ALGORITMOS Y ESTRUCTURAS DE DATOS 1

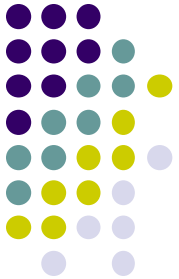
Práctica: CUACKER

Sesión 5

Recomendaciones y errores comunes

- No pretender “reservar memoria” para variables que son estáticas: ~~TablaHash t= TablaHash();~~
- En C++ las cadenas se comparan con ==, >, <, >=, etc. No usar cadena.compare(otra).
- Evitar todos los usos innecesarios de this->
- En el Makefile, recordar incluir las dependencias indirectas. main.o: main.cpp Dicc.h Tabla.h Cuac.h ...
- Descomposición modular: normalmente, cada clase va en un módulo propio.
- No acceder a *it cuando it = lista.end().
- Evitar el uso de variables globales.
- Diseñar una buena función de dispersión.

Introducción a C++



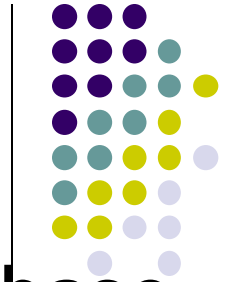
Videotutorial 9

- Genericidad en C++

Mecanismos de manejo de errores:

- Excepciones: throw y try…catch
- Asertos

Genericidad en C++

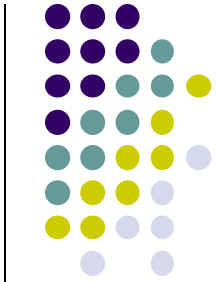


- **Genericidad (o parametrización de tipo)**: una función o una clase están definidas en base a unos parámetros de tipo, que pueden variar.
- Ejemplos:
  - Funciones: OrdenaArray<T>, EscribeArray<T>
  - Clases: Conjunto<T>, Lista<T>, Diccionario<C,V>
- La declaración de la función o clase genérica se hace mediante las **plantillas o *template***:

```
template <class T>
función o clase genérica
```

Aquí dentro se puede usar T como si fuera un tipo existente.

Genericidad en C++

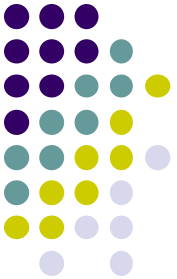


- Ejemplo: función genérica para escribir en pantalla un array de tipo T.

```
template <class T>
void escribir (T array[], int tam) {
    for (int i= 0; i<tam; i++)
        cout <<i<<". "<< array[i] << endl;
}

int ai[]= {65, 23, 12, 87};
string as[]= {"Hola", "Adios", "Bye"};
escribir(ai, 4);
escribir(as, 3);
```

Genericidad en C++

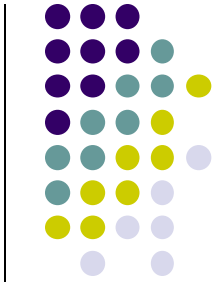


- Ejemplo: clase pila genérica de tipo T.

```
template <class T>
class Pila {
   private:
      T array[100];
      int tope;
   public:
      Pila ();
      void push (T valor);
      void pop ();
      T top ();
};
```

En esta pila solo caben 100 elementos.

Genericidad en C++



- En la implementación, hay que usar la cláusula ***template*** para todos los métodos.

```
template <class T>
Pila<T>::Pila () { tope= 0; }

template <class T>
void Pila<T>::push (T valor) {
   array[tope++]= valor;
}

template <class T>
void Pila<T>::pop () {
      if (tope) tope--;
}
```

Genericidad en C++

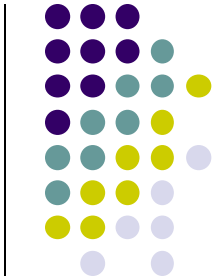
- El **uso** de la clase genérica se llama **instanciación**:

```
Pila<int> p1;
Pila<string> p2;
Pila<list<int> > p3;
p2.push("Hola");
...
```

Observar que se pone: > > para no confundir con el operador >>

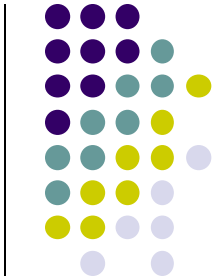
- **Nota sobre clases genéricas y modularidad**:
  - El compilador solo genera código para las clases genéricas cuando se instancian. Si no se instancian, no se genera código.
  - Por ejemplo, si tenemos un módulo pila.h y pila.cpp, la compilación de pila.cpp no genera código → Problema.
  - **Solución**: en la implementación de las clases genéricas debe ir todo en el fichero de cabecera.

Excepciones: throw y try…catch



- Similar a las excepciones en Java, pero no hace falta declararlas en las funciones (throws...).
- Las excepciones son un mecanismo para señalar y tratar errores en los programas.
  - El código que detecta el error **lanza** la excepción (throw).
  - La excepción se **propaga** hacia los procedimientos que lo han llamado…
  - Hasta que alguno de ellos la **captura** dentro de un bloque try…catch.
  - Si la excepción se propaga hasta el main y no se ha capturado, se detiene el programa.
- Lo que "se lanza" es un objeto, que puede ser de cualquier clase.

Excepciones: throw y try…catch



- Ejemplo: programa que lanza una excepción de división por cero, creando una clase específica.

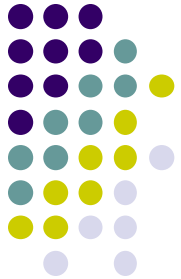
```
class DivisionPorCero {};

double divide (int a, int b) {
   if (b==0) throw DivisionPorCero();
   return double(a)/double(b);
}
```

- También se puede lanzar un string, un entero, etc.

```
double divide (int a, int b) {
   if (b==0) throw "Error de división por cero";
   return double(a)/double(b);
}
```

Excepciones: throw y try…catch



- Ejemplo: captura de una excepción.

```
double d;
int a, b;
try {
   cin >> a >> b;
   d= divide(a, b);
}
catch (DivisionPorCero e) {
   cerr << "No se puede dividir " << a << " entre "<< b;
}
```

Captura solo excepciones de ese tipo. El objeto lanzado se almacena en la variable e.

- Para capturar cualquier tipo de excepciones.

```
try { d= divide(a,b); }
catch (...) {
   cerr << "Error en división por cero";
}
```

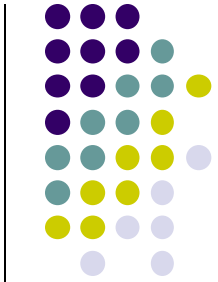
Asertos

- Los **asertos** son puntos de comprobación de una condición.
- En C/C++, los asertos no son parte del lenguaje, sino una función de la librería <assert.h>.

```
void assert (int expresión);
```

- **Significado**: comprobar si se cumple la condición booleana dada en la expresión. Si no es así, se interrumpe la ejecución del programa y se muestra un mensaje de error.
- Los asertos no generan excepciones, no se pueden capturar. Se usan para detectar posibles errores de programación.

Asertos



- Ejemplo: división comprobada con asertos.

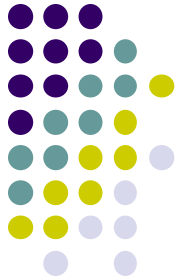
```
double divide (int a, int b) {
   assert(b!=0);
   return double(a)/double(b);
}
```

- En caso de no cumplirse la condición, se mostrará un mensaje del tipo:

```
a.out: pr.cpp:8: double divide(int, int): Assertion `b!=0' failed.
Aborted
```

- Los asertos solo deberían usarse cuando “no se sabe lo que hacer” en caso de error. Se pueden desactivar poniendo:

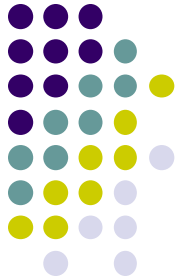
```
#define NDEBUG
```



Semanas 6, 7 y 8: ejercicios 300, 301 y 302

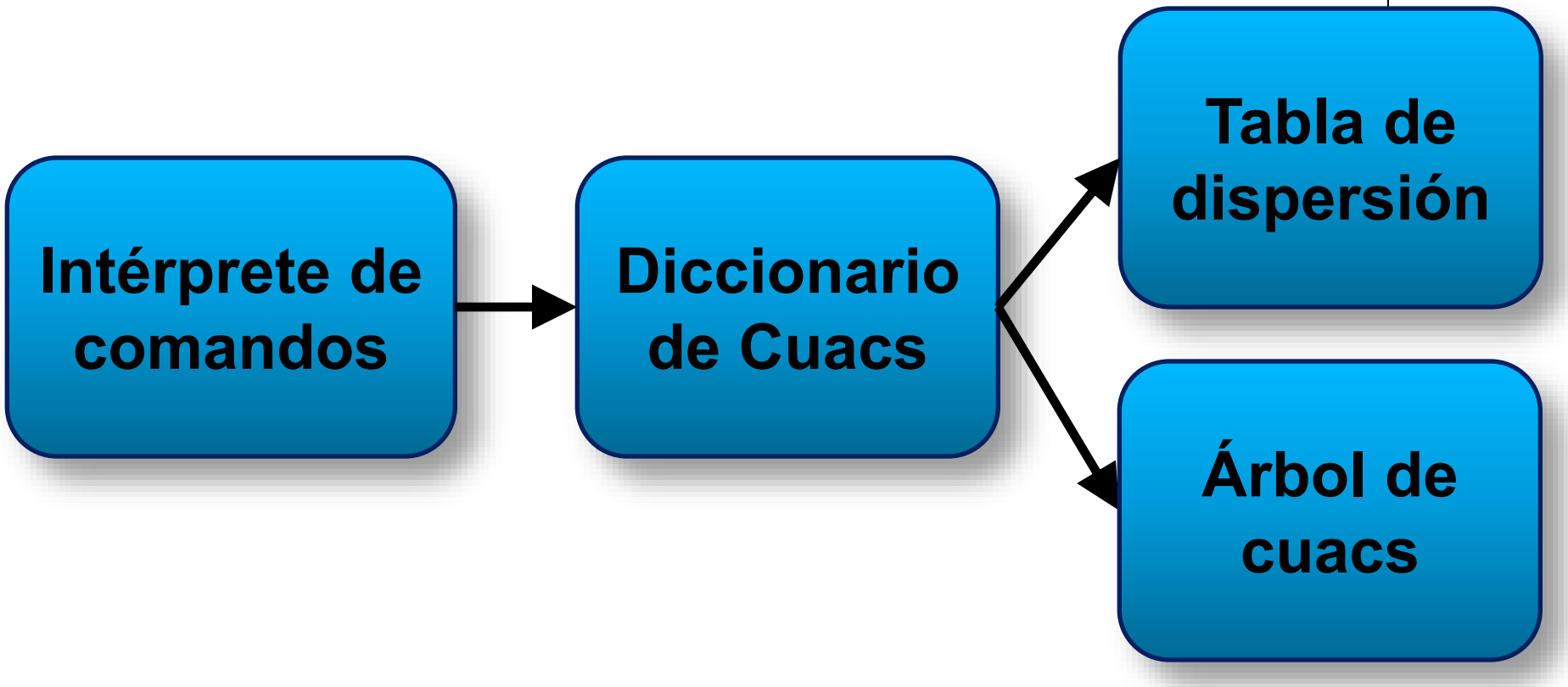
Planificación práctica

300 y 301 – Árboles de Cuacs para Last y Date



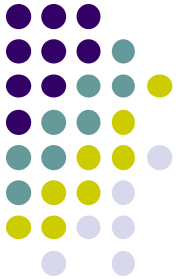
- Implementar una estructura de **árboles de cuacs**, añadiéndola al diccionario existente.
- Árbol ordenado por el orden de los cuacs.
- Resolver las consultas por fechas: last (300) y date (301).
- ¿Qué tipo de árboles? Árboles trie, AVL o B. La decisión queda a elección de los alumnos.
- Los cuacs no deben estar duplicados: el árbol contendrá punteros a la tabla de dispersión.

300 y 301 – Árboles de Cuacs para Last y Date



- Seguimos manteniendo el mismo diseño. El **diccionario de cuacs** controla la tabla de dispersión y el árbol.

300 y 301 – Árboles de Cuacs para Last y Date



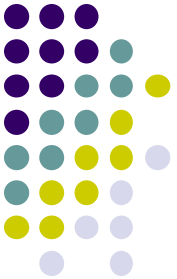
```
class DiccionarioCuacs {
   private:
     TablaHash tabla;
     Arbol arbol;
   public:
     void insertar (Cuac nuevo)
       { Cuac *ref= tabla.insertar(nuevo);
         arbol.insertar(ref); }
     void follow (string nombre) ...
     void last (int N)
       { arbol.last(N); }
     void date (Fecha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

Ojo, el árbol almacena punteros a los cuacs de la tabla.

¿Cómo obtener un puntero a un Cuac de una lista?

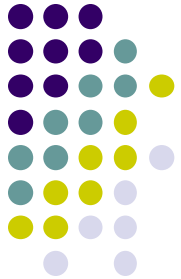
```
lista.insert(it, cuac);
it--;
&*it
```

300 y 301 – Árboles de Cuacs para Last y Date



```
class Arbol {
  private:
     ...
  public:
     Arbol ();
     ~Arbol ();
     void insertar (Cuac *ref);
     void last (int N);
     void date (Fecha f1, Fecha f2);
};
```

300 y 301 – Árboles de Cuacs para Last y Date



- Es adecuado definir un **tipo Nodo**, en el que se basa la definición del tipo Árbol.

```
class Nodo {
   private:
      Nodo *hijo;
      ...
   public:
      Nodo ();
      ~Nodo ();
     ...
};
```

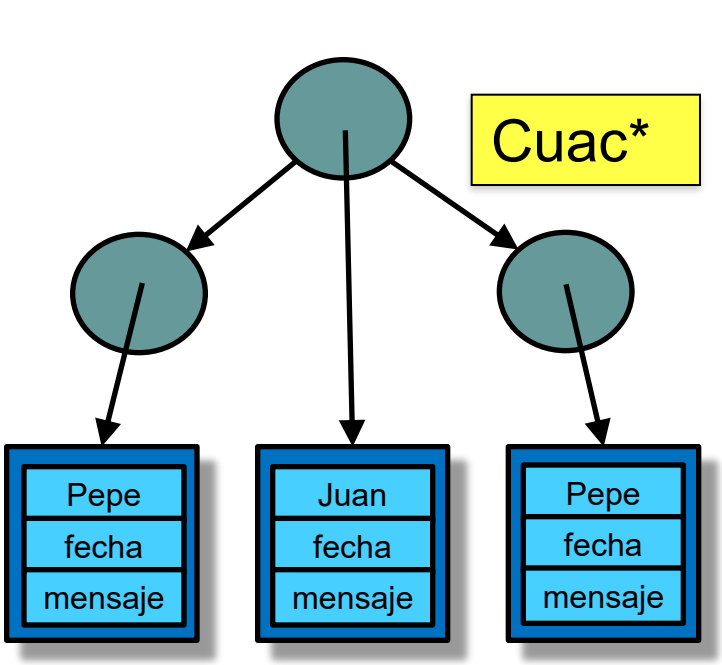
```
class Arbol {
  private:
     Nodo *raiz;
  public:
     Arbol ();
     ~Arbol ();
     ...
};
```

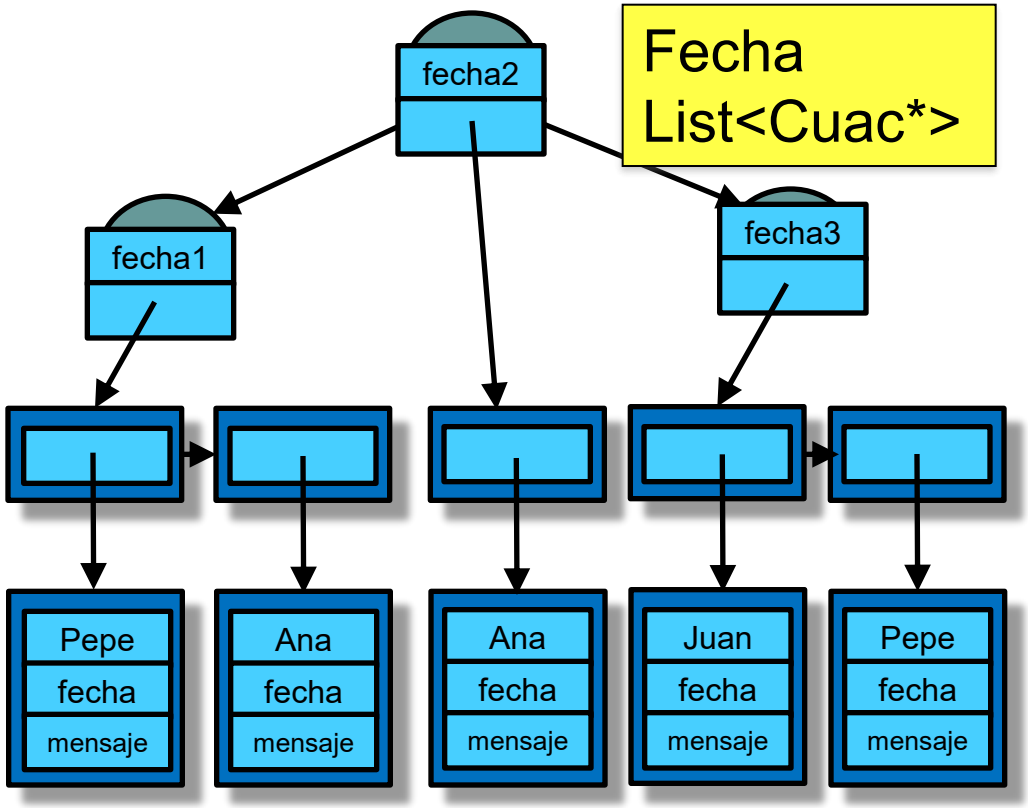
- Definir los constructores y destructores de ambos.
- Repartir bien la funcionalidad entre nodo y árbol.
- ¿La raíz se inicializa a NULL o a un **new** Nodo? Depende del tipo de árbol y del diseño.

300 y 301 – Árboles de Cuacs para Last y Date

- Dos opciones de diseño:

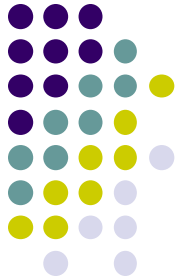
Árbol de punteros a cuacs | Árbol de listas de punteros a cuacs





- ¡Mucho cuidado con la reestructuración de la tabla de dispersión! Si tenemos list<Cuac> *T, al reestructurar, el árbol apunta a los cuacs antiguos.

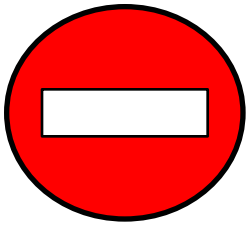
300 y 301 – Árboles de Cuacs para Last y Date



- Elección del tipo de árbol:
  - Analizar qué tipo de árbol es más adecuado para el problema.
  - Diseñar, implementar y verificarlo.

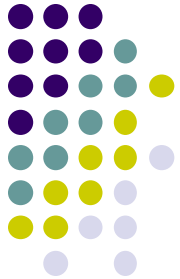
Más decisiones de diseño para incluir en la memoria final.

- **Prohibido**:



  - Usar árboles binarios de búsqueda sin balanceo.
  - No hacer las operaciones de destrucción del árbol y de los nodos.
  - Crear operaciones que ocupan varias páginas.
  - Repartir de forma incorrecta la funcionalidad de nodos y árboles.

300 y 301 – Árboles de Cuacs para Last y Date

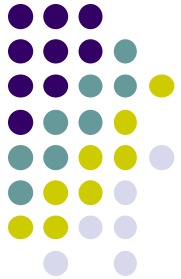


- Algunas indicaciones con **árboles AVL**.
  - Tienen un buen comportamiento para las operaciones de inserción, consulta y listado.
  - ¿Dónde incluir las rotaciones?
    - **En la clase Nodo**: las rotaciones cambian la raíz. Por lo tanto, al rotar un nodo, la nueva raíz es otro nodo. Pero el puntero **this** nunca se debe modificar...
    - **En la clase Árbol**: el nodo se pasa como parámetro. Además, debe ser un parámetro por referencia (&), puesto que se actualiza la nueva raíz.

```
class Nodo {
     ...
     Cuac *cuac;
     ...
  public:
     Nodo *RSI ();
     ...
};
```

```
class Arbol{
     ...
  private:
     void RSI (Nodo *&A);
     ...
};
```

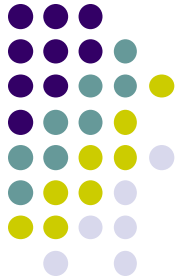
300 y 301 – Árboles de Cuacs para Last y Date



- Algunas indicaciones con **árboles TRIE**.
  - Nodos: representación con listas o con arrays.
  - La clase Nodo debe incluir los métodos consulta, inserta y las de las marcas.
  - La clase Árbol ¡no debe acceder a la represen-tación interna del nodo!, sino que debe usar los anteriores métodos de Nodo.

```
class NodoTrie {
  private:
    char car;
    NodoTrie *sig, *ptr;
    Fecha f;
    list<Cuac*> lista;
  public:
    NodoTrie ();
    ~NodoTrie ();
    NodoTrie *consulta (char letra);
    void inserta(char l);
    bool HayMarca ();
    void PonMarca ();
    void PonEnLista (Cuac *ref);
    list<Cuac*> getLista ();
    ...
};
```

300 y 301 – Árboles de Cuacs para Last y Date



- Algunas indicaciones con **árboles B**.
  - Árboles B de orden p.

```
class NodoB {
  friend class ArbolB;
  private:
    Fecha f[p-1];
    list<Cuac*> L[p-1];
    NodoB *pt[p];
    int numElem;
  public:
    NodoB ();
    ~NodoB ();
    ...
};
```

```
class ArbolB {
  private:
    NodoB raiz;
  public:
    ArbolB ();
    void insertar (Cuac *ref);
    list<Cuac*> buscar (Fecha fecha);
    ...
};
```

Como siempre habrá por lo menos un nodo, lo definimos estáticamente. Así nos ahorramos el destructor.

- Las operaciones sobre el árbol tienen un carácter netamente recursivo.

300 y 301 – Árboles de Cuacs para Last y Date

- Sobre los **destructores**.
  - Recordar: todo lo que se reserva dinámicamente (*) debe liberarse en los destructores.
  - El destructor de una clase debe liberar sus atributos dinámicos.

```
class Nodo {
   private:
      Nodo* hijo1;
      Nodo* hijo2;
      ...
   public:
      ~Nodo () {
         delete hijo1;
         delete hijo2;
      }
      ...
};
```

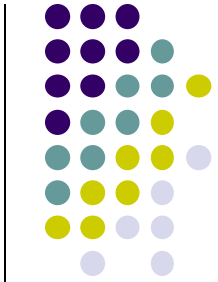
```
class Arbol {
  private:
     Nodo *raiz;
  public:
     ~Arbol () {
        delete raiz;
     }
     ...
};
```

Al hacer **delete** raiz; la liberación de la raíz llama al destructor de nodo.

De esta forma se propagan las liberaciones de todos los nodos del árbol.

Al llegar a un nodo NULL, el **delete** no hace nada y acaba la recursividad.

302 – El Motor de Cuacker



- Resolver todos los comandos obligatorios de la práctica:
  - pcuac, mcuac, follow, last, date, exit
- Si lo hemos hecho bien, el programa es el mismo que el del ejercicio 301.
- Probar los casos de prueba del 200, 300 y 301.