Murdoch
UNIVERSITY

Data Structures and Abstractions

Encapsulation & Linked Lists & 2D Structures/Encapsulated Data structure & Abstract class and Interface

Lecture 6

Records using `struct`

- C++ records (structs):

```
typedef struct
{
    string firstname;
    string surname;
    int    age;
} Person;
```

Murdoch UNIVERSITY

Records using `class`

- Encapsulating a record in a class:

```
class Person //any special behaviour for Person?
{
...
private:
    string firstname;
    string surname;
    int    age;
};
```

Murdoch University

When to Encapsulate?

- The question is, which should you use?
- If there are *any* input processing or output methods to be performed on a data structure *or* it is composed of other objects, then it should be encapsulated. [1]
- And, of course, if you encapsulate things in a class, then you can test all the methods and operators *in isolation* before having to combine the code with the rest of your program. UNIT TEST [1]

Murdoch UNIVERSITY

Arrays vs Lists

- We know how to declare and use “raw” array. [1]
- We have looked at how to declare and use a list.
- The main differences are:

An array has an initial size	A list starts with 0 size
It is difficult to change the size of an array	lists automatically resize as they grow
Arrays have no inbuilt functions	lists have lots of inbuilt functions

- Obviously the list is better when it comes to memory use.

Murdoch UNIVERSITY

When to Encapsulate

- The rule for records also holds true for arrays and lists. As long as they are data stores that require little in the way of processing, then you can just use them as is.
- However, if you need to do bounds checking or have any processing that needs doing, *or* they contain objects, then they should almost certainly be encapsulated.
- And, of course, if you encapsulate things in a class, then you can test all the methods and operators *in isolation* before having to combine the code with the rest of your program. (as before). ALWAYS UNIT TEST before use in your program.

Murdoch UNIVERSITY

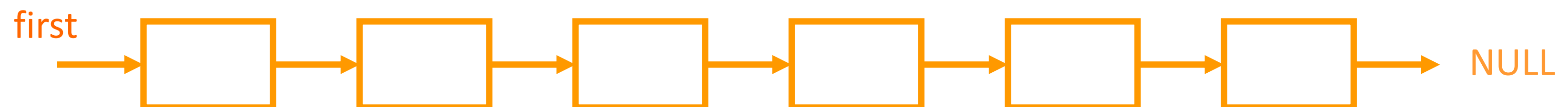
Advantages of Encapsulation

- The class can be tested in isolation before being used in a program. UNIT TEST
- Changes and new code can be tested in isolation before being used in a program. UNIT TEST
- This means that the *testing* of the program becomes modular and hence easier, and more likely to be thorough.
- Which in turn means that programs are more likely to be robust and errors are easier to find.
- It is easier to re-use code.
- Bounds checking is done in one file.
- Code is less complicated, and therefore easier to maintain.
- It becomes easy to alter *how* something is done without altering the main (client or user) program.
- It becomes easy to alter how something is stored without altering the main (client or user) program
- Memory can be more easily allocated dynamically in a safe manner.

Murdoch UNIVERSITY

Linked Lists

- We took a first look at linked lists in an earlier lecture note.
- Linked lists are an abstract class that model a particular type of behaviour. In a linked list, you have:
  - each node contains data and a pointer;
  - the data can only be accessed in a serial manner from the previous piece of data;
  - access to the container as a whole is done via the first element;
  - the last element must point to NULL (nullptr) to ensure algorithms cannot process past the end of the list.



The Linked List

- We looked at the Node class (data plus pointer).
- A linked list class simply contains a Node or pointer to a Node.
- If it contains an actual node, it makes processing easier, but wastes the space of that Node.
- The node is called a ‘dummy header’ as it stores no actual information (data which the other nodes store).

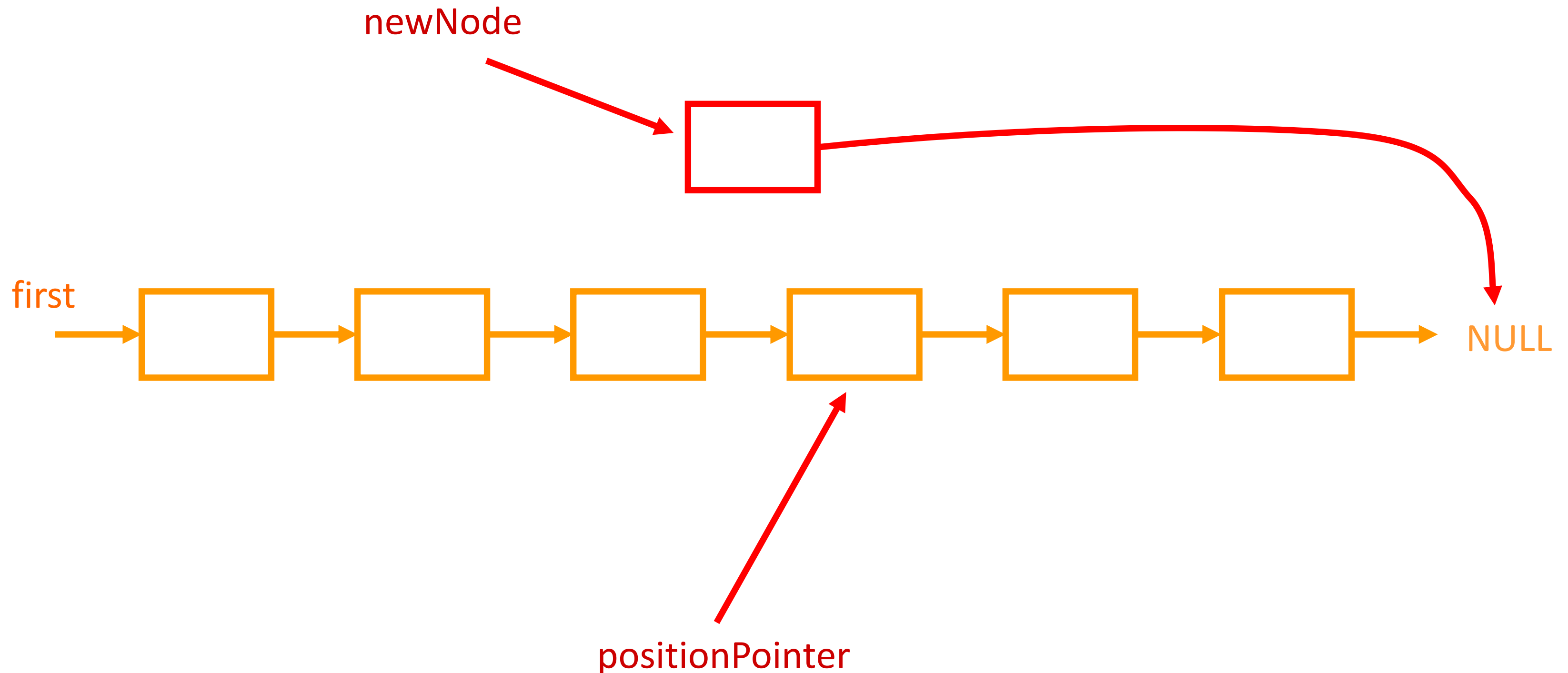
Linked Lists vs Arrays

- Linked lists are containers as are arrays/lists.
- Unlike an array/list, you cannot access data directly in a linked list.
- Therefore access to an array element is done in constant time, but to a linked list element takes O(n).
- However, if you want to insert or delete into an array it takes O(n) time, whereas with a linked list it takes constant time.

Murdoch University

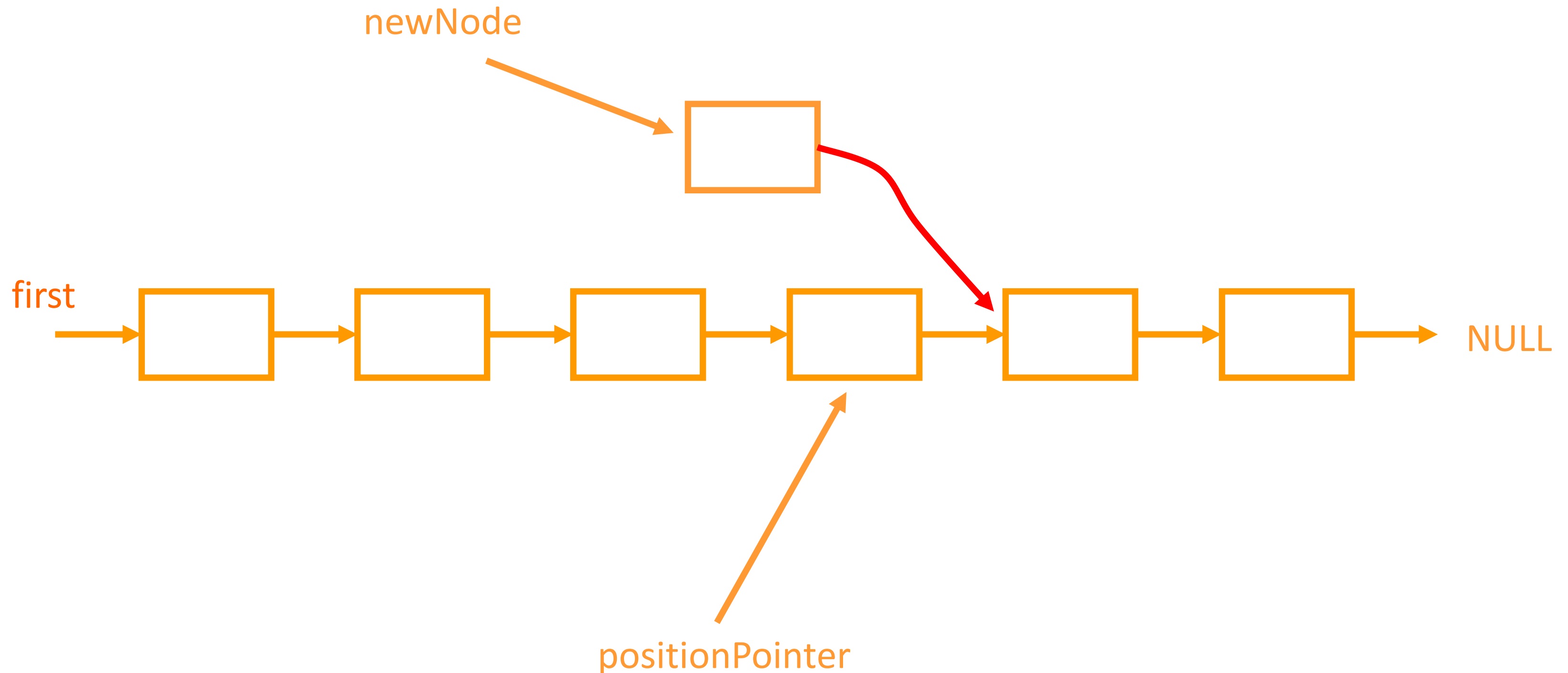
Insertion into a List [1]

- Locate node in front of the insertion point



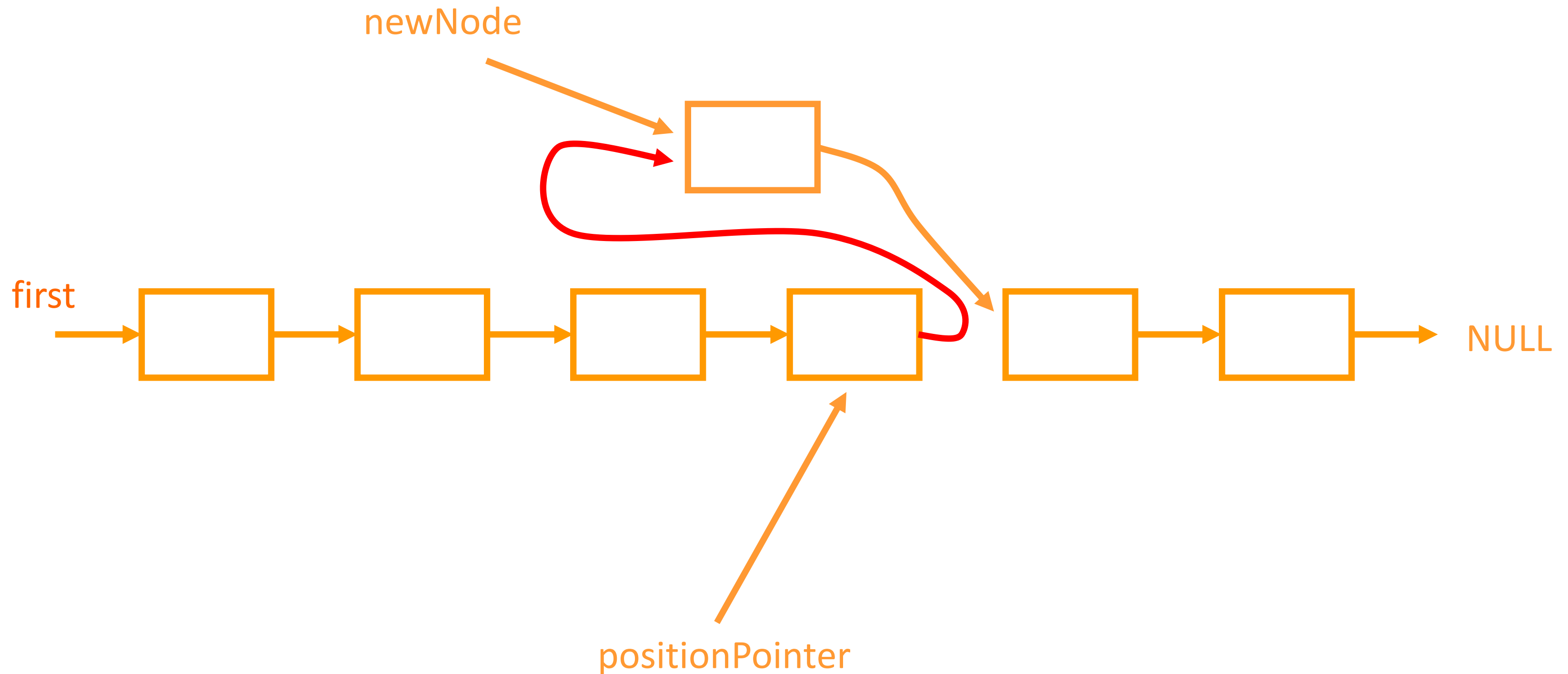
Insertion into a List

- Reassign the ‘next’ pointer of the new node



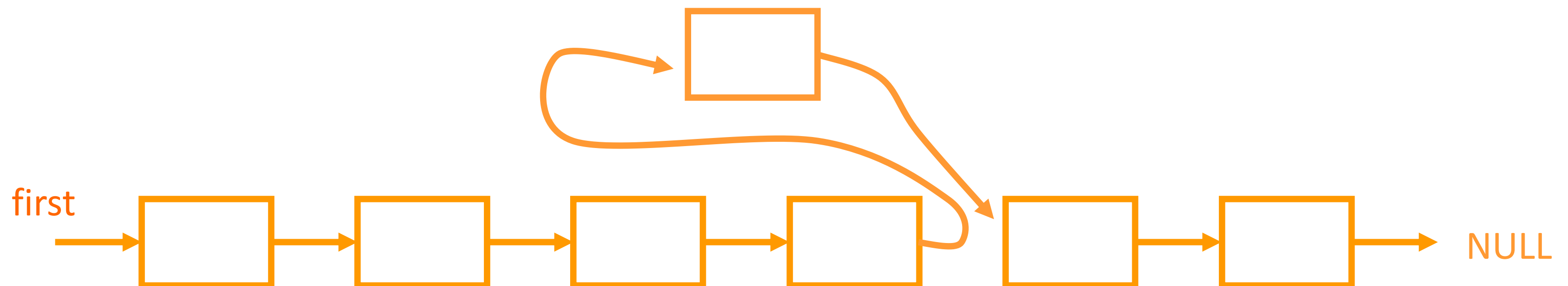
Insertion into a List

- Reassign the ‘next’ pointer of the node in front of the new node



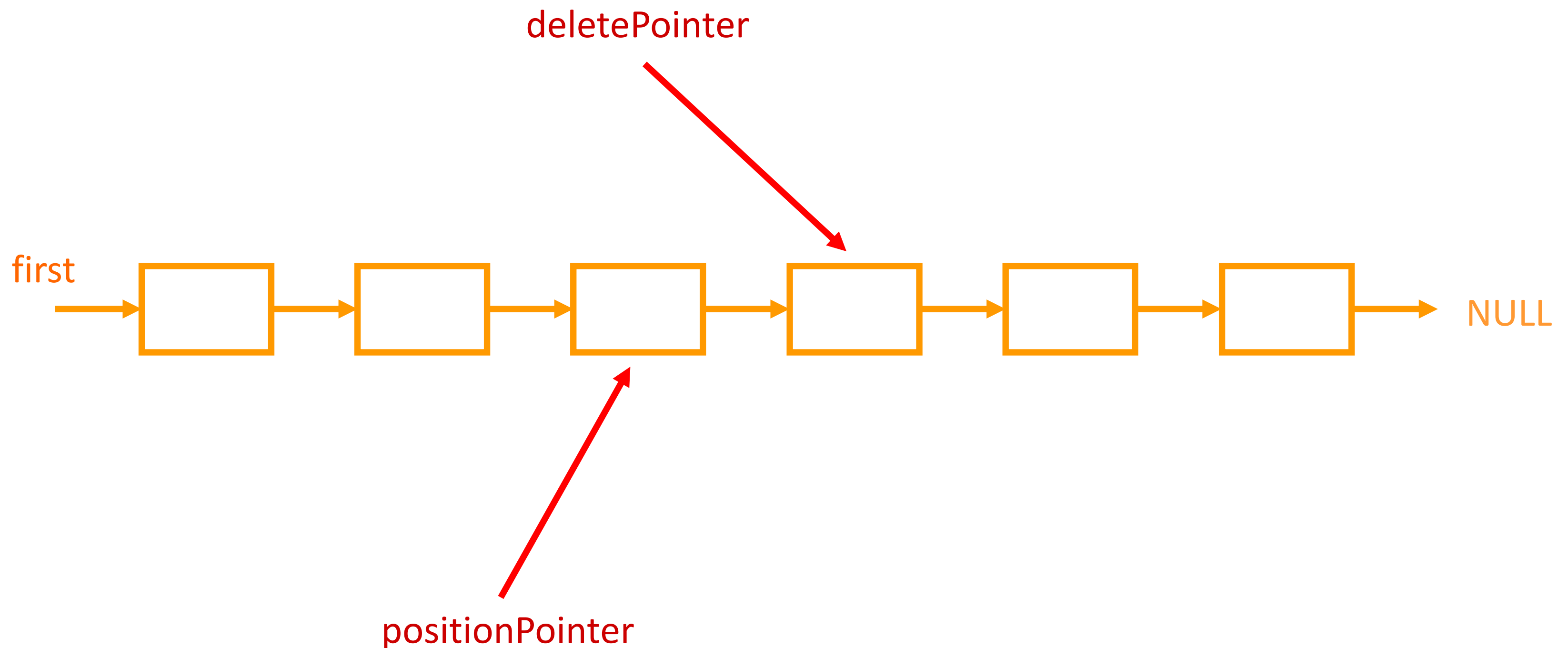
Insertion into a List

- The two other pointers are no longer needed as the node is now part of the list.



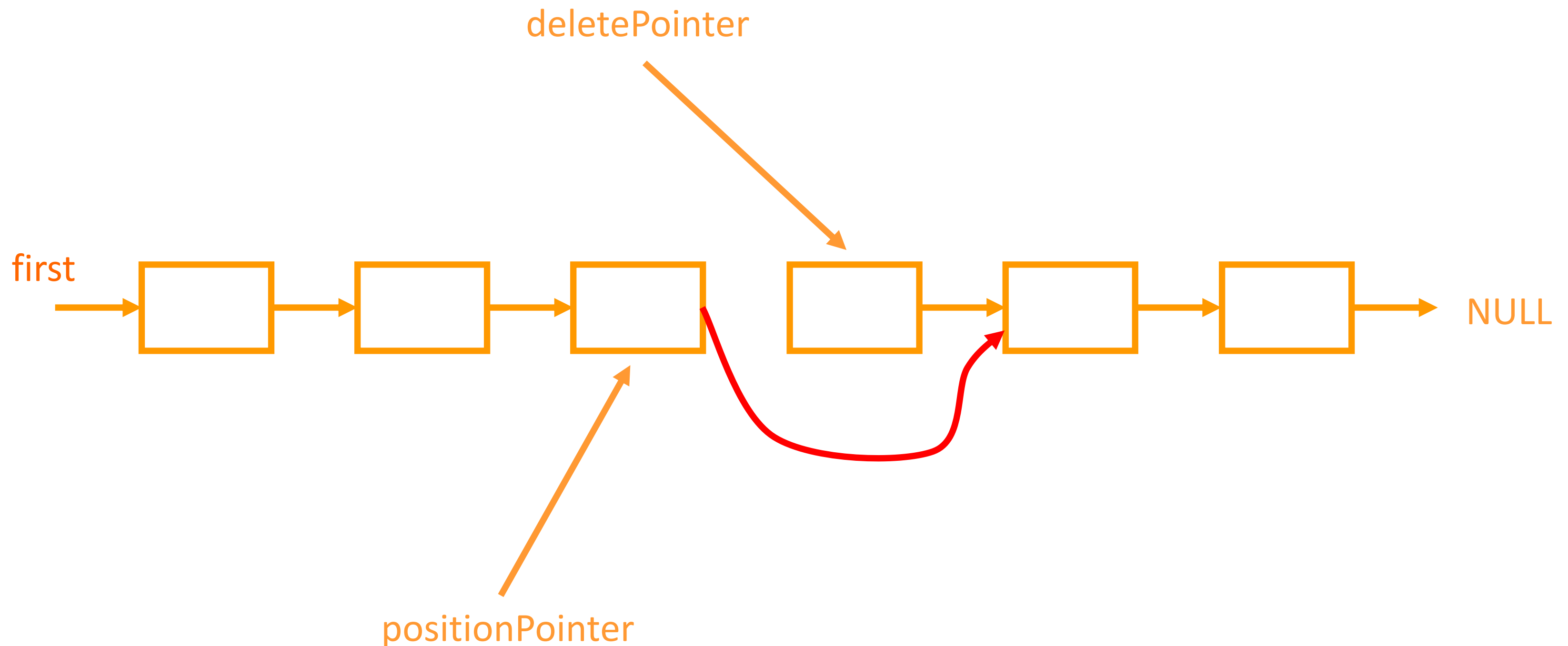
Deletion from a List

- Locate the node in front of the node to be deleted, as well as the node to be deleted.



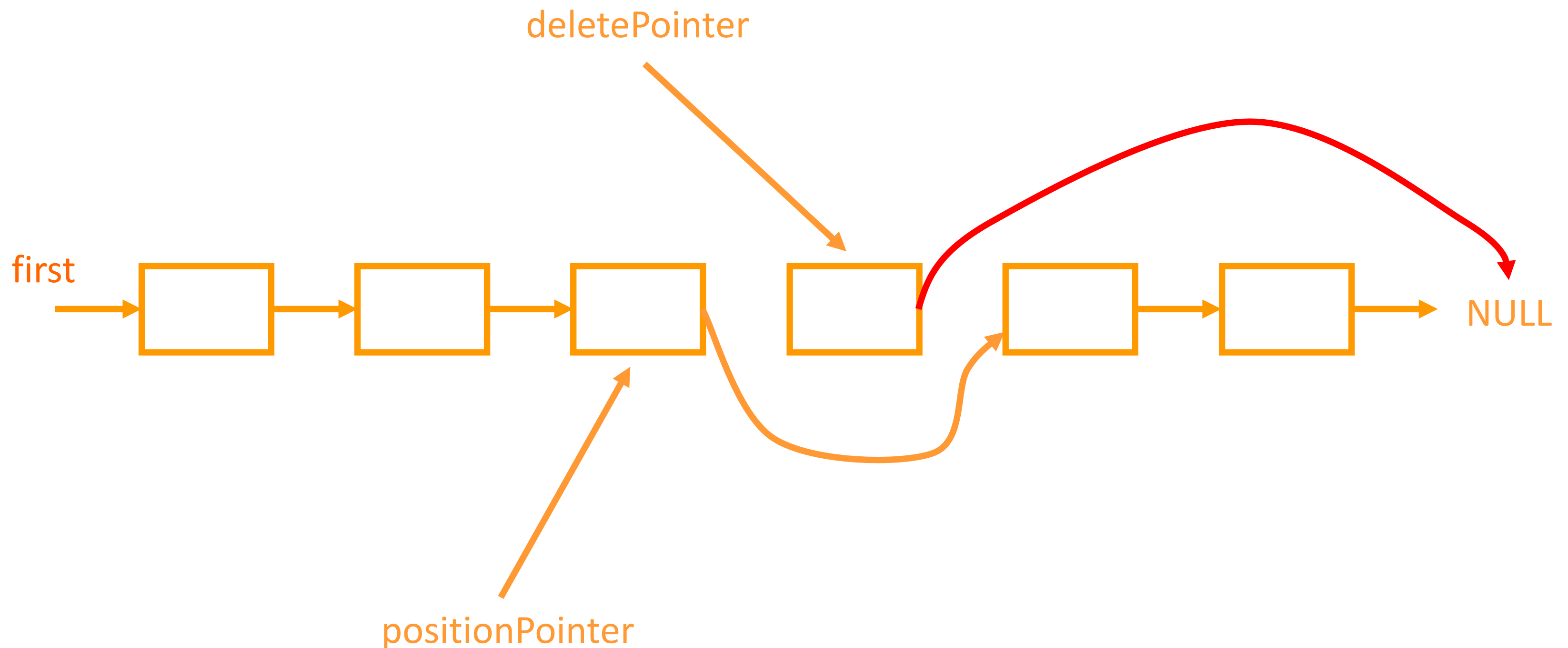
Deletion from a List

- Reassign the 'next' pointer of the node in front of that to be deleted.



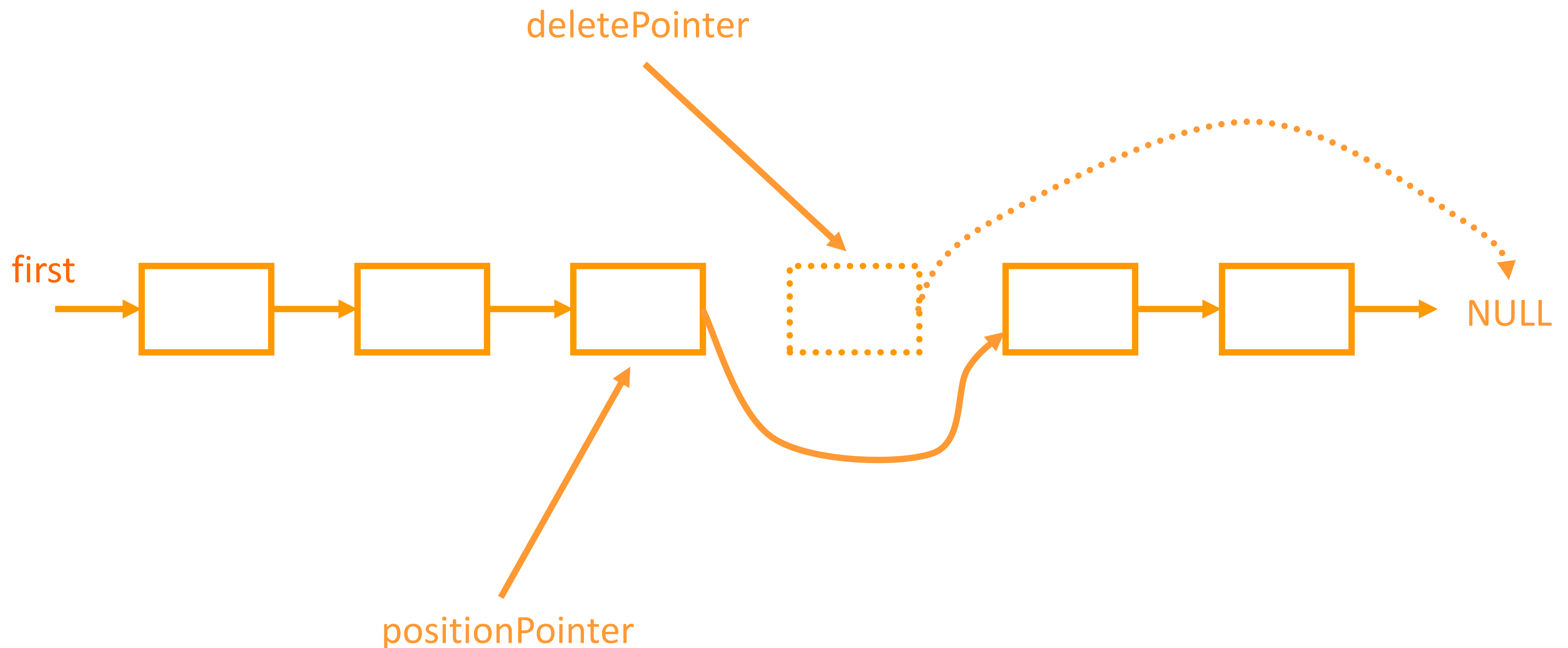
Deletion from a List

- Reassign the ‘next’ pointer of the node to be deleted, setting it to NULL.



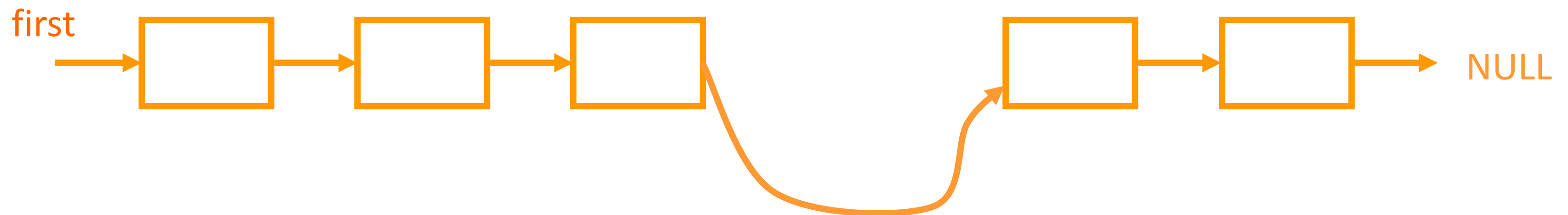
Deletion from a List

- Release the old node's storage back to the OS, using 'delete'.



Deletion from a List

- The two pointers are no longer needed as the node is no longer part of the list.



The STL List

- The STL has a linked list template.
- Just as the vector replaces the array, the list template replaces ‘home-coded’ linked lists. [1]
- As with the list, sometimes it needs to be encapsulated, sometimes it doesn’t.
- If you find yourself repeating code that accesses a linked list, then encapsulate it!
- The list requires the <list> header file.
- It is declared in the same way as a list:

```
typedef list<int> IntList;
IntList mylist;
```

Murdoch UNIVERSITY

STL list Methods

Method	Description
mylist.clear ()	Empties the list.
mylist.empty ()	Returns true if the list is empty.
mylist.erase (<various>)	Erases a part of the list.
mylist.insert (<various>)	Add data to the list.
mylist.push_back (data)	Add one piece of data to the end of the list.
mylist.pop_back ()	Delete the last item in the list.
mylist.push_front (data)	Add one piece of data to the front of the list.
mylist.pop_front ()	Delete the first item in the list.
mylist.begin()	Returns an iterator that points to the first item in the list.
mylist.end()	Returns an iterator that points to just after the last item in the list.
mylist.size()	Returns the size of the list.
mylist.sort()	Sorts the list.
mylist.swap (mylist2)	Swaps the contents of the two lists.

Murdoch UNIVERSITY

Seem Familiar?

- Yes, these are almost exactly the same methods as listed for the STL std::vector class.
- The huge advantage of the STL is that the classes all have almost identical methods and operators.
- There are a few that are unique to one or other class, but on the whole they are the same.
- Here, the two that are in list and not in vector are push_front, pop_front and sort.
- Almost all of the STL classes can also all be passed to the same algorithms in the algorithm class.
- If they can't then the compiler will soon let you know!

Murdoch UNIVERSITY

Advantages of Encapsulation Again

- Lets suppose you want a container of Lights.
- When you first code it you use a vector of Lights as the data structure.
- After a while you realise that a linked list would be a better container and you decide to change to a list.
- If you had not encapsulated it, you now have to go through *possibly* thousands of lines of code in multiple files to alter it from a vector to a list.
- If you encapsulated it, you probably only have to change a few lines in only two files. This is because the underlying container would have been private and all the other code in the various files would not have direct access to it.
  - If you designed your encapsulation well, there would be no need to change the public access methods just because the underlying container was changed from a vector to a list.
- A very big time saver!!

Murdoch UNIVERSITY

Readings

- Textbook: Chapter on Linked Lists.
  - Go through the programming Example on video store at the end of the chapter.
- Chapter on Standard Template Library
- Re-read chapter 1 “The Object Oriented paradigm in Design Patterns Explained: A New Perspective on Object-Oriented Design. See Topic 1 readings. Available as an ebook from the library.

Murdoch UNIVERSITY

Further exploration

- For a more details of linked lists with some level of language independence, see the reference book, Introduction to Algorithms section on “Linked Lists” in the chapter on “Elementary Data Structures” (10).
- For more on STL containers see http://www.cplusplus.com/reference/stl/

Murdoch UNIVERSITY

Two Dimensions

- Two dimensional structures are complicated.
- Therefore they should always be encapsulated.
- This also gives great freedom in how they should be implemented.
- And great freedom to change the implementation if required.
- Some possibilities are:
  - an old-fashioned two dimensional array
  - an array of vectors
  - a vector of arrays
  - an array of lists
  - etc
  - in other words an array/list/vector of array/list/vector

Murdoch UNIVERSITY

Which One?

- The choice will depend on the what you are trying to model.
- Ask yourself:
  - Do you know the dimensions in advance?
  - Are there always going to be the same number of columns in each row?
  - Does there need to be a set number of rows, even if there is nothing in each row?
  - Will you need to add/delete rows or columns at the ends?
  - Will you need to insert/delete rows or columns in the middle?
  - Will you need to insert/delete a single piece of data at the end of a single row?
  - Will you need to insert/delete a single piece of data in the middle moving along the other data in that row only?
  - Do you need direct access to the data?
- When you can answer all these questions, you will be able to choose the correct combination of data structures for the task.

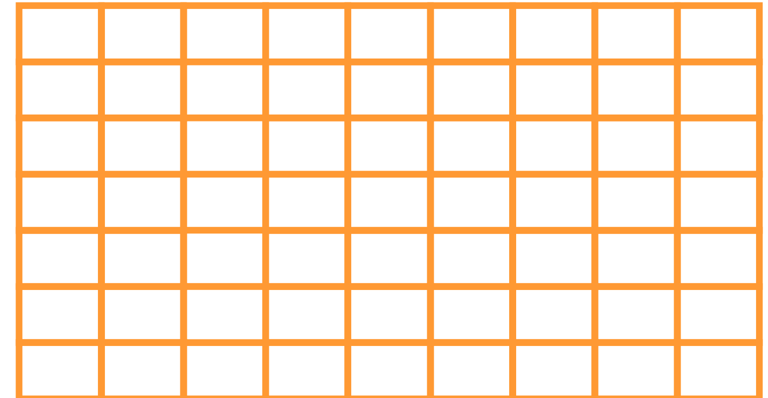
Murdoch UNIVERSITY

An Old Fashioned 2D Array

- // A two dimensional array of DataType objects
- typedef DataType TableType[ROWS][COLS]; // [1]

- ...

- class Table
- {
- public:
- ...
- private:
-     TableType m_array;
- }



Uses exactly ROWS x COLS slots of the size of DataType

Possible Application
Icon Storage, but can be anything else

Murdoch UNIVERSITY

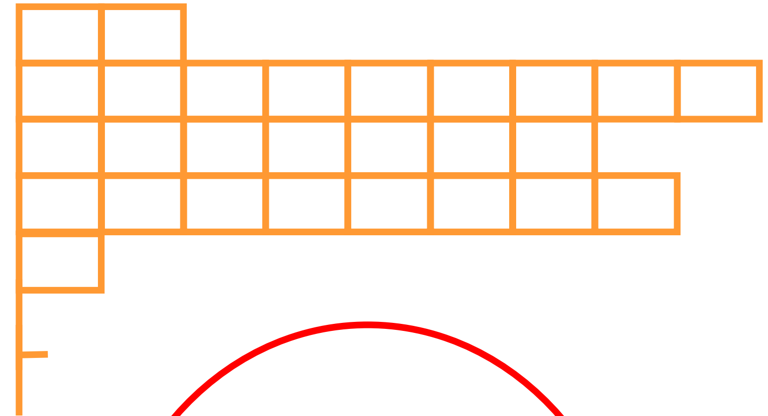
An Array of Vectors

```
// A vector of DataType objects
typedef vector<DataType> Row;

// Rows of these vectors
typedef Row TableType[ROWS];

...

class Table
{
public:
...
private:
    TableType m_array;
}
```



Each row can be a different size, but there are still exactly ROWS number of rows, even if some are empty.
The STL vector takes care of the rows.

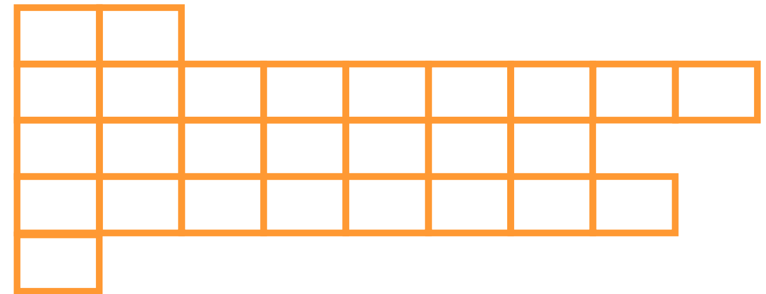
Possible Application
Accumulator for a fixed rows and variable columns

A Vector of Vectors

- // A vector of DataType objects
- typedef vector<DataType> Row;

- // vector of these vectors
- typedef vector<Row> TableType;

- ...



- class Table
- {
- public:
- ...
- private:
-     TableType m_array;
- }

Possible Application
Accumulator for an unknown number of items

Each row can be a different size, and now we only have the rows we actually want. Variable columns

Murdoch UNIVERSITY

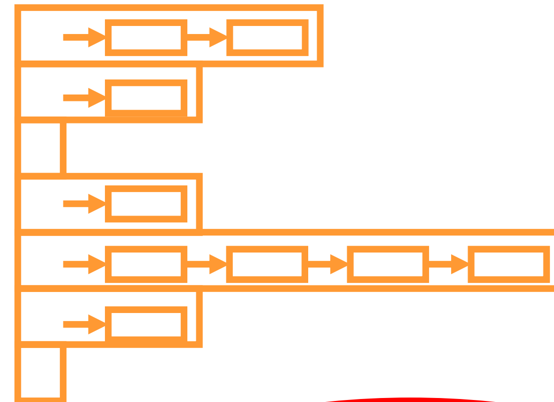
An Array of Lists

```
// A list of DataType
typedef list<DataType> DTlist;

// An array of these lists
typedef DTList TableType[ROWS];

...

class Table
{
public:
...
private:
    TableType m_array;
}
```



Possible Application
Lists of students in units

Each list initialises itself, so this structure is safer than the last few. There are ROWS number of lists, which may or may not be the best structure.

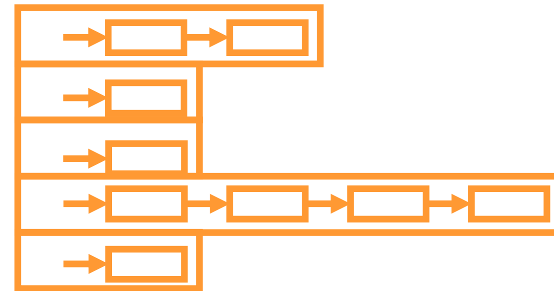
A Vector of Lists

```
// A list of DataType
typedef list<DataType> DTlist;

// A vector of these lists
typedef vector<DTList> TableType;

...

class Table
{
public:
...
private:
    TableType m_array;
}
```



Possible Application
Lists of students grouped under country of origin

Once again we only have the number of rows required. Grow as needed

Murdoch UNIVERSITY

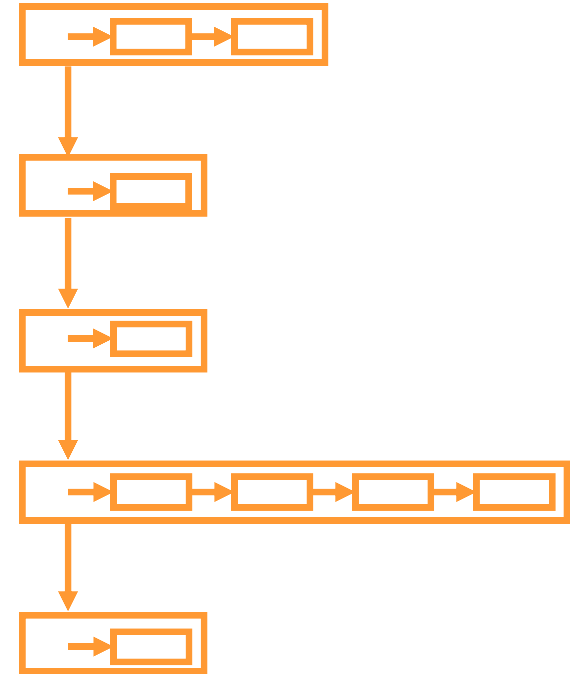
A List of Lists

- // A list of DataType
- typedef list<DataType> DTlist;

- // A list of these lists
- typedef list<DTList> TableType;

- ...

- class Table
- {
- public:
- ...
- private:
-     TableType m_array;
- }



Possible Application
A list of people on the carriages of a train.

Complicated but versatile!

Murdoch UNIVERSITY

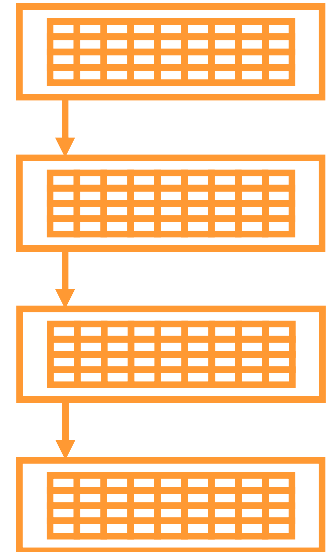
A List of Arrays

```
// A list of DataType
typedef DataType Array2D[ROWS][COLS];

// A list of these lists
typedef list<Array2D> TableType;

...

class Table
{
public:
...
private:
    TableType m_array;
}
```



Possible Application
A simulation of the seats in the carriages of a train.

This can be considered to be a three dimensional structure, and should be coded as a list of (e.g.) Carriages.

Murdoch UNIVERSITY

Etcetera!

- The possibilities are 2^n, ,where n is the number of different types of 1D structure.
- Choosing the right one is the only difficulty.
- But if you encapsulate it, changing your mind only costs some time and effort within 2 files the interface (header) and implementation (source) files.
- Change when not encapsulated could mean a great deal of work indeed!

Murdoch UNIVERSITY

Full Encapsulation

- Layering the encapsulation makes maintenance easier and easier.
- Therefore rather than making the inner layer the raw container type, it would be a class.
- As would the outer layer.
- As well as making maintenance easier, it will make processing simpler and clearer.
- And, of course, the structure is clearer.

Murdoch UNIVERSITY

Readings

- Textbook: Chapter on Arrays and Strings, sections on Parallel Arrays, Two and Multidimensional Arrays.

Murdoch UNIVERSITY

Background – Data Type

- Data Type
  - Has a name
  - Has a set of values
  - Has a set of operations on these values
- Data Types
  - “atomic” data types
    - values are “not” decomposable, e.g. Integer
  - Data Structures
    - Values are decomposable
    - Values are related, e.g array of integer

Murdoch UNIVERSITY

Background – Data Type

- Abstraction of Data Types
  - Abstract Data Type (ADT)
    - Product of our imagination
    - Only essential properties – no details of implementation
  - Virtual Data Type (VDT)
    - Exists on a virtual processor – e.g. Programming language
  - Physical Data Type (PDT)
    - Exists on the machine – the machine representation
- VDTs implement ADTs
- PDTs implement VDTs

Murdoch UNIVERSITY

Background – Data Type

	Abstract	**Virtual**	**Physical**
Atomic	Number of chairs	C++ or Java .. etc integer	“series” of bits
Structured	List of chairs	C++ or Java .. etc Array of classes or structs	“series” of bytes

At the Abstract level, you do **not** think of the Programming Language.

When you are doing OO design, think at the Abstract level. You want your classes (at the virtual level of abstraction) **mimicking the Abstract level**.

Murdoch UNIVERSITY

Abstract Classes

- Do not forget the big picture on what is "Abstraction" covered earlier. When we are considering Abstract Classes in C++, we are considering the virtual level of abstraction. The fact that the word "virtual" is used – see later – can be helpful but can also be a source of confusion. [1]
- When one class inherits from another class, a method might be replaced. In the parent class the method is designated a virtual method:

```
virtual DoSomething (); // polymorphic method
```

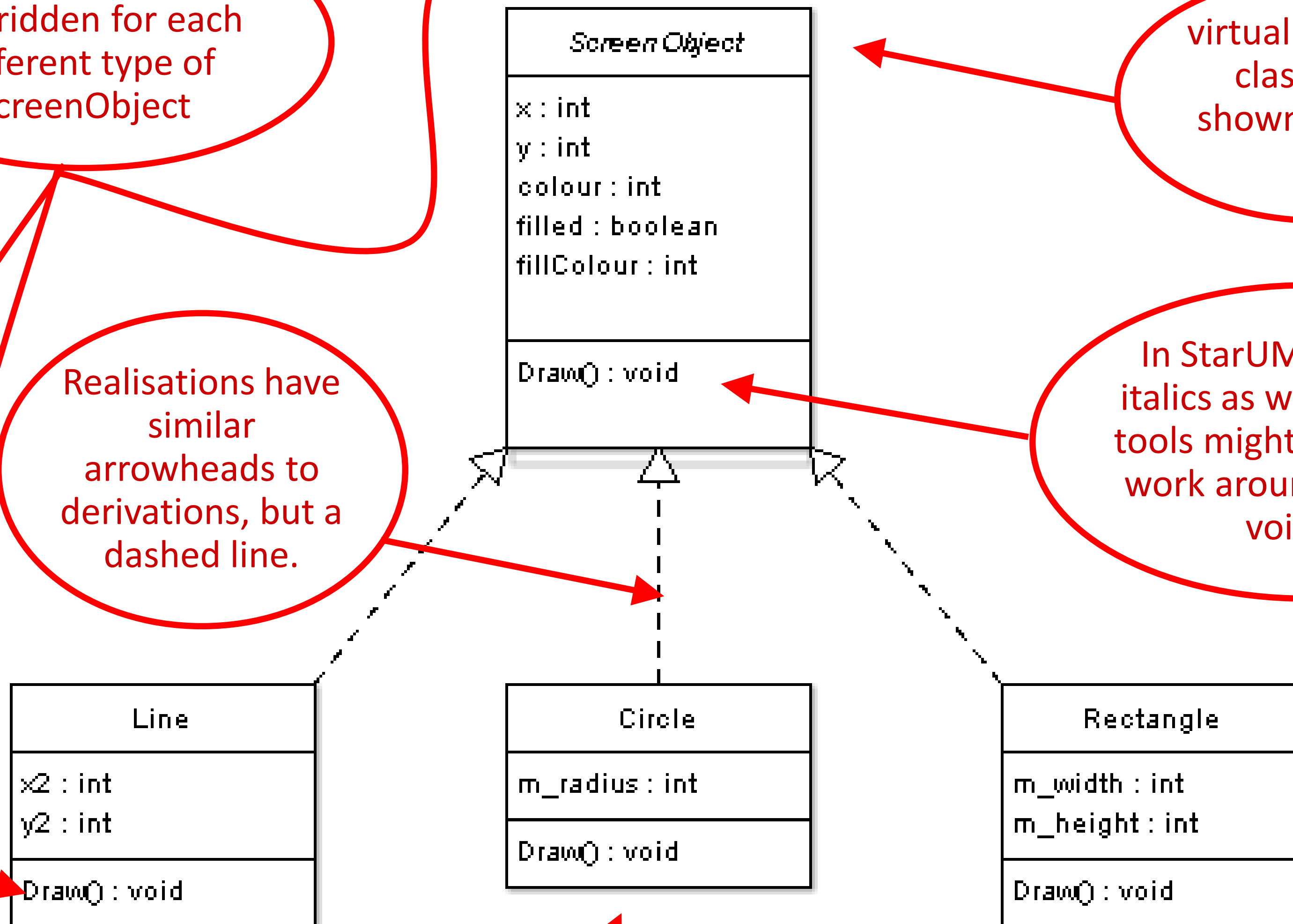
- If the method in the parent class is to be replaced, but is not actually to be defined in the parent class, then the virtual method must become a 'pure virtual method':

```
virtual DoSomething () = 0; // parent doesn't
  have a code body
```

  - Any class that contains pure virtual methods is—by default—an abstract (pure virtual) class: it cannot ever be instantiated as an object because there is missing code body.
- In UML, virtual classes are indicated by using italics for the class name, and the relationship of the derived classes is called a 'realisation'. [2]
- In C++ realisations are implemented using inheritance in the same way as are derivations but with dashed line.

Murdoch UNIVERSITY

Pure Virtual Classes in UML



The Draw() method is overridden for each different type of ScreenObject

virtual (abstract) classes are shown in italics [2]

Realisations have similar arrowheads to derivations, but a dashed line.

In StarUML, this is in italics as well but other tools might not do it, so work around by saying void [1]

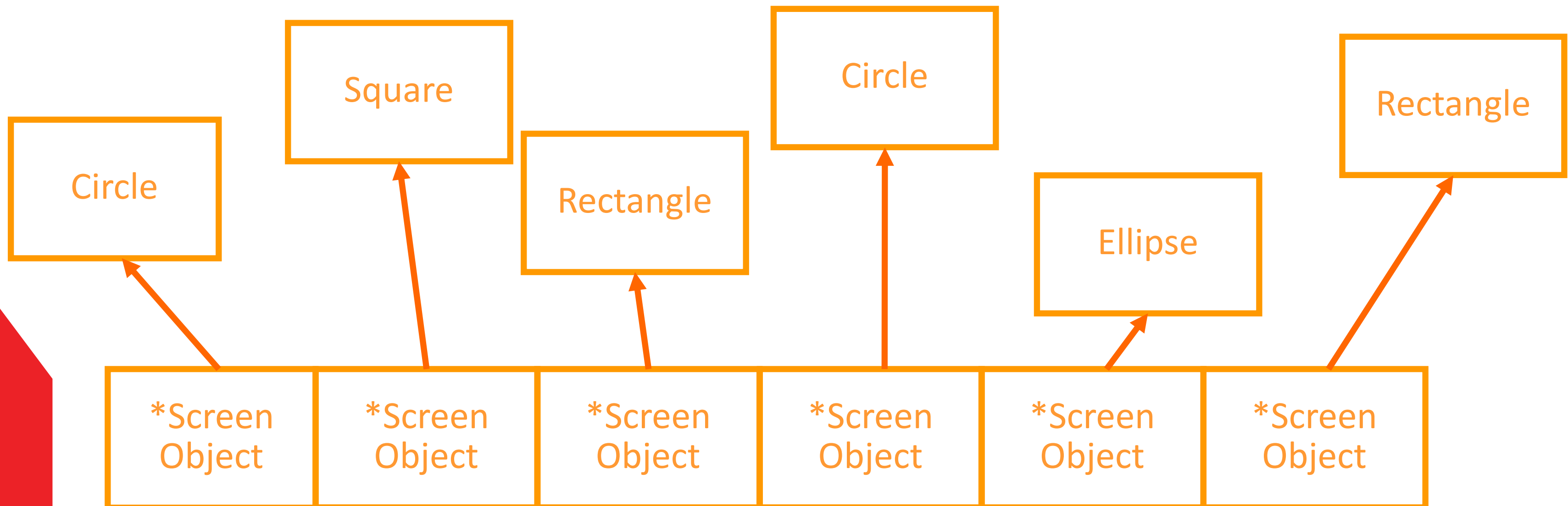
Note that Set and Get methods not shown in this UML diagram.
Protection is not shown either.

Uses of Abstract Classes

- One of the most common reasons for having an Abstract Parent class is so that different things can be grouped together in a single container. Polymorphism is used to distinguish the behaviour.
- For example, in a drawing program you need to have some kind of list of all the objects currently part of the drawing. As all of the objects can be drawn, the draw method is declared virtual in the parent. This is one of the conditions needed for polymorphism to occur in C++.
- And you need to be able to iterate through this list when drawing, saving, printing etc. You will need to access the contained object via the parent pointer (or reference). This is the other condition for polymorphism to occur.
- However strongly typed languages don't usually support an array (or list) of disparate things: [1]

Murdoch University

- However, in C++ what you *are* allowed to do is have an array/list of *pointers* to disparate objects:



- When you iterate through the list, the program calls the *ScreenObjects Draw() methods, which are different for each of the classes as the determination of which Draw() to use gets delayed until run-time. Polymorphism is occurring [1].

Murdoch University

- // Shape.h
- // A base class for drawing shapes.
- // Version
- // 01 - Nicola Ritter
- // 02 – smr – see actual code in the Realisation project for this lecture
- //   includes additional explanation
- //---

- #ifndef SHAPE
- #define SHAPE

- #include <iostream>
- #include <string>
- using namespace std;

- //---

- const float ASPECT_RATIO = 12.0/8.0; // [1]

- //---

Characters in a DOS box are usually 12x8 pixels [1]

```
// Read this together with the actual code in Realisations project
class Shape
{
public:
    Shape() {m_height = 0;}
    virtual ~Shape () {};  // designed for inheritance, so virtual destructor

    virtual void Input (); // virtual needed for polymorphism – see actual code
    virtual void Draw () const = 0;

protected:
    int m_height;
    string m_description;

};

//------------------------------------------------------------

#endif
```

A pure virtual method, therefore this is an abstract class. [1]

Attributes are protected not private, so that derived classes can access them.

Murdoch UNIVERSITY

- #include "Shape.h"

- //---

- void Shape::Input ()// code for illustration only
- // I/O makes the class have reduced usage. [1]
- {
-     cout << "Enter " << m_description << " height: ";
-     cin >> m_height;
- }

- //---

If m_description is given a different value by each derived class, then this output will inform the user about the type of shape

Murdoch UNIVERSITY

```
// Square.h
// Version
// 01 - Nicola Ritter
//--------------------------------------------------------------

#ifndef SQUARE
#define SQUARE

//--------------------------------------------------------------

#include "Shape.h"

//--------------------------------------------------------------

class Square : public Shape
{
public:
        Square() {m_description = "square";}
        virtual ~Square () {};

        virtual void Draw () const;  // was declared pure in Shape, so this is needed

private:
// nothing here
};

#endif
```

Need to include parent header

Derived from Shape

Need to initialise description

Draw() method has to be defined as it was not defined in Shape.

Murdoch UNIVERSITY

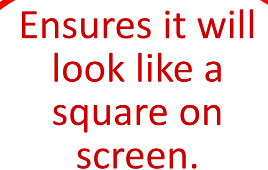
```
// Square.cpp

#include "Square.h"

//-----------------------------------------------------------------

void Square::Draw () const
{
        for (int row = 0; row < m_height; row++)
        {
                for (int col = 0; col < m_height * ASPECT_RATIO; col++)
                {
                   cout << '*';
                }
                cout << endl;
        }
        cout << endl;
}

//-----------------------------------------------------------------
```



Murdoch UNIVERSITY

```cpp
// Triangle.cpp

#include "Triangle.h"

//-------------------------------------------------------------

void Triangle::Draw () const
{
    for (int row = 0; row < m_height; row++)
    {
        for (int col = 0; col < row+1; col++)
        {
          cout << '*';
        }
        cout << endl;
    }
    cout << endl;
}

//-------------------------------------------------------------
```

Triangle.h is almost identical to Square.h

For Triangles we don't care about the aspect ratio

Murdoch UNIVERSITY

```
class Rectangle : public Shape
{
public:
        Rectangle();
        virtual ~Rectangle () {};

        virtual void Draw () const;
        virtual void Input ();


private:
        int m_width;
};

#endif
```

An extra attribute

```
// Rectangle.cpp

#include "Rectangle.h"

//------------------------------------------------------------

Rectangle::Rectangle ()
{
    m_width = 0;
    m_description = "rectangle";
}

//------------------------------------------------------------

void Rectangle::Input ()
{
    Shape::Input ();
    cout << "Enter rectangle width: ";
    cin >> m_width;
}
```

Both the width and the description must be initialised.

First the height is input using the Shape's Input() method, and then the extra information required by Rectangle is requested.

Murdoch UNIVERSITY

```
//-----------------------------------------------------------------

void Rectangle::Draw () const
{
    for (int row = 0; row < m_height; row++)
    {
        for (int col = 0; col < m_width * ASPECT_RATIO; col++)
        {
            cout << '*';
        }
        cout << endl;
    }
    cout << endl;
}

//-----------------------------------------------------------------
```

Murdoch UNIVERSITY

Driver/Main/Test Program

```
// Realisations.cpp
// Version
// 01 - Nicola Ritter, date1
// 02 - Nicola Ritter, date2
//      Refactored into smaller functions that will fit into
//        powerpoint.
// 03 – smr, date 3, polymorphism is highlighted.
//------------------------------------------------------------

#include "Triangle.h"
#include "Square.h"
#include "Rectangle.h"
#include <vector>

using namespace std;
```

- //---

- typedef Shape *ShapePtr;
- typedef vector<ShapePtr> ShapeVec; //vec of ptrs

- //---
- // Subroutine prototypes – **forward declaration**
- void Draw (const ShapeVec &array);
- void Input (ShapeVec &array);
- char Menu ();
- Shape *GetShape (char ch);

Murdoch UNIVERSITY

```
//-----------------------------------------------------------
// READ THIS TOGETHER WITH THE REALISATIONS CODE PROJECT

int main()
{
    ShapeVec array;

    Input (array);
    Draw (array);

    cout << endl;
    return 0;
}

//-----------------------------------------------------------
```

Murdoch UNIVERSITY

- //--------------------**Polymorphism in action**---------------------------------------

```
void Draw (const ShapeVec &array)
{
      int size = array.size();
      for (int index = 0; index < size; inde...
      {
         array[index]->Draw();
      }
      cout << endl;
}

//------------------------------------------------------------
```

The arrow dereference symbol is used when a method is called on a pointer to an object, rather than on the object itself.

The use of the correct Draw() method during the run of the program is a result of *dynamic binding* – ***polymorphism***. *[1]*

Murdoch UNIVERSITY

```
void Input (ShapeVec &array)
{
        char ch = Menu();
        while (ch != 'Q')
        {
                ShapePtr shape = GetShape (ch);
                array.push_back(shape);

                ch = Menu();
        }

        for (int index = 0; index < array.size(); index++)
        {
                array[index]->Input(); //Polymorphic input method
        }
        cout << endl;
}
```

Get a choice from the user and then get a shape based on this choice. Finally add the pointer to the shape to the array

Next get the dimensions of the shapes

Murdoch UNIVERSITY

```
//-----------------------------------------------------------------

char Menu ()
{
    string str;
    do
    {
        cout << "S - Square" << endl;
        cout << "T - Triangle" << endl;
        cout << "R - Rectangle" << endl;
        cout << "Q - Quit entry" << endl;
        cin >> str;
    } while (strchr("STRQstrq",str[0]) == NULL);// what does this do?

    return toupper(str[0]);
}
```

We input a string not a single character so that we do not have to remember to read the <enter> key.

Users are forced to input a correct value.

Murdoch UNIVERSITY

```
//-----------------------------------------------------------

ShapePtr GetShape (char ch) // returns a pointer to parent
{
        ShapePtr shape = NULL;
        switch (ch)
        {
            case 'S':
                shape = new Square;
                break;
            case 'T':
                shape = new Triangle;
                break;
            case 'R':
                shape = new Rectangle;
                break;
        }

        return shape;
}
```

A pointer to a Shape can point to any class derived from Shape. [1]

Note that we are not breaking the rules as we are passing back a *pointer* function-wise, not an object.

Murdoch UNIVERSITY

Interfaces

- Occasionally an abstract class is defined where
  - there are *no* attributes defined;
  - all the methods are pure virtual methods – no body
- This type of class is called an *interface* and is used as just that: it defines the way in which all derived classes will interface with the outside world.
- In UML, they are shown with the word <<interface>> in double arrow braces above the name of the interface:

Murdoch University

Interfaces

- It is also a good idea to name interfaces starting with the letter “I” (IDraw instead of Draw).
- Also the name should be in italics (*IDraw)* along with all other abstract methods.
- There is an alternative way to represent interfaces using a lollipop or circle used in component diagrams as opposed to class diagrams that we are doing. We wouldn’t use lollipop representation in this unit.

Murdoch UNIVERSITY

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Textbook: Chapter on Inheritance and Composition, entire section on Inheritance up to but not including the short section on Composition.
- Chapter on Pointers, Classes, Virtual Functions, Abstract Classes, and Lists, start at section on Inheritance, Pointers and Virtual Functions.

Murdoch UNIVERSITY