Introducing... C++

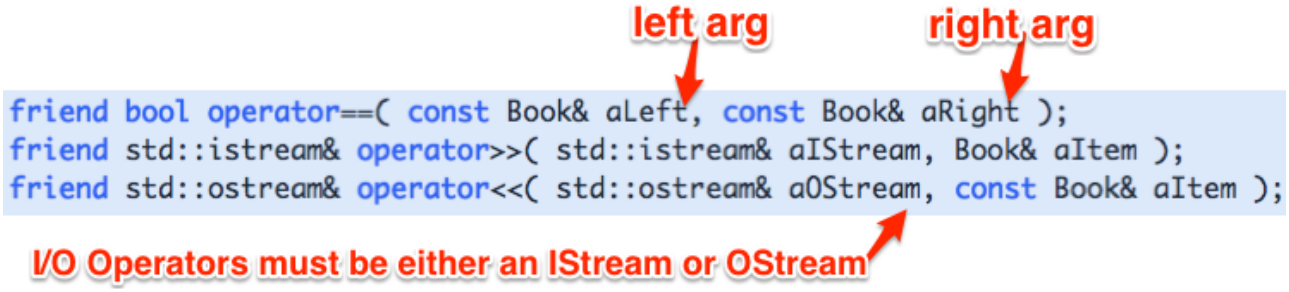
C++ Basics

Constructors

- All **Data Types** need to have constructors
- Constructors serve:
  - 1) To create new instances of the data type
  - 2) To initialise them with fields with **sensible** variables
- Constructors can be overloaded, to initialise them in certain ways
- **Default Constructors** should exist
  - They do not take any arguments
  - They exist only for how the data type should be used in **any** context
  - Where no default constructors exist, compiler will attempt to synthesise one for me.
    - Doesn't guarantee 100% logic!
    - Only works if and only if no other constructors are specified.

Friends

- Not an OO concept
- A friend enables **uncontrollable** access to all **private members** of a class
  - Circumvents OO Encapsulation
- I/O Depends on friend mechanisms
  - The I/O classes are in no way related to the class
  - But in order for **I/O to interact** with the class, we need access to private members
- Friend binary operators **must explicitly define** who is on **BOTH** sides of the operator:



- Compare this with **non-friend member operators**, which implicitly take *this as the LHS operator:

```
Book& operator+=( const Book& aRHS );
```

  - Return *this to return the *updated* Book (i.e., return updated *this*) in this method

A note on IO Stream Operators

- Always return the I/OStream argument to allow for **chaining** on the next argument:

```
aIStream >> lFloat >> lString >> lInt
```

- **aIStream** will return false where invalid data is provided by separated data (e.g., when lString is provided with a *float* and not a string)

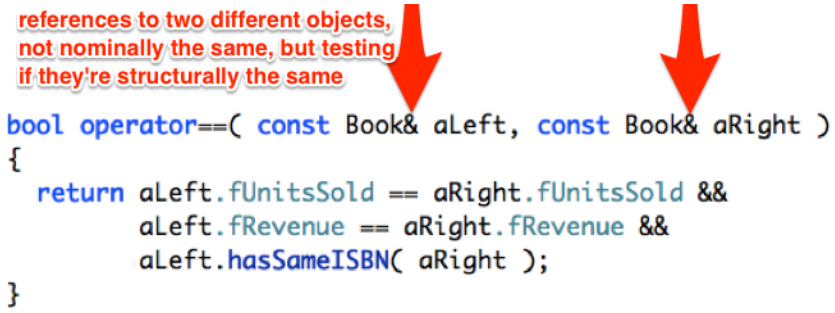
Operator Overloading Notes

- You cannot change the **priority** or **associativity** of an operator
  - If it's a ternary operator, you cannot change it to a binary operator
- Implement the predefined semantics of the operator
  - If it's an incremental operator, we expect it to increment the data type it's being applied on

Equivalence

Structural Equivalence

- *Structurally* the same value, even if not *technically* the same value:



  - E.g., the fields inside two objects (on heap) are the same; they're structurally equivalent.
  - They are not nominally equivalent; their values (pointer addresses) are *different*

Nominal Equivalence

- Two values are nominally the same if they *technically* the same:

```
int valueA = 100;
int valueB = 100;

bool nominallyEquivalent = valueA == valueB; // Results in true

typedef char byte; // A char is nominally equivalent with byte
```

Uniquely C++

Enumerators

- Defines a group of constants, separated by commas

- Each enumerator value is a const data value, where:
  - First one is set to zero, unless overridden
  - Others are **implicitly assigned** the increment of its predecessor

```
enum DaysOfTheWeek { SUN = 7, MON = 0, TUE /* = 1 */, WED /* = 2 */ ... };
```

Typedefs

- **Hide implementations** of a given type
- **Streamline type definitions** to make them easier to understand
- Allow a single type to be used over **different contexts**
  - E.g., we want a new type called byte, that is essentially a char, but used like a byte:

```
typedef char byte; // New type, byte
```

- Type definitions establish **nominal equivalence** between types

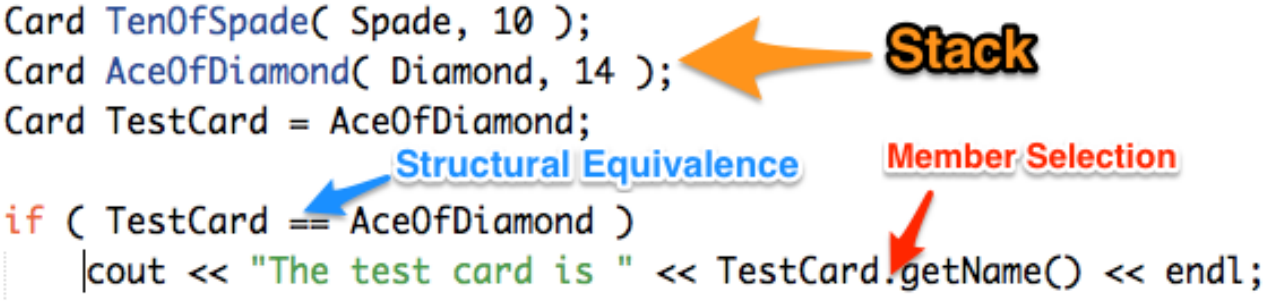
Const Qualifiers

```
#DEFINE B_SZ 128    // Macro definition – no address in memory (cp'ed in)
const int BUFFER = B_SZ; // Initialised at compile time – address in memory
```

Object Models

Value-Based

- Objects exist **on the stack**
- Accessed through **object variables** (object)
- Object memory **implicitly released**

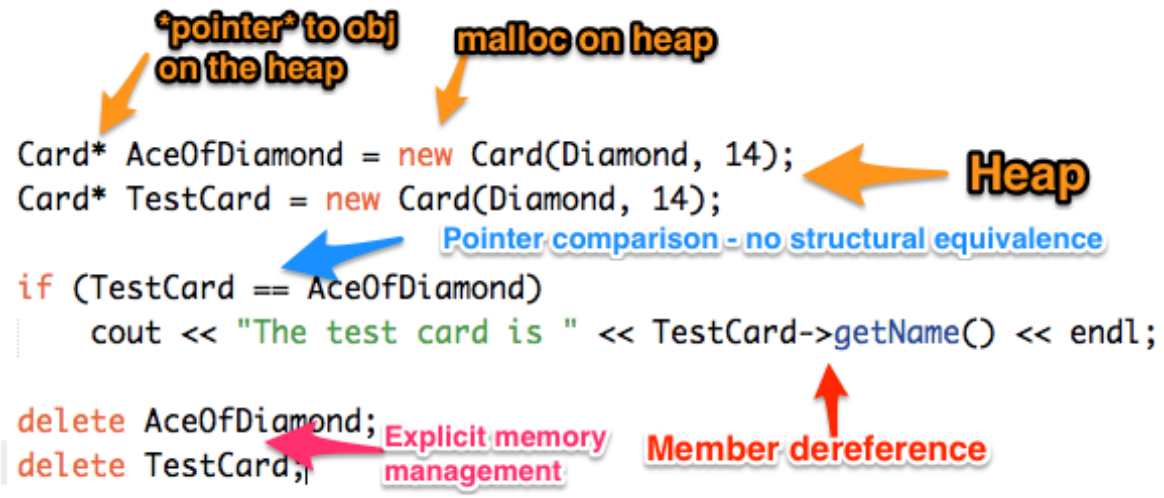


- Objects behave like arithmetic types
  - You can compare them as you would arithmetic numbers

Reference-Based

- Objects exist **on the heap**

- Objects are accessed through **pointer variables** (*object)
  - Dereference the object to access it
- Object memory **explicitly released**



- Objects exist on the heap—you are comparing pointer address values here
  - You are working with pointers to those objects, not the objects themselves

Referencing and Aliases

- References introduce a new name of an object

```
int   blockSize = 512;
int&  buffSize  = blockSize;
```

- Buffer size is a reference to the address of block size (an Alias of Block Size)
  - By mutating buffSize we mutate blockSize
  - You *cannot* change where the **reference addresses**
- **const** addresses give readonly access to the value that the references addresses
  - You *cannot* change the **value contained** where the **references addresses**

Pass By…

- **Value**
  - Side-affect-free definition
  - Doesn't change anything in the **calling context**
    - i.e., nothing changes in the context/stackframe where the call is made
  - Bad for unnecessary copies of objects
- **Reference**
  - Side-affect definition
  - A parameter is a reference to a variable

- Allows for **multiple values** to be 'returned' (mutated) by the procedure
  - Usually we return a boolean to signify if all mutations were successful
- **const** reference parameters are read-only and non-assignable

Reference Data Members

```
class RefMember {
private:
  SomeClass& fRefMember;
public:
  RefMember(SomeClass& aRef) : fRefMember(aRef) {...}
```

- Essentially a promise that an object will eventually occupy fRefMember **in the future**
  - There will be a SomeClass object that RefMember class can use later via this reference
- **WARNING!**
  - Reference data members, i.e., fRefMember, must be initialised before the constructor body is entered
  - We use a **member initialiser function** to do that—these will be called first.

C++ Inheritance

Access Scopes

- **public**
  - Yields a "is a" relationship—causes **subtyping** polymorphism
    - I can use the subtype (BankAccount) wherever I can use the supertype (Account)
    - I can use any supertype member in the subtype
  - Adverse affect: You get **everything**—you get every single public facet of the member
    - Get's a bit fuzzy for 500 methods on a button—most of them are just GUI inheritance
- **protected**
  - Yields a "is implemented in terms of" relationship—causes **subclassing**
  - The inheritor is not a subtype of the class from an external viewpoint—i.e., no subtyping polymorphism will guarantee that you can use either class interchangeably
    - Internally you see the parent, externally you don't
  - 'prunes' the access scope
    - Internally, I see all those inherited methods, but externally you only see my public

Constructors

- When a derived class is instantiated, multiple constructors will be invoked to construct the hierarchy tree, **starting from the root object down to me**

```
Btn() :< Widget() :< WindowController() :< GUIController() :< RootObject()
```

- Before we create a bank account, we **initialise the *super* instance members** by calling the super constructors— It doesn't actually **create** super objects

Destructors

- Destructors in the inheritance chain will be called **starting** from the most **recent class** and **ending** with the **root class** at the **top of the inheritance chain:**

```
Btn() >: Widget() >: WindowController() >: GUIController() >: RootObject()
```

  - Kill the parent first then me

Virtual Destructors

- When deleting a reference polymorphism subtype of a class, ensure that each class in the inheritance chain gets to run:

```
BankAccount *bankAccountPtr;
Account     *accountPtr;

bankAccountPtr = new BankAccount();
accountPtr     = bankAccount;       // Subtype polymorphism; subtype used
                                    // anywhere supertype is used

// Ensure virtual destructor on accountPtr (i.e., bankAccontPtr) will
// destroy bank account…

delete accountPtr;
```

- delete accountPtr… since accountPtr is **dynamically bound** to its subtype bankAccountPtr, the **most recent definition** of the class it is bound to will be called
  - (i.e., BankAccount is the most recent definition of the **value it contains**—even though its a Account class)
  - From the most recent definition, it will destruct from that class' virtual destructor onwards
    - i.e., we're not destructing Account—we're destructing the value within the account—a Bank Account
    - We need to have a virtual destructor so that Account's can be destroyed from their polynomially defined subtypes:

```
delete accountPtr;

–> BankAccount() –> virtual ~Account()  // virtual = destroyed virtually
```

Data Structures

Values

Constants

- Constants do not have an **address**:
  - They're usages are **substituted** with the const value at compile-time
  - As such, we cannot make an **alias** with a constant (since there's no address!)

Primitives & Composites

- Primitive values cannot be **further decomposed**
  - These values are implementation and platform dependent
  - *Truth Tables, Integers, Characters, Strings, Enumerands*
- **Composite values** are built-up from primitives and *other* composite values
  - The layout of composite values is implementation dependent
  - *Records, Arrays, Enumerations, Sets, Lists, Tuples, Files*

Pointers

- References to values; i.e., they **denote value <u>locations</u>**
  - They store the **address** of a value (variable or function)
  - You **cannot** change the actual **address** of where the pointer is pointing to; you can change ***where*** it points to, but you can't change what the address is!
  - Generally, a reference beyond two levels indicates that a composite data structure should be used:
    - I.e., don't go beyond a pointer to a pointer to a value

Sets

- A set is a **collection of elements** that are **possibly empty** and are **unordered**
- All elements satisfy a possibility complex characterising any property P:

$$\{ x \mid P(x) = \text{True} \}$$

  - The set defined as X is defined as the property P must be true for all X values
  - E.g., if X defined as a set of students, and the property P defined male, then:
    - *for all students, they must be male in the X set*
- **Inductive specification**
  - Does the following:

    - **1)** A way to **describe any infinite set** in a **finite number of steps**
    - **2)** It always results in the **smallest possible set (i.e., free of redundancy)**
- It's easier to define a set in terms of itself; this is inductive specification:
- *E.g.,* two properties:
    - **0** $\in$ **S** — this is the **base** clause
        - We start with this base element that exists already in the set; 0 is a subset of S
    - **Whenever x** $\in$ **S, then x + 3** $\in$ **S** — this is the **inductive** clause
        - Given x is an element of S, then we infer that x+3 is also a set
- Sets of strings (S):
    - S = ϵ | aS
        - ϵ — an empty string
        - a $\in \Sigma$ — a is an subset of sigma (the ASCII alphabet)
        - Hence, all strings are either empty or a subset any ASCII char

Indexed Sets

- To obtain an ordering relationship over our unordered sets:
    - we assign each unique element of S
    - to an ordered index set I
    - Thus:
        - $S_I = \{ a_i \mid a \in S, i \in I \}$
        - Thus, an Indexed Set S_I is a **combination** of:
            - set S **(the set of values)** and
            - set I **(set of indexed variables)**

Let A = { a, b, c, d } and I = (S×S, <), then

A_I = { a"1", b"2", c"3", d"4" }

- I is defined as **partial index order, which must match the properties**
    - **SxS** Check if the length of each key each is equal: "123" vs "1"
    - **<** Check if the letter is greater than the other, i.e., "A" < "B"; "4" < "3" etc.
    - Ai is therefore a dictionary/hash map, now ordered with he keys defined in I

Arrays

- A compound data type consisting of: a **type**, an **identifier**, a **dimension**
- Arrays are **ordered, homogenous, and limited (dimensionally)**
- Statically allocated arrays = **constant size** so compiler knows **how much mem'ry to alloc**

```
int STAFF_SIZE = 3;

double salaries[STAFF_SIZE];        // NO: STAFF_SIZE is not a constant!
double salaries[get_staff_size()];  // NO: get_staff_size() = non-constant!
```

- Explicit declaration:
  - **int arr[20] = { 1 }** — arr[0] is 1, arr[1..19] is 0
  - **int arr[] = { 1, 2, 3 } —** array of 3 integers [0] = 1, [2] = 2, [3] = 3
  - **int arr[10]** — array of 10 integers, uninitialised to unknowns (due to primitive)
  - **Banana arr[10]** — array of 10 Banana objects, initialised with their default constructor
- Multidimensional arrays:
  - **int board[3][3] = { { 1, 2, 3 } , { 4, 5, 6 } }** — only first two rows (rotocol) are initialised
  - All multidimensional arrays can have as **many dimensions as needed**
    - **BUT** the last dimension must have a specified size:
    - **int board[][][5]** — while dimensions 1 and 2 are unspecified, the last dimension is 5
- C-Strings are just arrays:
  - **char someString[] = { 'h', 'e', 'l', 'l', 'o', '\0 }**
  - Notice that the string ends in a **null terminator \0**
  - Thus string length is always hello (5) **+ null terminator (1)** = 6
  - Hence this array needs to be defined with a length of 6 should the length be specified:
    - char someString[**6**] = "hello"
- Arrays are **always passed by reference**
  - Arrays are always passed into a function as a **pointer to the first element in the array**
  - **We need to also pass in the size of the array too!**
  - As such, if we modify the array in the function, then it is **also modified in caller context**
  - Hence, when we access an element in the array, we will offset it by the elements:

```
void arrayEls (int array[], int array_sz)      // Must pass in array_sz
{
    for (int i = 0; i < array_sz; i++)
    {
        std::cout << array[i];                 // Access the element by
                                               // offsetting the array[0]
                                               // by i

        array[i]++;                            // Incrementing array here
                                               // implicitly increments
                                               // array in calling context
                                               // without needing a & ref
    }
}
```

Pairs and Maps

- $A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$
  - The cross product of both sets defines a **set of ordered pairs**
  - It is an **associative container**
    - The key is **a** as defined by set **A —** a is an **index into the map**
    - The value is **b** as defined by set **B —** b is a **value being stored or retrieved**
- **Associative array**
  - Elements are **indexed by their key** *not* by their **position**
  - Unlike **indexed sets,** associative arrays do not assume **continuous space in memory**
    - It looks and feels ordered, but **memory is non-continuously allocated**

$$a[i] = \begin{cases} v, & \text{if } i \mapsto v \text{ in } a \\ \bot, & \text{otherwise} \end{cases}$$

- Hence:
  - a[i] is defined as:
    - **v value** if **i is mapped to any value within a**
    - $\bot$ **or undefined behaviour** if there is i does not map a value in a (i.e., exception)

Indexers

- The i is the **index** in the example above
- The index **gives us random access to any element in the map**
- Thus, indexers are **non-sequentially used**

Defining an associative array class

Object that maintains int arrays accessible with a string key

```
class IntArrayIndexer
{
private:
    const int* fArrayElements;
    const int fLength;

public:
    IntArrayIndexer( const int aArray[], const int aLength );

    const int& operator[]( const std::string& aKey ) const;
};
```

The array itself and its length

Do not modify this object

Access a const int reference to the array at this string key

Initialise the array from an externally referenced one

```
const int& IntArrayIndexer::operator[]( const string& aKey ) const
{
    int lIndex = 0;

    for ( unsigned int i = 0; i < aKey.length(); i++ )
        lIndex = lIndex * 10 + (aKey[i] - '0');

    if ( lIndex < fLength )
        return fArrayElements[lIndex];
    else
        throw out_of_range( "Index out of bounds!" );
}
```

Char at i — null char \0

lIndex*=10 @ each loop

Reach the index of the array from the string that was transversed—i.e., convert string to an integer index number that used to offset array (or throw exception if out of range)

Then to use these defined functions:

Our initial array

The indexer passing in the array by ref and size by value

Each key matches an index

For each index (i.e., we must know the size here)

Access the i'th element using the i'th key

```
int a[] = { 1, 2, 3, 4, 5 };
IntArrayIndexer indexer( a, 5 );

string keys[] = { "0", "1", "2", "3", "4" };
int Sum = 0;

for ( int i = 0; i < 5; i++ )
{
    Sum += indexer[keys[i]];
}

cout << "Indexed sum of [1,2,3,4,5] is " << Sum << endl;
```

Iterators

- Unlike indexers, iterators give us **sequential access** to a collection set
- It is **an abstraction that supports flexibility Traversal** along a set
- It essentially gives us a **foreach** loop
  - We **don't** have to specify at which index to **start transversing** at
  - We **don't** have to specify at which index to **stop transversing** at

```
Iterate until iterator != endIterator (end boundary):

| ---------------------------------> | [endIterator] = iter.length+1
RB                                   RB
```

- Different kinds of iterators allow us to do different things:
  - **Input Iterator**
    - **Reads** forward
    - *Provided by istream*
  - **Output Iterator**
    - **Writes** forward
    - *Provided by ostream*
  - **Forward Iterator**
    - Read and write forward (action is up to you)
  - **Bidirectional Iterator**
    - Read and write forwards and backwards
    - *Lists, sets, multisets, maps, vectors, dequeue, string, array*
  - **Random Access Iterator**
    - Read and write in any order, jumping as many elements as needed.
    - Point the iterator to **any section** of the collection
- `begin()` **method**
  - Returns an iterator **at the first element** position (i.e., the **lefthand roadblock**)
- `end()` **method**
  - Returns an iterator **1 element after** the last index position (i.e., the righthand roadblock)

Expression	Effect	Iterators That Use It
`*iter`	**Dereference the iterator** Get to the actual element—iterators are contextually just a pointer to the actual element	Input, Forward, Bidirectional
`iter->member`	**Dereference the iterator and access member** Dereference to get to a composite element, then access that composite element's member	Input, Forward, Bidirectional
`++iter`	**Move iterator forwards** Return element at the new position	Input, Output, Forward, Bidirectional
`iter++`	**Move iterator forwards** Return element at the position before it was iterated	Input, Output, Forward, Bidirectional
`iter1==iter2`	**Boundary Check**	Input, Forward, Bidirectional
`iter1!=iter2`	**Boundary Check**	Input, Forward, Bidirectional
`*iter = value`	**Dereference the iterator and write** Gives write access to the actual value that the iterator is pointing to	Output
`iter1 = iter2`	**Assigns two Iterators for equality**	Forward, Bidirectional
`--iter`	**Move iterator backwards** Return the element at the new position	Bidirectional
`iter--`	**Move iterator backwards** Return element at the position before it was iterated	Bidirectional
`iter[n]`	**Read access at element n** Gives read access to any element in the set which the iterator is transversing through	Random Access
`iter += n`	**Steps n elements forwards** Moves the iterator n elements forwards	Random Access
`iter -= n`	**Steps n elements backwards** Moves the iterator n elements backwards	Random Access
`n + iter`	**Returns the iterator of the nth next element**	Random Access
`n - iter`	**Returns the iterator of the nth previous element**	Random Access
`iter1 - iter2`	**Returns the delta between two iterator positions**	Random Access
`i1 >/<(=) i2`	**Comparison operators between operators**	Random Access

Example

```
class IntArrayIterator
{
private:
  const int* fArrayElements;
  const int fLength;
  int fIndex;

public:
  IntArrayIterator( const int aArray[], const int aLength, int aStart = 0 );

  const int& operator*() const;
  IntArrayIterator& operator++();
  IntArrayIterator operator++( int );
  bool operator==( const IntArrayIterator& aOther ) const;
  bool operator!=(const IntArrayIterator& aOther) const;
  IntArrayIterator begin() const;
  IntArrayIterator end() const;
};
```

Reference to the array elements

Length of the array

The current index

Default value makes starting index pos aStart at 0

Dereference operator

Unused int argument declares postfix++

// prefix

// postfix (extra unused argument)

Postfix will return the previous value (i.e., copied tmp value)

Barriers

```
IntArrayIterator::IntArrayIterator( const int aArray[], const int aLength, int aStart ) :
    fArrayElements(aArray), fLength(aLength)
{
    fIndex = aStart;
}
```

Use member initialisers to initialise const members

Iter element access operator: iter*

```
const int& IntArrayIterator::operator*() const
{
    return fArrayElements[fIndex];
}
```

constant

reference

no changes to array itself

return the current index of the array elements

Note: *roadblocks make us assume that we don't need to check fIndex is in bounds!*

++postfix operator (returns new...)

```
IntArrayIterator& IntArrayIterator::operator++()
{
    fIndex++;
    return *this;
}
```

Increment the index

Return reference to this (i.e., el @ new index)

postfix++ operator (returns old...)

```
IntArrayIterator IntArrayIterator::operator++( int )
{
    IntArrayIterator temp = *this;
    fIndex++;
    return temp;
}
```

returns value; not a reference

postfix since empty int arg

get current (old) el

increment index

return 'old' el

```
bool IntArrayIterator::operator==( const IntArrayIterator& aOther) const
{
    return (fIndex == aOther.fIndex) &&
           (fArrayElements == aOther.fArrayElements);
}
```

Equivalent if indexes are same

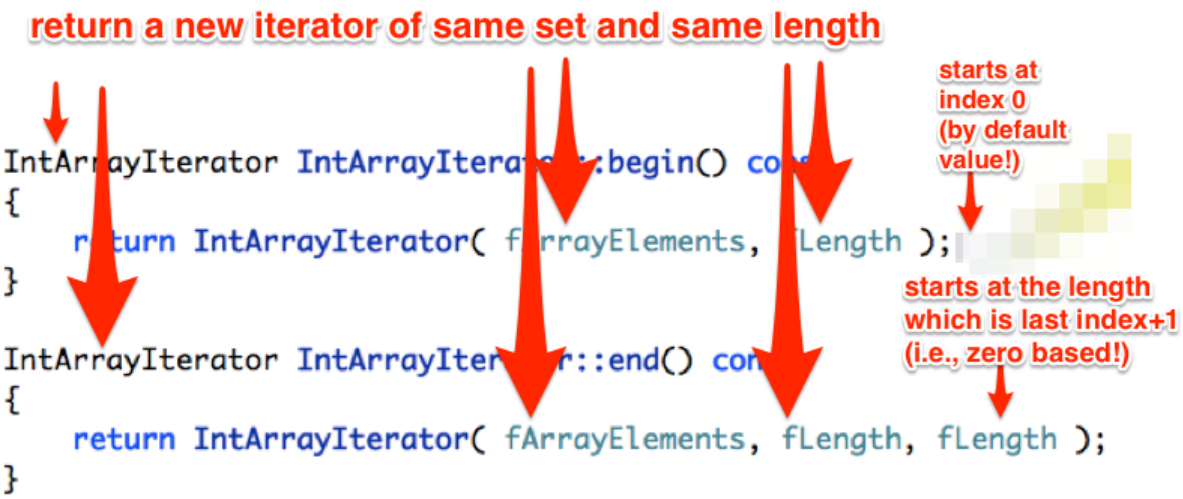
Equivalent if set is same

```
bool IntArrayIterator::operator!=( const IntArrayIterator& aOther) const
{
    return !(*this == aOther);
}
```

not equivalent if not equal :)

Don't reinvent the wheel!

- Road Block Implementation:



- Actually using the iterator:

```
int array[] = { 1, 2, 3, 4, 5 };
int sum = 0;

// For each loop: make the iterator access the elements, not accessing them
// directly!
for ( IntArrayIterator iter(array, 5); iter != iter.end(); iter++ )
    sum += *iter;
```

Patterns

What is a Design Pattern?

- Helps us solve **recurring problems** in programming
  - Identify the problem
    - BUT: **Hard** to recognise the **applicability** of the **pattern to use** with the **problem**
  - While we can't change the language itself, we can introduce standard patterns to use
- **Proven** and **good solutions** to these problems
  - Same solution approach to these problems
  - Don't have to reinvent the wheel
  - Use best practises which knowingly work
  - BUT: Only just **generalised** '**blueprints**'—not actual code abstractions
    - *Think Housing Codes—different houses, but same follow the same Housing Code*
- Reduces **maintenance** overhead
- **Design Patterns aren't about Design**
  - They describe the **iteration, communication and aggregation** of entities in programs
  - It's all about **managing complexity** of these entities
  - They **do not tell you** about application-specific abstractions

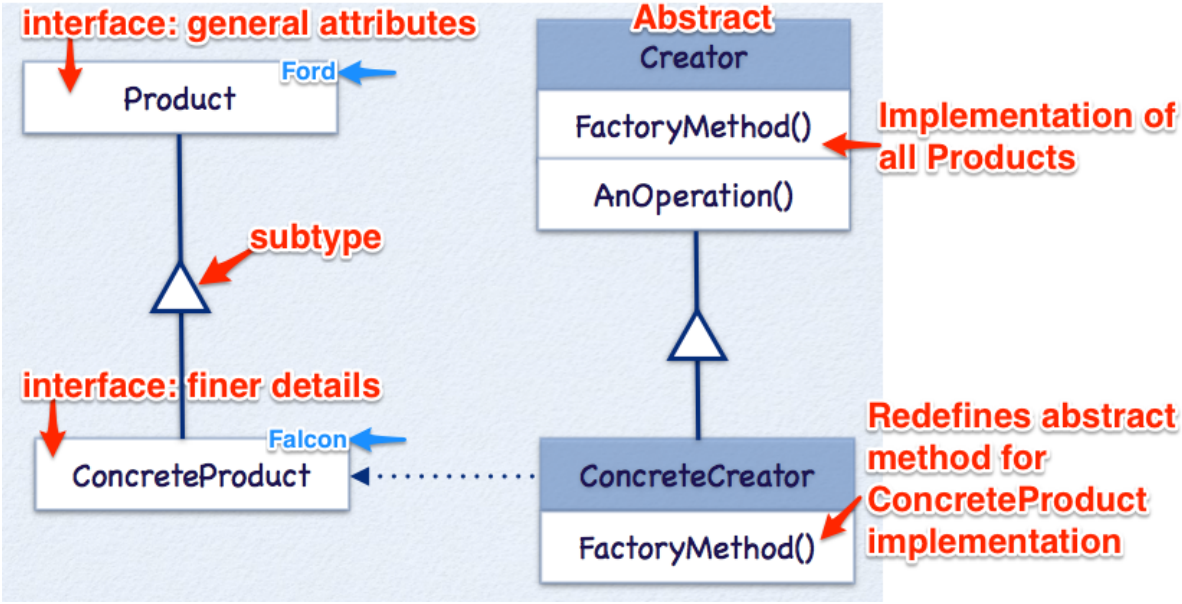
The Four Design Pattern Elements

- **1) Pattern Name**
  - Patterns have names to improve **communication** between programmers
  - Programmers **abide** by these names to know how a **program's application** of patterns
  - *My program applies Factory in XYZ… my program has MVC in ABC…*
- **2) Problem**
  - Describes **when to use** the pattern if the problem can be identified in your program
  - *An iterator patterns helps us with Traversal problems*
- **3) Solution**
  - Describes the **elements** that **constitute** the design
  - Remember, these are **general guidelines only**
    - An iterator pattern is but a basic guide
    - It is not application specific—need to **implement** for it to work in **your own code**
- **4) Consequences**
  - The results and trade offs when using and applying the pattern to your code

The Three Classifications of Design Patterns

(#1) Creational Patterns: *E.g. Factory Method*

- Focused on **Object and Class creation**
- They **abstract** the **instantiation process**
  - Propose solutions on how to **create** objects and classes
  - Propose solutions on **delegation** of an object to another object
- ***Example: Factory Method***
  - **Product:** An interface that defines **general** attributes which all objects of this class have
    - *These are all Ford cars*
  - **ConcreteProduct:** A subtype of Product that has the finer details of the Product
    - *These are Ford Falcons*
  - **Creator:** Defines abstract FactoryMethods that produce the **basic properties** of a Product
  - **ConcreteCreator:** Allows a more appropriate FactoryMethod to be overridden so it returns the appropriate ConcreteProduct
    - Relies on its subtype's FactoryMethod to return the instance of the appropriate ConcreteProduct



Ford Factory

- Send a request to the factory to create a car with 3 doors, V8 engine, color red
- Factory doesn't interface with the client on how the cars are made
- It just guarantees a car with the desired specifications will be made

Iterator Factory

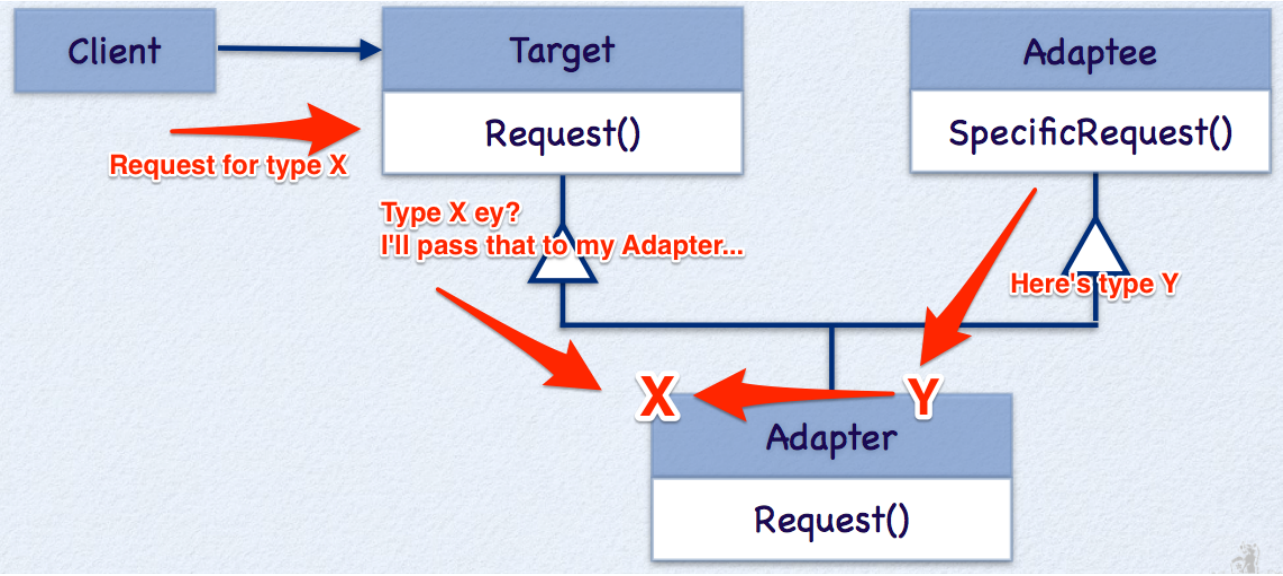
- **Its own Iterator class is a factory to the begin() and end() roadblocks**
- **These two methods are the factory methods for the beginning/ending iterators**
- **Clients will never see the implementation of how to these specialised roadblock iterators are *actually made* (i.e., I don't know what their constructors params are!)**

(#2) Structural Patterns: ***E.g. Class/Object Adapter***

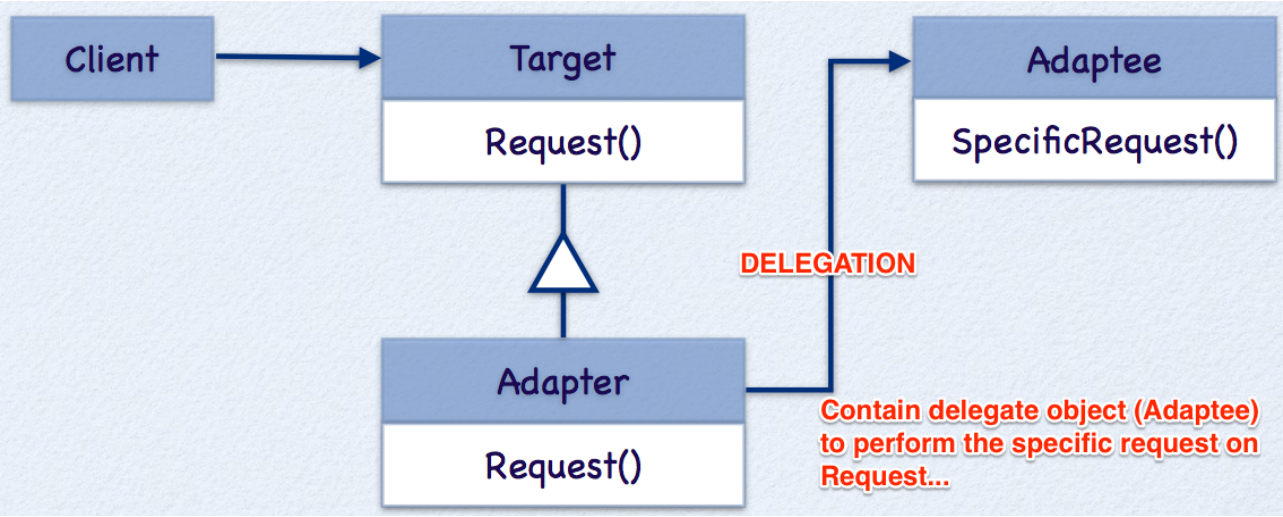
- Focused on the **organisation and composition of classes**
  - Use **inheritance** to **AGGREGATE** services provided by classes together
  - Expose the **final aggregation** of those services in the interface; all gets matched in the implementation
  - Combine classes to **realise new functionality—**aggregate to **get what I want when I want**
    - Derives from the ability to change composition at runtime; impossible with static classes
    - It's the **glue** to connect several objects
      - That glue is the **desired feature provided** by the pattern
- Useful for **large-scale projects** with lots of different interacting **classes**
- ***Example: Adapter***
  - **Convert** the interface of a class **into another interface** that **clients expect**
    - Overcomes mismatch with clients and services by wrapping features of two classes into one seamless class; adapts the two classes together
  - **Class-Based Adapter**
    - Class-based Adapters lets two classes work together, which could not otherwise be possible due to incompatible interfaces
      - *String to Number adapter—client wants to use a string on Target; Target doesn't know how to use strings, so it gets Adapter to get the string for it…*
      - Client's call operations on the **Adapter Instance** using the way that **they expect**
      - The Adapter Instance then translates that request so that the Adaptee can respond
    - **Target:** Clients request a service from the Target
      - The target doesn't implement this service!
      - But the Adaptee implements the service!
    - **Adapter:** Client's request on the Target is passed onto the Adapter—translate request to way client wants by converting the Adaptee's resp. to client's expectation

- **Adaptee:** Adapter's request is forwarded from the client to me; I respond with a specific request and the adapter will respond back in the way the client wants

- **Object-Based Adapter**
  - **Don't expose** the original adaptee object
  - Delegate requests to it from the original Target class



CLASS ADAPTER—Aggregate Classes to Get What I Want



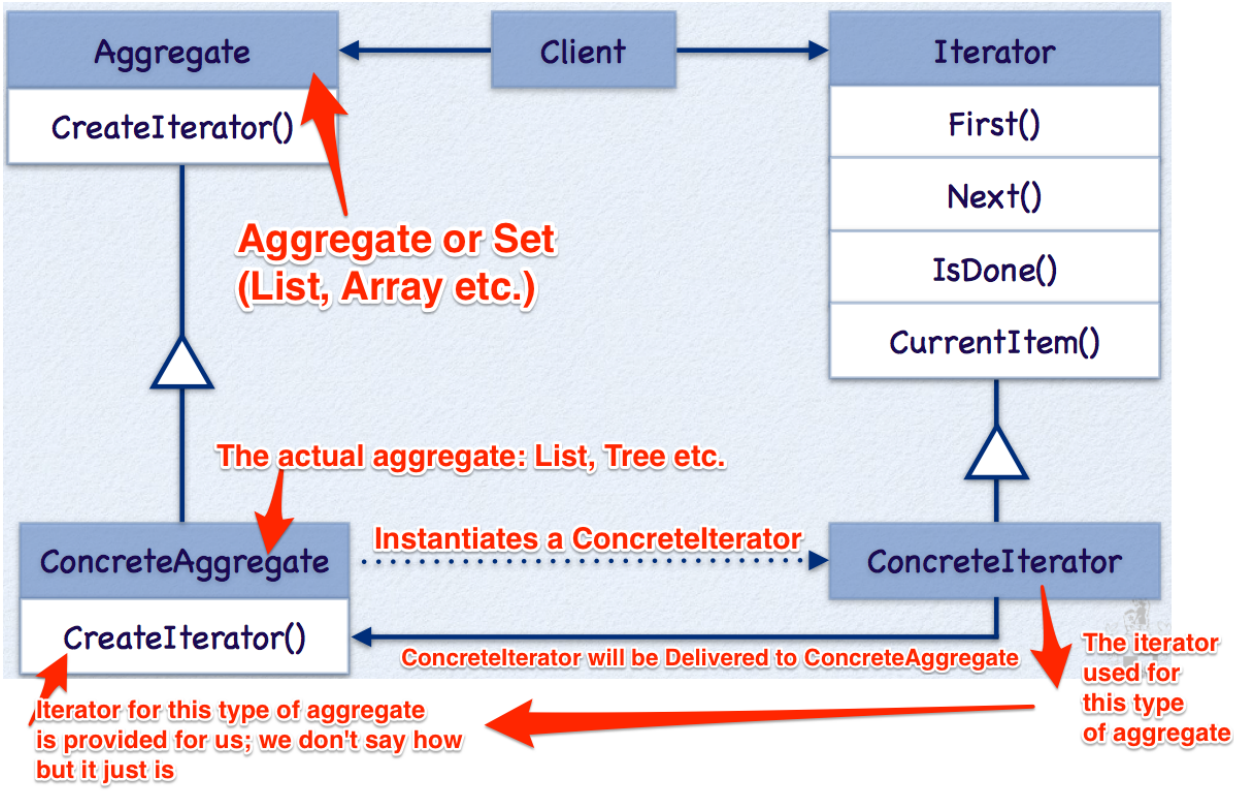
OBJECT ADAPTER—Contain Adapter within the class; Adapter delegates responsibility to Adaptee

(#3) Behavioural Patterns

- **Isolating responsibility** of aspects and **distributing behaviour**
  - Prevent responsibility of programming aspects from being held all over the place by different objects
  - **Different aspects** of the program should be contained by respective classes
    - e.g., Logic Aspect vs Concern in Aspect should be in two different classes

- Object composition, not inheritance—how a group of objects will work together so that all aspects are covered

- *E.g. MVC—separation of concerns (aspects!) into different objects*
- ***E.g. Iterator***
  - Provides **access and Traversal** to all elements in a set or aggregate
  - Abstract away implementation of how we know where we are in that set
  - **ConcreteIterators** keep track of the current object in the set and can compute the next object to be traversed



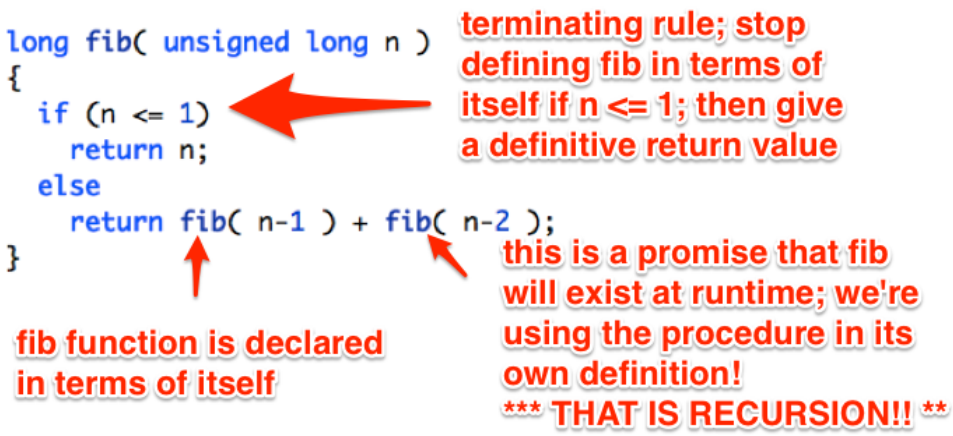
Recursion

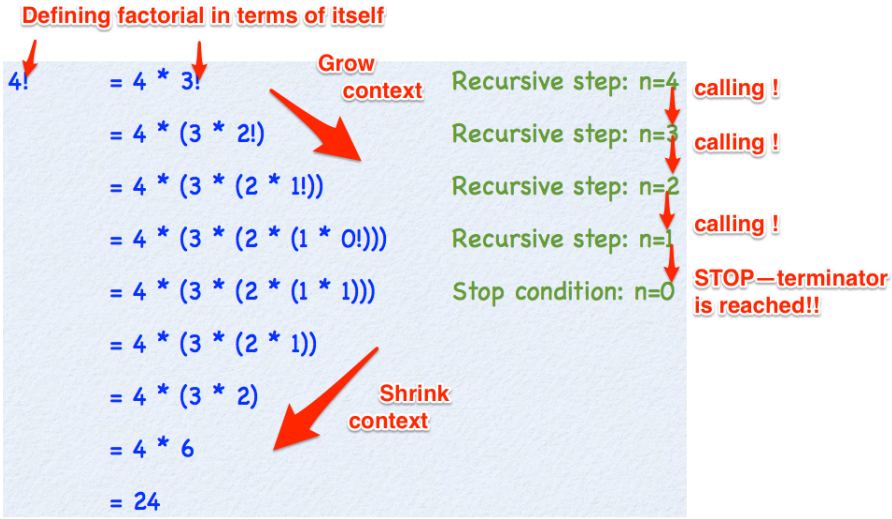
- Effectively[1] a **structural design** pattern
  - Decompose a **larger problem** into a **set of smaller problems** *of the same set* or form
  - We **aggregate** a self-similar composition into smaller parts composed of the same problem
- **"If procedures contain within the body a call to itself, then recursion has been used"**
- There must always exist a **terminating rule:**
  - The terminating rule is **one sub-problem** that can be solved directly **without recursion**
  - If there is no terminating rule, we get endless loops of recursion

[1] Not actually a design pattern from the group of four

- Recursion reduces a problem into smaller instances of the same problem:
  - X = … X …
  - The problem X is now declared in terms of itself

Examples





- **Warning with recursion**—each time we call ourself, we add ourself on top of the stack (consider context growing above)
- Stack size is limited to 8KB—**stack overflows are intermittent** and cause recursive failure
- **Types of Direct Recursion**
  - Try to use **Direct Left** and **Direct Right** Recursion to save stack space
  - **Direct Left Recursive**:
    - X = X T
      - *where X is what we do and T is the terminator*
    - do { X } while (T)
    - Note, we're doing X *THEN* checking for T

    - This can cause non-terminating by not checking terminator before we do X

- **Direct Right Recursive:**
  - X = T X
    - *where T is the terminator and X is what we do*
  - while (T) do { X }
  - Note, we'll only perform X if T is true—**T will terminate X before X occurs next!!!**

Singly-Linked Lists

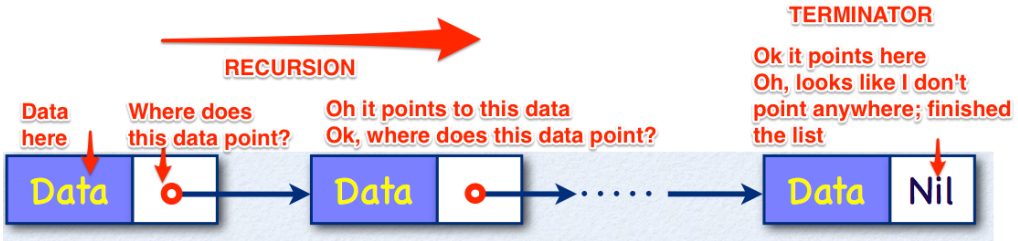
- **What's Wrong With Arrays?**
  - Arrays assume a **contiguous block of memory**
  - Side affects:
    - 1 - We must **allocate all the space** we'll ever need up front
      - Even if we're not sure we'll need all of it
      - Even if we'll only use 100% of it at certain times only

```
[ 3 ] [ 5 ] [ 9 ] [ 2] [ 4 ] [ _ ] [ _ ]
```

    - 2 - Resizing an array causes shifts

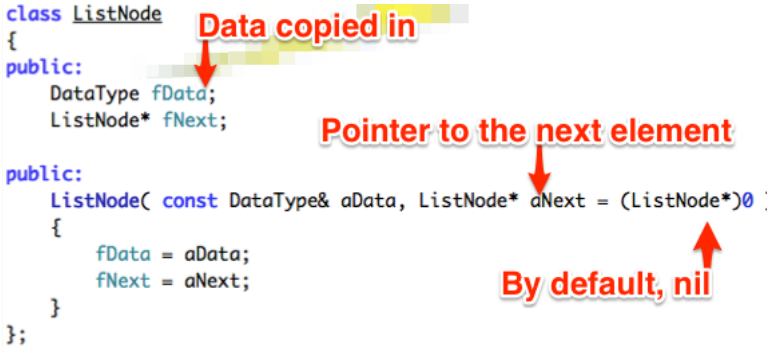
```
[ 3 ] ~~[ 5 ]~~ [ 9 ] [ 2 ] [ 4 ] [ _ ] [ _ ]
becomes shifted:
[ 3 ] [ 9 ] [ 2 ] [ 4 ] [ _ ] [ _ ]
```

    - 3 - Insertion and deletion of an array causes shifts
- **Using linked lists**
  - Looks like an array
  - But **not a contiguous block of memory**
    - (use pointers to point anywhere for next node)
  - Use **data value and a pointer (a node)** to have some data value point to the next node
    - A data value can be **primitive**, **composite** or even **another pointer**
  - It's a **recursive** data type:
    - nodes point to nodes of the same type until **nil pointer node**
    - **nil pointer (NOT NULL!) is the terminator** for the recursive data type



- **Set of pairs**
  - ***Element*** The datum that you want to store
  - ***Next*** The forward pointer to the next 'pair'
    - The next pointer creates a Traversal order from the first pair to the sentinel or terminator pair

Setting Up A Linked List ListNode



- The fData (with arbitrary type DataType) is **copied in**
  - *Passed by reference* to the constructor to avoid unnecessary copying
  - *Copied into fData by value* during construction
  - This means **changes of data outside of aData the list will not affect the list**
    - Thus we avoid aliasing by using **value-based containers**
- **Recursive data structure**
  - **fNext** is a **pointer to itself**
  - For a ListNode to be declared, it has its own definition inside its definition
  - Without the pointer to ListNode*, this would be impossible:
    - In order to define ListNode we need to know how much size to allocate to it
    - ListNode is recursively defined:
      - In order to determine the size of ListNode, we need to know the size of ListNode ad infinitum
    - BUT if pointers are used, we just **store the address** of the ListNode
      - A pointer's storage size is either a 32 or 64 bit address—thus we can allocate it in a **concrete size**
- **Sentinel nil:** `(ListNode*)0`
  - To terminate a group of ListNodes, **nil** pointer to fNext is set: 0 cast as ListNode pointer
  - This is **not null: NULL ≠ Nil**
    - NULL means **'nothing'**—it has no address (#define 0)
    - Nil means that fNext points to nothing—we still point to it (HAS AN ADDRESS!)
      - It is a **dedicated** value that represents the **end of the list**
      - ***Denotes*** no value, but not isn't **no value! It just has no value**

- **This is given as a default argument:**
  - If we don't **specify** an fNext in the constructor, then we assume **an isolated list**
- Remember:
  - References are aliases to objects that have a fixed and unique location as long as the reference is in use
  - Pointers store stack (0xFF—negative) or heap (0x++) addresses

ListNode Iterators

- Since a ListNode is a generic/template, our NodeIterator must too be an iterator!
  - By transitivity, if an X is a generic, then an X Iterator is also a generic:
    - We can use a typedef to make this easier to read:

```
typedef ListNode<int>         IntegerNode;
typedef ListNodeIterator<int> IntegerNodeIterator;
```

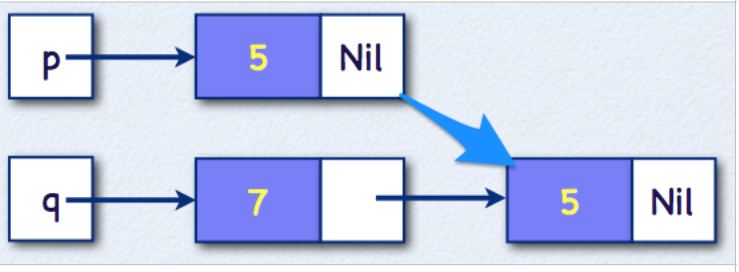
  - Thus, we need to be able to define a generic NodeIterator for all generic ListNodes

Node Construction

- Internally, we will use reference based objects to access our Nodes; it makes it easier for us to work with the pointers without having to worry about reference types.
- **Internally—nodes on Heap**

  Externally—Iterator on Stack

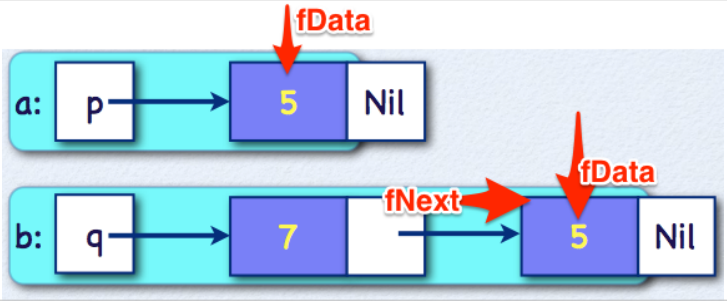
```
IntegerNode *p, *q
p = new IntegerNode(5);
q = new IntegerNode(7, p);
```



Node Access

- Dereference the pointer and access the fData member
- In this case, we can access p's 5 from via both **p and q**
  - **Use fNext to get to the next node that q points to—p!**

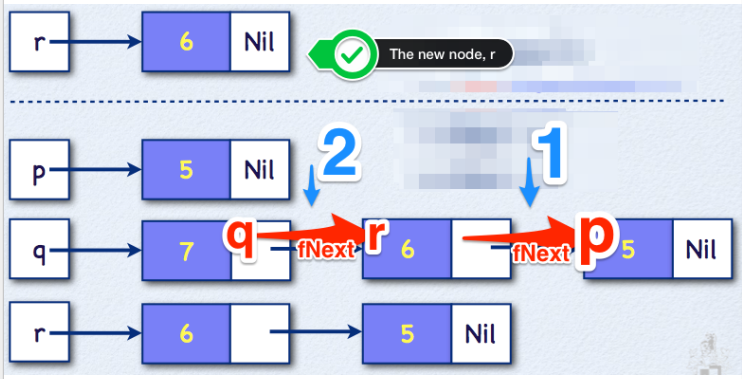
```
int a = p->fData          == 5
int b = q->fNext->fData   == 5

// since q->fNext == p!
```



Inserting a Node—insert r in between q and p

- 2 Steps:
  - **1)** Set where the new node, r, is going to point to
  - **2)** Set where the now dangling node, q, will now point to r
- Since we're going to disconnect p from q, it will be **dangling to p still**, hence step 2

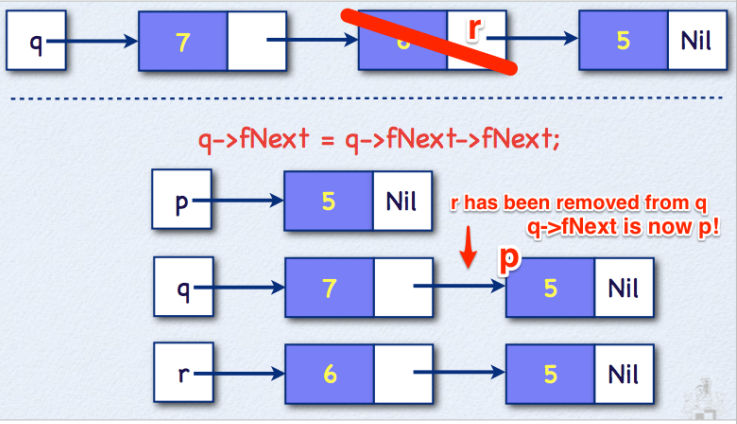
```
IntegerNode *r;

r = new IntegerNode(6);

r->fNext = p;         // STEP 1
q->fNext = r;         // STEP 2
```



Deleting a Node, r

- Simple: Setting the previous node you want to delete (n-1)'s fNext to fNext's fNext
- Set q's fNext to it's current fNext's fNext… you set it and it **deletes itself!**

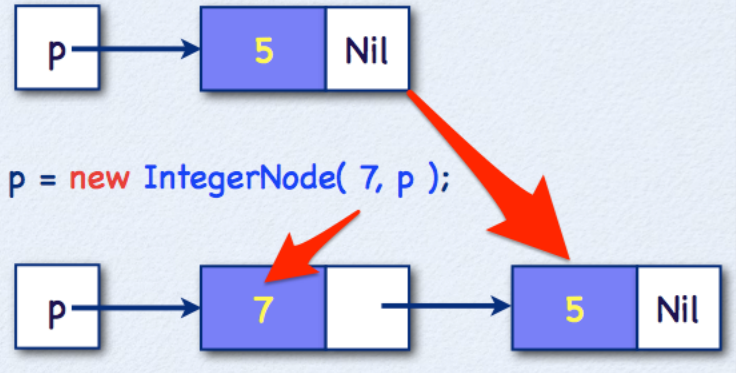
```
// To delete r that was in-between
// q and p...

q->fNext = q->fNext->fNext
```



Inserting at the Top of the List

- Set the topmost node to the new value **with a reference back to itself**
  - The topmost node becomes a tail as a reference back to itself

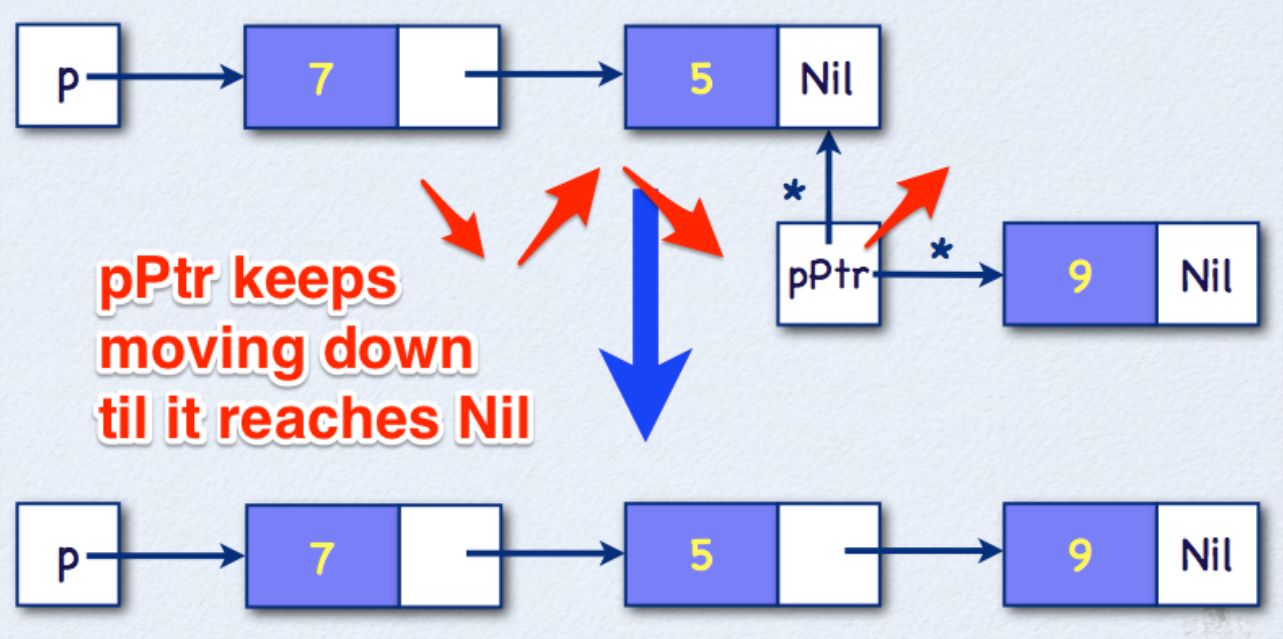
```
// Make p a reference to itself
// in the new IntegerNode

p = new IntegerNode( 7, p );
```



Inserting at the End of A list—Without Aliasing: Pointer to a Pointer

- We need to **search for the end** since we don't know where **nil** is!!
  - Create a pointer to a pointer to a ListNode
  - Keep on iterating while the pointer to a pointer **does not equal nil**
  - The pointer of the pointer becomes the **address of the next element**
- The pointer to the pointer **refers to the last, fNext element that is nil**
- *pPtr will keep moving down until it finally reaches a nil sentinel; it will modify the fNext of that terminator node to the new element and make that new element the nil pointer*

```
// p is nil essentially
IntegerNode *p = (IntegerNode*)0;

// pPtr (pointer-to-a-pointer) references nil
IntegerNode **pPtr = &p;

// While pPtr is not pointing to nil
while (*pPtr != (IntegerNode*)0)
    // pPtr is now equal to wherever the address (&) of the current IntegerNode
    // (*pPtr)'s fNext (i.e., set pPtr to the next non-nil node)
    pPtr = &((*pPtr)->fNext);

// Finally, out of loop: set wherever pPtr is pointing to to
// the new node to insert: new value is 9
*pPtr = new IntegerNode( 9 );
```



Inserting at the End of A list—With Aliasing: Record the last next Pointer

- Simply make a new pointer to the **first element** and a pointer to the **last element**

```
IntegerNode *pList = (IntegerNode*)0;

IntegerNode *pLastNode = (IntegerNode*)0;
IntegerNode *pNewNode = new IntegerNode( 9 );
    First Node is nil?
if ( pList == (IntegerNode*)0 )
    pList = pNewNode;                   // set first node
else
    pLastNode->fNext = pNewNode;        // append new node
    Set the lastNode's fNext instead to the new node
pLastNode = pNewNode;                   // make new node last
```


- **Problem with Singly-Linked Lists:**
  - The deletion of the last node requires a search from the top to find the new last node
  - There's no way of knowing who points to this node (backward!)
  - Has a **private constructor** to set up the NIL node (Black Hole!):
    - Note; static must be set up **OUTSIDE** the class definition

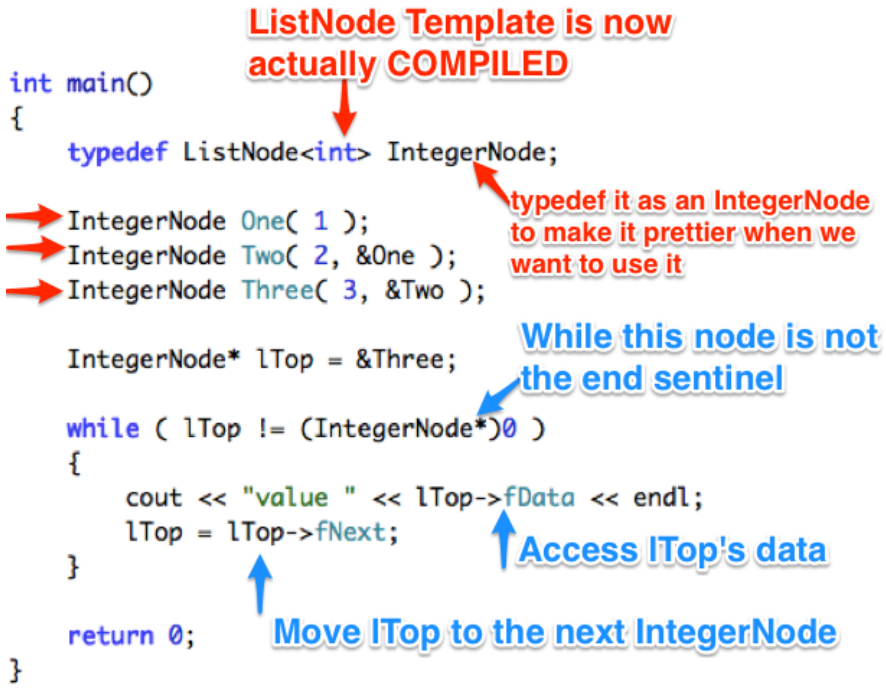
```
private:
    DoublyLinkedList()
    {
       fValue    = DataType();    // Set up fValue to default DataType
       fNest     = &NIL;          // Forward of NIL is NIL
       fPrevious = &NIL;          // Previous of NIL is NIL
    }
public:
    static Node NIL;              // Declare sentinel Node NIL, const above
```

Templates

- We can use templates to **automatically generate specialised classes**
  - **Templates are functions** that returns a **new class** based on a **type parameter**
    - The type parameter is usually denoted as T
    - Can be a class (denoted with **class keyword**), or even a primitive
    - Allows us to make unbound classes on the data types they work on
      - They work **uniformly** on the type variable passed in
  - Templates are **compiled** when they are **specified with a new type**
    - If a template is not specified, then it is not compiled!
    - **Code won't be syntax-checked unless you specified one use of a template!!**

SomeTemplate.h

```
template<class T1, class T2, class T3 ... class Tn>
class ATemplatedClass
{
private:
    T1 fData;    // fData is bound to T1's type, whatever that is...
    T2 fSome;    // fSome is bound to T2's type, whatever that is...
    T3 fThin;    // fThin is bound to T1's type, whatever that is...
};
```



- Templates provide us with a way to **make classes bound and sound**
  - To make all of our members **concrete** and application-specific
  - We don't know what this type will be at the moment, only when its used
  - We provide **the functionality** that is **independent of the member's types**
- **Iterators for Templates**

```
container<T>::iterator pos;
```

  - This will provide a **bidirectional** iterator for the container type of T

Abstraction

- Abstraction:
  - Fights **complexity**
  - Separates **relevant aspects** from **noise**
- Class are a form of abstraction: *they hide the data and algorithms into one ADT*

- By conforming to abstraction, you must distinguish **interfaces** from **implementation**
  - **Interfaces** serve as the ***representation*** of the ADT (Abstract Data Type)
    - The client **conforms** to the **contractual specification** of the Interface
    - Defines the **obligations** on how the client will **communicate** with the ADT
    - **Safety barrier** to the **internal architecture** of the ADT
  - **Implementations** serve as the ***guts or internal architecture*** of the ADT
    - The implementation **gives life to the interface**
    - Provides the **concrete functionality** of the ADT
    - It **internally manipulates** the **state** of the ADT
  - Not always 100% entirely safe
    - Can use pointer programming to bypass the interface and access the implementation
    - Is dangerous since it exposes the underlying implementation to the client
- **By State:**
  - **Set of instance variables** of an object is its state…
  - To modify state, you use the interface

Opaque Representations of ADTs

Black Box

- Black boxes provide uppermost, surface layer only
- Can't talk **at all** with the inner composition (implementation)
- Usually one-way holes for message passing…
- Need a **constructor** to declare instance variables
- **Con:** Hard to debug

White Box

- Full exposure to both the implementation and interface
- **Bypass** the interface and modify the guts of the ADT
- Typically OK if in a **DMZ environment**
  - You fully trust that the **client in the calling context *won't*** have **adverse affects** to your underlying data
  - E.g., Node and IteratorNode—clients will use IteratorNode, but not Node; thus have full access to IteratorNode
- Achievable when all members are **public**
  - To modify state, you can **directly mutate** instance variables since they are public
- Pros:
  - Easy to **debug**
  - Far more **flexible** to work with and **extend**
- Con: **Security!**

Grey Box

- Can view the implementation but **can only modify state to interfaces that allow it**
- Internal structure is available, but no interface methods exist to modify certain states
  - Public methods for modifying some states
  - No methods for modify other states
- Can see the implementation **but cannot bypass the interface**

Modifying Underlying Representations

- Abstraction allows clients never to know what is happening **under the hood**
  - Clients do not care how an ADT is being implemented
    - Thus we can change the implementation as much as we want
    - One **interface** can therefore have many **implementations**
    - ADTs are **implementation independent** since the implementation of the ADT is abstracted away from the client
  - Clients only care if the **service** they want (Interface) from the ADT is being fulfilled
    - They **do care** for fast (quick) and efficient (memory-wise) services

- **A stack can be an array**
- **Once it has > X elements, underlying architecture switches to a List**

The client never knows: They still can use…

`pop(), push(), peek(), size()`

The 3 ADT Aspects/Principles

- All ADTs must adhere to the three principles:
  - **Constructors (Creating the ADT)**
    - We need to provide the ADT with sensible values so it can be initialised properly
    - Some ADTs will need **default constructors**
      - Default constructs are **applicable to ALL calling contexts**
      - Have no assumptions or biases of the calling context
  - **Predicates (Readonly) (Checking if instance is an ADT)**
    - Test is a value is a **representation** of a particular ADT
    - C++ freebie: the typeof operator:
      - C++ provides us with the typeof operator for all ADTs when an ADT is compiled

```
someValue typeof someADT == true or false
```

  - **Manipulators/Observers (Modifying or Getting ADT State)**
    - Use public members (functions etc.) allow ADT **information access** or **mutation**

Container Types, Stacks and Queues

Container Types

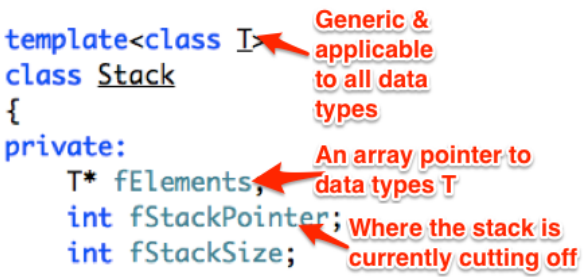
- Container types are Data Types that host/contain other values
- Uses a value based model **only** (in C++)
  - **Why?** C++ is the most consistent with Math
  - In Math: If a value is said to be part of a set, than that set will **own** that value
  - Therefore, we can't use reference based since the set doesn't own that value
    - It can still be modified even outside the set it's contained in!!!
  - Therefore we need to **copy values into** the container
    - Not exactly the same object: structurally the same, nominally different.

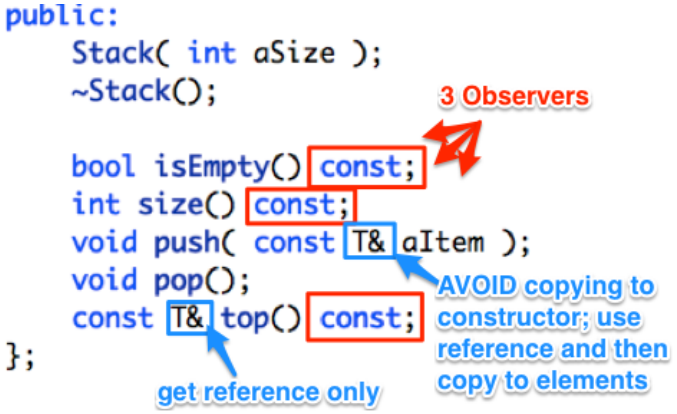
Stacks

- A stack is a linear list (it's a container type)
- Items can only be added or deleted from the top element only: **push** or **pop.**
- Stacks are **LIFO**
  - **Last** element ***pushed*** to the **top** is **first** one ***popped*** **out**
    - **Stack Underflow—**trying to pop when there's nothing to pop
    - **Stack Overflow—**trying to push when there's no left to push
  - We can use stacks for reversing input:

```
push A, push B, push C
pop  C, pop  B, pop  A  (Therefore, A and C swapped due to LIFO!!)
```

  - Stacks also used with Pascal stack frames
  - Each stack frame consists of multiple elements, with a pointer to each new frame where the frame begins and ends, accordingly
- Remember, a Stack is an ADT
  - It therefore must conform to **Construction and Manipulator/Observer methods**
  - We can tell Manipulators from Observers w or w/o const appended to the ADT method





Using Pointers To Dynamically Allocated Memory

- The stack's fElements dynamically allocates new heap memory for it to store the objects it contains
- This means that fElements must be a pointer to dynamically allocated memory as an array
- The pointer **stores the address of the first element in the array**
  - Whenever we point to the first element in the array, we can offset it:

```
fElements[x] means an offset read the array's first element by x
```

```
Stack( int aSize )
{
    if ( aSize <= 0 )
        throw std::invalid_argument( "Illegal stack size." );
    else
    {
        fElements = new T[aSize];
        fStackPointer = 0;
        fStackSize = aSize;
    }
}
```

SHOULD use unsigned (-ve stack size == NO!)

Dynamically allocate the array ON HEAP

Stack pointer to first element in array

```
~Stack()
{
    delete [] fElements;
}
```

Delete objects contained in the array

Delete the array itself

- Using delete [ ] syntax:
  - To release the memory contained within **each element** stored in the array
  - Then delete the array pointer itself (i.e., fElements)
  - **NOT NEEDED FOR ARRAYS OF PRIMITIVES, ONLY FOR ARRAYS OF ADTs!!!**
- **Remember &Address, Reference&!!!!**
- fStackPointer always references the **NEXT FREE ELEMENT!!!!! Therefore if we want to get the top value, we need to subtract one from it:**

```
fSize:              1         2         3         4         5         6
fElements:     [  abc  ] [  def  ] [  ghi  ] [       ] [       ] [       ]
fStackPointer:                                |||
```

```
void pop()
{
    if ( !isEmpty() )
        fStackPointer--;
    else
        throw std::underflow_error( "Stack empty." );
}
```

Trying to pop if empty == stack underflow and therefore undefined behaviour so therefore exception!!!!!!!!

Move the stack pointer down so it pointers lower in the array. DONT DESTROY the element

```
bool isEmpty() const
{
    return fStackPointer == 0;
}

int size() const
{
    return fStackPointer;
}
```

fStackPointer will move up whenever an element is being referenced in the array—hence it's the length of the number of CURRENT elements in the array (NOT SIZE = MAX!!!)

```
const T& top() const
{
    if ( !isEmpty() )
        return fElements[fStackPointer-1];
    else
        throw std::underflow_error( "Stack empty." );
}
```

const REFERENCE to the original value. NOT a COPY.

Can only get the top if not empty

Return the last non-free element

Dynamic Stack

- If we don't know the maximum size of our stack, we need to have a dynamic array
- We can achieve this by using a **list instead of an array**
- Uses the **Adapter Design Pattern**
  - *Adapts* a **list** as a dynamic **stack**
  - Makes a class for us to treat a list as if it were a dynamic stack—but client doesn't know that it's underlying architecture is a list!!!!

```
template<class T>
class DynamicStack
{
private:
    List<T> fElements;

public:
    bool isEmpty() const;
    int size() const;
    void push( const T& aItem );
    void pop();
    const T& top() const;
};
```

Different underlying architecture to support dynamic behaviour

i.e., uses a list with list nodes rather than a *static* array with dynamically alloc'ed elements

SAME public interface

Queue

- A queue is a linear list that allows us to have access to the **top and the bottom** of the list
- We queue things into the stack. They come out the other end when we dequeue them.
- Queues are **FIFO**
  - The **first** element to be ***queued*** (in) is the **first** element to be ***dequeued*** (out)
  - Queue underflows and queue overflows existent
- Dequeue from the list returns a reference to the datum:
  - const T& dequeue()

```
                end[len]                    front[0]
->[enqueue]   [ data ] [ data ] [ data ]   [dequeue]->
```

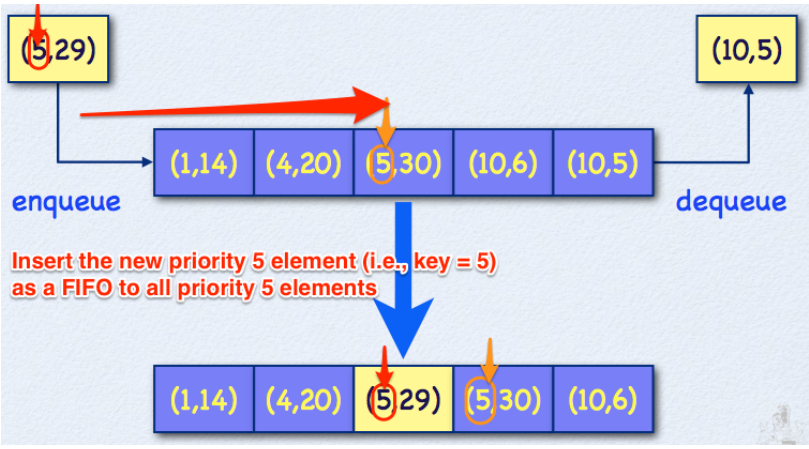
```
void enqueue( const T& aElement )
{
    fElements.append( aElement );
}

const T& dequeue()
{
    if ( !isEmpty() )
    {
        const T& Result = fElements[0];
        fElements.remove( Result );
        return Result;
    }
    else
        throw std::underflow_error( "Queue is empty!" );
}
```

use the list's append method
DON'T REINVENT THE WHEEL!

Return the first in the queue
(remember end of queue is @0)

Use remove to remove node from the list

Priority Queues



- Sometimes we want to ensure that some elements move towards the end of the queue than other elements
- Therefore, the elements in the queue will be prioritised (not LIFO per-se but priority LIFO)
- This will require:
  - A new data type **SortedList<T>**, which will be a list sorted by a certain condition
    - The sorted list ***removes*** **Prepend** and **Append** methods
    - This just leaves the **Insert** method (the SortedList will decide *where* to insert the node!)
  - A new data type **Pair<Key, T>**, which combines an value based on an orderable key
    - For the key to be orderable, it needs to define some sort of sorting mechanism
    - Therefore, it should implement the **operator<** to allow it to be sorted

```
void enqueue( const T& aElement )
{
    fElements.insert( aElement );
}

const T& dequeue()
{
    if ( !isEmpty() )
    {
        // increasing order of priorities
        const T& Result = fElements[fElements.size()-1];
        fElements.remove( Result );
        return Result;
    }
    else
        throw std::underflow_error( "Queue is empty!" );
}
```

As normal

Take of the highest priority only off the list (i.e., fElement's size!)

      - **i.e., is this.key < aOther.key**
    - Therefore, since keys can be sorted, so can a **Pair** (based on its **Key**)
- Therefore, PriorityQueue will just use the **SortedList for fElements, not List**
- **Herein lies starvation issues:**
  - You need to ensure that, just because a priority element is being enqueued, other elements still get a chance to be dequeued
  - Thus, your queue **needs to preserve fairness** to un-prioritised elements

- Do this by incrementing the priority **of all elements** as the queue is dequeued
  - That is, **age your elements** by a certain age factor (to their key—i.e., mature their priority) to ensure that all elements get a go at being dequeued

Copy Control & Memory Management

Memory

- Three main types of memory: Static, Heap, and Stack

Static

- All static memory is mapped to the **read/write .data segment**
- This segment is visible to the entire program
  - The **visibility modifier** of the static variable will vary **where** interaction can occur
  - This memory is typically mapped to virtual memory allocated by the OS
- Static keyword can be used for:
  - Mark the linkage of a variable or functional internal
    - This will make the variable or function visible only in that compilation unit
    - Think of it as a **local global variable**, local to the compilation unit (or .cpp/.h)
  - Retain the value of a local variable between execution calls
    - Recover **previous state** of a variable between its calls
    - *x is initially 10 at the start of the first call, then it's += 10; @ next call it's 20*
  - Declare a class instance variable
    - Variable tied to the class
    - Accessible from all instance members (private) or even public
    - Think Node::NIL
  - Declare a class method

someFile.cpp

```
           int globalProgramVar = 0;   // Globally available in entire prog
   static  int localCompUnitVar = 0;   // Globally available in this .cpp

class A
{
private:
   static  int fMember;                // Available by A::fMember
}

A::fMember = 0;                        // Static members must be declared
                                       // outside the class
```

- Static memory can also be **read-only**
  - Simply add the **const** modifier:

- *static **const** int localCompUnitVar = 0;*

- This will make the static memory be located in the .text segment
  - The .text segment is where the source lies
  - Thus, at compile time, constants are replaced with their constant values

Stack

- Temporary memory for the current stack frame
- All value-based objects are stored in the program's stack
  - Values automatically freed and allocated to stack memory
  - References to stack location are **valid when passed to callee**
  - Stack references ***cannot* be returned from a function**
    - Why? **Can't** traverse **back** from previous stack-frame in C-stack-style (no frame ptr)

Heap

- **Always** allocated heap memory, even if not used
- Used for heap **dynamically allocated objects**
  - Two factors:
    - dynamic allocation *new*
    - object is accessed via pointers
    - `Object* obj = new Object()`
- Not automatically freed once the pointer is out of scope; must delete **manually**

Copy Control in Value-Based Semantics

- Value-based semantics requires copy constructors for **deep copies of objects** when dynamic reference members are used within these objects
  - This ensures for uniqueness, but not *exactly the same object*
- Three factors: **Copy Constructor, Assignment Operator AND *Destructor***
  - Why? Automatic synthesis of these by C++ compiler may ≠ expected behaviour
  - By default, automatic synthesis will **not guarantee deep copies, but shallow copies**
    - **Shallow Copy: Copy** only the pointers; **same address** in pointer (same memory!)
    - **Deep Copy: Duplicate** the ***object***; **different address** in pointer (different memory!)

Copy Constructor

- Defined as a normal constructor that takes a **constant reference to its own type:**
  - `ClassName( const ClassName& instanceOfClass )`
  - `ClassName x = y` *`// will call copy constructor for x with arg y`*
- When implementing copy constructor, be sure to disjoin the members of **aOther with this:**

```
SimpleString::SimpleString ( const SimpleString& aOther )
{
    // 1 - Allocate NEW memory for this fCharacters:

    this->fCharacters = new char[aOther.size()];

    // 2 - Copy all the characters from fCharacters over individually:

    for ( int i = 0; i < aOther.size(); i++ )
        this->fCharacters[i] = aOther.fCharacters[i];
}
```

- This will cause a **deep copy** between the two objects:

```
SimpleString a("Hello");            // "Hello" array of char @ 0x100

// Deep copy 'Hello'
SimpleString b = a;                 // "Hello" deep copied to 0x150 ≠ 0x100


// Stack deletion generation:
<delete b>                          // Delete 0x150
<delete a>                          // Delete 0x100
```

- A **shallow copy** would make b's fCharacter's point to 0x100 still
  - That means it points to the same memory of a's fCharacter's at 0x100
  - This is because only the pointer values (i.e., addresses) are copied, not the heap objects at the end of the pointe
- Without the deep copy (i.e. shallow), we would be doubly deleting 0x100 == **deletion error**
  - ***Can't dealloc already dealloc'ed memory!***

Assignment Operator (b = a *NOT* String b = a)

- An assignment operator will, by default, synthesise shallow copies to the RHS object
- Therefore, the assignment operator will need:
  - **1) Destructor code** to delete members in `this`
  - **2) Copy constructor code** to copy members of `aOther` to `this`
  - **3)** Boilerplate to protect against accidental suicide (i.e., where aOther is this)
    - If you deleted members of this and aOther was this, then when you try step 2, fail!!!

```
SimpleString& SimpleString::operator= (const SimpleString& aOther)
{
   // 3) Protect against accidental suicide; the addr of aOther ≠ this addr
   if (&aOther != this)
   {
       // 1) Destruct this->fMembers
       delete this->fCharacters;

       // 2) Deep copy with same code as copy constructor
       this->fCharacters = new char[aOther.size()] etc etc etc etc… (see ^)
   }
   return *this; // 4) Chaining for assignment x = y = z;
}
```

Copy Control in Reference-Based Semantics

- In referenced-based semantics, you must **not just chain the pointers to the objects:**

```
SimpleString* a = new SimpleString("Hello");   // Make SS a @ 0x500

SimpleString* b = a;                           // Make SS b @ a @ 0x500
```

- In the above example, a **shallow copy** has occurred between the two reference-based obj
- Therefore, we will need a **clone method** that allows us to make a **copy** of the ref-based ob

```
SimpleString* a = new SimpleString("Hello");  // Make SS a @ 0x500

SimpleString* b = a.clone();                  // Make SS b as a clone @0x90
```

- What is clone?
  - An **identical** object of this but allocated on **new fresh memory**
  - Method needs to be **virtual**
    - Otherwise we will **call only the parent copy constructor**
    - *Student is a person*
      - *if not virtual, when I clone a student, I will call the clone on a person instead*
      - *that means all the attributes of a student is now missing*

```
class SimpleString
{
public:
// Copy Constructor for Deep Copies
SimpleString(const SimpleString& aOther);

// Virtual Destructor for any Virtual Class is C++ Protocol
virtual ~SimpleString();

// Virtual SimpleString clone method so children can override
virtual SimpleString* clone();
}

// Clone method just use Copy Constructor to make a reference to a
// new deeply copied clone of this:
SimpleString* SimpleString::clone()
{
   // Newly alloc'ed memory & deep copy this
   return new SimpleString(*this);
}

Why Virtual? So Children Override W/ Their Copy Constructor!
NotSoSimpleString : SimpleString…


NotSoSimpleString* SimpleString::clone()
{
    // Need to use my own subclass'ed Copy Constructor…
    return new NotSoSimpleString(*this)
}
```

Remember:

*this = ClassName&

Handle and Destruction in RBS

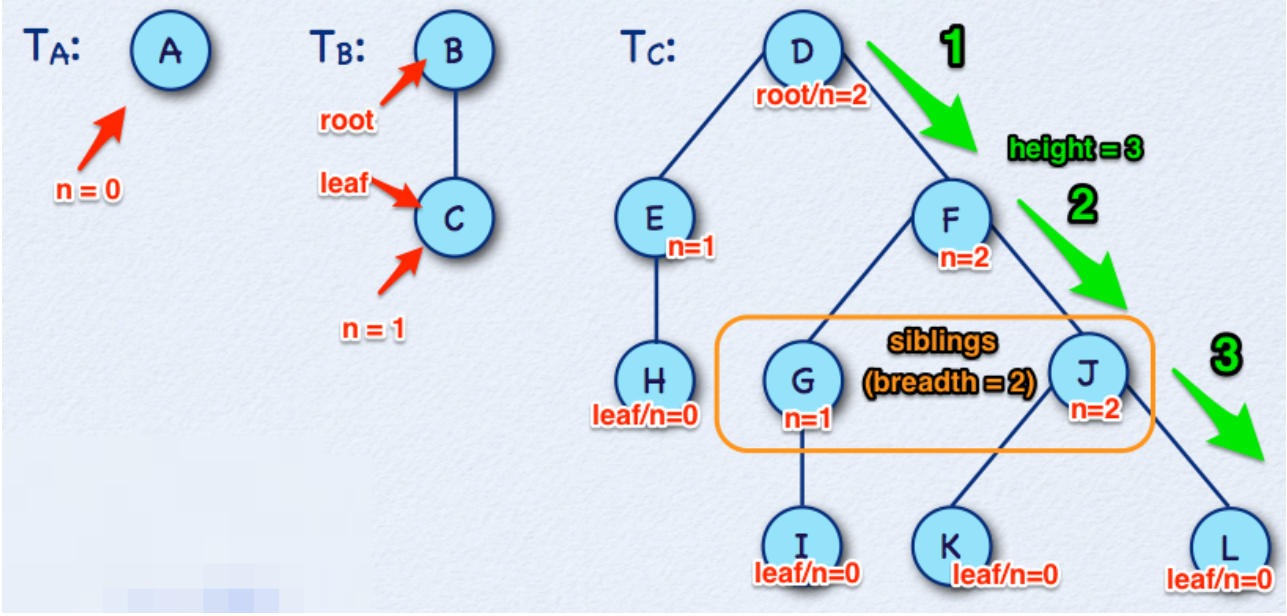
- We can use reference counting to count the number of pointers that are pointing to the heap-based object
  - When there is a **new pointer** pointing to the heap-based object, **add one reference**
  - When that pointer is destructed (i.e., **goes out of scope**), then **remove that reference**
- The Handle Class is a way to synchronise reference pointers between value-based pointers and reference-based objects
  - The synchronisation will occur by reference count
  - When the **reference count is zero** (i.e., no pointers pointing to the heap-based object) then **the heap-based object is destructed**
  - We need to synchronise the reference counter itself; this will also be on the heap too!

```
template <class T>
class Handle
{
private:

    T* fObj;   // A pointer to the Reference Based Object
  int* fCount; // The reference count to this object

  void retain()   // Adds a reference to fObj (hold onto fObj ref)
  {
     // Add one to the heap count of this obj
     ++(*fCount);
  }
  void release()  // Removes a reference to fObj (release fObj ref)
  {
     // If after removing a reference count we are @ 0 references?
     if (--(*fCount) == 0)
     {
       // Delete the count and the object (both heap based!)
       delete fObj;
       delete fCount;
     }
  }

public:

  // Default constructor will make fObj a reference to no object
  Handle( T* aObj = (T*)0 )
  {
     // Internal ptr will point to heap obj provided
     fObj = aObj;
     // A new int on the heap
     fCount = new int;
     // Make count 1 initially
     *fCount = 1;
  }

  // Copy constructor will appropriately duplicate the object
  Handle ( const Handle<T>& aOtherHandle )
  {
     // Simply switch over the pointers and add a reference to the
     // new pointer…
     this->fObj = aOtherHandle.fObj;
     this->fCount = aOtherHandle.fCount;
     // Retain this new reference
     retain();
  }

  // Destructor will remove the reference and delete if no more references
  ~Handle()
  {
     // When we're really destroying it, we're just removing reference…
     release();
  }

  // Assignment operator will appropriately duplicate the object
  Handle& operator=( Handle<T>& aOtherHandle )
  {
     // Suicide protection
     if (&aOtherHandle != this)
     {
        // I'll keep you, the new guy
        aOtherHandle.retain();
```

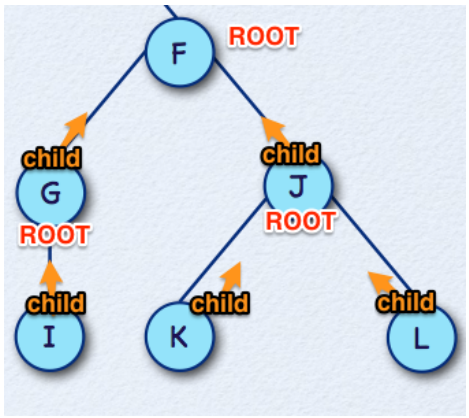
```
        // I'll say goodbye to you, the old guy
        this->release();

        // Now switch over pointers..
        this->fObj = aOtherHandle.fObj;
        this->fCount = aOtherHandle.fCount;
    }
    return *this;
}

// Deference the Handle to make interaction with fObj easy
T& operator*()
{
    // No object at the end?
    if (fObj == (T*)0)
      throw std::runtime_error("NULL pointer exception");

    // Return fObj on heap…
    return *fObj;
}

T* operator->()
{
    // No object at the end?
    if (fObj == (T*)0)
      throw std::runtime_error("NULL pointer exception");

    // Return the pointer itself…
    return fObj;
}
```

Trees

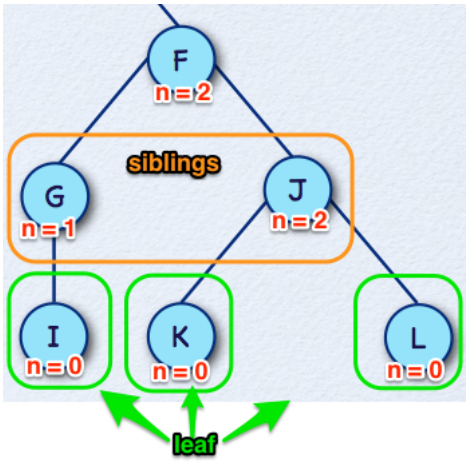
- A tree is a finite non-empty set of nodes **(a set of sets)** where
  - there is always a single, root node, r
  - the remaining nodes stem from r to n subtrees, where n >= 0



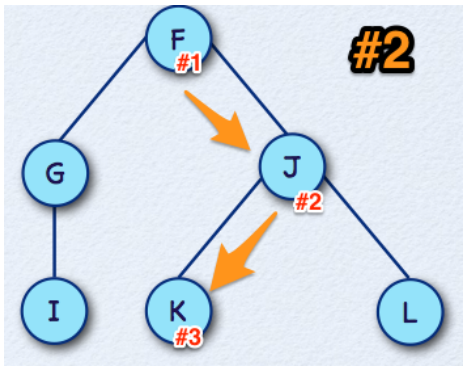
- **Parent vs Child**
  - root node r is a parent to all of its subtrees
  - each **root** subtree is a child of its parent
  - thus a child is also a root:
    - *G and J to the right:*
      - *G and J are **children to F***
      - *G is the **root to I***
      - *J is the **root to K and L***



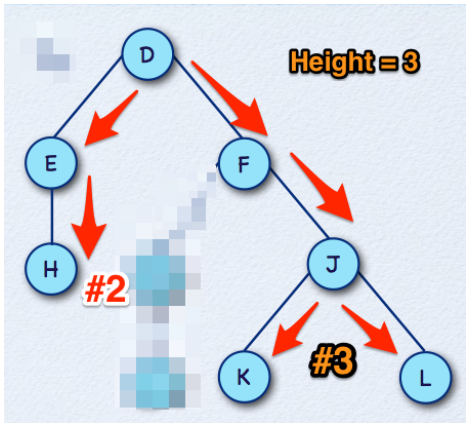
- **Degree**
  - The **degree of a node** indicates how many subtrees exist
  - Where degree is **zero**, then the node is a **leaf** of the tree
  - Where two subtrees share the same root, they are siblings



- **Path and Path Length**
  - A path is a sequence of non-empty nodes
  - It is the Traversal of all the nodes visited from any root node **down to a leaf**
  - Path length defines the number of linkages between the nodes visited
    - length is always #no nodes - 1
    - *To the right:*
      - *Path taken has 3 nodes*
      - *Path length is therefore 2 (2 links!)*

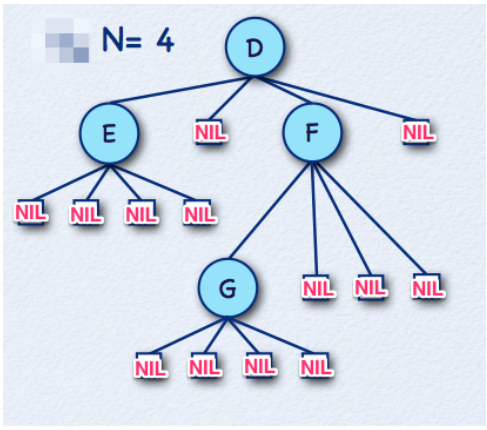


  - **Depth and Height**
    - Depth from any given root node to any **another node**
      - *The depth of G-I is 1*
      - *The depth of D-J is 2*
    - Height of a tree is the **longest possible path** in the tree from a node to its leaf
      - *D-E-H: #2 links*
      - *D-F-G-K and D-F-G-L: #3 links*
      - *Hence height of the three is 3*



N-Ary Trees

- N-Ary trees are trees where **all nodes have the degree**
  - Problem! Recursive structure *(I have 3 children, each child has 3 children ad infinitum)*
  - Therefore, needs some form of terminator
- The empty tree solves this
  - Empty trees are **sentinels** used to denote a node that is not part of the tree set
  - It is the representation of the empty set: ∅
    - In the C++ Specification, NIL will be the sentinel
    - NIL is a node that has N number of **children to itself—black hole**
  - We can fill in terminator sentinels to **terminate the recursive structure!**
    - *n=4 but D has 4 nodes: E and F, the rest **NIL…***
- N-Ary trees must have exactly n number of nodes where
  - the child node is a **NIL sentinel**
  - the child node is a node with **exactly n number** of children nodes



Type for Key
Degree

```
template<class T, int N>
class NTree
{
```

Key of the root node of the tree
Number of child nodes, set to degree N

```
private:
    const T* fKey;
    NTree<T,N>* fNodes[N];
```

PRIVATE constructor for NIL (clients cannot create NIL)

```
    NTree();
```

```
public:
    static NTree<T,N> NIL;
```

NIL sentinel declaration

```
    NTree( const T& aKey );
```

Create this tree's Root with aKey

```
    ~NTree();
```

To kill me and my children

```
    bool isEmpty() const;
    const T& key() const;
    NTree& operator[]( int aIndex ) const;
    void attachNTree( int aIndex, NTree<T,N>* aNTree );
    NTree* detachNTree( int aIndex );
```

Indexer for child node access

Attach/Detach Tree at aIndex

- **Private NIL Constructor**
  - We don't know what the Key is—it's NIL, and thus **is unknown**
  - NIL is a black hole, **meaning its children are itself**: each child is an address to itself

```
// Make NIL's fKey assigned to T* to 0 (i.e., unknown)
NTree() : fKey( (T*)0 )
{
    // Black hole; for every node slot to N degree, fill w/ address to NIL
    for ( int i = 0; i < N; i++ )
        fNodes[i] = &NIL;
}
```

- **Public Constructor**
  - Exactly the same with NIL in that initially **fill fNodes with NIL**
  - We will initialise it with aKey provided though, since that **we do know**

```
// Pass in aKey
NTree( const T& aKey ) : fKey( &aKey )
{
   // Initially, all children are NIL—but change in attach…
   for ( int i = 0; i < N; i++ )
       fNodes[i] = &NIL;
}
```

- **Destructor**
  - We only kill all of our non-NIL children
  - You cannot kill the empty set since this signifies that it's empty!!

```
~NTree()
{
   // For every node...
   for ( int i = 0; i < N; i++ )
      // ...that isn't a sentinel (cannot delete terminators!)...
      if ( fNodes[i] != &NIL )
          // ...delete the non-NIL node!
          delete fNodes[i];
}
```

- **Initialising static NIL**
  - All static instance variables need to be initialised outside the class definition
  - Since not providing this NTree (i.e., the NIL NTree) with a constructor, use Default Const.

```
// Uses default (private) constructor to make NIL
template <class T,int N>
NTree<T,N> NTree<T,N>::NIL;
```

- **Auxillaries**
  - Tree is empty if it is the empty set (i.e., NIL)
  - Therefore, compare against NIL and allow access to Key if not NIL (otherwise $\perp$)

```
bool isEmpty() const
{
   // Is this tree an empty set?
   return this == &NIL;
}

const T& key() const
{
   // If this tree is the empty set?
   if ( this->isEmpty() )
       throw std::domain_error("Cannot get key of the empty tree");
   // Otherwise, just return it
   return *fKey;
}
```

- **Attaching a New Subtree**
  - Injection of a new subtree can only occur where:
    - there is **room for the injection**
    - injecting **within the allowed domain**
    - not injecting **to NIL (empty set)**
  - That means, we cannot inject into a slot that **already has a tree there!**
  - Hence, must check non-NIL before injecting…
  - **Note:**
    - **Once injected, the tree now takes ownership of aNTree**
    - **This is why you pass in the pointer, and not just a reference!**

```
void attachNTree( unsigned int aIndex, NTree<T,N>* aNTree )
{
    // Cannot inject to NIL
    if ( this->isEmpty() )
       throw std::domain_error("Cannot inject into the empty tree!");

    // Cannot inject outside domain (remember 0-based array!)
    if (aIndex >= N)
       throw std::out_of_range("Cannot inject outside domain of tree!");

    // Cannot override a non-NIL slot
    if (fNodes[aIndex] != &NIL)
       throw std::domain_error("Can only inject into empty slots!);

    // Otherwise, all good!
    fNodes[aIndex] = aNTree;
}
```

- **Accessing the Tree**
  - You can access, but not modify, the internal tree
  - This is because the tree is still owned by the client
  - You will only get a **reference** back to the tree node accessed
  - Ensure index is **within range of the domain of the tree**
  - Ensure you're **not trying to access NIL tree!**

```
// Returned tree is still owned by root tree; you can't modify it!!!
const NTree& operator[]( unsigned int aIndex ) const
{
   // Cannot access a NIL node
   if ( this->isEmpty() )
       throw std::domain_error("Cannot access any empty tree nodes");

   // Cannot access outside domain (remember 0-based array!)
   if (aIndex >= N)
       throw std::out_of_range("Cannot access outside domain of tree!");

   // Otherwise, all good! Return readonly reference to the tree
   return *fNodes[aIndex];
}
```

- **Removing an existing Subtree**
  - Removal of a new subtree can only occur where:
    - Removal **within the allowed domain**
    - not removing from **NIL (empty set)**
  - Once removed, the node will be NILified
  - Client now **owns the node**
    - Hence, return type is the pointer of the removed node.

```
// You get the entire tree back and get to own it
NTree* detachNTree( unsigned int aIndex )
{
    // Cannot remove from NIL
    if ( this->isEmpty() )
       throw std::domain_error("Cannot remove from the empty tree!");

    // Cannot remove outside domain (remember 0-based array!)
    if (aIndex >= N)
       throw std::out_of_range("Cannot remove outside domain of tree!");

    // Take out the node (will return it)... from this index
    NTree<T,N>* lRetVal = fNodes[aIndex];

    // This node is now NIL'ified
    fNodes[aIndex] = &NIL;

    // Give back the node taken out
    return lRetVal;
}
```

Binary Trees

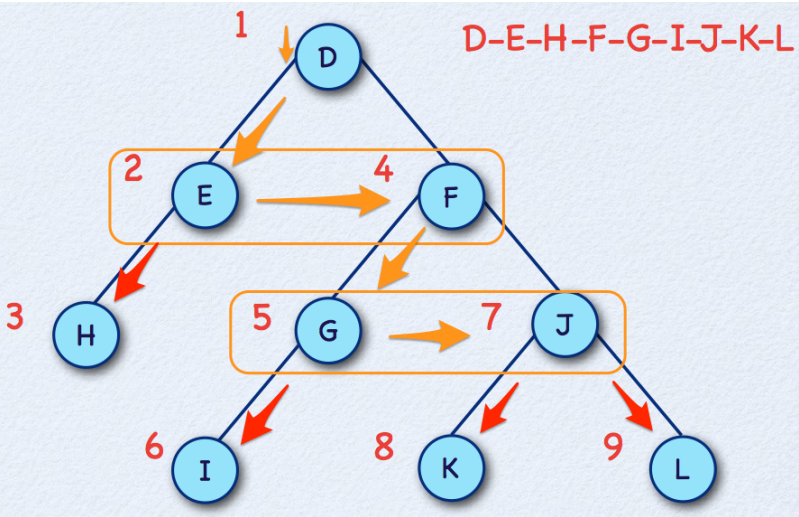
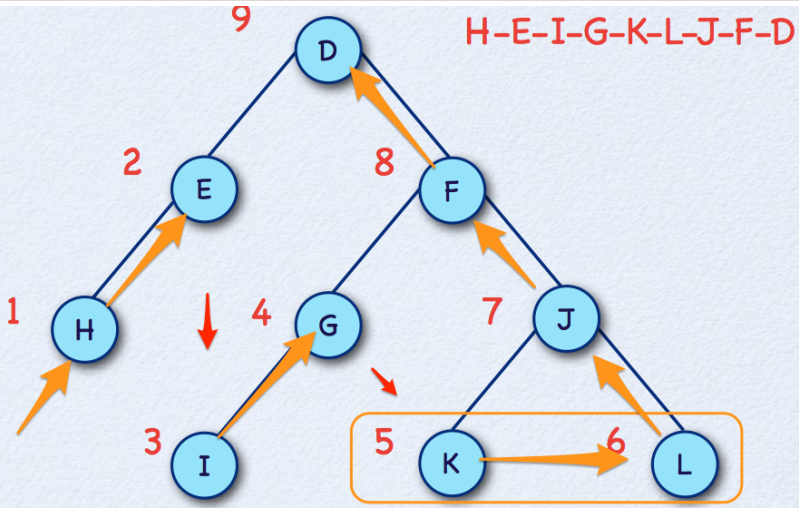
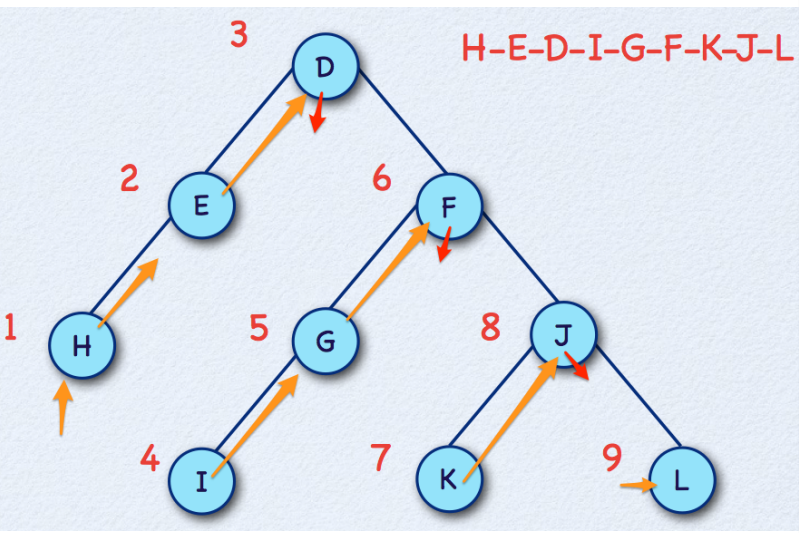
- Binary Trees are structurally just N-Aray trees where N = 2
- However, we improve the **semantic value** of the ADT by explicating treating the interface as if it is a binary tree—naming the nodes Left and Right etc.
- Therefore, this gives us a more distinguished meaning to just an N-ary node with 2 nodes
  - We have a left node
  - We have a right node

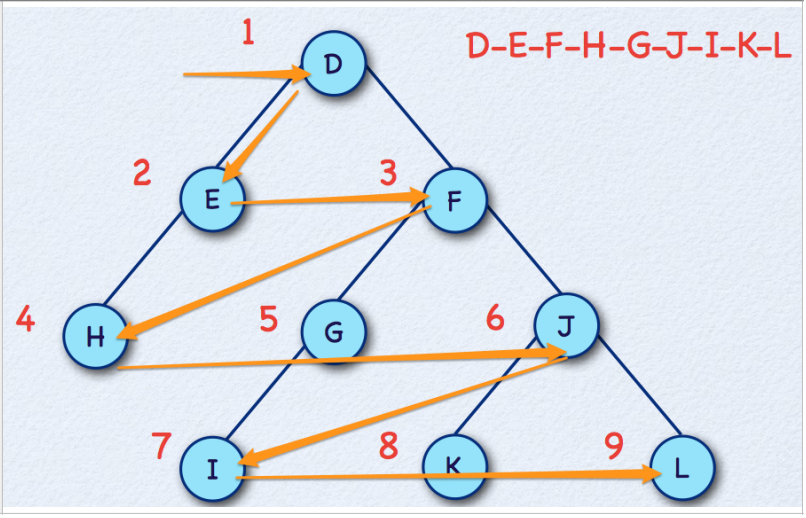
- As there are just two nodes, there is really no need for an array of nodes
  - Simply a fLeft and an fRight nodes will just suffice.
- This will mean more specialised semantics: left(), right(), attachLeft(), detachRight() etc.

Tree Traversal

- Trees allow us to make sense of data
  - We can **encode relationships between data using trees**
  - Infer ore about what the elements contain
  - There is **meaning** behind their position in the tree
- Systematic method of visiting all the nodes in the tree and follow their relationship
  - **Uniformly process** all the nodes in a tree
  - **Locating** a **particular node** in the tree
- Two methods for 'visiting' all nodes in a tree
  - **Depth-first Traversal (Vertical Approach)**
    - This means that we will always go **deep into children** of the tree first
    - Go as far possible down a particular path to its leaf
      - Walk down every path
      - When we hit a leaf, backtrack up and repeat for next sibling…
    - We can use a Stack to implement this (i.e., add all children to stack and keep popping until no more children)
  - **Breath-first Traversal (Horizontal Approach)**
    - This means that we will always go **across siblings (generational search)** of the tree
    - Go as far across the breadth of the siblings first
      - Start with LHS child, then goto all children in 2nd generation
    - We can use a queue to implement this (i.e., queue up all children of current node)

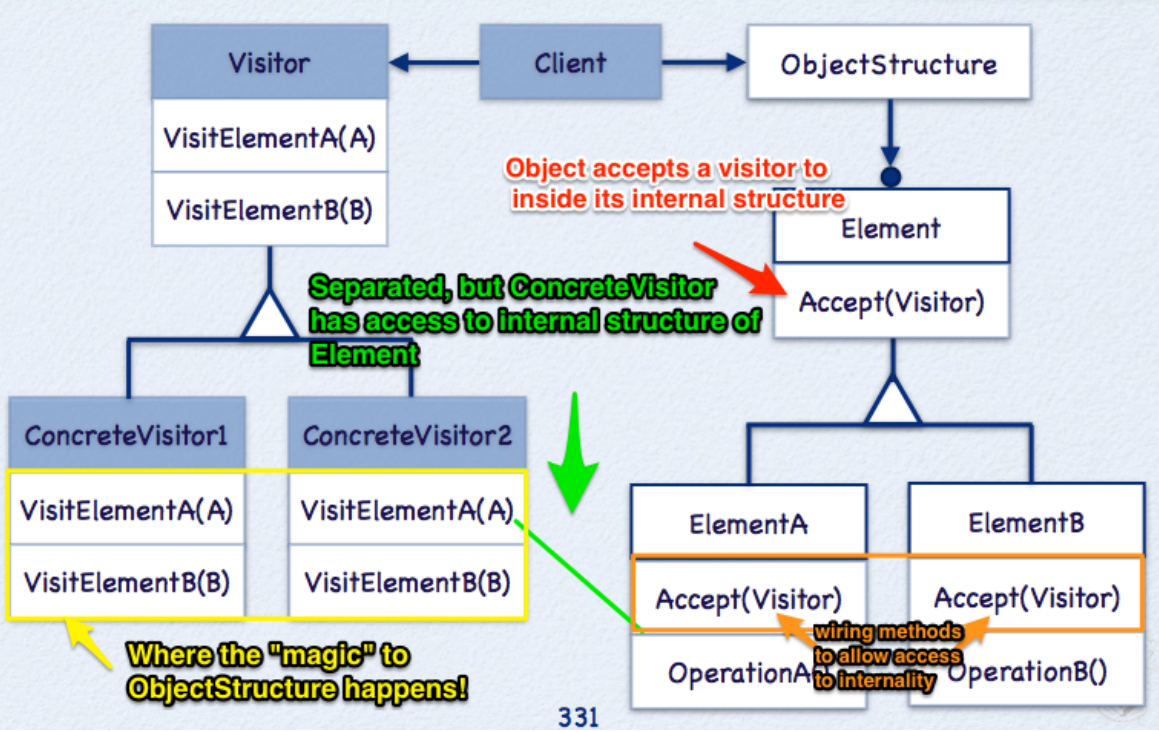
Traversal Methods

Method	
Pre-Order Traversal 1. Start at the root node 2. Go to the LHS child node 3. Continue for all child nodes to leaf 4. **Backtrack** to topmost root node 5. Move to RHS sibling and repeat	
Post—Order Traversal 1. Start at LHS leaf node 2. Move to next sibling leaf 3. Backtrack to root node at end 4. Move to RHS child 5. Repeat until at root	
In-Order Traversal 1. Traverse from the LHS leaf 2. Keep going up until root 3. Repeat for sibling RHS leaf *FUN FACT:* *Goes from smallest to largest node* *This is because H < E, D < E etc*	

Method	
Breadth-First 1. Start from root 2. Move across siblings 3. Once at last sibling of same degree go to first child of first sibling 4. Repeat	

Visitor Design Pattern

- Allows us to inspect **internal object structure** in a **non-invasive** fashion
- A surgical tool to add additional behaviour to the class
  - non-invasive since we add it from **outside** the class structure



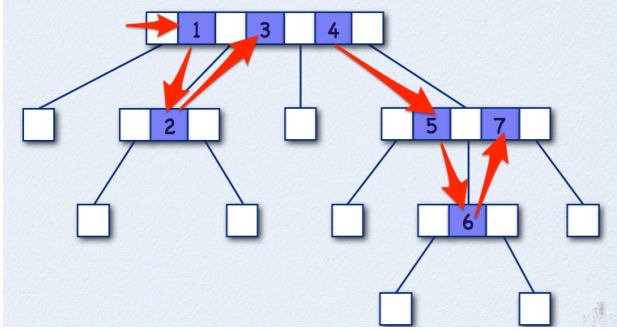
- In the accept "wiring" method, the object typically passes it's key private attributes to VisitElementA(A)
  - This therefore allows ConcreteVisitors to have access to internal structure of ObjectStructure
  - Overcomes **encapsulation**!

Tree Traversal Visitors: Depth-First

- Begin with the **Base, Generic Visitor Class** (i.e., non-ConcreteVisitor):

```
template<class T>
class BaseTreeVisitor
{
public:
    // Since ConcreteVisitors will inherit from me, need virtual destructor
    virtual ~BaseTreeVisitor() {}

    // Default behaviour of my children will implement one of the following
    virtual void preVisit  ( const T& aKey ) const {}
    virtual void postVisit ( const T& aKey ) const {}
    virtual void inVisit   ( const T& aKey ) const {}

    // What each child will do, respectively
    virtual void visit( const T& aKey ) const
    {
        std::cout << aKey << " ";
    }
}
```

- At present, you can see that preVisit, postVisit and inVisit (for each of the **pre-order, post-order and in-order** Traversal methods) do **nothing** in the TreeVisitor
  - This is because we will allow **potential** functionality within our children
    - We now have **3 subclasses** that **respectively respond** to each method



  - But our children will override each respectively to make them usable:
    - Preorder:

```
template<class T>
class PreOrderVisitor : public BaseTreeVisitor<T>
{
public:
    // Override preVisit for the PreOrderVisitor iterator to call base
    // visit method; now we have implemented that functionality within
    // this concrete visitor!
    virtual void preVisit( const T& aKey ) const
    {
       visit( aKey );
    }
}
```

    - And the same respectively for PostOrderVisitor and InOrderVisitor

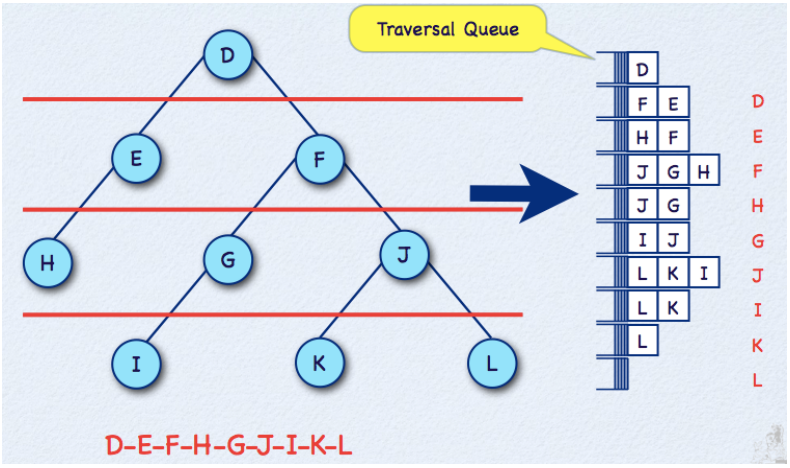
- Now in our BINARY tree, we will pass in a BaseTreeVisitor, and thanks to polymorphism, that can be either one of PreOrderVisitor, PostOrderVisitor and InOrderVisitor:

```
void traverseDepthFirst( const BaseTreeVisitor<T>& aVisitor ) const
{
    // Cannot traverse through the empty tree—recursive method needs
    // a terminator clause!
    if (!isEmpty())
    {
        // If aVisitor supports preVisit (i.e., aVisitor is an instance of
        // a PreOrderVisitor) then it will print to console the key
        aVisitor.preVisit( key() );

        // Use recursion to traverse through the left subtree:
        left().traverse( aVisitor );

        // If aVisitor supports inVisit (i.e., aVisitor is an instance of
        // a InOrderVisitor) then it will print to console the key
        aVisitor.inVisit( key() );

        // Use recursion to traverse through the right subtree:
        right().traverse( aVisitor );

        // If aVisitor supports postVisit (i.e., aVisitor is an instance of
        // a PostOrderVisitor) then it will print to console the key
        aVisitor.postVisit( key() );
    }
}
```

- The BINARY tree will now support traversal of each dept-first kind
  - We don't know which subtype aVisitor is (remember we passed in BaseTreeVisitor)
  - The visitor itself controls the behaviour of depth first traversal; **class is just wiring it up!**

Tree Traversal Visitors: Breadth-First

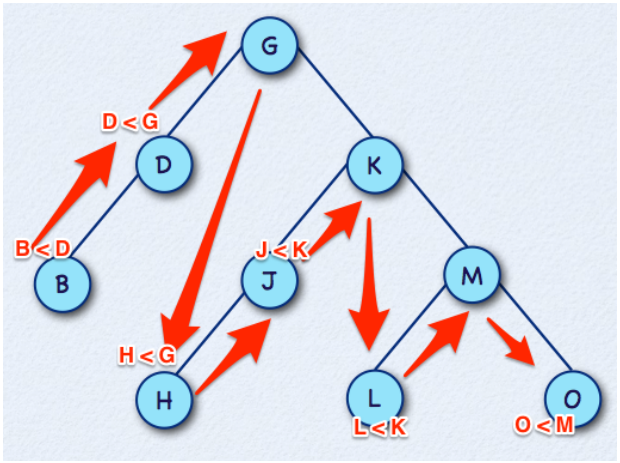
- Rather than using recursion, we can use a **queue** instead
- The queue will simply enqueue the current root tree, and then each of its subtrees
- That way, we pass through breadth first; first-in (siblings), first-out (siblings); children enqueued and we come to *them* later




```
// Since we're not using post, pre or in order visitors, we can just
// always use the base visitor instead!

void traverseBreadthFirst( const BaseTreeVisitor<T>& aVisitor ) const
{
    // A queue of all the trees we're going to traverse over
    Queue< BTree<T> > lTraversalQueue;

    // Given that this tree is not empty
    if ( !isEmpty() )
        // I will let aVisitor traverse over it
        lTraversalQueue.enqueue( *this );

    // Keep on traversing until the queue is empty
    while ( !lTraversalQueue.isEmpty() )
    {
        // Take off the topmost tree from the traversal queue
        const BTree<T> head = lTraversalQueue.dequeue();

        // Now visit it's key (i.e., make aVisitor do its job!)
        // given it wasn't empty
        if ( !head.isEmpty() )
            aVisitor.visit( head.key() );

        // Queue up each of my subtrees so that they can come in after
        // I've finished with this generation—given they are not empty too
        // (i.e., add children to next search...)
        if ( !head.left().isEmpty()  )
            aVisitor.enqueue( head.left() );

        if ( !head.right().isEmpty() )
           aVisitor.enqueue( head.right() );
    }
}
```

M-Way Search Tree

- An M-way search tree is a set of finite nodes where:
  - 1) either the set is empty (i.e., T = ∅) or
  - 2) each key is ordered by moving left and right by child
    - Each key is unique
    - All keys in the righthand subtree are greater than those in the lefthand subtree
    - All keys in the lefthand subtree are less than those in the righthand subtree



- A **2-Way search tree** also exists, wherein there are only 2 subtrees
  - 1) We traverse the left subtree
  - 2) Visit the root
  - 3) We traverse the right subtree

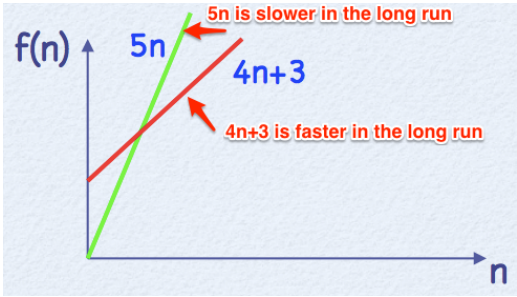
Algorithmic Patterns

- What is the **best algorithm?**
  - Structure, composition and readability (i.e., ease of the interface with the algorithm)
  - Time to implement
  - Extensibility of the algorithm (e.g., Node to List to Queue to Tree)
  - Space and time requirements
- Time Analysis depends on:
  - The **test input** (i.e., A(n) is faster than B(n) where n < 10; n >= 10 it is otherwise!)
    - How does the input size differ the output time?
    - What is the margin?
    - What is the relationship?
      - T is **constant** for all n
      - T is **linear** where a n+1 causes T+1
      - T is **exponential** where n + 1 causes T exponential increase
  - **Interruption** of execution environment (process interruption)

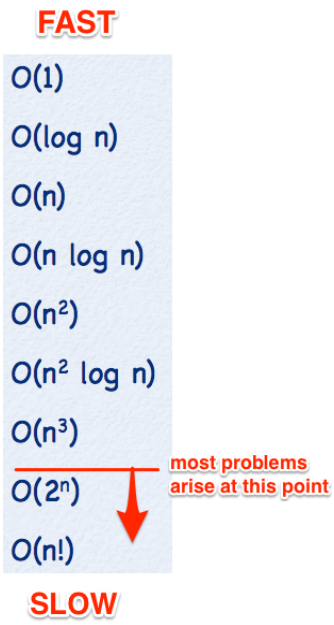
- **Execution** environment itself (Machine A vs. Machine B)

- **Computable:** Any input which is processed to produce some output in a sequence of step
- ***Effectively*** **computable:** humans are finite—we want the solution to be solved in:
  - **a finite time** with
  - **finite resources**
  - Susceptible to:
    - Halting Problem: We don't know if its still processing or its its broken
      - *Do I control-C? Or should I keep faith that it's still processing?*
      - This problem is uncomputable—if it processes forever then it may not be **effectively computable!**
      - *e.g., Ackermann Function for m, n > 4… would take EONS to process!*

The Big-Oh — *O*(*f*(*n*))



- Focuses on the **growth rate of running time** of input size that approaches infinity
- Does not focus about running time for small input… **only large input**
- **Performance Analysis:**
  - Best-Case
    - Solution is the first element in the array, therefore O(1)
  - Worst-Case
    - Solution is the last element in the array, therefore O(n) for n elements in the array
  - Average-Case*
    - Takes on averages n/2…
- Varying O values:
  - **O(1) Constant Time:** the algorithm will always take 1000 steps regardless of size
  - **O(n^k) Polynomial Time**
  - **O(log n) Logarithmic Time**—start fast, slow down (i.e., ROC is fast initially so takes longer initially, then ROC slows down so takes slower later)
  - **O(2^n) Exponential Time**—ROC start slow, speed up; may only be **effectively computable** for certain range of n



- **For Loops**
  - For a for loop with a running time of C inside the loop the for loop has a running time of at most Cn = O(n)
  - This means that **for loops have a linear running time**

- The more data introduced to the loop, the longer it processes, linearly

- **Nested For Loops**
  - A nested for loop with k nested loops will have a **polynomial running time by factor k**

```
for ( … )
    for ( … )
        // C
```

  - The doubly-nested for loop (k = **2**) means that running time is **$O(n^2)$**
    - Why? Process all of n in for loop A, then process all of n in for loop b
    - n process in a * n process in b = n * n = n^2
- **Consecutive Statements**
  - Equals the sum of all statements
  - At most, the running time will be the **maximum running time of the n'th statement:**

```
for (…)           // O(n^1)
   // C

for (…)           // O(n^3)
    for (…)
        for (…)
             // C
```

  - Hence, in the above example, max(1, 3) is 3; the O is $O(n^3)$
- **If-Then-Else Branch**
  - Equals running time of if statement and the larger of the branches
    - Always better to overestimate than underestimate (i.e., assume for the worst)

```
if ( foo () )      // O(n^2)
{
   bar()           // O(n^2)
}
else
{
   qux()           // O(n^5)
}
```

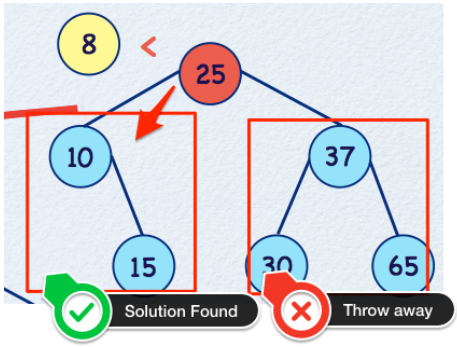
  - **$O(n^5) + O(n^2) = O(n^7)$**

Algorithmic Patterns

- Direct Solution Strategies
  - *E.g. Brute Force*
  - See the problem as sequence of decisions to be made
  - **Exhaustively enumerate** through **every single possibility** of the decisions
- Greedy Algorithms

- Like Direct Solution approach, but does not explorer all possibilities
- Targeted for a specific attribute

- Backtracking Strategies
  - Looks at all possible **outcomes** for each decision made
  - Checks if the solution is feasible
    - If it isn't it goes back (backtracks) up to where the decision way made
    - Walks back to **cross-road** and takes the **alternative approach** to solve the solution
- Top-Down Solution Strategies
  - *E.g. Divide and Conquer*
  - Partition the problem set up
  - Once partitioned, not all possibilities need to be explored
  - If we find the solution in one partition first, we don't even need to look at the other partition at all



- Bottom-Up Solution Strategies
  - *Dynamic Programming*
  - Solve the solution by computing the current solution to get to the next solution
  - **i.e., use one of the subproblems that have already been solved**
  - *E.g., Fibonacci:*

```
for (i = 1; i <= n; i++)
{
    next = curr + prev;
    pref = curr;
    curr = next;
}
```

  - Next was assigned to the previous solution plus the current problem…
- Randomised Strategies
  - Select elements in a random order to solve a problem
  - **Eventually,** all problems are solved
    - Results can be obtained faster or slower than last time
  - *E.g., Sorting a Deck of Cards by randomisation*
    - O(1/52!) — this is not effective at all in the event where O is reached
    - However, in the off-chance, we may shuffle to an already ordered deck, therefore **no wasted time at all**
    - There is a chance that we shuffle and **get an order we got before**—downside to sorting