

# Lección 5:

## Matrices y conjuntos de bits



- Matrices
  - Motivación
  - Definición de matriz
  - La clase matemática matriz
- Conjuntos
  - Motivación
  - Definición
  - Conjuntos
  - Conjuntos de bits

# Motivación



Un equipo de fútbol que juega la liga quiere llevar un control de los partidos que juega y de los goles que marcan cada uno de sus jugadores.

		jornadas										
		1	2	3	4	5	6	7	8			38
jugadores	1											
	2											
	3											
	4											
	18											

La fila 2 tiene los goles marcado por el jugador 2


La columna 4 guarda los goles marcados en la jornada 4

# Motivación



Se crea un array bidimensional **goles** que permite resolver cuestiones como:

- Cuántos goles ha marcado el jugador número  $i$  ?
- Cuantos goles se han marcado en el partido de la jornada  $j$  ?
- Cuántos goles ha marcado el jugador  $i$  en el partido  $j$  ?



		jornadas							
		1	2	3	4	5	6	7	8
jugadores	1								
	2								
	3								
	4								

A blue arrow points from the text **goles(2,4)** to the cell at row 2, column 4 of the table.



# Definición de matriz

- Un **array bidimensional** o **matriz** es un contenedor tipo vector que se maneja mediante dos índices, uno para las filas y otro para las columnas.
- Al igual que el vector también representan una zona contigua de memoria que almacena objetos de la misma clase.
- El concepto de matriz puede emplearse como objeto matemático y como contenedor.
- Un array bidimensional de tamaño  $(n \times m)$  indica que tiene ***n*** filas y ***m*** columnas.

# Definiendo matrices en C++

```
int a[5][4];
for (int i=0; i<5; i++)
    for (int j=0; j<4; j++)
        a[i][j] = 0;
```

*a* es un array de enteros con tamaño constante: 5 filas y 4 columnas; se inicializa a 0

```
int n = 5;
int m = 4;
int **b;

b = new int*[n];
for (int i=0; i<n; i++)
    b[i] = new int[m];

for (int i=0; i<n; i++){
    for (int j=0; j<m; j++){
        b[i][j] = i*j;
        cout << b[i][j] << " ";
    }
    cout << endl;
}
```

*b* es un array de enteros con tamaño dado en tiempo de ejecución; el array es inicializado a  $i*j$

Salida:

```
0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9
```

# Definiendo matrices en C++

```
int n = 5;
int m = 4;
int ***b;

b = new int**[n];
for (int i=0; i<n; i++)
    b[i]= new int*[m];

for (int i=0; i<n; i++){
    for (int j=0; j<m; j++){
        b[i][j] = new int;
        *b[i][j] = i*j;
    }
}

for (int i=0; i<n; i++){
    for (int j=0; j<m; j++){
        cout << *b[i][j] << " ";
    }
    cout << endl;
}
```

*b* es una matriz de  
punteros a enteros

Salida:

```
0 0 0 0
0 1 2 3
0 2 4 6
0 4 8 12
```

# La clase matemática matriz

```
template <class T> class Matriz{
private:
    unsigned n, m;
    T **mat;
public:
    Matriz(unsigned nn, unsigned mm, T &dato);
    Matriz(const Matriz<T>& orig);
    Matriz<T> &operator=(const Matriz<T> &orig);
    T &operator()(unsigned i, unsigned j);
    Matriz<T> operator+(const Matriz<T> &a);
    Matriz<T> &operator+=(const Matriz<T> &a);
    Matriz<T> operator-(const Matriz<T> &a);
    Matriz<T> &operator-=(const Matriz<T> &a);
    Matriz<T> operator*(const Matriz<T> &a);
    Matriz<T> operator/(const Matriz<T> &a);
    unsigned nFilas(){return n;}
    unsigned nColum(){return m;}
    ~Matriz();
};
```

Clase Matriz con  
sobrecarga de operadores

# La clase matemática matriz

```
template <class T>
Matriz<T>::Matriz(unsigned nn, unsigned mm, T &dato){
    mat = new T*[n=nn];
    for (unsigned i=0; i<n; i++){
        mat[i]=new T[m=mm];
        for (unsigned j=0; j<m; j++) mat[i][j]=dato;
    }
}
```

Constructor inicializado  
por defecto

```
template <class T>
T &Matriz<T>::operator()(unsigned i, unsigned j){
    if (i>=n) throw ErrorRango();
    if (j>=m) throw ErrorRango();
    return mat[i][j];
}
```

El operator() permite  
acceso a los elementos  
para L/E



# La clase matemática matriz

```
template <class T>
Matriz<T> &Matriz<T>::operator= (const Matriz<T> &orig){
    for (unsigned i<n; i++){
        delete []mat[i];
    }

    delete [] mat;

    mat = new T*[n=orig.n];
    for (unsigned i=0; i<n; i++){
        mat[i]=new T[m=orig.m];

        for (unsigned j=0; j<m; j++){
            mat[i][j]=orig.mat[i][j];
        }
    }
    return *this;
}
```

Es imprescindible  
sobrecargar el operator=  
porque el proporcionado  
por el compilador no es  
válido

# La clase matemática matriz

```
template<class T>
Matriz<T> Matriz<T>::operator*(const Matriz<T> &a){
    Matriz<T> result(n, a.m);
    if (m!=a.n) throw ErrorDimensiones();
    for (unsigned i=0; i<n; i++)
        for (unsigned j=0; j<a.m; j++){
            result.mat[i][j]=0;
            for (int l=0; l<m; l++)
                result.mat[i][j] += mat[i][l] * a.mat[l][j];
        }
    return result;
}
```

devuelve una copia  
de result:  $c=a*b$

```
template<class T>
Matriz<T> &Matriz<T>::operator+=(const Matriz<T> &a){
    if (n!=a.n || m!=a.m) throw ErrorDimensiones();
    for (unsigned i=0; i<n; i++)
        for (unsigned j=0; j<m; j++)
            mat[i][j] = mat[i][j] + a.mat[i][j];
    return *this;
}
```

devuelve una  
referencia:  $c+=b$

# Instanciación de la clase matriz

```
int main(int argc, char** argv) {
    int valorini = 20;

    Matriz<int> mimatriz(3,4,valorini);
    Matriz<int> otra(mimatriz);

    if (mimatriz(2,2)==otra(2,2))
        cout << "Es que son iguales \n";
    else cout << "Es raro que no lo sean \n";

    cout << mimatriz;

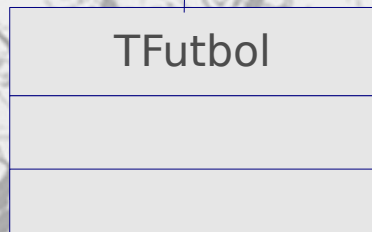
    Matriz<int> c = mimatriz + otra;
    Matriz<int> d = mimatriz - otra;
    valorini = 1;

    Matriz<int> e(4,5,valorini);
    e(1,1)=5;
    otra = mimatriz * e;
    cout << otra;
    mimatriz += d;
```

# Matriz como contenedor



- Una matriz puede servir sólo como contenedor, en este caso se puede volver a implementar o mejor heredar de ella.



Para gestionar la tabla con las jornadas de fútbol se puede utilizar la matriz anterior mediante el mecanismo de la herencia

```
class TFutbol: public Matriz<int> {
public:
    TFutbol(int nfil, int ncol):
        Matriz<int>(nfil,ncol){}
    TFutbol(const TFutbol& orig): Matriz<int>(orig){}
    ~TFutbol(){}
    void gol(int jugador, int jornada);
};
```

# Conjuntos: aplicación



Dos entidades bancarias andaluzas se van a fusionar, ambas tienen sucursales por distintas ciudades, pero no más de una oficina por código postal. La futura entidad también seguirá esa política:

- Cuantas oficinas quedarán tras la fusión?
- Cuantos distritos están aún sin oficina?
- Cuantas oficinas quedan repetidas?

Cajalandalus

23005  
18003  
18170  
23760 23660

Cajalquivir

18003  
23660  
21730 23006  
21609

# Definición



Un conjunto  **$P$**  es una colección de elementos no repetidos y sin orden predeterminado. Es un objeto matemático con las siguientes operaciones básicas:

- **$x \in P$** :  $x$  es miembro del conjunto  $P$
- **$P = \emptyset$** : es el conjunto vacío
- **$P \subseteq Q$** :  $P$  es un subconjunto del conjunto  $Q$
- **$P \cup Q$** : (unión) todos los elementos de  $P$  OR  $Q$
- **$P \cap Q$** : (intersección) los elementos de  $P$  AND  $Q$
- **$P - Q$** : resta, los elementos de  $P$  que no estén en  $Q$

# Implementación



La implementación más sencilla de un conjunto es mediante un array unidimensional

Cajalandalus				
23005	18170	23760	18003	23660

Cajalandalus

23005  
18003  
18170  
23760 23660

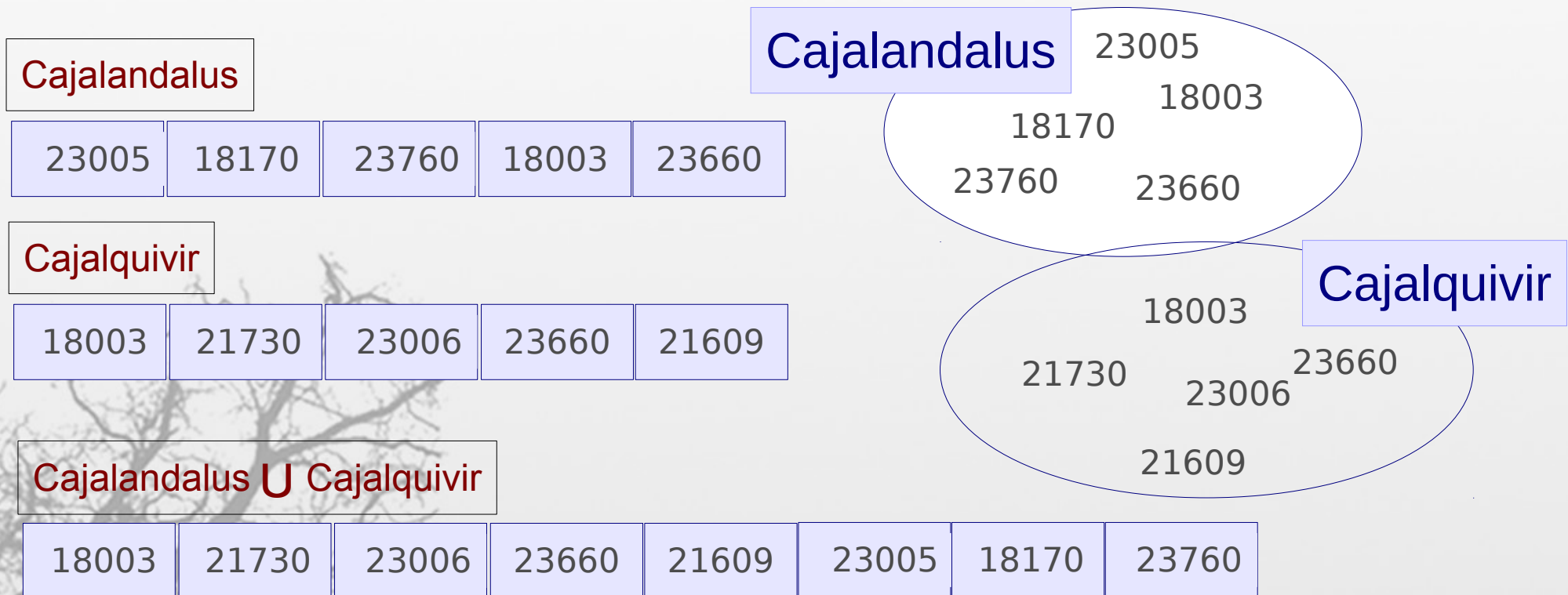
Si A representa al primer banco y B representa al segundo:

- Qué oficinas quedarán tras la fusión?  $\rightarrow A \cup B$
- Qué distritos quedarán sin oficina?  $\rightarrow T - A \cup B$  (siendo T el conjunto de códigos postales)
- Qué distritos tienen oficinas repetidas?  $\rightarrow A \cap B$

# Implementacion



Las operaciones deben considerar la no repetición de los elementos en el conjunto resultado





# Implementación con vectores

```
template <class T>
class Conjunto{
protected:
    T *arr;
    int tama, tamal;
public:
    Conjunto(int tama=10);
    Conjunto(const Conjunto<T>& c);
    ~Conjunto();
    Conjunto<T>& operator=(const Conjunto<T>& c);
    void inserta(const T& ele);
    bool elimina(const T& ele);
    bool contiene(const T& ele);
    Conjunto<T> operator+(const Conjunto<T>& c);
    Conjunto<T> operator-(const Conjunto<T>& c);
    Conjunto<T> intersec(const Conjunto<T>& c);
    int tama(){return tama;}
    void imprime();
};
```

Esta implementación almacena  
cada un elemento del conjunto  
en cada casilla

unión

diferencia

intersección

# Implementación con vectores

```
template <class T> void Conjunto<T>::inserta(const T &ele){  
    if (tamal >= tamaf) throw ErrorTamanio();  
    bool repe = false;  
    for (int i=0; i<tamal; i++)  
        if (arr[i]==ele) {  
            repe=true;  
            break;  
        }  
    if (!repe) arr[tamal++]=ele;
```

Comprobar que  
no están repetidos

```
template <class T> bool Conjunto<T>::elimina(const T& ele){  
    int pos = -1;  
    for (int i=0; i<tamal; i++)  
        if (ele == arr[i]){  
            pos = i;  
            arr[i] = arr[--tamal];  
            break;  
        }  
    return (pos == -1 ? false: true);  
}
```

Compactar tras  
eliminar

# Implementación con vectores

```
template<class T>
Conjunto<T> Conjunto<T>::operator+(const Conjunto<T>& c){
    Conjunto<T> caux(*this);

    for (int i=0; i<c.tamal; c++) {
        caux.inserta(c.arr[i]);
    }

    return caux;
}

template <class T>
Conjunto<T> Conjunto<T>::operator-(const Conjunto<T>& c){
    Conjunto<T> caux(*this);

    for (int i=0; i<c.tamal; c++) {
        caux.elimina(c.arr[i]);
    }

    return caux;
}
```

# Usando la clase Conjunto



```
void main (void){
    Conjunto<int> c;
    cout << "Se introducen 6 elementos\n";
    c.inserta(3); c.inserta(5); c.inserta(7);
    c.inserta(9); c.inserta(4); c.inserta(5);

    cout << "El tamaño es: " << c.tama();
    cout << " porque no admite repetidos" << endl;
    if (c.contiene(7)) cout << "el dato está\n";
    else cout << "El dato no está"<< endl;

    Conjunto<int> d = c;
    d.inserta(8); d.elimina(5); d.elimina(4);

    Conjunto<int> f = d.intersec(c);
    cout <<"Solo deben quedar 3 valores\n";
    f.imprime();

    Conjunto<int> g = c+f;
    g.imprime();
}
```

$c=\{3,5,7,9,4\}$

$d=\{3,7,9,8\}$

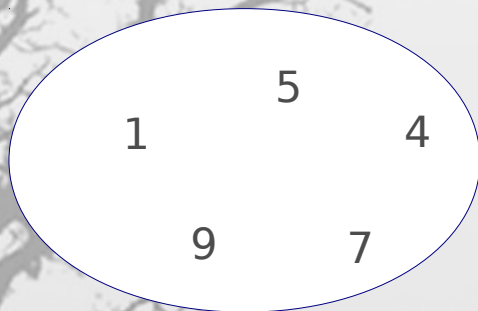
$f=\{3,7,9\}$

$g=\{3,5,7,9,4,8\}$

# Conjuntos de bits (bitset)



- La implementación anterior no resulta eficiente porque las operaciones de unión, intersección y diferencia necesitan un tiempo cuadrático
- Una mejor implementación para conjuntos de enteros con un máximo acotado y razonablemente pequeño son los conjuntos de bits
- El estado del bit  $i$ -ésimo indica si el entero  $i$  pertenece al conjunto o no



0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	1	0	1

# Conjuntos de bits (bitset)



- Se usa un vector de bytes. Cada byte empaqueta 8 bits
- Localizar el bit asociado al entero **n**:
  - $n / 8 \rightarrow$  byte a acceder
  - $n \% 8 \rightarrow$  bit a acceder dentro del byte
- Las operaciones se realizan a nivel de bit de forma eficiente:  $P = \{p_0, p_1, \dots, p_n\}$  y  $Q = \{q_0, q_1, \dots, q_m\}$ 
  - $P \cup Q = \{p_i \mid q_i\} \forall i \in \{0 \dots \max(n, m)\}$
  - $P \cap Q = \{p_i \& q_i\} \forall i \in \{0 \dots \max(n, m)\}$
  - $P - Q = \{p_i \& \sim q_i\} \forall i \in \{0 \dots \max(n, m)\}$

# Implementación de bitsets

La definición es similar a un vector de char

```
class Bitset{
protected:
    char *arr;
    int tama;

public:
    Bitset(int max);
    Bitset(const Bitset &b);
    ~Bitset(){ delete []arr; }
    Bitset& operator=(const Bitset &b);
    void inserta(int ele);
    bool elimina(int ele);
    bool contiene(int ele);
    Bitset operator+(Bitset &b);
    Bitset operator-(const Bitset &b);
    Bitset intersec(const Bitset &b);
};
```

# Implementación de bitsets



```
Bitset::Bitset(int max) {  
    tama = ceil((float) (max + 1) / 8);  
    arr = new char[tama];  
    for (int c = 0; c < tama; c++) arr[c] = 0;  
}
```

```
void Bitset::inserta(int ele) {  
    char mascara = 1 << (ele % 8);  
    arr[ele / 8] |= mascara;  
}
```

```
bool Bitset::elimina(int ele){  
    char mascara = 1 << (elem % 8);  
    arr[ele / 8] &= ~mascara;  
}
```

```
bool Bitset::contiene(int ele){  
    char mascara = 1 << (elem % 8);  
    return (arr[ele / 8] & mascara) != 0  
}
```

Calcular número de bytes  
necesarios





# Implementación de bitsets

```
Bitset Bitset::operator+(Bitset &c) {  
    Bitset *bitSetMayor = this;  
    Bitset *bitSetMenor = &c;  
  
    if (tama < c.tama) swap(bitSetMayor, bitSetMenor);  
  
    Bitset result(*bitSetMayor);  
    for (int i=0; i< bitSetMenor->tama; i++)  
        result.arr[i] |= bitSetMenor->arr[i];  
  
    return result;  
}
```

Función para intercambiar  
variables (STL)

```
Bitset Bitset::operator-(Bitset &c) {  
    Bitset *bitSetMayor = this;  
    Bitset *bitSetMenor = &c;  
  
    if (tama < c.tama) swap(bitSetMayor, bitSetMenor);  
  
    Bitset result(*bitSetMayor);  
    for (int i=0; i< bitSetMenor->tama; i++)  
        result.arr[i] &= ~bitSetMenor->arr[i];  
}
```

# Conclusiones



- La implementación mediante vectores de bits es más eficiente:
  - Unión, resta, intersección:  $O(n)$
  - Inserción, borrado y test de existencia  $O(1)$
- La implementación mediante vectores de datos es más versátil pero menos eficiente:
  - Unión, resta, intersección:  $O(n^2)$
  - Inserción, borrado y test de existencia  $O(n)$



# De ahora en adelante...



- Los vectores y matrices son zonas contiguas de memoria que permiten almacenar datos con acceso directo.
- El problema es que la inserción en posiciones intermedias no es eficiente, ni tampoco la búsqueda en caso de no tener orden establecido.
- Por el contrario, las listas enlazadas y los árboles no se constituyen por espacio contiguos, sino por espacios separados y enlazados entre sí.
- Es posible la implementación de conjuntos mediante contenedores de STL.