Murdoch UNIVERSITY

Data Structures and Abstractions

Abstract Classes

Lecture 16

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 [1]
  - 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**. [1]

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
```

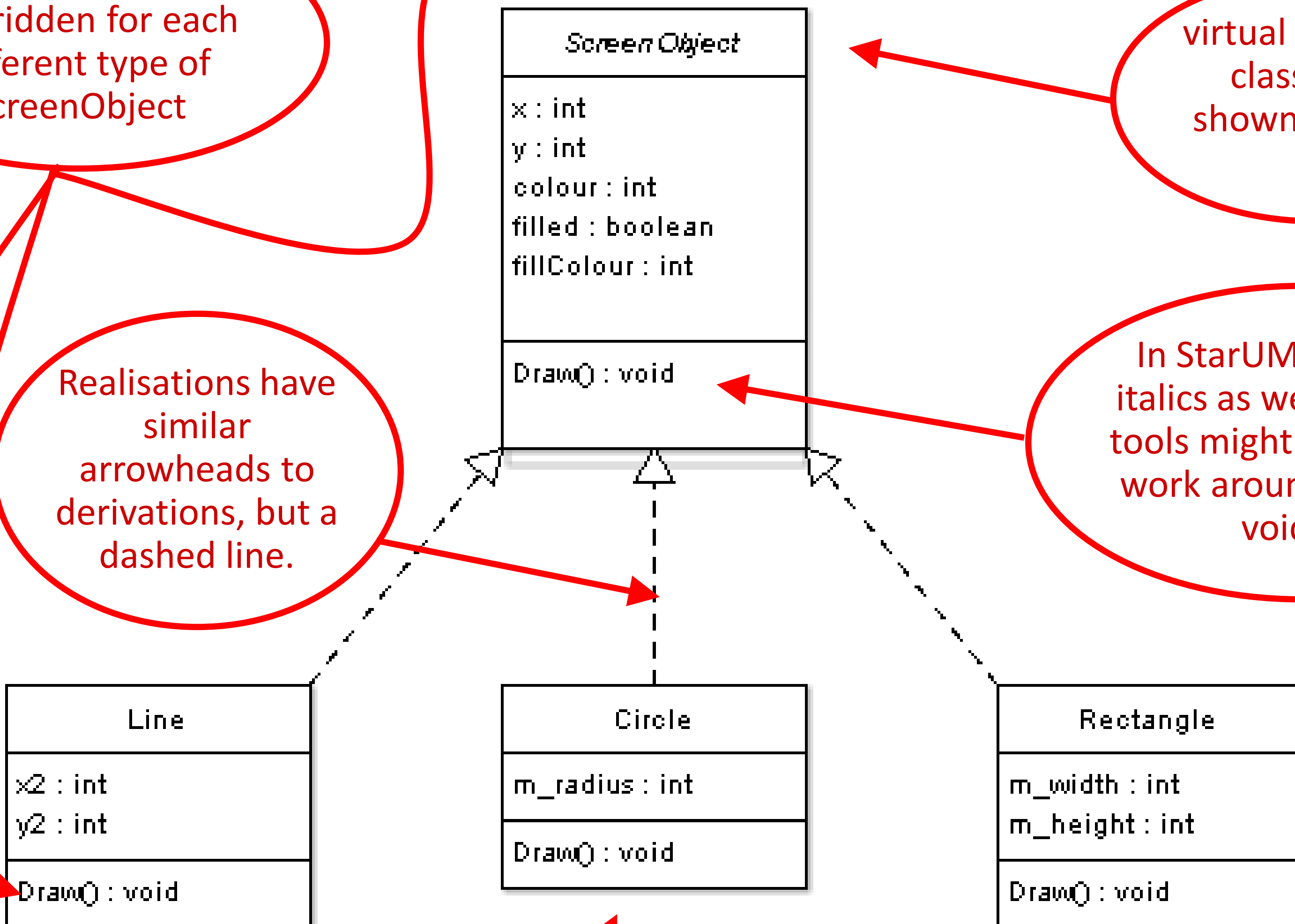
  – Any class that contains pure virtual methods is—by default—an abstract (pure virtual) class: it cannot ever be instantiated as an object because there is missing code body.
- In UML, virtual classes are indicated by using italics for the class name, and the relationship of the derived classes is called 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.

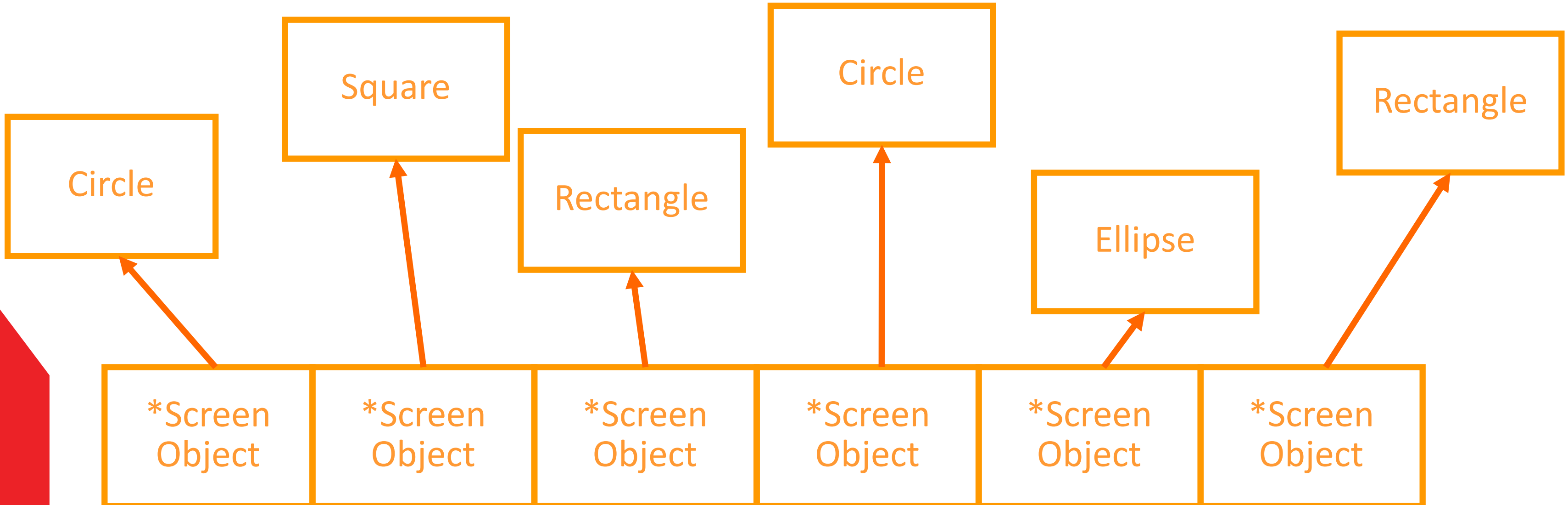
Murdoch UNIVERSITY

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]

Circle	Square	Rectangle	Circle	Ellipse	Rectangle

- 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].

```
// 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
//              IDENTIFY ALL DESIGN ISSUES
//-----------------------------------------------------------------

#ifndef SHAPE
#define SHAPE

#include <iostream> // should this #include be here?
#include <string>
using namespace std; // should this exposure happen

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

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

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

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

```cpp
// 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.

```
#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

```
// 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.

- // 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;
- }
- //---

Ensures it will look like a square on screen.

Murdoch UNIVERSITY

```
// 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

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

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


private:
        int m_width;
};

#endif
```

An extra attribute

Murdoch UNIVERSITY

```
// 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;
}

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

Driver/Main/Test Program

```
// Realisations.cpp
// Version
// 01 - Nicola Ritter first version written
// 02 - Nicola Ritter
//      Refactored into smaller functions that will fit into
//       powerpoint.
// 03 – smr, polymorphism is highlighted.
//-----------------------------------------------------------

#include "Triangle.h"
#include "Square.h"
#include "Rectangle.h"
#include <vector>  // uses the std::vector. Change to use your Vector class.

using namespace std;
```

- //---------------------------MAIN---

- typedef Shape *ShapePtr;
- typedef vector<ShapePtr> ShapeVec; //std::vector of Shape pointers.
- // Change to use your own Vector class
- // typedef Vector<ShapePtr> ShapeVec;

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

```
//-------------------------------------------------------------
//  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; index++)
    {
        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. [1]

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

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

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.

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

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;
}
```

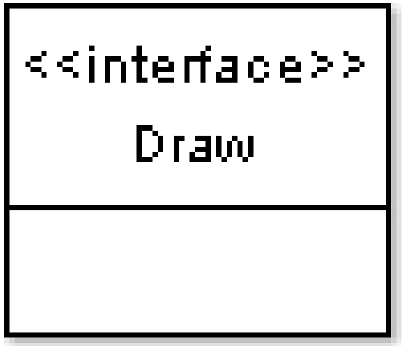
A pointer to a Shape can point to any class derived from Shape. [1]

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

Interfaces

- Occasionally an abstract class is defined where
  - there are *no* attributes defined;
  - all the methods are pure virtual methods – no body

- **Interfaces are abstractions**
  - **If you write your application to abstractions, the implementation of the abstraction can change without breaking your application. [1]**
- 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 other parts of your software.
- In UML, they are shown with the word <<interface>> in double arrow braces above the name of the interface:



Interfaces

- It is also a good idea to name interfaces starting with the letter “I” (***IDraw*** instead of Draw).
- 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

Interfaces, and Strategy design pattern [1]

- The strategy design pattern "favours composition over inheritance".
  - *Inheritance gets abused when it is used to implement code re-use (implementation hierarch instead of type hierarchy)*
  - *Code in this case refers to algorithms implementing behaviours.*
  - *The Strategy design pattern shows how to abstract behaviours into behavioral interfaces.*
  - *Allows program clients to be written to interfaces **not** to implementations. (**reminder: Interfaces are abstractions**)*
  - *You can see the problem created by inheritance for code reuse in his video on Strategy Pattern https://www.youtube.com/playlist?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc* ***[2]***
  - *The video refers to REQUIRED READING chapter 1 in the book Head first design patterns available as ebook from this unit's Myunit readings https://rl.talis.com/3/murdoch/items/7F748952-D8DF-E667-EA75-5603FDB25D83.html?lang=en*

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.
- Chapter 1 of the ebook Head first Design Patterns in they Myunit readings for ict283. Available as ebook *https://rl.talis.com/3/murdoch/items/7F748952-D8DF-E667-EA75-5603FDB25D83.html?lang=en*
    - *Or get the strategy design pattern explained to you https://www.youtube.com/playlist?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc*

Murdoch UNIVERSITY

Data Structures and Abstractions

Encapsulation and Linked Lists

Lecture 17

Records using `struct`

- C++ records (structs): [1]

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

Records using `class`

- Encapsulating a record in a class:

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

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]

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.

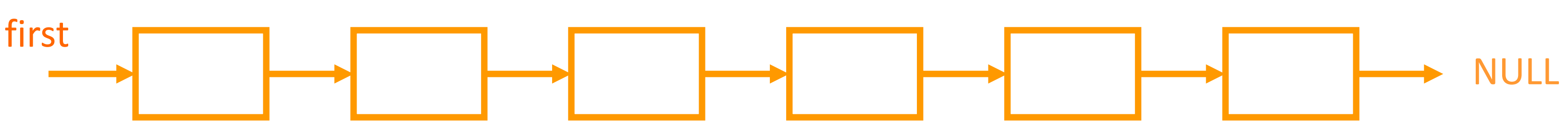
Murdoch UNIVERSITY

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.

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).

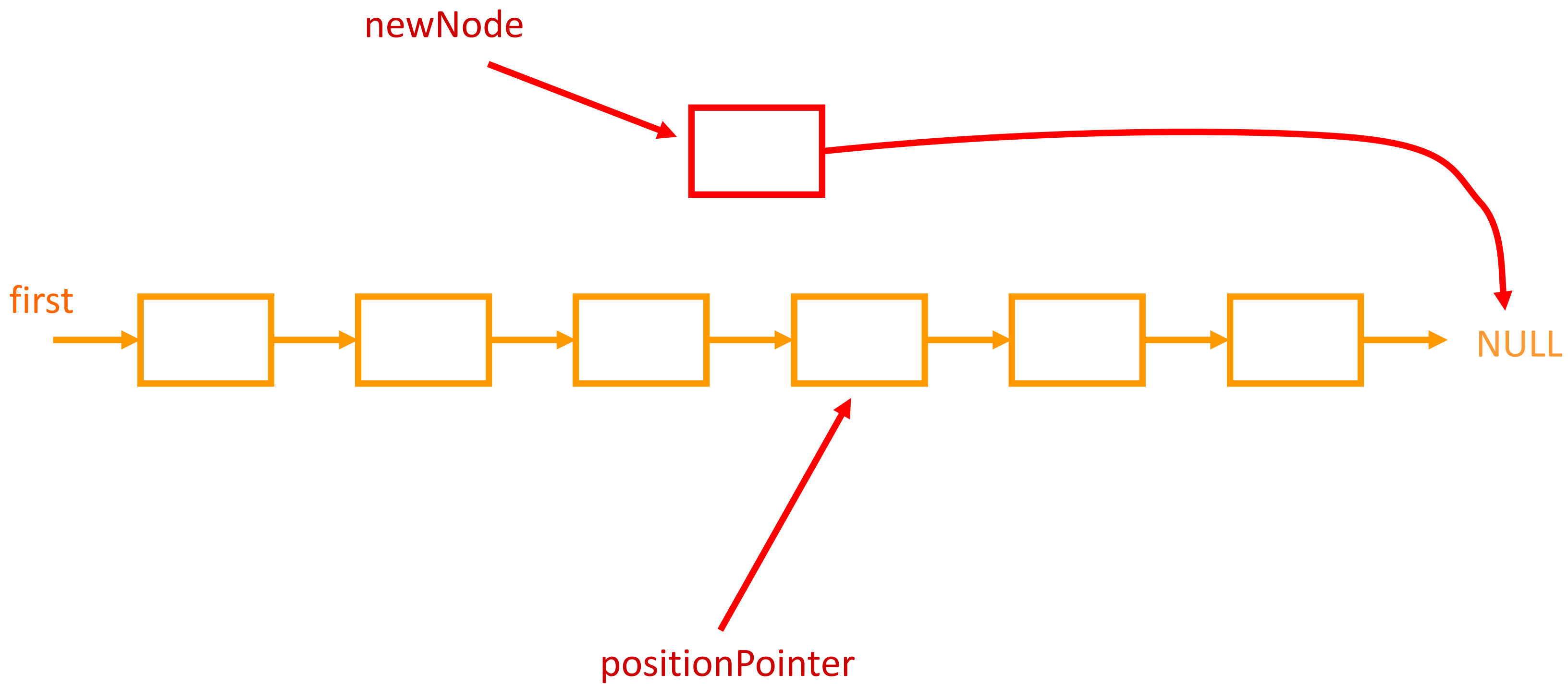
Linked Lists vs Arrays

- 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

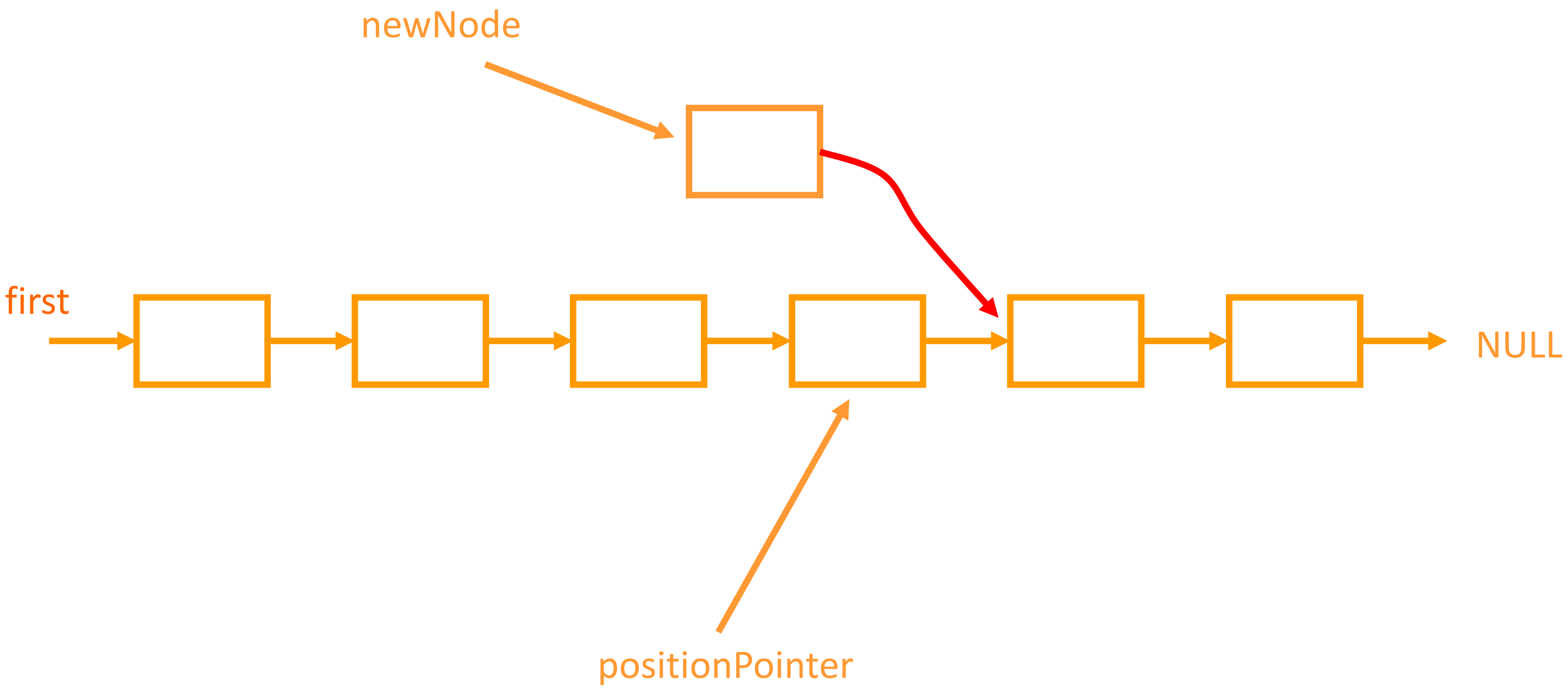
Insertion into a List [1]

- Locate node in front of the insertion point



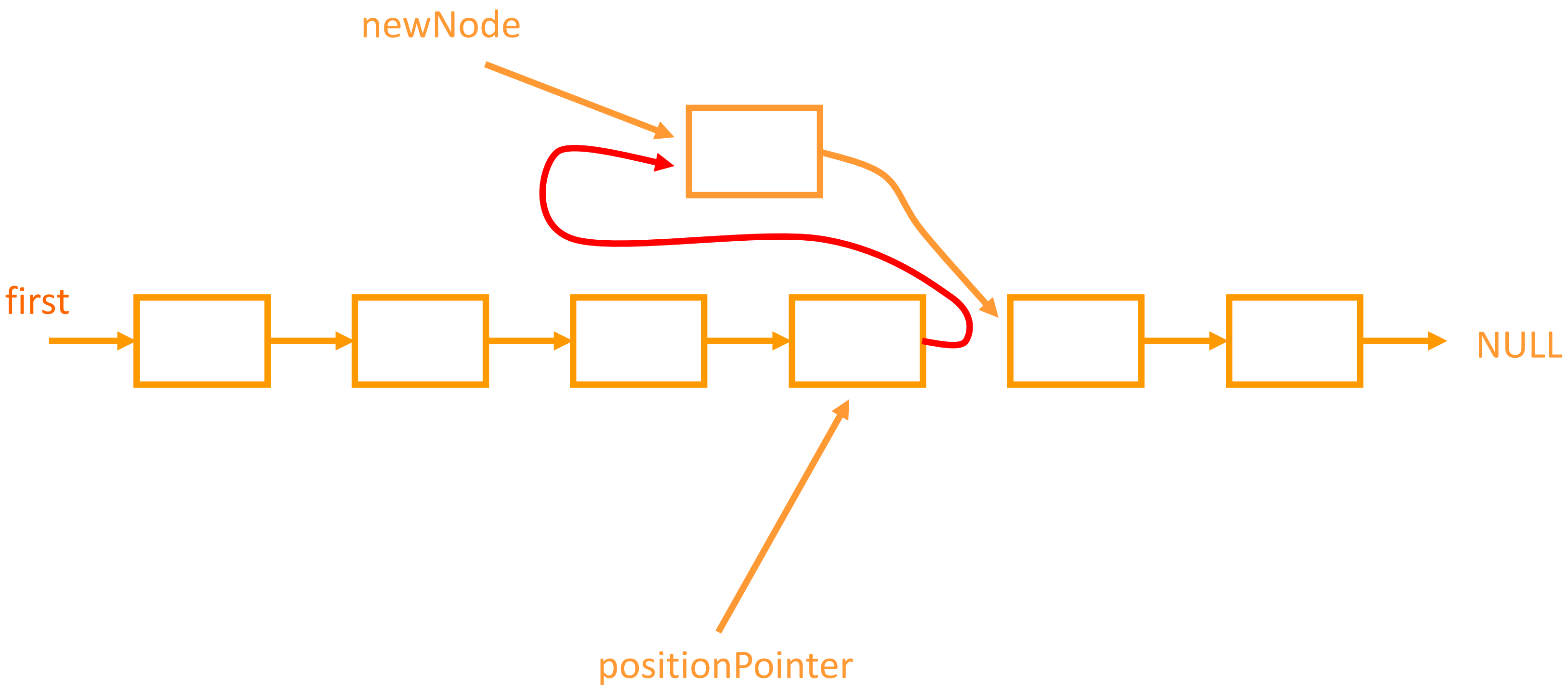
Insertion into a List

- Reassign the ‘next’ pointer of the new node



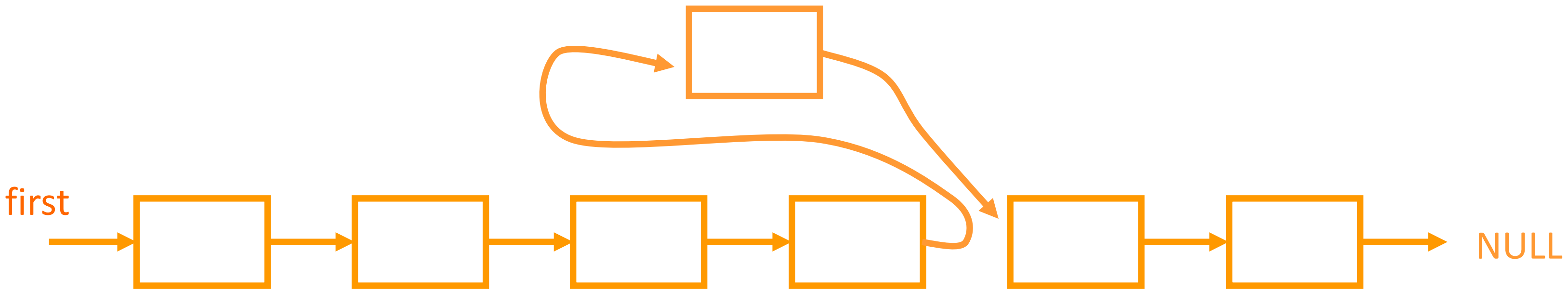
Insertion into a List

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



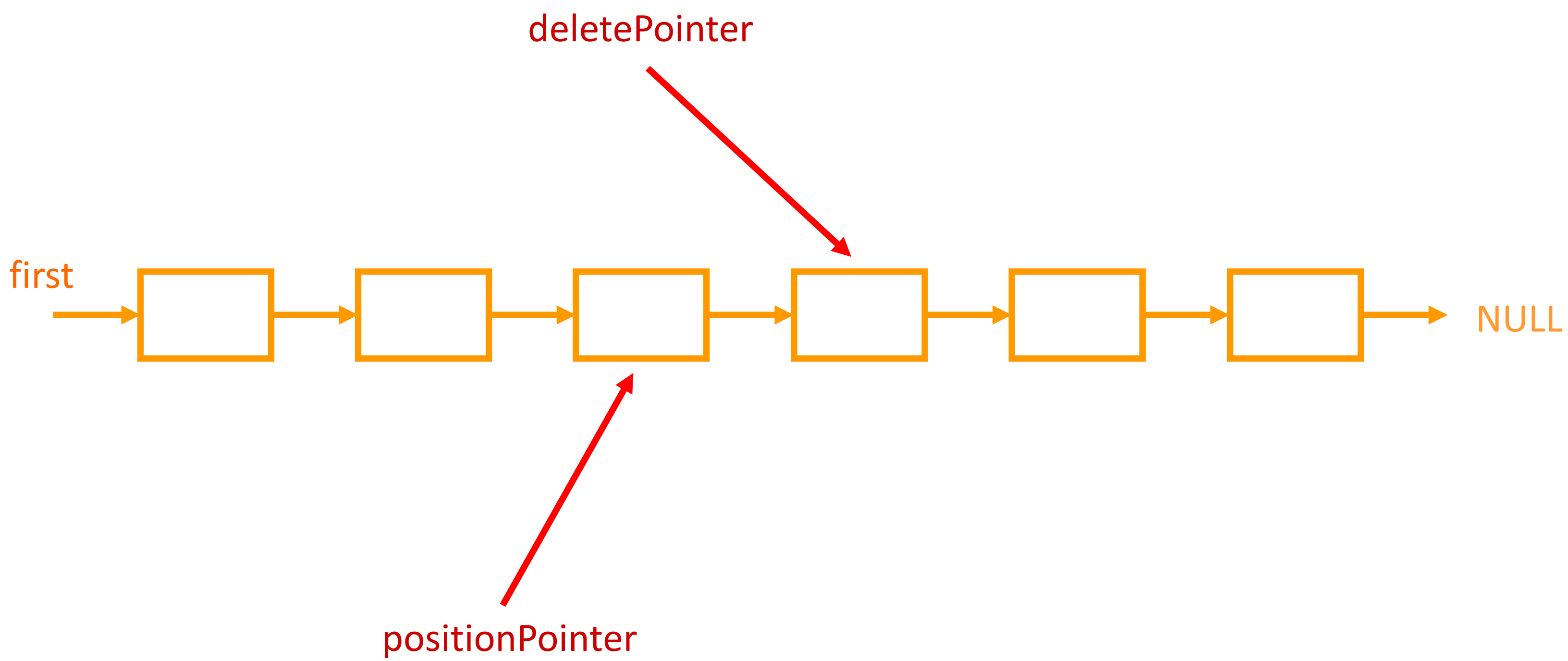
Insertion into a List

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



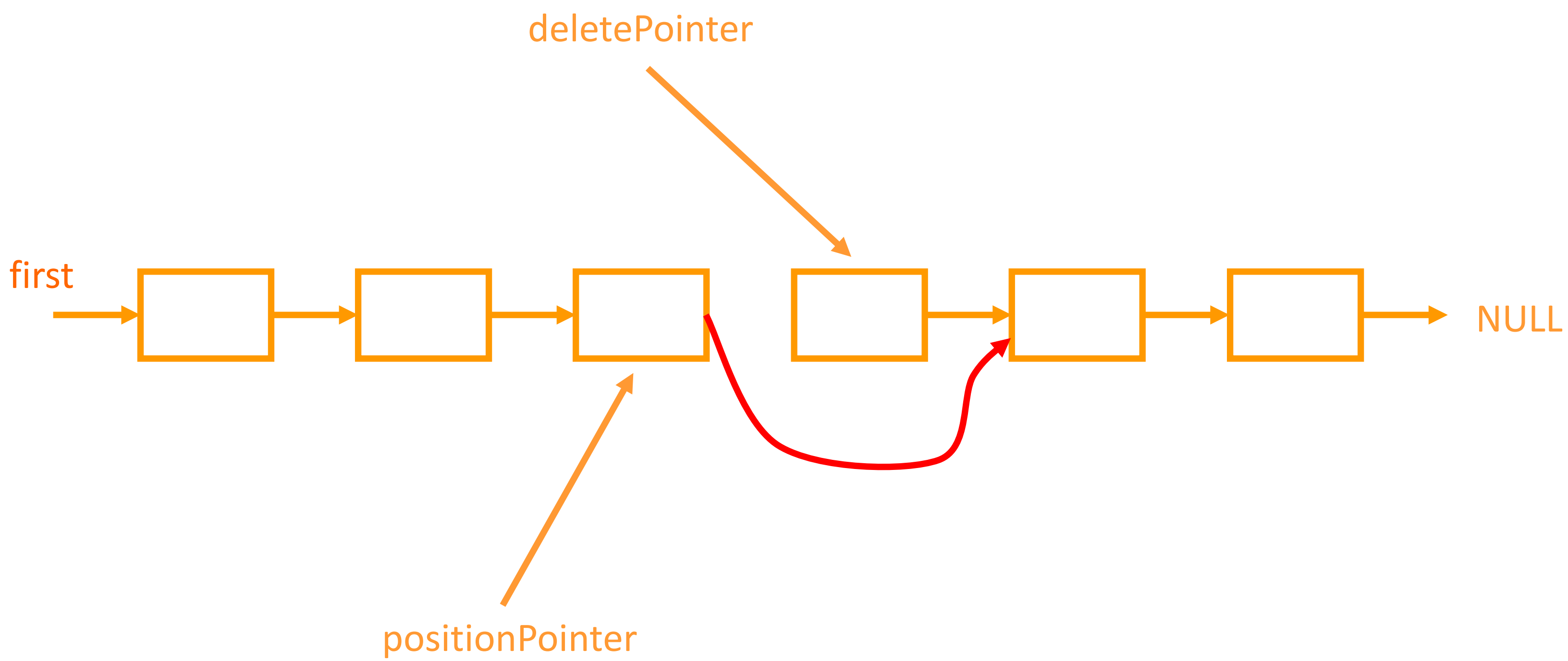
Deletion from a List

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



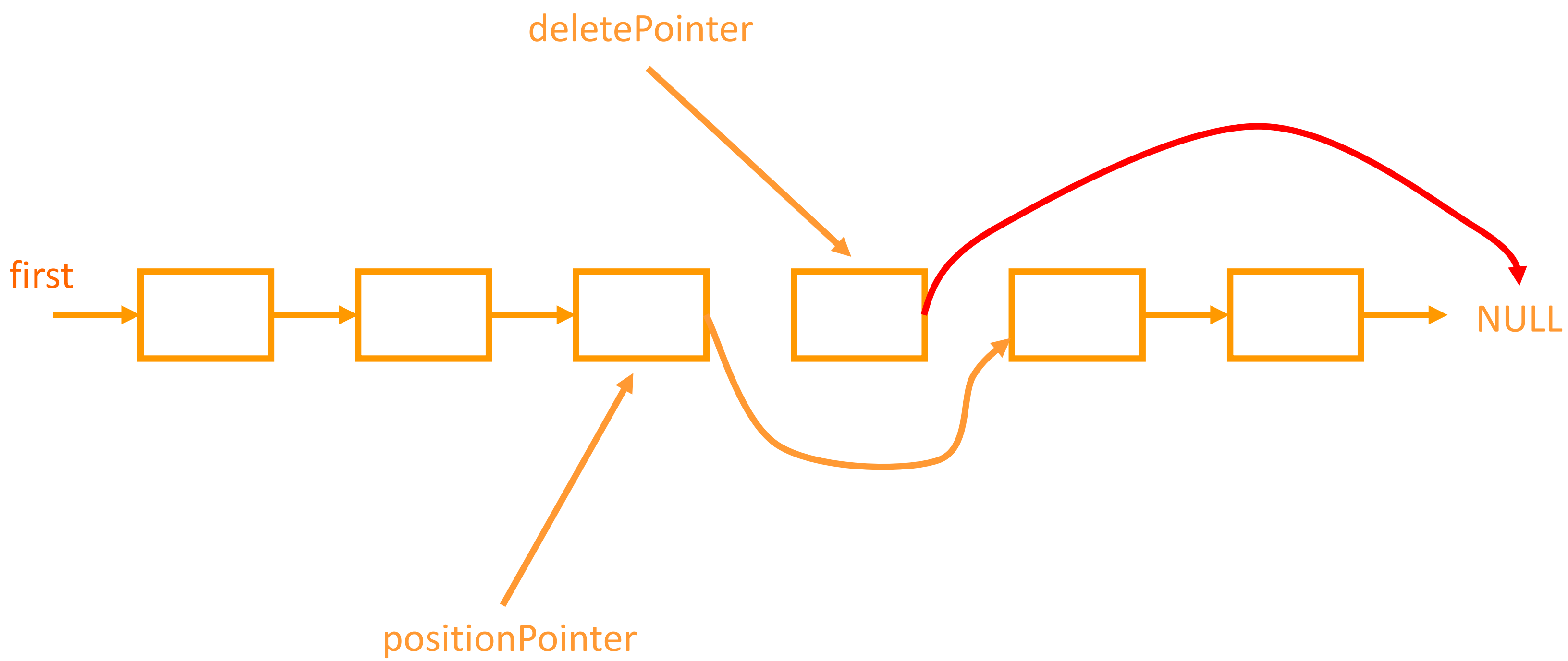
Deletion from a List

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



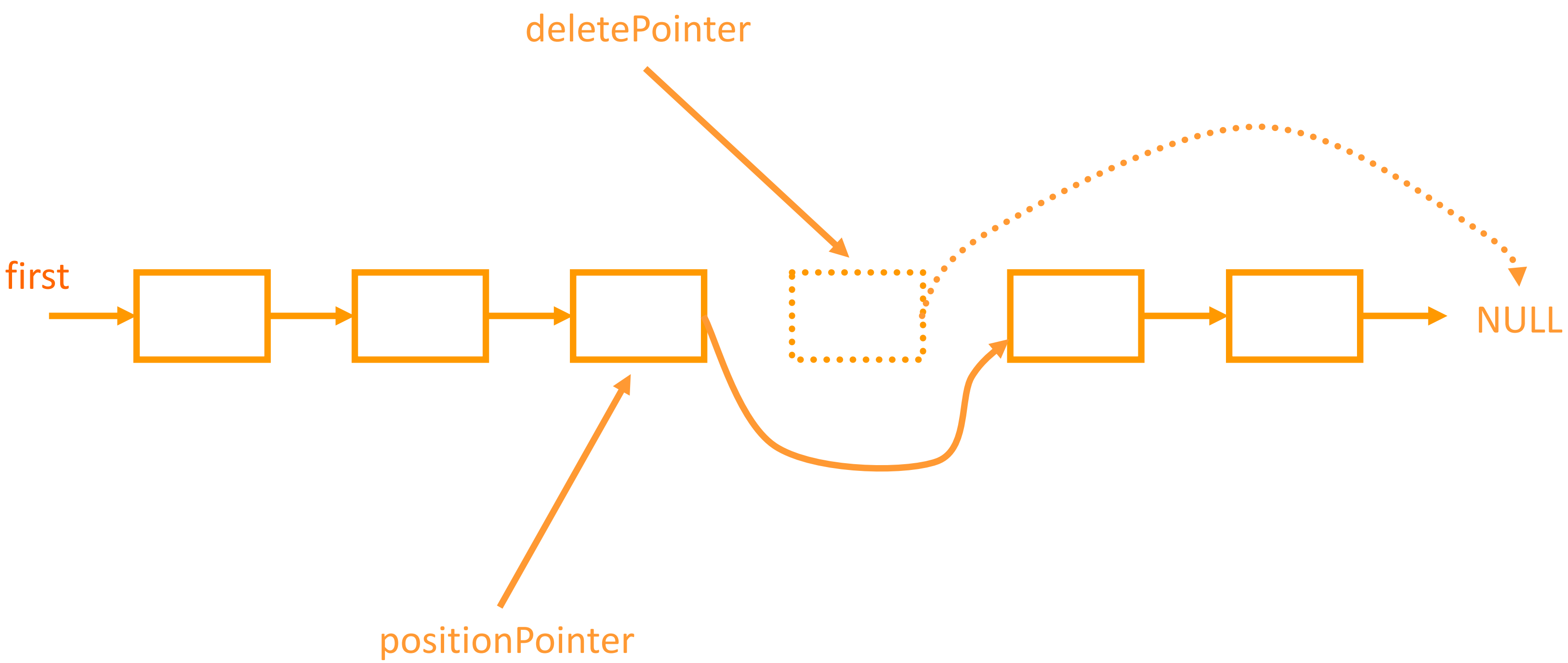
Deletion from a List

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



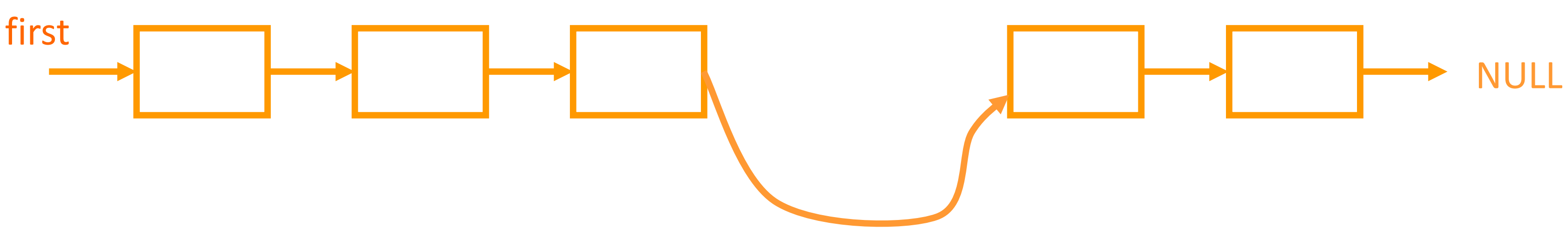
Deletion from a List

- 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.



END

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;
```

STL list Methods

`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

- Let's 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 must 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.

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

Data Structures and Abstractions

Two Dimensional Structures

Lecture 18

Two Dimensions

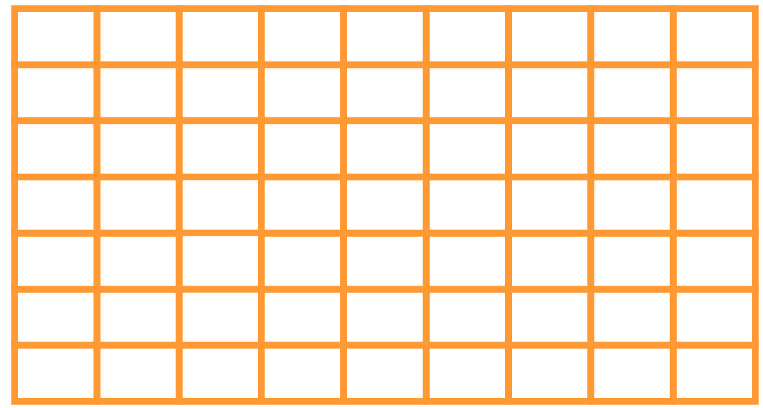
- Two dimensional structures are complicated.
- Therefore they should always be encapsulated.
- This also gives great freedom in how they should be implemented.
- And great freedom to change the implementation if required.
- Some possibilities are:
  - an old-fashioned two dimensional array
  - an array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

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.

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

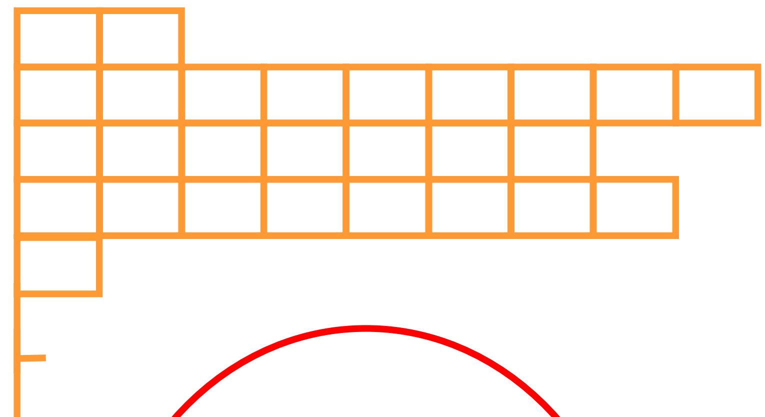
An Array of Vectors

```
// 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

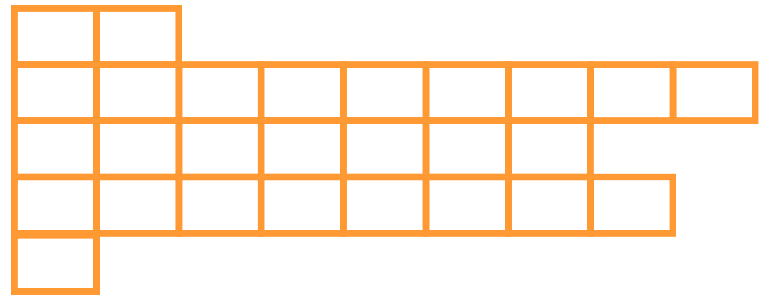
A Vector of Vectors

```
// 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

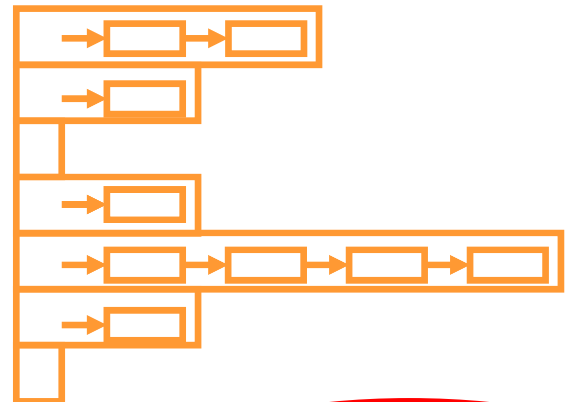
An Array of Lists

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

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

- ...

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



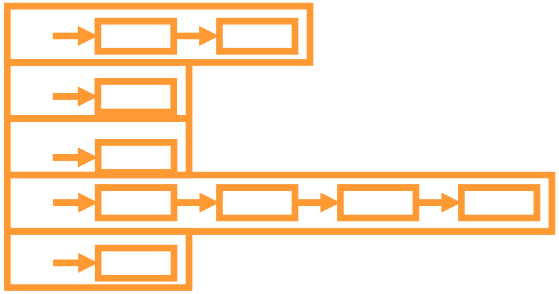
Possible Application
Lists of students in units

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

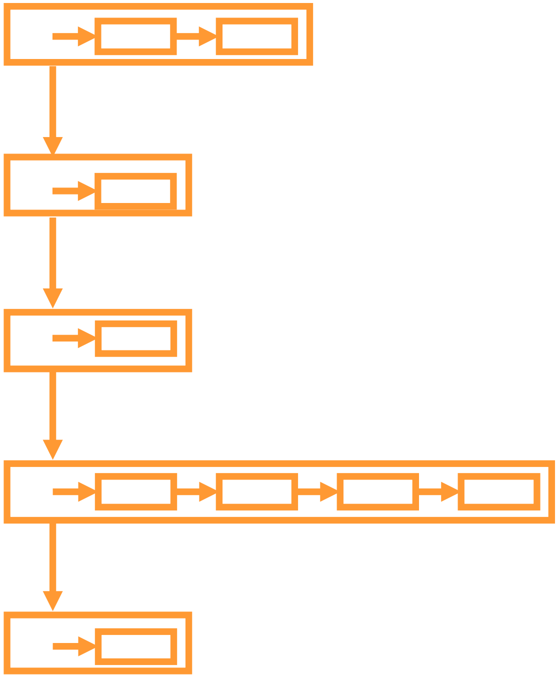
A List of Lists

```
// 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!

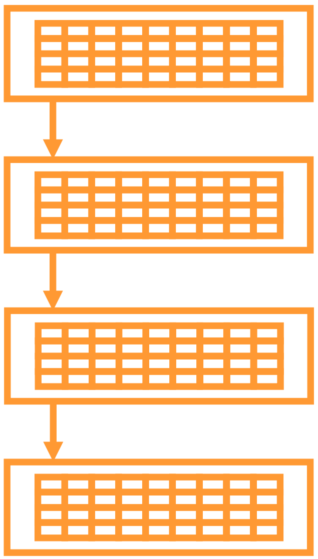
A List of Arrays

- // 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.

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!

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

Murdoch University

Data Structures and Abstractions

Sets

Lecture 19

Sets

- By definition, Sets are unordered collections of data.
  - A = {2, 1}
  - B = {1, 2, 2, 1, 8/4}
  - C = {x: $x^2 - 3x + 2 = 0$}
  - Note that A = B = C i.e., the sets above are equal to each other [1]
- They are used in maths as well as in many other fields.
- But in the computing domain, there are some variations in the way sets are dealt with.
  - Some sets can contain the actual data values, whilst others only keep a record of the presence or absence of data values. STL Bitsets
  - Elements of a set may not be repeated – a common variation [2]. If repeated elements are needed, then a multiset or bag is used. STL multiset
  - A set is explicitly defined to be unordered, but some implementations require ordering for efficiency reasons [3]. The STL set is an associative container and in STL associative containers are ordered.
  - The last two variations break the Set abstraction, but STL designers decided that is fine so sets and multisets are provided. [3]

Sets

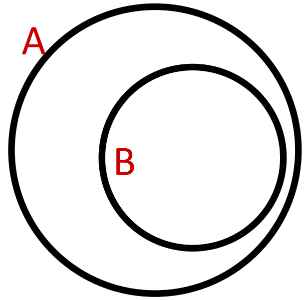
- There are some unique operations for sets:
  - subset
  - union
  - intersection
  - difference
  - element

Subset ⊂

- **Set B is a subset of Set A if all elements in B are also elements of A.**
- **Example 1:**

  if A = {a, b, c, d, g} and B = {c, g}

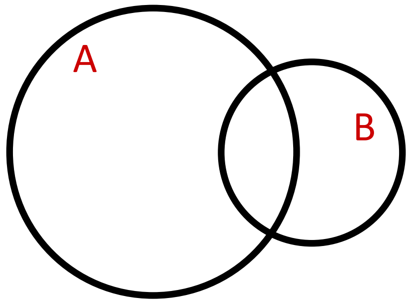
  then B is a subset of A.



- **Example 2:**

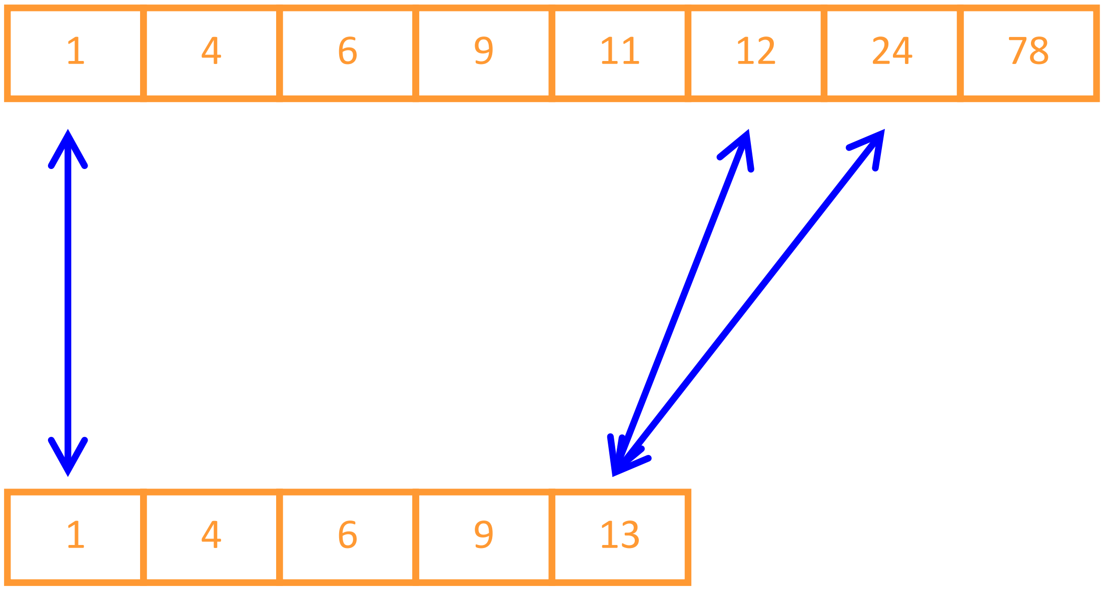
  if A = {a, b, c, d, g} and B = {c, g, u, w}

  then B is not a subset of A.



Subset Animation [1]

subset = false



END

Subset Pseudo-code

```
IsSubsetOf (other)[1]

    Boolean subset = true
    WHILE more elements in this set AND
             more elements in the other set AND
             subset = true
        IF this element = other element
            Get next element from each set
        ELSE IF this element < other element
            subset = false
        ELSE
            Get next element from other set
        ENDIF
    ENDWHILE
    IF more elements in this set
        subset = false
    ENDIF

END IsSubsetOf
```

Murdoch UNIVERSITY

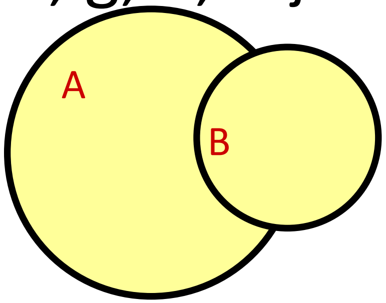
Union (or, ||)

- **The union of Set A and Set B is the collection containing all elements that are in *either* of them, removing double ups. [1]**
- **For example:**

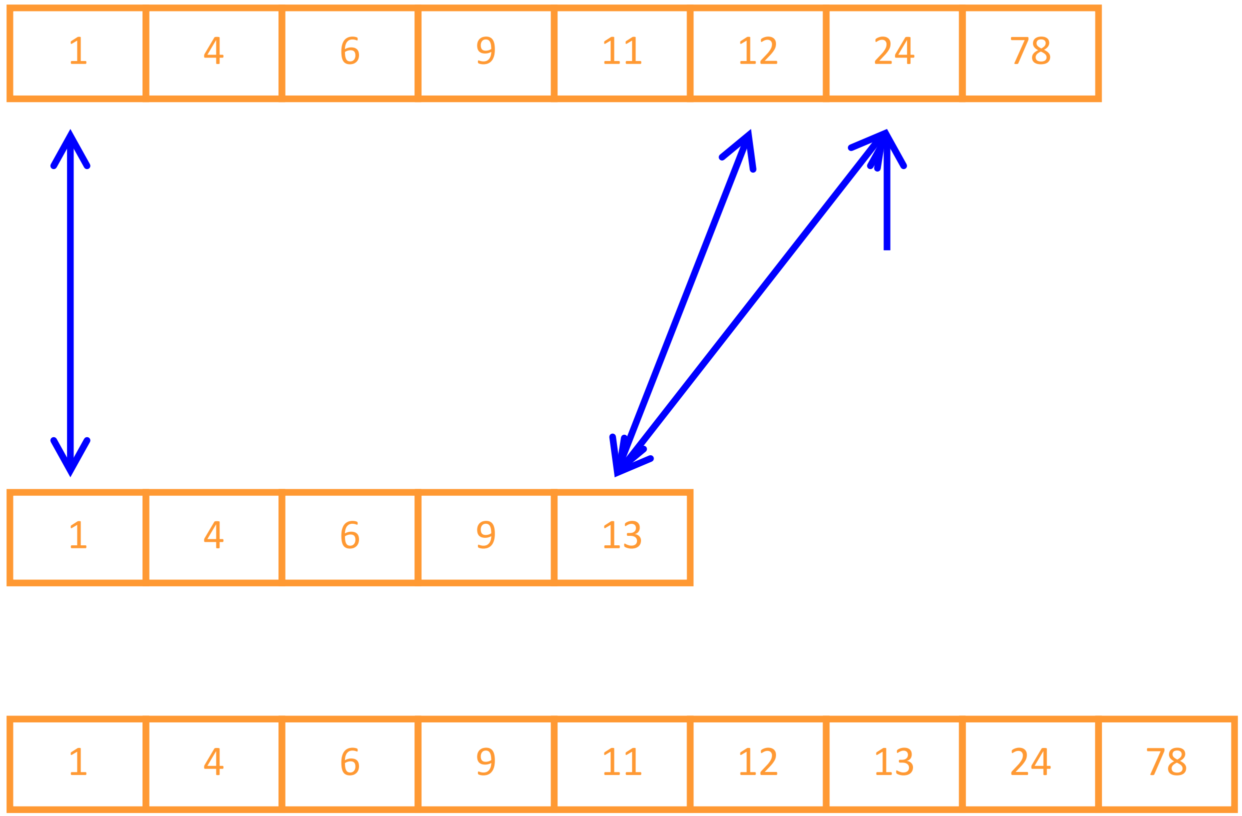
  if A = {a, b, c, d, g} and B = {c, g, u, w}

  then C = A *or* B = {a, b, c, d, g, u, w}

- **C is shown in yellow:**



Union Animation



newSet

Murdoch University

END

Union Pseudo-code

```
Union (other, newSet) [1]

     WHILE more elements in this set AND
              more elements in the other set
          IF this element = other element
               Add this element into newSet
               Get next element from each set
          ELSE IF this element < other element
               Add this element to newSet
               Get next element from this set
          ELSE
               Add other element to newSet
               Get next element from other set
          ENDIF
     ENDWHILE

     WHILE more elements in this set
          Add this element to newSet
          Get next element from this set
     ENDWHILE

     WHILE more elements in other set
          Add other element to newSet
          Get next element from other set
     ENDWHILE

END Union [2]
```

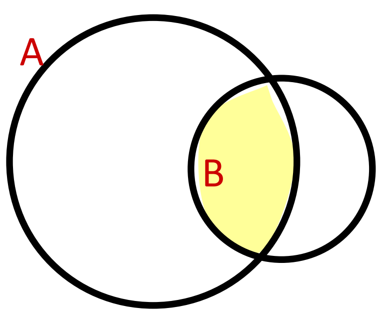
Murdoch UNIVERSITY

Intersection (and, &&)

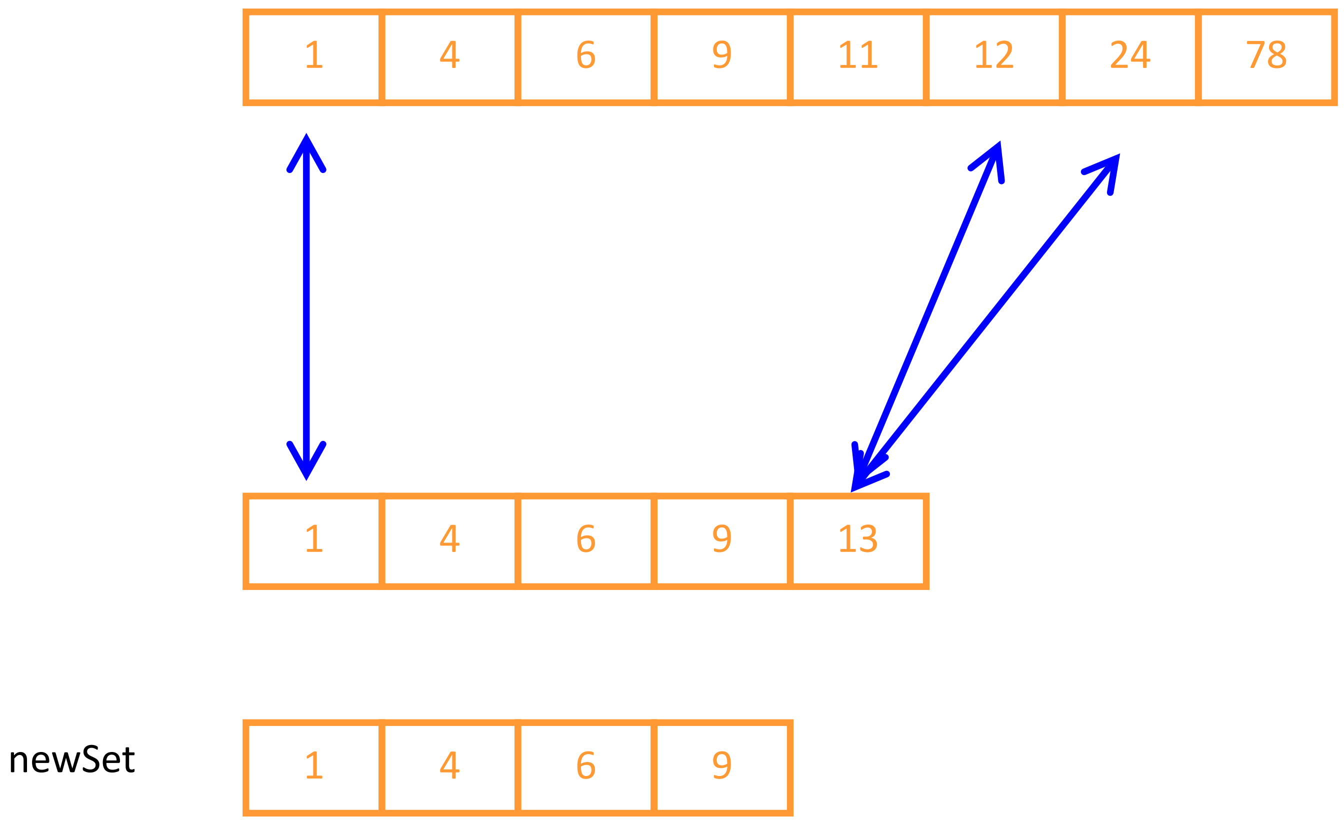
- **The intersection of Set A and Set B is the collection containing all elements that appear in *both* of them.**
- **For example:**

  if A = {a, b, c, d, g} and B = {c, g, u, w}

  then C = A *and* B = {c, g}
- **C is shown in yellow:**



Intersection Animation



END

Intersection Pseudo-code

```
Intersection (other, newSet) [1]

    WHILE more elements in this set AND
               more elements in the other set
        IF this element = other element
            Add this element into newSet
            Get next element from each set
        ELSE IF this element < other element
            Get next element from this set
        ELSE
            Get next element from other set
        ENDIF
    ENDWHILE

END Intersecton
```

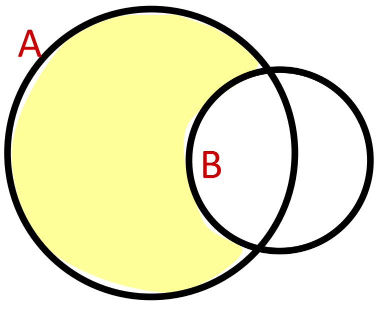
Murdoch UNIVERSITY

Difference (-)

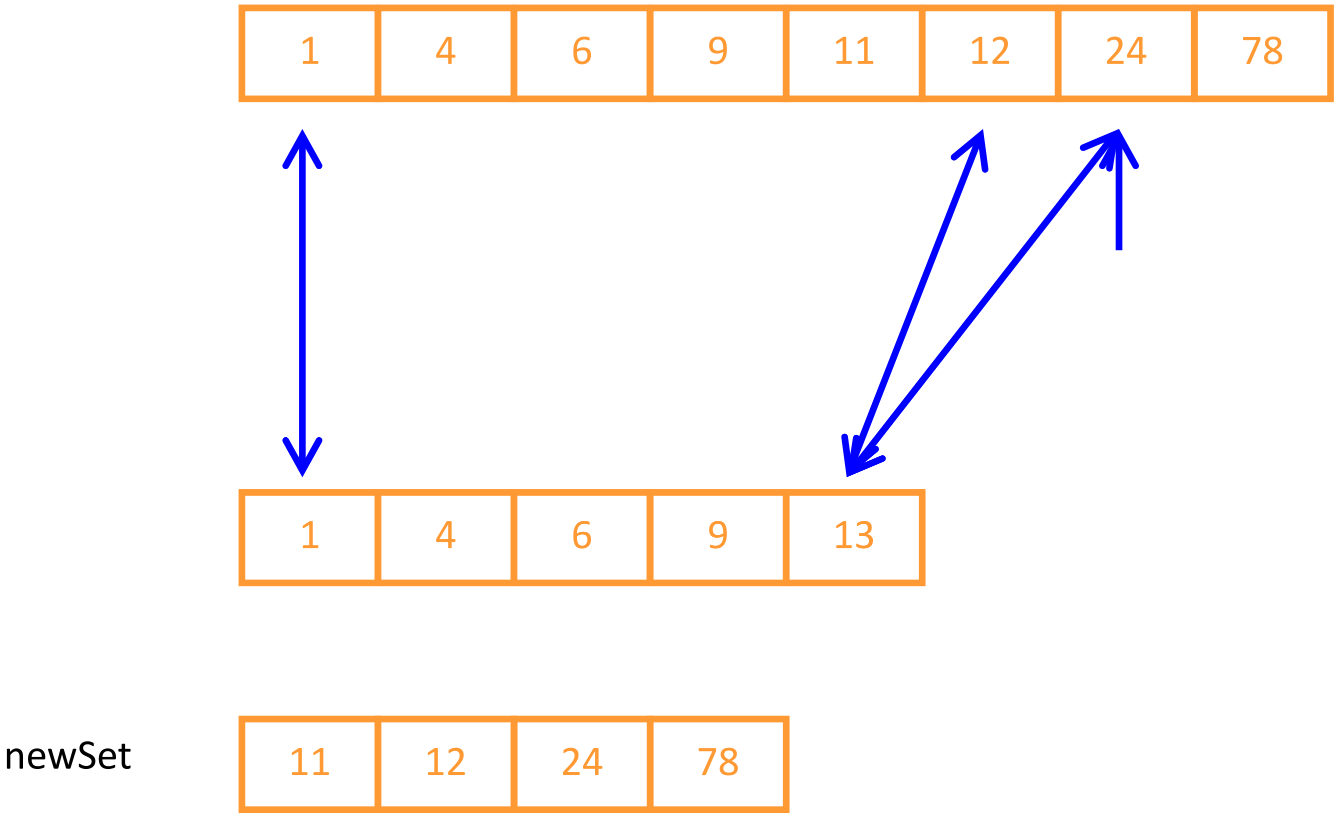
- **The difference of Set B from Set A is the collection containing all elements that are in A but not in B.**
- **For example:**

  if A = {a, b, c, d, g} and B = {c, g, u, w}

  then C = A - B = {a, b, d}
- **C is shown in yellow:**



Difference Animation



END

Difference Pseudo-code

```
Difference(other, newSet) [1]

    WHILE more elements in this set AND
            more elements in the other set
        IF this element = other element
            Get next element from each set
        ELSE IF this element < other element [2]
            Add this element to newSet
            Get next element from this set
        ELSE
            Get next element from other set
        ENDIF
    ENDWHILE
    WHILE more elements in this set
        Add this element to newSet
        Get next element from this container
    ENDWHILE

END Difference
```

The STL Set

- There is an STL set in C++.
- It requires the **`<set>`** header file.
- As for the others it is declared using:

```
typedef set<int> IntSet;
IntSet aset;
```

- The best place to go for information is (again as before):

  http://www.cppreference.com/cppset/index.html

STL Set Methods [1]

`aset.clear()`	Empties the set
`aset.empty()`	Returns true if the set is empty
`aset.begin()`	Returns an iterator to the first element in the set
`aset.end()`	Returns an iterator to past the end of the set
`aset.erase()`	Erase elements from the set
`aset.find()`	Find elements in the set
`aset.insert()`	Insert elements into the set
`aset.size()`	Returns the size of the set
`aset.swap()`	Swaps the contents of two sets

Murdoch UNIVERSITY

Set Algorithms

- For reasons of general utility, routines that could have been placed in the STL set class were placed in the algorithm class:

```
set_difference(set1.begin(), set1.end(),
               set2.begin(), set2.end());
set_intersection(set1.begin(), set1.end(),
                 set2.begin(), set2.end());
set_union(set1.begin(), set1.end(),
          set2.begin(), set2.end());
```

- These should have been *operations* on sets, because it would have been intuitive.
  - Or (preferred) as helper functions available when #include <set> (What is the Open-closed principle?) [1]
- As these routines have general utility, they can applied to other linear data structures like vectors. This approach can be argued to be good, as re-use of code is happening.
- And there is no subset but a subset set helper function can be written using algorithm's `includes` or `set::find()` function.

```
// Do the set difference using the insert iterator
set_difference(set1.begin(), set1.end(),
               set2.begin(), set2.end(), resultItr);
```

- An abstract representation could have operator-(..) defined, so that:

```
resultSet = set1 - set2
```

- For this reason—unless the task is trivial—the STL set needs to be encapsulated or a helper operator/function is provided.
  - Prefer the helper operator/function as this means the least amount to code for a given functionality.
  - The helper operator or function uses only the set's public interface.

Readings

- Textbook: Standard Template Library, section on Associative containers relating to set and multiset.
- Library EReserve: Preiss, Data structures and algorithms with object-oriented design, Chapter 12
- http://www.cplusplus.com/reference/stl/
- http://en.cppreference.com/w/cpp/container/set
- http://en.cppreference.com/w/Main_Page

Data Structures and Abstractions

MAPS

Lecture 20

Note 1 (legacy code only)

- When you compile some STL code in VC++ you might get a warning: [1]

```
ICT283\Code\Sets\SetDifference.cpp(59) : warning C4786:
'std::pair<std::_Tree<int,int,std::set<int,std::less<int>,std::allocat
or<int>>::_Kfn,std::less<int>,std::allocator<int>
>::const_iterator,std::_Tree<int,int,std::set<int,std::less<int>,std::
allocator<int>>::_Kfn,std::less<int>,std::allocator<int>
>::const_iterator>' : identifier was truncated to '255' characters in
the debug information
```

- This is the *only* warning you can ignore completely (a debug identifier)
- If it really annoys you, then add the following code before the includes in the file that is generating the warning:

```
#pragma warning (disable : 4786)
```

- Do *not* disable any other warning!!
  - DO NOT JUST DISABLE THE WARNING IF YOU ARE NOT GETTING THE WARNING.
  - Warning is only on older implementations of Visual C++, so you are not likely to see it now. If you do see it, please let me know. As we
  - **Legacy code and compilers may generate this issue, so just for noting**
  - **You wouldn't see this error in the work you are doing in this unit but be aware of issues like this with legacy code and older compilers.**

Murdoch UNIVERSITY

Note 2 (relevant now)

- If you get an error message such as:

```
ICT283\Code\Map\Map.cpp(57) : error C2440: 'initializing' : cannot convert from
'class std::_Tree<class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> >,struct std::pair<class
std::basic .....
    No constructor could take the source type, or constructor overload resolution
was ambiguous.
```

- Then it almost always means that you are passing an object as a const reference to a function that uses iterators. Iterators expect references not const references. So, for example, the code below would probably generate this error: [1]

```
void DoSomething (const IntSet &aset) //IntSet is some type with an iterator
                              // typically, this type has had a typedef
                              // typedef set<int> IntSet;
{
      IntSet::iterator itr = aset.begin(); // itr can be used to modify - error
}
```

- **To solve it**, use a const_iterator, instead of an iterator: [2]

```
void DoSomething (const IntSet &aset)
{


    IntSet::const_iterator itr = aset.begin();
}
```

Murdoch UNIVERSITY

Maps

- An association (pairing) is a connection between two things, for example the word "sanity" (*key*) is associated with the definition (*value*) "the state of having a normal healthy mind"*
- A dictionary or *map* is then a collection of key-value associations [1].
- The first part of the pair is often called a *key.*
- The data in maps is inserted, deleted and found using the key. So key needs to be unique but value need not be. [2]
- For example, if one had a map that *was* an English dictionary, then we would expect to be able to retrieve the definition of sanity using something like:

```
dictionary.GetDefinition ("sanity");
```

or even

```
dictionary["sanity"];
```

* *Australian Dictionary,* Collins, 2005

The STL Map

- The STL map is a very nice template indeed.
- The declaration requires two data types, the first being the key and the second being the data to be stored in association with the key. [1]
- For example, consider a class taking a vote on who should be the class president. We want to associate names with an integer number of votes:

```
#include <map>
...
map<string, int> Popularity;
...
Popularity pop;
```

A Simple Map Program

- // Normal comments up here
- #include <map>
- #include <iostream>
- #include <iomanip>
- #include <string>
- using namespace std; **// don’t do this – use the approach in the code that is provided separately.**
- //---
- const string END = “end”; // string object
- //---
- typedef map<string,int> Popularity;
- typedef Popularity::**iterator** PopItr;
- typedef Popularity::**const_iterator** PopCItr; // see textbook chapter on STL

It can be really useful to define an iterator for each STL type you use

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

void AddData (Popularity &pop);
void Output  (const Popularity &pop);

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

int main ()
{
     Popularity pop;

     AddData (pop);
     Output (pop);

     cout << endl;
     return 0;
}

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

```
void AddData (Popularity &pop)
{
        string name;

        // Prime the while loop
        cout << "Enter vote name, or “ << END << “ to finish: ";
        getline (cin, name);

        while (name != "end") // is this comparison efficient? [1]
        {
                // If they are part of the map already, this adds 1
                //   to their score.  If they are not, it puts them
                //   in the map and gives them a score of 1.
                // see missing code in the notes section [2]

                cout << "Enter vote name, or 'end' to finish: ";
                getline (cin, name);
        }
}
```

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

void Output  (const Popularity &pop)
{
        PopCItr winner = pop.begin(); // set a temp winner as the first item

        // For each entry in the map
        for (PopCItr itr = pop.begin(); itr != pop.end(); itr++)
        {
                // Output the first and second parts of the pair (association)
                cout << setw(20) << itr->first << " : " << itr->second << endl;

                // Now check if this person should be the winner
                if (winner->second < itr->second) // compare the value [1]
                {
                    winner = itr;
                }
        }

        // Output the winner
        cout << endl << "The new class president is " << winner->first
                << " with " << winner->second << " votes" << endl; [2]
}

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

Readings

- Textbook: Chapter on Standard Template Library.
- Map: https://en.cppreference.com/w/cpp/container/map
- Multimap: https://en.cppreference.com/w/cpp/container/multimap

Murdoch University

Data Structures and Abstractions

Stacks

Lecture 21

Temporary Storage

- When processing it is often necessary to put data into temporary storage.
- This can happen, for example, when:
  - processing events in an event-driven OS;
  - processing email in and out of a server;
  - scheduling jobs on a main-frame;
  - doing calculations;
  - sorting or merging;
- The most common data structures for temporary storage are stacks, queues, heaps and priority queues.

Murdoch UNIVERSITY

Stacks

- Stacks are ADS that emulate, for example, a stack of books: you can only put things on or take them off at the top.
- There are only two operations allowed on a stack: [1]
  - **`Push (something on to it)`**
  - **`Pop (something off it)`**
- Plus two query methods:
  - **`Empty ()`**
  - **`Full () // optional`**
- Since the last thing on is the first thing off, they are known as LIFO (Last In, First Out) data structures, or sometimes FILO (First In, Last Out).
- In essence, a stack reverses the order of the data.

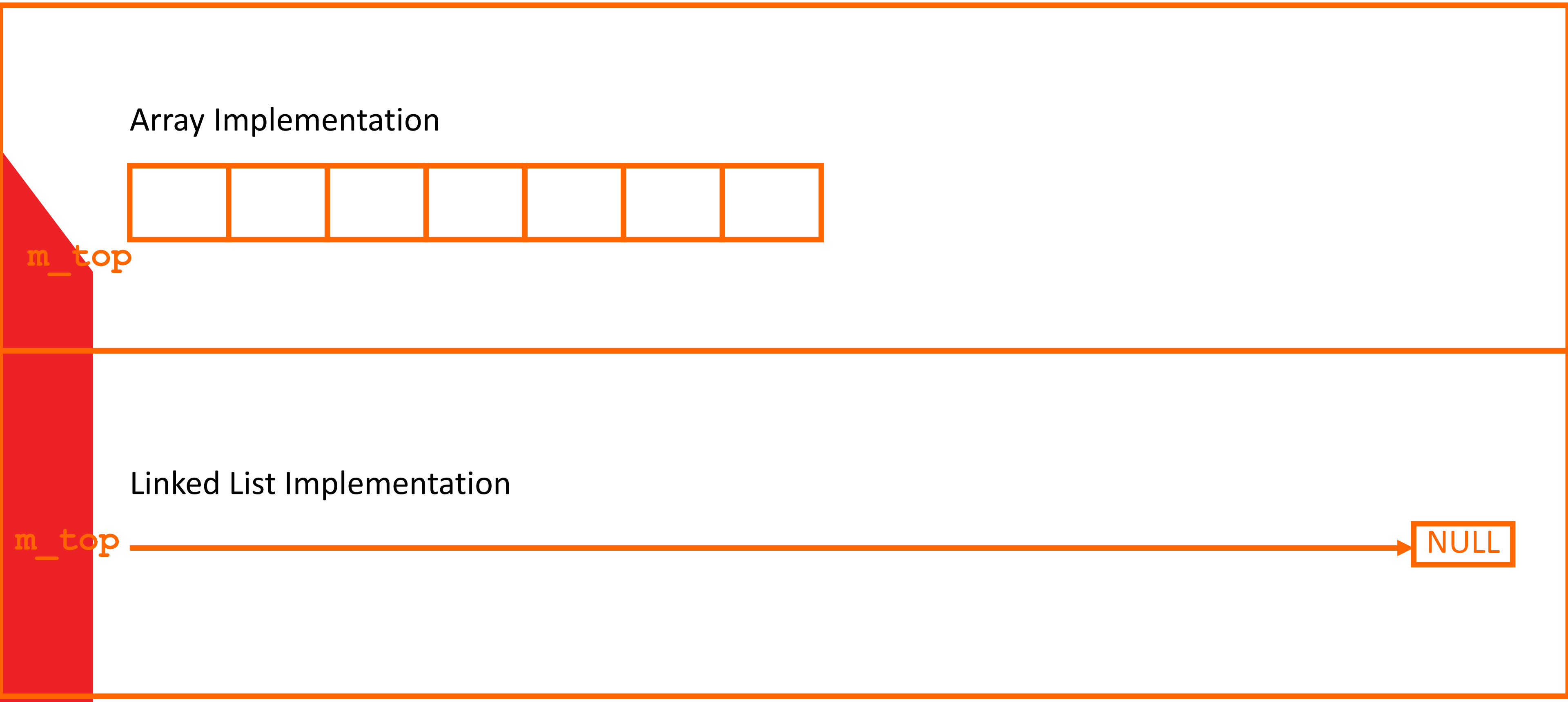
Stack Implementation

- Stacks can be implemented any way you want, the encapsulation of the container used ensures that it does not matter.
- As long as it only has **`Push, Pop, Empty`** and (optionally) **`Full,`** then it is a stack.
- Most commonly they are implemented using arrays, lists or an STL structure.
- If none of these exactly fit the required abstraction that we are after, they should be encapsulated inside our own Stack. [1]

Error Conditions for Stacks

- If you try to `Push()` onto a stack that has no free memory, then you get overflow.
- If you try to `Pop()` from an empty stack then you have underflow.
- So `Push()` and `Pop()` return a boolean to indicate if one of these errors has occurred.

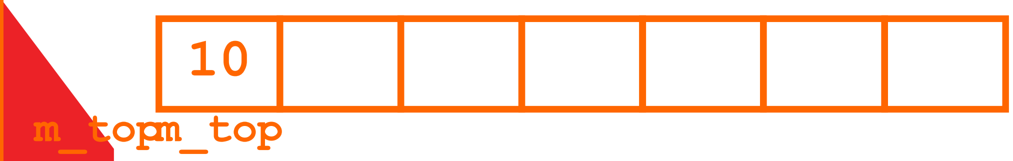
Stack Example (Animation)



Stack Example (Animation)

`Push(10)`

Array Implementation

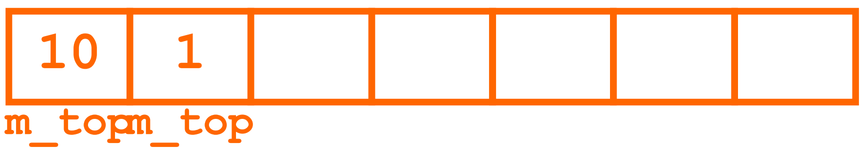


Linked List Implementation

Stack Example (Animation)

`Push(1)`

Array Implementation

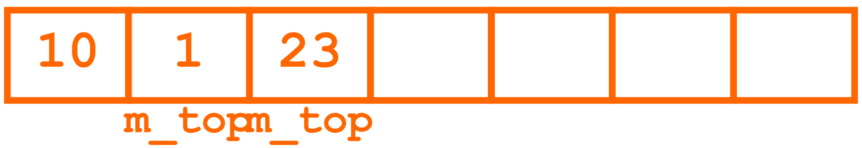


Linked List Implementation

Stack Example (Animation)

Push(23)

Array Implementation



Linked List Implementation

Stack Example (Animation)

`Pop (num)` **`num`** 23

Array Implementation

10	1	23				

m_top m_top

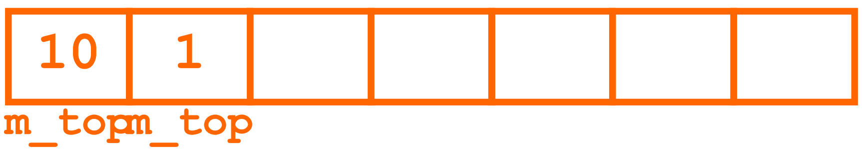
Linked List Implementation

m_top → 23 → 1 → 10 → NULL

Stack Example (Animation)

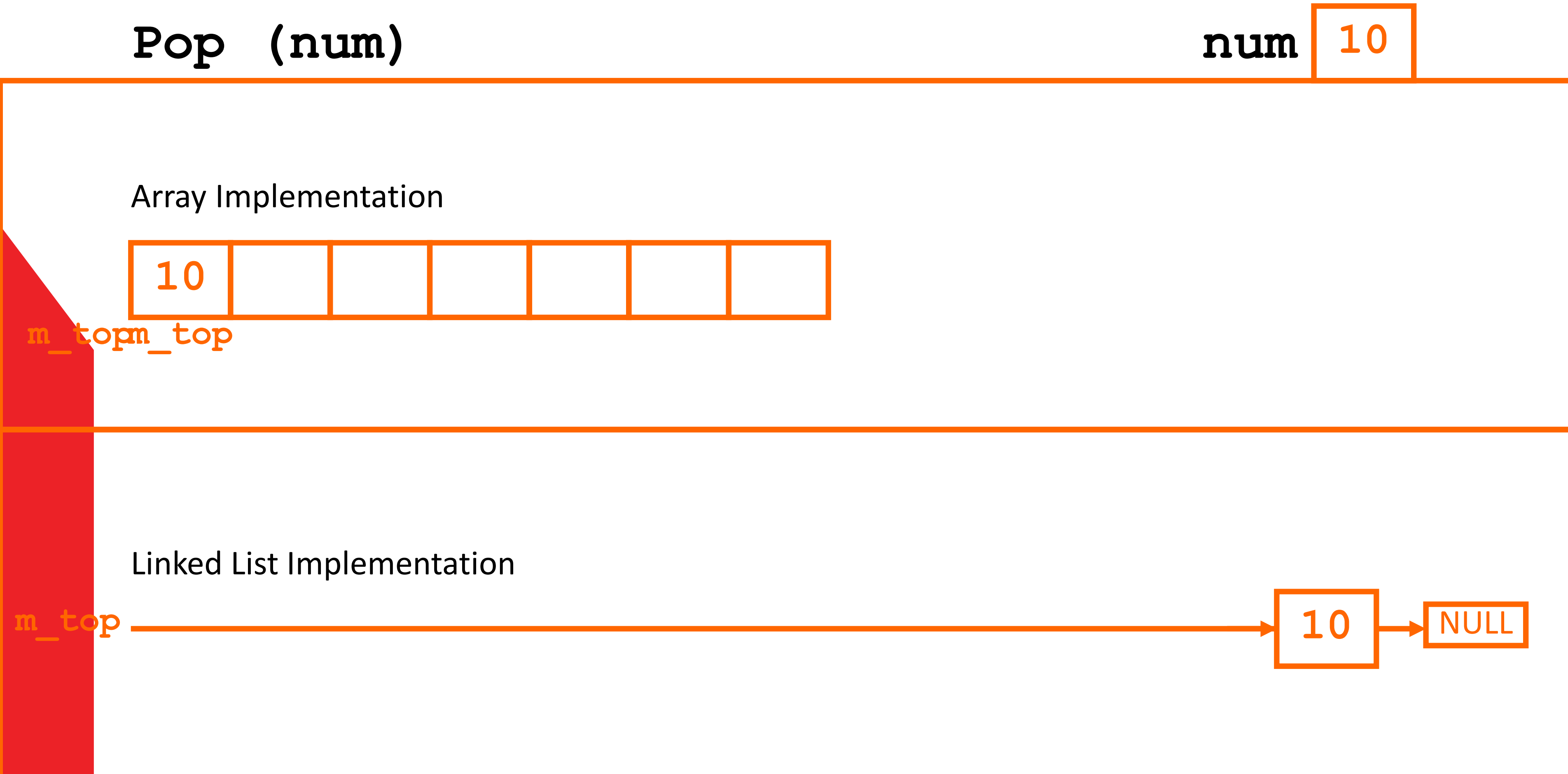
`Pop (num)` `num` 1

Array Implementation

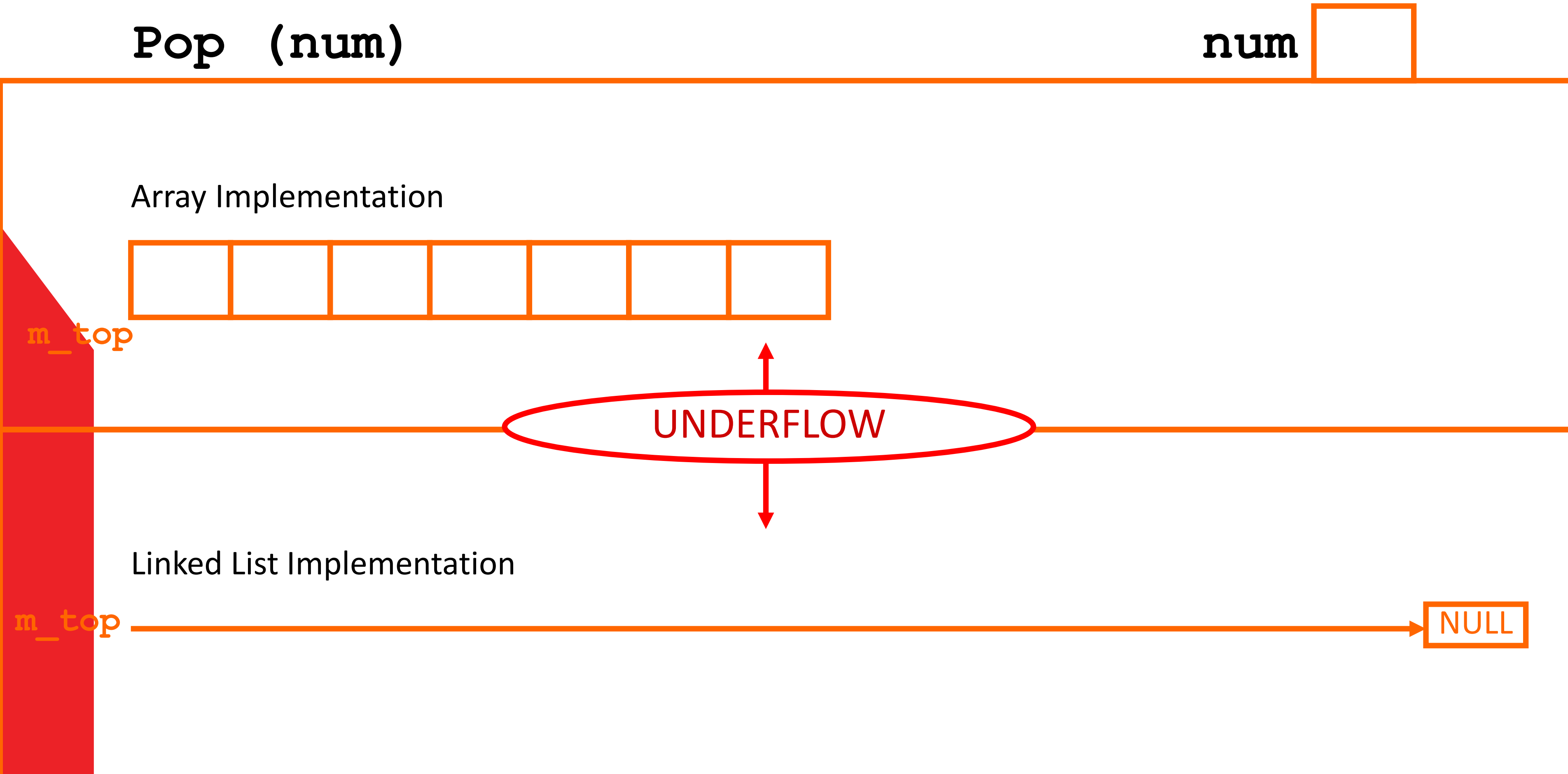


Linked List Implementation

Stack Example (Animation)



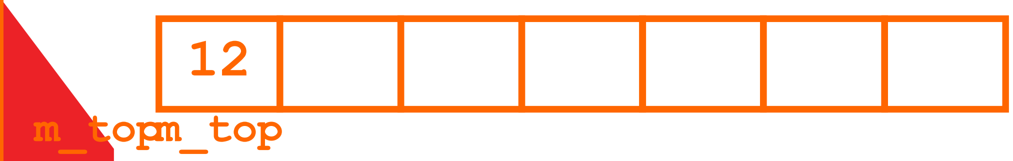
Stack Example (Animation)



Stack Example (Animation)

`Push(12)`

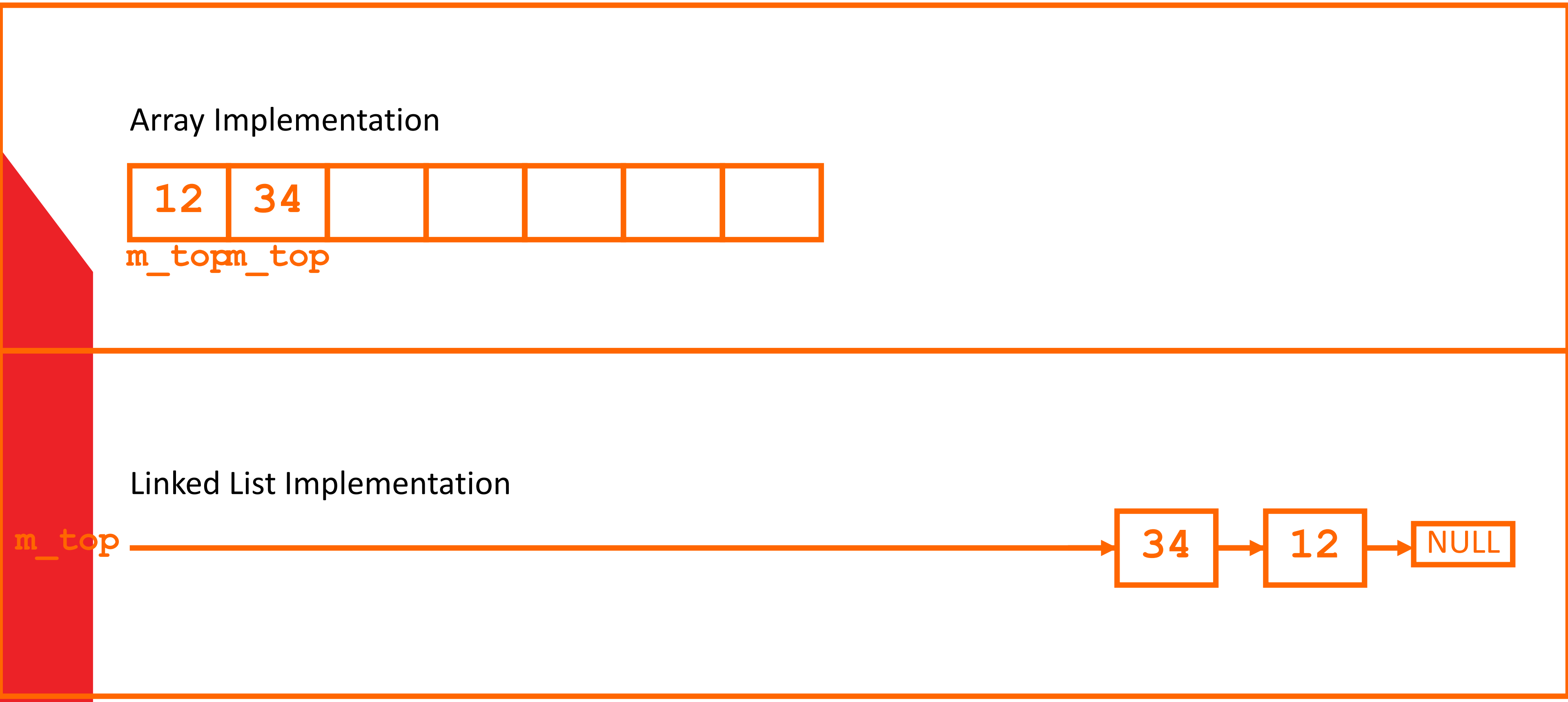
Array Implementation



Linked List Implementation

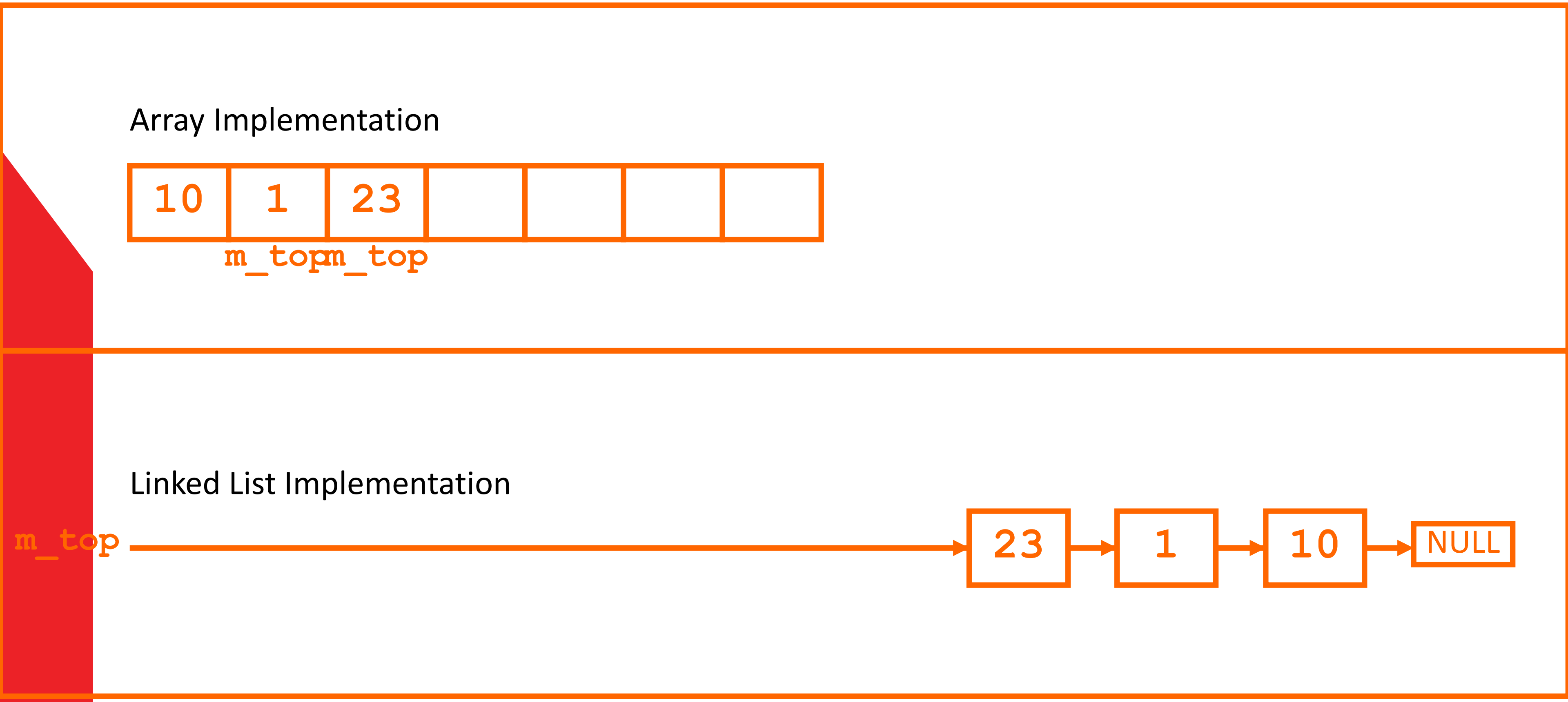
Stack Example (Animation)

`Push(34)`



Stack Example (Animation)

`Push(23)`



Stack Example (Animation)

`Push(36)`

Array Implementation

10	1	23	36			

`m_top` `m_top`

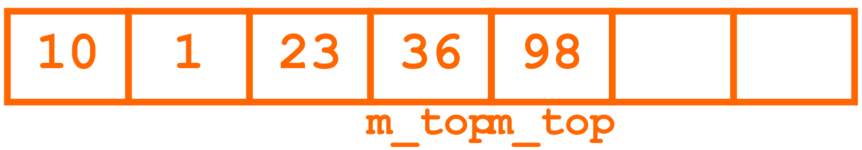
Linked List Implementation

`m_top` → `36` → `23` → `1` → `10` → NULL

Stack Example (Animation)

`Push(98)`

Array Implementation



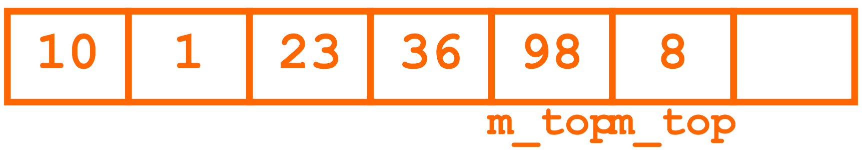
Linked List Implementation

Murdoch University

Stack Example (Animation)

`Push(8)`

Array Implementation



Linked List Implementation

Stack Example (Animation)

`Push(76)`

Array Implementation

10	1	23	36	98	8	76
					m_top	m_top

Linked List Implementation

m_top → 76 → 8 → 98 → 36 → 23 → 1 → 10 → NULL

Stack Example (Animation)

`Push(66)`

Array Implementation

10	1	23	36	98	8	76

`m_top` ← OVERFLOW

Linked List Implementation

`m_top` → 66 → 76 → 8 → 98 → 36 → 23 → 1 → 10 → NULL

Murdoch UNIVERSITY

END

Array Push Algorithm

```
PUSH (DataType data): boolean
    IF m_top >= ARRAY_SIZE-1
        return FALSE
    ELSE
        Increment m_top
        Place data at position m_top
        return TRUE
    ENDIF
END Push
```

Murdoch UNIVERSITY

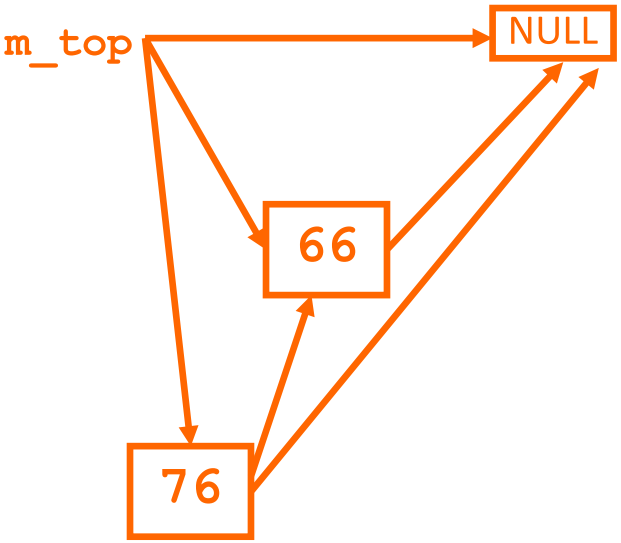
Array Pop Algorithm

- POP (DataType data): boolean
- IF m_top < 0
- return FALSE
- ELSE
- data = data at position m_top
- Decrement m_top
- return TRUE
- ENDIF
- END Pop

Murdoch UNIVERSITY

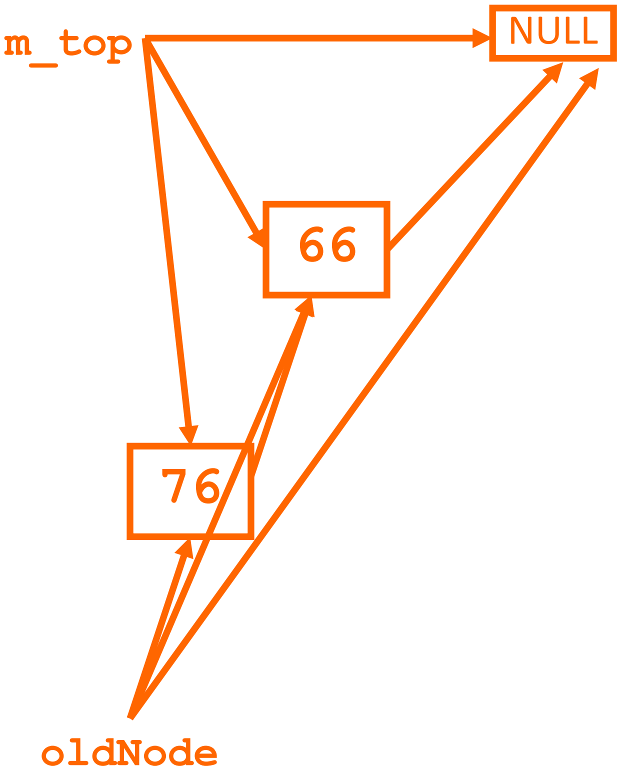
Linked List Push Algorithm

- PUSH (DataType data): boolean
- IF there is memory on the heap
- Get newNode from the heap
- Put data into the newNode
- IF m_top is NULL
- m_top = newNode
- ELSE
- newNode.next = m_top
- m_top = newNode
- ENDIF
- return TRUE
- ELSE
- return FALSE
- ENDIF
- END Push



Linked List Pop Algorithm

- POP (DataType data): boolean
- IF m_top == NULL
- return FALSE
- ELSE
- data = m_top.data
- oldNode = m_top
- m_top = oldNode.next
- release oldNode memory
- oldNode = NULL
- return TRUE
- ENDIF
- END Pop



Using the STL

- The other possibility is to use one of the STL structures.
- If using a vector or list, then the algorithms above barely change.
- However, remember that the structure *must* still be encapsulated in a class, otherwise it will not have just the **`Pop()`** and **`Push()`** that it is supposed to have.
- Finally, there is the STL stack class, (requiring <stack>), which is obviously the **best** STL class to use your Stack class.
- STL stack is an adapted STL container (container adapter) for special use as a stack. No iterators are provided.
- However, even this must be encapsulated if it does not conform to our abstraction of what a stack should be (pointed out earlier and see slide notes from earlier). [1]

Murdoch UNIVERSITY

Features of the STL stack which don't fit in with our Abstraction

1. Its **`pop()`** method, only removes the data, it does *not* pass it back to the calling method.

2. In fact there is a **`top()`** method which returns the data (by reference) at the top of the stack.

3. Neither **`pop()`**, **`top()`** nor **`push()`** return a boolean: overflow and underflow must be checked separately.

- Given the abstraction we are after, even the STL stack must be encapsulated.

Murdoch UNIVERSITY

Stack Header File using STL stack

```
// Stack.h
//
// Stack class
// Version
// Nicola Ritter
// modified smr
//--------------------------------------
// NO I/O HERE. LET THE CLIENT DEAL WITH I/O
#ifndef MY_STACK
#define MY_STACK

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

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

```
•	template <class DataType>
•	class Stack
•	{
•	public:
•		Stack () {};
•		~Stack () {};
•		bool Push(const DataType &data);
•		bool Pop (DataType &data);
•		bool Empty () const {return m_stack.empty();}
•	private:
•		stack<DataType> m_stack; // encapsulated STL stack
•	};
```

```
//----------------------------------------
// It is a template, so we have to put all the code
//   in the header file
//----------------------------------------

template<class DataType>
bool Stack<DataType>::Push(const DataType &data)
{
        bool okay = true;
        try
        {
            m_stack.push(data);
        }
        catch (...)
        {
            okay = false;
        }

        return okay;
}
```

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

template<class DataType>
bool Stack<DataType>::Pop(DataType &data)
{
    if (m_stack.size() > 0)
    {
        data = m_stack.top();
        m_stack.pop();
        return true;
    }
    else
    {
        return false;
    }
}

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

#endif
```

Simple Example of Stack Use

```
// StackTest.cpp
//
// Tests Stack classes
//
// Nicola Ritter
// Version 01
// modified smr
// Reverse a string
//
//-----------------------------------------------------

#include <iostream>
#include <string>
#include "Stack.h“          //Our stack

using namespace std;
```

```
•    //--------------------------------------------------
•    typedef Stack<char> CharStack;

•    void Input (string &str);
•    void Reverse (const string &str, CharStack &temp);
•    void Output (CharStack &temp); // const – check what it
•                          //does first??

•    //--------------------------------------------------
•    int main()
•    {
•         string str;
•         CharStack temp;

•         Input (str);
•         Reverse (str, temp); [1]
•         Output (temp);

•         cout << endl;
•         return 0;
•    }
```

Murdoch UNIVERSITY

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

void Input (string &str)
{
        cout << "Enter a string, then press <Enter>: ";
        getline(cin,str);
}

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

void Reverse (const string &str, CharStack &temp)
{
        bool okay = true;
        for (int index = 0; index < str.length() && okay; index++)
        {
            okay = temp.Push(str[index]);
        }
}
```

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

void Output (CharStack &temp) // would const  work?
{
        bool okay;
        char ch;

        cout << "Your string reversed is: ";
        okay = temp.Pop(ch);
        while (okay)
        {
            cout << ch;
            okay = temp.Pop(ch);
        }
        cout << endl;
}

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

Screen Output

- Enter a string, then press <Enter>: This is a string
- Your string reversed is: gnirts a si sihT

- Press any key to continue . . .

Advantages of Implementations

- It is assumed for each of the containers below, that our Stack encapsulates it.

Array	Linked List	list/vector/deque	STL stack
Easy to code	Full memory control	Easy to code	Easier to code compared to all the others.
	Memory 'never' runs out.	Memory 'never' runs out.	Memory 'never' runs out.

Murdoch UNIVERSITY

Disadvantages of Implementations

- It is assumed for each of the containers below, that our Stack encapsulates it.

Array	Linked List	list/vector/deque	STL stack
Can run out of space easily.	More difficult to code as it uses pointers.	Excess code sitting 'behind' the implementation.	Excess code sitting 'behind' the implementation.
		Only available in with some languages. e.g C++ has STL, Java has Java collections framework	Only available with some languages. e.g. C++ has STL, Java has Java collections framework

Murdoch UNIVERSITY

Readings

- Textbook: Stacks and Queues, entire section on Stacks.
- For amore details of Stacks with some level of language independence, see the reference book, Introduction to Algorithms section on "Stacks and Queues" in the chapter on "Elementary Data Structures". You will see how removed the STL stack is from the abstract stack. We want the abstract level – see earlier lecture notes on level of abstractions.
- Textbook: Standard Template Library, section on Container Adapters
- Library Ereserve: Deitel & Deitel, C++ how to program [ECMS]. Chapter 15 part A. [1]