

# Welcome to Algorithms and Data Structures! - CS2100

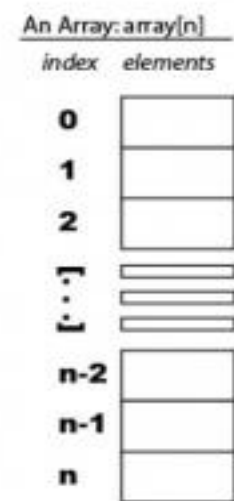
# Iteradores

# ¿Qué es un iterador?

Cada estructura tiene una manera particular de organizar los datos

Por tanto una manera diferente de recorrerse

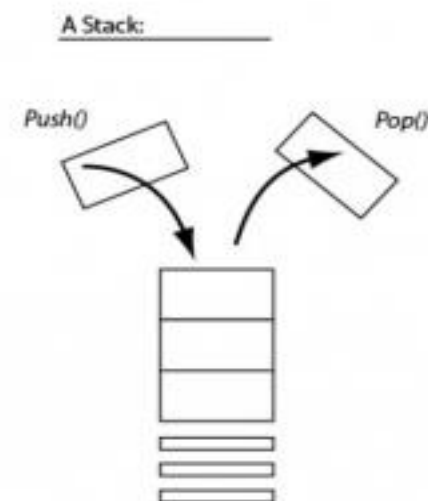
Usar iteradores nos lleva a dejar los contenedores independientes, por tanto no hacemos asumpciones del acceso



Typical features:  
indexing  
length/size  
copying

Good For:  
storing a fixed number of things  
and doing something to every one  
of those things

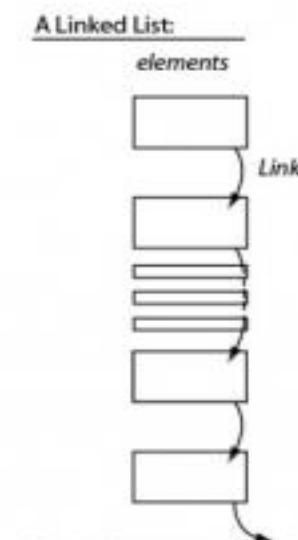
Bad For:  
Adding or removing elements  
Sorting things, in many cases  
Object Oriented thinking



Typical features:  
pushing  
popping  
size

Good For:  
dealing with a flow of things  
that need to be handled in certain  
groups.

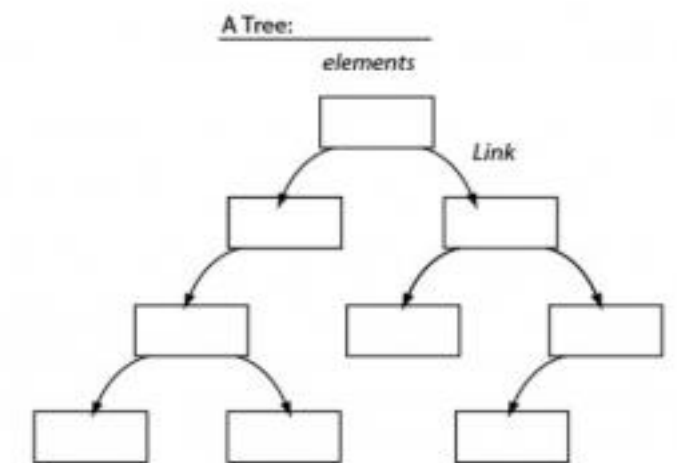
Bad For:  
Long term storage and access  
of complex data sets — just as  
piling things on your floor is not a great  
filing system



Typical features:  
get "next" element  
insert element  
remove element

Good For:  
dealing with a dynamically changing  
list — i.e. where you may need to insert  
or remove elements from anywhere  
in the list

Bad For:  
Well, it's still just a list. Almost always  
better than an array



Typical features:  
get "children"  
insert child  
remove element

Good For:  
storing things that belong in trees  
creating rapidly search-able data  
sets (i.e. decision trees)

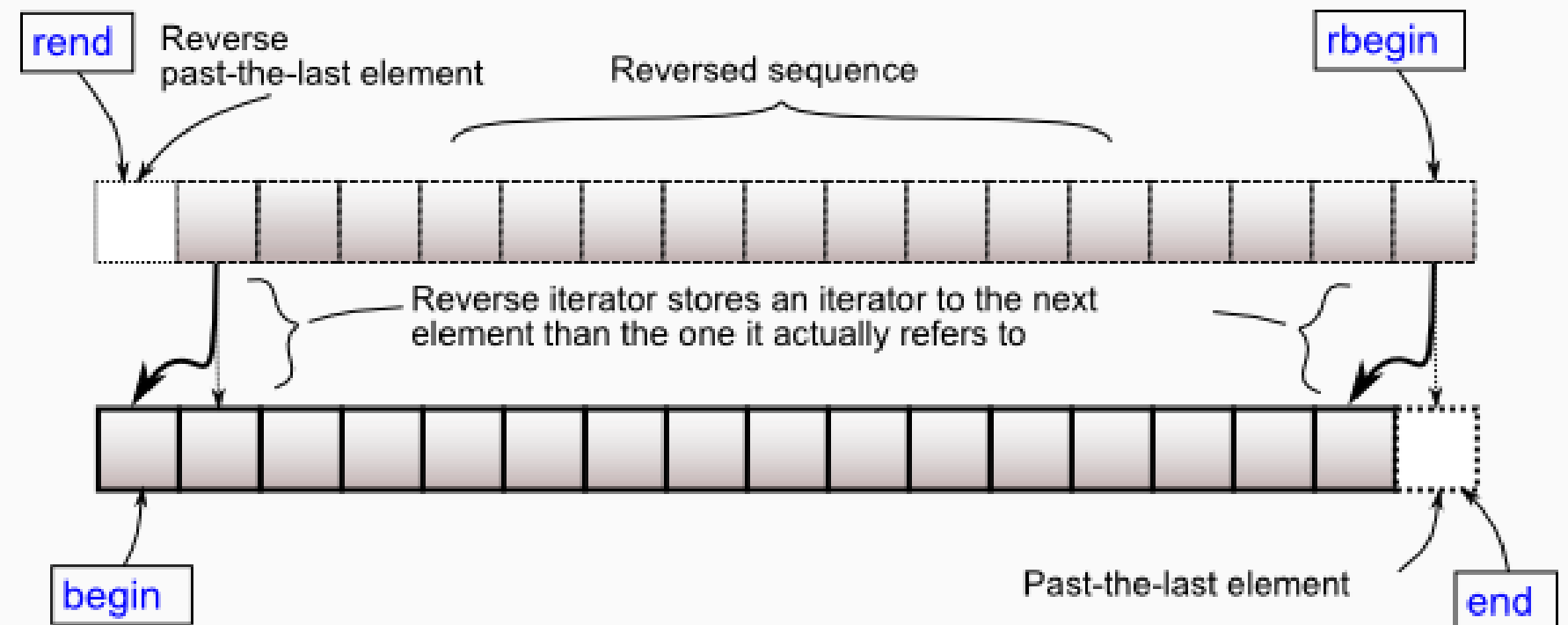
Bad For:  
Well, it's still just a list. Almost always  
better than an array

# ¿Qué es un iterador?

Es un objeto que representa un stream de elementos

Abstrae la manera en la cual iteramos sobre una estructura

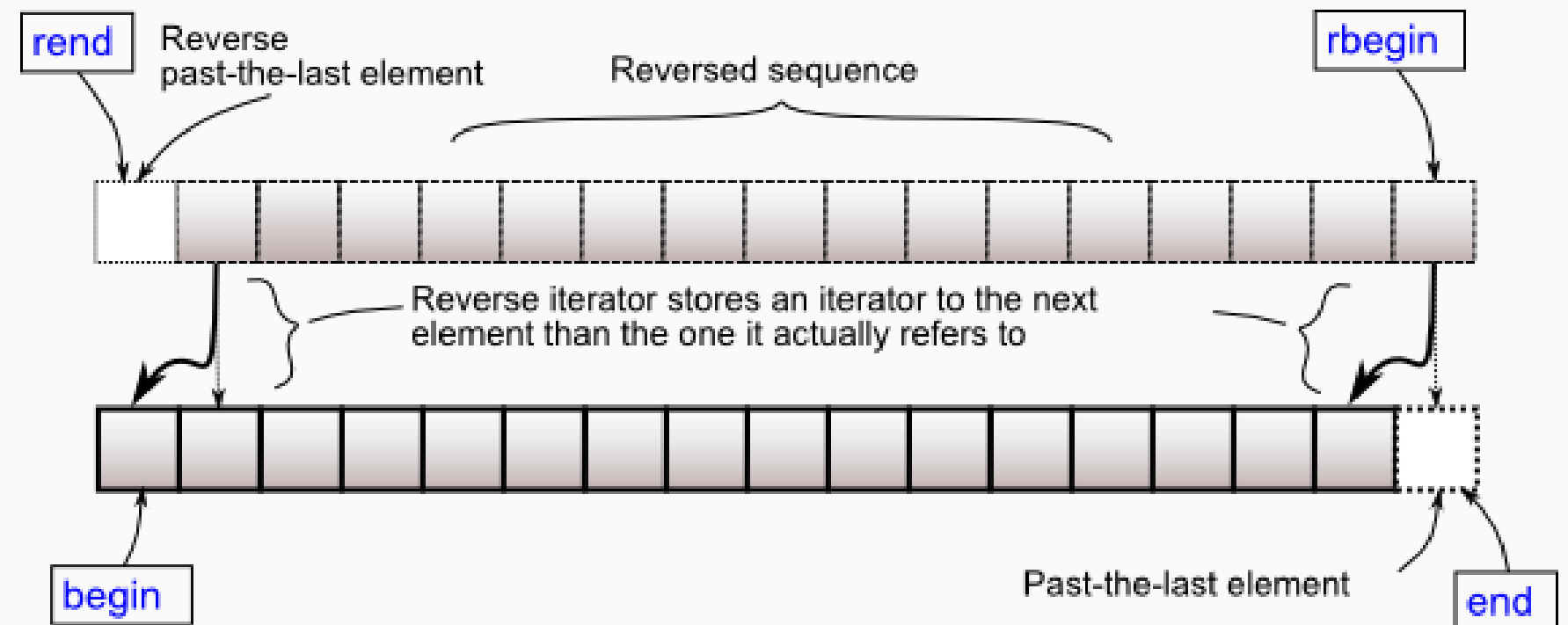
Reduce la complejidad de los programas



# ¿Qué es un iterador?

Algunas estructuras tienen diferentes tipos de iteradores para recorrerlas de diferentes maneras

Las funciones básicas de un iterador sirven para moverse entre elementos y obtener los valores almacenados



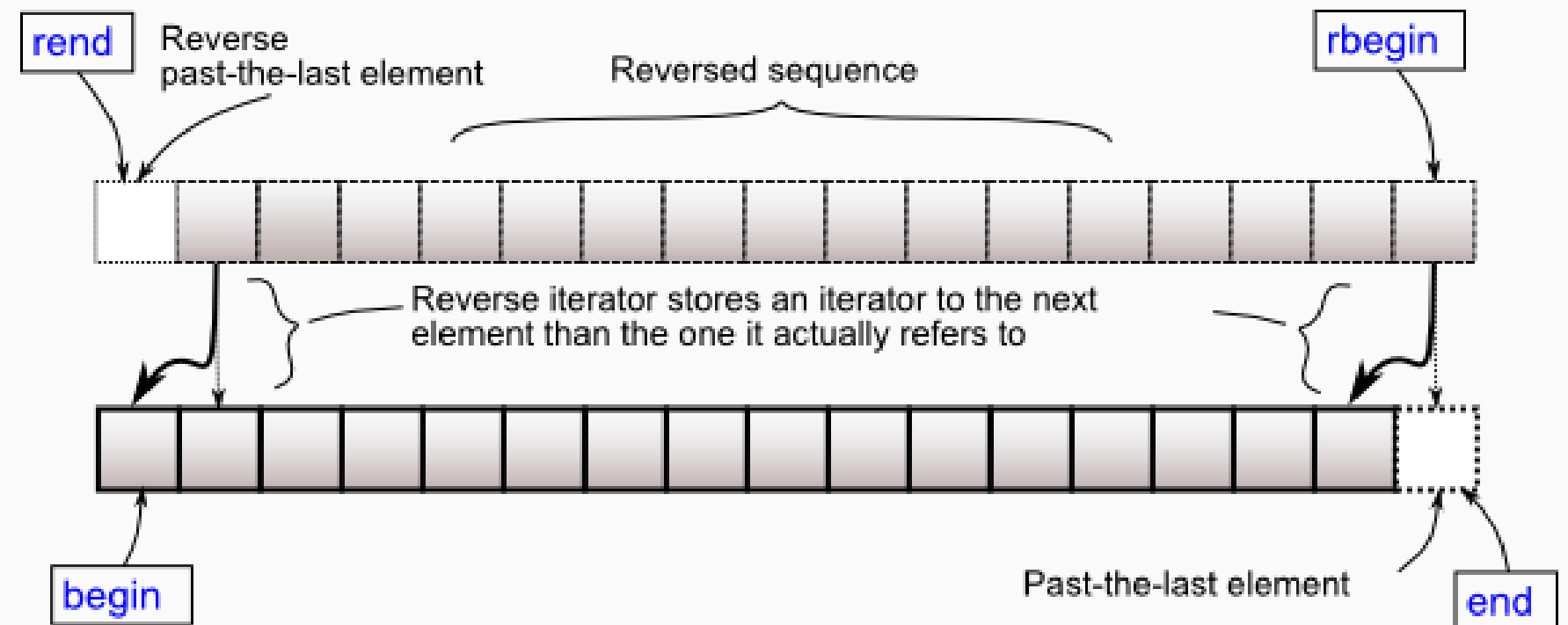
# ¿Qué es un iterador?

**Iterador constante**, provee solo acceso de lectura a los elementos.

**Iterador implícito**, proveen una manera de iterar sin tener que declarar al iterador.

```
for (int x : array) {  
    cout << x << endl;  
}
```

Cuáles serían las funciones básicas de un iterador?



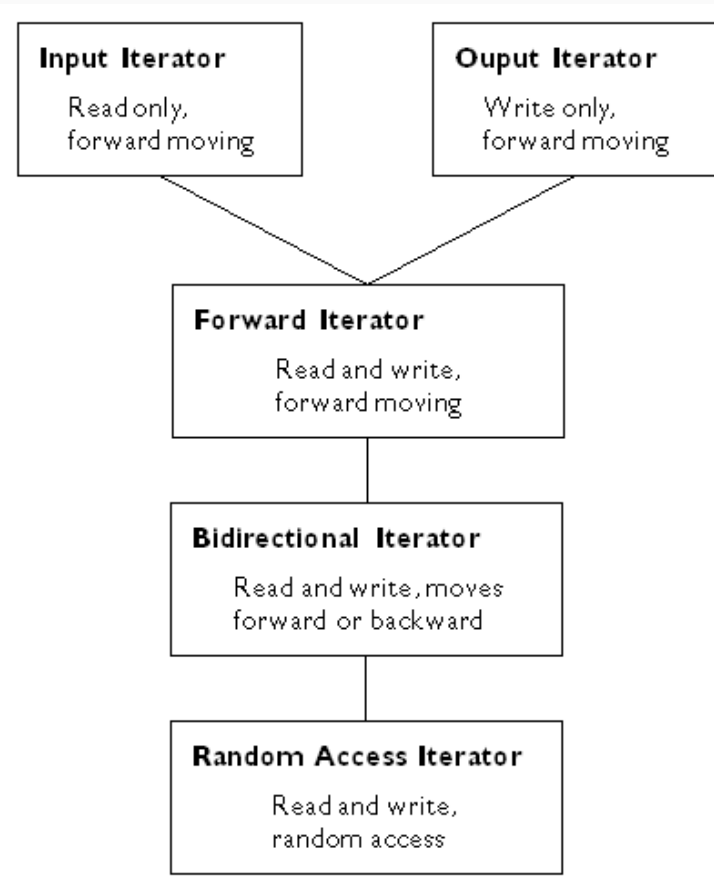
# ¿Qué es un iterador?

1. **begin()**: Return the **beginning position** of the container.
2. **end()**: Return the **after end position** of the container.
3. **operator++()**: Returns the iterator **after advancing a position**.
4. **operator--()**: Returns the new iterator **after decrementing a positions**.
5. **operator\*()**: Returns the content of the iterator **current position**.
6. **operator!=()**: Returns **true** if two iterators are **different**.
7. **operator=()**: **Assign** one iterator to another.

Qué otras operaciones se les ocurre que debería tener un iterador?

# ¿Qué es un iterador?

¿Todas las estructuras soportan iteradores?



CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported



# Ejemplos

- Forward

```
ForwardIterator<int> it;  
++it; // Usually faster than it++
```

- Bidirectional

```
BidirectionalIterator<int> it;  
++it;  
--it;
```

- RandomAccess

```
Iterator<int> it;  
it = it + 4;  
it = it - 2;  
++it;  
--it;
```

- Input/Output

```
int value = *it;  
*it = 5;
```

# Ejemplos

```
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;

int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };

    // Declaring iterator to a vector
    vector<int>::iterator ptr;

    // Displaying vector elements using begin() and end()
    cout << "The vector elements are : ";
    for (ptr = ar.begin(); ptr < ar.end(); ptr++)
        cout << *ptr << " ";

    return 0;
}
```

# Ejemplos

```
template<typename Iterator>
void my_algorithm(Iterator begin, Iterator end) {
    while(begin != end) {
        std::cout << *begin << '\n';
        ++begin
    }
}
```

El uso de iteradores te lleva cerca al concepto de independencia. No se asume nada acerca de la habilidad de acceso de las estructuras, sino que el contenedor te puede generar un iterador para ser recorrido.

Self Organizing List

Para búsquedas frecuentes

# Self Organizing List

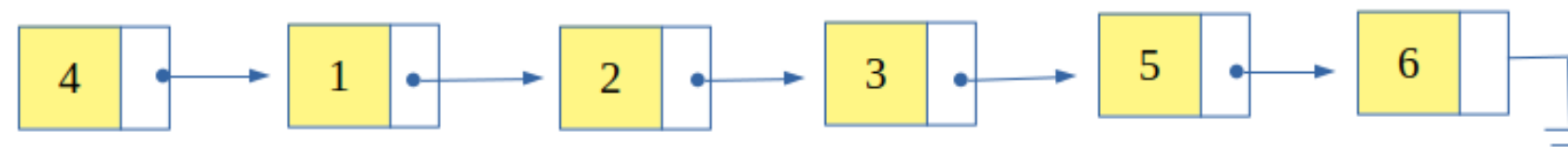
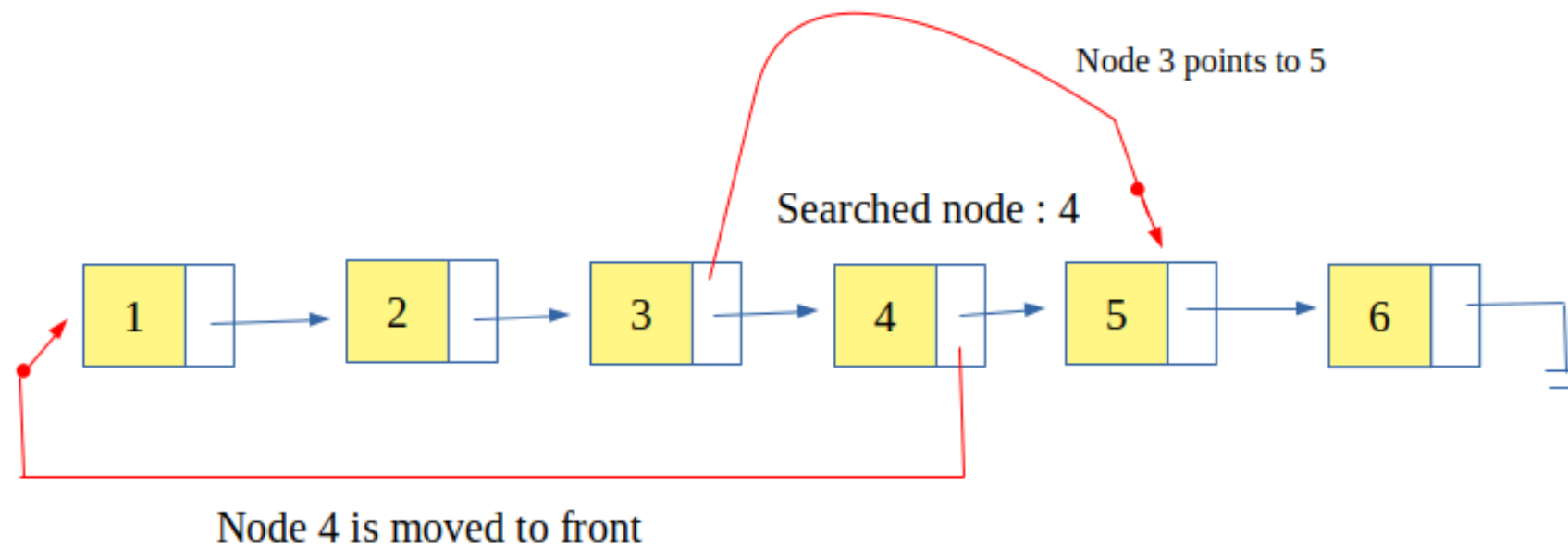
Es un tipo de lista que se va organizando mientras se solicitan elementos

Su peor caso es  $O(n)$  en la búsqueda, cuando el elemento es el último

Se reordena basada en 3 estrategias:

- Move-to-Front Method
- Transpose Method
- Count Method

# Move to front



# Move to front

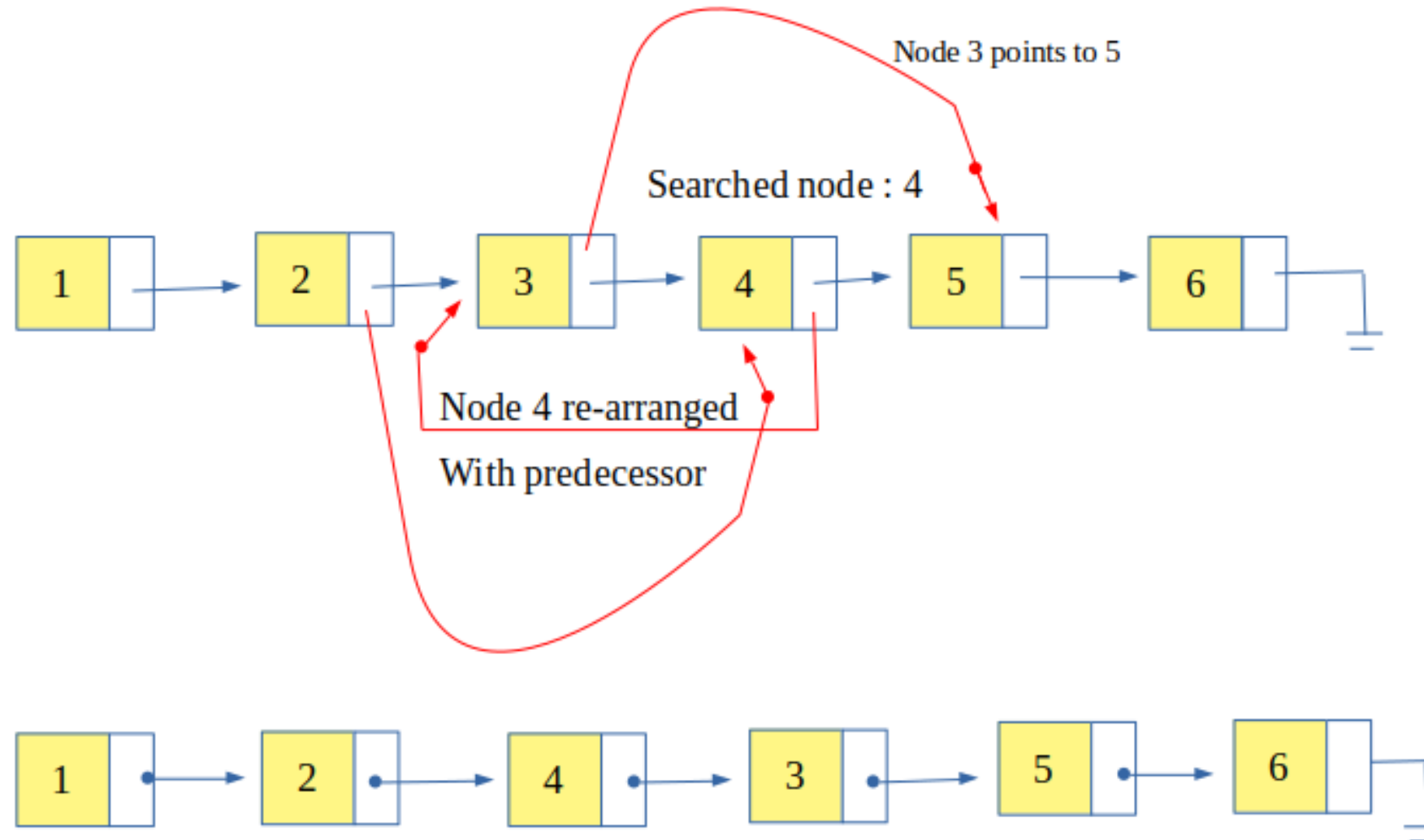
- Pro:

Este método se adapta rápido y es fácil de implementar

- Cons:

Puede mover al frente (prioriza) elementos que sean poco accedidos

# Transpose method





# Transpose method

- Pro:

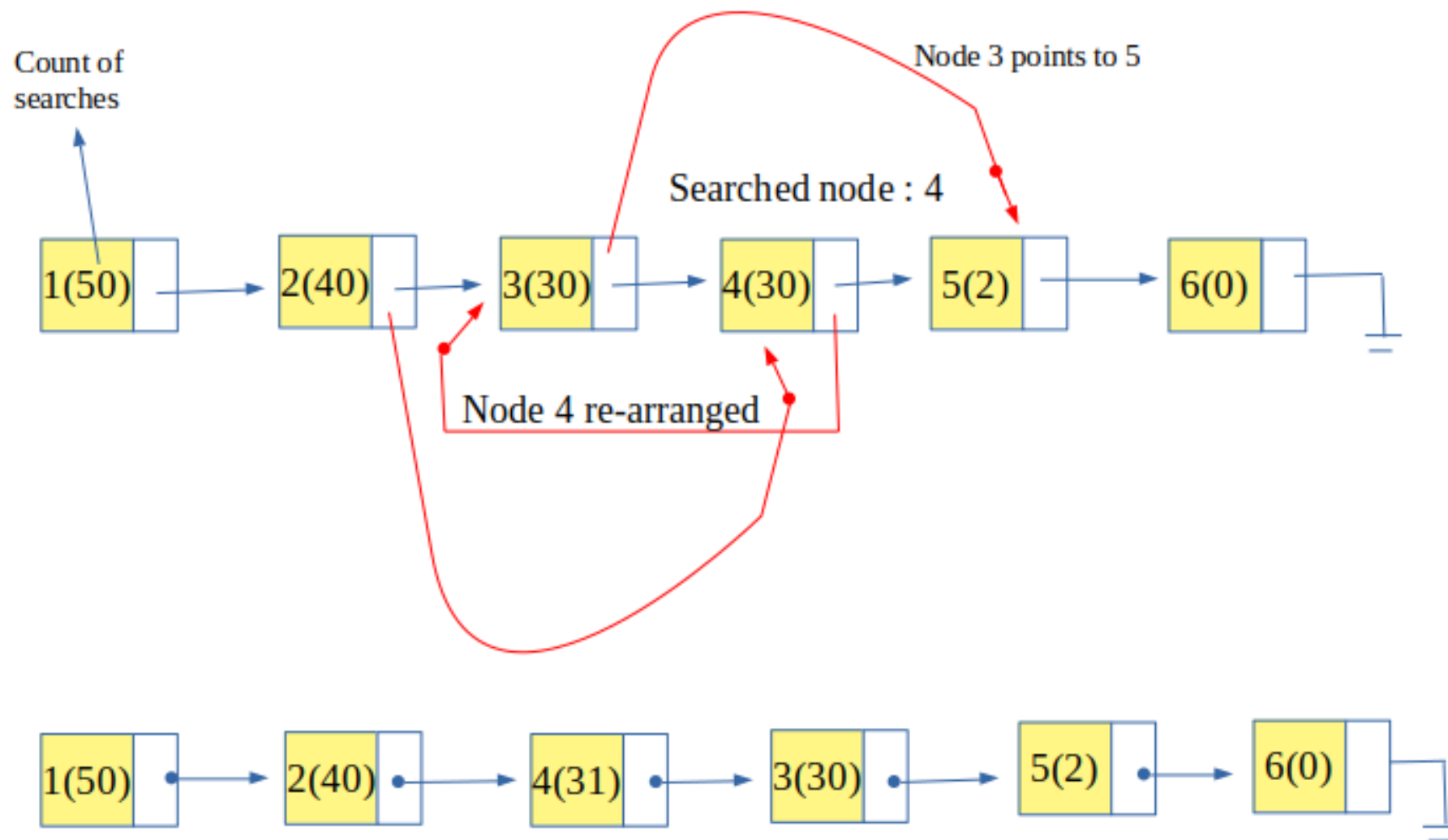
Fácil de implementar y requiere poca memoria

Mantiene nodos accedidos de manera frecuente al frente

- Cons:

Requiere muchos accesos para enviar un elemento al frente

# Count method



# Count method

- Pro:

Refleja el acceso a elementos de manera más real

- Cons:

Requiere un poco más de memoria, y no se adapta rápido a nuevos patrones de acceso

# Aplicaciones

- Inteligencia artificial
  - Redes neuronales
- Cache en algoritmos Last Frequently Used (LFU)
- Compiladores e interpretadores

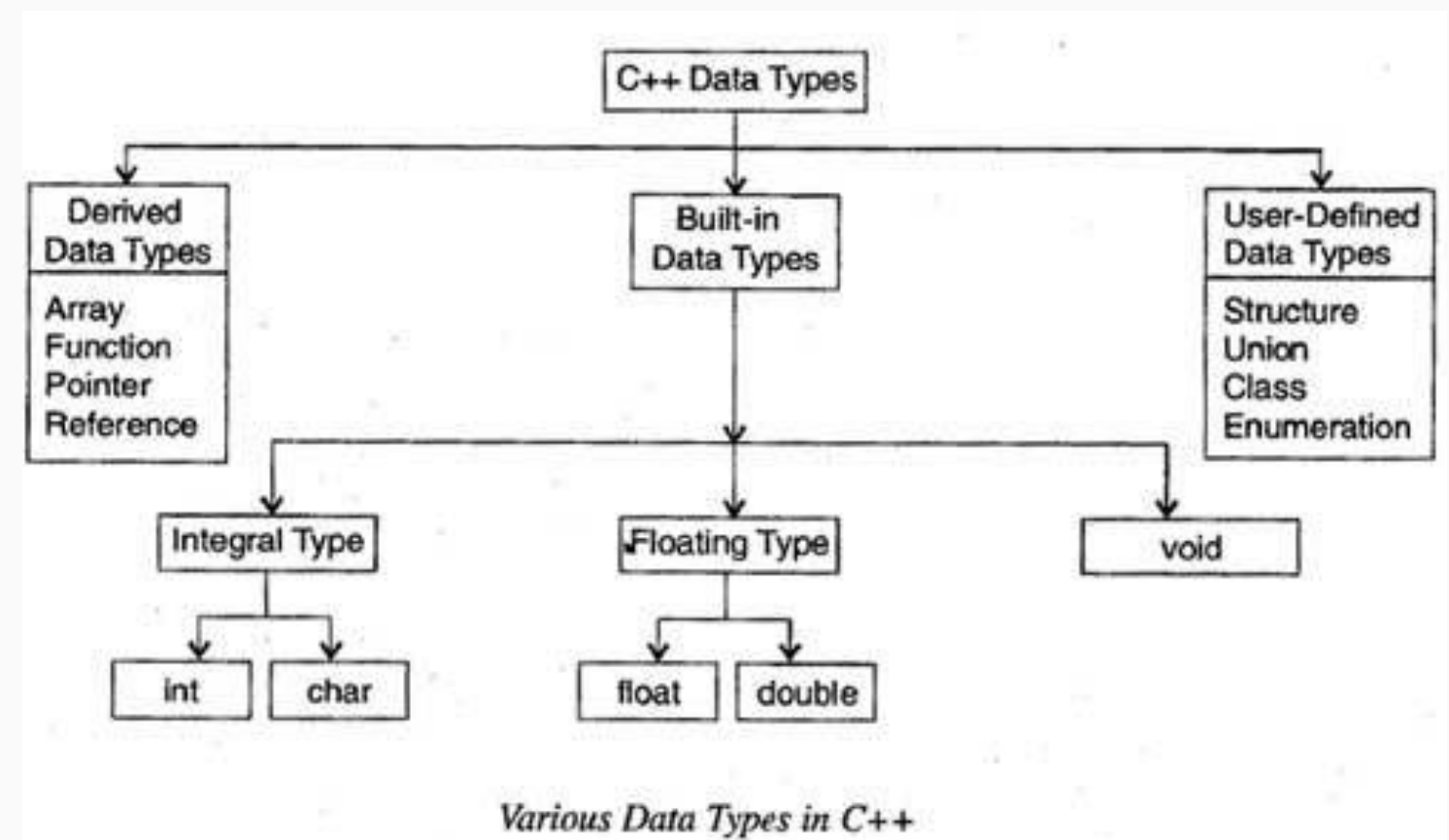
**Extra: Type Traits!**

# Type Traits!

Trabajar con distintos tipos de datos,  
requiere más detalle

Existen muchos tipos de datos

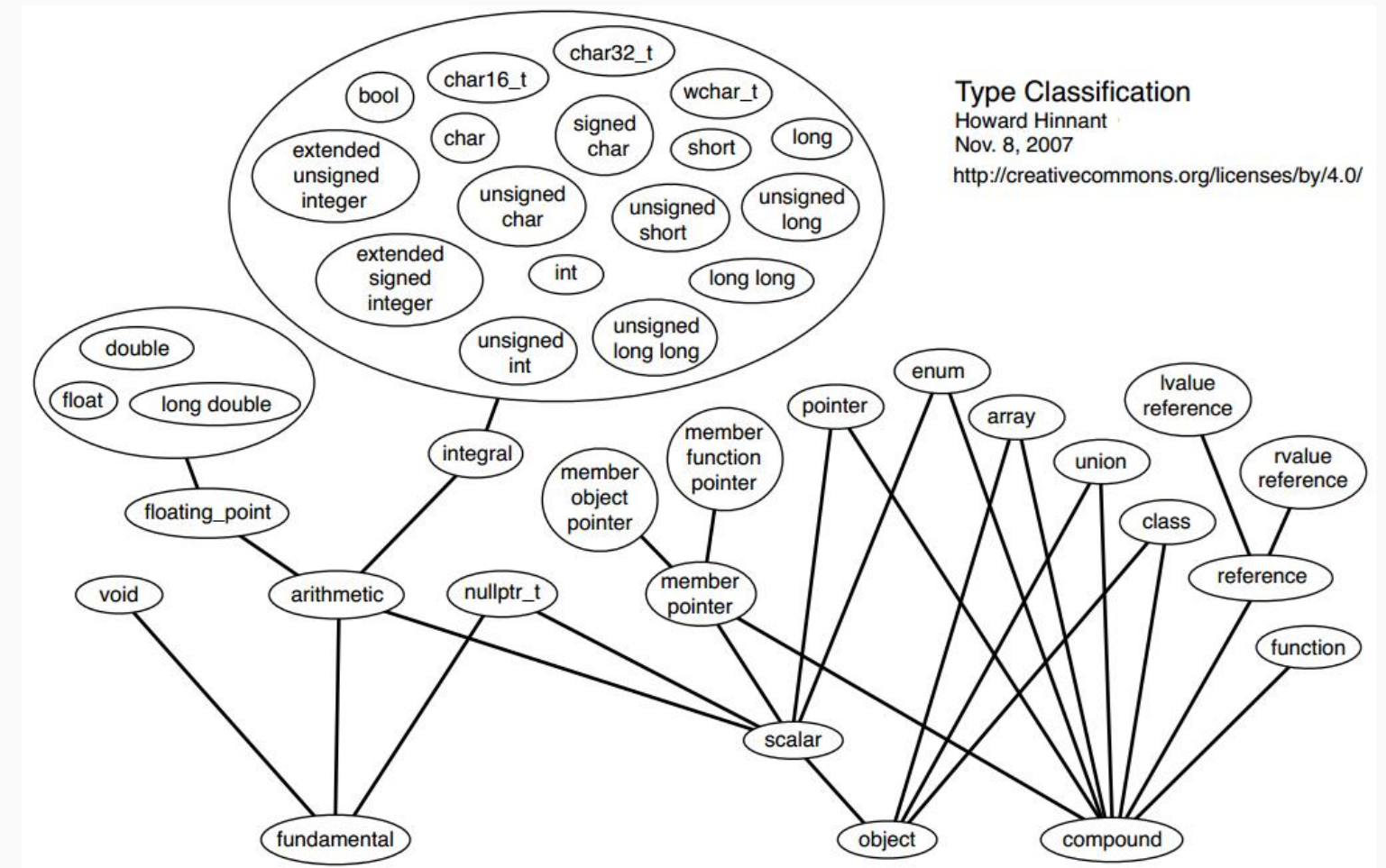
Algunos tipos contienen otros tipos de  
datos



# ¿Qué creen que es un type trait?

Es una estructura (template) que te da información acerca del tipo sobre el que trabajan

*Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine "policy" or "implementation details". - Bjarne Stroustrup*



# ¿Por qué son importantes?

- Nos permite elegir algoritmos específicos dependiendo del tipo de dato (e.g. ordenamientos)
- Colocar límites en nuestros tipos
- Restringir ciertos tipos de datos
- Definir una cantidad de elementos a mostrar (e.g. cantidad de decimales)
- No permitir que nuestra estructura funciona con ciertos tipos

```
template <typename T, typename U>
struct is_same_type {
    static constexpr bool value {false};
};
```

```
template <typename T>
struct is_same_type<T, T> {
    static constexpr bool value {true};
};
```



# ¿Por qué son importantes?

- Imaginemos una situación donde necesitamos que una función genérica no se ejecute con un tipo de dato (e.g. double), ¿qué harían?

```
template<typename T>
T funct(T value) {
    ...
}

template<>
double funct(double value) {
    assert(false && "Illegal");
    return value;
}
```

Pero, ¿qué pasaría si hubieran más excepciones?

```
template<>
char funct(char value) {
    ...
}

template<>
long funct(long value) {
    ...
}
```

# Por qué son importantes?

- Imaginemos una situación donde necesitamos que una función genérica no se ejecute con un tipo de dato (e.g. double), qué harían?

```
template<typename T>
struct is_functable {
    static const bool value = false;
};

template<>
struct is_functable<unsigned short> {
    static const bool value = true;
};
```

```
assert(is_functable<T>::value &&
    "Cannot funct this type");
```

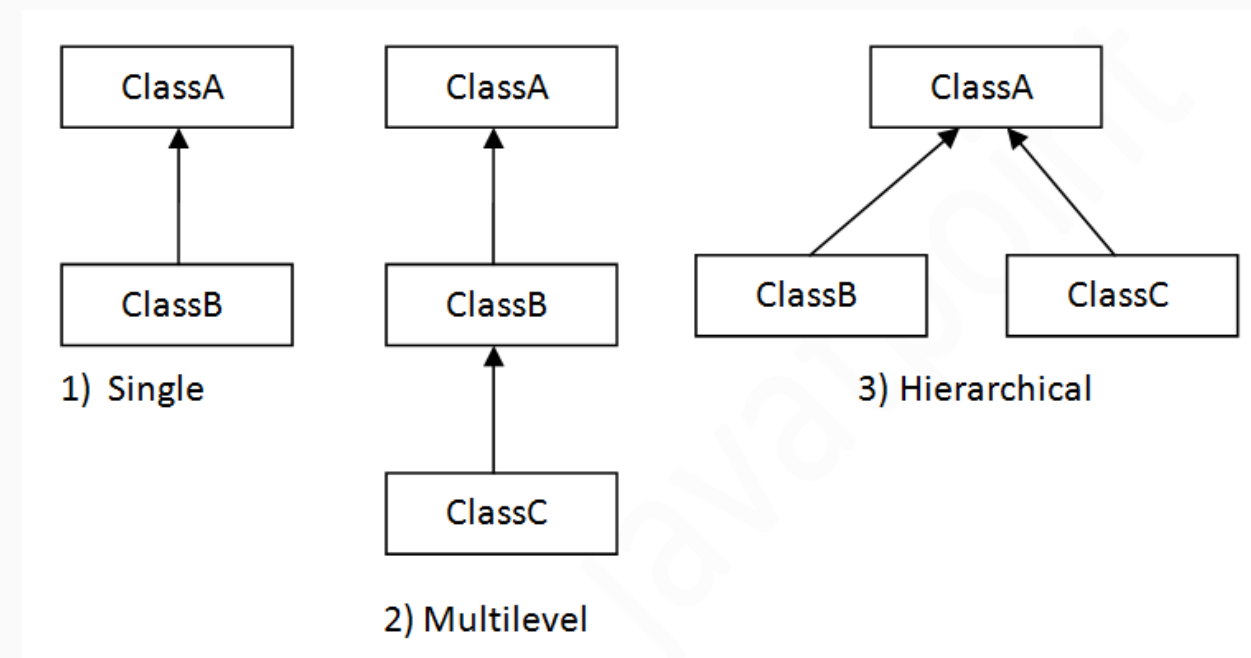
# Recuerden

Type traits resultan en tiempo de compilación, no ejecución

Usualmente se implementa como un template struct

Se suele utilizar herencia para tener un trait base

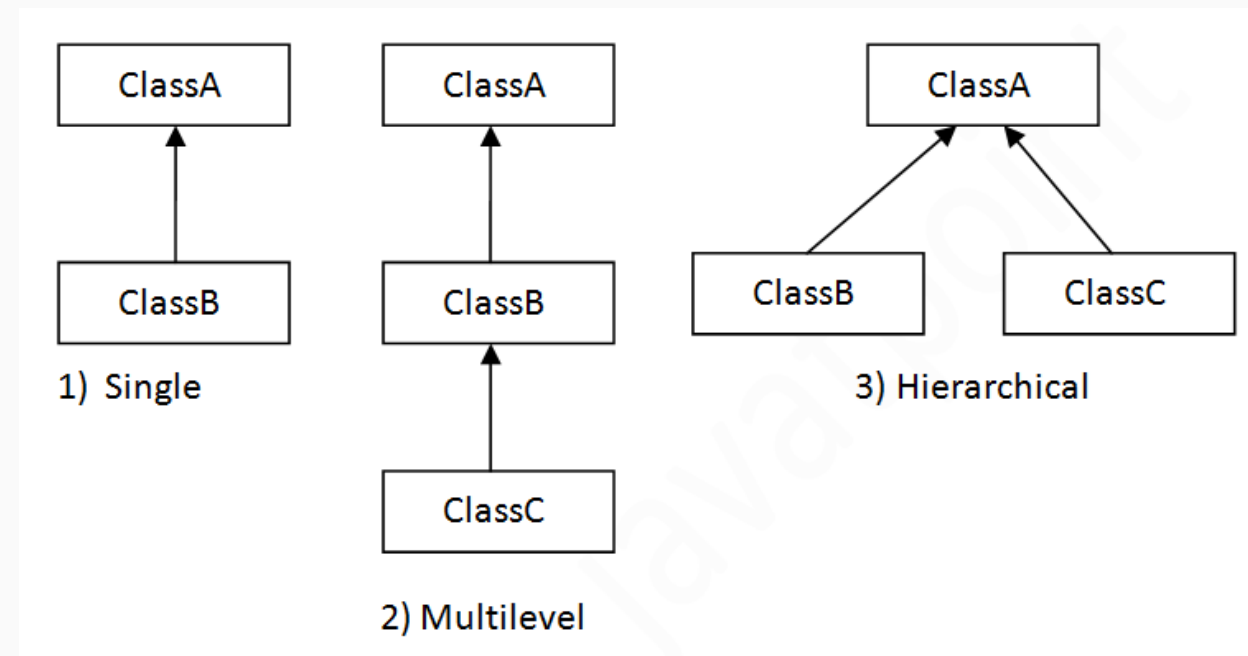
La base suele ser vacía o tiene valores por defecto



# Recuerden

En la mayoría de casos, las estructuras son pequeñas y proveen sólo una pieza de información

No hay un tipo de sintaxis especial para definir un type trait



# Ejemplos

- Limits:

*long max = numeric\_limits<long>::max();*

*int min = numeric\_limits<int>::min();*

donde max y min representan el valor máximo y mínimo para cierto tipo de datos

También pueden haber type traits de dos tipos e.g. *saber si un tipo es convertible a otro*

# Ejemplos

Imaginen dos type traits, una para floats (Float) y otro para ints (Integer).

Se desea que los ints se ordenen de menor a mayor, mientras que los floats de mayor a menor en una lista simplemente enlazada (al momento de ingresar).

**Cómo definirían los type traits?**

**Cómo implementarían una lista simplemente enlazada que use los traits para insertar y eliminar elementos?**

# Ejemplos

```
template <typename T>
struct Greater {
    bool operator()(T a, T b) {
        return a >= b;
    }
};
```

```
struct Integer {
    typedef int T;
    typedef Greater<T> Operation;
};
```

```
template <typename T>
struct Less {
    bool operator()(T a, T b) {
        return a <= b;
    }
};
```

```
struct Float {
    typedef float T;
    typedef Less<T> Operation;
};
```

# Welcome to Algorithms and Data Structures! - CS2100