I102 - Paradigmas de Programación

# INTRODUCCIÓN A CONTAINERS

3.5

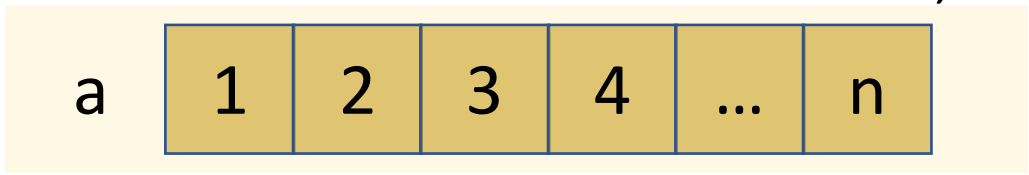

Prof. Mariano Scaramal

Ingeniería en Inteligencia Artificial

# Contenedores Secuenciales

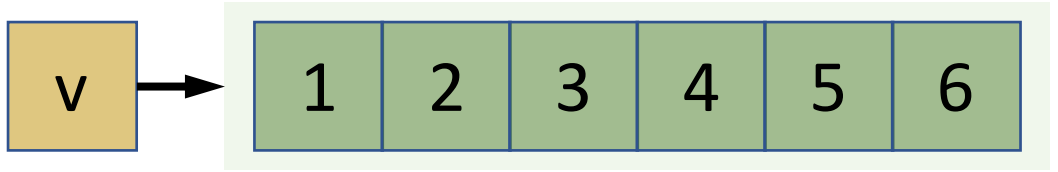
- Tamaño fijo, con datos almacenados en forma secuencial, no utiliza memoria dinámica.

array<T, n>



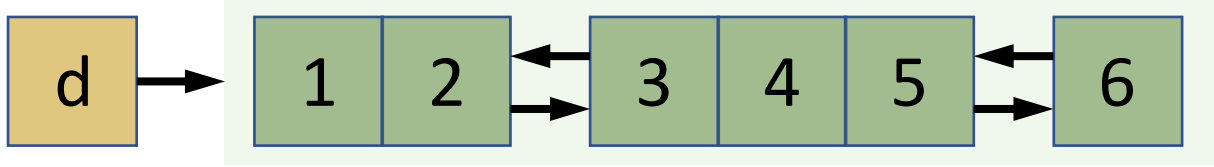

- Array de tamaño dinámico con datos almacenados en forma secuencial.

vector<T>



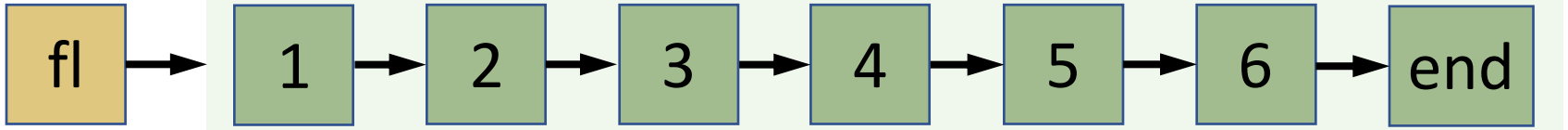

- Array de tamaño dinámico con datos almacenados en forma secuencial.

deque<T>



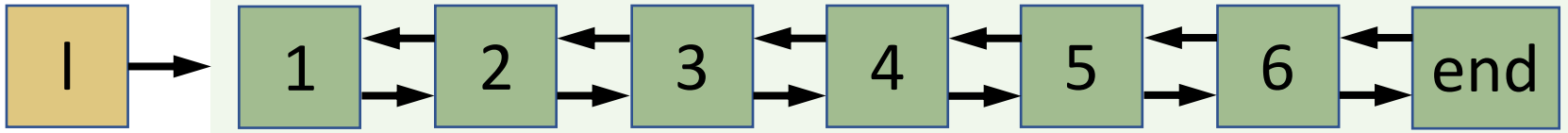

- Lista simplemente enlazada.

foward_list<T>




- Lista doblemente enlazada.

list<T>



# Array (std::array)

Este container funciona en forma similar al array ya conocido de C.

Al igual que en C, *std::array* requiere un tamaño fijo.

Debido a esto, no se aloca memoria en tiempo de ejecución.

Si bien en el array de C también se pueden almacenar objetos, *std::array* tiene funcionalidades útiles para el manejo de objetos.

```
#include <iostream>
#include <array>       // Se necesita incluir array

int main() {
   std::array<int, 5> arr = {3, 2, 6, 1, 5};
   // Acceder valores como un array de C y con .at()
   std::cout << "El primer valor es: " << arr[0] << std::endl;
   std::cout << "El cuarto valor es: " << arr.at(3) << std::endl;

   // Utilizar el for por rango
   std::cout << "Los elementos de arr son: ";
   for (const int n : arr)
      std::cout << n << " ";
   std::cout << std::endl;

   return 0;
}
```

# Vector (std::vector)

Este container encapsula un array de tamaño dinámico.

Debido a su versatilidad, suele utilizarse más frecuentemente que los arrays además de poseer varios métodos interesantes como ser insert, erase, clear, reverse, empty, capacity, at, etc.

```
#include <iostream>
#include <string>
#include <vector>     // Se necesita incluir vector

using namespace std;

void printVec(vector<int> v, string name){
   cout << name + " es: ";
   for (const int n : v)
      cout << n << " ";
   cout << endl;
}
```

```
int main() {
   vector<int> vec1 {1, 2, 3};
   vector<int> vec2 (4, 9);
   // Imprime los vectores
   printVec(vec1, "vec1"); printVec(vec2, "vec2");
   // Uso de push_back y pop_back
   vec1.push_back(4); vec2.pop_back();
   // Uso de size(), front() y back()
   cout << "Vec1 tiene un size de " << vec1.size() << endl;
   cout << "Vec1 comienza con " << vec1.front() << endl;
   cout << "Vec1 termina con " << vec1.back() << endl;
   // Imprime nuevamente con las modificaciones
   printVec(vec1, "vec1"); printVec(vec2, "vec2");

   return 0;
}
```

# Cola Doblemente Terminada (std::dequeu)

Es una cola donde se puede agregar y sacar información del frente o el final.

Al igual que vector, también tiene los métodos front, back, at, begin, size, empty, insert y erase (no tiene capacity), y más.

```
#include <iostream>
#include <deque>  // Necesario para deque
using namespace std;

void printDeq(deque<int> d, string name){
   cout << name + " es: ";
   for (const int n : d)
      cout << n << " ";
   cout << endl;
}
```

```
int main() {
   deque<int> Dq;
   // Agregar elementos al frente y al final
   Dq.push_back(3);   // Agrega al final
   Dq.push_back(8);   // Agrega al final
   Dq.push_front(5);  // Agrega al frente
   Dq.push_front(1);  // Agrega al frente
   // Se imprime con los datos agregados
   printDeq(Dq, "Dq");
   // Se remueve información
   Dq.pop_front();  // Remueve el primer elemento
   Dq.pop_back();   // Remueve el último elemento
   // Se imprime luego de las modificaciones
   printDeq(Dq, "Dq");

   return 0;
}
```

# Listas (std::forward_list y std::list)

Implementan una lista simplemente enlazada (forward_list) y una lista doblemente enlazada (list). No se provee acceso aleatorio en ningún caso.

Tienen algunos métodos similares a vector y los propios según la lista.

**Ejemplo con forward_list**

```
include <iostream>
#include <forward_list> // Para usar forward_list
using namespace std;

void printFL(forward_list<int> fl, string name){
    cout << name + " es: ";
    for (const int n : fl)
        cout << n << " ";
    cout << endl;
}
```

```
int main() {
    forward_list<int> myList = {5, 8, 3, 4, 1, 6};
    // Imprimir lista
    printFL(myList, "myList");
    myList.push_front(0); // Insertar al frente
    myList.push_front(9); // Insertar al frente
    myList.remove(3);       // Remover todos los 3
    // Imprimir luego de las modificaciones
    printFL(myList, "myList");
    myList.reverse(); // Invertir la lista
    // Imprimir lista invertida
    printFL(myList, "myList");

    return 0;
}
```

# Pilas y Colas

Las pilas y colas son implementadas con las librerías stack y queue, respectivamente.

En las pilas, se tienen los métodos push y pop, propios de este modelo. Nótese que se utiliza el modelo LIFO para este caso.

En las colas, también se tienen los métodos push y pop que funcionan con el modelo FIFO, típico de una cola.

Las pilas y las colas se pueden inicializar con push, con un vector o con una lista.

```
#include <iostream>
#include <stack>
#include <queue>
#include <vector>
#include <list>

int main() {
    std::vector<int> vec = {1, 2, 3};
    std::list<int> lst = {10, 20, 30};

    std::stack<int, std::vector<int>> st(vec);
    std::stack<int, std::list<int>> st2(lst);
    std::queue<int, std::vector<int>> q(vec);
    std::queue<int, std::list<int>> q2(lst);

    std::cout << st.top() << std::endl;     // Imprime 3
    std::cout << st2.top() << std::endl;   // Imprime 30
    std::cout << q.front() << std::endl;   // Imprime 1
    std::cout << q2.front() << std::endl; // Imprime 10

    return 0;
}
```

# Set y Multiset

El set es un conjunto de elementos que no puede tener repetidos. Por otro lado, el multiset es un conjunto de elementos que pueden ser repetidos.

Ambos containers mantienen los elementos ordenados. De usar valores numéricos el default es en orden ascendente.

```
#include <iostream>
#include <set>
#include <string>

using namespace std;

int main() {
    set<int> setDes = {5, 2, 8, 3, 10};
    multiset<int, greater<int>> setAsc = {3, 5, 9, 21, 5};
    // Imprime 2 3 5 8 10
    cout << "setDes es: ";
    for (const auto& val : setDes)
        cout << val << " ";
    cout << std::endl;
    // Imprime 21 9 5 5 3
    cout << "setAsc es: ";
    for (const auto& val : setAsc)
        cout << val << " ";
    cout << std::endl;

    return 0;
}
```

# Map y Multimap

El map asocia un key con un value. Similarmente a set, no permite duplicados.

Por otro lado, un multimap tiene permite duplicados.

También se hizo uso de un modelo pair. El elemento *std::pair* es de la librería <*utility*> y simplemente crear una asociación de par entre dos datos.

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

int main() {
    map<std::string, int> m;
    // Producto y precio por kg
    m["manzana"] = 5;
    m["banana"] = 2;
    // Para insertar un elemento se puede hacer con un pair
    m.insert(std::make_pair("sandía", 1));
    // Para recorrerlo
    for (const auto& pair : m)
        cout << pair.first << ": " << pair.second << endl;
    // También se pueden acceder los precios
    cout << "El precio de la banana es: " << m["banana"] << endl;
    // Borrar un elemento
    m.erase("banana");
    return 0;
}
```

Universidad de SanAndrés

# Preguntas?