Abstract Data Types
2501ICT/7421ICT

René Hexel
School of Information and Communication Technology
Griffith University

Semester 1, 2012
Outline

1 Abstract Data Types – Collections
   - Collection Types
   - Collection Operations

2 Collection Implementations
   - Linear Collections
Collections

- ADTs and Data Structures
- Collection Categories
- Common Collection Operations
- Traversal
- Serialisation
- Collection Implementations
- C, Objective-C
Abstract Data Type
Describes a collection of data items and the associated operations that can be applied
High-level concept of data organisation (what)
Data Structure
Physical Implementation of an ADT
How to represent ADT concept
There is no ‘best’ Implementation under All Conditions
Definition

- a Collection is a *group of items* forming a conceptual unit

Collections are

- represented by ADTs, and
- implemented through Data Structures
Collection Categories

- Linear
  - Arrays, Lists, Stacks, Queues, ...
- Hierarchical
  - Heaps, Trees, Hashes, ...
- Connected
  - Graphs
- Unordered
  - Sets, Bags, Maps, ...

→ Play an Important role in almost all Non-Trivial Programs!
Linear Collections

- Low Level Properties
  - Array/List operations
  - e.g. dynamic array, singly/doubly linked list

- High Level View
  - Stack/Queue/... functionality
  - e.g. Push-Down Stack, Priority Queue, Pipe
Hierarchical Collections

- Binary Tree
- Binary Search Tree
- Generic Tree
- Heap
- Red/Black Tree
- ...

Diagram:

```
    D1
   /   \
 D2    D3
```
Graphs

- Undirected Graph
- Nondirectional
Graphs (2)

- Directed Graph
  - Directional

Diagram:

- $D_1$
- $D_2$
- $D_3$
- $D_4$
- $D_5$
Unordered Collections

- Set
- Bag
- Map
  - Dictionary
  - Table
Common Operations

- Search and Retrieval
  - given certain search criteria (search properties)
  - return item or position (if found)
  - return distinguishing value like nil or −1 if not found

- Removal
  - delete a given item
  - delete item at specific position
Common Operations (2)

- **Insertion**
  - add a new item
  - usually at some particular position
  - e.g. at head, at tail, after item \( x \), . . .

- **Replacement**
  - combination of Removal and Insertion
  - ‘in place’ replacement
  - when atomic action is required
Common Operations (3)

- **Traversals**
  - visit each item
  - “do something” with that item

- **Test (the whole collection)**
  - for equality
  - greater than, more elements
  - less than, fewer elements
Common Operations (4)

**Size**
- Number of items
- Byte size

**Cloning**
- ‘deep copy’
- copy an entire collection
- each item needs to be cloneable!
Need for Sequential Traversal
- ADTs differ in Data Organisation
- Enumerator (iterator) is required
- Object or Function that makes traversal possible
- `nextObject for NSEnumerator`
  - return next item and advance
  - return nil if no next item exists
- `++ for STL iterators`
- `begin() and end()` methods mark start and beginning
Casting

- **Problem**: Collections may contain objects of any type.
- Cast to required subclass necessary in C++ for non-virtual methods!
  - unchecked: `(string)` – also works in C, Objective-C
  - checked: `static_cast<string>` – C++ only
- **Primitive types** (e.g. int) need a wrapper
  - e.g. `NSNumber` in Objective-C
  - e.g. `intValue` and `setIntValue`: access methods
- **Limited compile-time type checking!**
- **Type of object** can be tested during run-time using `isKindOfClass:`
Casting in plain C

- No ‘Collection’ Infrastructure
- Implement your own or use add-on library
- Use `void *` for generic objects
- Cast to Required ‘Object’ Pointer
- Primitive types (e.g. `int`) may need a wrapper
- No compile-time type checking – type of object cannot be tested during run-time!
- Casting to wrong pointers causes crashes!
Serialised stream representation of each object

**NSCoding** Objective-C Protocol for write/read
- `encodeWithCoder:` and `initWithCoder:`
  - Abstract set of Methods (**NSCoder**)
    - Structured Files and XML: **NSKeyedArchiver**, **NSKeyedUnArchiver**
    - Network Connections: **NSPortCoder**

No API support in C++
- traverse STL collection using **iterator**
- read/write data using **stream classes**
Example

```c
#import <Foundation/Foundation.h>

int main(int argc, char *argv[]) {
    @autoreleasepool {
        NSArray *array = @[NSArray arrayWithObjects: @"1", @"2", @"3", @"4", nil];
        /*
        * save the array to disk, to a file called "Array.bin"
        */
        [NSKeyedArchiver archiveRootObject: array toFile: @"Array.bin"];  

        return EXIT_SUCCESS;
    }
}
```
Example (prints: `Array: 1 2 3 4`)

```c
#import <Foundation/Foundation.h>

int main(int argc, char *argv[]) {
    @autoreleasepool {
        /* load the array from disk (from the "Array.bin" file) */
        NSArray *array = [NSKeyedUnarchiver unarchiveObjectWithFile: @"Array.bin"];

        NSEnumerator *e = [array objectEnumerator];
        id object;
        printf("Array: ");
        while ((object = [e nextObject]) != nil) {
            printf("%s ", [[object description] UTF8String]);
            putchar('
');
        }

        return EXIT_SUCCESS;
    }
}
```
Example

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
    vector<int> array(5, 0); // an array of ints
    for (int i = 0; i < 5; i++) array[i] = 2*i; // initialise

    /*
     * write vector to a file "Vector.txt"
     */
    fstream outputFile("Vector.txt", fstream::out);
    vector<int>::iterator enumerator = array.begin();
    while (enumerator != array.end())
       outputFile << *enumerator++ << endl;
    outputFile.close();

    return outputFile.fail() ? EXIT_FAILURE : EXIT_SUCCESS;
}
```

René Hexel
# Loading Data in C++

**Example (prints: 0 2 4 6 8 )**

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
    vector<int> array(0); // an array of ints
    string line;

    fstream inputFile("Vector.txt", fstream::in); // read vector from file
    while (getline(inputFile, line)) {
        int element = atoi(line.c_str()); // convert line to int
        array.push_back(element); // add to array
    }

    vector<int>::iterator enumerator = array.begin();
    while (enumerator != array.end()) // traverse array
        cout << *enumerator++ << " ";
    cout << endl;

    return !inputFile.eof() ? EXIT_FAILURE : EXIT_SUCCESS;
}
```
Machine-Independent Files

- Use Text-Based File Format
  - e.g. XML

- Write: Traverse each Element
  - print out content into stream, e.g. using `fprintf()`, `-encodeWithCoder:`, ...

- Read: Parse input Data ⇒ e.g. `NSScanner`
  - create Data Structure as you go
  - add each one to your collection
Saving XML Data in Objective-C

Example

```c
#import <Foundation/Foundation.h>

int main(int argc, char *argv[]) {

    @autoreleasepool {

        NSArray *array = @[@"1", @"2", @"3", @"4", nil];
        NSDataType *data = [NSMutableData new];
        NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc] initForWritingWithMutableData: data];
        [archiver setOutputFormat: NSPropertyListXMLFormat_v1_0];
        [archiver encodeObject: array forKey: @"root"]; // create XML
        [archiver finishEncoding];
        [archiver release];

        [data writeToFile: @"Array.xml" atomically: YES]; // write file
        [data release];

    }

    return EXIT_SUCCESS;
}
```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    @autoreleasepool
    {
        NSKeyedUnarchiver *array = [NSKeyedUnarchiver unarchiveObjectWithFile: @"Array.xml"];

        NSEnumerator *e = [array objectEnumerator];
        id object;
        printf("Array: ");
        while ((object = [e nextObject]) != nil)
        {
            printf("%s ", [[object description] UTF8String]);
            putchar('
');
        }
    }

    return EXIT_SUCCESS;
}
ADT Implementations

- **High-level Decisions**
  - accessing head/tail only?
  - random access needed?
  - map/dictionary functionality needed?

- **Low-level Decisions**
  - arrays (static vs. dynamic)
  - linked lists (linked data structures)
  - hash maps
ADT Implementations (2)

- Often: Multiple Implementations
  - time/space tradeoff
- Linear Collections
  - arrays
  - linked lists
  - hash maps
- Hierarchical Collections
  - different linkage models
Arrays

- One of the Most Commonly Used Low Level Data Structures
- Access Elements by Index Position
- Index Operation is Very Fast
- Constant time to access any element
- Element position does not affect access speed
Physical Array Size

- Capacity (max. size) of an Array
  - C: use sizeof (size in Bytes!)
    ```c
    char *students[100];
    sizeof(students) / sizeof(students[0]) = 100;
    ```
  - Objective-C
    ```objc
    transparent, but NSMutableArray can be optimised for a given capacity!
    NSMutableArray s = [NSMutableArray arrayWithCapacity: 100];
    ```
  - C++ `vector` class is also transparent, but optimised for a fixed size
Logical Array Size

- Number of Valid Items
- e.g. 4 items have been added to the array:

\[ D_1 \rightarrow D_2 \rightarrow D_3 \rightarrow D_4 \]

- A Dedicated Counter Variable is needed to Keep Track of the Size
Adding an Item to an Array

- Check if logical size equals physical size
- If so, we need to Increase the size of the array:
  - Create a new, larger array
  - Copy old array to new array
  - Refer the old array variable to the new array
  - Add the Item to the new array
Resizing in Plain C

Example (expanding an array in C)

```c
struct Object *temp;
int i;

if (logsz == physz)
{
    physz++;
    temp = malloc(physz * sizeof(struct Object)); /* allocate new memory */

    for (i=0; i < physz; i++) /* copy old array to new */
        temp[i] = array[i];

    free(array); /* free old array */
    array = temp; /* use new array */
}
```
Resizing is Costly
  - complexity shoots up from O(1) to O(n)
    ⇒ resize less often

Don’t just add 1, but double the size each time:
```c
physz *= 2;
```
Decreasing Size

- Similar to Increasing
- Frees up wasted space
  - Create a temporary, smaller array
    - As costly as increasing
      - Don’t decrease too often!
- Good strategy: decrease only if the logical size is less than \( \frac{1}{4} \) of the physical size
  - Decrease only by \( \frac{1}{2} \) to leave room for adding elements again
Inserting an Item

- Check Available Space
- Resize if necessary: $O(n)$
- Shift Items from target index to logical end one index down: $O(n)$
Removing an Item

- Shift Items from target index + 1 to logical end one index up: O(n)
- Check Wasted Space
- Decrease size if necessary: O(n)

\[
\begin{array}{cccc}
D_1 & D_2 & D_3 & D_4 \\
D_1 & D_2 & & D_4 \\
D_1 & D_2 & D_4 & \\
\end{array}
\]
Array Problems

- Insertions and Deletions incur some overhead
- Shifting items to open or close a gap
- Copying all items when resizing
- $O(n)$ Complexity in Time and Space
- Only efficient if mostly static!
Array Problems (2)

- Require Contiguous Memory
- Expensive for large data structures
- 1:1 Correspondence Between
  - logical position of a cell and its physical position in memory
- Decouple Logical/Physical Pos.
  → linked data structure
Linked Data Structures

- Consist of Elements Called Nodes
- A Node Contains
  - The actual data
  - One or more links to other nodes
- Dynamic Data Structures
- Memory is allocated only as needed
- Can be freed immediately if unneeded
Singly Linked Lists

Illustration:

Accessing a Node (an Item)
- Follow the links from the Head
- Last item has a null link
- Dummy link indicating the end of the list
Nodes

Illustration of a Node

A Node stores
- a Pointer to another Node (link)
- an Object (the actual data)
Singly Linked Structures

- Start With a Null Pointer (*nil* or *NULL*)

- Called the Head Pointer or an External Pointer
- Contains no data!
**Singly Linked Structures (2)**

- Add the first Node

```
head  link  NULL
  \    /    \
   D_1  
```

- Node Contains Actual Data
- Let head point to this first node
- Node itself points to NULL link as next node
Singly Linked Structures (3)

- Add the second Node
  
  ![Diagram of singly linked structures](image)

  - Then add third, fourth, etc.
  - All Nodes contain their own data and ➔ a NULL Pointer when added at the tail
  - ➔ Pointers get updated as new Nodes are added
Removing a Node

- To delete a Node

- Aim predecessor (or head) pointer at following node
- Release used up memory
Singly Linked List Complexity

- Adding an Item: $O(n)$
- Deleting an Item: $O(n)$
- Linear Search: $O(n)$
- Binary Search: $O(n \log n)$
- No direct access possible!
- Cache pointers
- Reduce frequent operations to $O(1)$