ALGORITMOS Y ESTRUCTURAS DE DATOS 1

Práctica: CUACKER

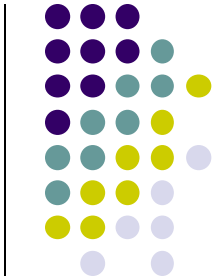
Sesión 4

Introducción a C++

Videotutorial 8

- El puntero this
- Clases amigas
- Declaración adelantada de una clase
- Clases internas
- Redefinición de operadores

El puntero this



- Similar al *this* de Java.
- Dentro de los métodos de una clase, **this** es un puntero que apunta al objeto receptor de la llamada.

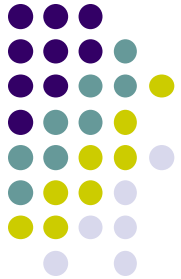
```
class Persona
{
  string nombre;
  long dni, telefono;
  void leer (void);
  void escribir (void);
  void setNombre (string nombre);
};
```

```
void Persona::leer (void) {
    cin >> this->nombre;
}
void Persona::setNombre (string
                          nombre) {
   this->nombre= nombre;
}
```

Ojo: this nunca puede ser NULL.

- En la mayoría de los casos se debe evitar el uso de this, puesto que suele ser innecesario.
  - En lugar de: this->nombre, this->dni, this->telefono, simplemente poner: nombre, dni, telefono.

Clases amigas

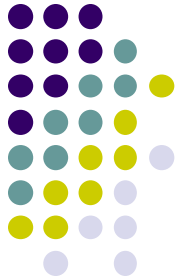


- Por principio, los miembros públicos de una clase deben ser solo los necesarios para los usuarios de la misma.
- Todo lo demás debe ser privado.
- Pero, en algunos casos, puede interesar que una clase acceda a la representación interna de otra.

```
class NodoLista
{
  private:
    string dato;
    NodoLista *sig;
  public:
    NodoLista(string valor="");
};
```

```
class Lista
{
   private:
      NodoLista *primero;
   public:
      Lista();
      ~Lista();
};
```

Clases amigas



- El mecanismo de clases amigas permite en estos casos *saltarse* la ocultación de la implementación.
- **Clase amiga**: permite que otra clase concreta acceda a su parte privada.

```
class NodoLista
{
   friend class Lista;
   private:
      string dato;
      NodoLista *sig;
   public:
      NodoLista(string valor="");
};
```

NodoLista permite que Lista acceda a todos sus miembros (públicos y privados).

Declaración adelantada de una clase

- En el ejemplo, Lista ↔ Nod
  clase declaramos antes?

Declaración adelantada: permite usar un puntero de la clase.

```
class Lista {
   ...
   NodoLista *pr
   ...
};
class NodoLista
   friend class
   ...
};
```

```
class NodoLista {
   friend class Lista;
   ...
};
class Lista {
   ...
   NodoLista *primero;
   ...
};
```

```
class NodoLista;
class Lista {
   ...
   NodoLista *primero;
   ...
};
class NodoLista {
   friend class Lista;
   ...
};
```

Clases internas

- Otra forma de definir la relación nodo-lista es mediante clases internas (o anidadas).
- **Clase interna**: dentro de una clase se puede definir otra clase, que puede ser pública o privada.

```
class Lista {
  private:
    class NodoLista {
      friend class Lista;
      private:
        ...
    };
    NodoLista *primero;
  public:
    ...
};
```

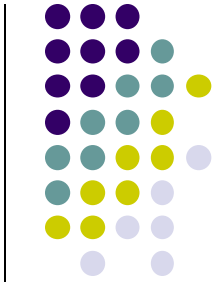
La clase contenedora r
acceso a la parte privad
clase interna, a menos
declare como friend class.

```
class Lista {
  private:
    ...
  public:
    class iterator {
      ...
    };
    ...
};

Lista::iterator it;
```

Si la clase interna es pública, la pueden utilizar los usuarios.

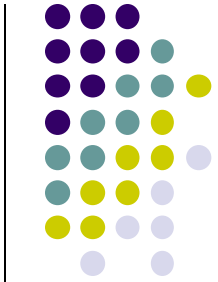
Redefinición de operadores



- Los **operadores** son las funciones básicas de un lenguaje, que normalmente se representan con símbolos, como: +, -, *, /, =, ==, <, >, >=, <=, >>, <<, &&, ||, etc.
- C++ permite **redefinir los operadores** de una clase, para que hagan lo que queramos.
- El operador tendrá el nombre: operator+, operator-, operator*, operator==, etc.

```
class Nombre {
    ...
    bool operator< (Nombre &otro);
    Nombre operator+ (Nombre &otro);
    ...
```

Redefinición de operadores



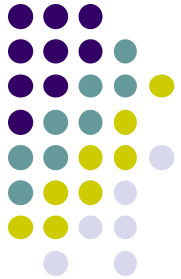
- También se puede redefinir operadores en otras clases. Ejemplo, redefinir el operador << para poder escribir con **cout << variable**.

```
class Persona {
   ...
   void escribe (ostream &co) {
      co << nombre << " " << edad;
   }
};
ostream & operator<<(ostream &co, Persona &p) {
   p.escribe(co);
   return co;
}
```

ostream es la clase de la variable cout.

Ahora podemos escribir con:

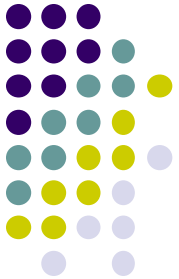
```
Persona p1;
cout << p1 << endl;
```



Semanas 4 y 5: ejercicio 200

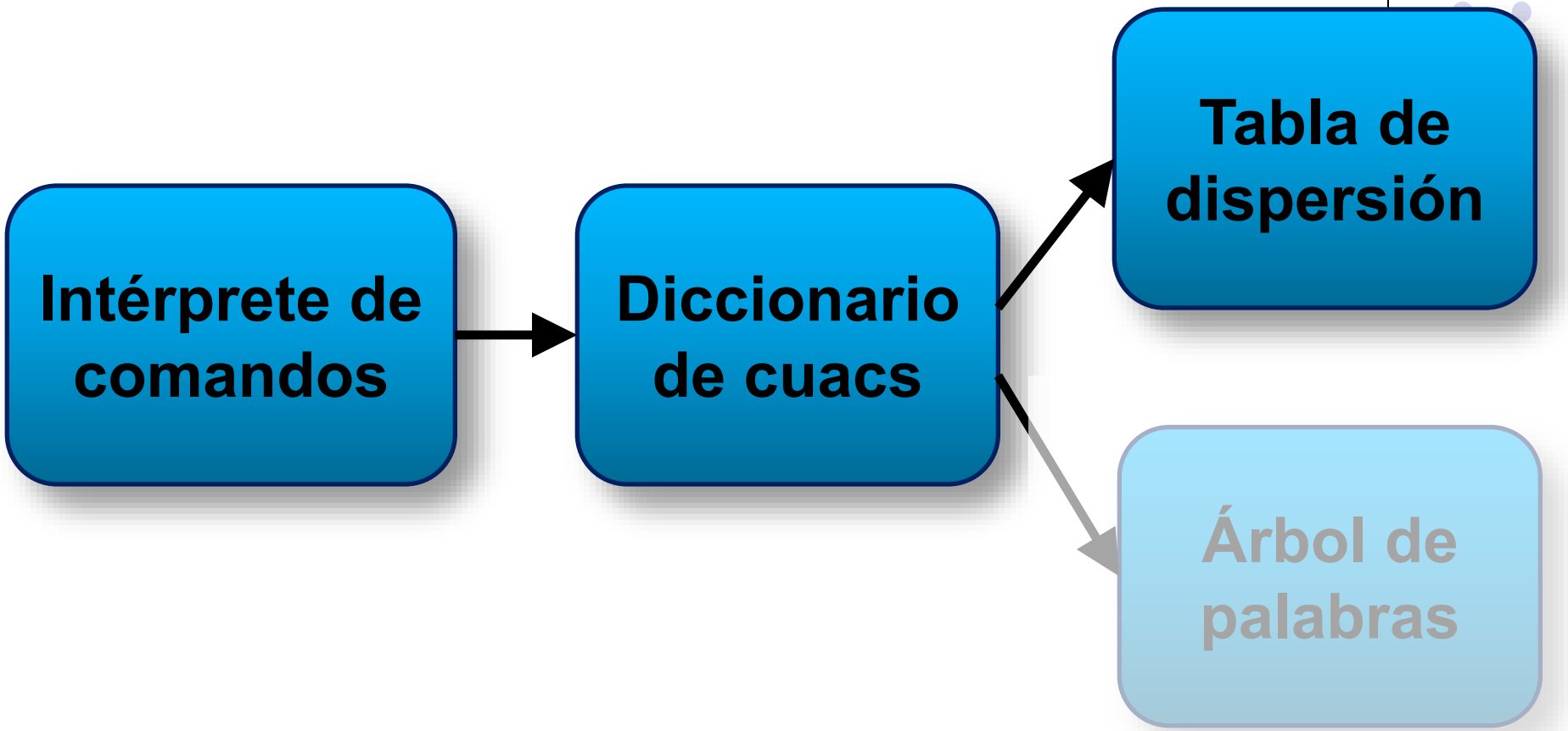
Planificación práctica

200 – Tablas de Dispersión de Cuacs



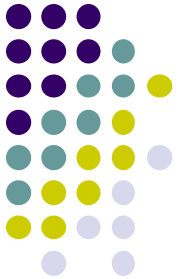
- Cambiar la implementación del diccionario con **list<Cuac>**, por una implementación más eficiente mediante **tablas de dispersión**, indexada por el nombre del usuario.
- La **eficiencia es fundamental**: una buena función de dispersión, tamaño de la tabla óptimo, tipo de dispersión, estrategia de reestructuración, etc.
- Respetando el principio de abstracción, no hay que modificar el intérprete de comandos.
- Solo cambia el **diccionario de cuacs** y se añade una **tabla de dispersión de cuacs**.

200 – Tablas de Dispersión de Cuacs



- El diccionario *centraliza* el almacenamiento de los cuacs. Usa las tablas de dispersión y (más adelante) los árboles.

200 – Tablas de Dispersión de Cuacs

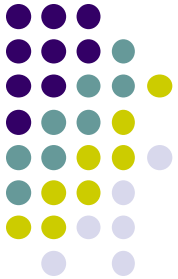


```
class DiccionarioCuacs {
   private:
     TablaHash tabla;
   public:
     DiccionarioCuacs ();
     void insertar (Cuac nuevo)
       { tabla.insertar(nuevo); }
     void follow (string nombre)
       { tabla.consultar(nombre); }
     void last (int N);
     void date (Fecha f1, Fecha f2);
     int numElem ()
        { return tabla.numElem(); }
};
```

¿Es necesario?

Hay que “vaciar” estas operaciones (no hacen nada).

200 – Tablas de Dispersión de Cuacs



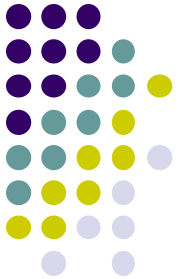
```
class TablaHash {
  private:
     ...
     int nElem;
  public:
     TablaHash ();
     ~TablaHash ();
     void insertar (Cuac nuevo);
     void consultar (string nombre);
     int numElem (void) { return nElem; }
};
```

Necesario si se usa memoria dinámica.

200 – Tablas de Dispersión de Cuacs

- Decisiones de diseño:
  - ¿Qué tipo de tablas utilizar? ¿Dispersión abierta o cerrada?
  - ¿Qué función de dispersión aplicar?
  - Si es dispersión cerrada, ¿qué función de redispersión usar?
  - ¿Tamaño fijo o reestructuración dinámica?
  - ¿Qué tamaño inicial de la tabla?
- Hay que analizar las opciones, razonar la elección, hacer distintas pruebas y justificar las decisiones tomadas en la **memoria final de la práctica**.

200 – Tablas de Dispersión de Cuacs



```
class TablaHash {
  private:
```

	Dispersión abierta	Dispersión cerrada
Tamaño fijo de la tabla (T)	`list<Cuac> T[TAM];`	~~`Cuac T[TAM];`~~
Tamaño variable	`list<Cuac> *T;`	`Cuac *T;`

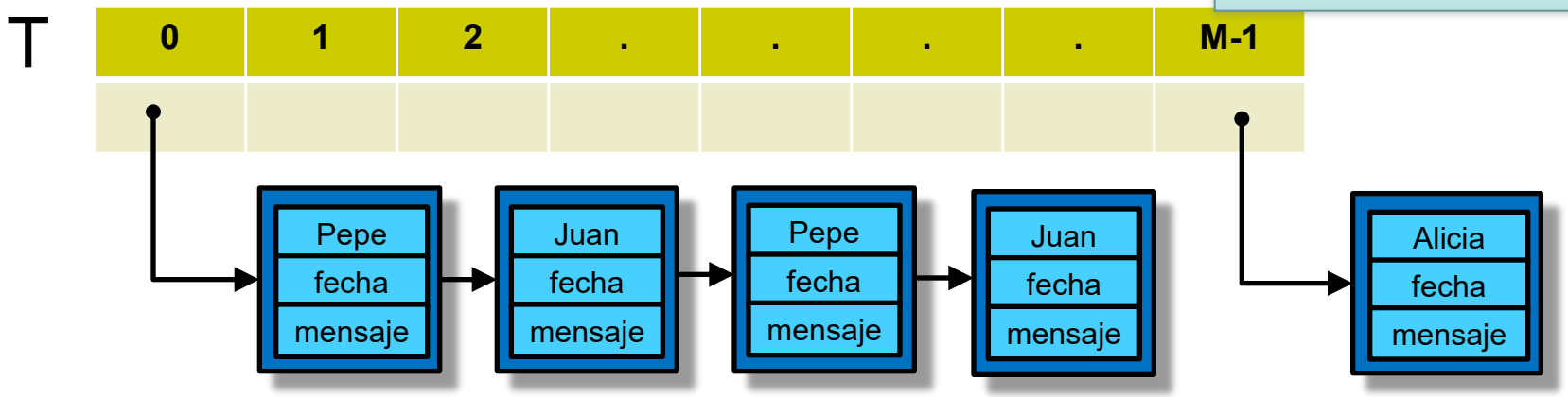
```
    int M;
    int nElem;
```

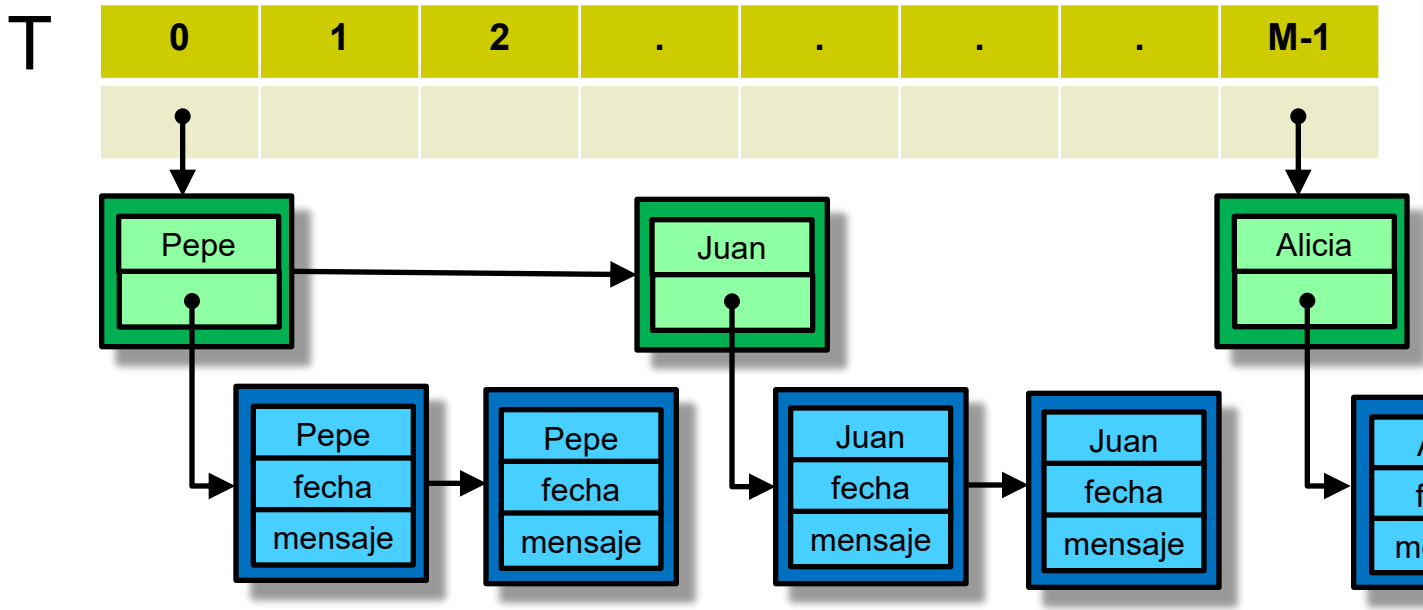
- **Tamaño fijo**: más sencillo, pero no se adapta al número de elementos que se van almacenando.
- **Tamaño variable**: se adapta mejor, pero es necesario:
  - En el constructor, reservar memoria con **new** (**new** list<Cuac>[M], **new** Cuac[M]).
  - En el destructor, liberar con: **delete**[] T;
  - Reestructurar la tabla si se llena mucho.

200 – Tablas de Dispersión de Cuacs

- ¿Cómo almacenamos los cuacs en la tabla? Dos opciones

List<Cuac> T[M];



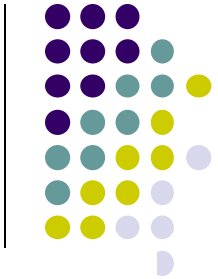



```
List<Par> T[M];
class Par {
   string nombre;
   list<Cuac> l;
};
```

200 – Tablas de Dispersión de Cuacs

- Recordar las propiedades de una buena función:
  - Que produzca un **reparto lo más uniforme posible** de los valores que produce (rango amplio de valores ≠).
  - Que se pueda **calcular rápidamente**.
- P.ej.: suma posicional, extracción, hash iterativos, etc.
- ¡**Prohibido** usar las siguientes!
  - Sumar los códigos ASCII → Produce pocos valores ≠.
  - Multiplicar los ASCII → No produce números primos.
  - Usar la primera letra como un índice (A=0, B=1, etc.) → Muy mal, genera poquísimos valores y mal repartidos.
- Cuidado con los **valores negativos**: los UTF (0xC3, 0xA1, 0x81, etc.) si se toman como char son valores negativos.
- El negativo también puede aparecer si se produce un desbordamiento (sumar o multiplicar valores grandes).

200 – Tablas de Dispersión de Cuacs



- **Probar distintas funciones de dispersión** y analizar el tiempo de cada una:

```
>> time ./a.out < 200a.in > salida
```

- Empezar con una versión sencilla (tamaño fijo de la tabla) y después implementar reestructuración.
- ¿Tamaño de la tabla? No sabemos a priori el número de elementos almacenados. Pero los casos de prueba nos dan una idea de los tamaños que tendrán, como mínimo.
- Recordar: si se usa memoria dinámica (*) hay que liberar la memoria en el destructor.