School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Abstract Data Types

Previously on ADDS

- Dynamic Array
  - Dynamic arrays are created using the *new* operator.
  - They are used like ordinary arrays.
  - Remember to call *delete[]* when your program is finished with the dynamic arrays.
- Multi-Dimensional Array
  - Ordinary 2-d array
  - Dynamic 2-d array

```
dataType arrayName[rowSize][columnSize];
```

- Pointer to Pointer
- Passing Array to Functions
- ADT

Abstract Data Types

A data type is called an ADT if the programmers who use the type have no access to the details of the implementation.

- Easy
- Safe

Separation

- Separate the specification of how the type is used from the details of implementation.
- Class abstraction
- Rules:
  - Make all member variables private
  - Make the basic operations public
  - Make any helping functions private

Interfaces

- The set of public member functions in our class
  + a description of what they do

- This should be all that someone needs to know to use your ADT.

ADTs and Black Boxes

- You can't see inside it.
- All a user can see is their interface.
- A user shouldn't NEED to know more than the interface.
  - Do you know how std::string or '+' are implemented?

- This is also known as *information hiding*.

Benefits

1. Can change Implementation without needing to change the usage.
2. Easy to manage a team of programmers working on the same project.
3. You can break your project down into smaller tasks
   - makes debugging much easier.
4. Safe

More Complex Data Structures

- Another really good application of ADTs is in controlling access to more complex data structures.

- Vector? Vector provides bound checking on the at() function

  – What is the key difference between the `[]` of the array and the `at()` of the `Vector` class?

- How could you implement a `Vector`?

Vector

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
vector <int> myvector(10);
myvector[0] = 1; // can use array notation

myvector.at(1) = 3; //Or 'at'

myvector.front() = 5; //1st element = 5

//To add an 11th element
myvector.push_back(64);

for (int i = 0; i < myvector.size(); i++)
        cout << i << ": " << myvector[i] << "\n";
  }
cout << "size"<< ": " << myvector.size() << "\n";

 for (int i = 0; i < myvector.size(); i++){
  cout << i << ": " <<myvector.at(i) << "\n";
 }
}
```

Example

- Define a circular queue
- We can do this in arrays, without any extra ADTs.
- What are the benefits and drawbacks of this approach?

The Circular Array

- Let's design and build a circular array class together.
  - One view: as we keep inserting elements, we "wrap around" to the beginning of the array

- Technically, this is an example of modular arithmetic: no matter how big the number we're dealing with is, we want the reference to fit into the range 0..(n-1)
  - How can we achieve this?

The Circular Array

- What's missing from a user's point of view?
- Does addElement have a problem?
- How can we fix it?
- Does the user of this ADT need to know about it?
- We will discuss the above points during the lecture.

Test

- How can we test it?

- A few things you can think about:
  - Normal functionality
  - Boundary value
  - Special cases
  - Error handling

Summary

- In this lecture, we refined our ideas as to what an ADT is and how to make a class an ADT.
- This kind of design and implementation activity is crucial in developing good data structures to solve problems.


THE UNIVERSITY
of ADELAIDE