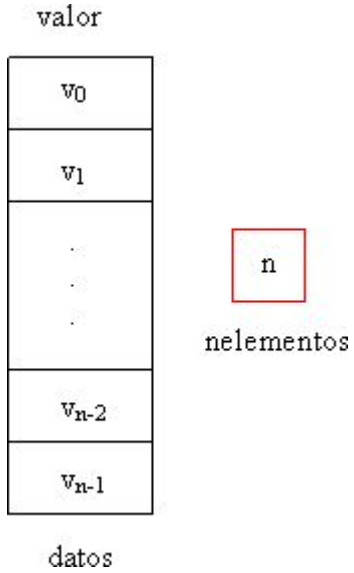
Vectores estáticos y dinámicos

Lección 4:

- **Vectores estáticos**
- **Vectores dinámicos**
- **Relaciones entre clases de objetos**



Parte I:
Vectores estáticos

- Definiciones
- La clase vector estático
- Búsqueda binaria

Motivación

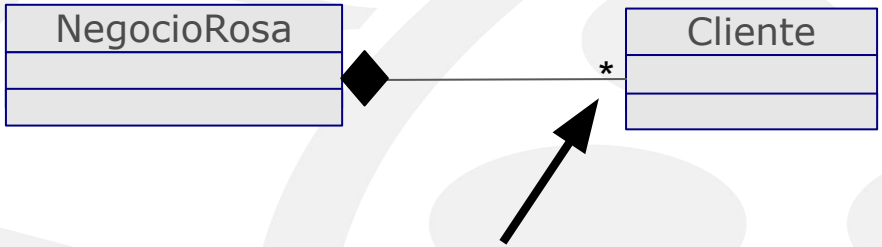
- Rosa vende por Internet abalorios que ella misma fabrica y quiere gestionar el listín de direcciones de sus clientes.
- Lo que pretende hacer es: ordenar, buscar y listar sus clientes.
- Calcula que tendrá unos 100, y elige un vector estático.


```
class Cliente{
     string nombre;
     string direcc, tfno;
public:
     Cliente(string...);
     ...
}
```

```
Class NegocioRosa{
     Cliente clientes[100];
public:
     NegocioRosa();
     ...
}
```

Motivación

- Al principio le va bien pero al cabo de los meses consigue más clientes y tiene que añadir 50 posiciones más a su vector.
- Se da cuenta que no es la solución, no le gustaría ponerle barreras a su negocio, querría que creciera o decreciera según su volumen de datos.
- Esto lo resolvería un vector dinámico.



El (*) en UML significa muchos, sin una cota máxima

Definiciones

- Un **vector (array)** es una zona contigua de almacenamiento en memoria que contiene objetos de un mismo tipo.
- Un **vector estático** tiene asignado una cantidad fija de memoria y no es capaz de crecer o decrecer en tiempo de ejecución.
- Un **vector dinámico** tiene asignado un espacio inicial que puede crecer o decrecer en función de que lo hagan las necesidades de almacenar datos.

Definiciones

Los vectores se pueden catalogar del siguiente modo:

- Son EEDD lineales y de acceso directo o aleatorio mediante un índice único (en tiempo **O(1)**).
- Son unidimensionales aunque pueden manejar dos dimensiones convirtiéndose entonces en matrices.
- Están disponibles en la gran mayoría de lenguajes.
- Son el **contenedor habitual** para otras estructuras de datos y adaptadores: pilas, tablas hash, etc.
- No son perfectas: la inserción en posiciones intermedias tiene un coste **O(n)**.

Vectores estáticos

- Los vectores estáticos son la solución para manejar datos con un tamaño máximo conocido.
- Aunque se reserve un espacio determinado (tamaño físico) puede usarse sólo una parte (tamaño lógico).
- Soportados directamente por la mayoría de lenguajes.

```
int arr[100];
int nElem = 0;

arr[0] = 77;
arr[1] = 99;
nElem = 2;
for (int i=0; i<nElem; i++) cout << arr[i] << endl;
```

Este código define un array de tamaño físico 100, inserta dos elementos y queda con tamaño lógico igual a 2. Luego se muestra su contenido por pantalla.

Vectores estáticos

Búsqueda secuencial:
Este código busca el valor 66
El resultado es: “No está 66”

```
int cbusca = 66;
bool encontrado = false;

for(j=0; j<nElem; j++)
     if(arr[j] == cbusca) {
          encontrado = true;
          break;
     }
if(encontrado)
     cout << "Encontrado "<<
cbusca << endl;
else cout << "No está " <<
cbusca << endl;
```

Este código ordena el array

```
#include <algorithm>
...
int main() {
     int values[] = { 40, 10,
100, 90, 20, 25 };
     sort(values, values+6);
     for (int i=0; i<6; i++) {
        cout << values[i] << "
";
     }
     return 0;
}
```

La clase Vector estático

Ejemplo de definición de vector estático para enteros, es mejorable añadiendo excepciones

```
#ifndef VESTATICOINT_H
#define   VESTATICOINT_H

class VEstaticoInt {
    int tama;
    int *v;

public:
    VEstaticoInt(int tama);
    VEstaticoInt(const VEstaticoInt& orig);
    VEstaticoInt& operator=(const VEstaticoInt& orig);

    int lee(unsigned pos) { return v[pos]; }
    void escribe(unsigned pos, int dato) { v[pos] = dato; }
    int& operator[](unsigned pos) {
     return v[pos];
    }
    void ordenar();
    int busqueda(int dato);
    int busquedaBin(int dato);
    ~VEstaticoInt() { delete []v; }
};
#endif    /* VESTATICOINT_H */
```

La clase Vector estático

El fichero cpp

```
#include <algorithm>
#include "VEstaticoInt.h"

VEstaticoInt::VEstaticoInt(int tama) {
    v = new int[this->tama = tama];
}

VEstaticoInt::VEstaticoInt(const VEstaticoInt &orig) {
    v = new int[tama = orig.tama];
    for (int i = 0; i < tama; i++) v[i] = orig.v[i];
}

void VEstaticoInt::ordenar() {
     sort(v, v + tama);
}

int VEstaticoInt::busca(int dato){
     for(int j = 0; j < tama; j++)
          if(arr[j] == dato) return j;

     return -1;
}
…
```

Búsqueda binaria

- Un vector debe estar ordenado para que funcione la búsqueda binaria o dicotómica.
- También usada en vectores dinámicos.

```
int VEstaticoInt::busquedaBin(int dato) {
  int inf = 0;
  int sup = tama - 1;
  int curIn;

  while (inf <= sup) {
      curIn = (inf + sup) / 2;
      if (v[curIn] == dato)
          return curIn;
      else if (v[curIn] < dato) inf = curIn + 1;
           else sup = curIn - 1;
      }
  }
  return -1;
}
```

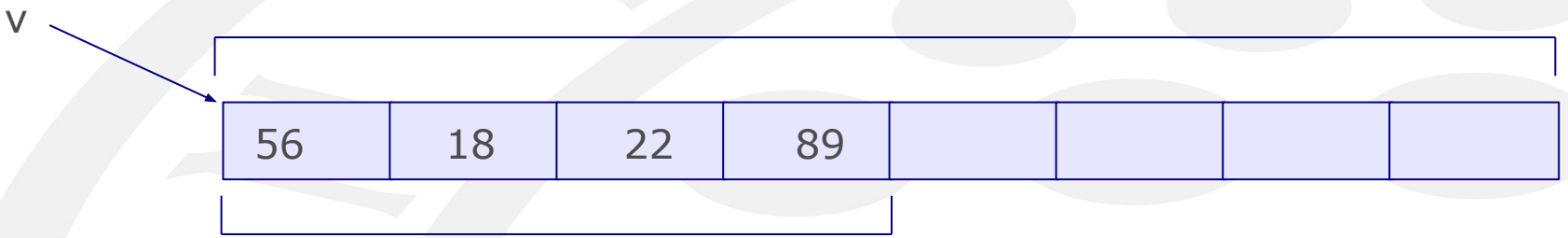
Parte II:
Vectores dinámicos

- Inserción
- Borrado

Vectores dinámicos

- Son vectores que tienen un espacio físico pre-reservado del cual se usa una parte
- Cuando el espacio reservado se agota, se reserva un espacio mayor y se mueven los datos al nuevo espacio

tamaf (tamaño físico) → Número de datos reservados en memoria



tamal (tamaño lógico) → Número de datos utilizados

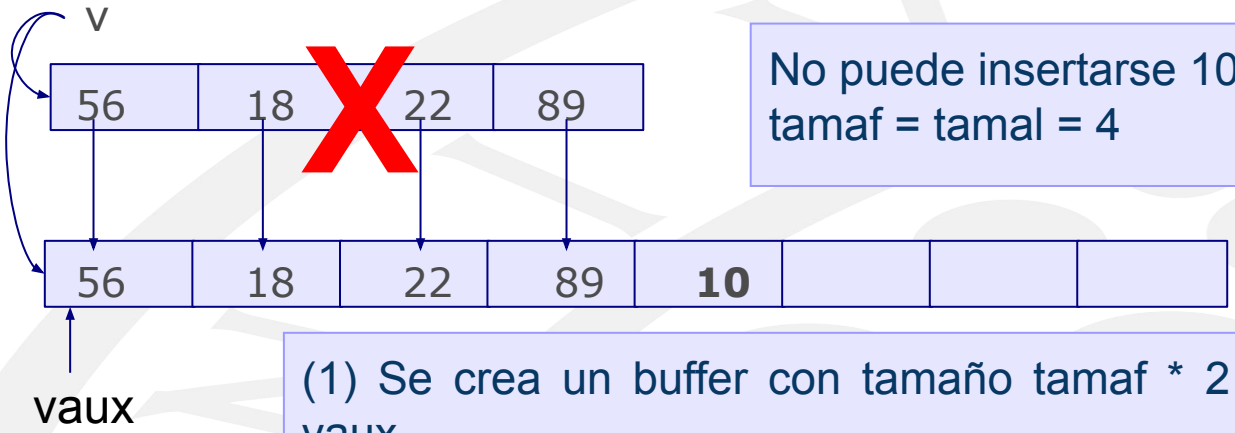
Vectores dinámicos

Los vectores dinámicos son más versátiles porque adaptan su tamaño físico al tamaño de los datos.

```
class VDinamicoInt {
    int tamal,tamaf;
    int *v;

public:
    VDinamicoInt():
    VDinamicoInt(const VDinamicoInt& orig);
    VdinamicoInt& operator=(const VDinamicoInt& orig);
    int lee(unsigned pos){ return v[pos]; }
    void escribe(unsigned pos, int dato){ v[pos] = dato;
}
    int& operator[](unsigned pos) { return v[pos]; }
    void inserta(unsigned pos, int dato);
    int elimina(unsigned pos);
    void aumenta(int dato); // Inserción por la derecha
    int disminuye(); // Eliminar dato por la derecha
    unsigned tama(){ return tamal; };
    void ordenar();
    int busca(int dato);
    int busquedaBin(int dato);
    ~VDinamico() { delete[] v; }
};
```

Inserción en vectores dinámicos

- La función aumenta() inserta un dato al final, pero si no cabe (tamaf==tamal), entonces el vector **crece al doble** para dejar nuevo espacio libre.



No puede insertarse 10 porque tamaf = tamal = 4

(1) Se crea un buffer con tamaño tamaf * 2 y se apunta por vaux
(2) se copian los datos al nuevo buffer
(3) se elimina el buffer antiguo
(4) se apunta v a vaux y
(5) se inserta 10 al final

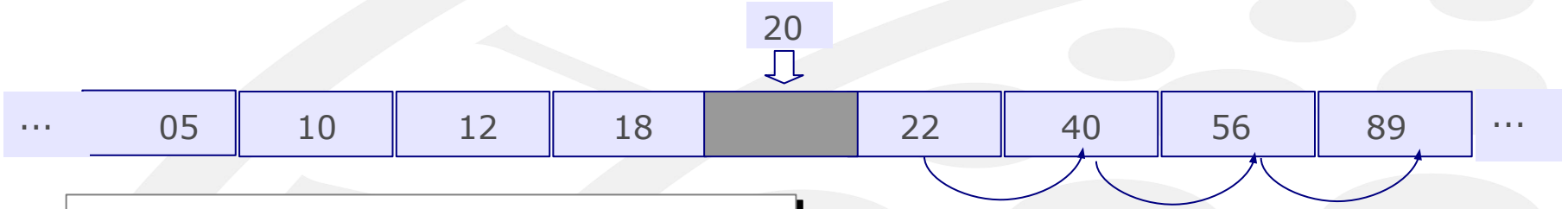
Inserción en vectores dinámicos

- Este proceso es más eficiente que crecer un tamaño constante, adaptándose a la magnitud de los datos.
- El tamaño físico será siempre potencia de 2

```
void VDinamicoInt::aumenta(int dato){
     if(tamal==tamaf) {
          int *vaux;
          vaux= new int[tamaf=tamaf*2];
          for(int i=0;i<tamal;i++)
               vaux[i]=v[i];
          delete[] v;
          v=vaux;
     }
     v[tamal++]=dato;
}
```

Inserción en vectores dinámicos

- Insertar al final de un vector tiene un tiempo constante (siempre que haya espacio)
- Insertar en otra posición es tiempo lineal porque en el peor de los casos hay que abrir un hueco de tamaño tamal




```
...
for(unsigned i=tamal-1;i>=pos;i--){
    v[i+1]=v[i];
}
v[pos]=t;
...
```

Este código de inserta() introduce dato en la posición pos

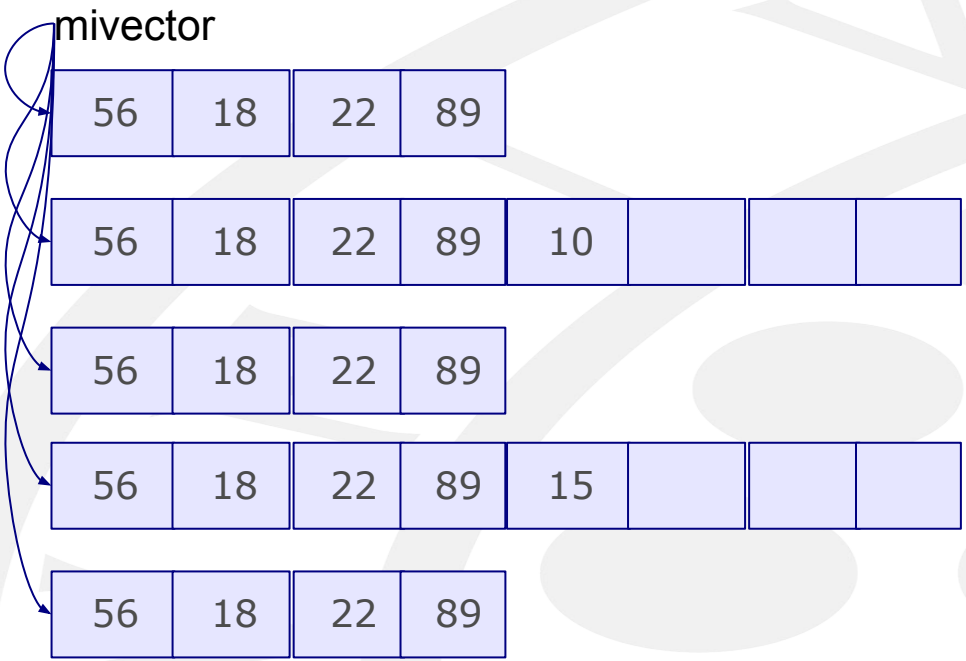
Borrar datos en vectores dinámicos

- El borrado en la posición final también es constante
- Si el vector sufre muchos borrados se debe disminuir su tamaño a la mitad

```
int VDinamicoInt::disminuye(){
  if(tamal*3<tamaf) {
      tamaf=tamaf/2;
      int *vaux = new int[tamaf];
      for(unsigned i=0;i<tamal;i++){
         vaux[i]=v[i];
      };
      delete[] v;
      v=vaux;
  }
  return v[--taml];
}
```

Borrar datos en vectores dinámicos

- La reducción del tamaño físico se hace cuando el tamaño lógico es la tercera parte del tamaño físico.

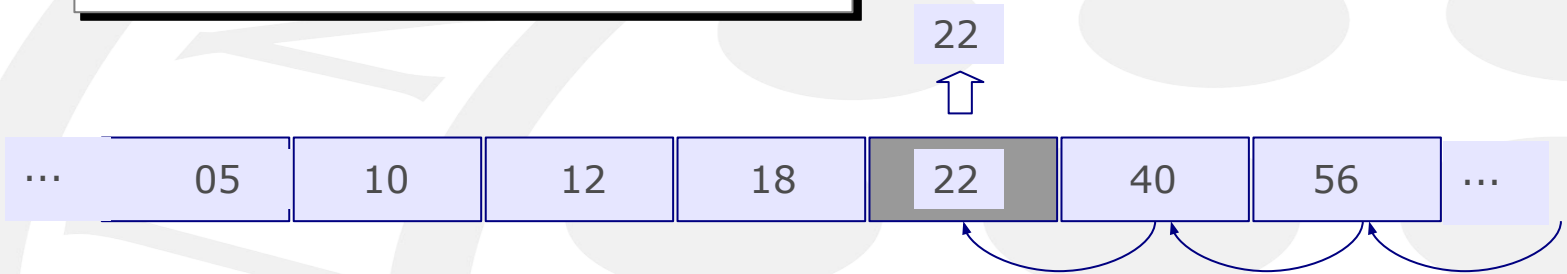


Esperar a un 1/3 evita esto:
- inserta 10, crece al doble
- borra 10, decrece
- inserta 15, aumenta
- borra 15, decrece

Borrar datos en vectores dinámicos

- El borrado en posiciones intermedias se considera operación en tiempo lineal.
- En realidad borrar significa sobreescribir posiciones.
- Se reduce el tamaño a la mitad si **tamal*3<tamaf**.

```
for(unsigned i=pos;i<taml;i++){
    v[i]=v[i+1];
};
taml--;
```



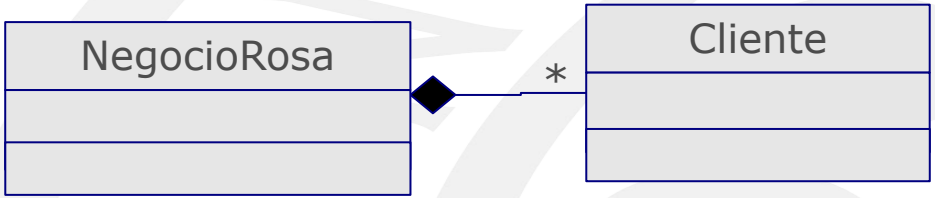
Parte III:
Relaciones entre clases de objetos

- Composición
- Asociación

Relaciones entre clases de objetos: composición

- Las **composiciones** pueden implementarse de dos modos:
  (1) insertando objetos dentro del vector o
  (2) insertando un puntero a dicho objeto
- En cualquier caso la clase contenedora crea y destruye los objetos

Recordemos el negocio de Rosa




```
class NegocioRosa{
     Cliente *cli;
     int tamal;
public:
     NegocioRosa(int n){
          cli = new Cliente[n];
          …
     }
    ~NegocioRosa(){
          delete[] cli;
    }
...
};
```

Relaciones entre clases de objetos: composición

- En las composiciones implementadas mediante punteros, la clase contenedora debe crear y destruir los objetos que contiene

```
class NegocioRosa{
    Cliente **cli;
    int tamal;
public:
    NegocioRosa(int n){
        cli = new Cliente*[n];
        ...
    }

    void nuevoCli(Cliente &c);

    ~NegocioRosa();
};
```

```
void NegocioRosa::nuevoCli(Cliente &c)
{
   cli[tamal++] = new Cliente(c);
}

NegocioRosa::~NegocioRosa(){
    for (int i=0; i<tamal; i++)
        delete cli[i];

    delete[] cli;
}
```

Relaciones entre clases de objetos: composición

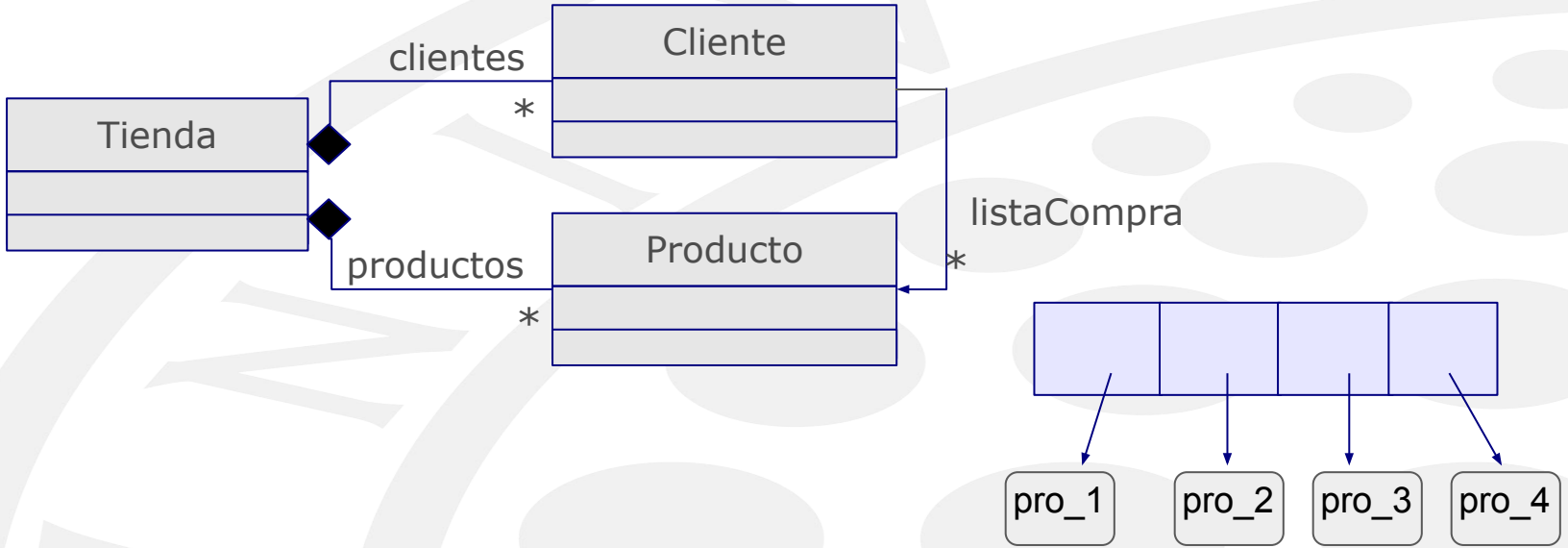
- Lo más práctico y sencillo es utilizar una plantilla de vector dinámico para implementar este tipo de relaciones

```
class NegocioRosa{
    VDinamico<Cliente*> cli;

public:
    NegocioRosa(int n):cli(n){}

    void nuevoCli(Cliente &c);

    ~NegocioRosa();
};
```

```
void NegocioRosa::nuevoCli(Cliente &c)
{
   cli.aumenta(new Cliente(c));
}

NegocioRosa::~NegocioRosa(){
    for (int i=0; i<tamal; i++)
        delete cli[i];
}
```

Relaciones entre clases de objetos: asociación

- Las **asociaciones** se implementan siempre a través de punteros a objetos ya existentes, cuyo ciclo de vida es independiente



Relaciones entre clases de objetos: asociación

- Las **asociaciones** se implementan siempre a través de punteros a objetos ya existentes, cuyo ciclo de vida es independiente
- Un objeto nunca debe destruir los objetos al otro lado de la asociación

```
class Cliente{
    VDinamico<Producto *>
listaCompra;
    string nombre, apellidos, nif;

public:
    Cliente(): listaCompra() {}
    void nuevoProd(Producto *p);
…
}
```

```
Cliente::nuevoProd(Producto*  p){
    listaCompra.aumentar(p);
}

Cliente::~Cliente(){}
```

No se borran los productos!!!

¿Cuando uso vectores?

- Los vectores son las estructuras de datos más simples y fáciles de manejar. Se aconseja usarlas:
  - Si el tamaño de los datos es pequeño.
  - Si puedo acceder mediante índices enteros: O(1)
  - Si las inserciones/borrados son por el final: O(1)
  - Realizo búsquedas binarias en un vector ordenado O(log2 n) y apenas hago inserciones O(n).
- Por tanto no son siempre la mejor opción:
  - Cuando haya que insertar en cualquier posición
  - Haya grandes masas de datos localizables por clave y que sufren altas/bajas/modificaciones.

Conclusiones

- Los vectores son las estructuras de datos más simples y fáciles de manejar.
- Siempre no son la mejor opción
- Relaciones entre clases
  - Composiciones: la clase contenedora crea y destruye el objeto
  - Asociaciones: la clase contenedora no crea ni destruye los objetos que contiene

De ahora en adelante...

- Los vectores son EEDD muy utilizadas por ser fáciles de manejar, pero tienen los inconvenientes citados anteriormente
- Utilizaremos listas enlazadas para mejorar el tiempo de inserción en posiciones intermedias
- Utilizaremos contenedores asociativos como árboles y tablas hash para mejorar las búsquedas de datos por clave

Lección 5: Matrices y conjuntos de bits