Abstraction & Containers

COSC1076
Semester 1 2019
Week 06

Admin

- Assignment 2 Preparation
  - Start forming groups
    - Groups registered by Week 7
  - Group size 4
  - Groups formed in Lab classes
    - There will be updates in labs

Data Structures

RMIT UNIVERSITY

Data Structures

- Data Structures are a core component of good software design
  - Choosing the right data structure makes significant difference to performance, execution, memory consumption, etc.
  - It is necessary to have a very good understanding of the strengths and weakness of each of the structures

Data Structures

- There are only a small number of very fundamental data structures from which more complex structures are derived.
- These structures are:
  - Arrays
  - Lists
  - Trees
  - Maps
- You have probably been using a number of these structures
- C++ Provides STL containers for a number these structures
  - Vector
  - Deque
  - Map

Data Structures

- In COSC1076 we will cover in detail:
  - Array
  - Linked Lists
- C++ Provides STL containers for a number these structures
  - Array - Fixed sized array
  - Vector - Dynamically sized array
  - List - Linked List
  - Deque - Double-ended Linked List
- Other data structures will be examined in Algorithms & Analysis

Primitive Arrays

- So far we have used primitive arrays.
- Advantages
  - Fast
  - Direct Random Access
  - Continuous sequential storage in memory
  - Efficient retrieval from RAM, and efficient for Cache
- Disadvantages
  - Fixed Memory size. Array's are not extensible and require new memory to be allocated if a large array is required
  - Inefficient to add or remove elements from the array

Abstract Data Types (ADTs)

RMIT UNIVERSITY

Monolithic Code

- At its most extreme, monolithic code consists of one giant main function!
  - This is clearly not a good thing to do!
- The essential algorithm for the application is intermixed with the algorithms for the data structures used by the program.
  - Even if multiple functions have been used, the program may still be characteristically monolithic as the essential algorithm and the data structure algorithms are intermixed.
- This makes it difficult to:
  - Modify or reuse a data structure.
  - Maintain software
  - Understand any of the software

Modular Software

- Modular software separates:
  - Algorithms
  - Data Structures
  - Third-party tools
  - etc.
- The modular approach divides a large program into small components
- The modular approach assists with:
  - Managing complexity of solution
  - Maintainability
  - Correctness!
  - Reusability
  - Distribution of workload (team of programmers)

Abstract Data Types (ADTs)

- An abstract data type (ADT) consists of:
  1. An interface
     - A set of operations that can be performed.
     - Usually contained in a header file
  2. The allowable behaviours
     - The way we expect instances of the ADT to respond to operations.
- The implementation of an ADT consists of:
  1. An internal representation
     - Data stored inside the object's instance variables/members.
  2. A set of methods implementing the interface.
     - Usually contained in a code file
  3. A set of representation invariants, true initially and preserved by all methods

Interface

- An ***interface*** specifies what a user can do with the module.
- It defines available:
  - Types
  - Constants
  - Methods/Functions
  - Plus, any ***restrictions*** and/or ***assumptions***.
- The user must use the module ***only*** via the provided interface
  - Doing otherwise compromises the ability of the module author to change the module implementation.

C++ STL Containers

RMIT UNIVERSITY

Array

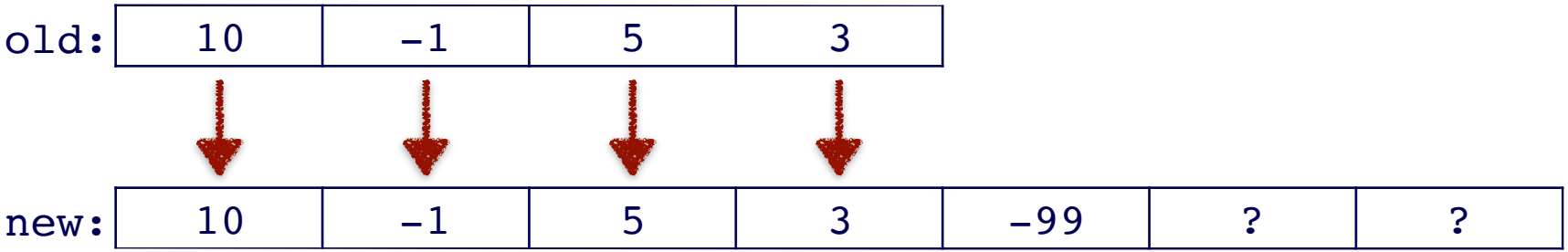
- Fixed sized array
  - Wrapper around a primitive C++ array
  - Provides generic accessor methods
  - Provides iterators (discussed Week 11)
  - Provides compatibility with C++ STL features
- Documentation: https://en.cppreference.com/w/cpp/container/array

Vector

- Dynamically sized array
  - Starts with a fixed sized array
  - Grows if the array “runs-out” of space
  - Provides generic accessor methods
  - Provides iterators (discussed Week 11)
  - Provides compatibility with C++ STL features
- Documentation: https://en.cppreference.com/w/cpp/container/vector

Vector Resizing

- To resize requires:
  - Creating a new larger memory space
  - Then copying the contents of the old array into the new array



Arrays vs Vectors

	Array	Vector
Element Types	All elements are same type	All elements are same type
Resizeable	No	Yes Requires Copy
Storage Layout	Contiguous	Contiguous
Cell lookup	Constant Time Random Access	Constant Time Random Access
Inserting Elements	Linear Time Requires Copy	Linear Time Requires Copy
Removing Elements	Linear Time Requires Copy	Linear Time Requires Copy

C++ Generics

- To use many of the C++ STL containers, you need to be familiar with how C++ Implements ***Generics***
  - It is similar to Java generics
- A generic class (or function) has ***incomplete*** and ***parameterised*** type(s) in its declarations and definition
  - The parameterised type(s) are ***instantiated*** when an object of the generic class is declared
  - The parameterised type(s) are placed inside angle-brackets < >
  - Unlike Java, any valid type may be used including:
    - Primitive types
    - Pointers & References
    - Classes

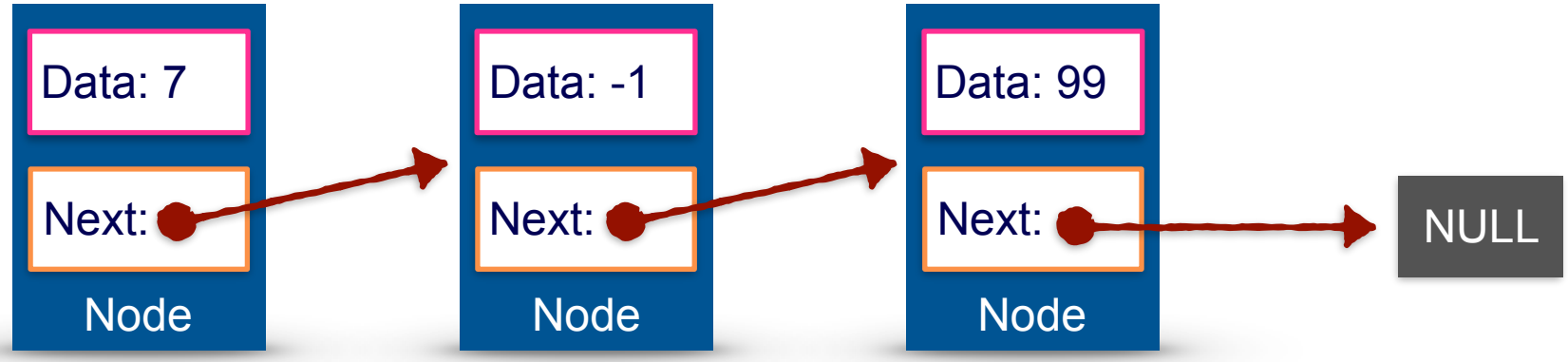
```
vector<int>    iVector;
vector<int*>   ptrVector;
vector<Class>  classVector;
```

Linked Lists

RMIT UNIVERSITY

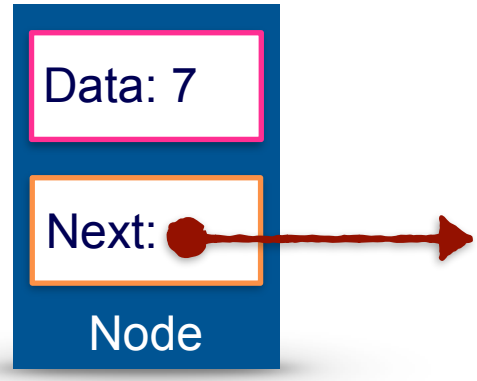
Linked List

- A ***Linked List*** is a data types that:
  - Stores elements of the same type
  - Stores the elements as a sequential “chain”
  - The last element of the chain is “null"



Linked List - Node sequence

- The core of a linked is a sequence (or “chain”) of ***nodes***
- A Node:
  - Stores a single element
  - Has a pointer to the “next” node in the sequence/chain



Node Class

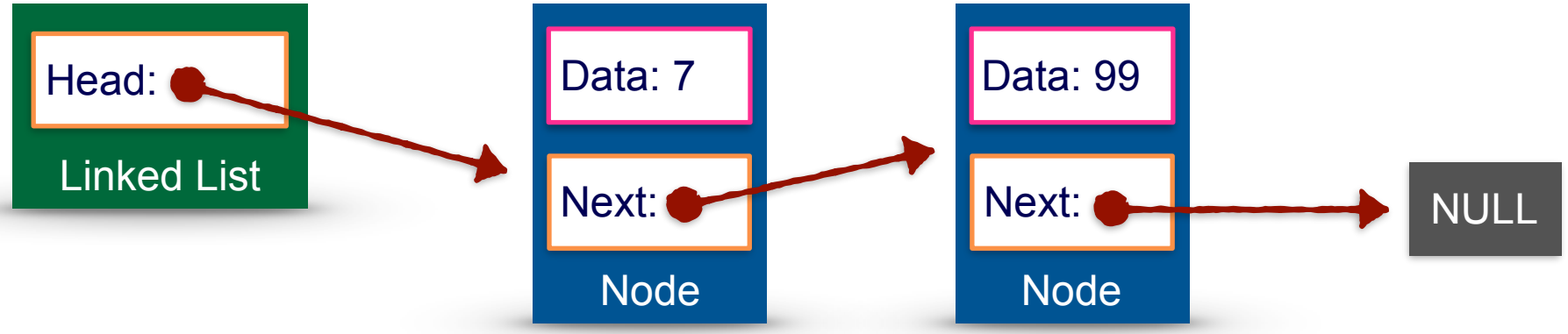
- A Node is represented as:

```
class Node {
public:

   Node(int data, Node* next);
   Node(Node& other);

   int   data;
   Node* next;
};
```

- Which specifies a sequence/chain, that is ***list***, of integers

Linked List ADT

- To use the list of nodes, a ***Linked List ADT*** is created which:
  - Internally contains a pointer to the ***head*** (first node) of the list
  - Defines methods to interact with the linked list
    - These methods are similar to those used on the ParticleList!



LinkedList ADT

- A Linked List is represented as:

```
class LinkedList {
public:
   LinkedList();
   ~LinkedList();

private:
   Node* head;
};
```

- Which specifies the head (first node) of the list

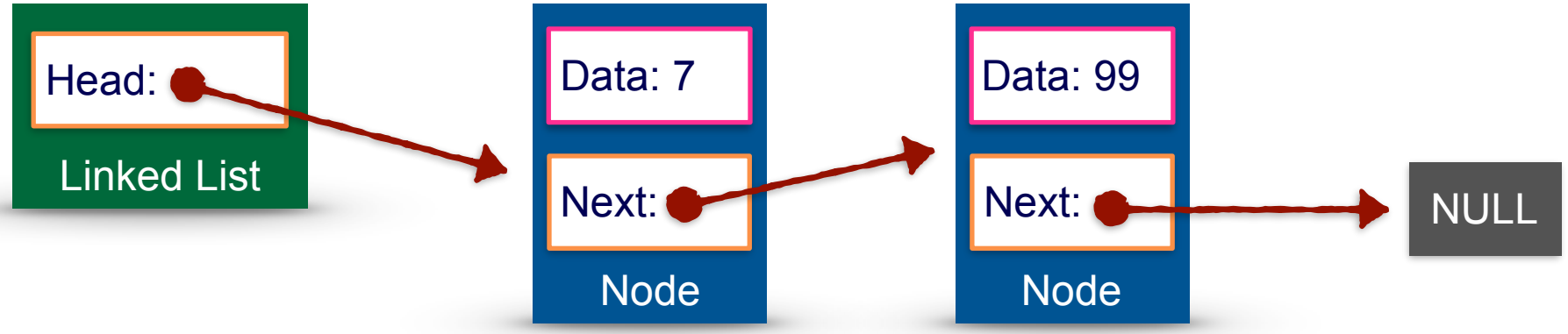
LinkedList ADT

- The ADT methods for a Linked List may include:

```
class LinkedList {
public:
   ...

   int size();
   void clear();
   int get(int i);

   void addFront(int data);
   void addBack(int data);

   ...
};
```

Linked List ADT

- This is the simplest form of a linked list. More complex lists might also:
  - Store a count of the number of nodes in the list
  - Have a pointer to the end of the list
  - And much more!



Implementing Linked List ADT

- In the rest of this lecture we will implement some of the ADT methods
- The golden rule:
  - ***Draw the diagram first!***
  - If you can draw the diagram, then it makes it easier to implement the code

Linked Lists vs Arrays/Vectors

	Array/Vector	Linked List
Element Types	All elements are same type	All elements are same type
Resizeable	Yes Requires Copy	Yes
Storage Layout	Contiguous	Non-Contiguous
Cell lookup	Constant Time Random Access	Linear Time Random Access

Linked Lists vs Arrays/Vectors

	Array/Vector	Linked List
Insert at Front	Linear Time Requires Copy	Constant Time
Insert at Back	Constant Time	Linear Time (if no back ptr) Constant Time (back ptr)
Delete at Front	Linear Time Requires Copy	Constant Time
Delete at Back	Constant Time	Linear Time (if no back ptr) Constant Time (back ptr)

Debugging

- Programs are either correct or wrong
  - So Debugging isn't the best term…
- The best debugging technique is to not write errors in the first place!
- Ideally all code would be correct the first time it is written
  - In practice, this is not the case
  - Debugging is about finding *every error* in a program

Static Techniques

- Static techniques involve inspecting code *without* executing the code
- Code Style
  - It is easier to spot errors in code that is well formatted
  - This is why we are pedantic about code style
- Static Code Analysis
  - Read source code and to follow it the way the computer does (ie. mechanically, step-by-step) to see what it really does.
  - Take the time to understand the error before you try to fix it. Remember the language rules and the software requirements before fixing bugs
  - Explain your code to someone else!
- Proving Code Correctness
  - Using static mathematical analysis, prove that a code block is correct

Dynamic Techniques

- Dynamic techniques find errors by executing the code
- Test-then-Develop paradigm to software development
  1. Decide upon the purpose of a code-block
  2. *First* develop a series of tests for that code-block
  3. Write the code
  4. Evaluate the code against the pre-developed tests
- Writing tests *before* writing the code is critical
  - This maintains the independence of the tests
  - You do not want the tests to be influenced by the development process
  - It is even better to use a third-party developer to write your tests!

Dynamic Techniques

- Black-box testing
  - Testing based on Input-Output
  - A test:
    - Provides input to a code-block
    - Provides the expected output for the code-block
  - The tests pass if the output precisely matches the expected output
- Unit Tests
  - Strictly unit tests target a specific function, module or code-block
  - Same as Test-then-Develop paradigm with Black-box testing

Dynamic Techniques (GDB / LLDB)

- A program may be inspected “live" as it is executed
  - Effectively the execution of the program is paused
  - The program state may be inspected
  - The evaluation of the code may be incrementally “stepped” through.
- The C/C++ live-debugging tool is GDB
  - Program is LLDB on MacOS

Dynamic Techniques (GDB / LLDB)

- gdb is a “symbolic” debugger which means that you can examine your program during execution using the symbols of your source code.
  - These symbols do not get saved by compilation by default. To save the symbols, compile with the -g option

```
g++ … –g –o …
```

- gdb can be run with the command:

```
gdb myprog
```

- gdb can also be used to inspect a program state immediately after a segmentation fault has occurred
- Once gdb is started, it has a number of commands to control the dynamic execution of the code
  - Type ‘h’ (help) to get a summary of the commands

Dynamic Techniques (GDB / LLDB)

- After you start up gdb you can issue commands to gdb to control, monitor and modify your program's execution. There are a large number of commands available. Here are some as example (the abbreviation for each command is shown in square brackets):

  `[l]ist` – display source code

  `[br]eak <function> | linenum`

  - sets a breakpoint at 'function' or at linenum of your source code
  - when the program reaches this breakpoint, execution is suspended and you can examine your program using other gdb commands

- For example:

  `br myFunc` – sets breakpoint at function 'myFunc'

  `br 120` – sets breakpoint at line no. 120

RMIT
UNIVERSITY