**Question:**

Can you pass `vec` by value (without the ampersand)? If you can, what are the differences?
Passing by reference

```cpp
void AddArrayToVector(Vector<int> & vec,
                       int array[], int n) {
    for (int i = 0; i < n; i++) {
        vec.add(array[i]);
    }
}
```
void AddArrayToVector(Vector<int> & vec, 
    int array[], int n) {
    for (int i = 0; i < n; i++) {
        vec.add(array[i]);
    }
}

Question:
Can you pass `vec` by value (without the ampersand)? If you can, what are the differences?
void AddArrayToVector(Vector<int> & vec, 
    int array[], int n) {
    for (int i = 0; i < n; i++) {
        vec.add(array[i]);
    }
}

Question:
Can you pass vec by value (without the ampersand)? If you can, what are the differences?

Almost always pass classes by reference.
void AskUserForInputFile(string prompt,
    ifstream & infile) {
    while (true) {
        cout << prompt;
        string filename = GetLine();
        infile.open(filename.c_str());
        if (!infile.fail()) break;
        cout << "Unable to open " << filename << endl;
        infile.clear();
    }
}

Note. Don’t forget `infile.close()` after reading/writing.
Idiom: Open a text file, p. 131

```cpp
void AskUserForInputFile(string prompt, 
                          ifstream & infile) {

    while (true) {
        cout << prompt;
        string filename = GetLine();
        infile.open(filename.c_str());
        if (!infile.fail()) break;
        cout << "Unable to open " << filename << endl;
        infile.clear();
    }
}
```

Note. Don’t forget `infile.close()` after reading/writing.

Study `revfile.cpp`, p. 130.

A text file as lines, an object of `Vector<string>`.
Grid class

Generalization of two-dimensional array.

- Variable dimensions.

Constructor

    Grid<double> matrix(3,2);

Specify row and column dimensions, in addition to the base type.

Methods. Table 4-2, p. 132
Example

CheckForWin for the tic-tac-toe game, p. 133.

```cpp
bool CheckForWin(Grid<char> & board, char mark) {
    for (int i = 0; i < 3; i++) {
        if (CheckLine(board, mark, i, 0, 0, 1)) return true;
        if (CheckLine(board, mark, 0, i, 1, 0)) return true;
    }
    if (CheckLine(board, mark, 0, 0, 1, 1)) return true;
    return (CheckLine(board, mark, 2, 0, -1, 1));
}
```

- check rows
- check columns
- check diagonal
- check antidiagonal
Stack class

Behavior: Last in, first out (LIFO). Only the top is accessible to the client.
Fundamental operations: push, pop
Stack class

Behavior: Last in, first out (LIFO). Only the top is accessible to the client.

Fundamental operations: push, pop

Applications. Nested function calls:

```plaintext
main() {
    call function F
}

function F() {
    call function G
}
```
Function G is called last and returns first.
Stack class (cont.)

Constructor

```cpp
Stack<double> calculator;
```

Specify a base type.

Methods. Table 4-3, p. 135.
Stack class (cont.)

Constructor

```cpp
Stack<double> calculator;
```

Specify a base type.

Methods. Table 4-3, p. 135.

Example. Scientific calculator (HP C-13)

```
50.0 * 1.5 + 3.8 / 2.0
```

Reverse Polish notation (RPN):

```
50.0 ENTER 1.5 * 3.8 ENTER 2.0 / +
```
RPN and stack

When the **ENTER** button is pressed, the previous value is pushed on a stack.

When an operator button is pressed

- Pushing the previous value
- Popping two values
- Applying the operation to the two values
- Pushing the result on the stack
Example

50.0 ENTER 1.5 * 3.8 ENTER 2.0 / +

Stack content, p. 136.
Example

50.0 [ENTER] 1.5 [*] 3.8 [ENTER] 2.0 [/ +

Stack content, p. 136.

Question. What is the key sequence for

50.0 * (1.5 + 3.8) / 2.0

Think the stack content.
Example

50.0 ENTER 1.5 * 3.8 ENTER 2.0 / +

Stack content, p. 136.

Question. What is the key sequence for

50.0 * (1.5 + 3.8) / 2.0

Think the stack content.

Outline

1. Vector Class
2. Grid Class
3. Stack Class
4. Queue Class
5. Map Class
6. Lexicon Class
7. Scanner Class
8. Iterators
Behavior. First in, first out (FIFO). Only the head and tail are accessible to the client.

Fundamental operations: enqueue, dequeue

Constructor

```cpp
Queue<int> queue;
```

Methods. Table 4-4, p. 139.

Application. Printer queue.
Example

Check-out line simulation.

Models

- Discretize time to serialize events.
- Arrival process: Poisson distribution. Average probability of a customer arriving in a particular time interval.
  Parameter: ARRIVAL_PROBABILITY
  Implementation: RandomChance(ARRIVAL_PROBABILITY)

- Service time: Uniformly distributed within a range.
  Parameters: MIN_SERVICE_TIME, MAX_SERVICE_TIME
  Implementation: RandomInteger(MIN_SERVICE_TIME, MAX_SERVICE_TIME)
Check-out line simulation (cont.)

- **Simulating time.**
  
  **Parameter:** SIMULATION_TIME
  
  **Implementation:**

  ```java
  for (int t = 0; t < SIMULATION_TIME; t++) {
    if (RandomChance(ARRIVAL_PROBABILITY)) {
      queue.enqueue(t);
    }
    if (serviceTimeRemaining > 0) {
      serviceTimeRemaining--;
      if (serviceTimeRemaining == 0) nServed++;
    } else {
      totalWait = t - queue.dequeue();
      serviceTimeRemaining =
      RandomInteger(MIN..., MAX...);
    }
  }
  totalLength += queue.size();
  ```
Outline

1. Vector Class
2. Grid Class
3. Stack Class
4. Queue Class
5. Map Class
6. Lexicon Class
7. Scanner Class
8. Iterators
Map class

Behavior. An association between a key (tag) and an associated value (can be a complicated structure). A generalization of Vector.

Fundamental operations: put, get

Application. Symbol table, an association between a variable name and its value.
Map class

Behavior. An association between a key (tag) and an associated value (can be a complicated structure). A generalization of Vector.

Fundamental operations: put, get

Application. Symbol table, an association between a variable name and its value.

Constructor

\[
\text{Map<double> symbolTable;}
\]

Note. The base type is the type of value, not tag. For simplicity, the type of tag is always string.
Map class

Behavior. An association between a key (tag) and an associated value (can be a complicated structure). A generalization of Vector.

Fundamental operations: put, get

Application. Symbol table, an association between a variable name and its value.

Constructor

    Map<double> symbolTable;

Note. The base type is the type of value, not tag. For simplicity, the type of tag is always string.

Methods. Table 4-5, p.147.
Map class

Behavior. An association between a key (tag) and an associated value (can be a complicated structure). A generalization of Vector.

Fundamental operations: put, get

Application. Symbol table, an association between a variable name and its value.

Constructor

```
Map<double> symbolTable;
```

Note. The base type is the type of value, not tag. For simplicity, the type of tag is always string.

Methods. Table 4-5, p.147.

Example. Airport codes. Figure 4-6, p. 150.
Outline

1. Vector Class
2. Grid Class
3. Stack Class
4. Queue Class
5. Map Class
6. **Lexicon Class**
7. Scanner Class
8. Iterators
Lexicon class

Behavior. A list of alphabetically ordered words.
Fundamental operations: add, containsWord
Lexicon class

Behavior. A list of alphabetically ordered words.

Fundamental operations: add, containsWord

Constructors

```java
Lexicon wordList;
Lexicon english("EnglishWords.dat");
```

Note. No parameterized type (always string)

Formats of the data file

- text file, list of words, one word per line
- precompiled data file
Lexicon class (cont.)

Methods. Table 4-6, p. 152.

Check every possible two-letter combinations ($26^2$), if it is contained in `EnglishWords.dat`. 
Lexicon class (cont.)

Methods. Table 4-6, p. 152.

Example. twoletters.cpp, p. 153. Check every possible two-letter combinations ($26^2$), if it is contained in EnglishWords.dat.

Why Lexicon now that we have Map?
Lexicon class (cont.)

Methods. Table 4-6, p. 152.

Check every possible two-letter combinations ($26^2$), if it is contained in EnglishWords.dat.

Why Lexicon now that we have Map?
Efficiency.
Outline

1. Vector Class
2. Grid Class
3. Stack Class
4. Queue Class
5. Map Class
6. Lexicon Class
7. Scanner Class
8. Iterators
Scanner class

Behavior. Divide up a string into tokens

- A sequence of consecutive alphanumeric characters, or
- A single-character string consisting of a space or punctuation mark.

Fundamental operation: hasMoreTokens, nextToken

Constructor

Scanner scanner;

No base type. (Always string.)

Methods. Table 4-7, p. 157.
Idiom: Scan a file

```java
ifstream infile;
Scanner scanner;

AskUserForInputFile("Input file: ", infile);
scanner.setInput(infile);
while (scanner.hasMoreTokens()) {
    string word = scanner.nextToken();
    ... do something with the token ... 
}

infile.close();
```
Outline

1. Vector Class
2. Grid Class
3. Stack Class
4. Queue Class
5. Map Class
6. Lexicon Class
7. Scanner Class
8. Iterators
Iterators

Iterator: A subclass (of Vector, Grid, Map, Lexicon, Scanner).

Behavior. Stepping through the elements of a collection class.

Fundamental operations: hasNext, next
Iterators (cont.)

Idiom: iterator, p. 158

Lexicon::Iterator iter = english.iterator();
while (iter.hasNext()) {
    string word = iter.next();
    ... code to work with the word ... 
}

Lexicon::Iterator A subclass of Lexicon
iter An object of the class Lexicon::Iterator
iter.next() Returns a value of type string (Lexicon) or base type (Vector or Grid or Map).
foreach mechanism

Usage

**Idiom: foreach**

```cpp
foreach (string word in english) {
    if (word.length() == 2) {
        cout << word << endl;
    }
}
```
**foreach mechanism**

**Usage**

**Idiom: foreach**

```csharp
foreach (string word in english) {
    if (word.length() == 2) {
        cout << word << endl;
    }
}
```

It is simple and easy to use, but you should understand the mechanism. The type of `word` (string) must match the base type of the class (Lexicon) of which `english` is an object.